# Simplifying SaaS/Multi-Tenant Application Development

Part 1

*A www.techcello.com initiative*

# Table of Contents

**1. Introduction ........ 1**

**2. On-Premise Model and its challenges ........ 2**

2.1 Customer's perspective of challenges towards in-premise model ........ 3

2.2 ISV's perspective of challenges towards in-premise model ........ 4

**3. Software as a Service Model ........ 5**

3.1 SaaS Definition ........ 6

3.1.1 ISV Perspective ........ 6

3.1.2 Customer Perspective ........ 7

3.2 SaaS – the Big Impact ........ 8

3.2.1 Business Model ........ 8

3.2.1.1 Long Tail 9

3.2.2 Product Architecture ........ 9

3.2.2.1 Current state of the product 9

3.2.2.2 SaaS Maturity model 9

3.2.3 Operational Structure ........ 10

3.2.3.1 Optimized usage of support bandwidth 10

3.2.3.2 Self-serving system 10

3.3 SaaS expectations from a customer prospective & ISV prospective ........ 11

3.3.1 SaaS expectations from a customer perspective ........ 11

3.3.2 SaaS expectation from an ISV perspective ........ 11

**4. Multi Tenancy Basics ........ 12**

4.1 Major Architectural models / patterns for building SaaS product ........ 12

4.2 Maturity level factors ........ 14

4.2.1 Operational Efficiency ........ 14

4.2.2 Maintenance efforts ........ 14

4.2.3 Scalability ........ 14

4.2.4 Time ........ 15

4.2.5 Engineering skill set ........ 15

4.3 Multi-Tenancy Decision ........ 15

4.4 Multi-Tenancy Degree ........ 16

4.5 Multi-Tenancy without SaaS ........ 17

**5. Technical Stack of Multi-Tenant SaaS Product ........ 17**

5.1 Non-Functional Requirements of a multi-tenant/SaaS product ........ 18

5.1.1 Security ........ 19

5.1.2 Scalability ........ 20

5.1.3 Performance ........ 20

5.1.4 Availability ........ 20

5.1.5 Integration ........ 21

5.1.6 Extensibility ........ 21

5.1.7 Configurability ........ 21

5.1.8 Auditing ..... 21
5.2 Plumbing Features ..... 22
5.2.1 Exception Management ..... 22
5.2.2 Instrumentation ..... 22
5.2.3 Caching & Distributed Caching ..... 22
Cache Policy 23
Distributed Scenario 24
5.2.4 Data Access Management ..... 24
5.2.5 Service Injection ..... 25
5.2.6 Policy Injection ..... 25
5.3 SaaS Engineering Features ..... 25
5.3.1 Data connection & Abstraction ..... 26
5.3.2 Authentication ..... 27
5.3.3 Authorization ..... 29
5.3.4 Data Security ..... 30
5.3.5 Query Generator ..... 31
5.3.6 Notifications ..... 31
5.3.7 Audit Trails ..... 31
5.3.8 Scheduling ..... 32
5.3.9 Workflow ..... 32
5.3.10 Business rules ..... 33
5.4 SaaS Operational Features ..... 34
5.4.1 Tenant Management ..... 34
5.4.2 Subscription Management ..... 35
5.4.3 Billing ..... 36
5.4.4 Customization ..... 36
5.4.5 Data Management Utilities ..... 38

**6. Migrating an existing application to SaaS ..... 39**

6.1 Product Engineering Considerations ..... 39
6.2 Migration Approach ..... 40
6.3 Engineering Process ..... 42

**7. SaaS Deployment Management ..... 43**

7.1 Self Hosting (With / Without Virtualization) ..... 43
7.2 IaaS – Infrastructure As A Service ..... 44
7.3 iPaaS – Infrastructure PaaS ..... 44
7.4 Migration PaaS ..... 45
7.5 Cloud Deployment Considerations ..... 45

**8. Summary of Part 1 ..... 49**
**9. Brief Introduction to Part 2: Advanced Topics ..... 50**

# 1. Introduction

Software as a Service (SaaS) is a paradigm shift in the way software is developed, purchased and used. “Pay only for what you use” mantra is widely accepted worldwide and companies like SalesForce.com, SuccessFactors.com, etc. thrive on this principle. Customers (end users) stand to gain a lot of benefits – no upfront investment - CAPEX, better management of cash flow as OPEX, efficient management of users/utilization of the application, etc. It is being increasingly adopted by both software buyers and software makers.

SaaS (also known as On-demand solutions or hosted software solution) is a software application delivery model where a vendor develops software and hosts/operates (independently or through a 3rd-party) over the internet. Software users pay the vendors on a “pay as you go” basis, usually a monthly or yearly subscription fee. In this model, the software users can access the software from anywhere over the Internet.

This hosted software solution is adopted in line of business services like CRM, HR, Payroll processing, etc. Consumer related services like Google Docs, Web-based-mails, and online banking are also on-demand solutions. SaaS has gained acceptance not only among the non-critical business processes like above, but some of the software products that service critical business functions (e.g. Hospital management systems) are also considering SaaS more actively for some of their product lines.

**What makes SaaS more desirable?**

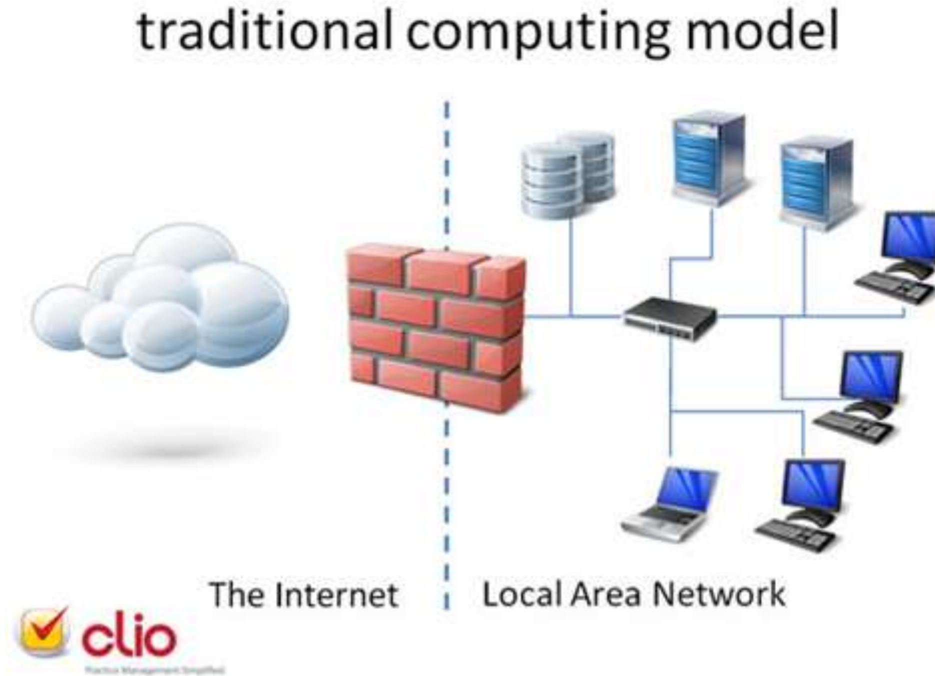
Both the software users and software vendors would like to be associated with on-demand solutions and there are several good reasons behind this trend. Some key attractions causing this increased adoption are highlighted below.

| For Software Vendors | For Software Buyers |
| --- | --- |
| - Single instance of the product services all the customers<br>- Upgrade and maintenance are one step<br>- Deployment process is simple and consumes less time and cost<br>- Customer acquisition is faster and easier; Shorter sales cycle<br>- Recurring predictable monthly revenue<br>- Access to the long tail | - Zero CapEx and pay as you go pricing structure helps manage cash flow better<br>- Add users as you go<br>- No additional hardware needed<br>- No initial installation and implementation struggle<br>- Less or no need for internal IT resources<br>- Easy-to-use, mostly very intuitive and less training requirements |

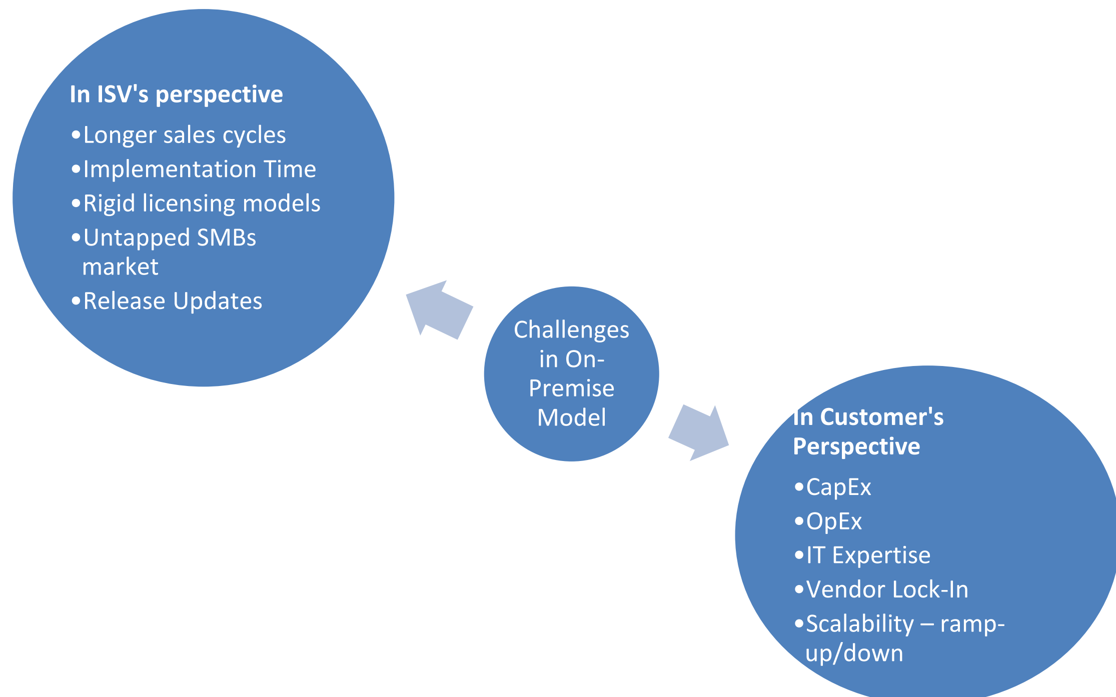
This multi-tenant architecture and its applicability are widely used in large, multinational enterprises as well. These enterprises may want to build an in-house HR system (or) shipping system to manage multiple business divisions across the world. IN In the current trend, SaaS / Multi-tenancy / Cloud based application are used interchangeably. While real meaning of each would differ, from the business value generation perspective, each refers the same as the end point. This ebook is aimed at providing insights into building a SaaS / multi-tenant application. It will help the readers get good perspective on the essential components of a SaaS application development and deployment.

## 2. On-Premise Model and its challenges

An on-premise model typically indicates a situation where the software gets installed inside the premises of the end user organization.




Challenges in the on-premise model can be viewed from two different perspectives: Customer's perspective and ISV's perspective.



## 2.1 Customer's perspective of challenges towards in-premise model

***'Pay-as-you-go' is the new 'mantra' for customers too, who want to leverage on acquiring on-demand SaaS solutions rather than owning dedicated, expensive IT applications.***

Starting with the customer's perspective, challenges here include aspects such as CapEx, OpEx, IT Expertise, Vendor Lock-In, and Scalability – ramp-up/down.

1. **CapEx:** This indicates a capital expenditure that an organization has to incur (such as license for the software bought) in order to procure the software. Hence, in an on-premise model, the licensing model tends to become a continuous, perpetual licensing situation. Cost of the software is typically high. This is because, when the ISV sells the software to the customer/organization, they apparently accommodate all their production cost in the selling price of the software. This cost invariably tends to become a lump sum amount for the customer/organization to invest in procuring and using the software.
2. **OpEx:** OpEx is ideally the Operational Expenditure. It is the expenditure that is related to maintaining the software brought by the customer/organization. Such costs include expenditure such as the maintenance charges that ISV may levy for the software or for the servers the ISV may be maintaining in the customers'/organization's premises.

***Note**: CapEx and OpEx are set of ongoing expenses that customers/organizations have to spend further on top of the initial expenditure that they had invested for the software. It has to be noted that there are customers/organizations that can afford this level of expenditure and also the ones who are not inclined towards this kind of expenditure.*

3. **IT Expertise:** Apart from the above-mentioned costs, IT expertise forms a vital aspect that may form a challenge for customers/organizations in an On-Premise model. For instance, let us take into consideration a Pharmacy run by non-IT personnel. Such personnel require the aid of IT expertise personnel who can resolve the issues that they may encounter in their servers, hardware/software. There may be requirements of data backup, server management, etc. Not all customers/organizations can actually afford an IT expertise to this level.
4. **Vendor Lock-In:** During many instances, customers/organizations undertake a long process of evaluating software from a vendor (ISV) and finally decide on acquiring and using the same. Given the pace of changes that happen in today's business scenario, the parameters that are used for evaluation might change or even become obsolete. This is another typical situation in On-Premise model, where the customer/organization is tied up to the software that they have bought. They would have already paid for the software for a certain period of time. Hence, the customer/organization, having made an investment on the software for a substantial period of time would not wish to let go waste the previous payments and is hence tied-up with the only option of using the software.
5. **Scalability:** Scalability is another common challenge that customers/organizations often encounter. Some of the standard licensing/standard mechanisms for software brought from

vendors (ISVs) work on number of users. For instance, if a customer/organization buys software for 50 users and is actually requiring the license for 30 users; or otherwise, the customer/organization may need the license for 60 users but for a shorter period of time. Such situations turn out in such a way that they are not properly scalable or are inflexible for the customer/organization in terms of scalability.

Customers have started looking for a better delivery model where they don't have to invest upfront, don't get tied down to the vendor and also continuously get better service from the ISVs.

## 2.2 ISV's perspective of challenges towards in-premise model

***According to conventional wisdom, the SaaS model fits best for SMBs. However, recent economic scenarios have made SaaS the attractive option to even larger organizations.***

Challenges in the software Vendor (ISV)'s perspective towards on-premise model include long sales cycle, implementation time, rigid licensing models, untapped SMBs market and release updates.

