SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2015)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2320

Migrating legacy software to the cloud: approach and verification by means of two medical software use cases

Pieter-Jan Maenhaut[1,2,*,†], Hendrik Moens[2], Veerle Ongenae[1] and Filip De Turck[2]

[1]*Department of Industrial Technology and Construction, Faculty of Engineering and Architecture, Ghent University, Valentin Vaerwyckweg 1, 9000 Ghent, Belgium*
[2]*Department of Information Technology, iMinds - IBCN, Ghent University, Gaston Crommenlaan 8 bus 201, 9050 Ghent, Belgium*

SUMMARY

Cloud computing is a technology that enables elastic, on-demand resource provisioning, allowing application developers to build highly scalable systems. Multi-tenancy, the hosting of multiple customers by a single application instance, leads to improved efficiency, improved scalability, and less costs. While these technologies make it possible to create many new applications, legacy applications can also benefit from the added flexibility and cost savings of cloud computing and multi-tenancy. In this article, we describe the steps required to migrate existing applications to a public cloud environment, and the steps required to add multi-tenancy to these applications. We present a generic approach and verify this approach by means of two case studies, a commercial medical communications software package mainly used within hospitals for nurse call systems and a schedule planner for managing medical appointments. Both case studies are subject to stringent security and performance constraints, which need to be taken into account during the migration. In our evaluation, we estimate the required investment costs and compare them to the long-term benefits of the migration. Copyright © 2015 John Wiley & Sons Ltd.

Received 25 March 2014; Revised 5 February 2015; Accepted 12 February 2015

KEY WORDS: cloud computing; software-as-a-service; multi-tenancy; migration; medical software

1. INTRODUCTION

Cloud computing is a technology that enables elastic, on-demand resource provisioning. Over the last few years, many companies have used clouds to build new highly scalable systems. However, legacy applications can also benefit from the advantages of cloud computing, and there is a general trend for moving applications to a cloud infrastructure, consolidating hardware, saving costs and allowing applications to react faster to sudden changes in demands. With the recent evolution of cloud computing [1] and Software as a Service (SaaS) in particular, an elastic, scalable multi-tenant architecture has gained popularity [2]. Elastic systems are able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner. With cloud computing, an optimal usage of available resources is recommended to reduce operating costs, as the infrastructure provider usually charges for the number of instances used. SaaS is a software delivery model in which the software and associated data are centrally hosted on the cloud, and the end-users are typically accessing the software through the browser or by using a thin client. As the number of clients grows, a scalable architecture for both the application and data is needed.

Multi-tenancy [3] enables the serving of multiple clients or tenants by a single application instance. The major benefits include increased utilization of available hardware resources and

*Correspondence to: Pieter-Jan Maenhaut, Department of Industrial Technology and Construction, Ghent University, iMinds-IBCN, Valentin Vaerwyckweg 1, 9000 Ghent, Belgium.
†E-mail: pieterjan.maenhaut@intec.ugent.be

improved ease of maintenance and deployment. Without a multi-tenant architecture, the cost savings using cloud computing are limited for applications requiring continuous availability, as for every new client (tenant), a separate Virtual Machine (VM) instance would have to be provisioned. This instance must then be available at all times, even if it is only used sporadically. Also, as every tenant has a dedicated instance, some resources would be wasted, especially for smaller clients. Using a multi-tenant architecture, a SaaS application could run on few instances that are shared between the different users, and the number of instances could dynamically grow with the current demand. Smaller tenants could be co-located on a single instance, minimizing costs and maximizing resource utilization.

Therefore, when migrating applications to the cloud, it is recommended to adapt the legacy software to support multi-tenancy. Some changes to the architecture will be necessary, coming at a one-time cost, but this cost is overruled by the long-term benefits. Apart from adapting the legacy software for supporting multi-tenancy, some other changes may be needed to support the migration to a public or hybrid cloud, as every Platform as a Service (PaaS) or Infrastructure as a Service (IaaS) provider will have its own limitations and possibilities.

In this article, we propose an approach for both migrating applications to a hybrid or public cloud, and for adding multi-tenancy to the existing software with a minimal overhead. We verify our approach using two different case studies of legacy medical applications that are migrated to the cloud and discuss the required changes. We describe the advantages and disadvantages of moving components of the software to the public cloud and evaluate the migration costs.

In the next section of this article, we will discuss related work. Afterward, in Section 3, we will present the approach for both migrating legacy software to the cloud and adding multi-tenancy. We verify this approach in Section 4 and Section 5 using two different case studies. In Section 6, we discuss our approach and present our evaluation results. In Section 7, we state our conclusions and discuss avenues for future research.

2. RELATED WORK

In previous work [4], we described the steps required to migrate an existing .NET-based application to the Windows Azure public cloud environment and proposed a specific approach for adding multi-tenancy to the application. In this article, we propose a generic migration approach for migrating legacy applications to the cloud. We describe the different steps of our approach in detail and verify our approach by means of two case studies. In this article, we also present an extended discussion and evaluation based on the results from the two case studies.

An approach for partially migrating applications to the cloud is presented in [5], together with a model to explore the benefits of a hybrid migration approach. The approach focuses on identifying components to migrate, taking into account various rules such as performance and security. We also focus on migration to a hybrid or public cloud, but extend their approach by going into detail about the complete migration process, and not only selecting the components to migrate. We also present an approach for adding multi-tenancy to the application to optimal benefit from the migration to a public cloud.

When migrating software to the cloud, some choices have to be made. Different cloud computing service models exist, each having its own advantages and limitations. Figure 1 provides an overview of the different cloud service models. The legacy software could, for example, be fully migrated to a public cloud or a hybrid approach could be used. When it comes to public cloud providers, CloudCmp [6] offers a system for comparing the performance and cost of the different providers. For the implementation, the authors use computation, storage, and network metrics. For the storage metrics, they selected some benchmark tasks and measured the response times, throughput, time to consistency, and cost per operation.

Cost savings and other organizational benefits and risks of migration to IaaS are discussed in [7]. We, however, do not only limit our approach to migrations to an IaaS provider but also consider migrations to a PaaS platform. When using an IaaS provider, the customer has full access to the operating system (OS), middleware, and runtime, hosted on a virtual machine. On the other hand,

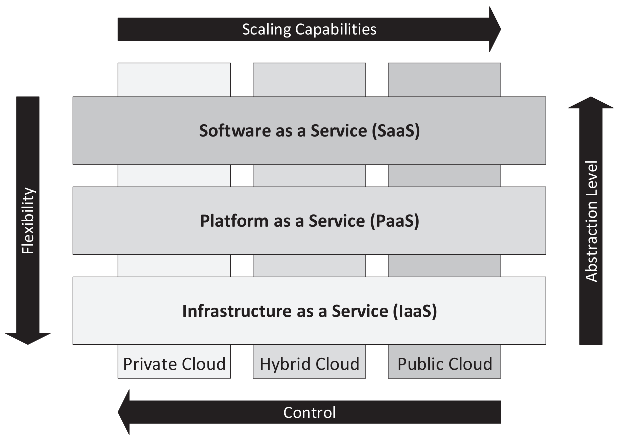


Figure 1. An overview of the different cloud service models used in cloud computing.

when using a PaaS provider, the customer only manages the application and data, which brings some limitations, such as the selected OS and supported frameworks and libraries.

In [8], a checklist is presented, which can be used to determine whether applications are compatible with a chosen PaaS provider. The approach is evaluated by three case studies where a Java application and two Python applications are migrated to Google App Engine. Three different and representative PaaS platforms are compared in [9], based on a practical case study, with respect to their support for SaaS application development. In this article, we focus on how complex applications can be executed on the public cloud, and for our case studies, we go into detail on migrating two different legacy applications. We do not only limit our work by determining whether the applications are compatible with the selected provider but also describe the different steps required in detail. Furthermore, we describe how multi-tenancy can be added, making it possible to better utilize individual application instances.

As our first case study handles a legacy application written in .NET, we selected Windows Azure to host some components of the legacy software. The migration of an on-premise web application to Windows Azure is described in [10], together with a comparison of the application's performance when deployed to a traditional Windows server versus its deployment to Windows Azure. While the cloud migration of a .NET application requires limited effort, Azure has no built-in support for multi-tenancy, so it must be added during the migration process. In this article, we discuss both the steps needed to migrate an application to the cloud and the steps needed to add multi-tenancy to the application.

To support highly customizable SaaS applications, we use a software product line-based customization approach, which we have previously discussed in [11–13] and [14]. In this approach, variability is modeled by defining multiple features and the relations between them. These features are then associated with separate code modules that are deployed separately. The application is then composed out of these multi-tenant components, resulting in an application that are both customizable and multi-tenant. For changes that do not impact the performance of the application, a multi-tenancy enablement layer can be used, which amongst others can be used for data isolation, feature management, and tenant-specific customizations [15].

In [16], we focused on the scalability of tenant data in multi-tenant applications and the impact on the performance of the application. The outcome of this research is used in [17] to build an abstraction layer for achieving high scalability for the storage of tenant data. This layer uses data allocation algorithms to determine an acceptable allocation of tenant data to different databases. The presented solution can be used for decoupling the databases and the management of tenant data; two of the steps in the approach presented in this article.

3. MIGRATION STRATEGY

In this section, we describe both the steps needed to migrate an existing application to the cloud and to redesign the application to support multi-tenancy. We start this section by briefly describing the concept of multi-tier architectures, a popular software architecture used by many applications, which we will refer to later in this section. Next, we discuss the different steps of our approach, as summarized in Figure 2.

