Web Engineering for Cloud Computing
(Web Engineering Forecast: Cloudy with a Chance of Opportunities)

Giovanni Toffetti

Faculty of Informatics, University of Lugano

Abstract. Web Engineering has always been concerned with modelling the *functional* aspects of Web applications. *Non-functional* (e.g., performance, availability) properties of Web applications have traditionally been a minor concern in the Web engineering community and have been seen as technology- or system-related issues. The advent of Cloud computing, with substantial delegation of "system concerns" to infrastructure or platform providers, seems at a first sight to confirm the validity of this choice. But is this really true?

We will argue that, in order to be able to actually profit from the Cloud computing paradigm, Web Engineering methodologies need several interventions transcending the platform-specific concerns of adapting to Cloud technologies.

In this position paper, we call for a long-due revamp of Web engineering methodologies to become more **sound engineering practices** with respect to both functional and non-functional aspects of Web applications. To this end, we propose a methodological framework that preserves the advantages of model-driven development, but also takes into account performance and cost considerations for Cloud-based applications.

1 Introduction

In a recent report Gartner predicts that by 2015 "most enterprises will have part of their run-the-business software functionally executing in the cloud, using PaaS services or technologies directly or indirectly" and "cloud-based solutions will be growing at a faster rate than on-premises solutions" [13]. Indeed, the pace of adoption of cloud-technologies is staggering, driven by the realization by companies that (at least for some applications) the advantages of the pay-per-use utility model of software, platform, and infrastructure as a service (SaaS, PaaS, IaaS) outweighs today the CAPEX and OPEX of traditional in-house data centres.

Cloud computing pledges to free Web application providers from the burden of managing the Web infrastructure and applications running on it. The direct emanation of a pay-per-use model is the focus on application *elasticity*, meaning that providers can at any time change the amount of resources (e.g., virtual machines, CPUs, disks) assigned to applications maintaining constant QoS (i.e., their performance) and adapting their cost to the incoming workload.

Some Cloud providers already advertise their platforms claiming that porting a traditional Web application to the Cloud is as simple as uploading its ".war" file

M. Grossniklaus and M. Wimmer (Eds.): ICWE 2012 Workshops, LNCS 7703, pp. 5–19, 2012.
© Springer-Verlag Berlin Heidelberg 2012

to their systems. This is true, in the sense that the application will effectively run, but there is a small catch: generally **it will not scale**. Thus, this naive approach would only work for applications that do not need to support a considerable and varying number of users, hence probably have no real financial motivation behind Cloud adoption. For an enterprise application, where SLAs, serving multiple users, and data-intensive usage are the norm, moving to the Cloud means it will have to **support elasticity** in all its architectural constituents: from dynamic reconfiguration all the way to scalable technologies.

In this paper, we advocate that Cloud computing is a **great opportunity for the Web engineering community**. On one hand, Web engineering (WebEng) methodologies can become the enablers of new SaaS providers that design, test, deploy, and manage applications in a single online environment without requiring programming skills. On the other hand, and in order for these applications to scale, WebEng methodologies need to, at the very least, adapt their runtime and code generation mechanisms.

We go the full way and claim that also the modelling primitives and methodologies need a revamp. As main evidence to support this observation we will elaborate on the following two considerations: 1) some systemic trade-offs (e.g., consistency vs. availability vs. partition tolerance [4]) imply a *range of decisions that vertically span multiple modelling layers* through the complete application life-cycle; and 2) some of the main reasons to move to the Cloud are elasticity, the pay-per-use model, and the savings they imply: *non-functional aspects of Cloud applications reflect directly on their* ***cost and rentability***, hence WebEng simply cannot afford not to consider them.

The main contributions of this paper are:

- An analysis of how to *enable Web engineering methodologies to address Cloud computing development* in terms of code generation, modelling primitives, Cloud patterns and elasticity;
- the prospect of *Web engineering as a service*, a viable opportunity to foster elastic application modelling;
- a *methodological framework* to address both the functional and *non-functional quality* of Web applications in the Cloud, preserving the advantages and flexibility of Web engineering methodologies.

The rest of this article is organized as follows: Section 2 gives a high level overview of cloud computing paradigms and common patterns; Section 3 justifies the need for adaptation, and describes the different integration scenarios between Cloud computing paradigms and Web engineering. Section 4 discusses whether the aspects currently covered by Web Engineering methodologies are still adequate for Cloud development, Section 5 proposes an extended methodological framework catering for non-functional aspect for continuous development in WebEng; Section 6 comments on the related work. Finally, Section 7 draws conclusions and illustrates our planned future work.

