The (5+1) Architectural View Model for Cloud Applications

Mohammad Hamdaqa, Ladan Tahvildari
Software Technologies Applied Research (STAR) Group
Department of Electrical and Computer Engineering
University of Waterloo, Canada
{mhamdaqa, ltahvild}@uwaterloo.ca

Abstract
Existing software architecture frameworks focus on application development, rather than the dynamic evolution of applications at runtime. Their view models reflect design considerations, more than service operations. However, the quality of a cloud application depends on its configuration and the architecture of its service model. For this reason, we need a view model that is constructed around deployment. This paper proposes a (5+1) architectural view model, where each view corresponds to a different perspective on cloud application deployment. The (5+1) view model has been realized as a layered, domain specific modeling language, and the capabilities of this language have been illustrated using a representative domain example. The model was derived by investigating the process of architecting cloud applications, and then providing a set of meta-models to describe cloud applications within their ecosystem.

1 Introduction
An architectural view model is a set of logical and consistent views that describe a complex system. Each view depicts the system from a different perspective to meet the information needs of different stakeholders [25]. This paper presents a view model for cloud applications that enables cloud stakeholders (e.g., providers, developers, administrators and financial managers) to leverage cloud platform capabilities to maximize availability, maintain performance levels, minimize costs, and leverage portability and scalability.

View modeling is a well-established practice. Existing frameworks include the (4+1) view model [25], the Three Schema approach [23], the Zachman framework [34] and the Department of Defense framework [13]. These frameworks all clarify software design, because that is when architectural decisions were traditionally finalized. Static deployment infrastructures were the norm. In cloud computing however, architecture evolves during deployment. Runtime operation needs as much architectural modeling as design does [31]. Cloud applications must morph at runtime to meet performance, availability, and scalability targets under changing conditions. This shift in emphasis from architecting for implementation to architecting for operation is illustrated in Figure 1.

Models of implementation architecture capture the design requirements to create design models that reflect the source code of the application. Models of service operation architecture captures the operational requirements to create run-
time models that describe the configuration space of the application. The practices complement each other. However, the latter is more malleable and so better illuminates the dynamic evolution of cloud applications.

This paper introduces a framework for understanding service operation architecture: the (5+1) view model. We also develop a cloud domain specific modeling language (DSML) called Stratus Modeling Language (StratusML) [19]. At the core of the (5+1) is the service model, which describes reconfigurable, executable, compositional units (Tasks [17]). Service models are further specified using four operational model views to represent performance, adaptation, availability, and execution scenarios; plus views that illuminate portability and operational cost. StratusML fragments models into artifacts that are easy to modify. By weaving these fragments into model views, cloud stakeholders can better understand the runtime dynamics and evolution of complex cloud applications.

Section 2 illustrates some of the challenges facing companies who use cloud applications. Section 3 explains how cloud applications must be architected to be malleable and manageable. Section 4 shows how the (5+1) meta-models facilitate achieving this. Section 5 realizes the (5+1) model using StratusML, and shows how it serves system stakeholders. Related work is summarized in Section 6, and Section 7 concludes by pointing out directions for future research.

2 A Motivating Example

Consider CoupoNet, a fictitious startup that offers a coupon service in the cloud. CoupoNet has a multi-tier, multi-tenant application that allows users to open a free trial or paid account, design coupons, and publish them to location-targeted customers. The application stores buy/sell data, and performs sophisticated analytics to rank offers, generate reports, and analyze system usage to dynamically update CoupoNet’s pricing models. The service was launched in the cloud to reduce costs, ensure reliability, and minimize administration.

CoupoNet is a startup, and does not know how much traffic it will get. The coupon market has seasonal peaks, last-minute pile-ons, and can exhibit slashdot effects. Demand surges are expected, but no one can predict the time. CoupoNet does not know how to distribute its service geographically to ensure high availability with low overhead. CoupoNet’s architect designed the application to minimize coupling and maximize cohesion. During deployment the administrator may redistribute the modules to minimize costs, and perform due diligence to initialize them with optimal ratios. However, this may contradict the architect’s original design decisions. As demand fluctuates, the administrator updates the deployment model and reconfigures the different services.

