

WHITE PAPER

# Right Engineering SaaS: Successfully deploying Software-as-a-Service Models

by Karthik Viswanathan

If you think building enterprise systems is complex, then building SaaS products can be even more complex, simply because you have to service multiple organizations, each one bringing its own idiosyncratic requirements, and of course, the inevitable Service Level Agreements (SLAs). Essentially, engineering SaaS products is not all that....

# Table of Contents

1. Introduction.....	2
2. Architectural Strategies for SaaS.....	2
2.1. Composition.....	2
2.2. Service Oriented Architecture (SOA).....	3
2.3. Compartmentalization (C18N).....	3
2.3.1. Distributed Responsibility.....	3
2.3.2. Trust Partitioning.....	3
2.4. Metadata driven architecture.....	4
2.5. Technology platform: Depth vs Breadth.....	5
3. Best Practices.....	5
3.1. Aspect Oriented Programming (AOP).....	5
3.2. Declarative Programming.....	5
3.3. Customization through Configuration.....	6
3.4. Plug-in Pattern.....	6
4. Database 2.0 for Web 2.0.....	6
4.1. Views.....	6
4.2. Horizontal Partitioning.....	7
4.3. Row-level-security.....	7
4.4. Service-Oriented Data Architecture (SODA).....	7
5. Application Tenancy and Virtual Appliance.....	7
6. Challenges and Pitfalls of SaaS adoptionrtual Appliance.....	8
7. References.....	9
8. About Aspire.....	9



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

## 1. Introduction

If you think building enterprise systems is complex, then building SaaS products can be even more complex, simply because you have to service multiple organizations, each one bringing its own idiosyncratic requirements, and of course, the inevitable Service Level Agreements (SLAs).

Essentially, engineering SaaS products is not all that simple and it typically takes longer to develop an on-demand product.

Hence, one needs to be really sure about the engineering aspects of moving to SaaS before making the transition. Remember, SaaS isn't shrink-wrapped desktop software and it demands a unique architecture to be easily customizable, and extensive user interface design to allow users to successfully adapt to a hosted SaaS application.

Tomcat, MySQL, XML, Struts 2.0, and Java/JSP are all it takes to build a SaaS solution using Java technology. However, several 'fashionable' tools and techniques are touted as the foundation for building SaaS products. While many of them are indeed valuable companions in the SaaS journey, a provider may fall into the trap of choosing some of these just because they are white-hot buzzwords. Naturally, incorrect architecture/design decisions will lead to short-term crisis or long-term pain or both.

The purpose of this paper is to put forth some fundamental elements that can be used to achieve the goal of right-engineering your SaaS product. It consolidates the technology strategies to build on-demand products and is the result of helping several ISVs transition to an on-demand environment. Of course, these technical guidelines are not specifically developed for SaaS; in fact, some of them have been around for several years and are widely used today to build enterprise products. This paper should be able to provide you with a good starting point for your engineering strategies to achieve SaaS.

## 2. Architectural Strategies for SaaS

### 2.1. Composition

It is a way of developing applications by assembling them from prebuilt components and modules, instead of building them from scratch. These prebuilt components are typically the software assets that an organization has already accumulated – eg: Web Service APIs, Business rules, Workflows and Reports.

Composite applications are both, an architectural style for developing applications, as well as a form of integration. Such applications are designed to support a company's business processes and to map them to underlying resources.

Enterprises of all shapes and sizes are increasingly bypassing the traditional styles of synchronizing data between closed silos, such as ETL (Extract, Transform, Load). These approaches require huge investments, process definition, and custom application logic to deliver real value. Instead, organizations are now adopting composite applications to provide real-time, on-demand integration via the user interface. Also known as "mashups," these new classes of solutions are light-weight in nature and generally have minimal, if any, data movement among systems. Their goal is to provide consolidated, cross-silo information to the end user.

Composition is a natural add-on for SaaS applications. Many organizations do not move critical internal data to an external SaaS provider's database, which means that the only way for the user of a SaaS system to see this data is to view it on screen at real-time. Thus, the composite application often simply occupies some screen real estate in the SaaS UI. Moreover, an ISV transitioning from on-premise to SaaS can significantly leverage the existing investments in software by adopting the principle of composition.



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

## 2.2. Service Oriented Architecture (SOA)

It is a technology architecture philosophy for building systems based on the interaction of loosely coupled, coarse-grained autonomous software units called Services. A service can be understood as the realization of self-contained business functionality in the form of software.