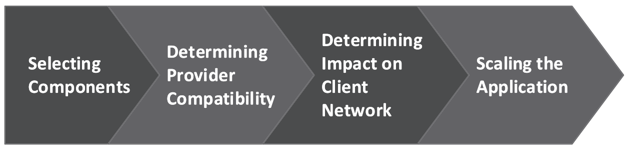
Many applications are designed using a multi-tier architecture, where the application is separated into multiple layers. A typical multi-tier architecture consists of three layers: the client layer, the business logic layer, and the database layer. We refer to this basic layered architecture as the *3-Tier architecture*. Most layered applications will have more than three layers, as more layers can be easily added to the architecture if needed. For example, when working with multiple database instances, an extra data access layer can be added between the business logic and database layer, responsible for load balancing and selecting the correct instance. Other architectures are possible, but in the remainder of this section, we will start from the 3-Tier architecture.

3.1. Cloud migration

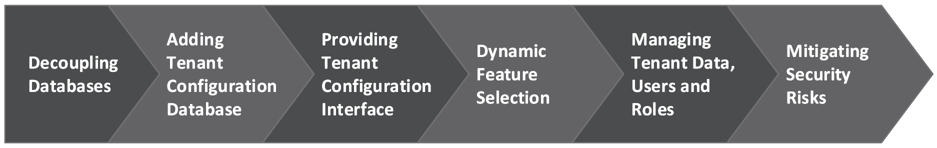
The process to migrate an existing application to a public or hybrid cloud can be summarized in a few steps, illustrated in Figure 2(a) and described below.

3.1.1. Selecting components. The first step during the planning phase should be to select the components of the software to migrate to the public cloud, as described in [5]. Both components of the business logic layer and the data access layers can be selected. The selection can happen based on the quality attributes of the application, to guarantee the required QoS and service level agreements (SLAs). In case the whole application is being migrated, this step is quite straightforward, but when only some components of the application are selected, the architecture might need to be reviewed. Special attention has to be paid to the communication between the different components, as the communication between the dedicated servers and the public cloud might need extra security, extra bandwidth, and usage of standardized protocols. When using a Service-Oriented Architecture (SOA), communication between the different modules could, for instance, make use of Simple Object Access Protocol (SOAP) or Representational State Transfer (REST) over HyperText Transfer Protocol Secure (HTTPS). Possible communication between the client layer and the components of the business logic layer should also be secured.

Figure 3 illustrates an example of the possible communication between the different components after migration to a hybrid cloud. The components are represented by server instances. Dark arrows denote communication where extra attention has to be paid regarding security and available bandwidth.


Selecting Components
Determining Provider Compatibility
Determining Impact on Client Network
Scaling the Application

(a) Cloud Migration


Decoupling Databases
Adding Tenant Configuration Database
Providing Tenant Configuration Interface
Dynamic Feature Selection
Managing Tenant Data, Users and Roles
Mitigating Security Risks

(b) Adding Multi-Tenancy

Figure 2. A summary of the different steps required to migrate an existing application to the cloud, and to add multi-tenancy to the application. (a) Cloud migration, (b) Adding multi-tenancy. The different steps are described in detail in Sections 3.1 and 3.2.

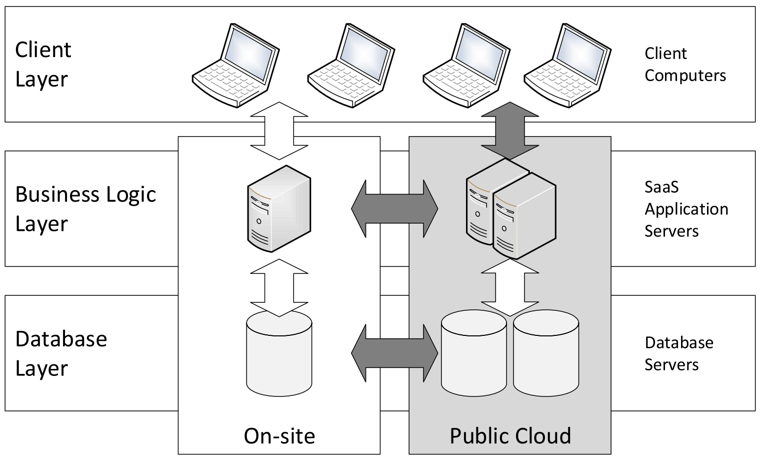


Figure 3. An illustrative example of the possible communication between components after migration to a hybrid cloud. Dark arrows denote communication that should be secured. SaaS, Software as a Service.

3.1.2. Determining provider compatibility. Apart from selecting the components for migration, some extra changes might be needed for migrating the application to the public cloud. Every public PaaS provider will typically have its own limitations and possibilities, so during the planning phase of the migration, it is best to verify that the provider will support all features of the software. In case no suitable PaaS provider can be found, an IaaS provider could also be selected to host some components of the application, but this again results in more maintenance overhead for the application provider. When comparing different providers, for example, by using CloudCmp [6], the balance should be made between the advantages of the selected provider and the overhead because of needed changes to the application. Different public cloud providers should be considered and evaluated, for example by using a small proof of concept (PoC), and the advantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

3.1.3. Determining impact on client network. A side-effect of the migration to cloud environments is that communication between some components of the software might need to pass over the Internet, especially when migrating the software to a hybrid cloud. As a result, more traffic bandwidth at the client network might be required. Before deploying the service, it is important to perform an impact analysis whenever the client configurations are changed. We have previously covered this in-depth in [18].

3.1.4. Scaling the application. When an instance is overloaded, extra instances can be added (up-scaling) and removed (down-scaling) in a few steps. This concept if often referred to as the elasticity of the (public) cloud. Some public cloud providers offer out of the box load-balancing and/or scaling, other providers only provide limited load-balancing possibilities, together with an API to support up-scaling and down-scaling from within the application.

When selecting the components to migrate, it is a good idea to take into account the scalability of the application. Components that should be highly scalable could be good candidates for migration to the public cloud, as the public cloud offers an unlimited resource pool. The application should also support decoupling of the components and handle synchronization and conflicts in data. Possible bottlenecks should be eliminated, as these could break the whole scalability of the application. Reviewing the architecture of the application to better support scalability will bring some overhead, but the advantages on the long-term will outweigh this one-time investment.

3.2. Multi-tenancy

In this subsection, the steps required to add multi-tenancy to an existing application are discussed. These steps are also summarized in Figure 2(b).

3.2.1. Decoupling databases. As multiple tenants will use the same application instance, each tenant will have its own application data stored in a shared or dedicated database instance. Using shared database instances is cheaper, while dedicated databases will lead to better performance and higher security, but at a higher cost. To connect to the correct database, a connection string is associated with each tenant. These connection strings can, for example, be stored in a shared database.

The application database needs to be decoupled, and support for multi-tenancy needs to be added to the data tables in case of shared instances. Also, the application needs to be modified to support dynamic database binding. An extra component can be added to the application, the *data access component*, responsible for both the correct handling and access control of all data requests by the application. Figure 4 illustrates a possible architecture of the application after decoupling the databases. The data access component is added to a new layer, the data access layer, situated between the business logic layer and the database layer.

For the design of the data access layer, the abstraction layer presented in [17] could be used. This abstraction layer mainly handles the security and isolation of tenant data, and the scalability of the database layer. In our approach, we partition tenant data over multiple database instances based on the tenant. By doing so, tenants can still store their data in a dedicated on-site database instance, for example, to comply with regulatory policies on data. Partitioning the data based on the tenant also provides a clear separation of tenant data. For large tenants with a dedicated database instance, the performance of the database will not be influenced by other tenants. As the number of tenants using a shared database instance will be limited, the possible damage due to an information leakage is also minimized. Different SLAs can be provided, based on the scenario of a dedicated or shared database instance.

3.2.2. Adding tenant configuration database. A new database, which we refer to as the *tenant configuration database*, needs to be added to store general information about all tenants. The connection strings introduced in the previous steps will be stored in this database, together with specific information and configuration parameters such as billing and contact information, and the selection of features for the tenant as described in Section 3.2.4. While this database is shared between all tenants, it only contains minimal information and is only accessed sporadically as the information inside this database can be cached by the application, so it should not become a bottleneck [16, 18].

3.2.3. Providing tenant configuration interface. Adding multi-tenancy to the application makes it possible to more flexibly select the application features used by different clients, as the tenant

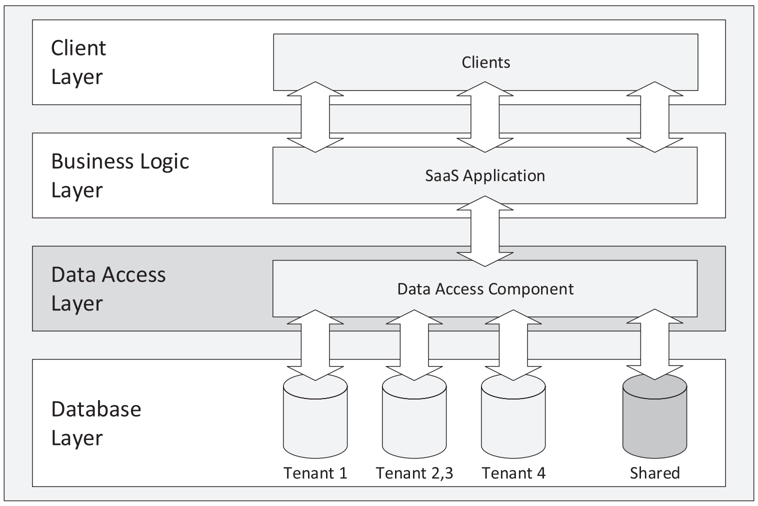


Figure 4. Possible architecture of the application after decoupling the databases.