CoupoNet faces common cloud challenges. They have to model multi-tenant and multi-tier applications, partitioning them into modules, and distribute them across locations. They may struggle with vendor lock-in, deployment architecture mismatches, uneven cloud expertise and clashing stakeholder priorities [18]. Our (5+1) view model in StratusML can help reduce all these problems.

3 Architecting for the Cloud

Cloud computing improves service availability, minimizes downtime, and scales applications on demand. This means cloud applications must morph during runtime without requiring redeployment or restart [18]. They must scale out by adding new instances to meet demand, and scale up to larger virtual hosting machines as needed. They routinely switch tasks on/off to alter their behaviour, and they may need to change the wiring between tasks that communicate through common storage (queue, blob, etc.). To support performance debugging, they must be able to switch between logging fine or coarse grained system dynamics.

Figure 2 shows that this level of flexibility can be achieved by separating the application service model from its configuration model and the provider’s specifications; and enabling workflow model composition based on the service model tasks at runtime. The service model specifies the tasks provided by the cloud service, their types, and their relationships. The configuration model further specifies the service model, by specifying the replication of the tasks, and their concurrency and distribution. The provider’s specifications specify the resource configuration of the underlying hosting environment. Lastly, the workflow model represents different usage scenarios for the service model tasks. We can change these models separately to meet desired operational requirements. In
order for the changes to be applied automatically at run time, an adaptation model should be defined to access all the elements and parameters of the models that need to be changed. The adaptation model uses key performance indicators to initiate change requests. These indicators can be gathered from the runtime model and represented in a performance model to enable performance analysis. We call this pattern the Malleable Application Architectural Style (MAAS).

MAAS is a common architectural pattern in cloud systems. It relies on separating executable components from their model structure, configuration, target resource specifications, execution scenarios and control. In Figure 2, the control model corresponds to the performance and adaptation models, which are defined to influence the actions of the Cloud Fabric Manager. The fabric manager is a platform specific autonomic manager [21] that uses a set of adaptation rules and performance indicators to enact change actions on the target model using APIs.

The (5+1) meta-models capture all essential information for architecting malleable and platform independent cloud applications. The (5+1) approach is based on the assumption that virtual machines can deal with code level mismatches by letting software components run on low level infrastructure. We argue that this assumption is reasonable given cloud dynamics. The constant updating and multiple service providers that characterizes cloud computing drives mismatches between infrastructure and applications that impact deployment architecture. The (5+1) view meta-models make this assumptions explicit.

4 The (5+1) Architectural View Model

View models must specify not only how different views are distinct, but also how they overlap. It is usually difficult to define clear-cut boundaries between what each stakeholder should view and manage. The aim of the (5+1) view model is to simplify cloud application management by providing views that correspond to different stakeholder perspectives. This section illustrates how this is done.

4.1 Overview

As shown in Figure 3, the (5+1) architectural view model consists of five model views that specify the core/service model view to address five different, but interleaved cloud concerns related to service deployment and evolution. Each model view conforms to its corresponding meta-model. Moreover, the top element in all the model view meta-models extends the (5+1) component (i.e., a meta-meta model element). The aforementioned modeling hierarchy makes it possible to integrate all the (5+1) meta-models and facilitates organizing the different modeling elements into categories (layers) based on the cloud concern they address.

The core meta-model is a fairly comprehensive pivot model that describes the application’s deployment architecture in terms of tasks and interactions. It clarifies the cloud service model and its requirements in terms most vendors would understand. Each of the other five meta-models further enrich the expressiveness of the core model. The performance meta-model enables annotating the core
model with performance parameters. The adaptation meta-model specifies adaptation rules and actions for each task or group of tasks in the core model. The availability meta-model defines the model configurations (i.e., configuration models in MAAS). Its components instantiate the core model and distribute its instances to different geographic locations. The service workflow meta-model facilitates creating different service scenarios, by composing and executing core tasks in series to achieve a certain goal. Finally, the provider meta-model creates a provider profile, which consists of the provider’s service templates and pricing profile.