When organizations decouple their services from the silos that an application focus has placed them in, they realize the intrinsic value of the service and its relationship to their business processes. The value of an application does not become less, but the nature of an application changes to one where services form the core of the undertaking. With that in mind, it becomes easier for Software as a Service (SaaS) to become a reality. An organization already making use of services finds it simple enough to integrate an external service. The reverse is also true. An organization using services finds it easier to expose its own services to external customers - in essence to take its services to market by providing Software as a Service.

Thus, the key principles of SOA, such as alignment between business and technology, loose coupling, composition, increased interoperability and federation make it an ideal foundation for a SaaS product.

## 2.3. Compartmentalization (C18N)

It is the technique of separating the parts of a system so that the failure in one subsystem has minimal ripple effect on the other subsystems. Large mission-critical applications rely on this principle to avoid complete breakdown of the system. As Service Level Agreements (SLAs) assume greater significance in on-demand environments, localization of failure is an important aspect of the product architecture.

C18N in a SaaS product can be achieved using the time-tested concepts of 'Abstraction' and 'Separation of Concerns'. Abstraction is a fundamental principle to cope with software complexity and its primary goal is to strengthen the essential and eliminate the irrelevant. Finding the right level of abstraction in software architecture is as much an art as it is science. The principle of Separation of concerns states that a given software engineering problem involves different kinds of concerns (properties), which should be identified and separated to cope with complexity, and to achieve desirable qualities such as robustness, adaptability and maintainability. Thus, an architecture based on highly cohesive and loosely coupled subsystems is a vital ingredient to build robust SaaS products.

As on-demand functionalities are delivered over the internet, security is an important consideration. C18N can be effectively utilized to develop defensive architectures that can guard against malicious attacks. For example, in order to protect one part of the system from a security failure in another, we can put each part in a separate security domain. There are also a few architectural patterns available to deal with such scenarios:

### 2.3.1 Distributed Responsibility

A security failure in a compartment can change any data in that compartment. A compartment has both an interface that is at risk of a security failure, and data that needs to be secure. How can this be achieved? Partition responsibility across components, such that, the components that are likely to fail do not have critical data. In other words, assign responsibilities in such a way that several of them need to fail in order for the whole system to fail.

### 2.3.2 Trust Partitioning

In a system architecture that is compartmentalized, an intruder may get hold of a compartment with super user rights or a compartment with maximum responsibilities and crash the whole system. How do you ensure



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

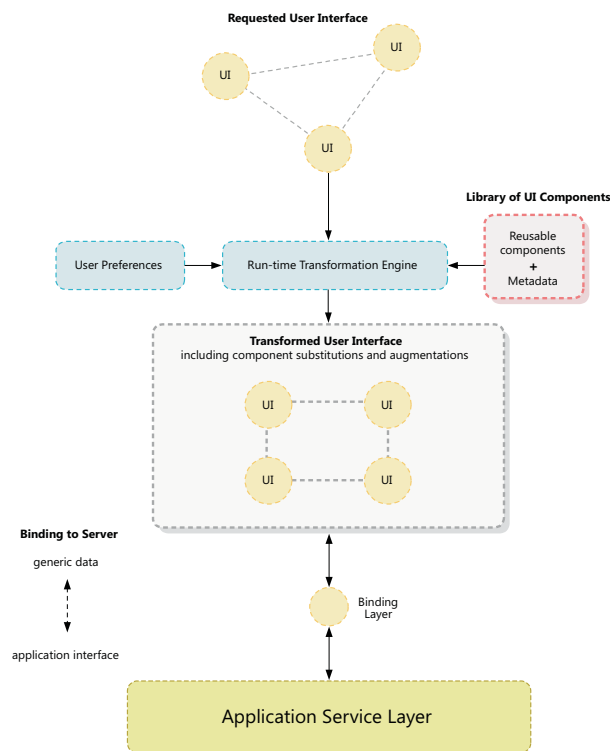
that the entire application is not compromised? Assign minimum privilege level to components according to the least privilege principle. Classify the owners of processes into different trusted and un-trusted groups. Design the components not to trust inputs from other groups and to validate inputs.

## 2.4 Metadata driven architecture

Metadata consists of information that characterizes the resource or data. A common example of metadata is the database schema. The schema does not show the actual data – it shows the definition of the data and entity relationships. An enterprise' metadata defines the enterprise – what the enterprise does and how it does it. Metadata can be used to represent workflow, activities, policies, logic and data. Common types of metadata are business, technical, operational and environmental metadata, that are usually stored in xml formats or in the database. The key advantages of building meta-driven-products are reusability, consistency and faster time-to-value.

