aws partner network

Select Consulting Partner

Building a Multi Tenant SaaS Architecture on AWS

By ClickIT SMART TECHNOLOGIES

Multi tenant Architecture SaaS Application on AWS

Introduction

Multi tenant Architecture is a Software Architecture that runs multiple single instances of the software on a single physical server, which serves multiple tenants. Multitenancy is another common term for this practice in which multiple tenants shared the memory of a server, dynamically allocated and cleaned up as needed.

SaaS applications are the new normal nowadays, and software providers are looking to transform their web applications into a Software As a Service application. For this, the only solution is to build a **Multi tenant Architecture SaaS Application.**

The next points are what we will explore in a Multi tenant architecture for your SaaS application, and my contributions will be:

Index

- **1**. Multi tenant architecture benefits
- **2**. SaaS Technology stack for an Architecture on AWS
- **3**. Which Multi tenant architecture suits better for your SaaS Application on AWS?
- **4**. Types of Multi tenant SaaS architectures: Application Layer
- **5**. Types of Multi tenant SaaS Architecture: Database layer
- **6**. Application Code Changes
- **7**. In a nutshell for Python Django
- **8**. Wildcard DNS Subdomain – URL based SaaS application.
- **9**. Web Server Setup – Nginx configuration to support Multi tenant SaaS applications.
- **10**. Follow the 12-factor methodology framework or die trying!
- **11**. What are the Multi-tenant SaaS architecture best practices?
- **12**. Conclusions

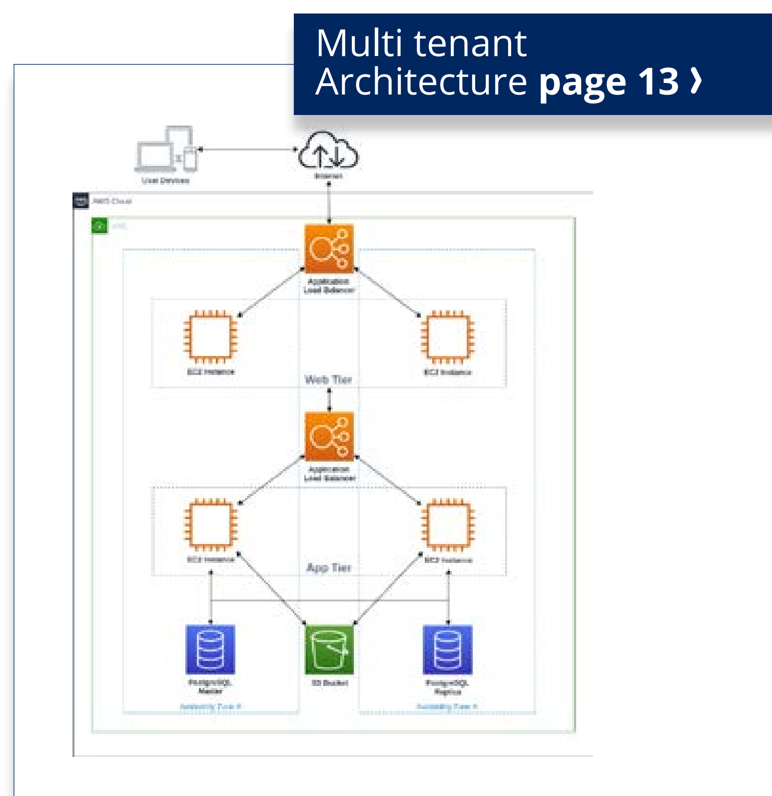
Have you ever wondered how Slack, Salesforce, AWS (Amazon Web Services), and Zendesk can serve multiple organizations? Does each one have its unique and custom cloud software per customer? For example, have you ever noticed that, on Slack, you have your own URL '*company.slack.com*'? You probably thought that in the background, they created a particular environment for your organization, –application or codebase–, and thinking that slack customers have their own server/app environment. If this is you, you might have assumed that they have a repeatable process to run thousands of apps across all their customers. Well, definitely No. And the real solution is a **Multi tenant architecture SaaS application on AWS.**

I will start with this impressive fact: **70% of all web applications are considered SaaS applications according to IDC Research.**

If you know about SaaS architecture and multi-tenant, you are probably covering 70% of the web application architecture landscape that would be available in the future.

This research is intended to showcase an overview of the strategies, challenges and constraints that DevOps and Software Developers are likely to face when architecting a SaaS multi-tenant application.