The meta-models described in this paper has been created using the following process: we started from existing domain artifacts of three cloud providers (i.e., Amazon AWS, Windows Azure and GAE), from which we created variability models to highlight their commonalities and differences. We built one core meta-model [17], realized it using a DSL modeling framework (i.e., Microsoft-DSL). Then, we built several views around the core meta-model by extending it with the required components to address various concerns. The result was a huge meta-model, where all the model elements inherit one of six elements each represent one of the six different meta-models in Figure 3. Each element in the meta-model is then mapped to one visual component in the MS-DSL framework. This paper does not cover the DSL syntax (the visual components) and semantics. The paper focuses on describing the (5+1) architectural view model views and elements by explaining the different meta-model elements. We used the process of cleaning and elimination to divide the meta-model into smaller meta-models to facilitate explaining them. A meta-model normally consists of abstract syntax, well-formedness rules, and semantics. While class diagrams are normally able to capture the syntax, and the natural language can be used to specify the semantics; the well-formedness rules requires using a constraint language to further specify the domain constraints. Due to page limits the current paper does not cover the meta-models well-formedness rules. However, a list of model validation constraints have been made available online on the StratusML webpage.

4.2 The Service (Core) Meta-Model

The service (core) meta-model allows developers to describe cloud services using platform independent components, providing a high level description of resource requirements independently from any target platform, and specifying operational requirements to be enforced and validated. More particularly, the core meta-model enables service developers to: (i) describe the structure of a cloud service composed of one or more Tasks, the types of tasks, and their relationships, (ii) cluster tasks into groups for easy management, (iii) assign each task a predefined or custom service template type, and (iv) assign availability level to every task (e.g., low, medium, high), which constrains how it should

---

Figure 3: The (5+1) Architectural View Model

---

1http://www.stargroup.uwaterloo.ca/~mhamdaqa/stratusml
Figure 4: The Core View Meta-Model

be instantiated and replicated in the availability model. A service template specifies the computation power, memory and storage of the virtual machines available to host a task. Service template types (e.g., small, medium, large, extra-large) vary based on the provider. The templates are specified using the provider meta-model.

Figure 4 depicts the core meta-model, which is an extension of the cloud application meta-model we presented in [17]. The original meta-model has been created as a carrier of the service model and configuration parameters needed to port cloud applications between different platforms. It is an abstraction layer on top of the PaaS layer that was defined after manually inspecting three cloud platform application packaging requirements, namely Amazon Web Services (AWS), Windows Azure, and Google Application Engine (GAE). For example, an Elastic Beanstalk App in AWS, a Role in Azure, and a Module in GAE specifications all refer to a virtual appliance that is prepared with the required software stack to run certain family of applications (i.e., web, back-end). All these concepts have been abstracted with the concept of Task in our meta-model.

The core meta-model captures all the concepts that are needed to specify a cloud configuration space, and facilitates dealing with inconsistencies between the different providers’ file structures. The main extensions of the core meta-model in Figure 4 over the one presented in [17] are the concepts of Service, Activity, GroupableCoreComponent, and Group.

A service inherits the CoreComponent. It contains at least one GroupableCoreComponent. A service is one level higher than a task in granularity. It is equivalent to a cloud application in the original meta-model. The new meta-model facilitates creating several services within the same application. Each of them consists of several virtual appliances (a.k.a. Tasks), Groups, and/or StorageMediums. An activity is a sub-process of a task. Each task has at least one activity. Activities are essential for performance analysis. GroupableCoreComponents can be nested so that certain properties apply to all of them. A Group is a container that applies the composite pattern; it inherits the GroupableCore-
Component elements at once.

The current core meta-model distinguishes three group categories: ScalabilityGroup, StorageGroup, and AvailabilityGroup. A ScalabilityGroup can overlap or be nested in any other group, regulated by adaptation rules. Components in the same ScalabilityGroup can be scaled via different ScalingFactors. A StorageGroup can nest only StorageTasks; although it inherits Group, it may not nest any other GroupableCoreComponent. This is enforced by a validation constraint. Finally, an AvailabilityGroup nests components, which need to be hosted for the same location. It is a superclass for the three geolocation groups (i.e., Zone, Region, and DataCenter). Availability groups will be discussed in Section 4.5.

