# A Vision for Better Cloud Applications

Keith Jeffery
STFC
Harwell Oxford
Didcot OX11 0QX UK
+44 7768 446088
keith.jeffery@stfc.ac.uk

Geir Horn
SINTEF
Forskningsveien 1
0314 Oslo, Norway
+47 93 05 93 35
Geir.Horn@sintef.no

Lutz Schubert
HLRS
Nobelstr. 19
70569 Stuttgart, Germany
+49 711 685 87262
schubert@hlrs.de

## ABSTRACT
In this paper, we provide an overview over the PaaSage project's approach to helping the developer in exploiting cloud environments according to their specific needs and requirements. Classical software engineering methodologies no longer apply in multi-tenant, elastic environments, if the full capabilities for cost reduction and availability are to be exploited. PaaSage aims at offering software engineering extensions covering the full application lifecycle from deployment to execution.

## Categories and Subject Descriptors
C.2.4 [**Computer-Communication Networks**]: Distributed Systems, D.1 [**Programming Techniques**], D.2.11 [**Software Engineering**]: Software Architectures, I.2.5 [**Artificial Intelligence**]: Programming Languages and Software - *Expert system tools and techniques*

## Keywords
Application Decomposition, Quality of Service, Development Support, Software Engineering Tools, Programming Expert System

## 1. INTRODUCTION
Cloud computing is revolutionising the Information Technology industry through its support for utility service-oriented Internet computing without the need for large capital outlays in hardware to deploy their service or the human expense to operate it.

Currently there exist several open source and commercial offerings at the Infrastructure as a Service (IaaS) level, like Windows Azure, Amazon Elastic Compute Cloud, Flexiant Flexiscale, Eucalyptus, OpenNebula and many others. Software developers targeting the Cloud would ideally want to develop their software once and be able to deploy it on any of the available offerings, reaping the benefits of a Cloud market without losing on performance, availability, or any other service properties. An impediment to this objective is that IaaS Cloud platforms are heterogeneous, and the services and Application Programmer Interfaces (APIs) that they provide are not standardised. These platforms even tend to impose a specific architecture on deployed applications. Accordingly, there is a significant dependency between client applications and the services provided by the platform, which is not well specified or appropriately communicated to the user. Knowledge with respect to which use case is most suited for which platform and how to exploit these features is therefore hard and costly to gain (cf. [1]).

It is generally up to the developer to specify and exploit these characteristics to her best knowledge. This however is the general crux: not only will the typical developer neither know how to use these characteristics, nor how they impact on the overall behaviour, and what is more, how they relate to a given Cloud infrastructure. This is complemented by the fact that most infrastructures do not even offer support to exploit these characteristics, such as location control, specification of scale out behaviour and so forth.

The paper is structured as follows: in section 2 we investigate the problems faced by potential cloud uptakers, outlining what kind of information is required by the developer to fully exploit the cloud capabilities. In the following section, we will elaborate how PaaSage contributes to generating and providing this kind of expertise and how the developer can make use of it. Section 4 will exemplify the benefits of this approach in a very concrete use case of wide public interest. We conclude the paper with section 5.

## 2. CLOUD WITH NO SILVER LINING
Even though clouds offer a high potential in reducing cost and increasing efficiency, reaching this potential is difficult for multiple reasons. Foremost, there is still a lack of experience how to build and maintain an application that can really exploit the system capabilities and still adheres to the requirements of the use case, respectively its users.

This is particularly due to the fact that applications exploiting these capabilities need to adhere to opposing principles at the same time: they need to scale seamlessly, but should maintain data consistency; they need to be highly available, but should reduce the number of resources used; they should be accessible from anywhere anytime, but need to be secure etc.

In order to fulfill these requirements, it is primarily necessary to (1) completely change the application structure, catering e.g. for different degrees of consistency and scalability. However, the characteristics of an application are not only defined by its structure / architecture, but are also highly influenced by (2) the capabilities of the underlying hosting infrastructure. Typically, different providers offer different capabilities and thus serve different needs – in particular for more complex applications, a combination of characteristics is needed though, depending on usage and requirements.