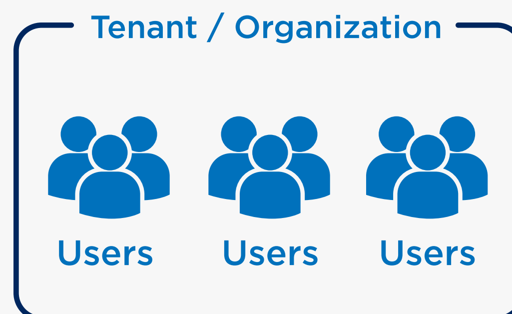
Multi tenant Architecture **page 13** ›



> **70% of all Web Apps are SaaS**

Tenant **=** Organization / Client / Customer

User **=** a user inside a tenant. A tenant/Organization can have multiple, and even, thousands of users.



1 Multi tenant architecture benefits:

The adoption of a Multi tenant architecture approach will bring extensive valuable benefits for your SaaS application.

Let's dive into the next contributions:

a) A Reduction of server Infrastructure costs utilizing a Multi tenant architecture strategy. Instead of creating a SaaS application/environment per customer, you include one application environment for all your customers. This enables your AWS hosting costs to be dramatically reduced from hundreds of servers to a single environment.

b) One single source of trust.Let's say again you have a customer using your SaaS, imagine how many code repositories you would have per customer? At least 3-4 branches per customer and this would be a lot of overhead, misaligned code releases. Even worse, visualize the process of deploying your code to the entire farm of tenants; it is extremely complicated. This is unviable and time-consuming. With a **multi tenant SaaS architecture**, you avoid this type of conflict, where you'll have one codebase (source of trust), and a code repository with a few branches (dev/test/prod). By following below practices, –with a single command (one-click-deployment)–, you will quickly perform the deployment process in a few seconds.

C) Cost reductions of Development and time to market. Having a single codebase, SaaS application environment, multi tenant database architecture, centralized storage and APIs, and following the *12-factor methodology*; all this will allow you to reduce development labor costs, time to market, and operational efficiencies.

Quickly Scale Your Business with a DevOps Team

Let's Start!

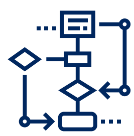
1.1 What is a Multi Tenant Architecture?

First of all, you need to understand what a single tenant architecture and a **Multi tenant architecture** is. *Single Tenant vs Multi Tenant: SaaS Architecture*.

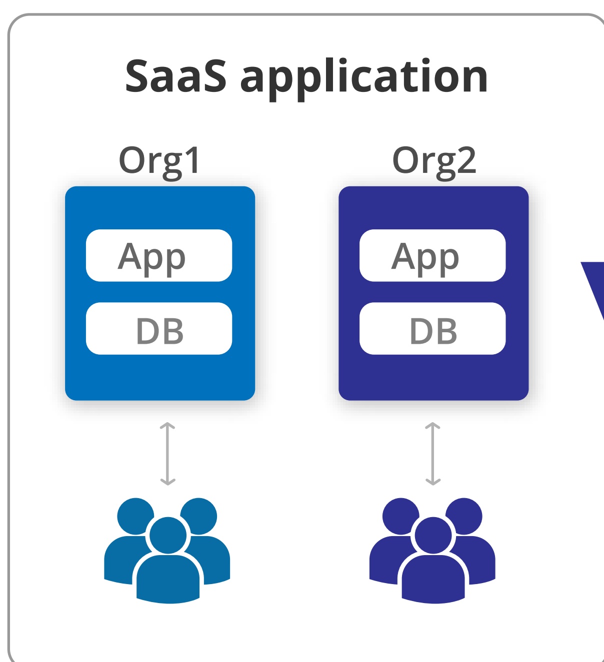
A single-tenant architecture (siloed model) is a single architecture per organization where the application has its own infrastructure, hardware, and software ecosystem. Let's say you have ten organizations; in this case, you would need to create ten standalone environments, and your SaaS application or company will function as a single tenant architecture. Additionally, it implies more costs, more maintenance, and a level of difficulty to update across the environments.

Multi-tenant architecture is an ecosystem or model, in which a single environment can serve multiple tenants utilizing a scalable, available, and resilient architecture. The underlying infrastructure is completely shared, logically isolated, and with fully centralized services. The multi-tenant architecture evolves according to the organization or subdomain (*organization.saas.com*) that is logged into the SaaS application; and is totally transparent to the end-user.