2 Cloud Computing in a Nutshell

The Cloud computing paradigm is typically characterized as either software, platform, or infrastructure *"as a service"*, respectively *SaaS*, *PaaS*, and *IaaS*, where being *a service* implies billing. Each characterization sees two actors: the cloud provider and the cloud user together with a different amount of responsibility and control.

In ***SaaS***, the service being delivered is an application, a software accessed with a remote client, typically provided with some guarantees in terms of quality (QoS) expressed in terms of service level objectives (SLOs). The Cloud user normally pays a flat rate amount (monthly, yearly) to access the service and delegates the complete responsibility of managing the application, platform, and infrastructure upon which it runs to the service provider. For the client this is the simplest approach to cloud computing: it does not require any knowledge or experience about platforms and infrastructure technologies; the service offered is the application functionality and all non-functional aspects are delegated to the provider's responsibility. An example of a SaaS instance is the service offered by Salesforce.com, a company offering pre-packaged (and customizable) customer relationship management (CRM) solutions "as a service" to other companies.

In ***PaaS***, the service being offered is a "platform" in the sense of an application runtime environment together with generic application functionalities (e.g., database storage, event buses, messaging, authentication). The cloud client pays for the service of utilizing a runtime (often also development) environment and delegates to the platform provider the activities of managing the elasticity and the infrastructure running the applications. Each platform in the PaaS sense is a coherent set of technologies selected and managed by the cloud provider. They are typically packaged in application programming interfaces (APIs) that are specific of the chosen PaaS provider. Currently this is one of the main obstacles to Cloud interoperability, since PaaS APIs are very heterogeneous due to the different technologies adopted by providers. Even though the PaaS market is supposed to be consolidating in a few years into a small numbers of "big" providers [13], this might of course still result in vendor lock-in situations. PaaS is intended for application developers, hence more advanced users with respect to SaaS, that are willing to pay cloud providers to manage the infrastructure and platform supporting their application logic. Two quite different examples of PaaS solutions are Google App Engine and CloudBees. The former allows developers to build applications using the same scalable solutions powering some Google products (e.g., BigTable, GFS) on the Google infrastructure; the latter is targeted at development and deploy of Java Web applications with additional generic services (e.g., monitoring, logging, storage) on top of third party infrastructure providers, for example Amazon EC2.

IaaS builds mainly on virtualization: Cloud providers offer infrastructural resources (e.g., CPUs, RAM, bandwidth) as a service for running third party applications typically packaged in virtual machines (VMs). The service level agreements between infrastructure providers and cloud users only deal with provisioning and availability of resources for the VMs. In this scenario, application developers have

the most control on their application: apart from developing its functional logic, they can select the platform to use for development and deploy, and they have control on the number of virtual machines (and therefore on the amount and cost of resources) assigned to their application. Infrastructure providers retain the control on the actual physical data center infrastructure. Examples of IaaS are for instance Amazon EC2, Microsoft Azure, Rackspace, and GoGrid.

2.1 Cloud Patterns

Modern massively-scalable Web applications rely on different patterns and technological solutions with respect to traditional Web applications. While the three-tier architectural model still holds, each of the tiers had to undergo some restructuring in order to enable elasticity. Given the space limitations, here we give only a short list of the main issues addressed in elastic Cloud applications, they are:

Load balancing and HTTP sessions: Load balancers are used to spread the load across the multiple instances of VMs that compose an application. HTTP Sessions are typically stored at the application server layer, and load balancers have to forward requests for each user to the application server hosting the correct session (*"sticky sessions"*). Some PaaS providers (e.g., Cloudbees) do not support sticky sessions for some usage configurations, and suggest using session-specific datasources to persist all session-related variables so that they are shared by all VMs.

Dynamic reconfiguration: A common pattern for enterprise solutions is to realize Web applications as compositions of Web services offered by different components. When horizontally scaling any of this components by adding or removing a VM, all the components that directly communicate with it need to be notified and/or reconfigured. A typical solution is using enterprise service buses (ESB) to achieve de-coupling.