Combined deployment and usage of "multi-cloud" requires however not only understanding of usage and impact of the system characteristics, but also (3) high expertise in deployment


*MultiCloud'13*, April 22, 2013, Prague, Czech Republic.

and configuration of the environment to fully exploit its features, since all providers not only offer different resources, but also different ways of dealing with them. The following sections will elaborate these aspects in more detail:

## 2.1 Development

Any application can essentially be regarded as a workflow, following a graph model. Modern software engineering principles base on services as software structures, where effectively each service can be compared to a task in the application workflow, thus leading to a higher level of granularity.

A good developer will try to break down the overall application requirements into sub-requirements per each task. The relationship between these aspects is however hardly ever clear and it generally requires a lot of experience and expertise on behalf of the developer to actually estimate the impact of each individual module's behavior on the overall application.

Whilst some general methodologies can be applied, such as applying asynchronous execution of some tasks, limiting the response times through time-outs etc., the general decomposition of these properties is still difficult. For example, how much a background task affects overall execution time due to resource sharing, how insecure the overall application becomes from an insecure database connection, whether consistency is maintained if two tasks are invoked asynchronously etc.

Non-functional properties, such as performance, security, safety, reliability and in particular cost play a major role in acceptance and usability, and hence success of an application. Good software teams will therefore build strongly on such expertise to build the code accordingly.

With the introduction of clouds, this decomposition problem has increased manifold in complexity: no longer is it just the properties of the application as a single structure that have to be analysed and maintained, but instead the whole usage context needs to be taken into consideration.

Complex modern cloud applications are designated by integration of multiple different services and the necessity to be (dynamically) available for multiple users whilst reducing the resource load. The primary factors that add to classical application structures therefore are: (1) system specifics and (2) multi-tenancy behavior.

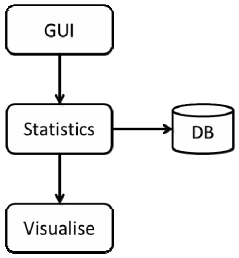


**Figure 1: a simple application workflow**

We can illustrate this with a very simple application providing data visualization to users consisting of a graphical user interface, a statistical database analyzer and a visualization service for displaying the resulting data (cf. Figure 1).

This code structure is sufficient for classical software engineering principles, i.e. for a single instance, single user environment. When the capabilities should be shared between multiple users, however, further aspects need to be taken into consideration, such as which data is shared under which conditions, which tasks / results can be reused etc.

In the given example, the requirements may lead to e.g. the GUI being required per each user (separate views, interactivity), the statistics being partially shared (same tasks), and the database being shared between all users. Visualisation in turn may be reused again and again for different data, if the cost for multiple instances is too high etc. thus leading to a dynamic view as depicted in Figure 2.

This should not be confused with a deployment or an extended relationship view, as the actual number of instances and their dependencies are mostly defined by the number of users and their specific requirements, respectively their context of usage. This can hardly be depicted with modern tools, or even modeling languages such as UML, let alone that there is a clear way of breaking down the conditions into requirements per task / module.

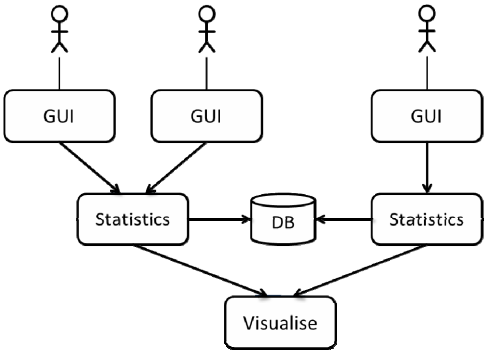


**Figure 2: a simple application workflow**

In more complex cases, even the boundaries between the modules will become less clear, making it even more difficult for the developer to specify the behavior or even to exploit. Generally so far, it requires highly skilled and experienced developers to generate applications that are typically highly adjusted to the specific use case, users and… the environment:

## 2.2 Provider Selection

As noted, each task exhibits different requirements for its best functioning, due to the usage conditions on the one hand, but also due to the specific implementation details on the other.