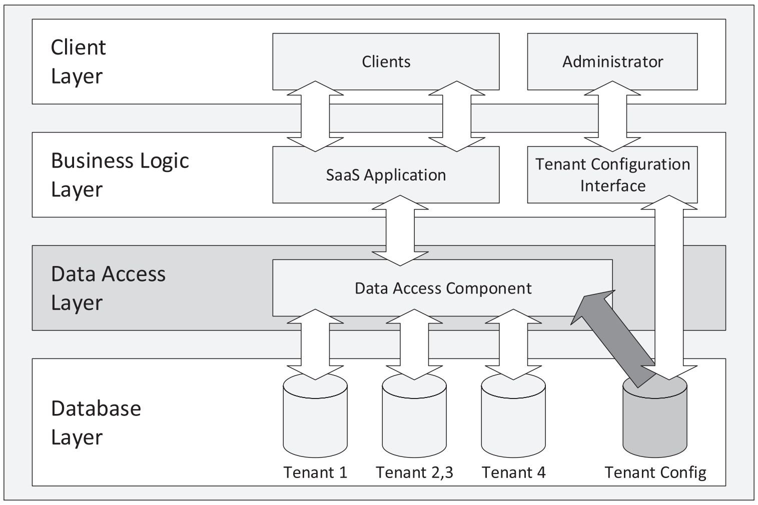


Figure 5. Possible architecture of the application after adding the *tenant configuration interface*. SaaS, Software as a Service.

configuration is stored in the shared *tenant configuration database*. It is, however, also necessary to create a separate application, the *tenant configuration interface*, which can be used by tenant administrators to modify the tenant configuration. This interface will be used to create, modify, and delete tenants in an easy way and change the configuration of a single tenant, for example the selection and configuration of features and the connection string of the tenant.

Ideally, the *tenant configuration interface* is the only component that has read/write access to the *tenant configuration database*, as the legacy application should only require read access. Figure 5 illustrates the changes to the architecture after adding this interface.

3.2.4. Dynamic feature selection. An application can have multiple features that will be dynamically loaded at start-up. As every tenant can have its own selection of features, and a tenant-specific configuration for these features, a tenant administrator should be able to select and configure these features using the *tenant configuration interface* introduced before.

The application itself needs to support the dynamic selection of features. For example, some features might require additional modules, and the application needs to support dynamic loading (and unloading) of the corresponding modules. Also, the user interface of the application might need to be automatically adapted for the different tenants, based on their configuration, representing the tenant's feature selection. The different features might run on the same instance of the application, or on dedicated machines. In the latter case, feature placement algorithms can be used to determine the optimal solution. We have previously covered this in [11].

3.2.5. Managing tenant data, users, and roles. Every tenant using the application will typically have its own data, users, and custom roles. The users could be stored in a shared common database or in the tenant database, or the application could support external identity providers. In case the users are stored in a global shared database, or when an external identity provider is used, the application could provide single sign-on scenarios. In case the users are stored in the tenant database, an administrator should be able to create and modify users and their corresponding roles from within the application. By introducing multi-tenancy, a tenant administrator role with permissions to create and modify the tenant configuration using the *tenant configuration interface* is required, different from the administration roles within a single tenant. These tenant administrators can be stored in the shared *tenant configuration database* and should have limited access to the multi-tenant application for every tenant if required. The management of users and roles could be moved to the *tenant* *configuration interface*, or could stay inside the application, depending on the application's requirement and the software license model. The question arises on how and where to store the tenant data and the different users and roles. Different approaches are possible, and we have previously covered this in-depth in [16].

3.2.6. Mitigating security risks. A major disadvantage of using multi-tenancy is an increased security risk, as by definition, multiple tenants will use the same application instance. These risks can be mitigated in multiple ways:

- **Implementing URL-based filtering of application requests, taking into account the permissions of the user and tenant.** Every tenant can have its own URL, for example by having a customized sub-domain. When a client wants to access the data of a specific tenant, the access module of the application needs to verify if the authenticated user and its corresponding tenant have an access to the requested data (the requested URL), to eliminate unauthorized access.
- **Separating the tenant configuration from tenant data.** Because the tenant data is stored in a different database instance as the tenant configuration, it is easier to configure tenant-specific access at the database level. Each tenant will have its own connection string, and the associated credentials will only have access to the tenant's database.
- **Offering single-tenant instances of specific components at a higher cost.** If the aforementioned methods are not deemed sufficient, tenants with a huge amount of confidential data can have single-tenant instances at a higher cost. Having a dedicated instance clearly improves security, as the tenant's data is not only virtual but also physically isolated from other tenants. Because the connection strings are stored separately for each tenant in the shared *tenant configuration database*, these connection strings can either point to a shared or dedicated database.

4. CASE STUDY: MEDICAL COMMUNICATIONS SYSTEM

4.1. Introduction

In this section, we verify our presented approach using the case study of a Medical Communications (MC) system. The MC system is responsible for the correct functioning of all communication peripherals located in a medical environment. The central functionality of this system is the *nurse call* system. The basic concept of a nurse call system is simple: A call device is located in every room. When a button is pressed on the device, a message is sent to a controller after which nurses are notified of the call. This concept can be enhanced by using ontologies and semantic reasoning to identify the urgency of a call or select the nurses to notify in a more intelligent way [19–21].

A nurse call system consists of many different elements, installed within a hospital. These elements include amongst others the following: (ii) end user equipment installed in the rooms, which patients can use to contact hospital personnel, and terminals used by the personnel; (ii) embedded servers, used to communicate between the terminals and management servers; and (iii) servers for logging, registration, and visualization. Figure 6 illustrates an example of the architecture of a nurse call system, with the possible communication between the different components when a patient calls a nurse (shown in arrows).

While the center of the MC application is the nurse call system, additional services, such as intercom, video over IP, access control, and other health services are being offered as well. Currently, the MC system is installed in multiple locations, ranging from big hospitals to small rest houses. The cost of installing dedicated servers, and the corresponding maintenance, is quite high. Migrating a part of the system to the cloud will minimize the cost, by eliminating the need of many of the dedicated servers, making it possible for smaller hospitals and rest houses to afford the system. However, considering the medical use case, the MC application is subject to stringent security and performance constraints, which need to be taken into account when the components to migrate to the cloud are selected.

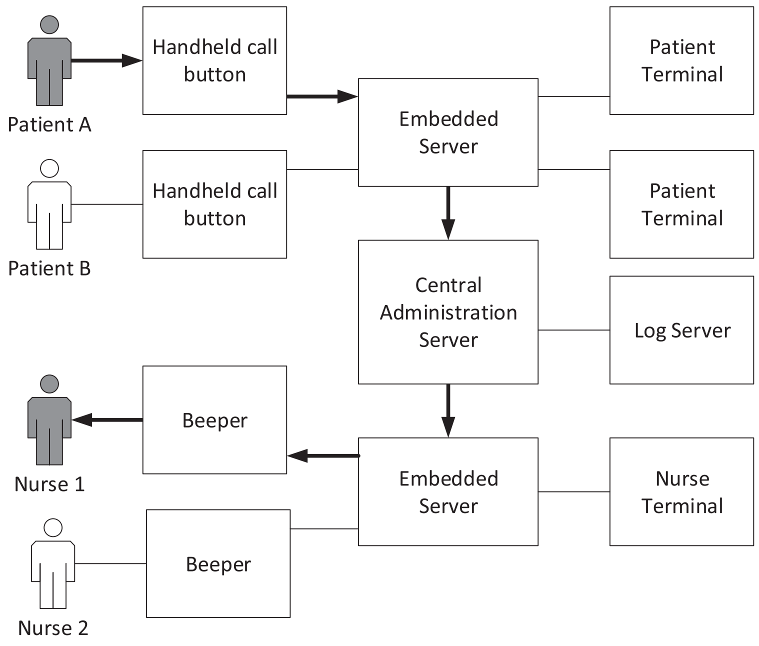


Figure 6. An example architecture of a nurse call system, with the communication between the different elements when a patient makes a call.

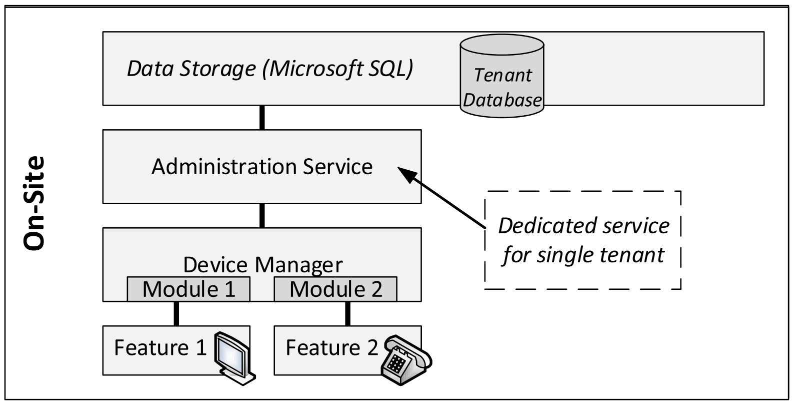
4.2. Cloud migration

4.2.1. Selecting components. The MC software consists of two main components: the *device manager* and the *administration service*. The *administration service* is the main application and is used to manage the different features and devices installed within the hospital. The *device manager* is a dedicated hardware box, running different modules mainly written in C++ for communicating with the different peripherals installed within the medical environment. The modules for the different features are dynamically loaded on start-up and can be configured from the administration service. Figure 7(a) shows the initial architecture of the application. The device manager and the different peripherals communicate over Ethernet, using a custom proprietary secure protocol.