The metadata mindset is fundamental to building on-demand products. All the key characteristics of SaaS – multi tenancy, configurability, security and integration can be achieved by basing the product architecture on metadata. Storing UI layout, client-side validations, properties of various controls such as visibility in a metadata repository and interpreting them at run-time will not only allow the product to display disparate behaviors but also handle presentation in various devices such as browsers, PDAs and smart phones.

As an example, consider the open source UI framework called 'Fluid', which is currently under development. Its focus is on layout, sizing, navigation, controls and tabbing schemes. The goal of the Fluid architecture is to deliver a framework that supports configuration time customization and runtime transformation. It uses metadata to define columns, current state, expandability/collapsibility, and primary content Vs. informational content Vs. error messages, enabled/disabled and so on.



**Meta-data driven UI Modeling**



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

UI modeled on the lines of such design philosophies is an ideal fit for the on-demand world even if such frameworks are not directly used off-the-shelf.

Meta driven application/data integration is also gaining widespread acceptance. If done correctly, a metadata-based approach will result in loosely coupled applications that reuse the various interface definitions and messages. A traditional approach, such as ETL, moves data from one repository to another, while meta-driven-integration creates integration hooks that allow applications to use data from sources where they already reside. Syntax information like field lengths and tags, validation rules, transformation rules and routing information are some of the key metadata required for integrating applications with this approach. Likewise, policy-based access control and enforcement of regulations like Sarbanes-Oxley have increased the adoption of metadata for implementing security.

Finally, the foundation of the flexibility in SOA is metadata. The three standards of SOA – SOAP, WSDL and UDDI are all technical metadata, enabling services to abstract the complexity of the source systems and making them reusable.

## 2.5 Technology platform: Depth vs Breadth

Traditional on-premise products are installed separately at each customer location and hence should have the capability to run in diverse environments and technology platforms. They should work consistently with different operating systems and databases. Hence, some of the unique or advanced features of their technology portfolio may not be used. The law of lowest common denominator takes precedence with its focus on wider agreement and compromise solutions. For instance, such products typically tend to use only the standard SQL statements that are commonly supported across databases. On the other hand, in an on-demand scenario, the engineering team has the advantage of leveraging the best features of their chosen technology platform because the software is hosted centrally. As examples, the Virtual Private Database (VPD) feature of Oracle and row-level-access feature of DB2 are excellent candidates for the SaaS model.

## 3. Best Practices

### 3.1 Aspect Oriented Programming (AOP)

Many programs and components in a system require features that do not fit in naturally within its scheme of things. Common examples include context-sensitive validations, error handling, and performance optimizations apart from the clichéd example of application logging. AOP terms such features ‘concerns’ and calls them ‘crosscutting’ because such behaviors are necessary in many areas of a software product. Using AOP, we can abstract (centralize) all these concerns and apply them at run-time in a context-sensitive way. To put it simply, AOP does to server side code what Cascading Style Sheet (CSS) does to HTML. Thus, AOP complements Object Oriented Programming by introducing another level of modularity. Its contribution towards the maintainability, adaptability and reusability of code makes it an ideal choice for on-demand products. AOP can be used to implement some of the basic requirements of a SaaS product such as monitoring, security and caching.

### 3.2 Declarative Programming

If a system has workflows embedded into the application, a lot of embedded SQL statements and quite a bit of UI text entangled in code, chances are that the paradigm of declarative programming is not being effectively used. Fundamentally, in declarative programming, ‘what’ needs to be achieved is mentioned in XML, text file and database while the ‘how’ is left to the system. For example, Java 5 added a feature called annotation, which is a mechanism for associating the declarative information with classes. Struts 2 has



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

built-in support for annotations and it simplifies, for example, the validation of user input. Annotations can be used for various purposes like transaction, security and dependency injection. They reduce the boilerplate code and results in elegant products that are easy to maintain in the long run. While they can be used for any kind of software development, it will not be an exaggeration to say that they are mandatory for SaaS products.

### 3.3 Customization through Configuration

An on-demand product should be customizable for each tenant. Some products allow minor customizations such as the look and feel, while others permit the business and database layers also to be tailored. However, the key challenge is to make this happen without doing code changes to reduce the go-to-value timeframe. It is widely known that installations of some on-premise products at a customer location can take up to several months. While SaaS promises to change that, it can be achieved only if the product polymorphism is achieved in a short period.

At the heart of its middle tier, a SaaS product should have an engine that triggers different workflows based on custom business rules. Rules typically detect the occurrence of a business scenario and raise a business event, which in turn triggers the execution of a workflow. The recommended approach is to have a light-weight open source rules engine and a workflow component. Even if a home-grown solution is used to achieve this functionality, the best practice is to have a clear separation between rules and workflows as tight coupling reduces their reusability – within a tenant or across tenants.