With respect to the example scenario in the preceding section, e.g. the database task is ideally hosted in an environment dedicated to databases, whereas statistics may require higher computing power and visualization benefits from GPU support – each of which may be offered through specialized PaaS providers dedicated to analysis and visualization tasks. In the simplest case, a single provider offers all the capabilities.

In particular on the level of PaaS, providers offer specific operation support dedicated to the type of platform offered. This generally improves the capabilities and reduces the development overhead for generating the application. At the same time, it increases the risk that the application is locked into that specific provider, i.e. that movement to another provider in the future becomes more difficult – this is however slightly outside the scope of this paper. Obviously, further criteria that play an immediate role are cost and potential legalistic concerns.

In complex, distributed cases, even more factors play a role in provider selection, though. First of all, no single provider will generally be able to satisfy all specified needs, but instead each provider will typically only address parts of the requirements, so

that either a compromise has to be found or a combination of providers need to be engaged.

Distributed deployment over different providers leads to implicit communication and interaction problems that have less impact in a local deployment. The distribution therefore needs to be carefully aligned with the task relationships. This is complicated even more by the circumstance that instances of each task (and therefore within any cloud environment) may be replicated and relocated dynamically. This means that the communication to and with it may alter dynamically, depending on the relative location and concurrent access to it.

In order to compensate for this behavior appropriately and in order to control the overall behavior according to the general requirements, the developer therefore not only has to provide the logical information of the code itself, but also of the tasks within the cloud. In other words, he has to provide the code logic and the behavioural logic in terms of how the tasks / instances replicate, relocate, scale down etc.

This obviously leads back to the problem of specialization and implicitly of vendor lock-in, since each provider exposes his own way of such control and what monitoring information is offered. Further to this, the developer needs to implement the necessary communication mechanisms across these provider boundaries, not only in terms of message exchange between the instances, but also in terms of monitoring information that influences the behavior of related instances:

## 2.3 Deployment

As indicated, the developer cannot simply select the most appropriate providers and deploy his task(s) in the according infrastructure / platform. Instead he will first of all have to adapt the code to the specifics of the respective provider – in particular in terms of exposed APIs and supported operations. What is more, however, he will have to encode the monitoring-control information in some form that allows enactment of the cloud-specific behavior within the destination platform.

This includes in particular such aspects as how and when to create new instances, how to maintain communication links, how and where to relocate instances etc. but implicitly also monitoring data such as current load, response time, number and location of instances etc. Not only do the providers offer different information and different capabilities, but more importantly, they expose different APIs to execute these operations, respectively to request the necessary monitoring data.

This means foremost that the developer will have to (re)program the respective task(s) for each destination provider individually and thus also has a problem migrating an application to another provider. This is commonly referred to as “vendor lock-in” and poses a serious issue in efficient usage of cloud environments.

What is more however, also the way of deploying an instance into a provider environment can vary strongly, non-regarding common image formats such as OVF by DMTF [2]. This affects not only the way in which the image provided, but also the interaction with the provider’s infrastructure. Frequently enough, this requires human interaction which is difficult to automate, thus limiting the degree of dynamicity.

Due to these circumstances, application providers are generally stuck to the selected providers, once their application has been created (respectively adapted) and the details for deployment been elaborated. This implicitly means that a large amount of effort has to be invested into the creation and deployment, and that dynamicity beyond the boundaries of the respective provider(s) is not possible, i.e. incorporating further clouds in order to extend the capabilities is only possible with a high degree of effort on behalf of the developer / application provider.

## 2.4 Consequences

All in all, developers and application providers face multiple problems when trying to adapt or create an application that makes effective use of the cloud infrastructure. Typical approaches will simply try to package and deploy the application as a whole without explicitly making use of the cloud capabilities. This may effectively lead to higher costs than straight-forward deployments in a dedicated data centre, depending on how well the application scales out as a whole and how dynamically it is used.