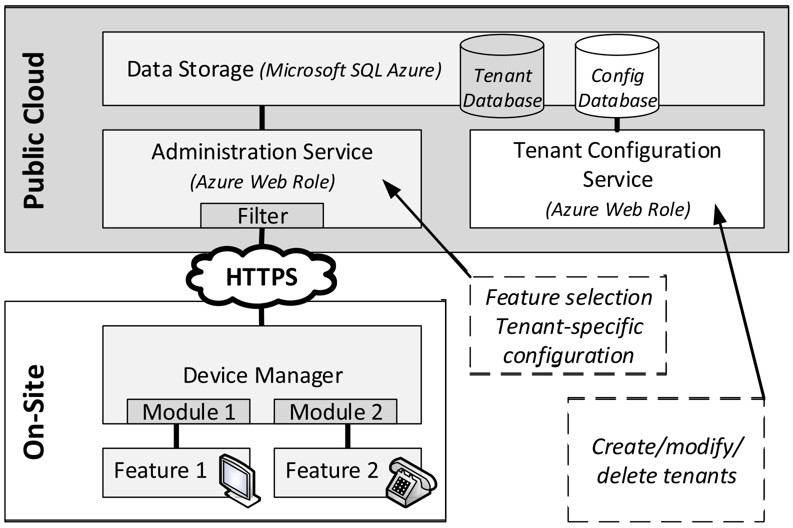
For our PoC, we selected the administration service and its corresponding database instances for migration to the public cloud. The MC system has a fallback mechanism, allowing the device manager to operate standalone in case the administration service is not available. Because the device manager communicates directly with the devices, migrating this component to the cloud would be tricky, as the devices need to operate when no connection to the public cloud is available. Passing all communication between the device manager and the peripherals over the public Internet would also result in slow response times and could make the system unreliable. However, as most of the processing is done in the devices, a single device manager will be sufficient to control all devices in a small or medium environment. If the peripherals could be adjusted to work standalone when the device manager is unavailable, migrating the device manager could also be an option in the future, but for this PoC, we started focusing only on decoupling the administration service.

After adding multi-tenancy to the application and migration to the cloud, a new component is introduced, the *tenant configuration interface*. The reviewed architecture after adding multi-tenancy and migration is shown in Figure 7(b). Because every tenant has its own features, the user interface of the administration service is automatically adapted for the different clients based on the tenant configuration.

4.2.2. Determining provider compatibility. As the application is written in .NET, migrating the administration service to Microsoft Azure seemed like an evident choice. Microsoft Azure [22]



(a) Initial architecture of the software before moving to the cloud.


Public Cloud
Data Storage (Microsoft SQL Azure)
Tenant Database
Config Database
Administration Service (Azure Web Role)
Filter
Tenant Configuration Service (Azure Web Role)
HTTPS
Feature selection Tenant-specific configuration
Create/modify/ delete tenants
On-Site
Device Manager
Module 1
Module 2
Feature 1
Feature 2

(b) Architecture after migration to the cloud and adding support for multi-tenancy.

Figure 7. Architecture of the application before and after migration to the cloud and adding support for multi-tenancy. (a) Initial architecture of the software before moving to the cloud, (b) Architecture after migration to the cloud and adding support for multi-tenancy.

currently offers two roles to choose from when creating an instance, web roles and worker roles, both based on Windows Server. The main difference between these two is that an instance of a web role runs IIS, while an instance of a worker role does not. In addition to the type of instances, Azure offers different sizes for both roles [23]. Table I gives an overview of the different standard instances available on Azure.

Both the administration service and the *tenant configuration interface* will be running on an Azure Web Role. While preparing the application for migration, these Azure Web Roles need to be added to the .NET project and can be tested in the Azure simulator. When using a third party assembly in the project, this assembly should be added as a reference to the project, with the Copy Local property set to true. A nice side-effect of this process is that many deprecated libraries were removed or replaced in the project, making it much easier for developers to locally install the application, as they no longer needed to configure and install third-party products on a clean environment before being able to compile and test the application.

The relational databases will be moved to SQL Azure. As a result, the connection strings inside the application should be altered to point to the SQL Azure instance. SQL Azure has some limitations regarding a dedicated Microsoft SQL Server, but for most .NET applications, this should not

Table I. Overview of standard instances on Windows Azure.

Name	Virtual Cores	Ram
Extra small (A0)	Shared	768 MB
Small (A1)	1	1.75 GB
Medium (A2)	2	3.5 GB
Large (A3)	4	7 GB
Extra large (A4)	8	14 GB

be an issue. Once the application is running correctly in the Azure simulator, the project can be packaged and deployed onto Windows Azure [24, 25].

4.2.3. Determining impact on client network. The traffic between the administration service and the device manager now has to pass the public Internet, and the internal network is also loaded with traffic between the device manager and the different peripherals. The total amount of traffic is depending on the selection of features, as some of the features might require more bandwidth. Both the internal network and the public Internet connection need to have sufficient bandwidth to support the MC system to operate. The service described in [18] was customized to support this PoC, making it possible to predict if a custom selection of features would be able to run on the client network. For this PoC, the different topologies of the client networks were implemented statically, but we introduced the option to easily replace these static topologies by a dynamically generated topology, which could be generated by tools using existing network discovery protocols, such as Neighbor Discovery Protocol and Link Layer Discovery Protocol.

4.2.4. Scaling the application. Azure allows the administrator to configure multiple instances with automatic load balancing, which will be required as the number of tenants grow. Recently, limited possibilities were added to Azure for automatic scaling, using the *Autoscaling Application Block* [26]. Alternatively, the creation and deletion of extra instances can be done manually (or in code) by the customer. Some third party products also exist, like AzureWatch [27], which will handle the scaling automatically, or the SaaS provider can create a customized system, for example by using advanced load prediction.

4.3. Multi-tenancy

4.3.1. Decoupling databases. In the initial single-tenant architecture, there is a dedicated relational database for every instance. The connection string to this database is hard-coded in the configuration file (Web.config). To support multi-tenancy, we introduced dynamic connection strings, stored in the *Tenant Configuration Database*. The connection string in the configuration file was replaced by a connection string to this shared database.

To support both shared and dedicated databases, we added an extra column in the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application itself does not need to know if the database is shared or dedicated, as multiple tenants can share the same connection string. The Data Access Component introduced in Section 3 is now responsible to select the correct tenant's data, for example by filtering on the corresponding *tenantID*.

4.3.2. Adding tenant configuration database. The *Tenant Configuration Database* is introduced to store the general information about the different tenants. It holds the connection strings for each tenant, together with some contact and billing information and the feature selection for the tenant. As the administration service only needs to get this information at start-up, read-only access to this database is sufficient for the main application. This also eliminates the risk of tenants modifying the configuration of other tenants. Figure 8 illustrates this by giving an overview of the possible communication between actors and components within the system.

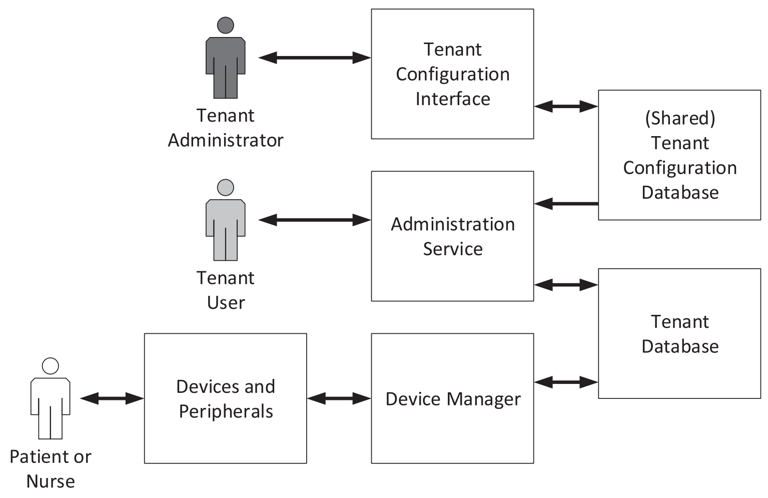


Figure 8. An overview of the possible communication between actors and the different components of the medical communications system.

4.3.3. Providing tenant configuration interface. A new application is introduced, the *Tenant Configuration Interface*, used by tenant administrators (like resellers or the application provider) to setup and configure the different tenants. This application has write access to the tenant configuration database, but as only tenant administrators have access to this application, there is no risk of tenants modifying the configuration of other tenants, or even their own configuration, making their system unusable. For this PoC, we did not spend too much time to build a full-blown interface, but in the final version, enough time should be spent building this application, as it is a key component in the multi-tenant application that can dramatically minimize the time needed to configure and modify new or existing tenants. The *tenant configuration interface* was designed as a web application running on an Azure Web Role, but to mitigate security risks, this interface could also be developed as an internal mobile or desktop application, accessing the *tenant configuration database* through web services.

4.3.4. Dynamic feature selection. The nurse call feature is the core feature of our MC system, but some other features are also implemented, for example, voice and video calling between different rooms using Voice over Internet protocol, and door access control with badges used by the hospital personnel. The selection of features for a single tenant depends on the available hardware and peripherals within the hospital, and the available bandwidth of both the internal and external network. The selection of features and general/technical configuration is done by a tenant administrator through the *tenant configuration interface*, while the tenant-specific configuration of the features can be done by different tenant users through the administration service. The initial application (administration service) was designed to support dynamic loading of the required libraries and modules at start-up. The modules kept running during the lifetime of the application, but as this application was installed on a dedicated instance with a lot of available resources, this was not really an issue. Converting the application to a multi-tenant application however introduced some new challenges. As every tenant can have its own selection of features, all features might need to be loaded on the single machine, and if the multi-tenant application is not well designed, some features might even be loaded multiple times. To overcome this issue, some changes are needed to the application:

- The required libraries and modules for a specific tenant are loaded as soon as a user logs in to the administration service.
- Libraries and modules should be loaded only once, and hence, can be shared between different tenants.