### 3.4 Plug-in Pattern

Usually, the user will want added functionality after a system is delivered. Moreover, in SaaS, users belonging to different tenants will require custom functionalities based on their specific needs. In order to accommodate these situations without a major re-write, a framework that allows for future additions of modules without breaking the existing code base needs to be implemented. A plug-in strategy is a good fit for such extensibility needs. Plug-in based frameworks will allow a program to "look for" add-in functionality at startup, and then allow that plug-in to cooperate with it. Many applications, such as Microsoft Office, currently use similar technologies to allow third-party developers to integrate with their existing applications to add functionality or robustness, otherwise missing from the application. While this approach applies to on-premise systems too, various flavors of SaaS (single-instance or multi-instance) products will also benefit by using this design pattern.

## 4. Database 2.0 for Web 2.0

While the software industry continues to move towards newer technological innovations, majority of the web-based applications continue to under-utilize their relational databases. It is rather commonly accepted (although some opposition exists) that business logic should not be written in the database layer. But that should not prevent one from tapping the true power of databases. For example, it is not true that stored procedures are inappropriate for SaaS products. Many traditional database features can be used in SaaS if they are found appropriate. Here are some common ways to leverage RDBMS investments effectively in an on-demand world.

### 4.1 Views

They can be used to secure the access of data from underlying tables. The database queries or stored procedures will use these views to access tenant specific data rather than accessing the tables themselves. In a multi tenant database, each tenant can be given permissions to use the views without giving direct access



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

to the tables.

## 4.2 Horizontal Partitioning

In this federated model, a data entity is split and stored in multiple database servers. This strategy, also known as sharding, is becoming very popular. With hardware costs coming down, this approach can be used to build scalable products. From a SaaS perspective, this means that data for different tenants can be stored on different server instances. Though sharding is not perfect, it offers higher availability and faster queries.

## 4.3 Row-level-security

Ensuring appropriate information security is a pressing concern for many businesses today, given privacy legislations such as the United States' HIPAA (Health Insurance Portability and Accountability Act), Gramm-Leach-Bliley Act, Sarbanes-Oxley Act, and the EU's Safe Harbour Law. Since on-demand businesses have the additional responsibility of ensuring security even when the data is lying outside the organization, we can leverage a few interesting vendor-specific features of databases.

Oracle's Virtual Private Database (VPD) is one such feature. VPD works by transparently modifying incoming requests to present a partial view of the tables based on a set of defined criteria. During runtime, predicates are appended to the queries to fetch only the rows that a user is supposed to see. For example, if the user is supposed to see only accounts of the manager XYZ, the VPD setup automatically rewrites the query "select \* from accounts" to "select \* from accounts where mgr='XYZ'". This is achieved by having a security policy on the accounts table. Since the access control is based on policies, these can be changed when necessary without the need to modify the application code. Oracle Label Security (OLS) is another feature that builds on VPD and supports row-based-security for restricting user access to specific data.

The Multi Level Security (MLS) feature of DB2 will also be useful to implement such fine-grained access control. Resource Access Control Facility (RACF) is used to assign security labels to resources (eg: tables/rows) and user profiles. Row level security can be implemented by adding a 'SECURITY LABEL' column to the tables. When a row is accessed, MLS prevents access if the label in the user profile does not dominate the label in the row.

## 4.4 Service-Oriented Data Architecture (SODA)

Inspired by Service oriented Architecture, SODA prescribes that the databases be partitioned according to well-defined service boundaries. At a logical level, a database service exposes a well-documented interface to data. This is not a general database interface for reading and writing data, instead it provides very specific functionality. For example, an inventory database service might expose methods for checking inventory levels, reserving inventory, removing products from inventory and so on. While this may sound similar to stored procedures, what is different is that access to data under the control of one database service is completely isolated from that of a different database service. Moreover, requests to database services are exposed as Web services (instead of a database connection).

## 5. Application Tenancy and Virtual Appliance

Application tenancy refers to the ability of a software product to offer its features to one or more client organizations or 'tenants'. A key architectural foundation for the success of any SaaS solution is the choice of tenancy i.e. multi-instance-single-tenancy, single-instance-multi-tenancy or an intermediate Virtual Appliance approach. This decision will have an impact on all the tiers (UI, Business and the Data) of the application. Some of the factors that influence this decision are time-to-market, business domain of the product, existing



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

technological investments and technical complexity.

The key considerations for the single-instance and multi-instance tenancy have been widely documented. Though multi-tenancy has almost become a synonym for SaaS, it is not without its own set of challenges, as outlined below.

