SCIENCE CHINA
Information Sciences

• REVIEW •

May 2014, Vol. 57 051101:1–051101:15
doi: 10.1007/s11432-013-5050-z

Software-as-a-service (SaaS): perspectives and challenges

TSAI WeiTek[1,2*], BAI XiaoYing[2] & HUANG Yu[1]

[1]*School of Computing, Informatics, and Decision Systems Engineering, Arizona State University, Tempe, AZ* 85287*, USA;*
[2]*Department of Computer Science and Technology, Tsinghua National Lab of Information Science and Technology, Tsinghua University, Beijing* 100084*, China*

Received June 3, 2013; accepted August 22, 2013; published online March 11, 2014

Abstract Software-as-a-service (SaaS) has received significant attention recently as one of three principal components of cloud computing, and it often deals with applications that run on top of a platform-as-a-service (PaaS) that in turn runs on top of infrastructure-as-a-service (IaaS). This paper provides an overview of SaaS including its architecture and major technical issues such as customization, multi-tenancy architecture, redundancy and recovery mechanisms, and scalability. Specifically, a SaaS system can have architecture relating to a database-oriented approach, middleware-oriented approach, service-oriented approach, or PaaS-oriented approach. Various SaaS customization strategies can be used from light customization with manual coding to heavy customization where the SaaS system and its underlying PaaS systems are customized together. Multi-tenancy architecture is an important feature of a SaaS and various trade-offs including security isolation, performance, and engineering effort need to be considered. It is important for a SaaS system to have multi-level redundancy and recovery mechanisms, and the SaaS system needs to coordinate these with the underlying PaaS system. Finally, SaaS scalability mechanisms include a multi-level architecture with load balancers, automated data migration, and software design strategies.

Keywords software-as-a-service, SaaS architecture, customization, multi-tenancy architecture, redundancy and recovery, scalability

Citation Tsai W T, Bai X Y, Huang Y. Software-as-a-service (SaaS): perspectives and challenges. Sci China Inf Sci, 2014, 57: 051101(15), doi: 10.1007/s11432-013-5050-z

1 Introduction

Software-as-a-service (SaaS) is one of three principal components of cloud computing [1], with the other two being platform-as-a-service (PaaS) and infrastructure-as-a-service (IaaS). SaaS runs on top of PaaS that in turn runs on top of IaaS. SaaS has not only its business model but also its unique development processes and computing infrastructure. At the system level, unlike traditional software that runs on operation systems, SaaS is usually deployed on a PaaS system such as GAE[1)], EC2[2)], or Azure[3)], or

*Corresponding author (email: wtsai@asu.edu)

1) Google App Engine. http://code.google.com/appengine/.
2) Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
3) Windows Azure. http://www.windowsazure.com/.

specialized SaaS infrastructure. To manage the software data, conventional systems often use relational databases that support concurrent processing and give readers priority over writers. Data schemas are usually normalized. Conversely, a PaaS system often has a large amount of data with big data solutions, such as NoSQL databases[4)] and MapReduce-style parallel processing [2]. A PaaS system may favor writers over readers, de-normalize data schema and adopt weaker consistency requirements such as eventual consistency. Conventional systems do not address multi-tenancy issues, whereas a key feature of PaaS is that it supports all tenant applications with one code base [3–6][5)]. For reliability, availability, and security, conventional systems use security kernels and redundancy and rollback mechanisms while PaaS leverages built-in testing, continuous validation, and automated triplicate writing and recovery as major techniques. A PaaS infrastructure often has built-in fault-tolerant facilities and supports scalable computing. SaaS is a new software delivery model. While SaaS can be constructed in a service-oriented manner, SaaS and service-oriented software are different. Today's SaaS is rather distinct from service-oriented computing (SOC). SOC[6)] emphasizes a composable architecture with which to integrate heterogeneous software systems. Hence, a service-oriented infrastructure is often composed of a stack of standard protocols for publishing services, and for orchestrating or choreographing services dynamically. SaaS emphasizes a customizable architecture of a massive scalable system built upon a cloud infrastructure [7–11]. Hence, it is enabled by database and PaaS design techniques to support customization and scalability at various levels including the user interface, workflow and persistence. While a SaaS service can be specified using service-oriented standards such as WSDL or OWL-S, current SaaS services focus on execution and utility services that users can use in their applications. Compared with SOC, SaaS has unique features of customization, persistency, and scalability, following multi-tenancy architecture (MTA). MTA is a key feature of SaaS [3–6,11]. It provides all tenants software to share the same code base with configuration data stored in databases or data stores. In this way, hundreds of thousands of tenants may use the same SaaS infrastructure with the same database of services. One tenant can be different from another tenant owing to differences in their configurations, while each tenant feels like a dedicated application. SOC software can be customized and one common customization is to use different compositions to account for differences in configurations [12]. In SaaS, customization is often achieved by MTA database design and uses a metadata-driven approach where metadata tables store information about tenants and the SaaS system uses these metadata tables to search, retrieve, and update tenant information. In SaaS, a database is often an integrated part of SaaS where a new tenant application can be composed using services stored in the database within the SaaS system [3–5]. Upon a user request from a tenant, the SaaS system will retrieve the related components from the database, and compose them into code that can then be compiled into executable code. The executable code is then executed, and the resulting data may be stored in the database, and returned to the user. Thus the SaaS application can be considered a database-centric operation where the database is an integral part of the SaaS system. Furthermore, well-known SaaS systems such as Salesforce.com often use a flexible schema rather than a rigid schema commonly used in relational database management systems (RDBMS) where data of different types use the same schema for storage. This reduces the schema design for each tenant as a SaaS may host hundreds of thousands of tenants. Scalability refers to the ability of a system to handle a growing amount of work with stable performance using a proportional amount of new resources. SOC software can be scaled topologically, across multiple locations, organizations, and business units. A SaaS service may have a large number of tenants, and each tenant may have hundreds of thousands of users, and thus a SaaS infrastructure needs to support millions of users with scalable performance.