- Loaded libraries and modules should be freed as soon as they are not used anymore, for example after a timeout, to eliminate the usage of unnecessary resources.

4.3.5. Managing tenant data, users, and roles. As already indicated in Figure 8, there are a different users and roles used in the MC system:

- The tenant administrators (application provider, resellers, and installers), having access to the *tenant configuration interface*. These users and their corresponding roles are stored in the *tenant configuration database*.
- The tenant users and their corresponding roles (mostly personnel of the different hospitals). Because every tenant can have its own users and roles, these are stored in the tenant database.
- The patients do not really require roles, but are in way guest users of the system. The peripherals however could count as visualized users with customized roles, and can also be stored in the tenant database, together with the tenant users and roles.

4.3.6. Mitigating security risks. Some of the security risks and a way to eliminate these risks are already described in the previous steps. To increase the security, we added URL-filtering to the application and altered the access module to take into account the requested URL (and hence, the identification of the specific tenant) and the authorized user and its corresponding tenant ID. The traffic between the device manager and the administration service and the tenant database now passes the public Internet and is secured by using HTTPS over Secure Sockets Layer/Transport Layer Security. Every tenant can have a dedicated tenant database, increasing the isolation of data, but this comes at a higher cost. In practice, big hospitals will typically have a dedicated database, and data from smaller rest houses belonging to the same entity (subtenants of the same tenant) will be co-located in shared databases. This way, we will not be mixing data from subtenants belonging to different tenants, and isolation of data is always guaranteed at tenant level.

5. CASE STUDY: MEDICAL APPOINTMENTS SCHEDULE PLANNER

5.1. Introduction

As a second case study, we migrated a medical appointments schedule planner to public cloud environments. This planner is used by both patients and medical staff to manage their appointments. The software was originally developed as a single-tenant application. Figure 9 illustrates the original layered architecture of the application. The end-users (patients) access the web application through the user portal, in order to manage their medical appointments. The application is running on a shared web server; the appointments and patient data are stored in a dedicated database on a shared database server. Medical staff accesses the application through the admin portal in order to approve and review the requested appointments.

As multiple clients started using the software, multiple independent copies of the software were installed and configured, running different versions, increasing maintenance complexity. Independent copies were deployed on the same shared web server and the average load increased over time, resulting in an increase in page load times due to the large amount of data and the required amount of data processing by the application. For this case study, we added multi-tenancy to the application to optimize the utilization of available hardware resources, and migrated the application to the public cloud environment in order to centralize the management and to increase the scalability. We deployed the application on two different cloud providers in order to compare the performance and the ease of deployment.

5.2. Cloud migration

5.2.1. Selecting components. The schedule planner consists of two main components: the user portal, used by patients to request medical appointments, and the admin portal, used by medical staff to approve and review the requested appointments and to manage their schedule. Both patients

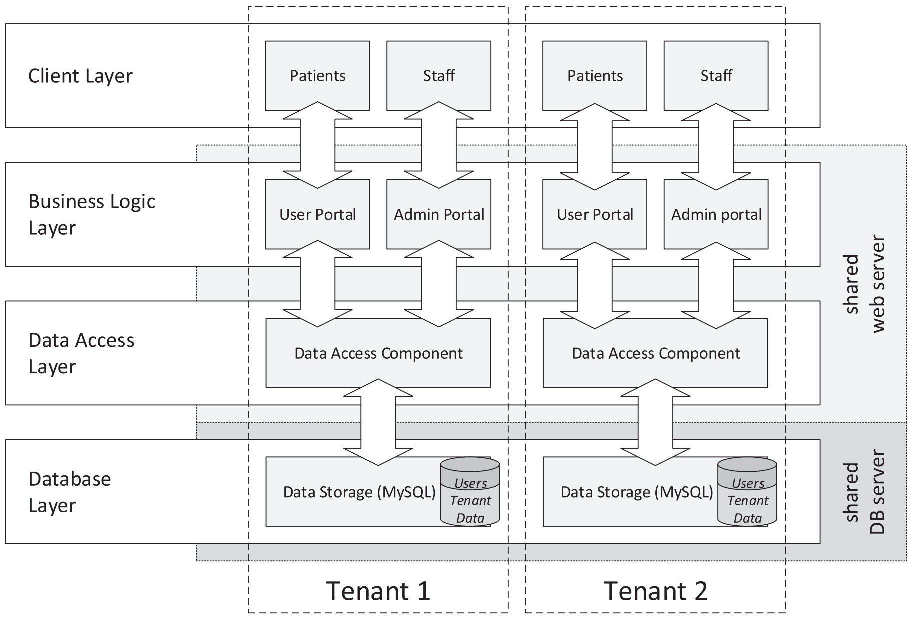


Figure 9. Pre-migration single-tenant architecture of the medical appointments schedule planner.

and medical staff can synchronize their appointments to their personal calendar using one of the available standard calendar formats, and confirmations and reminders are sent by email or by text message (SMS). Different departments are using this portal, but a department may only access patient information relevant for their appointments. Patients on the other hand can browse and request appointments at the different departments.

For this second case study, we selected the whole application for migration to a public cloud provider. This application is less sensitive for short downtime periods as the MC application from the previous case study, as both patients and medical staff have offline copies of their appointments. Therefore, no additional fallback mechanism is necessary inside the application.

The legacy application was developed to be used by a single medical department or an independent doctor (a single tenant), and independent copies of the software were installed and configured. After adding multi-tenancy to the legacy application, a single instance of the application is now shared between multiple tenants, and a new component is introduced, the *tenant configuration interface*, with a similar functionality as the interface from the previous case study.

5.2.2. Determining provider compatibility. The legacy web application is developed using HTML5, Hypertext Preprocessor (PHP), and Oracle MySQL for the persistent storage of data and is executed on a shared web server. For evaluating our approach, we migrated the application to both PaaS and IaaS environment. We selected Google AppEngine [28] as PaaS provider as they provide a PHP Runtime Environment, and Amazon EC2 [29] as IaaS provider. Google AppEngine requires more changes to the legacy application as it puts more constraints on applications, while Amazon EC2 requires more maintenance as they provide full control over the virtual machine.

Migrating an application to Amazon EC2 is straightforward. Amazon currently offers two types of EC2 instances, the T2 instances, which are burstable performance instances for development environments and early product experiments, and the M3 instances, which provide a good balance of compute, memory, and network resources. The different types of T2 and M3 instances currently available are listed in Tables II and III, respectively. For our PoC, we selected a t2.micro instance running Ubuntu Server 14.04 (Ubuntu, London, UK) for both the database and web server. We configured Apache and MySQL on the instance and deployed the application, and except from some configuration settings, no changes were required in the application.

Table II. Overview of the available T2 instance types on Amazon EC2.

Model	Virtual Cores	Ram
t2.micro	1	1 GB
t2.small	1	2 GB
t2.medium	2	4 GB

Table III. Overview of the available M3 instance types on Amazon EC2.

Model	Virtual Cores	Ram
m3.medium	1	3.75 GB
m3.large	2	7.5 GB
m3.xlarge	4	15 GB
m3.2xlarge	8	30 GB

Table IV. Overview of the most important changes for migration to Google AppEngine.

Item	Description
Relational data	Migrate MySQL databases to Google Cloud SQL and modify connection strings.
Temporary files	Replace local file storage by storage buckets on Google Cloud Storage.
URL rewriting	Replace mod_rewrite by a custom PHP script providing similar functionality.

Migrating the legacy PHP application to Google AppEngine on the other hand required multiple changes to the application as summarized in Table IV. First of all, as Google offers Cloud SQL instead of MySQL, some changes are required in the application to connect to the Cloud SQL database instance [30]. The original MySQL database can be exported to a file using an SQL dump, and this file can be used to import the data into a new Cloud SQL database instance. The MySQLi extension introduced with PHP version 5.0.0 can still be used to connect to the database, but the connection string differs from a traditional connection string as illustrated in [30].

In AppEngine, the local file system that the application is deployed to is not writable. However, if the application needs to write and read files at runtime, AppEngine provides a built-in Google Cloud Storage (GCS) stream wrapper that allows many of the standard PHP file system functions. A PHP application running on AppEngine can read and write files by using buckets as illustrated in [31]. The legacy application was developed using the Smarty PHP Template engine [32], which requires different physical file directories for reading and writing templates and configuration files. As a result, the Smarty engine needs to be reconfigured to use the GCS for storing the compiled templates and files. One major difference between writing to a local disk and writing to GCS is that GCS does not support modifying or appending to a file after closing it. Instead, a new file can be created with the same name, which overwrites the original. For the Smarty PHP Template engine, however, this is not really an issue as it only creates temporary files that are not modified after creation. Using buckets to store temporary files can have an influence on the performance of the application. There is no straightforward way to measure this impact, as local file storage is not supported by Google AppEngine. In Section 6, we however do compare the performance of the application running on Google AppEngine with other environments that are using traditional file storage.

Finally, the legacy application implemented URL rewriting by invoking Apache's mod_rewrite module. As Google AppEngine does not support this module, this functionality has to be simulated through the use of a PHP script referenced from the application's configuration file (app.yaml) that will in turn load the desired script, as described in [33]. The overhead introduced by this script is minimal, as it just parses the requested Uniform Resource Identifier and executes a simple conditional statement. Google however recommends to rewrite the application to operate without the mod_rewrite module, but this requires more effort as more changes to the source code are required.