4.3 The Performance Meta-Model

Figure 5 depicts the performance meta-model. Its elements were inspired by the UML performance analysis modeling profile (PAM), which is part of MARTE profile [1]. As shown in Figure 5, there are five association relationships that connect three core components to corresponding performance aspects. These links represent integration points between the cloud resource model and its performance specifications. Each Activity is linked to ActivityPerformance, Workload, and to itself through the Call reflexive relationship. The ActivityPerformance specifies the activity server performance parameters. It specifies that activity’s resource demand. The Workload specifies the intensity of demand on that activity. The Call component specifies the number of calls that each activity makes to others, the types of these calls, the communication mechanism (InteractionType) used, as well as the size of the input and output messages associated with the calls. While an activity can be the source of several calls, each call is associated with exactly two activities; a source and a target activity. An InteractionType can be synchronous, asynchronous, or forward, with the same semantics as in LQN [15]. The Endpoint is connected to a Network performance component that specifies the connection parameters. Finally, each Task is connected to a HostingEnvironment. The HostingEnvironment parameters can be inferred based on the task type and the target PaaS provided as specified in the provider meta-model.

In a nutshell, each task can perform several activities. Each activity has a phase that specifies
the order of execution within the task, and a type that specifies whether it is normal activity or a join, fork, or loop activity. Phase zero activities are always connected directly to the endpoint, they are the first activities to receive a request in a task. An activity can be linked to other activities within the same task or external tasks. Each activity belongs to one or more workloads. A workload logically groups all activities that are affected by the same traffic class and hence should be executed together, while a task groups all activities that share and utilize the same underlying resources. There are two types of workloads: open workloads, where requests arrive at a given rate; and closed workloads, where requests are generated by a fixed number of users (population). Since there is many-to-many relationship between activities and workloads, another entity is needed to specify the activity execution path for each workload. The workflow entity is part of the workflow view. Each performance solution (PSolution) has a list of workflows. Each workflow has a designated workload and an ordered list of the Activities performed by that workflow. A performance solution (PSolution) also specifies the solution technique used in solving the performance model (i.e., analytical, or simulation).

4.4 The Adaptation Meta-Model

The adaptation meta-model uses rules to control the core. This enables many dynamic features of the system, such as elasticity and security, and helps administrators ensure that the system continuously satisfies operational requirements (e.g., minimum operational cost, high performance, and high availability). A rule can control the number of virtual appliance instances, or enable a security guard or policy. A rule uses a set of key performance indicators as operands. These are usually collected through instrumentation and trace summarization.

Figure 6 shows the adaptation meta-model. Each component or group of components is associated with a set of actions. The meta-model enables two types of actions, predefined and custom actions. While predefined actions are known ahead of time (e.g., scaler actions), custom actions allow assigning external processes. An action is triggered based on a constraint or reactive adaptation rule. A ConstraintRule is a predefined (static) constraint, such as the min/max number of instances allowed at a certain time. A ReactiveRule is dynamic, based on evaluating runtime environment parameters against a set of key performance indicators (e.g., CPU utilization, queue length, response time).

4.5 The Availability Meta-Model

Figure 7 shows the availability meta-model. An availability zone refers to the distinct physical location of the available hosting data-centers of a provider. An AvailabilityZoneGroup nests components that need to be hosted in the same availability zone. It contains several RegionGroups that vary based on the provider. A RegionGroup nests components that need to be hosted in the same region and contains many DataCenterGroups. Lastly, a
Table 1: Availability and Fault Recovery Levels

<table>
<thead>
<tr>
<th>Specified Requirements (Core)</th>
<th>Validation Constraint to be Checked</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>Fault Recovery</td>
</tr>
<tr>
<td>Very Low</td>
<td>N/A</td>
</tr>
<tr>
<td>Low</td>
<td>Fast</td>
</tr>
<tr>
<td>Medium</td>
<td>Average</td>
</tr>
<tr>
<td>High</td>
<td>Slow</td>
</tr>
<tr>
<td>Very High</td>
<td>Fast</td>
</tr>
</tbody>
</table>

A task has one instance
A task has two instances in the same region
A task has two instances distributed into two different regions
A task has two instances distributed into two availability-zones
A task has three instances, two in the same region, and one in a different availability-zone
A task has four instances, two in the same region, one in different region but same zone, and one in different availability zone

DataCenterGroup nests components that need to be hosted in the same datacenter. Selecting a specific datacenter for an application is still not supported by most providers.