2 SaaS architecture

SaaS architecture needs to address customization [8], MTA, scalability [11] and rapid development. The four main SaaS architectures are: database-oriented architecture [13], middleware-oriented architecture,

4) NoSQL-Database.org. http://nosql-database.org/.
5) Chong F, Carraro G, Wolter R. Multi-Tenant Data Architecture. http://msdn.microsoft.com/.
6) OASIS. Web Services Standards. http://www.oasis-open.org/standards.

Table 1 Summary of SaaS architectures

	Database-oriented	Middleware-oriented	PaaS-based	Service-oriented
Customization	Comprehensive customization using code	Same customization as original software	Custom made software for the specific PaaS	Comprehensive customization using ontology and composition
MTA	De-normalization, complex design	Use the existing schema, rapid development	Use namespace for identifying tenants	Can use a variety of MTA design
Scalability	K-level scalability	K-level scalability	Depending on the PaaS scalability	K-level scalability

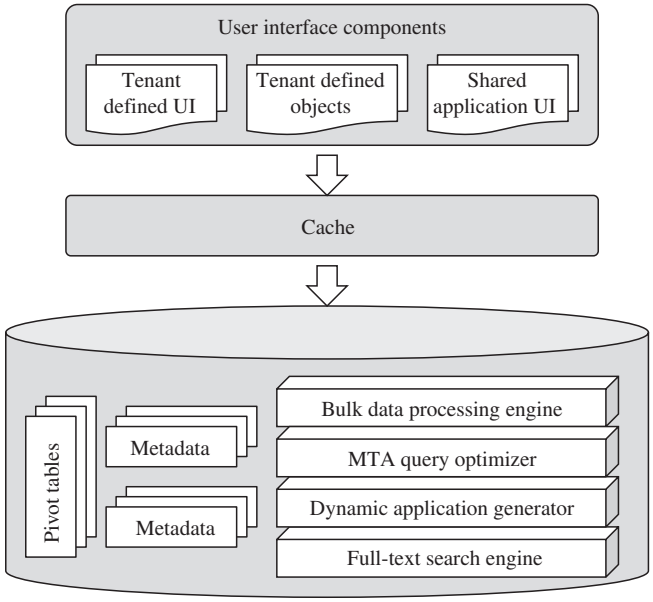

Figure 1 Force.com MTA architecture.

PaaS-based architecture, and Service-oriented SaaS architecture [1]. Table 1 compares the implementation of different SaaS Architecture styles.

2.1 Database-oriented SaaS

Salesforce is a representative system of database-oriented architecture[7] [13], see Figures 1 and 2.

Customization. The architecture supports customization by allowing engineers to develop code and link the code to software components such as service components of a graphical user interface (GUI).

MTA. The architecture shares databases and schemas among tenants. Because of this sharing, each tenant uses the same schema and database as every other tenant.

Scalability. The architecture uses two-level scalability mechanisms, where the top-level scheduler dispatches the user requests according to tenant information to different clusters (in case of Salesforce.com, this is called portal on demand), and the second-level scheduler assigns tasks to different servers within the same cluster where each server is stateless, but all servers share the same database. Various optimization algorithms can be applied to the data allocation in these clusters to address scalability; furthermore, the two-level scalability mechanism can be extended to K-level such as 3-level or 4-level with each level addressing specific aspects. Salesforce.com has developed many mechanisms supporting the execution of this architecture to maintain performance and reliability. Recently, it has also supported another level of MTA, sub-tenancy, where a tenant application can support multiple sub-tenants.

7) The Force.com Multitenant Architecture: Understanding the Design of Salesforce.com's Internet Application Development Platform. http://www.salesforce.com/.

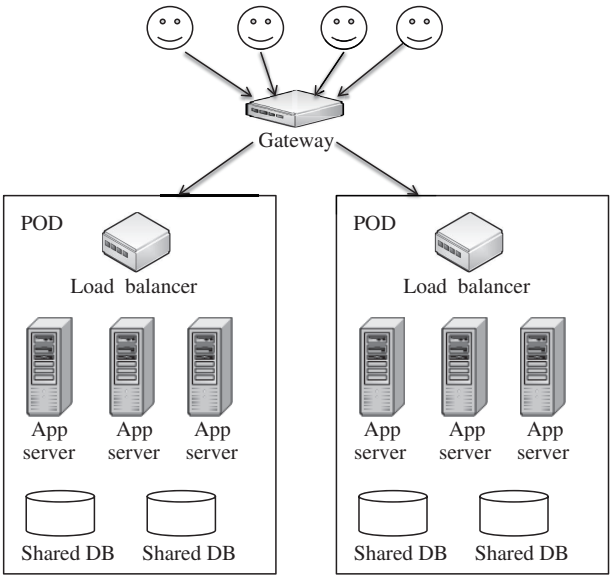

Figure 2 Force.com scalability architecture.

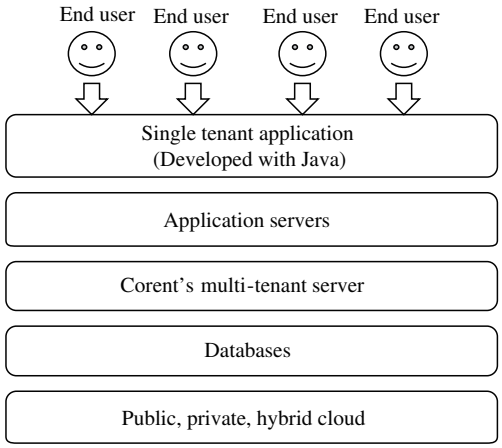

Figure 3 Corent Multi-tenancy architecture.

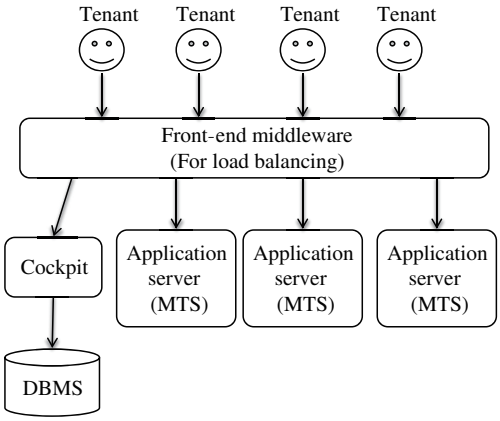

Figure 4 Corent scalability architecture.