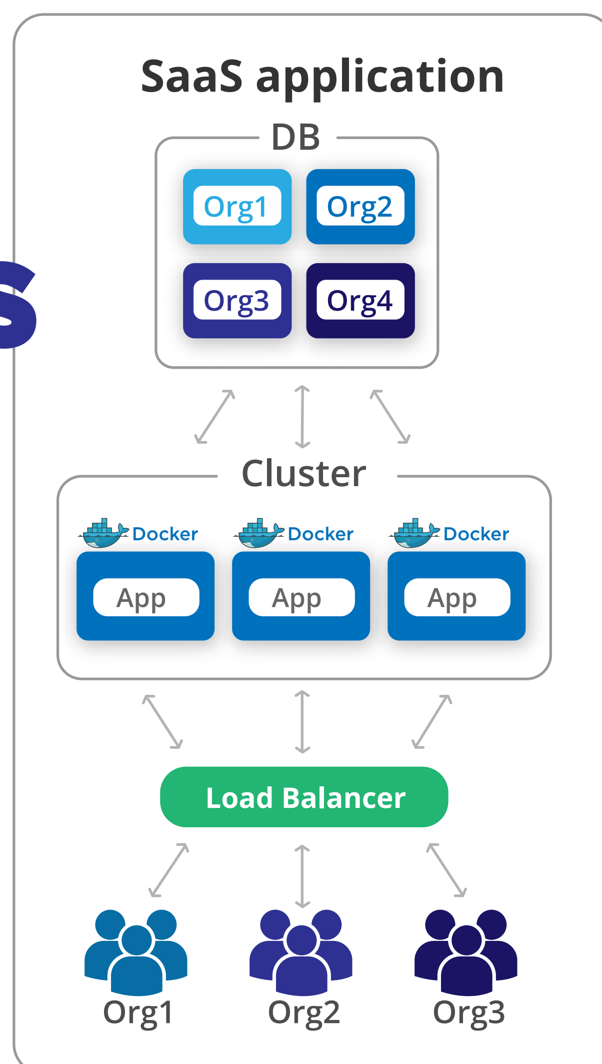
Bear in mind that in this paper, we will discuss two Multi tenant architecture models, one for the application layer and one for the database layer.



Single Tenant vs Multi tenant



VS



2 SaaS Technology stack for an Architecture on AWS

To build a multi tenant architecture, you need the correct AWS web stack, including OS, language, libraries, and services to AWS technologies. This is just the first step towards creating a next-generation multi tenant architecture.

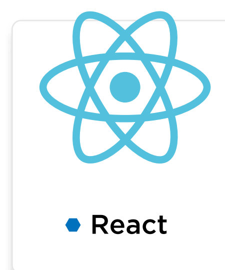
In case you haven't chosen your web stack, hereafter, I'll suggest you the ideal AWS SaaS stack. Even though we will surface a few other multi tenant architecture best practices. This article will be primarily oriented to this AWS SaaS web stack.

Let's dive into our SaaS Technology Stack on AWS:

Programming language: It doesn't really matter which language platform you select. What is vital is that your application can scale, able to utilize Multi tenant architecture best practices, cloud-native principles, and a well-known language by the open-source community. The latest trends to build SaaS applications are Python + React + AWS. Another "variant" is Node.js + React + AWS, but in the end, the common denominators are AWS and React. Probably, if you are a financial company, ML/AI, with complex Algorithms or backend work, go for Python. On the other hand, if you are using modern technologies like real-time chats, mini feeds, streaming, etc.; then go for Node.js. There is a market in the banking sector that is leveraging Java, but that's for established enterprises. Any new SaaS application better goes with the mentioned web stack. Again, this is just what I've noticed as a trend, and what the community is demanding.

Note: This data comes from a survey that we performed a few months ago for Financial Services and SaaS companies.

Ideal languages:



Cloud Provider: As a team of DevOps experts, I've noticed a cloud variation in the last two years, and which corresponds to these percentages: 70% of our DevOps implementations are based on AWS, 25% with Azure, and 5% go to GCP and Digital ocean. Each year the trend is similar, with the exception that Azure is gradually growing with the years. Those are not only my words but also ideas supported by multiple DevOps Partners. So, I strongly recommend deploying your SaaS application under AWS. It has a number of benefits; every day there is a new service available for you, and a new feature that facilitates your development and deployment.