Storage, NoSQL, and sharding: This is probably the most notable paradigmatic change for Cloud-based Web applications. In his first conjecture of the CAP theorem, Brewer postulated [4] that it is impossible for a distributed system to provide at the same time consistency, availability, and partition tolerance. Much in this direction, Cloud application providers have soon realized that relational databases are able to scale only up to a point, then consistency (the 'C' in 'ACID') eventually has to be given up for improved availability and better partition-tolerance (i.e., ultimately for horizontal scalability). In other terms, the "blocking" needed for consistency might be negligible for small applications, but might become a hinderance when apps need to grow in size. One of the outcomes of these considerations is the thriving world of *NoSQL* approaches for "not-so-structured" storage of key-value pairs, documents, blobs. NoSQL is the term vaguely identifying the wide range of data management systems relaxing some of the ACID properties for the sake of horizontal scalability (i.e., adding instances). Several NoSQL solutions are widely adopted, for instance BigTable, Cassandra, Memcached, MongoDB, Apache CouchDB, and Voldemort. Apart from scalability, these solutions are typically adopted

also because they offer very low latency and flexibility to schema changes or no predefined schemas at all. NoSQL solutions are no silver bullet: some application scenarios have hard consistency requirements and the best option in this case is still using RDBMS, but in some other cases (e.g. Web2 social applications) dirty reads might be tolerated and, if needed, recovery policies might be implemented at the application level. NoSQL solutions widen the spectrum of design choices for storage management. A recent comparison of scalable data stores is provided by Cattell in [6].

Multi-tenancy: a multi-tenant architecture is one that serves a single software to multiple "tenants". Tenants can be different organizations, company divisions, or generic clients. In a multi-tenant architecture data and processing for each client may have different ranges of sharing/isolation requirements (e.g., for data: dedicated physical servers, shared virtualized hosts, dedicated database on shared servers, dedicated schema within shared database, shared tables) as well as customization (e.g., schema customization, UI customization). Higher isolation translates into higher costs for the software (SaaS) provider. Multi-tenancy goes hand in hand with Cloud computing, since the SaaS provider can for instance leverage application elasticity to serve multiple global clients working across time zones with one customizable application and relatively few VM instances. A representative example is Salesforce.com that has "72,500 customers who are supported by 8 to 12 multi-tenant instances (meaning IaaS/PaaS instances) in a 1:5000 ratio. In other words, each multi-tenant instance supports 5,000 tenants who share the same database schema"[1]. Multi-tenancy is a key resource for software providers, since it allows optimizing and reducing costs for serving software to multiple clients.

MapReduce and large scale data processing: MapReduce [14] is a programming model for the *distributed* processing of large datasets. While its initial (and still main) purpose is to be used for the massive parallelization on commodity hardware of long-running tasks, in common practice it is also used in NoSQL databases for data processing and even simple aggregation queries over distributed datasets (e.g., in MongoDB[2]). The most widely known implementation of MapReduce is Apache Hadoop[3].

3 Web Engineering Concerns

Web engineering methodologies can potentially become the enablers of new Saas providers that use a single interface to *seamlessly model, deploy, run, manage, evolve, and sell* new Cloud-based Web applications. These SaaS providers will not need to own any computing infrastructure, but their only required assets will be modelling tools and the models of applications to be sold. This is very much in line with Gartner's expectations for the medium term: "By 2015, 50% of all

[1] http://www.computerworld.com/s/article/9175079/Multi_tenancy_in_the_cloud_Why_it_matters_?

[2] http://www.mongodb.org/display/DOCS/MapReduce

[3] http://hadoop.apache.org

new application independent software vendors (ISVs) will be pure software-as-a-service (SaaS) providers" [13].

In order to do this, it is up to the WebEng community to seize the opportunity and update the current methodologies and tools. In the following paragraphs we will discuss why and how, but first, with the sole purpose of identifying the main areas of intervention, we need to briefly sketch what a WebEng approach looks like. Simplifying, we can roughly say that a generic Web engineering approach consists of:

- a *methodology* covering the complete Web application life-cycle;
- *platform-independent models* (data, "navigational", etc.);
- tools for model visualization and *editing*;
- tools for model transformation / *code generation*;
- libraries for the *runtime* support of the generated Web applications.

In WebEng, the separation between *platform independent* and *platform dependent* models and realizations is the abstraction that frees Web engineers from considering the technological details of the actual Web deployment letting them concentrate on the (core) functionality of their applications. The realization of platform-independent models (e.g., data and navigational models) into platform-dependent models (the application code / library configurations) is accomplished by the code-generation tools that embed the mapping / realization choices hiding them (and typically preventing changes) from developers.

We claim that for WebEng to support Cloud-based Web applications interventions are needed to:

1. adapt the runtime support and code generation techniques to Cloud APIs and technologies (platforms). This is a technological aspect required to address *elastic* Web applications;
2. include paradigmatic changes and patterns in the platform-independent models, and consequently update model editors. The aim is that of generalizing common patterns and make them available as clear design options to the application designer/analyst;
3. make modelling tools available for online usage, integrate with PaaS/IaaS providers realizing the view of *"Web modelling as a Service"*.

Admittedly, the third is not really a hard requirement, but rather a new and promising opportunity for WebEng approaches.