2.2 Middleware-based approach

A representative system of the middleware-based approach is Corentech.com, see Figures 3 and 4[8)].

Customization. Customization is not a priority for Corenttech.com as it emphasizes rapid development. Traditional software can be updated to MTA software by changing the data access portion of the software, and such updates can be done in hours.

MTA. A kernel is used to manage the MTA layer where each data access to a specific database is changed to a call to the kernel, and the kernel then channels the request to appropriate databases below the kernel. Conventional databases can be used below the kernel, and thus, migration from an existing application to an MTA SaaS can be efficient as existing software (run on top of the kernel) requires minimum changes, and existing databases (called by the kernel) can also be used.

Scalability. Corenttech.com uses two-level scalability mechanisms where the top-level scheduler allocates requests to different clusters according to tenant information. In summary, Corenttech.com focuses on rapid development and MTA, and existing software and databases require minimal changes while keeping existing properties such as atomicity, consistency, isolation, and duration (ACID) properties in

8) CorentTech Multi-tenancy Server. http://www.corenttech.com/.

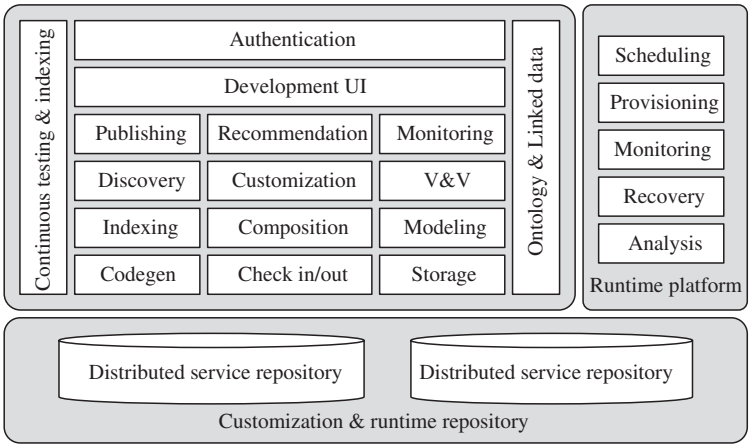

Figure 5 EasySaaS architecture.

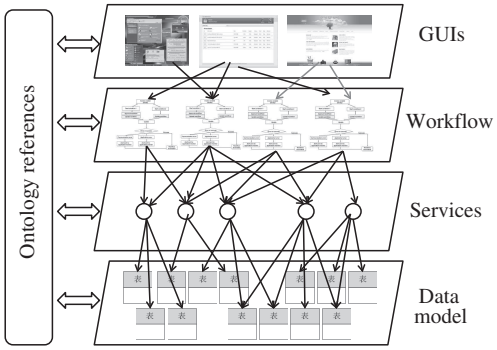

Figure 6 EasySaaS customization architecture.

the database. The security concern is less than that for the Salesforce.com architecture because tenants do not share the same database or schema. Furthermore, ACID properties can be supported by the RDBMS used in the back-end for the tenant applications.

2.3 Service-oriented SaaS

EasySaaS is a representative service-oriented SaaS system, see Figures 5 and 6 [1]. Essentially, this is an SOC approach to SaaS. Not only can tenant applications be developed in a service-oriented manner (i.e., by publishing, discovery, composition, deployment, and execution) but also the SaaS infrastructure can be developed in a service-oriented manner where major SaaS infrastructure components are services, and different SaaS infrastructures can be composed using different component services.

Customization. The architecture supports heavy customization as it allows developers to compose a tenant application in a service-oriented manner. Furthermore, a consumer-oriented approach can be used where existing application templates can be reused to compose applications [14]. As shown in Figure 6, a tenant application can be composed by linking GUI, workflow, service and data components stored in the SaaS database or new components can be constructed and stored in the database [8]. A recommendation system can be used to suggest appropriate components according to structural and semantic information of the tenant application and components stored. This is the grapevine approach [15].

MTA. The approach supports MTA, and a variety of MTA designs can be used including de-normalization used by Salesforce.com or each tenant can have its own database.

Scalability. The two-level scalability mechanism can be used with other mechanisms such as automated migration.

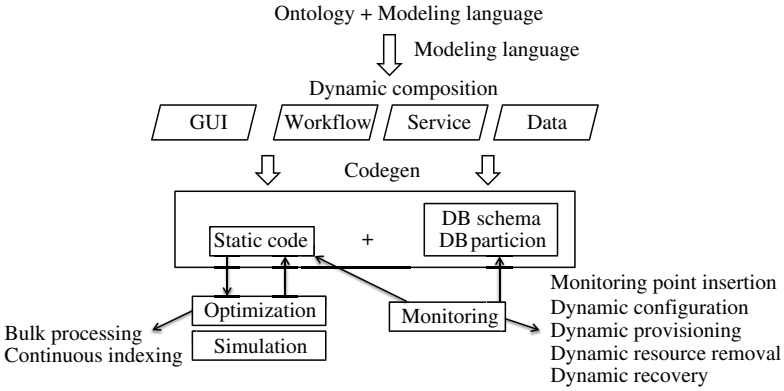

Figure 7 Model-based CodeGen for a PaaS.

2.4 PaaS-based approach

The PaaS-based approach depends on the underlying PaaS to support MTA SaaS including its scheduling, fault-tolerant mechanisms, and scalability.

Customization. The customization feature needs be developed by the SaaS developer. Existing PaaS often does not allow direct access of the underlying data store or database. Therefore, although customization is possible, it requires considerable design as the software needs to specify various places where customization can be made, and the number of options for each customization place can be limited or might need to be developed manually by the SaaS developer.

MTA. Each PaaS has its MTA support features. For example, GAE provides the namespace for MTA, each tenant is uniquely identified by an entry in the namespace and that information is used in the PaaS to distinguish one tenant from another. Microsoft Azure has similar mechanisms.