1. **Long Sales Cycle:** Organizations generally take long time in the decision making process of whether to buy a software or not. Vendor lock-in could also partly be the reason for this delay, adding more pressure on the decision makers in making the right decision. Typically sales cycle for an ISV is 12 to 18 months to take decision whether to buy it or not. A long sales cycle means that ISV's are spending more money in terms on assessing customers throughout the sales cycle period, which will in turn increase the price of the software.
2. **Implementation Time:** This is another challenge where ISVs typically encounter in the On-Premise model. With the mindset of heavy investment being made, customer sat times may go overboard in customizing the software to be tailored to their needs or to exactly match their organizational requirements. This results in longer cycles for the ISV to implement the software for the customer/organization. On an average, in an on-premise model it takes 6 to 9 months for an ISV to implement the software for the customer/organization.
3. **Rigid Licensing Models:** Most ISV's in general do not display flexibility in their model or framework to allow customers/organizations to pick and choose the licensing model. ISV's do not offer any variety in licensing where customers/organization scan pay for just parts of the software they would be using or pay for a limited period of time they would be using the software.
4. **SMB market untapped:** In today's world, Small and Medium businesses (SMBs) is a huge market that is in need of cutting edge software to increase their operational efficiencies. However, for most SMBs, the CapEx investment in buying the software/hardware upfront in itself is a huge cost, which they are not ready to invest. The inability to tap the SMB market is another huge challenge in the On-Premise model.
5. **Release Updates:** Software are bound to get updated over a period of time in order to take care of new technologies and solutions. However, while customers understand the advantages of the new technologies many a times they tend to stay with the version they currently have to avoid the hassles of going through the UAT process again. This leads to a

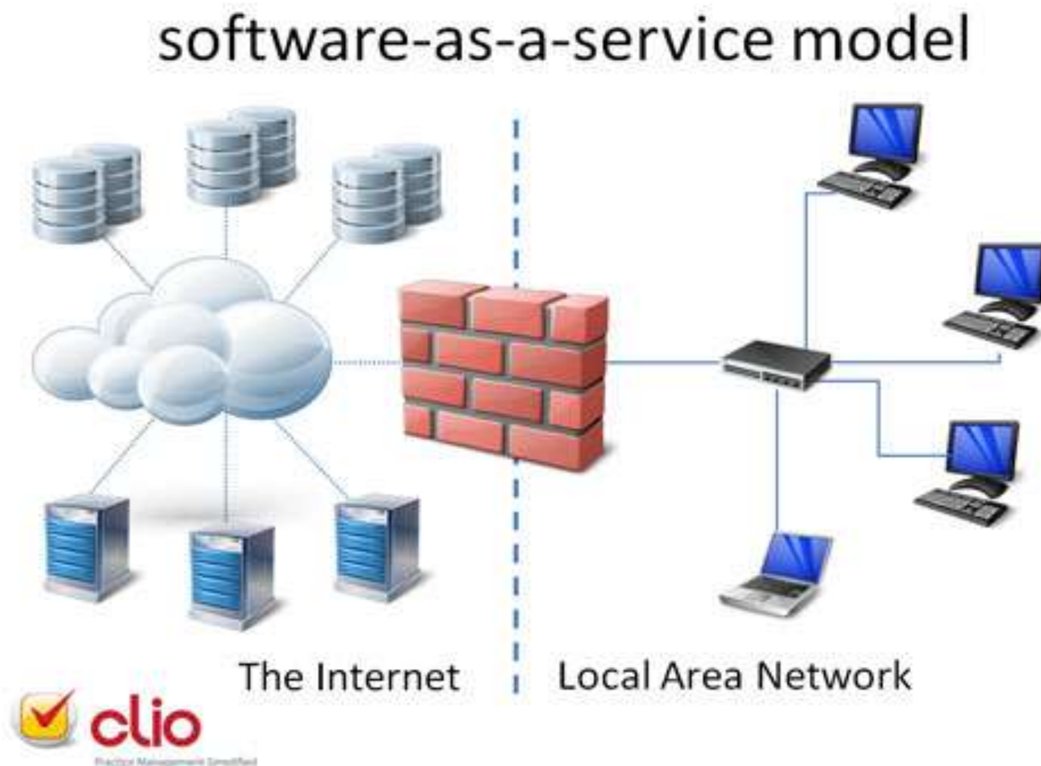
situation where the ISVs end up maintaining/supporting multiple versions of the software, which increases the operational cost.

These challenges make the ISVs think about a different business model where they can overcome these and also attract a larger set of customers. SaaS business model lends itself to overcome these and also help ISVs to grow their business. It is a Win-win situation for both customers and ISVs.

## 3. Software as a Service Model

***Systems based on SaaS model are marked by their subscription policy based 'pay-for-what-you-use' pricing. The 'icing in the cake' is their web-based interfaces coupled with room for building upgrades easily into their subscription pricing.***

In a SaaS model, the software portion lies outside the physical control of the organization. In fact, the organization will not even be worried about the Nitti gritty of how the software is deployed and managed. Users of the organization will access the software through the internet.



## 3.1 SaaS Definition

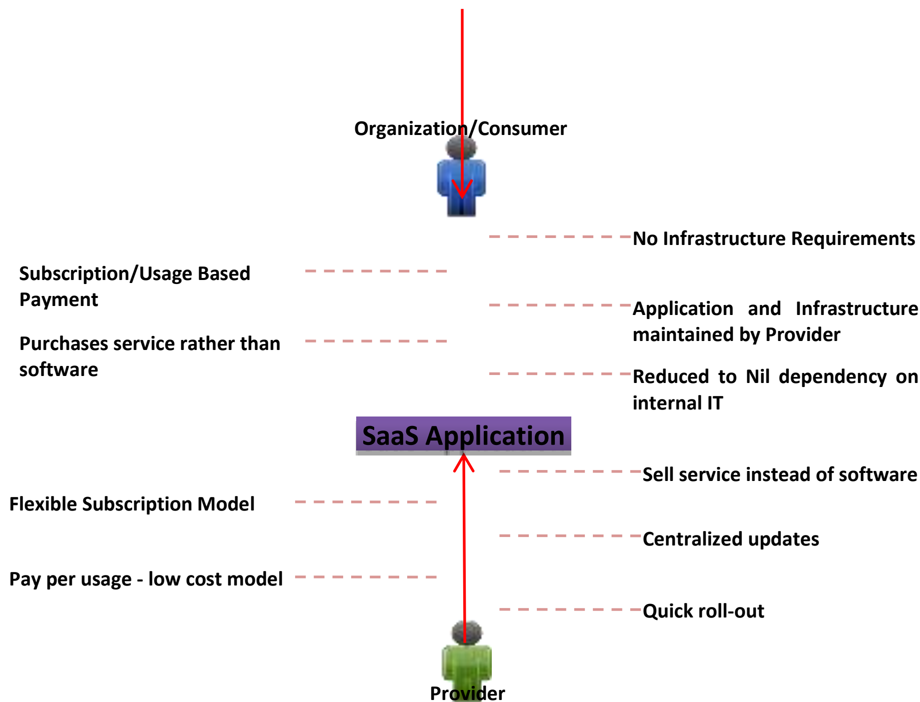
A very close example to quote prior to defining SaaS is that of renting a car rather than buying one. In SaaS, organizations buy the services of the software rather than buying the software itself.

***Often described as the 'on-demand-future-technology', SaaS is a delivery model for Software applications whereby Customers can access all or required (subscribed) parts of the application anytime and anywhere through Internet.***

The owner ship of the application exists with the SaaS provider / vendor (ISV). Customers would not be buying the software as such, but would be buying services of the software.

The Software application is delivered and managed remotely by the SaaS provider / vendor (ISV).

The Software application is used on Pay per use or Subscription basis. The more popular phrase for defining SaaS hence goes as, "Pay for what you are using, based on usage / transaction that an organization undertakes".




Let us analyze how the SaaS model addresses the challenges of both ISVs and Customers.

### 3.1.1 ISV Perspective

- **Centralized updates:** The software is loaded in a central system and users/organizations are provided with varying access levels depending on their

licensing term. ISVs now have the complete control of the production environment and can plan to take their software upgrades uniformly across to all the customers.

- **Pay per usage:** Now that the ISV has complete control of the software, they can come up with innovative models for using the system on a pay per use model. In the traditional on-premise model the software is beyond the reach/control of the ISV and hence, there is no mechanism in place to monitor the usage of the system, which prevents them from trying out new licensing models. In SaaS model, ISV gets complete control of the software, which allows them to monitor the usage of the product by a customer. This opens up a lot of opportunity for the ISVs to come out with innovative licensing models, which in turn provides the customer with flexibility on what they want to use, how long they want to use and how many times they want to use. This results in a "pay for what you use" model, which can be leveraged to tap the SMB market.
- **Quick roll-out:** The SaaS model also facilitates quick roll-out of the software. This is because, ISVs can more frequently release new versions of the software in SaaS model as compared to the On-Premise model (in an On-Premise model, a new version release is performed on an average of once a year, whereas in the SaaS model the same can be done more frequently)

### 3.1.2 Customer Perspective

**No Infrastructure Requirement:** From a consumer perspective, no infrastructure investment is required. All that is required is access to the internet. This ideally means that the CapEx is totally removed.

**Subscription / Usage based:** The Subscription and usage based payment concept of SaaS is also attractive for consumers. The consumers can actually pick their most suited software package option for their use and pay only for what they have selected and would be using.

**Application / Infrastructure maintained by provider:** In a SaaS model, the Applications and Infrastructure related to the Software are maintained by provider. This ideally means that the customer will have no operational expenditure. The Software Service Providers will ensure the aspects of application maintenance and standard checks of related hardware. Customer is spared of these tasks, which is taken care by the Software Service Provider.

**Purchase service rather than software:** With the SaaS model Customers are no more buying/owing the software rather they consume the services offered by the software provider. This eliminates the vendor lock-in concern from customers, as they are free to switch the service provider at any point in time.

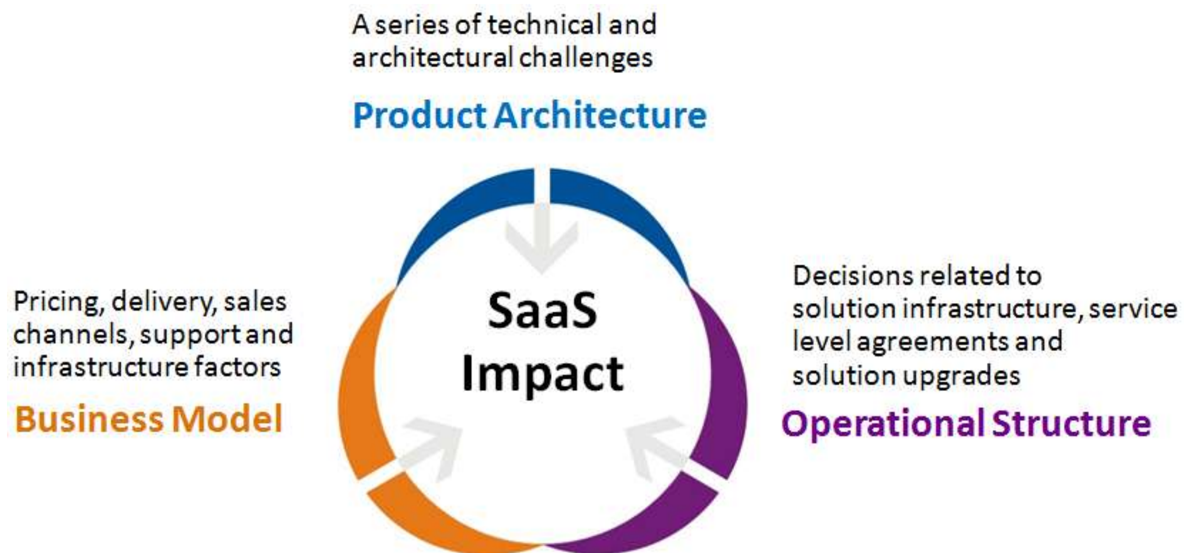
**Nil dependency on IT team:** In a SaaS model, buying software services reduces the need for recruiting specific IT resources to manage the same. However, in general instances, skilled IT manpower has been mandatory to manage and maintain IT resources such as IT infrastructure, Servers etc.

## 3.2 SaaS – the Big Impact

SaaS is not a technology change rather it's a change in the overall business model – the way in which you develop, deliver and realize revenue. Adopting a SaaS model can have impact across the organization. This impact can be broadly classified in to three areas,

***The SaaS model is more than just technology. There are three significant areas where SaaS has a mammoth impact.***

- Business model
- Product Architecture
- Operational structure

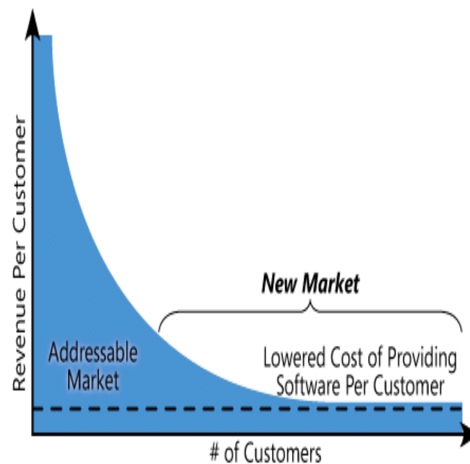



### 3.2.1 Business Model

***The primary advantage of SaaS as a business model is that it is more Customer-driven, and beneficial to both parties, viz., ISVs (Vendors) and Customers.***

Aspects such as pricing, product delivery, support, sales channels and infrastructure form significant parameters of the SaaS Business Model. Among them, pricing forms the most vital part. The primary aspect that an ISV would have to focus on is to deal with the drastic shift in pricing policy. Rather than charging a lump sum for the entire software sold, the ISV's would be shifting to a pricing policy of 'charging only for what you use'. ISVs would have to be careful to ensure in this latter scenario that they do not overcharge and end up losing vital customer base at the same time they should not be underselling, which may affect the profitability. They would have to come up with appropriate pricing policy keeping in mind the fact that they are now selling a service rather than purely software alone.

#### 3.2.1.1 Long Tail




Earlier, ISVs normally focused on large customers who can buy a large volume of software paying a lump sum. However, SaaS framework allows flexibility for the ISVs to cater to both large as well as SMB customers with flexible options of the software package to select and opt. Hence, as the chart above indicates SaaS model allows the ISVs to target the long tail of SMB customers.

### 3.2.2 Product Architecture

This constitutes the technical changes that will have to happen in the product in order to work in a SaaS model. The **technical** changes may range from few tweaking to complete re-architecture.

#### 3.2.2.1 Current state of the product

In a SaaS model the software is delivered over internet; this requires the product to be in a web compliant mode. Say for example, if the current product is in client/server or standalone model then the product should ideally be re-architected to a web model, which might result in major technical changes. Having said that let me also mention that there are virtualization technologies that can be used to deliver (SaaS Maturity Model Level 1) products in their current state (client/server or standalone).

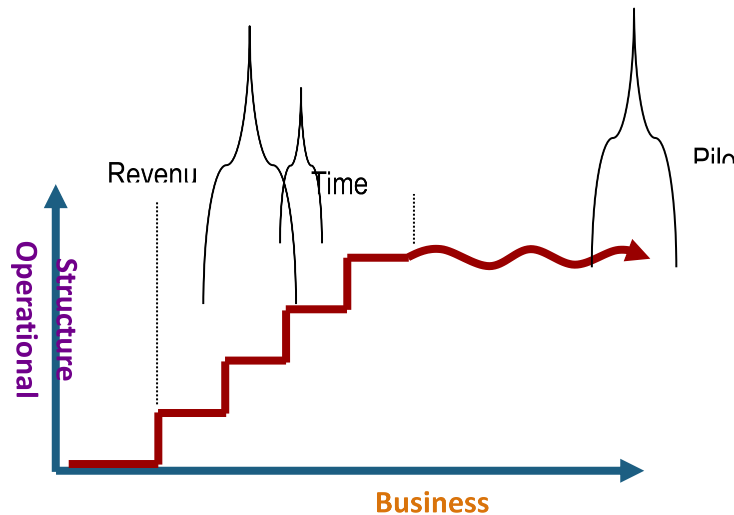
#### 3.2.2.2 SaaS Maturity model

There are 4 maturity levels in constructing a SaaS solution. Each of these levels has their own complexity and advantages. We will be discussing these maturity levels in the next section.

### 3.2.3 Operational Structure

The entire SaaS module revolves around the SLA (service level agreement) that is entered into between the ISV and the Customer. When customer buys the Software, both parties are bound by a set of service agreement. The term and preconditions within the SLA framework indicate scenarios and their corresponding outcomes such as, wherein the Software may stage outside the SLA, or whenever the software fails in performance etc. There are quite a few operational challenges that would have to be addressed by the ISVs and Customers while operating within a SaaS framework.

Sample Revenue Flow From a New Customer




The chart indicates an initial stage when a pilot project is stared that does not generate any revenue. In due course, while the project gains adequate Customer patronage, the project does start generating revenues. During most instances, Customers do not perform an outright purchase of all modules of the software, but would opt for anything between say, 1 or say 10 modules out of the total modules available in the software. All said and done, once the business grows cash flows are bound to increase for the ISVs as and when customer usage grows.

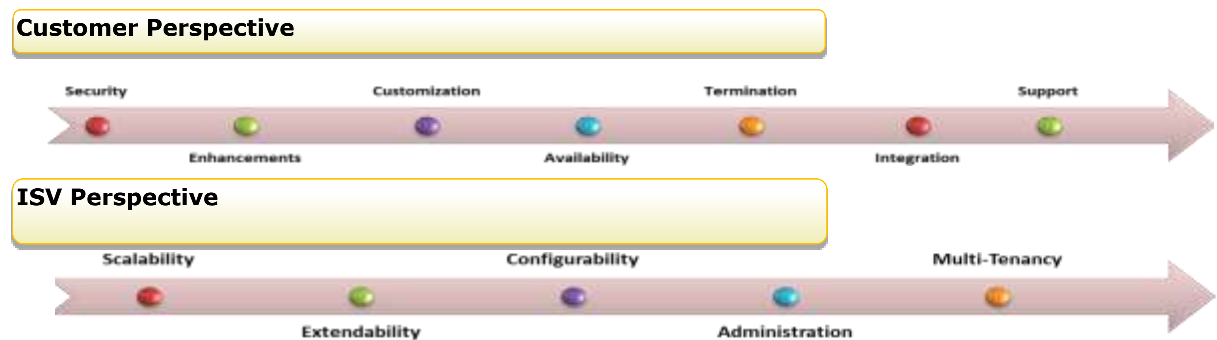
#### 3.2.3.1 Optimized usage of support bandwidth

In a SaaS model, cash flows are quite less in the initial stages. Hence, it is vital for the ISVs to have an optimal support team as well as the resources they are using.

#### 3.2.3.2 Self-serving system

The goal of an ISV again, should be to serve the end-users queries independently through the software, without need of much support. FAQs and online help manuals can be provided for the customers along with the software for support, thus ensuring a self-serving system.

## 3.3 SaaS expectations from a customer prospective & ISV prospective




### 3.3.1 SaaS expectations from a customer perspective

- **Security**: Indicates the level of security of customer's data. In case, the software is hosted outside the customer's premises, the framework should allow for enough security in such instances.
- **Enhancements**: Indicates aspects such as the frequency of availability of software upgrades. Customers naturally expect speed and frequency at which the upgrades are available.
- **Customization**: Customization is a pre-requisite that all customers require to customization depending on basic needs of software.
- **Availability**: How can the availability of software be guaranteed to the customer during instances when they want to move out.
- **Termination**: How can I get my data back? How easy to get data back?
- **Integration**: There are instances where customers would have already developed software in house. Integration hence indicates the provision for integration of the new software with their old application and be able to work in synchronization.
- **Support**: How good is the available support? Who will take care of the software? These form basic queries in terms of support in a customer's mind.

### 3.3.2 SaaS expectation from an ISV perspective

- **Scalability**: Given that ISV are going to target the long tail of SMB customers, the product should be scalable to handle large customer base without any major re-architecture.
- **Extendibility:** Product should have the capability to extend (or allow to extend) itself to meet the specific requirements of a customer. For example, in a HR SaaS product, customer may ask for additional information to be maintained as part of the employee record. The product should allow this to do with minimal to zero effort.
- **Configurability**: The large customer base is going to lead to a situation where different customers ask for different kind of changes. Therefore, the product should

have a solid and flexible framework that can allow various kinds of aspects (UI, Business Rules, Workflow, etc.) to be configurable at a customer(tenant) level.

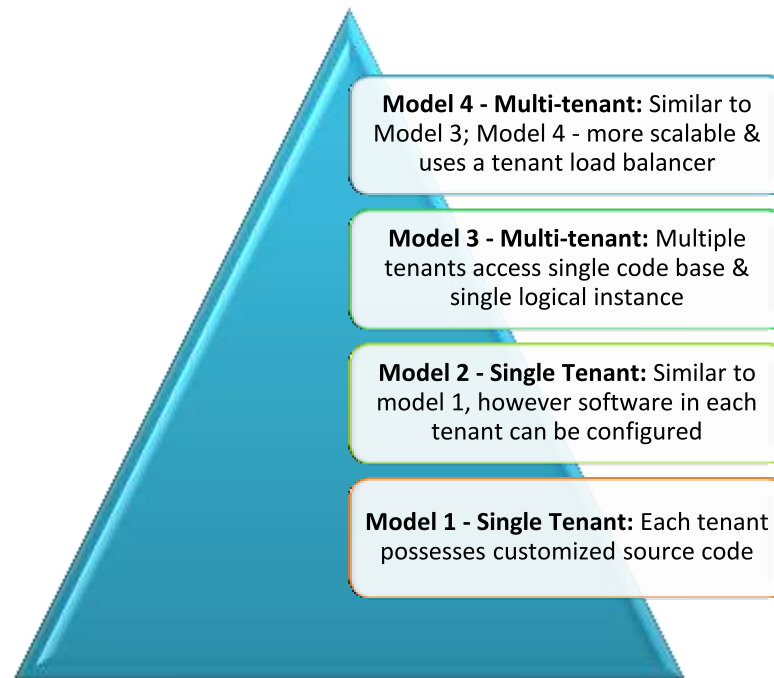
- **Administration**: Product should support administrative screens and dashboard screens, which can be used to perform administrative actions without getting in to too much of details. For example, if a new tenant has to be added the administration module should provide a simple interface that can collect some basic data from the user based on which it can completely automate the tenant creation process.
- **Multi-Tenancy:** This is required to operate the product in Level 3 or 4, which is highly cost effective and scalable.

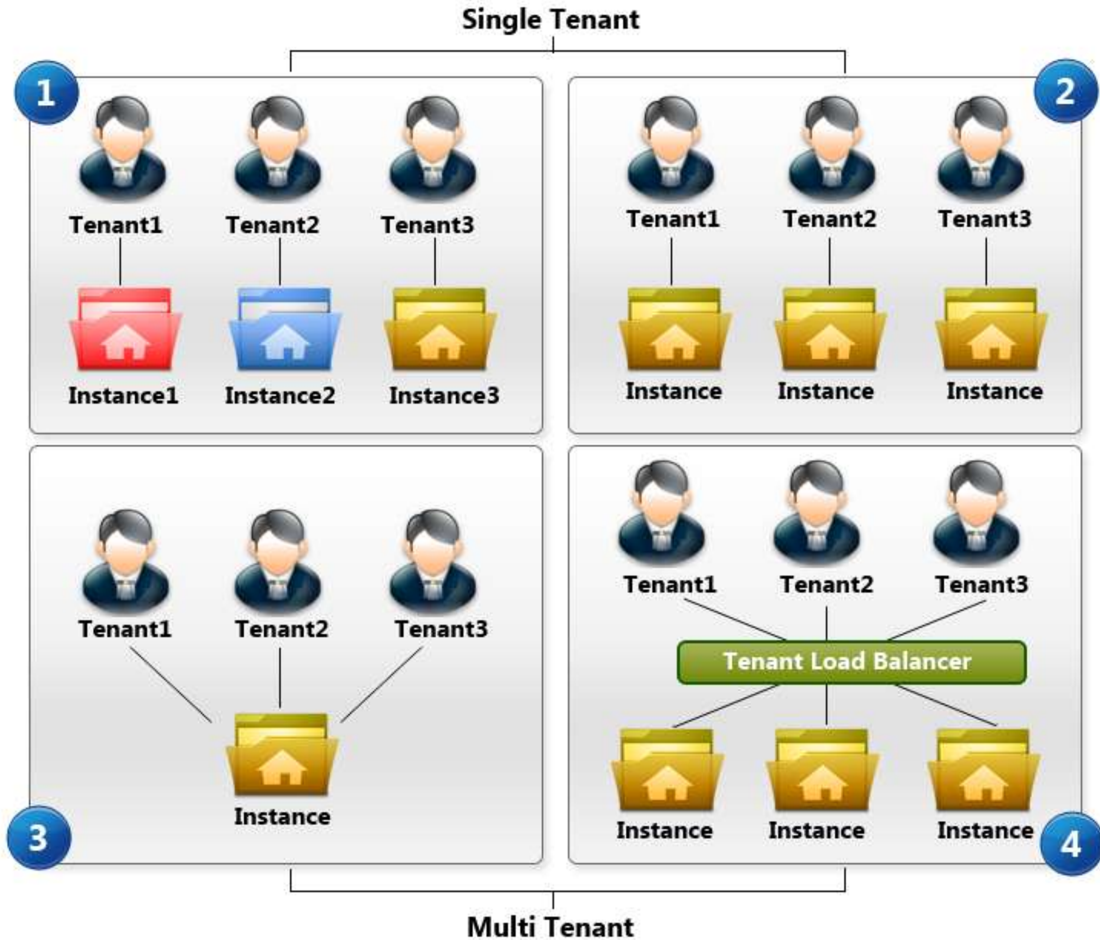
## 4. Multi Tenancy Basics

Multi Tenancy is the ability to handle multiple organizational data and applications in a single instance. Four kinds of maturity models are available while building a SaaS based product. Out of the four models, two of them are single tenant or multi instance models. The rest of the two are Multi-tenant models. Moreover, there are variations within the single-tenant model as well as the multi-tenant models too.

### 4.1 Major Architectural models / patterns for building SaaS product

There are the four major architectural models / patterns based on which organizations can select and build a SaaS product.






**Model 1 - Single Tenant:** Considering the case of the first maturity model with in the single-tenant type, this model allows each customer/tenant to have a specific code base for themselves. This ideally means that different customers can use their own different customized source codes. This model is similar to an ASP model that was in popular use a few years ago purely in terms of the software deployment.

**Model 2 - Single Tenant:** The second maturity model with in the single-tenant type is similar to the first maturity model in terms of software deployment. However in this model, the software possesses the capability for configuration. Hence, though each tenant has different needs, the source code would be the same for each tenant. However still, from the deployment point of view, they all are run in their own logical instance.

**Model 3 – Multi-tenant:** As the nomenclature suggests, the third, multi-tenant maturity model involves multiple tenants accessing a single code base and single logical instance.

**Model 4 – Multi-tenant:** There is only a slight deviation in the fourth multi-tenant maturity model from that of the third multi-tenant maturity model. The fourth multi-tenant maturity model is more scalable compared to the third multi-tenant maturity model, having a tenant load balancer in place.

## 4.2 Maturity level factors

Further, there are different factors that govern whether or not an organization should implement multi-tenancy. These factors are with respect to time, cost, business volume and product size. The following table lists various maturity level factors that form the basic parameters in deciding whether or not to implement multi-tenancy:

| Factors | Level 1 | Level 2 | Level 3 | Level 4 |
|---|---|---|---|---|
| Operational Efficiency | Very low | Low | High | Very high |
| Maintenance Efforts | Very high | Medium | Low | Very low |
| Scalability | Very low | Low | Medium | High |
| Time | Very low | Medium | High | High |
| Transition Cost | Very low | Low | Medium | Very high |
| Engineering Skill set | Same as on-premise | Medium | High | Very high |
| Customer Value Add | Very high | High | Same as Level 2 | Same as Level 2 |

Each of the factors mentioned in the table above are significantly and variedly affected in the following aspects, at each of the four levels of maturity models.

### 4.2.1 Operational Efficiency

When a single instance serves several tenants, the effort to maintain the same such as implementing the patches, capturing backups or cost required for hosting, etc. would be relatively less. Hence, a multi-tenancy model with level 4 is significantly high in operational efficiency as compared to other maturity level models.

### 4.2.2 Maintenance efforts

Whenever a new version or a patch is released, amount of time and money required for this task in a single-tenant model is higher as compared to multi-tenancy. Industry study claims that the multi-tenant savings could be as much 16X times compared to a single tenant model.

### 4.2.3 Scalability

A multi-tenant model possesses better capability to add more customers/tenants or add more volume to the product as compared to single-tenant models.

### 4.2.4 Time

Time is a major factor to be considered while deploying a multi-tenant model. Time taken for moving and deploying an existing application to any of the multi-tenancy model, is far higher as compared to its shift and deployment to a single-tenant model

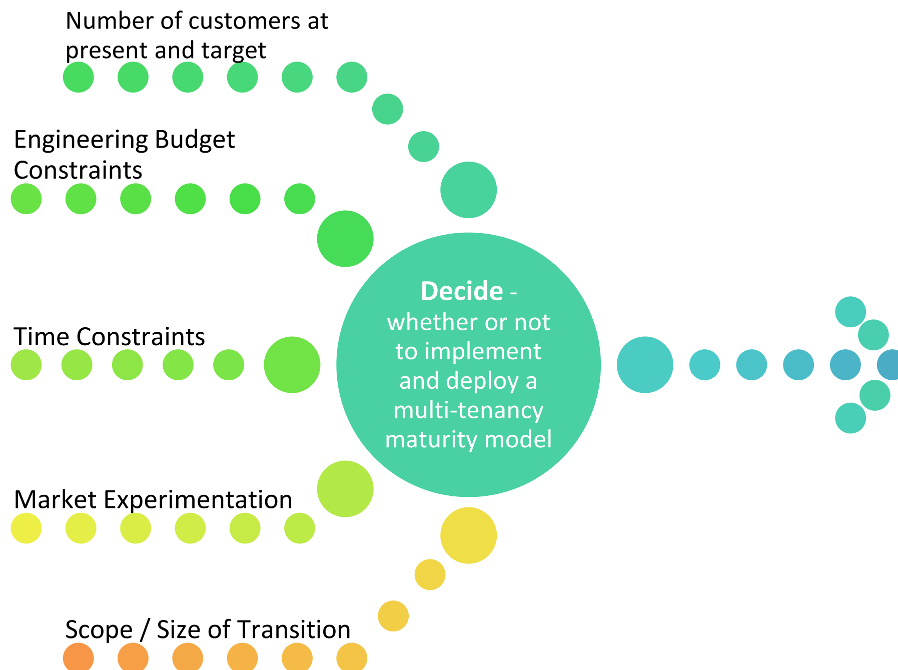
### 4.2.5 Engineering skill set

Technology and engineering skill sets required to build a multi-tenant system are quite high as compared to implementing and deploying level 1 or level 2 (single-tenant) maturity models.

The striking parameters that determine the selection of a suitable multi-tenant model is its operational cost and maintenance. In order to maintain a profitable business, operational cost would have to be brought down to the minimal possible extent. There is an approximate 16 times cost advantage in selecting a multi-tenant maturity model over a single tenant maturity model. Since multi-tenancy helps in achieving this goal, a multi-tenant model is definitely a striking preference over a single-tenant model.

## 4.3 Multi-Tenancy Decision

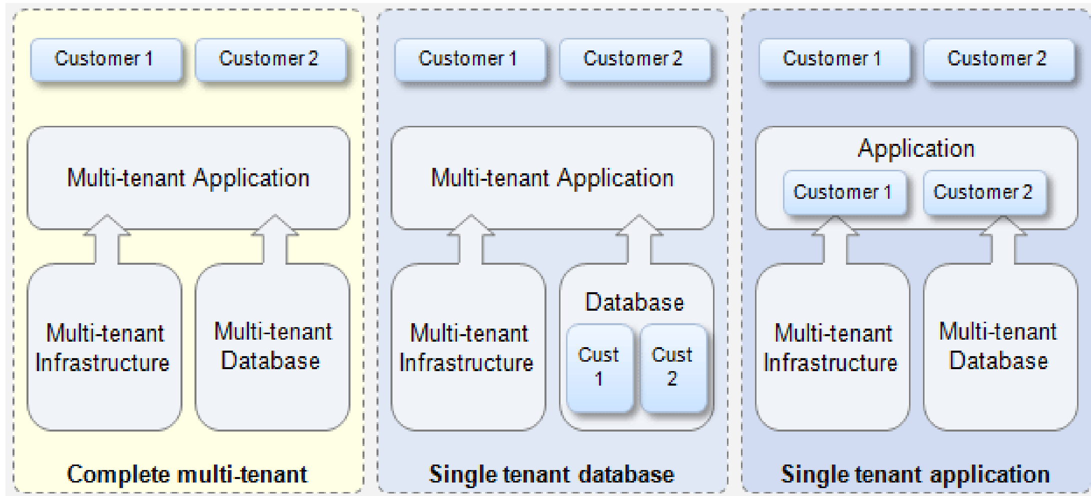
The following factors play a major part for a business to analyze and determine whether or not to move into a multi-tenant model:



| | |
|---|---|
| Number of customers at present and target | For businesses with a large customer base is benefited tremendously by moving and adopting a multi-tenant maturity model. |
| Engineering Budget Constraints | It is best for a business with stringent budget to deploy a single-tenant model. |
| Time Constraints | Similar to the Engineering budget constraints, if a business is facing short time to prepare for the market, then the single-tenant model would best suit the business than the multi-tenant model. |
| Market Experimentation | There are instances when SaaS as a business model is still unproven for the line of the company's business and its viability too is yet to be tested. During such instances, the company would surely not be willing to spend further on modification of the current application. In such cases, deploying a single-tenant model would be the ideal option. |
| Scope / Size of Transition | If a specific feature would have to be implemented in a SaaS model, then the same can be quickly included within a single-tenant model and implemented. |
| Reusability of current application | During instances when the code base of a huge product has to be reused instead of adopting the risky method of rewriting the same, then the single-tenant model can be implemented. |

## 4.4 Multi-Tenancy Degree

Different types of architecture are available for selection in multi-tenancy.




As indicated earlier, multi-tenancy is a model where, one logical instance/application would be serving multiple customers. However, within that application the primary factor that needs consideration is the Data Architecture. The data architecture itself could be single-tenant or multi-tenant. Hence, multiple databases can be maintained for each of the client. For example, Tenant1 can have a separate data store and Tenant2 can have a separate data store and so on, while the application is multi-tenant. However, the data base would not be as seen in the

middle figure. Even though this may still be known as a multi-tenant application, however from the data architecture point of view this would be a single-tenant database.

Considering the database, the storage for all the application would have to be considered such as a database or cache or files that are stored within the system. Hence, multi-tenancy would have to be structured in all these areas too. Here, the first model illustrates the fact that all these aspects would be structured into a multi-tenant model. These are hence the variations seen in multi-tenancy itself which could be adopted into.

### 4.5 Multi-Tenancy without SaaS

There is also a need for multi-tenancy in a non SaaS environment, for example – a large multinational enterprise such as General Electric. Multi-tenancy is an architectural pattern and it is not unique only to SaaS. This pattern can be applied to an internal application too. For example, building an in-house HR-payroll processing application. This enterprise needs to cover different business units and different geographies (countries). Considering that each country has different payroll taxes, application will have to be architected to take care of these nuances. Multi-tenant architectural model ideally will take care of these needs.

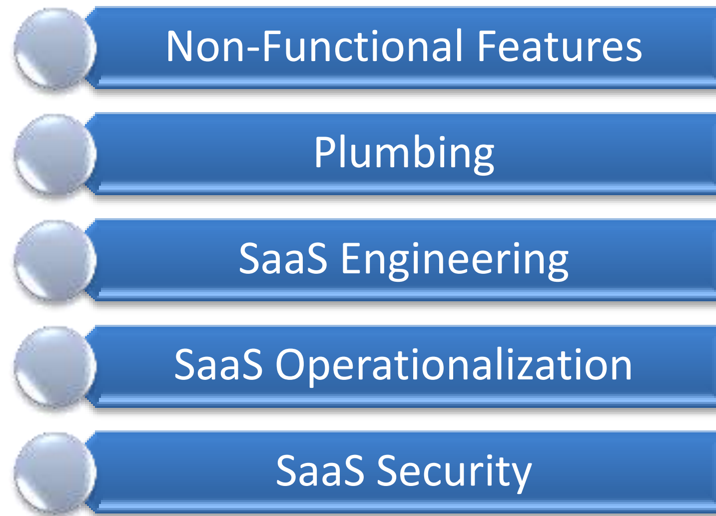
## 5. Technical Stack of Multi-Tenant SaaS Product

Most of the developers assume that multi-tenancy is all about filtering data at the database end by adding a tenant id. At this juncture, the fact that the developers would focus on is:

- to filter the queries that are yielded out of the database in terms of the tenant as well as while they get stored
- appropriately identify the records and assign them correspondingly to the tenant.

However, multi-tenancy goes beyond just storing and retrieving tenant based data. Main challenge in multi-tenancy is how to manage the configuration, workflow changes, etc. at functional level without having to change the underlying code for each tenant. And, also consider non-functional requirements such as security, scalability, performance, etc.

At a higher level, following block diagram explains the technical stack for a multi-tenant SaaS product.



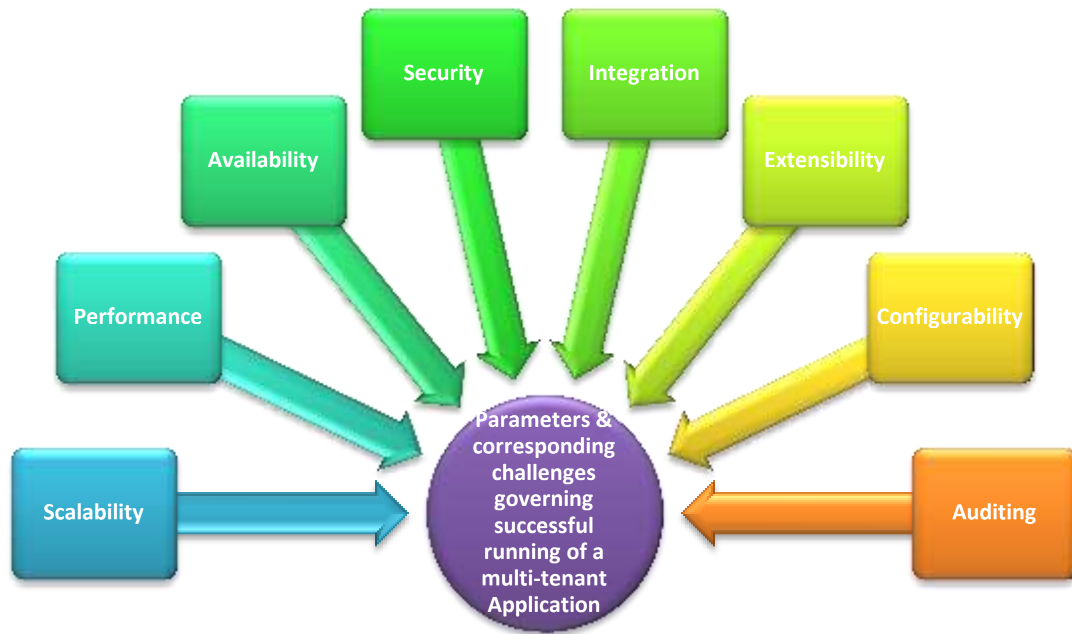

This section explains about all these aspects.

## 5.1 Non-Functional Requirements of a multi-tenant/SaaS product

Technical aspects of deploying complex multi-tenant model involve all non-functional requirements that are not directly translated to business functionalities but are rather the parameters that govern the successful running of the product while serving:

| Parameters | Challenges |
|---|---|
| Scalability | Design to handle current and future loads, Optimum use of hardware and other resources |
| Performance | Response time, Bandwidth constraints |
| Availability | SLA Compliance, Offline mode of working |
| Security | Physical and Network Security, Role based access, Data Encryption |
| Integration | Support for in-bound and out-bound integration<br>Standards compliance (HL7, cXML, etc.) |
| Extensibility | Custom field support, Support for dynamic forms |
| Configurability | Personalization/"Org"analization, UI/Business Rule/Workflow |
| Auditing | Entity/Data level tracking |

The table above lists non-functional requirements of a multi-tenant application. These non-functional features are complex in a multi-tenant environment and require being a conceptualized part and parcel of its architecture, to eliminate occurrences of bottlenecks.




Let us now analyze how these parameters get altered and evolve to form a multi-tenant application as compared to an on-premise application or a single-tenant application.

### 5.1.1 Security

Security is a key aspect for a multi-tenant application. In a SaaS model, data lies outside the organization – so, generally customers are wary of the security – physical, network and data security. In such instances it has to be ensured that the customers are confident and convinced with type of security standards that is provided to them. Hence, it is best to comply with suitable security standards and certifications to maintain the confidences of customers.

Primary aspect of security deals with role based access, expected to be a flexible factor, however is generally considered complicated in a multi-tenant model. This is because, tenant systems possess configuring capabilities. Hence, a role based access security in a multi-tenant environment is different from a single-tenant model and would have to possess a customizable design.

Data encryption is another vital aspect of security to be carefully considered and planned within a multi-tenant model. The responsibility of ensuring secure data lies with the ISV, and hence has to adopt a planned encrypted security model for storing data. The ISV also has to consider different types of encryption mechanisms to suit different tenants.

### 5.1.2 Scalability

SaaS based businesses involve a volume game and profitability increases only with the increase of customer base. The very nature of multi-tenancy is intended to support huge volume of customers / users from a single instance running. This ideally means that the product being deployed would have to be highly scalable to handle high volume data and transactions.

Scalability can be discussed at various levels and not just about a web application layer. This is because varied technologies are available to scale-out the web layer. For example, scaling a .Net application requires a load balancer in place. Developer must consider state maintenance, whether or not a sticky session can be handled, and aspects related to storing related details while coming out of a sticky situation, etc. While implementing multi-tenant web application the complexity actually may rest in other aspects such as bottlenecks in database. Hence, scalability is a vital parameter that would have to be well-planned and integrated right from the initial stages of developing the application.

### 5.1.3 Performance

Another vital parameter to be significantly measured is the performance of the multi-tenant system, especially for the increased volume of data and transactions. There are a few vital aspects to be planned well ahead with respect to the performance while designing a multi-tenant model:

- An Instrumentation mechanism should also be in place to measure the performance of the system at the level of each tenant.
- a performance count is necessary to be in place for every tenant.
- One of the other vital aspects to be planned and handled is about the bandwidth constraints. For instance, if a service layer or middle layer has been opened up for integration, then the usage of bandwidth would be higher. Further, if there is an offline support capability in place such as data synchronization, then there could again be a larger usage of bandwidth.

### 5.1.4 Availability

SaaS application needs to be highly available. If you look at 99.99% uptime, the application would have an approximate 5 minutes of downtime every year. This would be a significant aspect of the SLA that will be drafted in the contract with the customer. Hence, it is very important to have an advanced disaster recovery mechanism in place.

### 5.1.5 Integration

For implementing a SaaS framework, it is mandatory to have an integration of some type. Communication from a SaaS perspective could transpire both ways viz., inbound and outbound. Similarly, integration can occur from an on-premise to SaaS application and vice versa too. During such instances, certain web services would have to be invoked or certain services would accrue on-premise end. Hence, it is best to implement a specific framework to handle and manage such off the station instances.

Further, considering aspects through the multi-tenant scenario is also quite significant. SaaS application would communicate with multiple on-premise applications of the organization. It is mandatory from an integration perspective, to design a seamless orchestration, with a single central layer managing all these aspects.

### 5.1.6 Extensibility

Product designed for customers require being extensible. It is hence to be left at the discretion of the tenant to define as to what would suit best for their requirement in terms of the look and feel, forms, grids etc. of the application.

### 5.1.7 Configurability

One of the biggest challenges in a SaaS product is to provide the level of configurability that would ease the life of both provider (ISV) and buyer (Organization). From a buyer perspective there could be several areas that may require configurability – UI, Themes, Business Rules, workflow, reports, templates, notification, integration, authentication, etc. The more configurability options an ISV provide more the edge they have against their competitor.

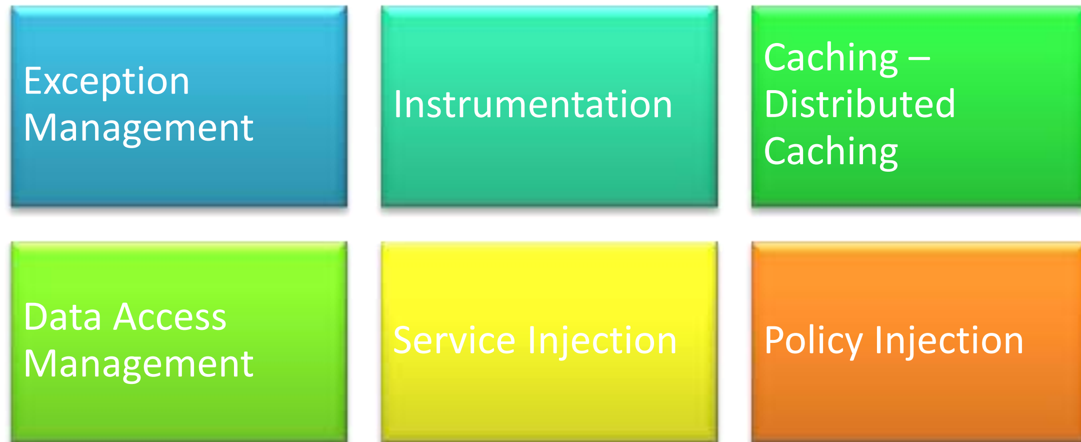
However, if we look at from the ISV side of it they would want to provide this configurability but at the same time they cannot afford to spend a lot of (implementation) time in achieving this configuration. Having a solid architecture/framework that can provide support for the various configurable options makes a big difference.

### 5.1.8 Auditing

Auditing is an important aspect of multi-tenancy design. Auditing requires being an important, integral part of a multi-tenant system, at the level of every event or a transaction. Systematically, and properly audited messages are a mandatory feature of a SaaS application. Further, there should also be a provision in place for audit by tenant or to access the audit reports available by tenants.

## 5.2 Plumbing Features

Plumbing is the primary layer that is laid together foremost, while developing any application in general and Multi-tenant application in particular. Major blocks that would have to be handled here include Exception Management, Instrumentation, and Caching etc. Considering the basic cross cutting components found in any application, in a multi-tenant application, these components are driven by *Tenant's context.*




### 5.2.1 Exception Management

Exceptions can be written for an on-premise application. However, in the case of Multi-tenant applications parses specific for every tenant is required. It would not be adequate to log exceptions alone. In order to trouble-shoot better, it is vital to quickly locate the exceptions occurring at every tenant's end. Hence, there is a necessity to design and build a layer that abstracts the tenant's context and handles the same at the back.

### 5.2.2 Instrumentation

Instrumentation would be necessary for exceptions or for performance or for specific events that occur in the application. Such provisions are available in the on-premise application. While viewing instrumentation messages, it is also necessary to possess the capability to fetch them identified with respect to the specific tenant from which it is sourced.

### 5.2.3 Caching & Distributed Caching

Caching and Distributed Caching are vital blocks that have to be handled in the Plumbing layer.

A Cache is used for storing data that is static in nature so as to increase the performance of the system. In general, Java or .Net applications are equipped with cache and are hosted

on the Internet. However, in scaled out scenarios, these would not work. This is because, in such scenarios, multiple machines would be reading the cache and also in turn would be writing back to the cache. Hence, it would be difficult to achieve cache synchronization. For instance, considering that data is being stored or altered in the cache of a machine/system, say, A. The next machine B would be serving the subsequent request and would have no clue on machine/system A's activities. Hence, it becomes quite necessary to have a central cache for these two systems.

- **Cache Policy**
  - Data to cache
  - Read only cache Vs Updatable cache
  - Cache expiration
  - Time based expiration
  - Notification based expiration
  - Frequent Poll Expiration
- **Distributed Scenario**
  - Out of proc Cache
  - Cache distribution
  - Local Cache
  - Object Serialization

## Cache Policy

Decisions on data caching would have to be made right from the initial stages of product development. This includes decisions such as the kind of data to be cached, type of data to be cached (read only data or transactional data) etc.

### Cache expiration

Numerous cache expiration policies would also have to be supported, such as whether the cache expiration would be a trigger from application itself, or could it be a time sliding expiration etc. Again depending on the volume of the tenant, different cache mechanism would have to be selected. For instance, if a tenant has high volume of transactions then it would not make any sense to cache the data for that particular User. Hence, different caching policies would have to be framed, customized for the needs of each tenant. In general, the following 3 types of expiration can be encountered and framed:

- Time based expiration
- Notification based expiration
- Frequent Poll Expiration

### Distributed Scenario

Distributed caching is required in a scaled out scenario. In a distributed scenario, following techniques would have to be adopted to make the application performance better in a multi-tenant situation.

**Out of proc Cache**

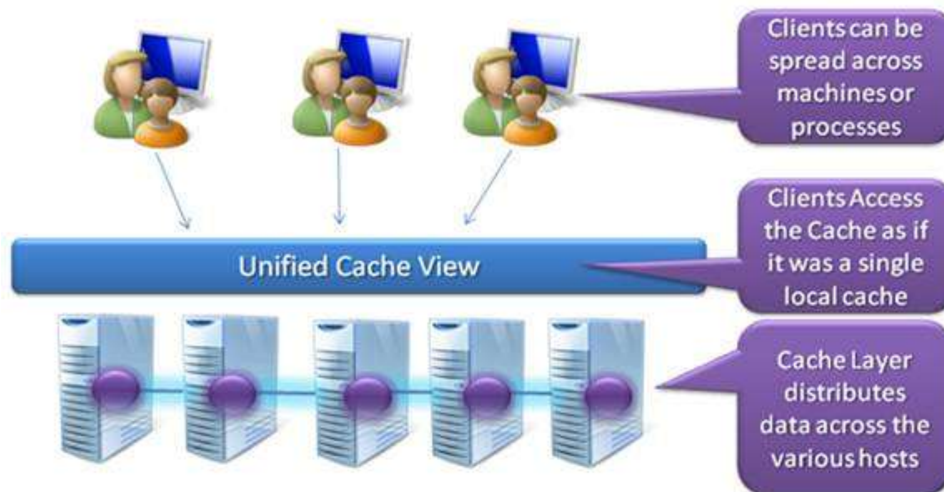
An Out of proc Cache ideally means that the cache does not reside inside the application's memory.

**Cache distribution**

Distributed caching helps in balancing the loads sourced from different tenants.

**Local Cache**

In a Local Cache scenario, a local application cache is maintained, which gets synchronized with the central cache cluster or the Server. Further, this is also the reason for the necessity to have a local cache.
Considering the current technological trends numerous such caching blocks are available. For instance, windows app fabric is distributed in nature. Even if the application is hosted in cloud such as amazon or azure, there is support for distributed caching. Most of these caching blocks are available out of shelf and can be used directly for leveraging.




### 5.2.4 Data Access Management

In a multi-tenant scenario, Data Access Management plays a critical role in segregating the data by tenant. Apart from managing the Data I/O operations, this layer can be designed in such a way to handle the tenant context related operations. This also gives provision to auto inject tenant related filtering of data so that the developers need not worry about explicitly filtering data by tenant.

### 5.2.5 Service Injection

Service Injection in simple terms means, discovering a service for a particular case and being able to plug the discovered service into the same. This allows for extreme extensibility. In a multi-tenant scenario, a mechanism to inject the service based on a tenant is required. Having loosely coupled services that are discovered at runtime would help in solving problems.

### 5.2.6 Policy Injection

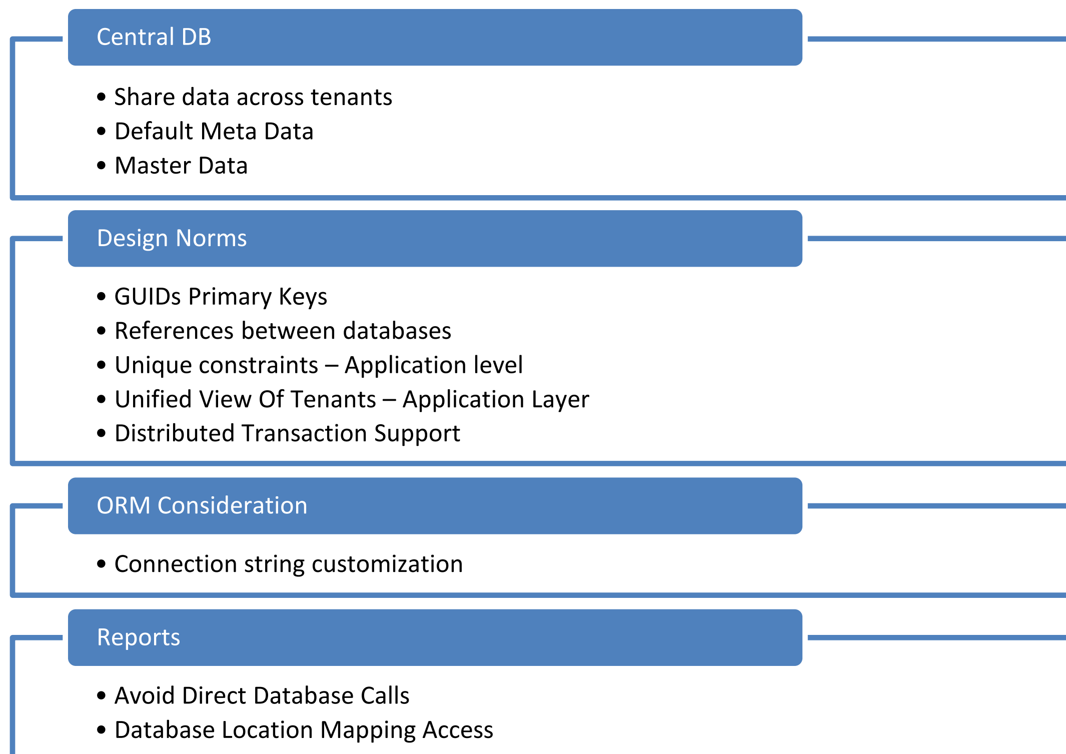
Policy injection provides the ability to inject code at various levels of operations of any product. It helps in bringing aspect oriented programming. For example, if calls would have to be instrumented to a particular method, say, ABC, of a service, then this method, ABC, should have the ability to be plugged in for a particular tenant at runtime. This makes the product loosely coupled and would render the ability to add more external services to business functionality at runtime.

## 5.3 SaaS Engineering Features

Laying down the SaaS framework forms the next level to laying the basic block, viz., Plumbing, for developing an application. Following paragraphs analyze components of a framework that require multi-tenancy. The components indicated in the following figure are the basic building blocks of a SaaS framework.

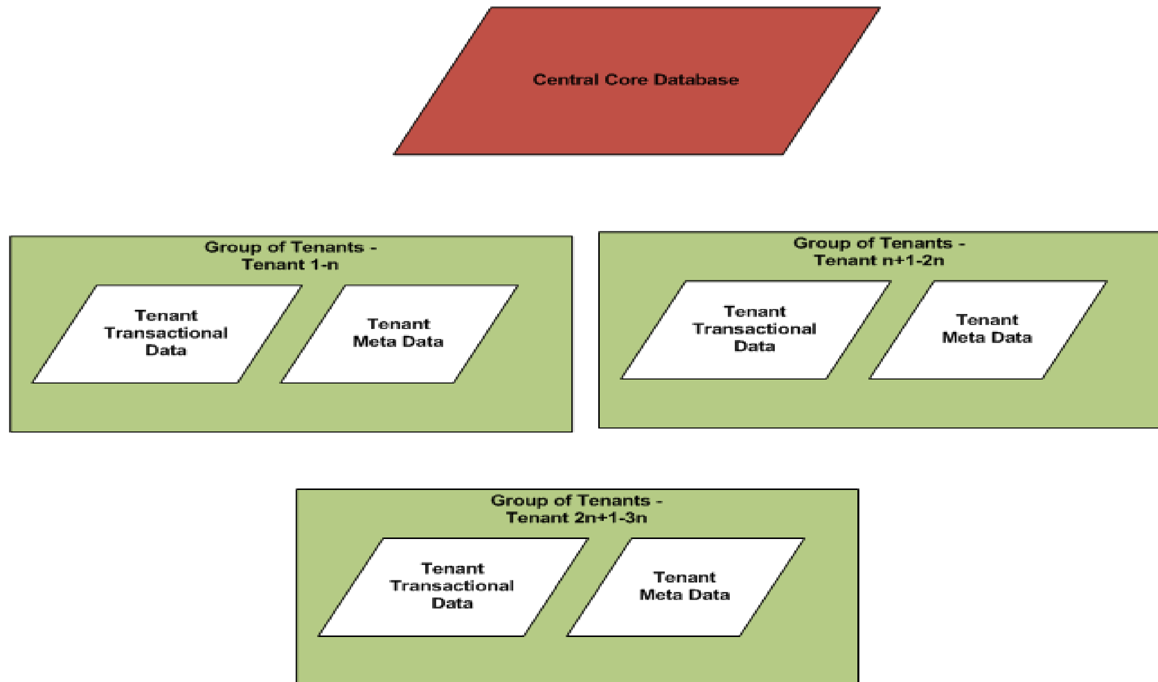
### 5.3.1 Data connection & Abstraction

In a multi-tenant scenario, controlling the isolation between tenants is an important aspect to be handled by the application framework. In order to maintain a secure layer from the application, there is a need for developing and maintaining a complete abstraction that isolates different tenants and brings in the security later at the framework. The purpose of developing and maintaining an abstraction is to aid the developers not to be developing this every time. It is important to adapt a suitable kind of isolation mechanism. This isolation mechanism would have to be brought right at the framework level.




**Central DB:** A partition scenario also leads to a scenario having a central Database. The central database would contain all the master data and the metadata information would reside on top of the same. This is an aspect to be fetched right from the initial stages of carrying out data isolation.

**Design Norms:** Design norms are quite important if a partitioned database is opted. IDs or the primary key would be unique across all the databases. For instance, if there are 50 tenants in one server and a sudden load increase is encountered, then data would have to be moved from one or more tenants to another Server. The one and only way to achieve this is to pre-design the databases with unique primary keys.
If a horizontal data partitioning is opted for, then, these are some of the design norms that would have to be considered right from the initial stages of design.




The above structure highlights a high-level view of how a data architectures. There would be a central database and partitioned databases.

### 5.3.2 Authentication

The primary aspect that could get complicated in multi-tenant scenario is single sign-on. In a multi-tenant scenario, a number of tenants are served, each of which can have their own way of authentication. Hence, in order to support single sign on, it would be best to go in for federated authentication, with standards such as SAML in order to support single sign-on, which is more of an industry standard. These standards completely eliminate the need to write code or program the product specifying the mechanism and authentication identity. A federation server can be adapted to. For example, an ADFS server from Microsoft can be used for a datacenter which is more like a federation and also fits very well into federated mechanism. On the contrary, if the application has to be hosted in cloud such as Azure, a support for access control service will be available which will enable performing multi-tenant based federated authentication.

**Authentication**

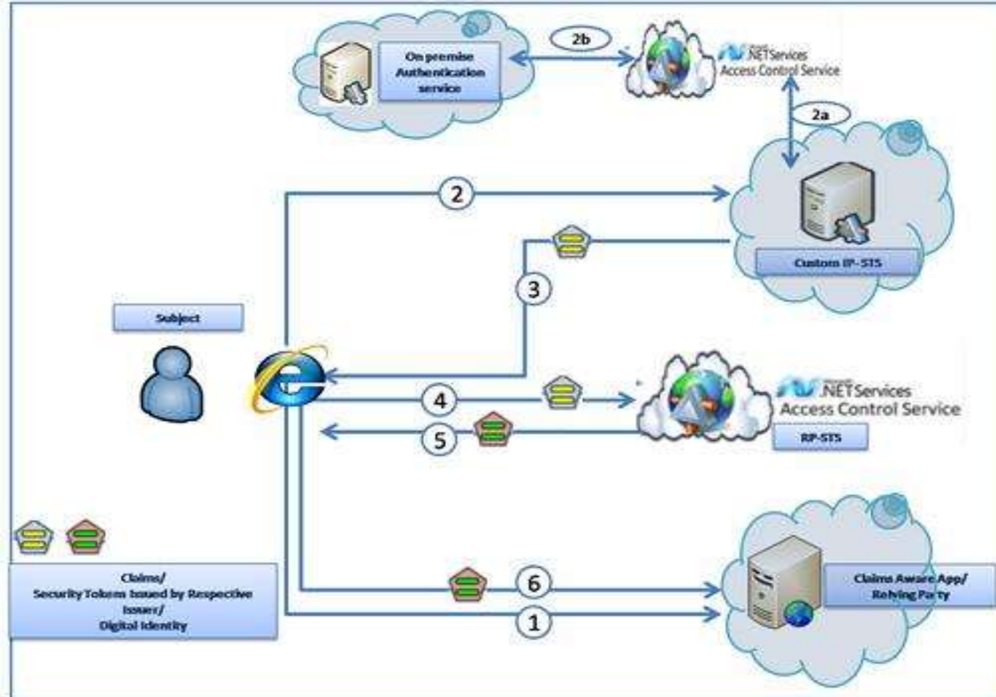
- Tenant Code Based Authentication
- Single Sign On
- SAML
- Federated Authentication
- Support claims
- Identity Provider
- Password Policies

Federated authentication support becomes quite mandatory in a SaaS application. In federation, a multi-tenant scenario has to be thought about. Further, the scenario would not be a straight forward federation where an application would be communicating to an identity provider. It would in fact be a scenario where communication would be between one -to- many authentications providers for different tenants. Hence, it would be required to select a federation server that can fit well into this need and further analyze how to plug in multi-tenancy in that layer too.

In this scenario, the application would also have to support claims. Since Claim is more about the identity of the User, it would not be a simple scenario. Hence adapting to standards such as SAML would make the scenario far easier to handle.

| Term | Definition |
|---|---|
| Claim | A statement that one subject makes about itself or another subject. For example, the statement can be about a name, identity, key, group, privilege, or capability. Claims are given one or more values, and then they are packaged in security tokens that a security token service (STS) issues. |
| Claim type | The type of statement in the claim that is made. Example claim types include FirstName, Role, and Date of Birth. The claim type provides context for the claim value. |
| Claim value | The value of the statement in the claim that is made. For example, if the claim type is FirstName, a value might be Matt. |
| Digital identity | A set of claims that represent a subject. |
| Identity provider | An organization that issues claims in security tokens for a particular transaction. For example, an identity provider, such as a university, might issue a claim in a security token that enables one of its students to purchase books and materials from the university's Web site that is managed by a contracted relying party organization. |
| Identity provider security token service (IP-STS) | A software component or service that an identity provider uses that issues claims and then packages them in security tokens. |
| Policy | A set of parameters that an STS requires to identify relying parties, identity providers, certificates, account stores, claims, and the various properties of these entities that are associated with the STS. |
| Relying party | An organization that uses the claims in security tokens that an identity provider issues. For example, an online auction Web site organization might receive a security token with claims that determine whether a subject can access all or part of a relying party's application. |
| Relying party application | Software that can consume claims in security tokens that an identity provider issues to make authentication and authorizations decisions. |
| Relying party security token service (RP-STS) | A software component that uses the claims in security tokens that an IP-STS issues. The RP-STS receives the security token, verifies the signature, maps the claims, and then generates and signs new security tokens that a relying party application consumes. |
| Security token | A collection of one or more claims. Security tokens are usually packaged in standard formats for interoperability. Common formats for security tokens include Security Assertion Markup Language (SAML), X.509, and the Kerberos authentication protocol. |
| Security token service (STS) | A software component that is used to issue claims and packages them in security tokens. |

The following figure provides a quick view of how authentication works in federated scenario.




### 5.3.3 Authorization

A flexible access control policy is mandatory in a multi-tenant system. This ideally means that the application requires having flexibility to let the tenants define their requirement in the system. This further means that the access control policy would have to be most flexible. Application access routines would have to be written based on privileges rather than on roles. In most of the on-premise applications, the general pattern follows writing access routines based on roles. Hence, privileges would have to be defined first and the codes would have to be written against the privileges. The roles would have to be then customized by the tenants. The roles can be mapped to the privileges by tenants themselves. Tenants can pick and choose the privileges to be assigned to each role. Once the user is mapped with the roles, they get the privileges automatically and indirectly. Based on this, the framework would have to filter what the user does in the system.

**ACL Policy**

- Privilege Based Authorization
- Named Privileges
- Entity Action Privileges
- Custom Access Control Policies
- Global Roles
- Tenant Defined Roles
- License and Privileges
- Features Privilege mapping
- Role Privilege Mapping
- Role Hierarchy
- Access Control Units

**Action**

- Field With an Entity/ Form
- Reports
- Data – Datascope

**Consideration**

- Adhoc Reports – Build with a user friendly model

There are different kinds of access controls. Access Controls could be based on an entity such as adding an employee record, or updating an employee record. The access control could also be based on a field. For example, an user may be able to update the details of an employee but not the salary information. Access control could be based at the form level too. For example, the user may be able to view all the information of the employee, except for the salary information. Finally access control can be also based at the data level. For example, a user can view only those employee records who are reporting to the user. This scenario is called as Data Scope.

Another vital consideration often ignored is the security aspects to be analyzed while providing reports, notifications etc. This forms a vital aspect of providing secure authorization to users.

### 5.3.4 Data Security

While storing sensitive fields in the database, enough care would have to be taken on encrypting the same. On encryption, aspect that needs to be considered is that, keys should be properly stored. A dual storage would have to be adopted, least of all to ensure that keys are not available at just a single location.

There may be scenarios where different tenants have different keys. Hence it is vital to adopt a key storage or key management technique for each specific tenant. This may further affect the application's performance. Hence, a trade-off between security and performance would have to be adopted.

**Data Security**

- Strong Algorithm
- Key Storage
- Minimal Dual Storage
- High Privileged User For Key

**Consideration**

- Performance
- Direct Data Access In Reports

### 5.3.5 Query Generator

Query generator helps customers to fetch data from the central database. This feature aids the Service Provider by not allowing customers to directly access the database.

### 5.3.6 Notifications

Notification involves the system sending emails, files etc. A notification engine should support multiple types of notifications. It could be email or an ftp, or sometimes even an SMS support. Notification should also further allow batch processing tasks. A vital aspect to be considered under notification is, maintaining and handling multiple notification templates. Here, each tenant may have different sets of data to be sent. Hence, there is a need for maintaining multiple notification templates.

**Notification**

- Email Notification
- FTP Notification
- Notification Templates
- Notification Audits
- Batch Notifications
- Background Jobs

### 5.3.7 Audit Trails

Different kinds of audit trials can be adopted. Events occurring in the system can be tracked. For instance, tracking information such as the details of user who has logged in, who is logged out, who has created invoice and so on. The next level of tracking can be the tracking of changes. For instance, details of the user who has performed a particular task

can be tracked. Tracking can be performed at the field level too. For instance, details of a field updated by a particular user, on a particular date and with a particular value can be tracked and viewed. These details will be recorded in the history.

History as such can be tracked in two ways.

1. Change tracking: Change tracking involves routine tracking of changes.
2. Snapshot recording: The history table is added continuously with all the appended history records.

Tracking history can be configured at a tenant level. For a few cases, change tracking is adopted and for few other cases snapshot recording is adopted.

**Types**

- Event Auditing
- Snapshot recording
- Change tracking
- Performance Audits

**When to Audit**

- Time based audits
- Event based audits

### 5.3.8 Scheduling

One of the common needs in any software is the ability to run certain tasks at certain pre-configured intervals. Therefore, one of the important utilities in a framework is a scheduler. Scheduler has two parts, one addressing the frequency of the schedule and the second part refers to the task that needs to be executed. Scheduler can be invoked from any part of the application.

### 5.3.9 Workflow

**From msdn.microsoft.com**: Workflow is fundamentally about the organization of work. It is a set of activities that coordinate people and / or software. Communicating this organization to humans and automated processes is the value-add that workflow provides to our solutions. Workflows are fractal. This means a workflow may consist of other workflows (each of which may consist of aggregated services). The workflow model encourages reuse and agility, leading to more flexible business processes.

In a SaaS parlance, workflow will vary from one tenant to another tenant. For example, if you look at budget approval workflow, in certain tenants, they may have 1 level approval and in certain tenants, multi-level approval would be required for approving budget for certain items.

SaaS product should be easily configurable to accommodate different workflow requirements for different application scenarios.

### 5.3.10 Business rules

A business rule engine (BRE) is a component of software allowing programmers or end users to change the business logic in any real world enterprise application. To carry out a business policy or procedure, a business rule or statement is required. Business logic uses data in a database and a sequence of operations to carry out the business rule. A business rules engine (BRE) separates business logic from your mission-critical applications in order to gain agility and improve operational performance. To get the most benefit from this application architecture, you need a business rules engine that:

- Empowers business users to create and manage business rules with minimal involvement from IT staff.
- Supports sophisticated, powerful rules that can capture your business workflow and your policies and procedures in all their dynamic complexity.
- Integrates seamlessly with your existing IT assets and scales for enterprise-class performance.

## 5.4 SaaS Operational Features

Management of a SaaS product requires numerous backend operations. For instance, the entire gamut of the tenant system would have to be maintained, subscription packages would have to be managed and so on. The operation module helps in automating the management of SaaS application. Some of these blocks are as follows:

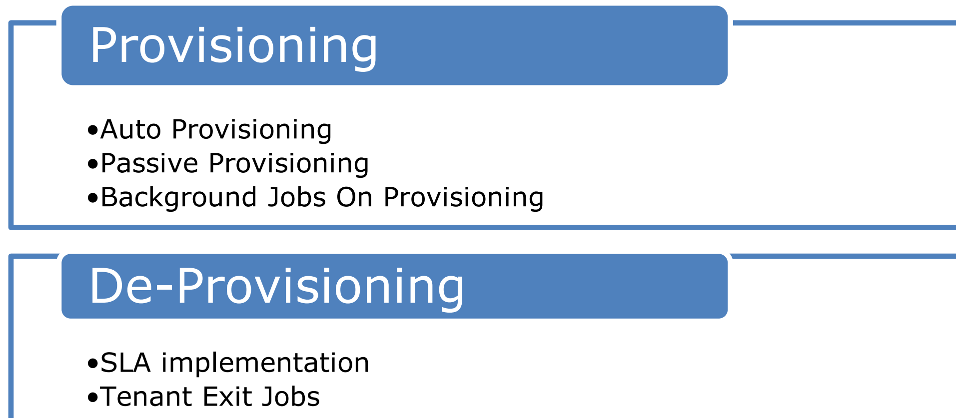
### 5.4.1 Tenant Management

Tenant management primarily deals with provisioning and de-provisioning of a tenant, defining details of the tenant, activating a tenant, deactivating a tenant and mapping a tenant to a specific license module and so on. There are two major aspects to tenant management:

1. Have details of the tenant and
2. Have a mechanism to provision the tenant.

Once a tenant comes into the picture, there are numerous tasks to be run and configured for the tenant. These include creating default scripts for tenant, setting up the workflow & business rules, etc. These would have to be undertaken by the framework.

De-provisioning of tenant involves instant, automatic stopping of services to the tenant, once the services to the tenant are cancelled/de-provisioned. Subsequent tasks such as capturing back up of the data base would have to be activated when a tenant is de- provisioned.



### 5.4.2 Subscription Management

Subscription Management helps grouping different features or modules in the system. It packages them for licensing the same to the user. It is required that the system possesses the capability to handle complex subscription models.

Subscription management is a vital aspect to be handled under operations management. Subscription management facilitates and should allow saving what each of the tenant is purchasing. It should also allow grouping of multiple features into modules and multiple modules as a package. Tenants should be able to pick the package they require to subscribe. Further, the license module should also be able to tie up with the security system internally. License module is one of the inputs to security or to the privilege based system. It validates the user actions against the corresponding roles and actions. Privilege is an intersection of the user's role and the user's purchase. The users may purchase features/packages or even be usage based plans.

**Package Management**

- Modules and Features Based Subscription
- Usage Based Subscription
  - Fixed
  - Blocks
- Seat Based Subscription
- Custom Subscription – Pick and Choose

**Access Control By Subscription**

- Privileges Based On Subscriptions

**Metering**

- Usage Audits
- Usage Reports

**Billing**

- Cloud Billing Providers

Subscription Management is a feature based subscription model that provides usage based subscription. In this model, prebuilt packages can be created such as Enterprise, Executive etc. Users can hence select specific packages of their choice. Users can also be allowed to select their customized package (A-la-carte model). This would facilitate tying the users with a licensing model. Subscription is an input to metering. Depending on the package selected by the user, as well as the usage limit level, metering is performed and usage alerts are raised.

Considering the billing aspect, numerous external billing service providers such as eVapt, Vindicia, etc. are available. One such provider can be suitably selected and their module can be integrated to subscription management.

The following screen displays a sample page for creating and selecting subscription packages. Usages are not listed here, however suitable usage based packages can be easily designed and developed.

### 5.4.3 Billing

For the purpose of automating a billing function, a separate billing module is required to be in place within the application. Further a metering component is also required within the application to track the usage data. Billing module contains rating engine which apply a specific rate plan for metered data to generate the invoice. A payment gateway too would have to be vitally integrated with the billing module to entirely automate the billing cycle.

### 5.4.4 Customization

Customization is a vital aspect to be considered in a multi-tenant scenario. There would be a single codebase serving multiple customers. Hence, customizations at different levels would have to be analyzed.

**View Customization:** Customization can be provided at the level of views such as customization of logos, number of fields in the form etc.

**Data customization:** Data customization involves allowing tenants to customize the data they require to capture in the system. There should be a capability to support the complete life cycle of a customer. For instance, adding fields, report back the data etc.

Report Customization, Business Rules and Workflows are all the vital aspects to be considered while customizing.

**Customization Units**

- View Customization
- Data Customization
- Report Customization
- Business Rules
- Workflow

**View Customization**

- URL Customization
- Logo, Themes, Layout
- Form Fields – Visibility, Editability, Mandatory
- Labels

Following is a sample of how data is extended. The following screen displays an addition transaction of a sample called Area Code to an entity. Again data type for the extended fields would have to be supported. The data type could even be a pick up list. Hence, customization capability should be enabled at such different levels of data that is captured.