For each task in the core model, the availability meta-model allows administrators to specify the number of instances (TaskInstance) to instantiate (replication) and their distribution to different geolocations (AvailabilityGroups). Recall that in the core model a service developer specifies a provider for each task, and the required availability and fault recovery levels. The availability model assigns one of that provider’s available geolocations for each task. This enables hybrid cloud deployments, where an application can span multiple providers. Once the administrator creates the availability model, and before the actual artifacts required for packaging and deploying the application on a target platform are generated, a set of constraint rules are validated to ensure that the availability model conforms with requirements. Table 1 shows a list of availability and fault recovery objectives, and their corresponding validity constraints.

4.6 The Workflow Meta-Model

To represent system behaviour and enable operational analysis, the (5+1) model adopts a simple workflow meta-model. It specifies activity control structures that need to be executed in sequence to perform a certain work. As shown in Figure 8, the two main components in the workflow meta-model are activities and activity incidents. A workflow model represents the actual occurrence of the activities that are defined in the core model along with the workload generator (from the performance model) that is responsible for the activity enactment. Each activity incident belongs to one activity. This allows us to compose different workflows (execution scenarios) for the activities according to their runtime execution.

An activity can be classified as a normal, composite or control activity. A control activity links source and target activities and describes their execution sequence and concurrency. Depending on the control activity type, it can link one source to one target (sequence), one source to many targets (fork), or many sources (join). Activities that succeed a fork control can all be executed concurrently (and fork), or one at a time based on evaluating a boolean (or fork). Similarly, in an (and join) all predecessor activities must be executed before the join, while in an (or join) the execution of any of the predecessor activities is enough to start execution of the join activity. The normal activity is an elementary activity that cannot be further decom-
posed, while a composite activity contains (n) normal or complex activities and their control joint activities.

Even though the workflow model is essential for performance analysis, it is better to model activity definitions, performance specifications and actual usage scenarios separately. This separation of concerns maximizes modeling flexibility, fosters reuse, and facilitates modeling of the application’s behaviour at runtime.

### 4.7 The Provider Meta-Model

![Diagram of the Provider Meta-Model]

Figure 9: The Provider View Meta-Model

Traditional software architectural approaches presume that an application will be developed for one organization, deployed on premises, and maintained locally. In the public cloud, we develop for a multi-organizational systems of systems. The Provider meta-model aims to model different providers’ templates, offers and costs. Figure 9 depicts the Provider meta-model. Each provider provides a list of availability zones, and service templates. Availability zones represent the physical locations of provider data-centers, while service templates capture different resource specification bundles (i.e., CPU speed, number of cores, memory size, disk space) that serve as templates for service tasks. The ability to specify resources using predefined templates is a turning point in automating performance analysis. This meta-model enables reusability at the resource model level, providing ready to use templates that represent actual cloud provider templates. Each template has a price that captures how much the resource configurations cost per bundle. These prices usually depend on a provider’s pricing models, contract period and terms (e.g., free, weekly, pay-as-you-go).

### 5 StratusML

We validated the feasibility of our approach using StratusML [19]. StratusML is a cloud application modeling language that realizes the (5+1) view model. It enables developers to design high quality, distributed, component-based applications tailored for cloud deployment. StratusML permits the layering of views, letting stakeholders toggle between partial and holistic views, which makes complex models easier to use. StratusML visually models adaptation rules and constraints. It automates the generation of corresponding artifacts for the target cloud fabric, using elegant template-driven transformations to generate complete platform specific artifacts. A full list of the capabilities of StratusML is beyond the scope of this paper. However, a recorded demo is publicly available [19]. Here, we summarize the StratusML framework, and examine it against the motivating example presented in Section 2 to demonstrate its usefulness in modeling cloud application requirements.