Scalability. The salability of SaaS applications using a PaaS-based approach depends on the scalability mechanism of the underlying PaaS. In fact, other SaaS features such as fault-tolerant computing also depends on the corresponding mechanisms in the PaaS. For example, if each data write is at least triplicated in the PaaS as done in GAE, each write request by the SaaS application will be triplicated into three different chunks of 64 MB, and recovery mechanisms will operate at the chunk level where they recover data one chunk at a time using the metadata in the PaaS.

2.5 Platform-independent SaaS development

Some existing SaaS systems were developed together with their underlying PaaS system. Thus, SaaS developers access the PaaS internal to optimize the SaaS. However, such luxury is not extended to other SaaS developers that use an existing PaaS system to implement SaaS.

It is possible to develop the SaaS in two stages: the first is the platform-independent SaaS model development, to be followed by platform-dependent SaaS development in which code is generated for a specific PaaS. This is shown in Figure 7 [11]. This is similar to the OMG approach of software development, where there was a platform-independent phase before a platform-dependent phase.

In the first phase, a tenant application can be modeled including its overall architecture, workflows, services, and data and once modeled, the model can be analyzed, simulated, and verified at the model level. Once the model is verified, the model can be extended to a specific platform or PaaS such as GAE or Azure by generating platform-specific code. For example, for GAE, it may be necessary to use the namespace to support the MTA, and it is necessary to generate code in specific languages supported by the underlying PaaS such as Python.

In this approach, the availability of PaaS simulators is critical as they can simulate the performance of different PaaS for the same SaaS. Taking this approach, a common platform-independent infrastructure can be developed where tenant application developers can compose their applications in a model-driven

manner using components stored in the repository, and according to model evaluation and simulation using PaaS simulators, they can select an appropriate PaaS or conventional system for execution.

Recently, many innovative SaaS architectures also appeared. For example, the Workday SaaS uses an object-oriented approach to design its software, and uses model-view-controller (MVC) architecture with scalability mechanisms[9].

3 Customization

Customization has been studied and practiced for over 40 years in terms of software families, object-oriented design patterns, object-oriented application frameworks, and product-line engineering. The process of customization is as follows:

• Identify those areas that are likely to be changed (variation points or variant parts), and those areas that are likely to remain constant invariants parts.

• Place an abstraction layer around the variants parts so that actual instances that can be used are listed as options to be selected at runtime or design time.

• Carry out this process in an iterative hierarchical manner until all potential variation points are identified, and design a family of programs to accommodate those actual instances that can be used at the variation points.

In SaaS, tenants can have their customized applications stored in the database, and often the applications are not stored as a unit in the database. Instead, each tenant application is decomposed into its GUI, workflows, services, and data components, and each component is stored in the database together with components of the same kinds. For example, a SaaS GUI database contains all the GUI components used by all the tenants. The following is the sequence of actions taken by a SaaS when a user request arrives:

1. When a user request arrives, first identify the tenant ID of the user, and retrieve all the components used by the tenant if these components are not in the cache.
2. Compose the tenant application at runtime if the application is not in the cache.
3. Compile the tenant application if it has not yet been compiled.
4. Execute the user request using the executable code produced in Step 3.
5. Return the results to the user.

Thus, in a SaaS system, components are shared among tenants, and only active or recently active tenant applications and/or components are available either in the cache or memory. In this way, a SaaS system can be viewed as an executable system with an heavily used database of components.

To support SaaS customization, components of different types need to be stored, and they need to be annotated so that they can be discovered by tenant developers for customization. Common service specification techniques such as WSDL can be used to specify those components, and once specified, information can be stored in an ontology database for searching, discovery, and reasoning.

A tenant application may also specify its requirements in formal, semi-formal, or informal languages, and automated tools such as text processing [16] can be used to analyze the requirements and identify reusable components in the SaaS database. The matching can be done using a keyword search and the semantic distance to identify relevant components. For example, the sorting keyword in the tenant requirement document can be used to search reusable components in the SaaS database, and it may identify GUI components that support search queries, workflows that support sorting algorithms, and a collection of sorting services using different algorithms, and data components to store items to be sorted. Once the components are identified, the tenant developer can make the final selection and link these components together to form a complete program.

The performance of a SaaS database is critical as many queries will be executed at runtime to serve user requests, and large searches will be performed in the SaaS database to identify appropriate components. Thus, a recommendation system can be used. Typical algorithms used in recommendation systems

9) Workday innovative technology. http://www.workday.com/.

Table 2 SaaS customization options

Customization options	Description
Fixed variation points and fixed options	Fixed options for tenants to choose, each options already verified by the SaaS infrastructure before deploying.
Fixed variation points but allow tenant-supplied options	Allow both fixed options and tenant-supplied options, each variation point may come with a verification mechanism such as regression testing to verify options supplied by tenants.
Allow tenants to create their own variation points and options	The SaaS may provide a list of templates for variation points specifying the kinds of options allowed and their constraints, and verification mechanisms for each variation points. The variation template can be stored in the SaaS database for other tenants to reuse.
Intelligent customization	Allowing every component in the tenant application to be selected and composed using the components in the SaaS database. Furthermore, the SaaS has a recommendation system to provide various templates and suggestions for composition.
Customizable SaaS infrastructure	Not only can tenant applications be composed, the specific SaaS infrastructure can be selected. Notable selections include DB selection (individual, shares, and other options), table organization (row, column or hybrid), redundancy and recovery (no redundancy, double redundancy, triple redundancy) distribution (centralized redundancy within the same cluster, redundancy at multiple clusters), execution (parallel, stay-in-memory, periodic, redundant computing).
SaaS and PaaS configuration	In addition to those features provided in the above row, provide selection of PaaS platform, execution models, redundancy management, scheduling priority, memory and storage requirements.

include collaborative filtering [17] and content-based filtering [18,19]. Recommendation systems using a hybrid approach are also popular [20–22]. A good summary of collaborative filtering techniques was presented in [23].