Microservices: If you are planning to leverage the cloud, you must leverage *cloud-native principles*. One of these principles is to incorporate microservices with Docker. Make sure that your SaaS application is under microservices, which brings multiple benefits, including flexibility and standardization, easier to troubleshoot, problems isolation, and portability. Just like the cloud, Docker and microservices have transformed the IT ecosystem and will stay for a long while.

Container orchestration platform: This is a complicated and abstract decision; there are three options in AWS to manage, orchestrate, and create a microservice cluster environment.

Why Build SaaS on AWS?

First of all, did you know that you can't perform the digital transformation without the AWS Cloud? Over here, you'll be able to find more details about Why Build SaaS on AWS.

Amazon ECS. It is the natural Amazon container orchestration system in the AWS ecosystem. (Highly recommended for startups, small SaaS, and medium SaaS).

Amazon Fargate. Almost Serverless, price, and management is per task. Minimal operational effort vs. ECS. There are some studies conducted by *our DevOps team*; in terms of performance. Fargate can be slower than ECS, so for this particular case, I would recommend Amazon ECS, instead of Fargate. Another thought is that if your team is pure developers and not planning to hire a DevOps engineer, perhaps Fargate is the way to go.

Amazon EKS. It is a managed service that makes Kubernetes on AWS easy to manage. Use Amazon EKS instead of deploying a Kubernetes cluster on an EC2 instance, set up the Kubernetes networking, and worker nodes. (Recommended for large SaaS apps and a sophisticated DevOps and web development Team).

Which is the best? What shall you use? In this blog, I explain the main differences, pros, and cons of *Kubernetes vs Amazon ECS*; which is the best container orchestration? Or watch our video below.

Kubernetes vs Amazon ECS: Best Container Orchestration

Watch here

Looking for a hint? In the end, we chose *Amazon ECS*

Database. The inherent database will be PostgreSQL with Amazon RDS. However, I strongly recommend that if you have a senior development team, and are projecting a high-traffic for your SaaS application, –or even hundreds of tenants–, you'd better architect your database with MongoDB. In addition to this, utilize the best practices that will be mentioned below about Multi tenant Database.

In this case, I would go for Amazon RDS with PostgreSQL or DynamoDB (Mongodb).

> **If you are projecting a high-traffic for your SaaS application, you'd better architect your database with MongoDB**

GraphQL or Amazon AppSync. Is a query language and an alternative to a RESTful API for your database services. This new and modern ecosystem is adopted as a middleman among the client and the database server. It allows you to retrieve database data faster, mitigate the over-fetching in databases, retrieve the accurate data needed from the GraphQL Schema, and mainly the speed of development by iterating more quickly than a RESTful service. Adopting a monolithic backend application into a Multi tenant microservice architecture is the perfect time to leverage GraphQL or AppSync. Hence, when transforming your SaaS application, don't forget to include GraphQL!

Note: I didn't include this service in the AWS stack architecture diagram, because it is implemented in multiple ways, and it would require an in-depth explanation on this topic.

Automation. You need a mechanism to trigger or launch new tenants/organizations and attach it to your multi tenant SaaS architecture. Let's say you have a new client that just subscribed to your SaaS application, how do you include this new organization inside your environment, database, and business logic? You need an automated process to launch new tenants; this is called Infrastructure as Code (IaC). This script/procedure should live within a git/bitbucket repository, one of the fundamental DevOps principles. A strong argument to leverage Automation and IaC is that you need a mechanism to automate your SaaS application for your code deployments. In the same lines, automate the provisioning of new Infrastructure for your Dev/Test environments.

Infrastructure. As Code and Automation Tools
It doesn't matter which one to use, they are both highly known by the DevOps community, as well as useful and do the same job. I don't have a winner, they are both good.

Terraform (from Hashicorp). A popular Cloud-agnostic tool. Used widely for all DevOps communities. Easier to find DevOps with this skill.

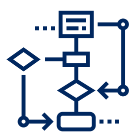
Amazon CloudFormation. Easier to integrate with Amazon Web services, AWS built-in Automation tool. Whenever there is a new Amazon technology just released, the compatibility with AWS and CloudFormation is released sooner than Terraform. Trust on an *AWS CloudFormation expert* to automate and release in a secure manner.