3.1 Code Generation

Current WebEng tools can generate Web application code for different platforms (e.g., Java, .NET) but are typically not concerned with systemic aspect of the production deployment, such as load balancing, scaling out, replication, and performances in general. In Section 2, we have seen how considerable effort has been invested in addressing application elasticity in Cloud development together with a brief list of the specific technologies that make this possible. These technologies can be accessed in different ways depending on the chosen Cloud model (PaaS or IaaS).

PaaS Deployment. In this scenario, the set of available elastic technologies is limited and depends on PaaS provider choices and expertise. This allows PaaS providers to control and manage the scaling out of applications. The biggest problem with PaaS is that in most cases access to scalable services and components needs to go through the PaaS provider API, *locking-in* applications to a single provider's infrastructure.

Considering that addressing all current PaaS providers' APIs is not a viable option, the first step in adapting a WebEng methodology for PaaS deployment would therefore be the choice of the (set of) PaaS provider(s) to support.

Then, for each PaaS provider API, the code generation and runtime support of every single modelling primitive (e.g., units in WebML [7]) would have to be rewritten and tested to invoke the correct provider methods. For instance, in Google App Engine the *default* data storage mechanism is the *"App Engine Datastore"*, an implementation of a NoSQL storage for objects that needs to be reached through a specific API in Java[4]. Only recently, Google extended the App Engine storage options with a relational offer (*"Google Cloud SQL"*) based on MySQL; for the moment being it is still in preview and with some limitations. It is a perfectly plausible scenario one in which, for reasons of opportunity, some application domain entities will be mapped to the relational data storage and some others to the NoSQL one. This is an example of design decision that will have to be made at model level, but needs to be supported by the code generation.

IaaS Deployment. In this scenario the atomic units of deployment are virtual machines (VMs). While each infrastructure provider supports its own VM specification (e.g., AMIs for Amazon EC2) and interoperability is not given as-is, lock-in issues are less severe in IaaS since application providers can freely choose their development platforms and OSs and can (re-)package them as disk images supported by each hypervisor technology.

The main problem instead is identifying and managing a set of components (load balancers, application servers, data storage implementations) so that they can be packaged in a coherent set of VMs, configured through code generation, and appropriately scaled out at runtime. Given the vast number of technologies currently available, it is by itself a challenging endeavour.

A sensible approach for adapting a Web engineering tool for IaaS might be building on pre-packaged (and modular) VMs adopting Cloud-specific patterns such as the ones provided by AppScale[5] [8]. A viable solution would then be updating code generation and runtime libraries to support for example at least one implementation per type of scalable storage (SQL vs NoSQL).

3.2 Model Extension

In our opinion, some of the aspects and patterns we introduced in Section 2.1 deserve to be promoted into platform-independent model concerns. We leave

[4] `http://code.google.com/appengine/docs/java/datastore/queries.html`
[5] `http://appscale.cs.ucsb.edu/`

the full proposal of how to include them and the evaluation of the possible alternatives to future work. In this paper, we want to give extensive justifications as to why and in which direction WebEng methodologies need to be updated.

For instance, the fact that some PaaS providers do not support sticky sessions can either be kept hidden in the code generation logic or explicitly taken into account at model level. In the first case, when generating code for a specific PaaS, all HTTP session-related logic has to be translated into storage queries and updates. Another option is to make designers aware of this choice by using warning mechanisms to prevent them from using HTTP session. The second choice impacts heavily on navigational modelling, and in case of multiple platforms might require model changes. The positive aspect is that the designer will have a clear perception of what the application actually does and will be aware of the eventual monetary costs and implications of the modelling choices, which might eventually lead to alternative designs.

As we stressed in Section 2, NoSQL storage is probably the Cloud-specific pattern with the most implications. Nowadays it is common for Cloud applications to use a combination of RDBMS and NoSQL data storage respectively to manage entities with hard consistency requirements (e.g., orders, item stocks, tickets, seats) and entities with eventual consistency (e.g., message chats, comments, events, logs). Designers will need to decide which kind of consistency they need for entities in their domain models. We argue this is not just a data mapping concern, since the type of entity (hard- or eventual- consistency) might need different navigational patterns.

We can give some practical examples. For instance, NoSQL objects are typically designed as self-contained elements to be accessed "in one scoop" with a query over a single collection (there are *no join operations* in NoSQL). For example, a blog post and all its comments can be stored and retrieved as a single navigable "object" (very much like beans in EJB) requiring a specific navigational pattern. Also giving up consistency for scalability (using NoSQL) one can find ways to mitigate transactionality: single object in NoSQL can support atomicity, and some failure and roll-back policies could be explicitly modelled at application level. For example, an online shop could use NoSQL data storage to manage items in stock and send out order cancellation notifications at a later time in case of stock depletion.