#### 5.1 The StratusML Framework Architecture

Figure 10 shows the StratusML framework architecture. It is a model-view-controller framework that transforms, reconciles and adapts models. Transformation, validation, and analysis are performed whenever models are updated to keep them consistent. Both the validator and editor/viewer use the Stratus meta-model, which realizes the (5+1) meta-models in the Microsoft DSL definition language [11]. Figure 10 shows each (5+1) meta-models in its own layer, viewable in the modeling IDE. StratusML supports two transformation types: a model transformation that refines Platform Independent Models into Provider Specific Models; and template-based transformation that generates Platform Specific Models for target platforms. Figure 10 shows the transformation engine using the validated StratusML model, and applying template transformation on it to generate a target model. The model encapsulates essential data about the entities it must generate, while the template dictates the syntax of the target model. The last component in the framework is the architecture-aware adaptation
manager, which deploys the generated models to the target platform, and monitors the application’s behaviour after deployment to reflect changes in configuration and design. The adaptation manager uses the auto-generated rules to manage the application at runtime. It completes the application management process (i.e., Design, Plan, Package, Deploy, Run, Tune, Re-plan/Re-design). A detailed description of model validation, transformation and the architecture-aware adaptation have been omitted due to space limits. StratusML was built as an extension of Microsoft Visual Studio 2012. It utilizes the Microsoft DSL for model definitions and uses the Text Template Transformation Toolkit (T4) to turn models into desired output files [11]. StratusML lets stakeholders build models, and create (or use ready made) transformation templates.

5.2 Meeting CoupoNet Team Requirements

We used StratusML to model the CoupoNet application described in Section 2, generating the artifacts required to: (i) deploy the application to multiple providers, (ii) manage the application, and (iii) analyze its performance. This involved three web tasks and two worker tasks. Each web task was a frontend web MVC-style application that was accessed by specific user groups (i.e., Coupon Providers, Coupon Buyers, Admins and Marketing Researchers). The first worker task was the application backend that handles all operations (logic tier). The second was the analytics/data-crunching engine, which processed buy/sell data. There was also a storage tier, which consisted of blobs for storing data collected from buy/sell dumps, and queues for asynchronous communication between worker and web tasks. We do not provide a step-by-step design of this project using StratusML in this paper (a StratusML usage scenario has been made available on the StratusML webpage); instead, we highlight how the different layers and features of StratusML can foster collaboration between stakeholders to facilitate creating and analyzing cloud projects. Table 2 lists the requirements of the CoupoNet stakeholders, extracted from the example, and the StratusML layer that can be used to address the stakeholders needs. The following discussion explains how the features provided in the different layer(s) address stakeholders requirements.

**Core Layer:** The core layer furnishes various groups (i.e., storage, availability, and scalability) that assist in modeling multi-tenancy (R1) using different service/data partitioning strategies. Log-
Table 2: Stakeholder Requirements Matched to Corresponding Views in StratusML.

<table>
<thead>
<tr>
<th>CoupoNet Stakeholder Requirements</th>
<th>Stakeholder</th>
<th>(5+1) View / StratusML Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1. Model Multi-tenant applications.</td>
<td>Service Developer</td>
<td>Core</td>
</tr>
<tr>
<td>R2. Model Multi-tier applications.</td>
<td>Service Developer</td>
<td>Core</td>
</tr>
<tr>
<td>R3. Communicate architectural decisions to administrators.</td>
<td>Service Developer</td>
<td>Core + Availability</td>
</tr>
<tr>
<td>R4. Estimate the number of instances required.</td>
<td>Administrator / QA</td>
<td>Performance + Workflow</td>
</tr>
<tr>
<td>R5. Uncertainty about the required resources.</td>
<td>Administrator</td>
<td>Adaptation</td>
</tr>
<tr>
<td>R6. Service distribution into multi geographic locations.</td>
<td>Administrator</td>
<td>Availability</td>
</tr>
<tr>
<td>R7. Evaluate different provider offerings.</td>
<td>Manager</td>
<td>Provider</td>
</tr>
<tr>
<td>R8. Migrate between different providers.</td>
<td>Administrator/Manager</td>
<td>Provider</td>
</tr>
<tr>
<td>R9. Minimize administration and configuration tasks.</td>
<td>Administrator/Manager</td>
<td>All</td>
</tr>
<tr>
<td>R10. Manage model co-evolution.</td>
<td>Administrator</td>
<td>All</td>
</tr>
</tbody>
</table>