Once the application is running correctly in the simulated environment of Google App Engine Launcher (part of the Google App Engine SDK), it can be deployed onto the public cloud.

5.2.3. Determining impact on client network. As the original application was already designed to be accessed over the web, and the full application is migrated to the cloud, there is no real impact on the client network after migration to the public cloud.

5.2.4. Scaling the application. Amazon offers CloudWatch [34] to monitor Amazon Web Services cloud resources and applications. This service provides a clear insight in the current demand using different metrics such as CPU utilization, data transfers, and disk usage activity. Application developers can also create custom metrics, and customize automated actions and alarms. For a production-ready application on Amazon EC2, CloudWatch can be used to provide compliance with specific SLA targets, and to handle the automated scaling of both the computational resources.

Google AppEngine on the other hand has built-in support for high scalability. An application running on AppEngine can be deployed on multiple instances, and instances are automatically created or removed depending on the current load. No action is required from the developer, but the developer has more limited control than with Amazon EC2. During our experiments as described in Section 6, multiple instances were automatically created.

5.3. Multi-tenancy

After adding multi-tenancy to the application, the original architecture was slightly modified. Figure 10 illustrates the modified architecture after adding multi-tenancy and migration to a public cloud provider. These modifications are discussed in detail in the remainder of this section.

5.3.1. Decoupling databases. In the initial single-tenant architecture, every tenant has a dedicated MySQL database on a shared database server. In order to support multi-tenancy, we introduced dynamic connection strings, as in the previous case study. All users are however stored in a single database, separated from the tenant databases. By doing so, a single user can access multiple tenants, and multiple copies of the same user object are eliminated.

As with the previous case study, we added an extra column to the data tables, holding the identification of the tenant (*tenantID*). By doing so, the application supports both shared and dedicated database instances, and multiple tenants can share a single database. The data access component

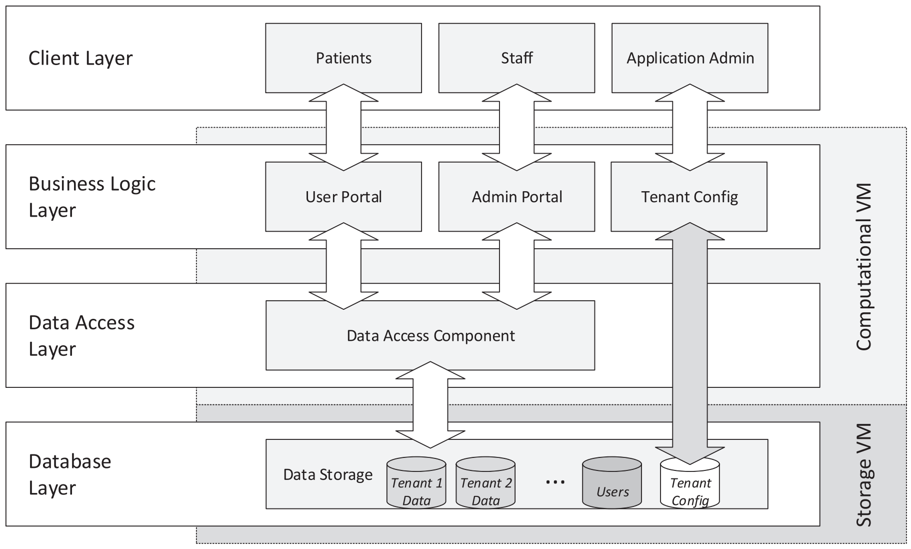


Figure 10. Revised architecture of the schedule planner after adding multi-tenancy to the application and migration to the public cloud.

of the data access layer was modified to support the dynamic behavior of the tenant databases, and to filter tenant data based on the *tenantID*. This filtering is required in order to provide transparent isolation of tenant data, especially when multiple tenants are sharing a single database instance.

5.3.2. Adding tenant configuration database. The shared *Tenant Configuration Database* contains general information about the different tenants, and a connection string to the database instance where the tenant data is stored. This database is small in size, which is why it is also used to store the different user objects. However, should this database ever become a bottleneck, the user data can easily be decoupled from the general tenant information, as separate connection strings are used for the user database and the *tenant configuration database*. For our PoC, these connection strings refer to the same *tenant configuration database* instance.

5.3.3. Providing tenant configuration interface. A *tenant configuration interface* was added to the application, used to manage the different tenants. As in the previous case study, this configuration interface communicates directly with the *tenant configuration database*.

5.3.4. Dynamic feature selection. Tenants can have optional features enabled, for example notifications by text messages or export options to different calendar formats. A tenant administrator can configure these features through the *tenant configuration interface*. The feature configuration is stored in the *tenant configuration database* together with the general tenant information, and both the application's user portal and admin portal take the feature selection of the selected tenant into account.

5.3.5. Managing tenant data, users, and roles. The user objects are stored in the shared *tenant configuration database*; the roles are stored together with the tenant data in a tenant database instance. This instance can be either a dedicated database instance or an instance that is shared between multiple tenants. Relevant medical information belonging to a certain patient is stored together with the role in the tenant database. By doing so, sensitive patient data are inaccessible by other tenants, as all data queries are filtered based on the *tenantID* by the data access component.

By using a single database instance to store all user objects, multiple roles for different tenants can be created for a single user object. This allows for a single sign-on, where the user object is loaded when the user logs in the application, and the relevant roles are loaded when the user wants to access one of the tenant's restricted pages.

5.3.6. Mitigating security risks. As mentioned previously, sensitive data belonging to a certain patient are stored together with the user role in the tenant database. As all data queries are filtered by the data access object based on the *tenantID*, queries can never return data belonging to different tenants.

Communication between a client computer and the web server is encrypted, as all communication uses HTTPS over SSL. This was already the case with the legacy application.

6. DISCUSSION AND EVALUATION

Moving applications to the cloud and adding multi-tenancy introduces new opportunities for our presented use cases. First of all, there is the increased flexibility and elasticity. When the workload on the application increases, new instances can be created and deployed automatically. Similarly, when the workload decreases, instances can again be removed. For new customers, deployment times decrease as there is no need to physically install a new server. By using a PaaS platform instead of IaaS, there is no need to install, configure, and manage the guest OS, further reducing the deployment times. The hardware maintenance cost is also eliminated as the virtual machines running in the cloud are automatically migrated when the hardware fails. Combining multi-tenancy and migration to a public cloud makes maintenance easier, as the software is deployed centrally, and no on-site intervention is needed, for example to install patches or updates. Adding multi-tenancy to the application also improves the efficiency of resource utilization, decreasing the costs, and eliminates the need for installing and configuring independent copies of the same software, sometimes running different versions of the software. In this section, we will highlight some of the major advantages of the migration, and compare them with the overhead of the migration.

6.1. Increased flexibility and elasticity

As the amount of available resources in the cloud is quasi unlimited, application developers do not have to worry about selecting the right amount of resources to host the software. Novel multi-tenant applications can start with a single instance, and the number of instances can grow as the workload increases. In cloud computing, elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time, the available resources match the current demand as closely as possible. In our approach, we have presented some possibilities for building elastic applications in the step of *Scaling the Application*. For the two case studies, we also started with a single instance, and determined the possibilities to scale the application as the demand grows.

6.2. Decreased deployment time

The addition of multi-tenancy to the application and migration to a public cloud yields a significant decrease in deployment times, especially for new tenants. Tables V and VI illustrate this for the MC software case study by giving an estimation of the needed deployment time for a new tenant, respectively, before and after the migration process.

Before migration, a physical server was installed and configured on-site for every new tenant, together with the device manager and peripherals. A local copy of the administration service was installed and configured on the dedicated physical server together with an SQL Server instance. A total of 6 man-days was required to perform the installation and configuration of both the server and the device manager and peripherals.

After migration, the initial configuration time is largely reduced. Only the device manager and peripherals need to be installed, and the configuration of the device manager can be done remotely by using the multi-tenant administration service running on the cloud. As only a new tenant needs to be created, no local copy of the administration service needs to be installed and configured. An

Table V. Initial configuration of new tenant before migration: time estimation for the MC software case study.

Task	Time
Install and configure on-site server for application	1 day
Deploy administration service on on-site server	1 day
Configure SQL server and initial tenant database	1 day
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

MC, Medical Communications; SQL, Structured Query Language.

Table VI. Initial configuration of new tenant after migration: time estimation for the Medical Communications (MC) software case study.

Task	Time
Create new tenant and initial tenant database	1 hour
Impact analysis on client network (automated)	1 hour
Configure on-site hardware (device manager and peripherals)	2 days
Verification and testing	1 day

Table VII. Initial deployment after migration: time estimation for the MC software case study.

Task	Time
Deploy administration service on Azure	2 hours
Deploy *tenant configuration interface* on Azure	2 hours
Create initial databases on SQL Azure	2 hours
Create tenant administrators	2 hours

MC, Medical Communications; SQL, Structured Query Language.

estimated total of 3.5 man-days are required for the initial setup after migration, mainly for the installation and configuration of the on-site device manager and the peripherals, and for full testing.

Migrating the device manager to the cloud could further reduce the deployment time, but this however introduces additional challenges that were mentioned before in Section 4.2.1. For completeness, Table VII shows an estimation of the initial deployment of the application on Microsoft Azure. This initial deployment needs to be done only once, and not for every new tenant.