Multi-tenancy is another very interesting pattern for WebEng. Depending on the level of isolation, a SaaS provider could for example: 1) deploy multiple instances of the same application model with different UIs for each customer; or use the same application model and 2) explicitly trigger user-specific customizations at model level (e.g., add functionalities) as a sort of personalization, or 3) extend the modelling approach to add client customization (e.g., change UI with customer) as a separate/implicit concern to be dealt with at runtime. Many other possibilities are available and deserve some investigation, the message here is that WebEng can prove itself a powerful technique to achieve the rationalization of the SaaS provider multi-tenancy offering.

3.3 Web Engineering as a Service

The natural consequence of "everything as a service" is *Web engineering as a service*, where what is offered is a Web engineering approach as a modelling, development, and management platform. Let's not consider the practical implications of realizing such an environment for the moment; the advantages "on paper" would be:

- no need to install any software, accessibility from everywhere and for everyone (possibly with a browser?);
- easier and more controlled user licensing model for the WebEng approach provider;
- a single interface for modelling, generation, deployment, management of Web application;
- integration with PaaS / IaaS providers, access to online monitoring and management of the deployment;
- one-click deploy of staging / production application models;
- online sharing of models, components, patterns;
- online consulting to solve, help with, signal issues with applications and models;
- immediate signalling and propagation of failure and bug warnings across all deployed applications (in a software product line way).

Clearly, considerable work to extend the current modelling tools would be needed in order to deliver the full functionality. Some of the above advantages are already within reach, some others require investments and a costs and benefits analysis. We believe that in the end it will prove being worth the effort, especially if the WebEng approach can yield measurable benefits when applied to the Cloud paradigm, as we will argue in the following paragraphs.

4 Putting Non-functional Aspects Back on the Map

The Cloud pay-per-use model yields the most financial benefits to companies providing elastic applications (or having flexible requirements). Following this consideration, it is clearly of tantamount importance for elastic applications to be able to ***scale efficiently***, that is avoiding to waste resources (hence money) while offering elastic performance. In WebEng concerns, we argue that while it is a good idea to keep platform-specific aspects out of the way, now is a very good time to partially reconsider, since elastic scenarios make non-functional considerations critical. In this section and the next one we suggest how put these aspects back on the map in a non-intrusive (sort-of aspect-oriented) fashion.

As we have already said, platform-independent models in WebEng abstract from technological details to let designers focus on functionality. The downside to this is that, once the technological / system aspects are out of the way, it is very hard to put them back in the loop for an engineer to consider. As a consequence, Web engineering methodologies do not account for *non-functional* aspects (in particular performances) of the modelled applications.

In our opinion the fact that Cloud applications are elastic should be an **incentive**, rather than an excuse not to be concerned with the non-functional properties of Web applications. We try to contend in this direction with two examples.

Let's first consider a PaaS scenario, Google's App Engine for instance. Taking a look at the billing documentation[6] one can see that, apart from virtual machine usage, developers are charged also for datastore operations, and that, in order to limit running costs, they can set a maximum budget for running their apps. While a thorough look at how Google API high-level datastore operations map to low-level (billed) operations might save some money, a developer should make modelling decisions (and receive confirmatory feedback that they work by deploying) preventing the application from consuming its resource quotas and simply stop working[7]. In fact, application instance scaling is controlled by the number of requests in the queue of each machine instance[8]: complex pages mean higher latency, which means lower throughput, queues filling up, and *higher cost for running the application*. The trade-off is therefore between complex feature-rich pages and low latency and lower cost. In the end, given the relatively small control PaaS offer to developers, *fine tuning the application logic* and *knowing in detail the* ***performance profile*** of each of its page / operation / component are the only instruments available to achieve performance- and cost-effective applications.

For enterprise applications the predicted evolution is that they will partially move to the Cloud, keeping mission-critical resources on premises and following a hybrid (private/public cloud) approach [13]. In this case, a IaaS scenario can accommodate for more technological flexibility than PaaS since application providers can decide which components and platforms to use in their own virtual machines. This will be the most common requirement when adapting/migrating legacy applications to a hybrid cloud environment. In this case, non-functional aspects of applications become even more critical: enterprise applications have to deliver in terms of ***QoS*** (Quality of Service, e.g., with service level objectives on response times and availability) and face the risk of monetary penalties in case of violations. In IaaS, the design space for application logic, services, components, and VMs is considerable, and application elasticity (and their cost) is under the complete control of the application provider. Hence, in IaaS it is even more essential than in PaaS for application providers to have a **complete grasp of their application behaviour** in functional (F) and non-functional (NF) terms in order to make the right design decisions. The sought equilibrium is striking the balance between under- and over-provisioning, considering the effects that design decisions will have on performances, cost, and revenue.