To make actual use of the cloud capabilities, expertise and experience is generally lacking, let alone an appropriate development and support tool. Such tools would have to be able to help in the decomposition of the properties and aligning them to the modular architecture of the application, and assess the impact of a given deployment on the fulfillment of these properties. The properties are however strongly influenced by the actual use case, i.e. expected usage behavior, so that additional information of this type needs to be provided.

Effectively, this means that the full application and usage context needs to be analysed when structuring and developing the data. Any support system therefore needs to integrate right into the full design process and use the information gained to support the adaptation and deployment process – this means transformation of the code for the respective destination cloud, provisioning of monitor-control loops for behavior enactment and injection of appropriate communication mechanisms to maintain data exchange according to the workflow specifications.

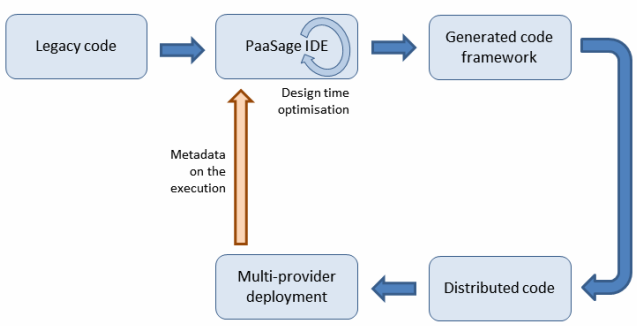


**Figure 3: the analysis and usage cycle according to PaaSage**

In order to create the desired behavior, it is implicitly necessary to monitor the behavior of the tasks / modules at execution time and on the one hand enact adaptation processes in accordance with the behavior policies generated during the design and modularization and on the other hand, distribute the information to all instances directly affected by the behavior, so that the application as a whole can react accordingly.

This information can be put back into context of the usage scenario and the selected deployment, so that it may be helpful to reinvestigate it in future adaptation and deployment cycles. PaaSage aims furthermore at exploiting this information for further improvement of the tool’s knowledge base and thus capability extension. PaaSage thus foresees a development support cycle as depicted in Figure 3.

# 3. THE PaaSAGE TO THE FUTURE

This chapter will elaborate the concrete support by PaaSage in more detail.

Obviously, it is thereby up to the user whether to share this information in the first instance.

This workflow covers the following logical steps (cf. Figure 4):

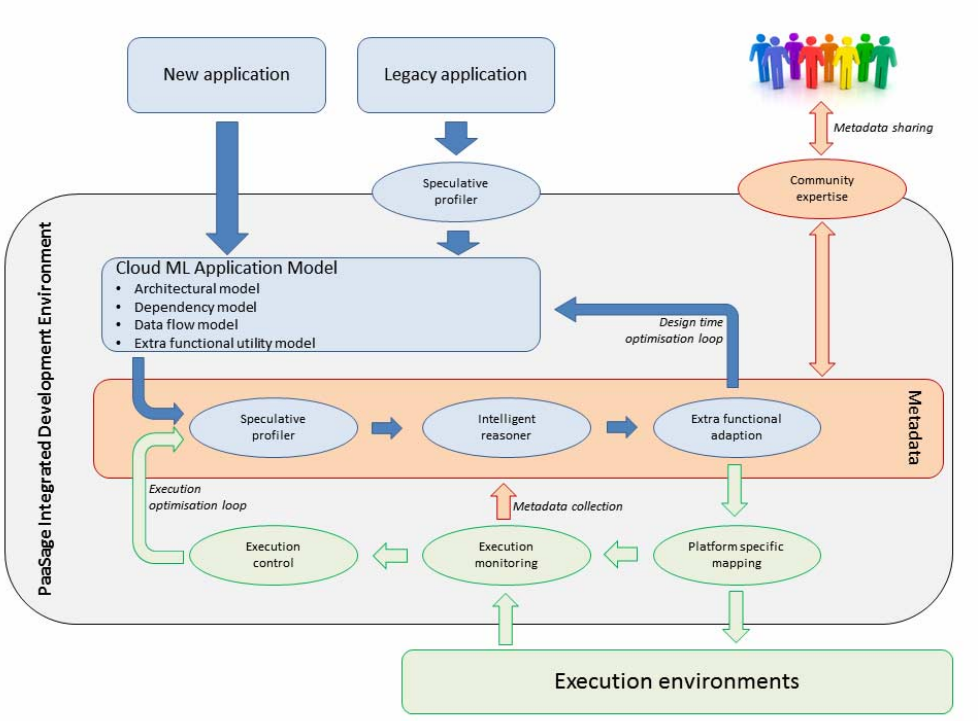


**Figure 4: the detailed PaaSage workflow**

## 3.1 Architecture

There are three main components in our vision:

1. An **integrated development environment (IDE)** extending the popular open source development platform Eclipse supporting our Cloud Modelling Language (CloudML) and integrating the various components developed by PaaSage.
2. The **executionware** dealing with the platform specific mapping of the application to the architectural model and Application Programming Interfaces (APIs) of the execution infrastructure of the Cloud provider, and with capabilities of monitoring the running application and possible reconfiguration to optimise its behaviour.
3. The **upperware** which is a collection of tools and components used to assist the application development or porting at design time, and then integrate with the execution ware at run-time to facilitate the optimisation of the running application and execution platform.

These main parts should all integrate on the same service and component metadata database containing historical information about past service invocations like the cost of execution, the performance date for different execution platforms and Cloud providers. It should be possible for a developer to draw on metadata gathered by other users of the same Cloud platform or software service This is important since many properties are stochastic and having more data available allows better decisions.

### *3.1.1 Application definition*

If the starting point is an existing application, it is first passed through the profiler trying to understand the structure of the application, and to automatically provide a starting point for the Cloud application model. The speculative profiler[1] will try to build automatically a starting point for the Cloud ML application model, and estimate initially as many of the model's parameters as possible. A new application can be directly defined in the modelling language using the IDE.

### *3.1.2 The application model*

PaaSage aims to cater for all types of applications, including parallel applications. Traditionally they have an architectural model consisting of a *Directed Acyclic Graph* (DAG) essentially defining the different execution models or processes of the application and how data is passed from one module to the next as execution progresses. Any application that can be seen as a composition of services can be viewed in this way. The DAG view is simplistic, as re-entrant processes or iterations requires cycles, and a good model will need not only a formal syntax for describing the relation among modules, the edges of the graph, but also a rich semantics with quite complex structures on the edge modelling the relationship between two successive modules; and the pre- and post-constraints for execution of each module.

[1] Observe that there are two uses for the output of the speculative profiler: The one simplistically indicated in this step, where the Cloud ML model is build, and another where the Cloud ML application is profiled.

The model will also specify the data flow model, with annotations covering the volume of data, the storage location and potentially the format of the exchanged data. Finally, the model must also include extra functional properties and the user's preferences for maximised utility - for example shortest possible execution time of a module, or minimal execution cost with a trade-off on how to balance these two potentially conflicting properties.

### *3.1.3 The design time optimisation loop*

The modules and services defined in the application model can be seen as examples drawn from the broader class of similar modules and services available in the market. The role of *the speculative profiler* in the design optimisation loop is to draw on the stored historical metadata information on the modules and services, and map out alternative realisations of the application[2]. Note that this also includes choices for the execution – e.g. a data intensive application should be executed close to the data, whereas computationally intensive modules will benefit from the fastest possible execution platform. The *intelligent stochastic reasoner* will then try to find a *feasible* realisation satisfying the constraints of the application when the historical metadata statistics are taken into account. Finally, the *adaptation* module evaluates alternatives and identifies variation points and "triggers", or thresholds, on extra functional properties like cost or execution time that can be used at run-time to initiate a "re-planning" of the application. For instance, a given slow service may be proposed by the reasoner minimising the cost, however if the application execution time violates a threshold this should be replaced by a faster, but more expensive alternative implementation. The adaptation module will also identify potential alternative service providers. The end result of this optimisation is an enhanced and annotated application model with provided values for expected values of constraints and extra functional properties informing the developer that executing this particular application realisation will take *this* long on average and is expected to cost *this* much. If this result is unsatisfactory for the user, she can then decide to revert to the modelling in step 2 to use other services or modules or change the application structure or parameters. If the feasible realisation is acceptable the user may decide to proceed to execution in the next step.