Tenant Management | User Management | Access Management | Configuration Management | Rules

Manage Field — Save Back

| Add Field | | | |
|---|---|---|---|
| Entity field identifier * | Area_code | Name * | Area Code |
| Data type | Int | PickUpField | --Select PickUp List-- |
| Length | 0 | Validation regular expression | |
| Is Unique | ☐ | | |

Save Back

Following displays a sample of a Business Rule configuration:

Tenant Management | User Management | Access Management | Configuration Management | Rules

**Rules Configuration**

IF

Any of the following (OR)

CustomerInfo_NoOfEmployees >= 3500 >>

[ Add Condition ]

THEN

CustomerInfo_Discount = CustomerInfo_NoOfEmployees*5.37

[ Add Action ▼ ]

ELSE

CustomerInfo_Discount = CustomerInfo_NoOfEmployees*2.98

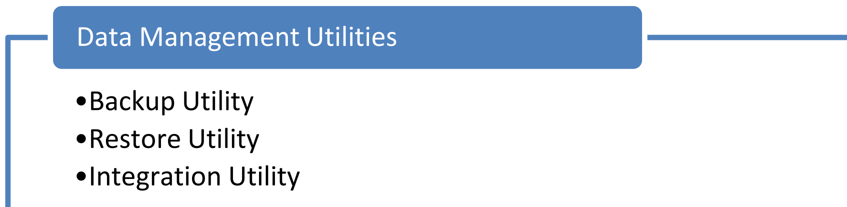
[ Add Action ▼ ]

Save Cancel

### 5.4.5 Data Management Utilities

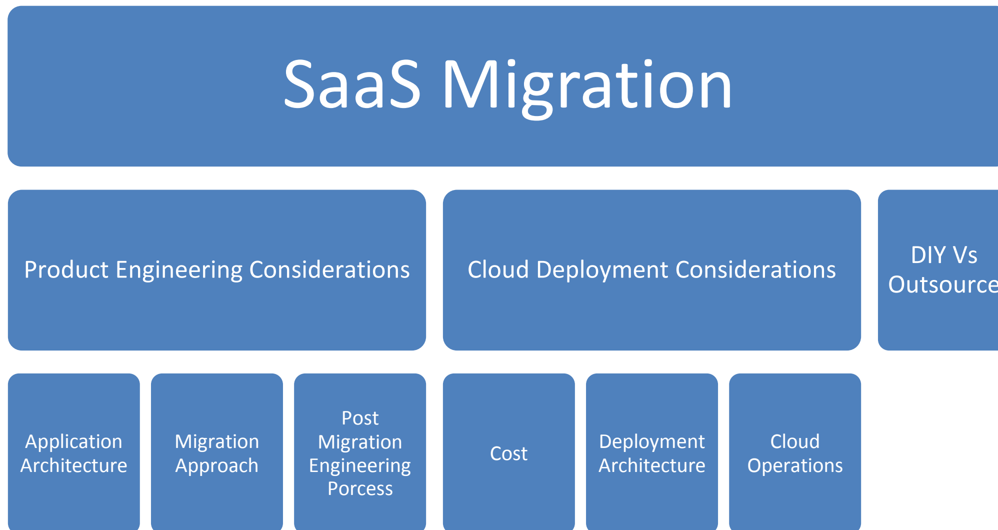
It is implicit that tenants can no longer work in data bases directly. Suitable utilities would have to be provided to them for performing basic operations such as restoring back the data at some point of time, capturing back-ups, exporting and importing of data and so on.

The following figure illustrates a list of features that can be offered to a tenant. A fee can be charged against provision of these services as well.



# 6. Migrating an existing application to SaaS

Analysis of challenges to be confronted while migrating and deploying SaaS framework based applications are more based on the Multi Tenancy Model. This analysis would delve deep into the how's and why's of the subject and focus on the aspects and factors that require to be considered while planning to deploy SaaS Solutions.




SaaS Migration can be analyzed under the following major aspects:

- **Product Engineering Considerations:** Analysis of the Application Architecture, migration approach and the engineering considerations to be made, etc..
- **Deployment/cloud Considerations:** Cloud deployment considerations include analysis of the cost economics of cloud, importance of deployment architecture and analysis of migration engineering process & post migration on cloud operations, etc.
- **Do it Yourself Vs Outsource:** How the above two aspects can be achieved, either using in-house resources or by outsourcing the tasks?

## 6.1 Product Engineering Considerations

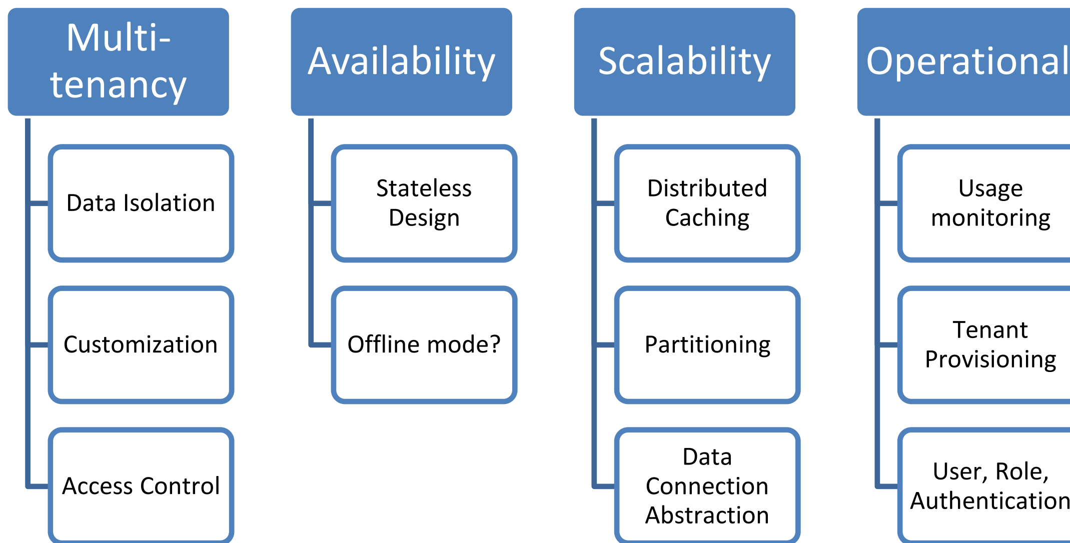
### Application Architecture

Analyzing the Architecture Aspect, it can be approximately stated that around 75% of the applications currently work based on the On-Premise model. Hence, while architecting a multi-tenant model, the primary aspects that affect the performance and effectiveness of an application would also have to be considered and architected:

- **Availability**: A stateless design is one of the factors which contribute to availability.
- **Caching**: Caching provides scalability.
- **Sharing or partitioning**: Also provides scalability.
- Data connections would have to be extracted and cannot be embedded.
- similarly analyzing the operational aspects, even though monitoring tools may be available, and tenant management system would be available at the cloud level to increase operational

efficiency. However, at the application architecture level there are certain aspects which would have to be handled effectively.

At the application layer while architecting the application, usage monitoring has to be analyzed, depending upon business model that would be used. For example, considering that a survey product is being built. While building a survey application, the aspect to be analyzed would be the number of surveys completed. Or otherwise, if an email marketing tool is being built, the aspect to be analyzed would be the number of emails sent to customers, so that the license and packaging of the product can be linked to the subscription customers based on their usage and their usage limit. These aspects would have to be handled at the application level and not at the cloud or deployment level.




Similarly, further aspects to be clearly planned for integrating into the architecture right at the design stage are:

- Ability to create a new tenant and provision a new tenant to run scripts.
- Creating default data for the tenants.
- Creating roles and authentication for various tenants.

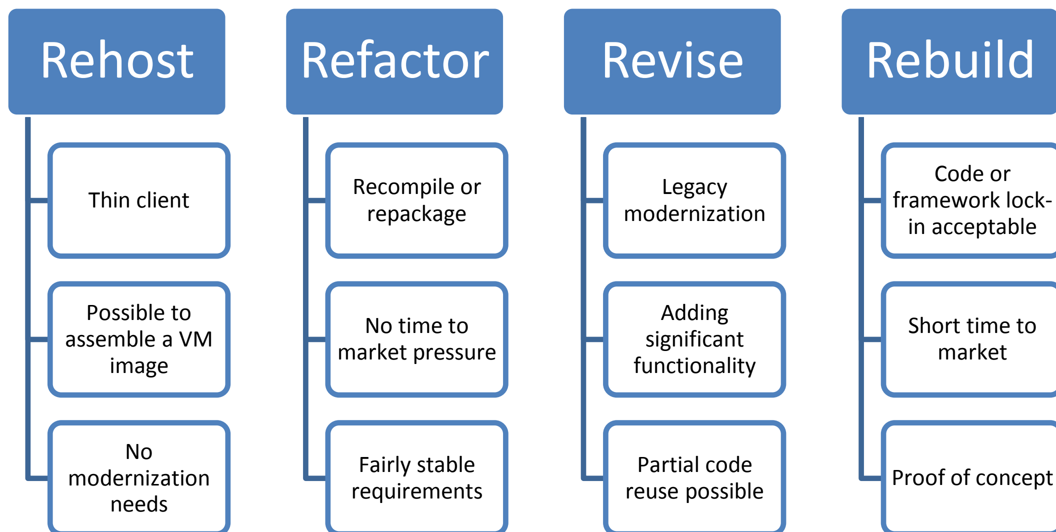
Similarly, while architecting the multi-tenancy aspect, the following vital parameters would have to be planned and paved way for right at the design stage before development of the application or migration from the on-premise application to a SaaS architecture is started:

- How to isolate data between various tenants?
- How to provide customization capability to different customers while still maintaining the single Code base?
- Where would single instance be shared amongst multiple customers?
- How to customize the access policies for different customers?

## 6.2 Migration Approach

Subsequent to analyzing and deciding on the application architecture, the next step would be to decide on the migration approach to the target architecture. The key to start working on selecting the best migration approach is to analyze the current state of affairs of the system and subsequently select suitable, right approach. There are 4 primary migration approaches:

1. **Re-host:** A re-hosting migration approach is possible wherever a thin client application prevails such as a good Web architecture. Further, it is easy for a System Administrator to assemble and create a VM image of the same. A re-hosting approach does not require the application to be updated with any type of modernization codes from legacy codes.
2. **Re-factor:** An application is hosted in IaaS is a good starting point for initiating the coding stage for achieving multi-tenancy. However, during instances that are a bit more complicated, then a re-compilation and repackaging of the application code becomes necessary. A Re-compilation and repackaging of the application code enables the application to communicate better with the infrastructure and database. During instances when there are no market pressure in terms of fetching the SaaS product to market and also during instances when customer requirements are fairly stable, then refactoring the code would be the best initial approach to migration.
3. **Revise:** However, during instances when there is a necessity to significantly add functionality to the application, then legacy codes would have to be updated and modernized. A partial legacy code re-use is also best to undertake. One of the PaaS platforms can be used to revise the codes.
4. **Rebuild:** When the existing product has become legacy from technology perspective and SaaS is the way to go from future roadmap perspective, rebuilding or vendor locking becomes quite necessary. For example, during instances when a POC for SaaS framework has to be undertaken, then Rebuilding /rewriting the application code would be the best solution.




The key to a successful migration approach lies in the selection of the correct engineering approach.

Application should not be over engineered to the entire extent before testing the SaaS Market and hence waste a lot of upfront investments and loose market feasibility of the application. This is because SaaS is a new model to transition for an application and would be a new way of undertaking business. Hence, careful and balanced engineering approach would be necessary to establish the new SaaS based application in the market.

## 6.3 Engineering Process

The following paragraphs illustrate these engineering processes through suitable examples. A vital aspect to be noted at this juncture is that, the processes illustrated below are an integral part of the larger, best practices techniques and not just exclusive for SaaS framework alone. Two aspects are quite vital for a successful SaaS framework deployment:

- Short Release cycles [the Long updates are rolled out]
- Frequent Releases [incremental Features and features]

In a Multi tenancy model, quality benchmarks both at the functional and non-functional side are extremely high. This is because a multi-tenancy model includes systematic SLAs. Further, downtime penalties are quite high and severe in a multi-tenancy model. The primary aspect to be taken care of is about gearing up the engineering processes to adequately handle these scenarios. Some of the practices followed for cloud based products even for the on premises product have proven success for agile development processes. These are iterative processes having complete traceability in the development lifecycle.

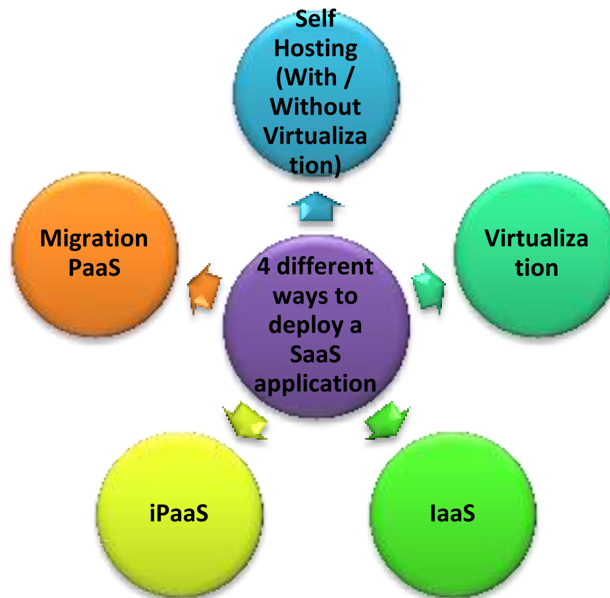
The other aspect here is about the automated patch management. In an on premise model, a quick patch would be released and applied at the customers' IT department. In a SaaS model, this responsibility of patch management lies with the vendor. Further to this, since this model is not a decentralized model, a common window is provided to all tenants, to whom patches are applied simultaneously. Patch application would have to be done in real time, without interruption as much as possible. Hence, the model should be prepared to handle such scenarios too. Similarly on the testing side, it has to be ensured that the system is well-equipped with necessary tools, practices and experienced resources to test the system Performance scalability, availability, reliability, etc. The following aspects form the key to the success of any SaaS product, especially in a multi-tenant scenario:

- Short Release Cycles
- Agile Development
- Traceability
- Traceable Patch Management
- High Quality Benchmarks
- PSRA Testing
- Continuous Integration
- Automated Functional Testing
- Continuous integration
- Daily builds
- automated testing
- A production like environment

All the above aspects are vital for achieving quality. The applicability of these factors depends on the kind of product being developed/deployed. However, researches have shown that successful SaaS Companies have adopted these approaches.

# 7. SaaS Deployment Management

There are four different ways to deploy a SaaS application.




## 7.1 Self Hosting (With / Without Virtualization)

The SaaS provider hosts the software in a data center with or without virtualization. Self hosting helps for both single-tenant and multi-tenant. However, in a single-tenant application, virtualization technique becomes more relevant. Sometimes it is mandatory to use virtualization otherwise more servers would have to be deployed.

**Virtualization:** Is the ability to spin out multiple virtualized instances from a same server. Further, each of these instances are completely partitioned from one other so that they virtually act like different servers. This is one of the ways of giving out access to a legacy application to have multiple virtual instances hosting the legacy application. The legacy application can be opened up remotely so that end users can access the same. This however carries the disadvantages of a single-tenant application. The virtualization cost could be higher and maintenance too is quite high.

**Pros:**

- Self hosting extends complete control on technology. Irrespective of the type of license or software used in the system, there are no limitations to the choice of technology that can be used.
- No vendor lock-in. The User is free to remove the application any time without losing the code base.
- They also have a substantial amount of control on functionality.

**Cons:**

- The primary disadvantage of self-virtualization is about higher capital expenditure (Capex), because a dedicated server would have to be acquired for the same. Even for virtualization, the virtualization software costs higher.