There are many options for SaaS customization, as listed in Table 2. A SaaS maturity model was proposed[10] in 2006. The model classifies SaaS customization into four levels: ad-hoc/custom, customizable or configurable, multi-tenant efficient, and scalable levels. This classification fits well with initial SaaS implementations; however, recent SaaS development does not necessarily follow this model. In fact, some systems have multi-tenancy support and scalability features first. Additionally, each level can be further classified into multiple sub-levels. Customization can be further classified as follows:

Level 1 Existing customization. At this level, SaaS components can be retrieved from the database, and tenant developers will develop code to link these components together. If the needed component is unavailable, the tenant developer needs to develop the component, and submit for publication in the SaaS database after validation.

Level 2 Intelligent customization. At this level, tenant developers can use techniques such as service specification, search, discovery, and automated dependency analysis, and recommendation systems such as the Grapevine model for an existing PaaS. However, SaaS components can be added into the SaaS database for the development and customization of tenant applications.

Level 3 SaaS infrastructure configuration. At this level, the SaaS infrastructure can be changed, in other words, the SaaS infrastructure may be viewed as another SaaS system, and each SaaS infrastructure is considered as a tenant application of the SaaS infrastructure system. For example, the SaaS infrastructure system may provide three options for each SaaS infrastructure: one for row-oriented processing, a second for column-oriented processing, and a third for both row-oriented processing and column-oriented processing. A new database design approach DBaaS where each database system instance can be individually customized, and DBaaS can be used at this level.

Level 4 SaaS + PaaS configuration. This level provides the most sophistication as both SaaS infrastructure and PaaS infrastructure can be configured for optimal performance. For example, a specific tenant may need to perform certain kinds of processing only, and the access patterns are regular. Thus,

10) Chong F, Carraro G. Architecture Strategies for Catching the Long Tail. http://msdn.microsoft.com/.

Table 3 Database (DB) design for tenant isolation within a cluster

Various approaches	Issues
Each tenant has a DB with its own schema.	Good for tenant isolation, significant schema design effort.
Each tenant has a DB, but all tenants use a collection of schema only, possibly one schema for one application domain.	Good for tenant isolation, reasonable effort in schema design.
Each tenant has a DB but all share the same schema.	Good for tenant isolation, less effort in schema design.
Shared DB, but each tenant has its schema.	Reasonable solution for tenant isolation, significant schema design effort.
Shared DB and schemas.	Significant effort in isolation.
Each extension is a table.	Lots of join operations may be needed.
Sparse columns for tenant information	Sparse tables
Hybrid solutions: for example, the SaaS has a collection of schema, critical tenants have their own DB, but some tenants share the same DB with the same schema, and different clusters have different database approaches for their own tenants.	Various tradeoffs in these hybrid solutions, one key is to reduce the number of schemas needed, so to reuse database software.

the SaaS infrastructure can be configured so that the SaaS database is optimized for such processing, and the PaaS infrastructure is also configured to process these tasks in the most efficient manner.

Levels 3 and 4 represent a configurable SaaS infrastructure for configurable tenant applications, with two levels of customizations.

4 Multi-tenancy database design

Most SaaS architecture use databases to support MTA. The issue is the selection of appropriate database management systems and the schema design. Table 3 lists considerations of database design choices for MTA-associated SaaS; specifically, the table describes whether each tenant has its own database, tenants share a database but each has its own schema, or tenants share a database and the same schema.

If each tenant has its own database, there is a benefit for tenant isolation as the firewall is created at the database level to ensure tenant isolation. For this design option, several alternatives are possible to minimize the engineering effort.

1. One choice is that while each tenant has its own database, all the tenants share the same schema, and thus the software needed will be the same.

2. Another choice is to allocate individual databases for large and/or critical tenants. Small tenants and/or tenants that do not have strict security requirements can share databases and even the same schema.

3. Another choice is that each database assigned to a tenant is a customized database from a DBaaS (database as a service). In this way, the DBaaS will maintain one database code base, but multiple database instances can be created, each for a specific tenant.

If tenants share the same database but each tenant has its own schema and thus private tables, then there is a reasonable compromise between having an individual database for each tenant versus everything being shared. However, designing individual schema will require much more effort, unless the design of database schemas according to user requirements can be automated. This approach may require tenant developers to design a schema, which will need to be verified by SaaS infrastructure engineers before it can be deployed. Note that MTA SaaS systems require high performance, and database processing software requires tuning and experimentation before its performance is acceptable. Thus, asking tenant developers to develop a schema appropriate for specific SaaS and PaaS infrastructure may be an issue. There are several ways to address this issue.

1. One way is that a collection of schemas with trade-offs can be offered to tenant developers, and

they can pick their own schema from the collection. In this way, the number of MTA database software applications will be limited.

2. Another way is to take advantage of the multi-level scalability structure. Each cluster can have its own database approach; i.e., one cluster may share a common database and schema, while another has individual databases and a collection of schemas for its tenants.

If tenants share both a database and schema, the number of database processing software applications will be minimized, but the data of one tenant may occupy the same table as the data of other tenants. With the multi-level scalability structure, the number of tenants occupying the same database is reduced, but all tenants in the same cluster share the same database and schema, and there can still be a large number of tenants in a cluster. As all tenants in all clusters use the same schema, database processing software can be optimized for the entire SaaS system, thus saving effort.

Other database design approaches are possible, e.g., having private tables for each tenant, providing tenant customization as extensions to tables, using a sparse table to store tenant information, and using a multi-dimensional database to store tenant information.

While many solutions have been proposed for MTA database design with supporting data, few designs have been subjected to experiment in a realistic SaaS infrastructure with a PaaS. Thus, the conclusions are preliminary at best. Furthermore, it is necessary to consider scalability, reliability including redundancy and recovery mechanisms, performance issues, customization, and engineering effort while designing MTA at the same time. The use of in-memory databases, NoSQL databases, new SQL databases and XML-based databases in the context of MTA SaaS are also interesting problems.

5 Redundancy and recovery (R&R) mechanisms

Modern SaaS systems often have extensive built-in R&R mechanisms at multiple levels. Specifically, Salesforce.com has the following R&R mechanisms[11)].