### *3.1.4 The execution optimisation loop*

The first and major step in this loop is the *instantiation of* the application and the mapping of its modules to the architectures and APIs of the underlying execution environments. The core idea of PaaSage is to offer full flexibility to mix and match the best offerings from private Clouds with various commercial Cloud offerings and services, thus the typical mapping will be to map the application on many execution platforms. The result of this mapping is the source code of the *distributed* application. If necessary for a new application, the different communicating modules can then be implemented by programmers; or if this is a module of a legacy application the functionality can be imported from the corresponding legacy module. The final stage of the mapping is to deploy the code on the execution platforms. During execution, the different modules will be *monitored*. The data gathered serves two purposes: Firstly, it is fed back to populate the metadata database improving the knowledge of historical modules and services executions. Secondly, if some of the thresholds identified by the adapter during the design time optimisation are exceeded, the *execution control* is triggered to do a "models@runtime" adaptation and redeployment of the application. However, there could be cases where there is no better variant ready for execution identified at design time. In this case the execution controller may decide to invoke again the speculative profiler, then the stochastic reasoner, and the adaption to produce a new feasible application that will be mapped and deployed. Note that this optimisation will typically be performed with constraints different from the original ones with the objective to improve the execution of the remaining modules of the application run, so that the original execution targets for the application can be met. This optimisation may also be undertaken at the normal application termination in order to perform better next time the application is executed.

## 3.2 The Issues

Realising PaaSage architecture is highly challenging. However, this challenge has to be met if it is to be made possible to deploy elastic services across different CLOUD platforms.

One major issue thereby is the metadata which is (a) stored in the metadata database and (b) wrapped around the black box artefacts (services, components) of PaaSage. The metadata database includes not only the metadata associated with the characterisation of the artefacts themselves (partitioned into functional and non-functional sets of attributes) but also the historical information of required characteristics (SLA, QoS) and the characteristics actually achieved in execution under PaaSage.

The functional attributes cover the functional signature of the artifact, i.e. the variables (or data structures) input and output together with any functional control parameters such as optional controls (e.g. fixpoint, termination point, precision required). The non-functional requirements come at different levels: the user requirement level (e.g. elapsed execution time or execution termination point, execution time, cost) and at the systems requirement level (e.g. need for multicore, need for GPUs, latency limits, memory requirements, dataset dependencies - including locality).

It is thereby not so much the problem to identify the right metadata, but in particular to break it down into specific *actionable* attributes for each module that the executionware can apply at runtime. As such, the executionware in itself faces more technical challenges and constraints (API exposed by the infrastructure, monitorable data etc.) than a complex logic. One particular aspect worth mentioning in this context consists in the challenge of maintaining data consistency and communication over boundaries that were not directly foreseen by the application itself. It must thereby be kept in mind that the monitor-control loop creates further communication overhead (with the metadata-database, other modules etc.) that further impacts on performance. Thus, resiliency and adaptability must be carefully weighed against performance of the whole system.

## 3.3 The Method

The architectural design is framed by (a) end-user requirements, documented as a high-level business case, a ‘recipe’ of processing

[2] Please note that owing to the "combinatorial explosion", i.e. exponential growth in the number of possible realisations, we do not intend to construct them all. This is where the *speculation* comes in, as the profiler will try to use past experience and heuristics to identify only realisations that are likely to be feasible.

steps and a UML diagram; (b) the exploitation plans of the project partners balancing the desire for an open platform with additional features for commercial exploitation by the software vendors but also for the application stakeholders. Within this framing - and coordinated by the architectural design - the upperware and executionware will be designed, closely linked with and by the CML (CLOUD Modelling Language) and the metadatabase.