6.3. *Ease of maintenance*

Having the core of the MC software, the administration service hosted in the public cloud makes maintenance a lot easier, as installers no longer need to go on-site to make small configuration changes. Eventually, one could argue that having Virtual Private Network (VPN) connections to the customer sites could also bypass this, but this requires a VPN setup to the hospitals, or public access to the internal network, which again introduces some security risks, together with a stable external connection at the client side. Having a single multi-tenant application also has the advantage that every tenant uses the same version of the software, and software updates can be deployed centrally, for all tenants at once. For installing software updates, a second instance can be deployed and configured in an isolated environment on the public cloud, and switched with the current instance once the configuration and testing is done. Software updates and patches for the device managers can be pushed from the central administration service, as under normal circumstances, the device manager has a persistent connection to the administration service and will frequently check for updates.

For our second case study, the schedule planner, deploying the multi-tenant application on the public cloud results in similar benefits. Before adding multi-tenancy to the application, different independent versions were deployed, and updating and maintaining the application became harder as the number of tenants grew. After migration to the cloud, only one copy of the multi-tenant application is running on multiple instances in the cloud, and all tenants are running the latest software of the application.

6.4. *Migration cost*

Migrating software to the cloud and adding multi-tenancy to the application come at a cost. The architecture and code might need to be changed, and the software needs to be tested thoroughly. Extra attention needs to be paid to the security aspect, as the whole application or some components are now hosted remotely. For the MC software, we spent a total of six man-months to implement the changes described for in Section 4. For a production-ready application in the cloud, an estimated additional 14 man-months would be required, as summarized in Table VIII. The remaining tasks mainly focus on adapting the cloud application to support the existing SLAs, investigating possibilities for automatic backup and restore, providing training for installers and full testing.

For the schedule planner, our second case study, only two man-months were required to implement the changes described in Section 5, as this application is less complex than the MC software. Adding multi-tenancy to the legacy application required a bit less than two man-months. For the migration to Amazon EC2 only 1 day was sufficient, whereas for Google AppEngine, a few man-days were necessary to implement the required changes. For a production-ready application in the

Table VIII. Tasks to be done for a production-ready application in the cloud: time estimation for the MC software case study.

Task	Time
Azure-specific tasks	three man-months
Explore monitoring options	
Define backup and restore strategies	
Verify SLA constraints on Azure	
Adapt cost model	
Administration service	eight man-months
Improve security of service and features	
Make complete mapping for all features	
Migrate remaining features to Azure	
Add support for newest hardware nodes	
Study impact of shared versus dedicated database instances	
Other remaining issues	
Impact analyzer	one man-month
Support dynamic generated topologies	
Other tasks	two man-months
Training and development courses for developers	
Training for installers, retailers, and clients	
Full testing of application	

MC, Medical Communications; SLA, Service Level Agreements.

cloud, an estimated additional four man-months would be required, for finishing the application and full testing.

6.5. *Remaining risks*

As some components are now hosted on a public cloud, there is an increased security risk. For our first case study, the MC software, extra attention should be paid to the new risks introduced by moving the software to the public cloud. A side-effect of the migration is that the Administration Service is now hosted in the public cloud, and could become a bottleneck if the number of tenants grows significantly. Therefore, extra attention should also be paid to the scalability of this service. For our second case study, the main security risk is due to the addition of multi-tenancy, so the application needs to guarantee isolation of sensitive data, for example by restricting all queries to filter data based on the *tenantID*.

6.6. *Change in cost model*

Migrating the software to the cloud brings a change in the cost-model of the MC software. Before the migration, the software and hardware were typically sold as a single package, including the necessary hardware, licenses for the software, initial installation, and configuration. As most public providers work with monthly fees, the application provider should adapt the cost-model to reflect this cost model. Instead of selling a one-time license for the software, the end users can pay a monthly fee, depending on the size of the tenant, covering the hosting on Azure and future software changes and updates. The *Tenant Configuration Interface* can be designed to support this cost-model, and could be linked to the financial software. This change in cost model introduces a new opportunity by expanding the customer market, as smaller clients (for example small rest houses) are now able to start using the software at a lower cost, as the costs of implementing the system can now easier be spread over time, less hardware is required on-premise, and computational resources are shared between multiple tenants.

The schedule planner software was already being sold using a license-based cost model. Adding multi-tenancy and migrating the application to the cloud introduce no visible changes in cost model.

Sales prices could however drop as utilization of available resources is optimized by adding multi-tenancy to the application, and the infrastructure cost is reduced by using a public cloud provider.

6.7. Performance comparison

The performance of an application running on the cloud depends on both the selected cloud provider and the selected instance type. We selected the second case study, the medical appointments schedule planner, for evaluating the performance of the selected cloud providers, as this application is fully migrated to the public cloud, whereas the MC application is only partially migrated, making the performance depends on more factors such as the on-site client network topology and capacity and on-site available hardware.

In order to evaluate the performance of the medical appointments schedule planner, we deployed the application to four different environments, as summarized in Table IX. We measured the average page generation time; this is the time needed for the PHP interpreter to generate the page, for different pages of the application. This metric does not take into account the network latency, as the generation time is measured at the server side by the PHP interpreter itself. We also measured the end-to-end transaction time; this is the total load time as perceived by the client, as this metric does include the network latency. The schedule planner application was configured with a custom database based on data from existing production databases, combining historical data from different tenants over the last 3 years. We selected three specific pages for this experiment. The first two pages perform complex merge operations on the tenant data, as these pages were reported by existing users as being too slow. The third page is a normal page with an average load time. The experiments were executed on the cloud platforms in January 2015. Table X provides the measured average page generation times together with the standard deviations for the selected test pages over 50 iterations, and Figure 11 illustrates the same results graphically. Table XI and Figure 12 are similar, but for the end-to-end transaction times.

As can be seen from these results, the Amazon EC2 Engine provides a good performance, as it is even faster than the local VM, even though we used the lightest instance available, a t2.micro instance. The Google AppEngine on the other hand is rather slow, as it takes up to 7 seconds to

Table IX. An overview of the different deployment environments. The mentioned cost values are valid at the time of submission of this article.

Label	Description	Estimated Cost
Local	A dedicated virtual machine with 1 GB Ram and 1vCPU, running on a physical linux server with a Linux server with an Intel Core i7 CPU (2.80 GHz) with 8 GiB of memory.	±20.00 USD/month + one-time infrastructure cost
Shared	The shared web server on which the legacy application was running before migration to the public cloud.	1.82 USD/month
EC2	Amazon EC2 t2.micro instance	14.28 USD/month
AppEngine	Instance running on Google AppEngine	depends on usage

Table X. Average page generation times (in seconds) and standard deviations for three test pages over 50 iterations.

	Local		Shared		EC2		AppEngine	
	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ
Page 1	0.19588	0.00596	3.20709	0.81836	0.19557	0.00437	6.28543	0.84406
Page 2	0.20555	0.00688	3.00282	0.23948	0.19974	0.00372	6.38447	0.91441
Page 3	0.02882	0.00216	0.33663	0.03426	0.02183	0.00039	1.38429	0.56244

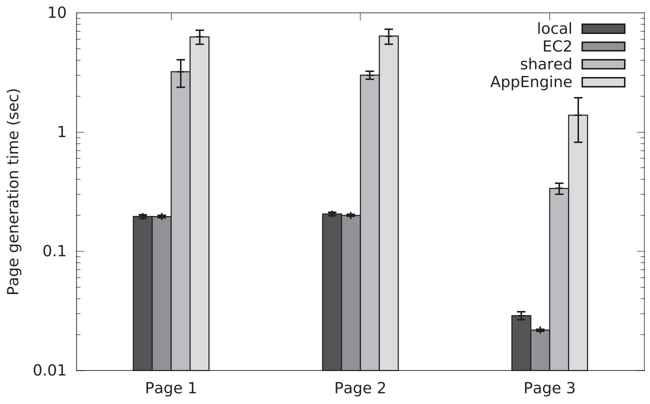


Figure 11. A comparison of the average page generation times for three test pages over 50 iterations.

Table XI. Average end-to-end transaction times (in seconds) and standard deviations for three test pages over 50 iterations.

	Local		Shared		EC2		AppEngine	
	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ	$\overline{y}$	σ
Page 1	0.39160	0.01721	3.39200	0.74018	0.54840	0.10075	6.68200	0.79333
Page 2	0.37520	0.00934	3.11800	0.23113	0.47580	0.08840	7.03000	0.83211
Page 3	0.22260	0.00439	0.43580	0.03662	0.40920	0.03746	1.57200	0.56211

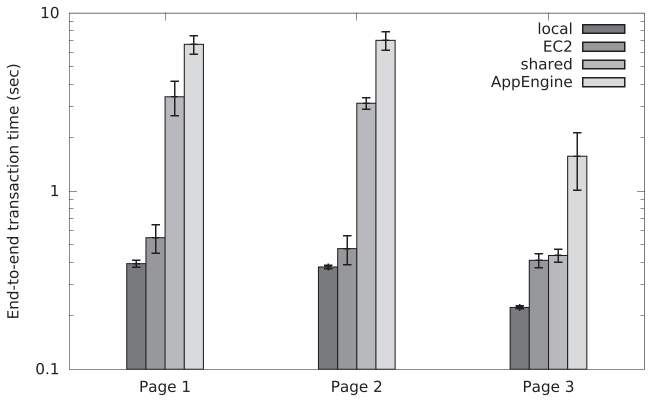


Figure 12. A comparison of the average end-to-end transaction times for three test pages over 50 iterations.

generate one of the selected heavy pages. A possible explanation for this is that PHP support by Google AppEngine is still experimental, and the engine is not yet optimized for PHP. We however would like to note that the performance of Google AppEngine has already improved considerably over the last months, as in September 2014, the similar experiments were executed, and the same page could then take up to 45 seconds to generate. For a production-ready application in the cloud, Amazon EC2 will however be selected to host the application, as it currently is a clear winner in the executed experiments.