Physical (i.e., storage, scalability) and physical (i.e., availability) partitioning can be used to achieve isolation between the tenants to ensure privacy, manage service access privileges, and scale the application components together. Each StorageMedium or Task in the core model can be configured according to one of the following multi-tenancy models: Single Instance Multi Tenant (SIMT), Multi Instance Single Tenant (MIST) and Multi Instance Multi Tenant (MIMT). The core meta-model also provides platform independent task-templates that can be used for modeling frontend, backend and storage processes. This facilitates modeling multi-tier applications (R2). Plus since groups can overlap, grouped components can migrate and scale together. Using groups, architects can ensure their decisions (e.g., reducing coupling, increasing cohesiveness) are preserved (R3), and not carelessly overridden.

Performance and Workflow Layers: Estimating system performance requires both performance and workflow layers. StratusML provides performance components to specify the information collected through monitoring. Figure 11 shows a screenshot of the performance information that can be specified for task activities. Once all the required performance information is specified transformation templates can be used to generate a target performance model. The example in StratusML web page shows how templates can be used to generate performance models from the specified performance model information for various performance analysis tools. Currently StratusML supports generating LQN models [15], which can be used to estimate the number of instances required (R4). A detailed example for generating performance models and using them to scale the application is out of the scope of this paper and will be addressed in future work.

Adaptation Layer: Using the adaptation layer, StratusML facilitates modeling adaptation rules and actions, with a focus on scalability actions. A user can specify constraints and reactive rules, and associate them with a task or scaling group. The framework generates the rule-based configurations required to automate resource provisioning. This solves the problem when required resources cannot be estimated at the beginning of the project (R5). Figure 12 shows a screenshot of a task that is associated with an adaptation action that is activated based on a default constraint rule.

Availability Layer: The availability layer provides groups that makes it easier to manage the locations and counts of task instances. This helps administrators model cloud service distribution to multi-geographic locations (R6). Figure 13 shows an example of three tasks located in the same data.
center. Each of them has only one instance as shown from the link cardinality. An administrator can easily instruct the cloud fabric to relocate the instance into a different location within the same provider or even a different provider by changing the instance properties.

Figure 13: StratusML Availability View Excerpt

Provider Layer: This layer captures provider specifications. Users can import a supported provider’s specifications, or create a custom one. Once imported, an admin can specify a designated provider for each availability group, and then distribute task instances to different groups. To evaluate different provider offerings (R7) the administrator or financial manager can select the availability groups where instances are distributed, change them all for a specific provider, or allow hybrid settings. The pricing algorithm will loop on all instances and use the pricing profile of each instance, reading off the instance size and the provider, to calculate the total cost of the configuration. By modeling service structure and configuration independent of platform, and refining models for target platforms, StratusML lets applications migrate between different providers (R8). StratusML makes this easy via transformation templates customized by platform, to auto-generate target platform artifacts. This reduces admin and configuration (R9), and facilitates model co-evolution (R10).

Figure 14 is a snapshot that shows how platform specifications and pricing information can be specified for different providers (i.e., Windows Azure).

6 Related Work

Our work has taken shape in the context of a rich literature focused on viewpoint architecture frameworks, model-driven quality prediction, and cloud DSMLs.

Viewpoint Architectural Frameworks: Using multiple views to explore systems from different angles is common practice in software engineering [13, 23, 25, 34]. There are many such frameworks used in software development. Krutch [25] famously proposed four views: logical, development, process and physical; plus “use cases” as the fifth view, to completely describe the system lifecycle. Zou and Pavlovski [36] extended the (4+1) view model by adding a control case view to address non-functional system requirements. Most such architectural frameworks are generic enough to describe large class of systems. However, they only consider the dynamism and variability of the design models and assume a static deployment infrastructure. Even when deployment models are considered, they are considered as part of the lifecycle and not a driver of architectural evolution.