In the following section we introduce the framework we foresee to support the all-around (F+NF) quality of Web applications deployed on the Cloud.

[6] `http://code.google.com/appengine/docs/billing.html`
[7] `http://code.google.com/appengine/docs/quotas.html`
[8] `http://code.google.com/appengine/docs/adminconsole/instances.html`

5 Model-Driven F+NF Quality Framework

The *functional quality* of model-driven applications is one of the **cornerstones** of Web engineering. It stems directly from *software product lines* [18] concepts combined with model-driven development and domain specific languages (DSLs). Briefly, WebEng DSLs allow modelling primitives to be implemented once by expert developers and then consistently reused by instantiating and configuring these primitives in the application models. In WebML [7] for example, the runtime service implementing each type of "unit" is a single parametric class that gets configured at runtime through unit descriptors: one descriptor specifying which attributes to show, how to sort, how to query for each unit instance in the model. The net result of this is that, once the parametric class for a unit has been thoroughly tested, the only way for one of its deployed instances to produce a failure is wrong configuration[9]. Simply said, WebEng DSLs strongly enforce **reuse**, a proven way of **reducing faults** and **improving quality**.

The other great advantage of WebEng approaches with respect to traditional development is **productivity**. Notwithstanding the (still) limited industrial adoption of WebEng techniques, it has been repeatedly proven in practice (see for instance the work by Acerbis et al. [1]) that, where the language primitives are sufficient to represent the needed application logic, model-driven approaches yield an advantage in terms of time and effort needed to design, develop, deploy, and maintain Web applications. There is a general trend that recognises the benefits of factoring out some concerns to be considered as separate aspects (e.g., CSS for style and appearance) in Web development, but so far only WebEng modelling has been able to offer instruments to do it effectively for Web application logic. In the end, increased productivity allows more **flexibility** in face of enterprise-specific and ever-changing requirements, maintenance, and continuous development of new features that are typical of the Web today. No need to recode, re-factor, or deal with software modules and artefacts, just update the model and generate the new code.

Cloud computing ***complements the functional flexibility of WebEng DSLs with flexibility in the non-functional aspects***: application elasticity. In order to preserve and combine the benefits of both worlds, we propose an extension to WebEng methodologies that caters for quality (both functional and non-functional) of Cloud-based Web applications across their complete life-cycle. A high-level overview of the framework is shown in Figure 1. For space reasons, we only illustrate it considering the more complete and complex scenario of IaaS deployment. PaaS deployment would only use a subset of the considered activities, which are:

1. **DSL modelling:** to achieve reuse, quality, productivity, and flexibility of a model-driven approach.
2. **Model transformation:** in order to add elasticity to the equation, WebEng *model transformation*, code-generation, and runtime support need to leverage

[9] To mitigate this, most WebEng tools provide warning generation mechanisms to prevent misconfigurations by checking the models through language-specific rules.

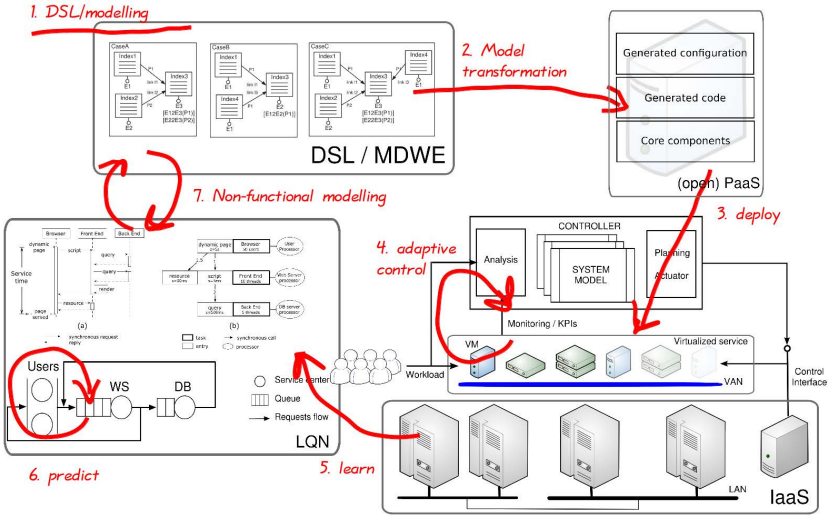

Fig. 1. Quality framework for Cloud-based Web applications

scalable technologies and patterns. Furthermore, appropriate monitoring probes need to be implemented and embedded in VMs to enable runtime monitoring and control [9].