7. CONCLUSIONS

Cloud computing and multi-tenancy allow providers to improve the scalability of applications while reducing hosting costs. In this article, we presented a generic approach for migrating legacy software to the public cloud, and adding multi-tenancy to the application. We briefly described the different steps needed to convert the dedicated application to a cloud application, and the steps required to add multi-tenancy to the application. We verified our approach using two case studies from medical software: medical communications software and a medical appointments schedule planner. For the MC software, we migrated some components to a public cloud provider, creating a hybrid cloud, whereas for the schedule planner, we did a full migration of the legacy software to two different public cloud providers.

Migrating an application to the public cloud only requires a limited number of changes, while the conversion from a single-tenant to a multi-tenant application requires more steps as the latter requires limited changes to the application architecture. These modifications are however necessary to fully benefit from the opportunities of public cloud computing. We presented a proactive approach by identifying and eliminating possible future risks, for example by mitigating security risks and analyzing the architecture regarding its scalability.

In our evaluation, we described the advantages of both the cloud migration and the addition of multi-tenancy, and took into account the costs of the migration and remaining risks. After migrating the MC software, the time needed for the initial creation of a new tenant is strongly reduced (from 6 to an estimated 3.5 man-days, including the initial setup of the dedicated hardware), and maintenance has become much easier after migration. The reduction in initial costs and management costs also enables supporting smaller clients for which the costs used to be prohibitive. Supporting these additional clients may present new business opportunities in the long term.

For scenarios where the performance is critical, different public cloud providers should be considered and evaluated, and within a single provider, different instance types might exist. For our second use case, Amazon EC2 has a clear advantage over Google AppEngine for running the schedule planner, and yielded even better results than a dedicated virtual machine on a physical Linux server. The advantages of using a PaaS provider should also be weighted against the increased control gained when using an IaaS provider.

Migrating legacy software to the cloud comes at a cost, and some application components may need to be modified or rewritten. However, by following the multi-step migration approach presented in this article, the benefits of a cloud migration could outweigh the costs of implementing the described changes, as can be seen in the evaluation section of this article.

REFERENCES

1. Armbrust M, Fox R, Griffith A, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M. Above the clouds : a Berkeley view of cloud computing. *Technical Report*, University of California at Berkley, 2009.
2. Schatzberg D, Appavoo J, Krieger O, Van Hensbergen E. Scalable elastic systems architecture. *Proceedings of the ASPLOS Runtime Environment/Systems, Layering, and Virtualized Environments (RESoLVE) Workshop*, ASPLOS, Newport Beach, California, 2011; 1–2. Available: http://communities.vmware.com/docs/DOC-14979 [last accessed January 2015].
3. Guo CJ, Sun W, Huang Y, Wang ZH, Gao B. A framework for native multi-tenancy application development and management. *9th IEEE International Conference on E-Commerce and the 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services, 2007. CEC/EEE 2007*, Tokyo, 2007; 551–558.
4. Maenhaut P-J, Moens H, Verheye M, Verhoeve P, Walraven S, Joosen W, Ongenae V. Migrating medical communications software to a multi-tenant cloud environment. *IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, Ghent, Belgium, May 2013; 900–903. Available from: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber6̄573107 [last accessed January 2015].
5. Hajjat M, Sun X, Sung Y-wE, Maltz D, Rao S, Sripanidkulchai K, Tawarmalani M. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. *Proceedings of the ACM SIGCOMM 2010 Conference*, New Delhi, India, 2010; 243–254.
6. Li A, Yang X, Kandula S, Zhang M. Comparing public-cloud providers. *Internet Computing, IEEE* 2011; **15**(2): 50–53.
7. Khajeh-Hosseini A, Greenwood D, Sommerville I. Cloud migration: a case study of migrating an enterprise IT system to IaaS. *2010 IEEE 3rd International Conference on Cloud Computing*, IEEE, Miami, FL, July 2010; 450–457.
8. Vu QH, Asal R. Legacy application migration to the cloud: practicability and methodology. *2012 IEEE Eighth World Congress on Services*, IEEE, Honolulu, HI, June 2012; 270–277.
9. Walraven S, Truyen E, Joosen W. Comparing paas offerings in light of saas development. *Computing* 2013; **96**(8):669–724. Available from: https://lirias.kuleuven.be/handle/123456789/413659 [last accessed January 2015].

10. da Costa PJP, da Cruz AMR. Migration to Windows Azure - analysis and comparison. *Procedia Technology*; **5**(0): 93–102. Available from: http://www.sciencedirect.com/science/article/pii/S2212017312004422.
11. Moens H, Truyen E, Walraven S, Joosen W, Dhoedt B, De Turck F. Feature placement algorithms for high-variability applications in cloud environments. *Proceedings of the 13th Network Operations and Management Symposium (NOMS2012)*, Maui, HI, 2012; 17–24.
12. Moens H, Truyen E, Walraven S, Joosen W, Dhoedt B, De Turck F. Developing and managing customizable software as a service using feature model conversion. *Proceedings of the 3rd IEEE/IFI Workshop on Cloud Management (CloudMan)*, Maui, HI, 2012; 1295–1302.
13. Moens H, Truyen E, Walraven S, Joosen W, Dhoedt B, De Turck F. Cost-effective feature placement of customizable multi-tenant applications in the cloud. *Journal of Network and Systems Management* 2013; **22**(4):517–558.
14. Moens H, De Turck F. Feature-based application development and management of multi-tenant applications in clouds. *Proceedings of the 18th International Software Product Line Conference - Volume 1*, SPLC '14, ACM, New York, NY, USA, 2014; 72–81.
15. Walraven S, Truyen E, Joosen W. A middleware layer for flexible and cost-efficient multi-tenant applications. *Middleware 2011*, Lisbon, Portugal, 2011; 370–389.
16. Maenhaut P-J, Moens H, Decat M, Bogaert J, Lagaisse B, Joosen W, Ongenae V, Turck FD. Characterizing the performance of tenant data management in multi-tenant cloud authorization systems. *Proceedings of the 14th Network Operations and Management Symposium (NOMS2014)*, Krakow, Poland, May 2014; 1–8.
17. Maenhaut P-J, Moens H, Ongenae V, De Turck F. Scalable user data management in multi-tenant cloud environments. *10th International Conference on Network and Service Management 2014 (CNSM 2014)*, Rio de Janeiro, Brazil, November 2014; 268–271.
18. Moens H, Truyen E, Walraven S, Joosen W, Dhoedt B, De Turck F. Network-aware impact determination algorithms for service workflow deployment in hybrid clouds. *Proceedings of the 8th Conference on Network and Service Management (CNSM2012)*, Las Vegas, USA, 2012; 28–36.
19. Ongenae F, Duysburgh P, Verstraete M, Sulmon N, Bleumers L, Jacobs A, Ackaert A, De Zutter S, Verstichel S, De Turck F. User-driven design of a context-aware application: an ambient-intelligent nurse call system. *2012 6th International Conference on Pervasive Computing Technologies for Healthcare*, IEEE, San Diego, CA, 2012; 205–210. Available from: http://dx.doi.org/10.4108/icst.pervasivehealth.2012.248699.
20. Ongenae F, Claeys M, Dupont T, Kerckhove W, Verhoeve P, Dhaene T, De Turck F. A probabilistic ontology-based platform for self-learning context-aware healthcare applications. *Expert Systems with Applications* 2013; **40**(18):7629–7646. DOI: 10.1016/j.eswa.2013.07.038.
21. Ongenae F, Hristoskova A, Tsiporkova E, Tourwé T, De Turck F. Semantic reasoning for intelligent emergency response applications. *10th International Conference on Information Systems for Crisis Response and Management, Abstracts*, Baden-Baden, Germany, 2013; 1–2.
22. Introducing windows azure, 2013. Available from: http://www.windowsazure.com/en-us/develop/net/fundamentals/intro-to-win dows-azure/ [last accessed January 2015].
23. Azure pricing, 2013. Available from: http://www.windowsazure.com/en-us/pricing/details/cloud-services/ [last accessed January 2015].
24. Betts D, Densmore S, Narumoto M, Pace E, Woloski M. Moving applications to the cloud on microsoft windows Azure, 2010. Microsoft ; O'Reilly distributor.
25. How to create and deploy a cloud service. Available from: http://www.windowsazure.com/en-us/manage/services/cloud-services/how-to -create-and-deploy-a-cloud-service/ [last accessed January 2015].
26. How to use the autoscaling application block. Available from: http://www.windowsazure.com/en-us/develop/net/how-to-guides/autoscaling/ [last accessed January 2015].
27. Monitoring and autoscaling features for windows azure with azurewatch indepth - azurewatch. Available from: http://www.paraleap.com/azurewatch [last accessed January 2015].
28. Google app engine - free platform-as-a-service (paas). Available from: https://cloud.google.com/appengine/ [last accessed January 2015].
29. Aws - amazon elastic compute cloud (ec2) - scalable cloud hosting. Available from: http://aws.amazon.com/ec2/ [last accessed January 2015].
30. Using google cloud sql - php. Available from: https://cloud.google.com/appengine/docs/php/cloud-sql/ [last accessed January 2015].
31. What is google cloud storage? - google cloud storage. Available from: https://cloud.google.com/storage/docs/overview [last accessed January 2015].
32. Php template engine - smarty. Available from: http://www.smarty.net/ [last accessed January 2015].
33. Simulate apache mod_rewrite routing - php. Available from: https://cloud.google.com/appengine/docs/php/config/mod_rewrite [last accessed January 2015].
34. Amazon cloudwatch - cloud & network monitoring services. Available from: http://aws.amazon.com/cloudwatch/ [last accessed January 2015].