• Data centers. Multiple data centers interconnected by high-speed networks are capable of backing up each other in case of the failure of one center.

• Network level. There are multiple network carriers with redundant routers, and fail-over configured firewalls. There are redundant hubs and switches at virtual local area networks (VLAN).

• SaaS level. There are multiple load balancers with their own loads balanced, and clustered Web, application, application programming interface (API), search, cache, index, and batch Servers.

• Database level. Oracle RAC EE runs on four-way clustered nodes with excess capacity to carry the load when a node fails.

• Storage level. There are multiple paths that ensure reliability by connecting four DBMS servers, and alternative paths to storage directors, and the storage systems have built-in redundancy.

One interesting trade-off is the R&R mechanism at the SaaS or PaaS level, knowing that there are a variety of SaaS and PaaS interactions. For example, the SaaS PaaS interaction in the approach taken by Corenttech.com will be different from the corresponding interaction in the approaches taken by SOA SaaS and Salesforce.com.

Most PaaS systems provide their own R&R mechanisms. For example, in GAE, each data write is written at least three times to ensure reliability, and critical components and data have more redundancy. However, not all the R&R mechanisms are open for SaaS developers to use. For example, some of the R&R mechanism of GAE is at the chunk level where binary data are stored, and GAE uses its metadata table to interpret the data.

Most PaaS systems use metadata to identify appropriate data and determine appropriate actions to act on the data, and thus, metadata are critical to PaaS operations. Thus, a PaaS system often has extensive backup mechanisms for the metadata tables. Similarly, many SaaS systems use metadata to identify data and to determine actions to act on the data, and thus, metadata are critical. Additionally,

11) C. Moldt. Behind-the-Scenes at Salesforce.com. http://salesforce.vo.llnwd.net/.

in [24], SaaS metadata information is duplicated into SaaS data tables to ensure that metadata are not easily lost.

One way to improve the R&R mechanisms is to annotate the data stored in each chunk and to store the annotation in the chunk in addition to the metadata table. In this way, as additional space will be used to store tenant information, less space is available to store other data, but each chunk can identify the tenant information without the metadata table. The Workday SaaS takes this approach and applies a role-based security model to each chunk. This tenant-aware concept can be applied to many resources in the PaaS system, including storage and networks. Each packet in transmission may store the associated tenant information, and routing and scheduling can be done according to tenant information. Cisco, VMware and NetApp have started a joint project where a cloud infrastructure can be made tenant-aware. Tenant awareness can be achieved at multiple levels:

- each tuple with a tenant ID;
- each table with tenant IDs;
- each chunk with tenant IDs;
- each message with tenant IDs;
- each machine with tenant IDs;
- each clusters with tenant IDs; and
- each load balancer with tenant IDs.

This is similar to the capability-base system where each data packet or message in transmission or data packet in stable storage or memory, large or small, is tagged with its ID.

While the underlying PaaS system may provide excellent R&R mechanisms, a SaaS system may need its own R&R mechanisms. Specifically, a SaaS-level R&R mechanism may contain information more relevant to the SaaS and tenant developers. For example, SaaS components such as ontology, data and metadata tables can be duplicated to ensure reliability [6].

6 Scalability

In MTA SaaS, each SaaS component may be shared by multiple organizations (tenants). Each tenant may have hundreds or thousands of users, and thus number of concurrent accesses from users can be high.

There are generally two solutions to scale a software system: scale-up and scale-out. Scale-up (i.e., vertical scaling) means running the application on a machine with a better configuration, including more computing resource, more memory, higher disk bandwidth and larger disk space. Scale-out (i.e., horizontal scaling) means running the application distributed on multiple machines with similar configurations. Because the resources of a single machine cannot be increased infinitely, and the increase in cost is not proportional to the increase in resource, scale-out is necessary.

Scalable design principles for application servers include divide-and-conquer, asynchrony, encapsulation, concurrency, and parsimony [25]. Divide-and-conquer means the system tasks should be divided into smaller tasks with single functions, and a system should be well partitioned into components. Asynchrony means work can be done on a resource-available basis, and this may imply distributed and/or self scheduling and background processing. Encapsulation means system components and architecture such as layers are well encapsulated. Concurrency means tasks can be done in parallel taking advantages of the distributed nature of hardware and software. Parsimony means that the design considers the cost efficiently.

Other common techniques include system partitioning, service-based layered architecture, pooling and multiplexing, queuing and background processes, data synchronization, distributed session tracking, and intelligent load distribution. For example, staged event-driven architecture (SEDA) shows how these principles can be used to develop a scalable architecture. The original complex event processing is divided into different stages (divide and conquer) and encapsulated with queues for interacting each stage, and each stage can be executed asynchronously and tracked and monitored.

Table 4 Scalability factors

Factors	Issues
Levels of scalability mechanisms	Adding one level increase scalability, and each layer addresses one issue, with automated scaleup/down at each layer.
Automated migration	Support load balancing, need DB design to support migration, and optimization algorithms are needed.
Tenant-awareness	Consider tenant-specific customization (optimized clustering those tenants sharing processes or GUIs), tenant-aware automated migration (asynchronous migration first and index construction later).
Fault-tolerance and recovery	Detection of failed nodes, incremental recovery, minimize data movement, rapid and distributed recovery.
Architecture and DB access	System structuring by SOA, partitioning by functionality, API design.

Several important factors affect the scalability of the SaaS application, as listed in Table 4, including levels of scalability mechanisms, automated migration, tenant awareness, workload support, recovery and fault-tolerance, software architecture, and database access.

Levels of the scalability mechanism. A typical SaaS application has three tiers-storage, application and presentation - and each processor can handle all three. However, SaaS systems often add one or more load balancers at the cluster level to route tenant requests to different processors for processing; furthermore, several more load balancers are responsible for routing tenant requests to different clusters. With these load balancers, a SaaS system can add new clusters into the system when the load increases, and each cluster can have additional processors, without changing the overall system architecture. Similarly, one can remove one processor from a cluster, and/or remove a cluster from the system when the load decreases.