Message Queue System (MQS)
The common MQS are Amazon SQS, RabbitMQ, or Celery. What I suggest here is to utilize the service that requires you less operation, in this case, is Amazon SQS. There are multiple times that you need asynchronous communication. From delaying or scheduling a task, to increasing reliability and persistence with critical web transactions, decoupling your monolithic or micro-service application, and, most importantly: using a Queue System to communicate Event-driven Serverless applications (Amazon Lambda functions).

Caching System. AWS ElastiCache is a caching and data storage system that is fully scalable, available, and managed. It aims to improve the application performance of distributed cache data and in-memory data structure stores. It's an in-memory key-value store for Memcached and Redis engines. With a few clicks, you can run this AWS component entirely self-managed. It is essential to include a caching system for your SaaS application.

Release Products Faster with our Nearshore DevOps Team
Let's Start!

Amazon CloudFront CDN and Amazon S3 for your static content
All static content, including images, media and HTML, will be hosted on *Amazon S3*, –the cloud storage with infinite storage and elasticity. In front of Amazon S3, we will include AWS CloudFront to cache the entire static content and reduce bandwidth costs. Integrating this pair of elements is vital.



2.1 SaaS Multi tenant web stack architecture and diagram on AWS (Amazon Web Services).

User Devices
Internet
CI / CD
AWS Cloud
Infrastructure Managment
Cloudfront Distribution
Developers Team
(Plan)
VPC
Public subnet
Public subnet
S3 Media
Application
Load Balancer
DevOps Team
Github
(Code)
Github
(Control Version)
Private subnet
EC2 instance contents
Organization 1
EC2 instance contents
Organization N
Microservice A
Microservice B
Microservice C
Microservice A
Microservice B
Microservice C
ECR
Docker Images
(Package)
ECS Cluster
Jenkins Pipeline
(Build / Deploy)
Terraform
Create new tenant
Private subnet
SQS
Message Queue
PostgreSQL
Organization 1
Elastic Cache
PostgreSQL
Organization N

3 Which Multi tenant architecture suits better for your SaaS Application on AWS?

It is paramount to decide which multi tenant architecture you'll incorporate in your SaaS application from the application and database layer. We will explore the two layers needed to enable your application to act as a real SaaS application.

As a side note, we will discuss two types of multi tenant architecture: Application layer Multi-tenancy and Database layer Multi-tenancy.

4 Types of Multi tenant SaaS architectures: Application Layer

4.1 SaaS Monolithic architecture for Multi tenant SaaS applications

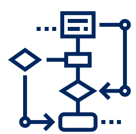
Probably, if you haven't seen this article before, –or if you have already developed and architected your own SaaS application–, I'm sure you have fallen into this approach. The monolithic components include EC2 instances in the web tier, app tier, and Amazon RDS with MySQL for your Database. The monolithic architecture is not a bad approach, with the exception that you are wasting resources massively in the mentioned tiers. At least around 50% and 70% of your CPU/RAM usage is wasted due to the nature of the monolithic (cloud) architecture.

Cons:

- Waste of AWS resources.
- Not very fault-tolerant per service.
- If the app tier goes down, the whole backend is down.
- When deploying your code, you have to deploy all your services within the app-tier, and not just with a specific isolated service.
- Not flexible to maintain.
- Slows time to market.
- HIPAA and PCI compliance constraints.

Pros:

- Easy-to-build approach.
- Minimal configuration.
- Multi tenant database.



Multi tenant architecture example: Monolithic diagram

User Devices
Internet
AWS Cloud
VPC
Application Load Balancer
EC2 Instance
EC2 Instance
Web Tier
Application Load Balancer
EC2 Instance
EC2 Instance
App Tier
PostgreSQL Master
S3 Bucket
PostgreSQL Replica
Availability Zone A
Availability Zone A

4.2 4.2. Microservices Multi tenant Architecture for SaaS with containers and Amazon ECS.

Microservices are a recommended type of architecture since they provide a balance between modernization and maximum use of available cloud resources (EC2 instances and compute units). As well as it introduces a

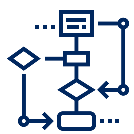
> By 2022, 90% of all new apps will feature microservices architectures that improve the ability to design, debug, update, and leverage third-party code; 35% of all production apps will be cloud-native.
>
> ***- Forbes, 2019***

decomposed system with more granular services (microservices). We won't touch much **about the Microservice benefits since it's widely expressed in the community.** However, I'll recommend you to utilize the formula of Multi-tenant architecture + AWS Services + microservices + Amazon ECS as the container orchestrator; they can be the perfect match. Mainly, consider that Amazon ECS gives fewer efforts to configure your cluster and more NoOps for your DevOps team.