The (5+1) model was inspired by the (4+1) model process view [25]. In fact, if tasks are considered processes, the (5+1) model can be seen as an extension of the process view, distinguished by
its focus on the cloud application ecosystem (represented by variable deployment models, dynamic infrastructure, and standardized service offerings). The (5+1) model augments its core with elements that enable quality prediction. Its views are limited but expressive enough to cover operational requirements and the application lifecycles in the cloud.

**Model-Driven Quality Predication (MDQP):** Several MDQP approaches appear in the literature. Many were surveyed in [12, 22, 24]. According to Koziolek’s [24] classification, the main three categories for MDQP approaches are prediction based on: (i) UML (e.g., CB-SPE [5], SPT [28], MARTE [1]) (ii) proprietary meta-models (e.g., CBML [33], KLAPER [10], ROBOCOP [7], PALADIO [4]), and (iii) middleware (e.g., NICTA [26]). Our approach relies on a proprietary metamodel. MDQP frameworks can be differentiated by (i) the non-functional properties they model, and (ii) their accurate depiction of both software and resource models. Most focus on performance analysis and software modeling. Some frameworks that support non-functional properties and enable hardware and software modeling include MARTE [1], and Descartes (DML) [8]. Like Descartes, we model dynamic systems on dynamic infrastructure, but we differ from both MARTE and Descartes by being cloud specific. Both generic and specific frameworks have pros and cons [9]. For example, while both we and Descartes model adaptation requirements and actions, Descartes’ generic method models multiple strategies. In comparison, we currently only model rule-based adaptation. However, since most cloud providers support rule-based adaptation and provide rule engines, our framework does generate the actual artifacts (rule files) needed to control applications at runtime. Descartes requires framework-specific strategies, which can complicate automation and model transformation.

**Runtime Models:** The problem described in this paper is also related to managing and configuring software systems at runtime [6]. Using runtime models to manage software systems is not new. Several approaches have been developed to address this problem [14, 29]. Recently, Zhang et al. leveraged runtime models for the management of diverse cloud resources [35]. Other than being cloud domain specific, our approach distinguishes itself in the way it weaves the different models together with the service model, and its ability to provide partial and holistic views to describe and analyze the cloud systems from different perspectives.

**Cloud DSML:** There have been several recent proposals for using MDE to develop cloud DSMLs [2, 27]. MODAClouds [3], and CloudML [16] are most related to our approach. Methods can be differentiated based on the features they provide. What distinguishes StratusML from other languages is its ability to provide partial and holistic views of the different cloud application concerns using layers, its ability to visually model adaptation rules and actions, and its ability to deal with platform heterogeneity using template-based transformations.

## 7 Conclusions

We presented a new (5+1) architectural view model for cloud applications. It was realized as a cloud DSML, StratusML, using layers to enable both partial and holistic views. The model promotes flexibility, portability, reusability and productivity. We showed how StratusML supports cloud stakeholder collaboration. The (5+1) model was designed for the cloud domain, but it applies to any large distributed system where software evolves in a dynamic environment.

The original contributions of this research are:

1. a new cloud specific architecture view model taking service models as first class entities, and building views that address cloud application evolution at runtime;
2. naming the “Malleable Application Architectural Style”, commonly used in cloud computing, to facilitate managing cloud configuration spaces and promote architectural self-awareness;
3. addressing the main service operational attributes needed to architect high quality cloud applications using proprietary meta-models that augment the cloud core service/resource model with quality-related concepts and attributes; and
4. realizing the (5+1) view model as a cloud DSML, using layers to enable toggling between partial and holistic views, facilitating dynamic behaviour modeling using adaptation rules and actions, and “weaving” stakeholder concerns together to generate useful artifacts.

Future research directions include extending the scenario presented here to (i) showcase other features of StratusML, (ii) show how the models presented here can be used for performance prediction, software adaptation, availability maximization and
cost minimization, and (iii) to present concrete examples for model transformation and platform specific artifact generation.

References