Scalability of multiple levels provides the SaaS application with flexibility and control. Each level can be scaled independently without affecting or being affected by the scaling mechanisms and operations at other levels. Furthermore, scalability mechanisms at different levels may use different techniques as each level has its own constraints and objectives. For example, a SaaS can be scaled with respect to tenants at the top level, and each cluster is tasked to handle certain tenants only. When new tenants arrive at the system, although no cluster can handle any more tenants, a new cluster can be created. This has the advantage of simple tenant management; if some tenants are heavily loaded (while other tenants are not so heavily loaded), their requests are still handled by the same clusters. Another approach commonly used in a SaaS system is to make each server stateless in a cluster, and the cluster load balancers can then route any tenant requests to any lightly loaded servers or processors.

K-level scalability structure is a natural extension of two-level scalability structure, and each level can address multiple issues at a time; possible issues related to tenants, tasks, application domains, security, geographical locations, and primary/backup storage.

Automated migration. Tenant data need to be migrated occasionally for better performance. Several issues need to be considered. First, it is necessary to determine if the migration will be done online or offline. Online migration (i.e., the migration will take place while the applications are still in operation) is more difficult than offline migration, where the migration takes place when the SaaS shuts down its services for maintenance. Although online migration can provide 24/7 services for SaaS customers, the added complexity of online data migration is high. Second, it is necessary to determine the data to be moved for scalability. One strategy is to move the minimum amount of data (to minimize the bandwidth demand) and to move it to the closest node (to minimize latency delay), but other approaches are possible. For example, one strategy is to move the indices with the tenant data, whereas an opposite strategy is to move the tenant data only. The second option requires reconstruction of indices after data migration. Third, the storage design should take tenant data into consideration. For example, all the data belonging to the same tenant should be grouped together for migration.

Tenant awareness. The storage design needs to be tenant-aware to support customization, maintain tenant isolation, and facilitate data migration. Tenant awareness can be supported by two types of

traceability: forward and backward traceability. Forward traceability means that given a data item to be queried or inserted, the system knows which tenant owns the data item and where to query/insert the data item by tracing forward from the tenant table. Backward traceability means given a stored data item, the system can trace back from the data item to the tenant table. If the system supports only forward traceability, the system can perform data migration by tracing the tenant table. However, if the system also provides backward traceability, data may be moved autonomously and asynchronously upon detection of the need for data migration. Providing both types of traceability for each data item is expensive, but providing both for a group of data items, especially those belonging to the same tenant, will be useful in autonomous data migration.

Workload support. Different workloads require different scalability mechanisms. For an on-line analytical processing (OLAP) workload, a high portion of the requests are reading data from the system. In this case, the system should be able to scale in case of a high volume of read operations. For the an on-line transaction processing (OLTP) workload, write operations are dominant. In this case, the system should be able to distribute the write operations to avoid a bottleneck at a single node. For a mixed workload where the proportions of read and write operations are close, the architecture needs to be designed to ensure there is no bias towards either type of operation because the bias may result in poor scalability.

Recovery and fault-tolerance. Recovery and fault-tolerance mechanisms also the system scalability. First, the system should be able to detect the failures of nodes. When a node fails, the system should automatically scale down without significant performance degradation. When the failed node comes back, the system should automatically scale out and recover its previous working status.

Software architecture. It is necessary to avoid coupling in the architecture so that each part of the application can be independently scaled, without affecting other parts of the software system. SOC can be leveraged to decouple such a large software system to provide scalability. Many scalable systems, such as DynamoDB[12)], adopts SOC.

Database access. Accessing a database is often time consuming, and thus likely to be the bottleneck of a SaaS application. Access to a database can be either direct or indirect. Direct access to a database allows applications to connect directly to the database, and perform necessary operations. Indirect access to a database goes through APIs exposed by certain database services wrapped on the underlying database. Indirect access allows the software and database to have their own scalability mechanism.

Such design can be critical to the system performance and development. For example, Amazon requires that 1) everything wrapped as a service and 2) there is no direct database access. According to Amazon's Chief Executive Officere, "all teams will henceforth expose their data and functionality through service interfaces", and only via these interfaces. This means that there is no direct linking, no direct reads of another team's data store, no shared-memory model, and no back-doors whatsoever. The only communication allowed is via service interface calls over the network. Additionally, service interfaces must be well designed from an external point of view with any exceptions. Furthermore, "anyone who doesn't do this will be fired."

7 Conclusion

Because of its unique architecture, SaaS has had a significant impact on both software engineering and databases. Specifically, SaaS introduces new software lifecycle models that are distinct from traditional software lifecycle models. SaaS has two distinct lifecycle models. One for tenant application development given a SaaS infrastructure running on top of a PaaS, and the other is the lifecycle model for the SaaS infrastructure giving a specific PaaS or allowing both the SaaS and PaaS to be developed concurrently.

Tenant applications are often integrated with a database. Tenant applications are developed online using components stored in the SaaS databases if components are available. If the tenant application is developed in a platform-independent manner, a tenant application model needs to be verified and

12) Amazon Architecture. http://highscalability.com/amazon-architecture.

simulated before it can be used to generate platform-dependent code for a specific PaaS for execution.

The development of SaaS infrastructure is much more involved than the development of tenant applications, as it involves the development of a system to support tenant applications on the Web using components stored in SaaS databases. Furthermore, it needs to compile tenant applications and execute them in a target PaaS system, and monitor the progress. In case of failures, the SaaS needs to perform recovery and reconfiguration. The SaaS also needs to address the scalability of tenant applications using the resource and capabilities provided by the underlying PaaS.

As most tenants will reuse components stored in SaaS databases, tenant application requirements may need to specify items stored in the SaaS databases. For example, if the SaaS database stores object-oriented items such as classes, instances, and methods, a tenant application may need to specify its requirements using the same terms. If the SaaS database contains SOC items, tenant applications need to specify their requirements in terms of services, workflows, services and data using vocabulary from an ontology system. Thus, the SaaS system imposes a specific way of tenant software development. Note that testing methods for SaaS are different [26–30].