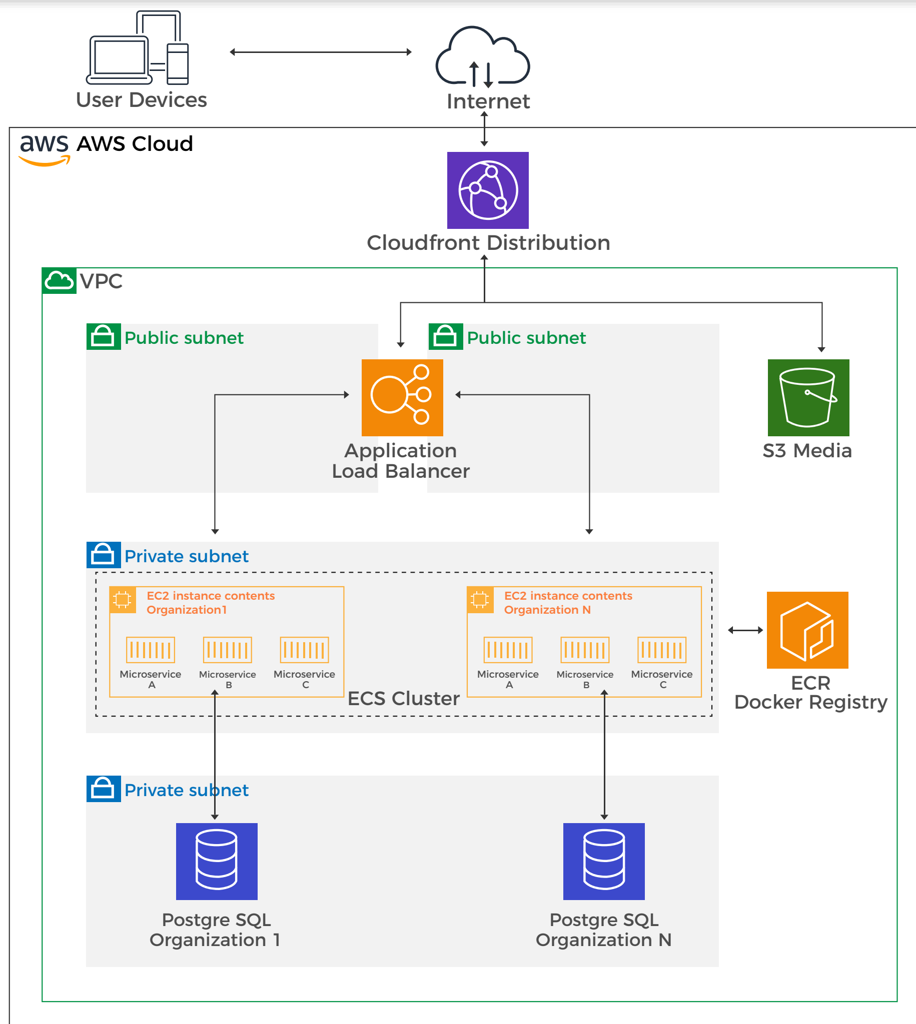
With a talented team, the best Multi-tenant architecture approach would be this use case scenario. Along the same lines, it covers the SaaS software and architecture's main attributes, including agility, innovation, repeatability, reduced cycle time, cost efficiency, and manageability.

The perfect match

Multi tenant architecture + AWS Services + microservices + Amazon ECS (as the container orchestrator).



Multi tenant Architecture example: for SaaS with microservices.



Pros:

- Adds a key DevOps principle: Loosely coupled architecture.
- Easier to deploy new code to production.
- Helps perform smaller deployment per microservice. Better agility.
- Pure and real distributed service.
- Repeatability and manageability.
- A much better level of resources utilization than monolithic.

Cons:

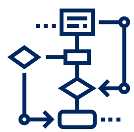
- A decent grade of complexity to create the microservices architecture and the ECS clustering.
- Amacon ECS natively lives in the AWS cloud; and you can't port this service into another cloud provider due to it is a proprietary service from AWS.