# 4. SERVICES FOR THE CITIZENS

Applications in the public sector are well placed to illustrate the above concepts. Such applications will typically consist of an integration of many legacy applications, as well as interfaces where the citizens can provide input or get access to information stored on them by the public authorities. As an example one may consider an automated tax bill application. This will typically link with one or more public registers showing who lives where, collect information about income from employers and wealth from various banks using some kind of unique key, and information about family constellations from other registers. The amount of taxes due should be computed based on the legal framework and a tax form should be produced for each citizen. Finally, the citizens may themselves be invited to log into the system to verify and confirm the information. The final tax bill may then again serve as the basis for various social security support systems.

It is intuitively understandable that developing this application will entail integrating a huge amount of applications offered as services by a variety of private and public institutions, with provider specific data protocol mappers and interfaces. Therefore, just designing the application as a workflow or data flow among different sub-systems and modules is a challenge in its own right.

Deploying such an application in the Cloud is currently prohibitively difficult. There are conflicting concerns related to the handling of personal data which should not be put in the Cloud versus the elasticity of the application. There are legacy systems and databases, which might have been optimised for certain computing platforms (mainframes), and cannot be easily virtualised. Finally, there may be budget constraints for the different organisations involved that limits the flexibility.

When this application is designed for the cloud it is important that all these constraints can be modelled in order to ensure that the legal requirements are satisfied. For instance, tightly coupled modules like the database containing the sensitive information on income and wealth should be placed in the same datacentre as the modules running the algorithms for the tax computations. For privacy reasons this datacentre should be within the national borders. However, for performance issues one might want to distribute the data on the citizens across several databases, for instance based on their domestic locality, and then run the tax computations in parallel. Thus, one would establish a private Cloud with several national datacentres.

When the computations are done, one might expect that many citizens would want to check their tax bills when it is released so that there will be a huge load on the web servers for a limited amount of time. This part of the application could in principle be delegated to the public Cloud provided that one can establish a secure and scalable way to grant access and display or record information. For the tax authorities it would be cost efficient to rent a huge amount of virtual servers for a limited time period instead of investing in hardware that would remain unused for most of the fiscal year. The application model must in this case capture the modules that can be deployed as multiple instances, together with the appropriate architecture for load balancing the incoming user sessions and the back-end database requests, and provide mechanisms to deploy these on heterogeneous Clouds.

The application life-cycle finally entails monitoring of the running application, and ability to self-optimise in case of bottlenecks and self-heal in case of errors. These short term corrective actions must be supplemented with feedback to the application developer aiding the design time optimisation of both the structural application model as well as the deployment goals. Over time, this will enable the application to provide the best possible service to the citizens, and to guide public investments in both hardware and software to the places needed for removing bottlenecks.

# 5. CONCLUSION AND FUTURE WORK

As has been shown by this paper, full exploitation of future (cloud-based) infrastructures leads to multiple challenges that have not been properly addressed so far. In fact, to develop an application that really benefits from the cloud capabilities requires a high degree of experience *and* expertise by the developer – which cannot really be expected yet. Future approaches must not only address the programmability, but in particular mechanisms to incorporate essential expertise right from the beginning.

The paper has presented PaaSage's approach, which essentially aims at extending the capabilities of the developer to generate efficient cloud applications. In other words, builds a combined programming and execution environment that incorporates this expertise throughout the whole lifecycle. The project thus not only generates the necessary tools, but provides adjusted programming and execution principles that directly address explicit challenges and cloud provider / user will face and that current principles don't reflect.

It has been shown how applications developed according to these principles allow the developer to address multiple platforms with reduced effort in decomposing the non-functional requirements, as well as deploying the application in a fashion that these conditions are met and data relationships are maintained. PaaSage thus not only paves the way towards efficient cloud computing, but also towards exploitation and realization of federated clouds.

# 6. ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 317715.

The views expressed in this paper are those of the authors and do not necessarily represent those of the consortium.

# 7. REFERENCES

[1] Schubert, L., Jeffery, K., Neidecker-Lutz, B. (2012): Advances in Clouds – Research in Future Cloud Computing. Cordis (Online). Brussels, BE: European Commission. Retrieved from. http://cordis.europa.eu/fp7/ict/ssai/docs/future-cc-2may-finalreport-experts.pdf

[2] DMTF (2010): Open Virtualization Format Specification v1.1.0. Retrieved from: http://dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf