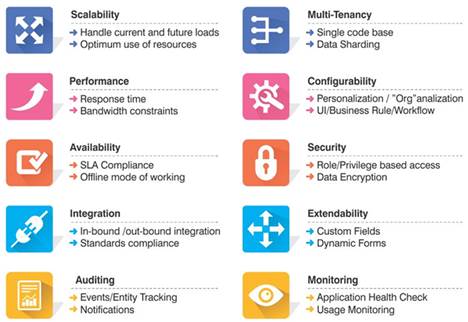
# NFRs:

<https://blog.aspiresys.com/software-product-engineering/producteering/top-10-nfr-in-software-architecture-part-1/>



**Scalability:**

In an on-demand business model, it’s extremely difficult to predict the load on the system. At the same time, you cannot plan for a peak load scenario as that will consume high levels of cost and resulting in inefficient use of resources. Therefore, the software should be designed to dynamically scale up and scale down based on the real-time load on the system. This is where software architects will have to leverage the cloud model to take advantage of the on-demand resource consumption model.

**Performance:**

With the constant increase in internet speed and bandwidth availability there is a general tendency to expect speedy response from internet based applications. Software users are going to expect the same, regardless of the type of application or the volume of processing that happens behind the screens. Therefore, architects will have to consciously take in to considerations the potential performance bottlenecks and implement designs that can help leverage concepts like asynchronous processing, micro services architecture, multi-data availability, etc.

**Availability:**

Probably the most important of all the NFR. The software has to be available (online) in the first place for other NFR to come in to play. Availability of the software is the biggest concern, particularly if the software addresses a business critical solution. Unplanned downtime of software can lead to heavy loss for Customers, and consequently can ruin the software provider’s business. Architects will have to understand the SLA that is targeted and design the deployment model in such a way that there is no single point of failure. They should also consider Recovery Time Objective (RTO), Recovery Point Objective (RPO) factors while designing their DR strategy.

Resilience is the important factor here to consider. Use of Polly.Net kind of policy based retry mechanisms in the web based apps\services play an important role.

**Integration:**

We are living in a highly connected world today, and this is only going to increase in the coming years. Customers are very particular about choosing software that not only addresses the expected features but also its ability to gel well with the existing setup at the customer’s end. This leads to a situation where the software will have to talk to a disparate set of internal and external systems. Architects will have to design the software as an open system with sufficient hooks so that integration is not only feasible but also can be completed with minimal effort.

**Auditing:**

From a Software provider perspective, the ownership of the system and its functioning is with the provider. Therefore, it’s the responsibility of the software provider to implement the appropriate measures to track the usage of the system and the events happening within it. This information will be critical for both diagnosis purposes as well as resolving conflicts with customers. Auditing design should ensure that all the user and system actions are thoroughly recorded and stored properly so that it’s easy to trace and identify the exact sequence of events that happened in the system. It’s also important to store the data change (old data vs. new data) along with the timestamp and user details that induced the change.

**Multi-Tenancy:**

Multi-Tenancy is the most complex NFR that cuts across the [**SaaS application**](https://www.aspiresys.com/saas-application-development-in-the-cloud/) as well as the other NFRs. Multi-tenancy is a design approach that facilitates a single instance of a system (saas application) to function as logical isolated instances serving the customers. Multi-tenancy architectures are complex to design but once done properly can help in significantly reducing the operational expenses (compared to a single tenant or hosted model). The complexity further increases with concepts like tenant hierarchy and virtual tenants, which provides sophisticated mechanisms to deal with varying degrees of multi-tenancy. Multi-tenancy can be applied to both web/app layer and database layer. However, it’s also possible to apply only at the web/app layer keeping the databases isolated between tenants (customers).