The SaaS also has a significant impact on database management systems. Salesforce.com demonstrated that a de-normalized but relational database can serve a huge number of users with satisfactory performance and security. It also demonstrated that, using this approach, each tenant can still enjoy its own customization, and data can be reliably stored in the SaaS and recovered in case of system failures.

Acknowledgements

This project was sponsored by United States National Science Foundation Project DUE (Grant No. 0942453), National Science Foundation China (Grant No. 61073003), National Basic Research Program of China (Grant No. 2011CB302505), and Open Fund of the State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2009KF-2-0X). It is also supported by Fujitsu Laboratory.

References

1 Tsai W T, Huang Y, Shao Q H. EasySaaS: a SaaS development framework. In: Proceedings of IEEE International Conference on Service-Oriented Computing and Applications, Irvine, 2011. 1–4
2 Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. Commun ACM, 2008, 51: 107–113
3 Aulbach S, Grust T, Jacobs D, et al. Multi-tenant databases for software as a service: schema-mapping techniques. In: Proceedings of ACM International Conference on Management of Data, New York, 2008. 1195–1206
4 Bezemer C P, Zaidman A, Platzbeecker B, et al. Enabling multi-tenancy: an industrial experience report. In: Proceedings of IEEE International Conference on Software Maintenance, Timisoara, 2010. 1–8
5 Elmore A J, Das S, Abbadi A El. Towards an elastic and autonomic multi-tenant database. In: Proceedings of International Workshop on Networking Meets Databases, Athens, 2011
6 Tsai W T, Shao Q H, Huang Y, et al. Data partitioning and redundancy management for robust multi-tenancy SaaS. Int J Softw Inform, 2010, 4: 437–471
7 Nitu M. Configurability in SaaS (software as a service) applications. In: Proceedings of the 2nd India Software Engineering Conference, Pune, 2009
8 Tsai W T, Shao Q H, Li W. Oic: ontology-based intelligent customization framework for SaaS. In: Proceedings of IEEE International Conference on Service-Oriented Computing and Applications, Perth, 2010. 1–8
9 Gao J, Pattabhiraman P, Bai X Y, et al. SaaS performance and scalability evaluation in clouds. In: Proceedings of IEEE 6th International Symposium on Service Oriented System Engineering, Irvine, 2011. 61–71
10 Krebs R, Momm C, Konev S. Architectural concerns in multi-tenant SaaS applications. In: Proceedings of the 2nd International Conference on Cloud Computing and Service Science, Shanghai, 2012. 426–431
11 Tsai W T, Huang Y, Bai X Y, et al. Scalable architectures for SaaS. In: Proceedings of IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops, Shenzhen, 2012. 112–117
12 Tsai W T, Xiao B N, Paul R, et al. Global software enterprise: a new software constructing architecture. In: Proceedings of IEEE International Conference on and Enterprise Computing, E-Commerce, and E-Services, San Francisco, 2006. 55
13 Wong T, Kao L, Kaufman M. Salesforce.com for Dummies. Wiley. com, 2010
14 Tsai W T, Xiao B N, Chen Y N, et al. Consumer-centric service-oriented architecture: a new approach. In: Proceedings of SEUS-WCCIA, Gyeongju, 2006. 175–180

15 Tsai W T, Huang Y, Bai X Y. Grapevine model for template recommendation and generation in SaaS applications. In: Proceedings of the 3rd Asia-Pacific Symposium on Internetware, Tempe, 2011
16 Huang A. Similarity measures for text document clustering. In: Proceedings of the 6th New Zealand Computer Science Research Student Conference, Christchurch, 2008. 49–56
17 Goldberg D, Nichols D, Oki B M, et al. Using collaborative filtering to weave an information tapestry. Commun ACM, 1992, 35: 61–70
18 Lang K. Newsweeder: learning to filter netnews. In: Proceedings of the 12th International Conference on Machine Learning, Take Tahoe, 1995. 331–339
19 Mooney R J, Roy L. Content-based book recommending using learning for text categorization. In: Proceedings of ACM Conference on Digital libraries, New York, 2000. 195–204
20 Miranda T, Claypool M, Gokhale A, et al. Combining content-based and collaborative filters in an online newspaper. In: Proceedings of ACM SIGIR Workshop on Recommender Systems, Berkeley, 1999. 60
21 Melville P, Mooney R J, Nagarajan R. Content-boosted collaborative filtering for improved recommendations. In: Proceedings of the National Conference on Artificial Intelligence, Alberta, 2002. 187–192
22 Schein A I, Popescul A, Ungar L H, et al. Methods and metrics for cold-start recommendations. In: Proceedings of ACM Conference on Research and Development in Information Retrieval, New York, 2002. 253–260
23 Su X, Khoshgoftaar T M. A survey of collaborative filtering techniques. Adv Artif Intel, 2009, 2009: 4
24 Tsai W T, Shao Q H, Huang Y, et al. Towards a scalable and robust multi-tenancy SaaS. In: Proceedings of the 2nd Asia-Pacific Symposium on Internetware, New York, 2010
25 Roe C, Gonik S. Server-side design principles for scalable internet systems. IEEE Softw, 2002, 19: 34–41
26 Bai X Y, Li M Y, Chen B, et al. Cloud testing tools. In: Proceedings of IEEE 6th International Symposium on Service Oriented System Engineering, Irvine, 2011. 1–12
27 Yu L, Tsai W T, Chen X J, et al. Testing as a service over cloud. In: Proceedings of IEEE International Symposium on Service Oriented System Engineering, Nanjing, 2010. 181–188
28 Gao J, Bai X Y, Tsai W T. Cloud-testing: issues, challenges, needs and practice. Softw Eng, 2011, 1: 9–23
29 Tsai W T, Li W, Sarjoughian H, et al. SimSaaS: simulation software-as-a-service. In: Proceedings of the 44th Annual Simulation Symposium, 2011, San Diego, 77–86
30 Tsai W T, Huang Y, Shao Q. Testing the scalability of SaaS applications. In: Proceedings of IEEE International Conference on Service-Oriented Computing and Applications, Irvine, 2011. 1–4