3. **Cloud deployment:** the advantage of using Cloud computing in this case is the relative simplicity in setting up dynamic staging and production environments thanks to virtualization of resources.
4. **Adaptive control:** it consists in the management of the application elasticity. In production, most IaaS providers offer a user interface for dynamically adding and removing virtual machines at runtime, another common approach is to set up *auto-scaling rules* based on some low level metric (e.g., CPU load, memory usage). When considering enterprise applications, with several components and complex service mixes, manual scaling and rules can be too simplistic and model-based controllers are needed [2]. Traditional design-time capacity planning techniques (e.g., linear models, simple queues) also have drawbacks in a Cloud computing setting as they are unable to keep up with changes and emerging behaviour and tend to become unrealistic over time [3]. The solution we suggest is using system performance models to achieve *autonomic control*. For instance, a statistical model and an evaluation is proposed by Bodík et al. in [3]; while the authors propose an approach using Kriging models and their evaluation of model prediction quality in [17].
5. **Learning:** Autonomic controller models keep updating themselves at runtime. The knowledge gathered in this learning process makes the controllers *robust* to the environmental (e.g., workload peaks) and application changes to which Cloud-based systems are constantly exposed.

6. **Prediction:** Collected knowledge from the running system can be leveraged for parameter estimation of layered queue networks (LQN) or other models, allowing performance prediction across the whole application elasticity range.
7. **Non-functional modelling:** Finally, performance models from data obtained at runtime and LQNs can be combined with WebEng models to close the loop. This, as we proposed in [12], enables *modelling taking into account non-functional concerns*. For each modelling action, Web developers can receive an immediate feedback of the effects it will have in terms of system performances and costs. By providing a model of the expected workload range and performance service level objectives (SLOs) for each page / operation, a warning system can be implemented to explicitly signal when SLO violations or budget exceedances are predicted in the workload range.

The users of the framework are not required to be performance experts nor system engineers. In this aspect, all non-functional concerns can be considered separately using software automation. Given a formal set of SLOs, solutions like the ones we proposed in [12] and [17] can be integrated and work in a totally autonomous fashion. In this way, novice users will only receive warnings in case the modelled pages / operations might violate SLOs, while more advanced users will be able to get complete predictions and monitoring of the response time, throughputs, and system configurations. The advantage of this approach is that it allows Web engineers to **change their design incrementally and in a controlled way** keeping at all times application's performances and costs under strict supervision.

6 Related Work

To the best of our knowledge, this is the first work that addresses the problem of adapting Web engineering methodologies for the Cloud computing paradigm; some of its aspects are however not new. The concept of Model-Driven Performance Engineering, for instance, was introduced by Frietzsche and Johannes in [11] and consists annotating models with performance information to enable performance analysis, for example with LQNs as in [15]. As we discuss more extensively in [12] our approach is similar, but in our case we can benefit from a continuous online estimation of LQN parameters and an arbitrarily accurate definition of the LQN starting from WebEng and PaaS models.

Various authors have addressed the topic of migrating applications to the Cloud, focusing in particular on legacy applications as in [16]. With respect to these works, our proposed approach has the unique feature of dealing with model-driven methodologies that, by leveraging reuse, allow for a rationalized migration process. In fact, only the runtime primitives of each modelling language need to be migrated to Cloud-specific technologies (as we explain in Section 3), then model transformation can generate the rest of the needed configuration.

A relatively large amount of work has focused on automatic control of application elasticity, we cited some relevant works in Section 5. With respect to the framework we propose, these are all interchangeable solutions that can be adopted (e.g., different autonomic controller implementations), the paper claims no contribution in this area.

Escalona and Koch survey how WebEng approaches deal with capturing, specifying, and validating Web requirements [10]. However, the considered non-functional requirements are mainly concerning *usability* aspects; no formal specification or assurance of *performance* aspects is explicitly considered by any methodology at platform-independent model level. Instead, in this work, we claim that performance aspects directly reflect on application running costs in a Cloud computing scenario, and therefore should be explicit.

Finally, the concept of "Web engineering as a service" is not new, as it is introduced as "modelling as a service" in [5]. However, this is the first contribution that proposes (admittedly at very high level) a methodology for it that caters for functional, performance, and monetary aspects in the continuous-development life cycle of Web applications.

7 Conclusions

In this paper we discuss how to enable Web engineering methodologies to address Cloud computing development. We report that one of the main economical reasons for Cloud computing adoption is in the savings that the "pay-per-use" model enables for elastic applications.