**Configurability:**

SaaS model of delivering applications inherently brings in the complexity of addressing multiple, (sometimes) conflicting requirements. Many SaaS applications continue to remain in single tenant/hosted model due to this reason. However, when carefully analyzed and designed, each layer of the SaaS application can be built with sufficient configurable options, which can help in achieving the customer specific requirements through configurability instead of hardcoding. The standard layers that have to be considered for configurability are UI, Branding, Authentication, Role/Privileges, business rules, business processes, integration and database.

**Security:**

Security of a SaaS application has to be looked as a comprehensive integrated engine that connects the subscription, tenant level security, usage restrictions, data restrictions, encryptions, user and role level privileges. Taking a holistic view of all these aspects in the design of security architecture is the key step for a successful SaaS application. Having this consolidated as a unified engine not only helps in manageability of the system but also facilitates changes in a systematic manner.

**Extendability:**

SaaS applications seldom are used out of the box by customers. While customers understand that SaaS applications cannot be drastically customized to meet the specific needs, but they still want to make those fine changes that will help in fitting the application with the practical implementation level details. Given the revenue model of SaaS, there is no luxury of customizing the application for each customer. This is where the configurable architecture comes to play. In addition to it, there could be certain areas in the application that will have to be extended to meet the additional requirements. For example, a customer might want to capture additional fields as part of a standard application screen. In this case, you should be able to include, store and manage the additional fields but only for that customer. Rest of the customers should not be able to see this change.

**Monitoring:**

Pro-active monitoring of SaaS application’s health can go a long way in ensuring the availability of the system and tackle any unexpected scenarios in production. There are multiple levels of monitoring including application layer monitoring, database layer monitoring, application usage monitoring, error monitoring, trial monitoring, event monitoring and alert monitoring. It’s important to design the architecture in a way the data points required for the above mentioned monitoring are easily available. It’s also important to track this information at a tenant level so that responses to customers can be expedited.

We use the Splunk dashboards for this.

RunScope is another tool used to check whether services are alive, periodically.

**Recoverability and Disaster Recovery (D R systems)**

We will discuss these non-functional requirement groupings together as there are a **significant amount of common ground** and overlap between the two, example below:

* The database must run a full backup once a week with incremental backups daily,
* The database backup must run after any overnight batch processing and not compromise the batch window,
* A disaster recovery test will be scheduled and executed annually to ensure recovery from primary site failure is achievable, and the steps to perform this process are well documented and regularly revised.
  + Your Jenkins pipelines must also run to update DR servers for the relevant software releases along with your productions server deployments.

[Infrastructure as code](https://en.wikipedia.org/wiki/Infrastructure_as_code) is important as it allows you to build environment from **machine readable definition files** when developing using [DevOps principles](https://en.wikipedia.org/wiki/DevOps). This should be considered when starting to build your **recovery strategies**.

The use of the tools like Chef (Infrastructure as Code), which lets you write the ruby scripts to manage the hardware and systems.

Difference between On-Premise, IaaS, PaaS and SaaS.

A screenshot of a cell phone

Description automatically generated

Disaster Recovery:

2 regions are made available

1. London – data centre
2. Frankfurt – data centre

If there is no response for 6 minutes from a given data centre the automatic switch over will happen to another region data centre.

This is the set up for Web servers. For database server (which is SQL server, doesn’t have sharding or clustering options) we have another database server in other region, which will do periodic sync. Due to the slight delay existence for sync up, there may be a data loss. So the recovery is done from the transaction logs.

Fault Tolerance:

We have around 10 servers updated with the apps and services any time. Out of this 10, 6 will be active servers and 4 are passive servers. Whenever a server goes down or increase in the load, the load balancer will automatically bring up these passive servers into active servers list.

We are in the process of migration to Azure, where it supports automatic scaling of instances (scale out).

SLA Management:

We have 100s of services running, their dependency will add upto the SLA or response time. Hence the Apigee gateway is configured to invoke these services in parallel and aggregate the results.

Scalability:

Service instances are controlled via Load balancer (which manages 6 active and 4 passive servers)