- Ⓐ **Re-architecture:** A traditional on-premise application needs a major re-design/re-architecture to become a multi-tenant SaaS application. This major endeavor involves significant investments of money and time.
- Ⓑ **Hosting:** Hosting for multi-tenant architectures needs a lot of preparatory work and that is why not all hosting providers can be a good fit. The database configurations, the shared infrastructure and other related issues can make it complicated to find the right hosting partner.
- Ⓒ **Availability:** In a single-instance application, there is always the danger that a single accidental /malicious user operation can bring down all the tenants.

An emerging alternative to multi-tenancy is the usage of virtual appliances that are designed to eliminate the installation and maintenance costs associated with running complex software stacks. A virtual appliance is an application combined with Just Enough Operating System (JeOS) that readily installs on industry standard hardware or inside a virtual machine. JeOS refers to an operating system (eg: Linux) customized to precisely fit the need of a particular application. In a nutshell, a single-tenant software product can be 'packaged' as an appliance and deployed for multiple tenants without the need for redesigning the entire product.

A virtual appliance is usually built to host a single application, and so represents a new way of deploying network applications. Furthermore, in contrast to the multi-tenancy approaches to SaaS, a virtual appliance can also be deployed on-premises for customers that need local network access to the running application, or have security requirements that a third-party hosting model does not meet. The underlying virtualization technology also allows for rapid movement of virtual appliances instances between physical execution environments whereas traditional approaches to SaaS fix the application in one place on the hosted infrastructure. But the downside is that the technology is still evolving though there are a few key players in the market. Virtual appliance may not be the replacement for single-instance multi-tenancy but it is an option to be considered (at least as a medium-term solution), in case the effort and time required for building a multi-tenant product is preventing you from entering the on-demand world.

## 6. Challenges and Pitfalls of SaaS adoption

- Ⓐ **Giant Leaps:** An Independent Software Vendor (ISV) should not adopt a big bang approach in rolling out the SaaS version. If you are a client-server product vendor, you should first think about web-enablement and then move to SaaS. Another big bang approach to be avoided is the multi-year waterfall model for software development. Iterative and agile methodologies are naturally highly recommended.
- Ⓑ **Customization cliff:** Building highly customizable products is a complex activity. Even though 'Customization through Configuration' is identified as a best practice, it remains a Holy Grail for the technical teams. It is equally difficult to maintain multiple versions of your software that are highly customized for each tenant. There is no magic wand available and hence you should bite only what you can chew.



# Right Engineering SaaS : Successfully deploying Software-as-a-Service Models

- **Designing with buzzwords:** SaaS is not 'SOA, AJAX As a Service'. In fact, we have helped ISVs build database-driven multi-tenant systems without using SOA or middleware products. So, it is important to consider SaaS to be 'Service as a Software' and make the appropriate choice of architecture, technology platform and toolkits.
- **Anti-Patterns:** Building on-demand products just like traditional on-premise systems; not starting with xml/metadata as a key foundation; or, overdoing the usage of metadata thus increasing the application complexity; lack of clear understanding of the performance costs associated with loose coupling are some of the common anti-patterns of building applications in the cloud.
- **Buying SaaS:** SaaS is not a product that you can buy; rather an ISV should transition to the SaaS model with support from the business team, technology partners and hosting providers. Unfortunately, SaaS is a journey that has few shortcuts and it is important to be wary of quick fixes and short enablement programs. Instead, right-engineering your product can go a long way towards making your Software-as-a-Service truly successful.

## References:

1. Architectural Strategies for SaaS - <http://hillside.net/patterns>
2. Metadata driven architecture - <http://wiki.fluidproject.org/display/fluid/User+Interface+Adaptation+and+User+Preferences>
3. Service-Oriented Data Architecture (SODA) - <http://www.soamag.com/l2/1106-3.asp>

## ABOUT ASPIRE SYSTEMS

Aspire Systems is an Outsourced Product Development firm committed to helping our customers build software products better and faster. We work with some of the world's most innovative Independent Software Vendors and software-enabled businesses, ranging from start-ups to established industry leaders, transforming the way software is built.

Aspire provides complete product lifecycle services, ranging from new product development and product advancement to product migration, re-engineering, sustenance and support. Our product development teams are spread between our Global Innovation Center in Chennai, India and offices in the United States.



Aspire Systems India Private Limited  
Plot No 1/D-1, SIPCOT IT PARK, Siruseri, Tamil Nadu - 603 103  
Tel : +91-44-67404000. Fax: +91-44-67404234  
E-mail : [info@aspire.sys.com](mailto:info@aspire.sys.com)  
Web: [www.aspiresys.com](http://www.aspiresys.com)