We argue that current Web engineering methodologies do not make use of elastic technologies and would need an update to support Cloud development. More importantly, we sustain that Web engineering methodologies are generally too focused on functional concerns to be able grasp the main drivers of Cloud-based Web applications, which are performances and costs. To this end, we discuss the main interventions we deem necessary in order to combine the functional flexibility of WebEng DSLs with the flexibility in the non-functional aspects offered by the Cloud paradigm. In particular we relate on: 1) updating code generation and runtime libraries to support elastic technologies; 2) exposing Cloud-specific design decision in platform-independent models; 3) upgrading modelling tools to achieve Web engineering "as a service". Additionally, we propose a methodological framework to extend WebEng methodologies to consider non-functional (and in particular performance) concerns that are crucial in the Cloud pay-per-use model.

Acknowledgments. I wish to thank Florian Daniel at UniTN, Alessio Gambi at USI, Emanuele Molteni and Roberto Acerbis from WebRatio, and Piero Fraternali at PoliMi for the comments and discussions on the subject. I also wish to thank the anonymous reviewers for the valuable suggestions on how to improve the paper.

References

1. Acerbis, R., Bongio, A., Brambilla, M., Tisi, M., Ceri, S., Tosetti, E.: Developing eBusiness Solutions with a Model Driven Approach: The Case of Acer EMEA. In: Baresi, L., Fraternali, P., Houben, G.-J. (eds.) ICWE 2007. LNCS, vol. 4607, pp. 539–544. Springer, Heidelberg (2007)

2. Barna, C., Litoiu, M., Ghanbari, H.: Model-based performance testing: NIER track. In: ICSE, pp. 872–875. IEEE (2011)
3. Bodík, P., Griffith, R., Sutton, C., Fox, A., Jordan, M., Patterson, D.: Statistical machine learning makes automatic control practical for internet datacenters. In: Hot Topics in Cloud Computing, p. 12. USENIX Association (2009)
4. Brewer, E.: Towards robust distributed systems. In: ACM Symposium on Principles of Distributed Systems, pp. 1–12 (2000)
5. Brunelière, H., Cabot, J., Jouault, F.: Combining Model-Driven Engineering and Cloud Computing. In: Modeling, Design, and Analysis for the Service Cloud - MDA4ServiceCloud 2010, Paris, France (June 2010)
6. Cattell, R.: Scalable sql and nosql data stores. ACM SIGMOD Record 39(4), 12–27 (2011)
7. Ceri, S., Fraternali, P., Bongio, A.: Web Modeling Language (WebML): a modeling language for designing Web sites. In: WWW, pp. 137–157. North-Holland Publishing Co., Amsterdam (2000)
8. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., Wolski, R.: Appscale: Scalable and open appengine application development and deployment. In: Cloud Computing, pp. 57–70 (2010)
9. Clayman, S., Galis, A., Toffetti, G., Vaquero, L.M., Rochwerger, B., Massonet, P.: Future Internet Monitoring Platform for Computing Clouds. In: Di Nitto, E., Yahyapour, R. (eds.) ServiceWave 2010. LNCS, vol. 6481, pp. 215–217. Springer, Heidelberg (2010)
10. Escalona, M., Koch, N.: Requirements engineering for web applications-a comparative study. Journal of Web Engineering 2, 193–212 (2004)
11. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: Giese, H. (ed.) MoDELS 2007 Workshops. LNCS, vol. 5002, pp. 164–175. Springer, Heidelberg (2008)
12. Gambi, A., Toffetti Carughi, G., Comai, S.: Model-driven web engineering performance prediction with layered queue networks. In: Proceedings of Model-Driven Web Engineering Workshop (MDWE) (2010)
13. Gartner, Inc. Paas road map: A continent emerging (2011), Report ID Number: G00209751
14. Ghemawat, S., Dean, J.: Mapreduce: Simplified data processing on large clusters. In: Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, CA, USA (2004)
15. Gu, G.P., Petriu, D.C.: Xslt transformation from uml models to lqn performance models. In: WOSP 2002: Proceedings of the 3rd International Workshop on Software and Performance, pp. 227–234 (2002)
16. Hajjat, M., Sun, X., Sung, Y.-W.E., Maltz, D., Rao, S., Sripanidkulchai, K., Tawarmalani, M.: Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In: Proceedings of the ACM SIGCOMM 2010 Conference, SIGCOMM 2010, pp. 243–254. ACM, New York (2010)
17. Toffetti, G., Gambi, A., Pezzè, M., Pautasso, C.: Engineering Autonomic Controllers for Virtualized Web Applications. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 66–80. Springer, Heidelberg (2010)
18. van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)