- A complete IT team bas to be mobilized and maintained to manage servers and virtualized instances. This increases the operational expenditure (Opex).
- Infrastructure security needs to be managed including network security and firewall security.
- Inflexibility to ramp up and ramp down.

## 7.2 IaaS – Infrastructure As A Service

IaaS means an Infrastructure is available for use as a service. Hence in a pay-as-you-go model, hardware is available for use from a central location. Typical example of IaaS is Amazon EC2.
IaaS provides complete computing resources on-demand. Instances can be requested at any time. All resources, right from hardware to network and firewall would be provided. Once the instance is purchased, payment would be based on the usage. This is quite an attractive model because of the lack of upfront purchase cost. This model reduces CapEx.

**Pros:**

- The primary advantage of IaaS is its complete in-built infrastructure security, taken care of by the vendor.
- Flexible ramp up and ramp down: An extra instance can be acquired easily whenever required and could be removed as easily too.
- Other advantages include:
  - Lower Total cost of ownership(TCO)
  - Full control on technology
  - No Vendor Lock-in when only a hardware infrastructure is opted for

**Cons:**

- Partial IT team required
- Application monitoring and scaling needs to be manual

## 7.3 iPaaS – Infrastructure PaaS

iPaaS is inclined towards infrastructure. Examples for iPaas are Azure or Google App Engine. iPaaS provides all the capabilities that an IaaS provides, but in addition to that it also provides the following capabilities – storage, content delivery, runtime environment. These capabilities help in expediting the development of applications and also reduce the ownership that developers need to have.

| **Pros**: | **Cons:** |
| --- | --- |
| • Auto scale<br>• Lower TCO<br>• Better control on functionality | • Partial lock-in, proprietary technology<br>• Technical design limitations |

## 7.4 Migration PaaS

On-premise applications can be developed and hosted on migration PaaS to make the same multi-tenant during run time. A typical example is SaaSGrid. A multi-tenant application is not required to be developed here. This also possesses other functionalities that are typically required for a SaaS product such as subscription, metering, billing, inbuilt access control management etc.

| Pros: | Cons: |
|---|---|
| • Faster time to market<br>• Low development cost<br>• Business user friendly development<br>• In-built scalability | • Vendor lock-in<br>• Functional limitations<br>• Tech design limitations<br>• Does not support hybrid model |

## 7.5 Cloud Deployment Considerations

Analyzing the Cloud Aspect which is a hosting model, there are three options available to organizations that require cloud adoption.

- Self Hosted
- Hosted SaaS
- Cloud

Cloud that is adopted here is the Public Cloud.

**SaaS Deployment models**

| | Self Hosted | Hosted SaaS | SaaS on Cloud |
|---|---|---|---|
| **Cost** | Capital expenditure | Estimate what you need and pay a fixed "rental" fee. | **Pay only for what you consume** |
| **Maintenance** | Requires IT resources to set up and maintain hardware and software | Requires some IT resources to set up and maintain software | **Requires minimal IT resources to maintain** |
| **Provisioning** | Takes long | Somewhat shorter but still long | **On demand** |
| **Hardware** | Servers, networks, storage + hardware for backup and scaling in your premise | Basic infrastructure is hosted. May require additional hardware for backup and scaling. | **All hardware is provided as a service. By the cloud provider** |

According to a Microsoft study the TCO reduction could be as high as 80% depending on the workload pattern

**Public Vs Private Cloud**

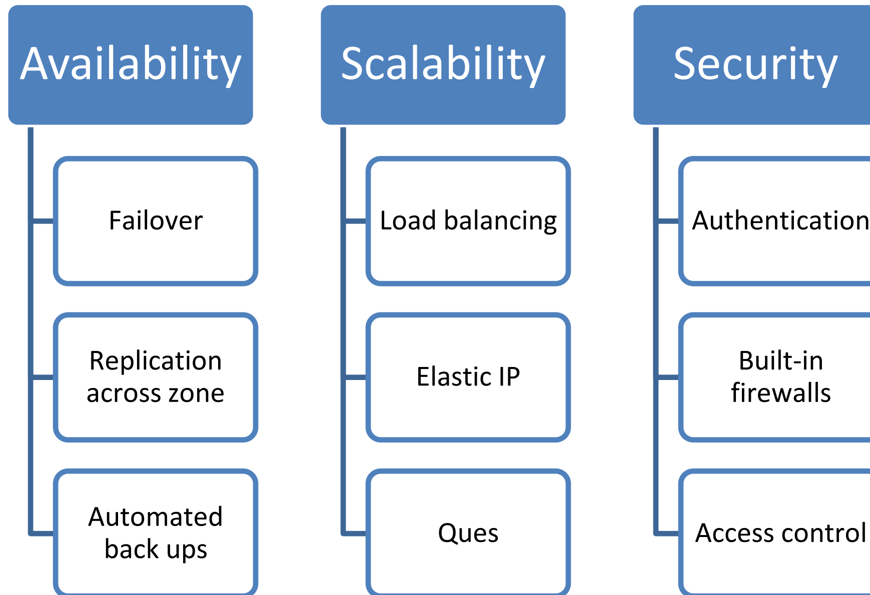
"Cloud infrastructure" provides elasticity. You can scale out or ramp-up or ramp-down the computing resources on-demand. Major difference between a public cloud and private cloud is where the cloud infrastructure is located. As the name indicates, public cloud is located outside an organization's firewall and typically shared by several customers. Private cloud sits within an organization (ISV)'s firewall. Examples of public cloud would be Amazon EC2, Microsoft Azure, etc. It is easier for vendors to consider the public cloud option rather than investing in their own infrastructure. It may however become necessary to setup a private cloud in case of demands prevailing from business issues, the regulatory issues, the compliance, privacy and network related issues. From the cost aspect, self-hosting would always have an upfront investment hosted SaaS [Pay as you go]. All said and done, a true cloud is one which is SaaS based with a pay as you go or Pay as you drain concept. Similarly when it comes to the maintaining aspect if a private data center has to be set up, then this definitely would require skilled IT resources to take care of the maintenance aspects.

You must consider the kind of application and other statutory requirements before you can finalize on using a public cloud or private cloud.

**Cloud Considerations**

There are various considerations to be made for selecting cloud as an option. Here, the application's architecture is quite vital. A cloud model would include various other quality attributes such as availability, scalability and security. Generally, failover mechanisms are provided by cloud service providers. There could be master slave relationship with replications all over the servers or using elastic IPs to bring up new instances. However, there are technologies provided by vendors such as Amazon, where the cloud strategies adopted are quite specific to the platform or possessing automated backups features. Similarly when it comes to the scalability, cloud provides more load balancing. Further, there areelastic IPs attached to the load balancing instances. For the messaging services, queues are provided so that if components are decoupled then to couple them, there are queues to connect them. Similarly a multi-tier authentication side network security has to be considered during instances of denial of service attacks. Further considerations include Ports scanning and access control using access control services, for e.g. Windows Azure.

If SaaS migration is decided upon and cloud is taken as the hosting / deployment model, then the key lies in carefully making choices on how many of the quality attributes would have to be addressed at the application level and how many of the attributes would have to be addressed at the deployment level.



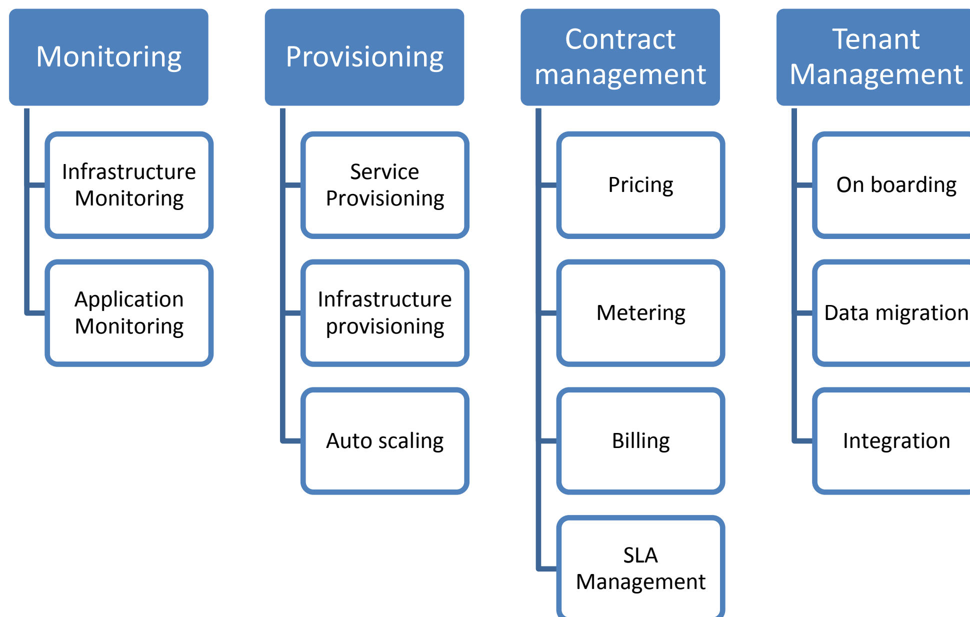

The experience here is that, if the features of cloud are leveraged at one level, say trying to achieve everything at the application level would definitely reduce investment levels. Hence, a careful balance has to be thought between how the model would have to be architected. The primary question would be that, having migrated to cloud how the operations would be managed. In a SaaS model, the responsibility of tasks performed at the infrastructure side and the IT resource of the premise customer, now is shared between the SaaS vendor and the infrastructure provider/the cloud service provider. Hence, a thorough knowledge, skills and resource base require being in place for deploying a successful cloud model.

Further to migrating to cloud, the applications and infrastructure have to be monitored. Hence, infrastructure such as usage of memory, number of CP usage, latency, and number of calls received form parts of infrastructure monitoring [Input and output]. Analyzing performance tuning, aspects such as patch management at the OS level, the application level and then monitoring uptime availability form its integral part. Hence, this is a critical aspect for cloud because continuous decisions would have to be taken, as this data are captured. This holds true especially when a fresh migration to cloud is undertaken. Similarly when it comes to provisioning, one of the ways of providing access in SaaS is through a service catalogue. Considering a scenario where multiple services are provided, in which rather than having features and modules are exposed as APIs through web service. Here, customers can invoke, provision and configure services. It is critical to analyze how to provide these services in the cloud environment. Infrastructure provisioning is on demand. The best way to achieve infrastructure provisioning is through the auto scaling feature. This is because the main advantage of cloud is its 'pay as you go' feature, which cannot work unless the auto scaling feature is available. The normal misconception is that if an application is migrated to cloud all these features will be automatically provided, which is actually not the case.

These features are facilitated by the cloud provider. There is still a layer between the applications and the cloud service provider which has to be taken care. This could be in the form of one time development like a management console or portal or using the APIs or it could be an ongoing service.

For instance, if contract management is considered, the vital aspect to be analyzed is how to facilitate pricing, metering or how to convert into billing, etc. In SLA management, for example, if the cloud provider provides 99.99 as SLAs, however the enterprise customer who buys the product expects 99.999 SLAs. In such a scenario, the basic analysis would be on how to improve on availability, how to monitor and improve SLA for all those credits and debits.

Similarly when it comes to tenant management key aspects to be considered are, creating a user, sys admin providing them access to the site, and allowing them to create users, provisioning the instances if it is not multitenant, how to help tenants to onboard their customers quickly, etc.. Earlier cases had long implementations. Hence data migrations and integrations even in semi- automated model were acceptable. However, the current scenario demands shorter time to market. Hence, all the end customers would have to be on boarded very quickly. This is a critical aspect which has to be planned as a part of the migration strategy and as part of the overall roadmap for moving to SaaS framework.



# 8. Summary of Part 1

Building a SaaS/multi-tenant application is both interesting and challenging. The delivery model changes the way how a business is done – right from business planning to pricing to delivery to customer service. SaaS has clearly redefined the traditional business model and has opened up many avenues for the ISVs as well as consumers. One has to remember that SaaS affects every department within the organization and they have to be aligned properly in order to support SaaS way of doing business.

We also saw there are several technical approaches that can help in developing a SaaS based solution. It requires careful analysis of the business and technical requirements to come out with the best suitable approach.

Delivering a successful technical SaaS solution and achieving profitability in the SaaS business model are totally different from each other. Achieving profitability depends on how well the service is delivered in cost effective manner. One of the common challenges that we have noticed with ISVs is that they tend to overlook the operational aspects, which can turn out to be a bottleneck when they scale up with more customers. It’s extremely important for the ISV to stay prepared to service the large customer base. This is where the Operational stack plays a critical role in facilitating services to customer. Modules like tenant management, configuration, customization, workflow, business rules, etc. can *not* only save time from the ISV end but can also allow the customers to address their needs through self-service portals. Therefore, do remember to include the operational modules while designing your SaaS framework.

Technical stack described in this book forms the core of the SaaS product – SaaS engineering & operational Stack. Techcello, a Gartner cool vendor, provides a cloud ready multi-tenant application development stack based on .NET. This helps ISVs reduce the time to market and also increases the quality of the end application significantly. Developers can essentially focus on building the business functionalities.

We hope this book is helpful in providing insights for conceptualization, design and development of SaaS solutions. You can contact info@techcello.com for any questions or further discussions.

Wish you a successful SaaS journey!

# 9. Brief Introduction to Part 2: Advanced Topics

Part 1 has covered the basics of SaaS/multi-tenant application development. This would give a jump-start for the technology decision makers and architects to layer their application architecture. There are few advanced topics that come in as a part of multi-tenant application development. Part 2 of this book will focus on these advanced topics:

**Tenant Hierarchy**: Imagine a situation of distributor / dealer kind of scenario for a large manufacturing company. In this case, there will be a multi-tenant hierarchy. How do you design your product for having multi-tenant hierarchy? What are the data security issues that you need to solve while introducing multi-tenant hierarchy? How can you share a single user across multiple tenants (eg. Loyalty management system where a consumer is a loyal customer for multiple retail stores).

**Workflow**: there are two kinds of workflow – one with human intervention (ideally managed by a non-technical person) and other advanced workflows to be written by a developer. How do you incorporate both kinds of workflows in multi-tenant scenario? How do you know the status of workflow at a tenant level?

**Business Rules**: These form the core of the system in several applications – for example, payroll application or insurance computation system. How do you use manage the business rules in a multi-tenant scenario?

**Development process**: What is the best way to go about building your product? Is there any specific development process that is more suited for SaaS development? What are the metrics you can use during the development? How do you release versions / patches to your product (which is already live and customers using the product)?

**Testing your product**: How do you test your product functionalities? How do you test your SaaS product for data security / releasing a function specific to a tenant (or) a group of tenants?

**Deployment management**: What are the options in front of an ISV for deploying their product? How do you auto-scale your application on the cloud? How do you monitor your application performance? What are the best practices you need to follow while deploying your product on the cloud?

**Support infrastructure**: How do you design your support mechanism for providing support for your customers? When your customers face difficulties in using a feature, how can you as ISV support? How do you integrate your support process with the development process?