4.3 SaaS Kubernetes: Multi-tenant architecture with Kubernetes (Amazon EKS)

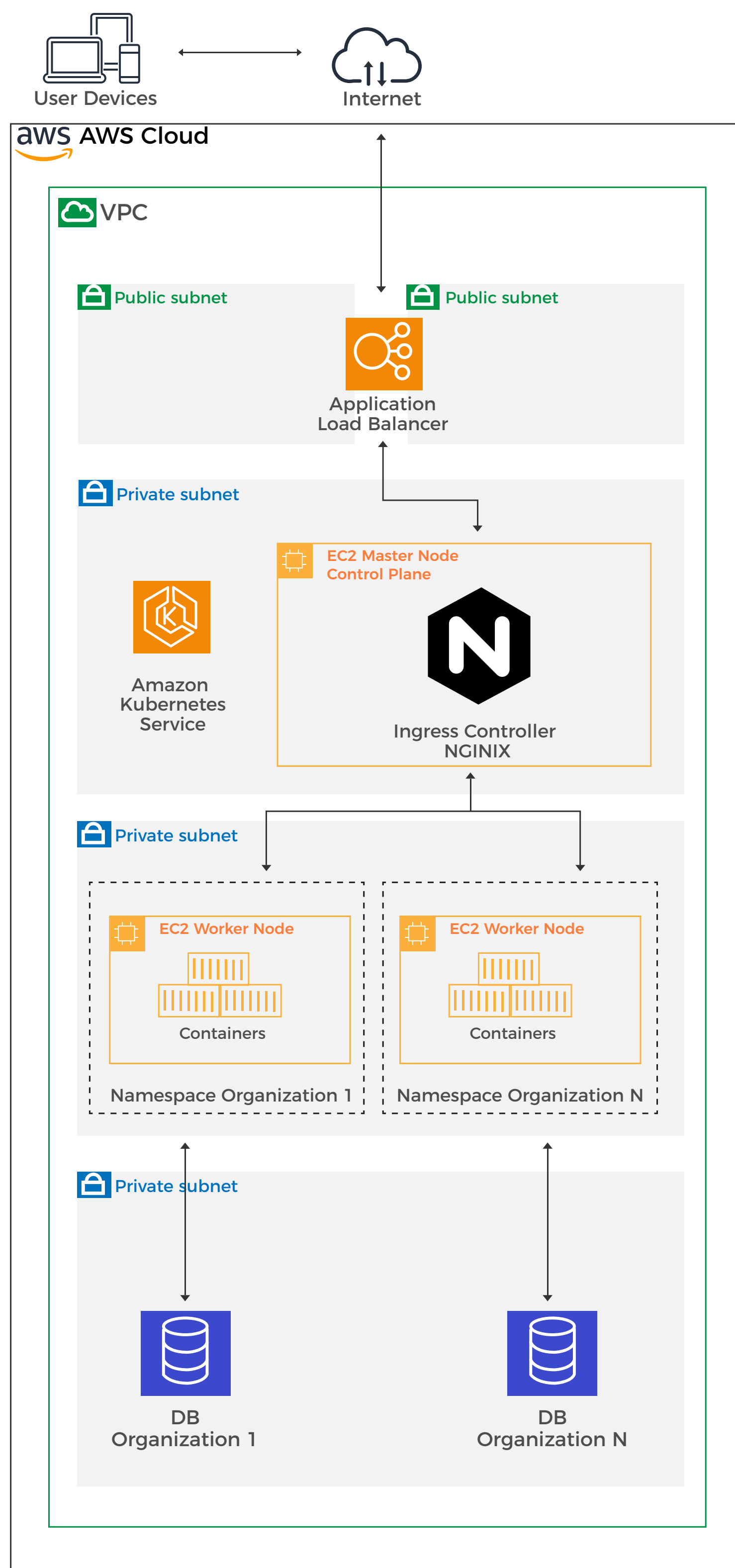
You might be wondering... what about Kubernetes or Amazon Kubernetes Service (EKS)? Well, Kubernetes is another alternative of microservice architecture which adds an extra layer of complexity in the SaaS equation. However, you can overcome this complexity by leveraging Amazon EKS (The Managed Kubernetes service from Amazon), which is a de facto service by the Kubernetes community.

So, how would a Kubernetes Multi-tenant architecture look like? The interesting part of this component from the rest of the architectures is that it provides the use of namespaces. This attribute aids to isolate every tenant and its own environment within the corresponding Kubernetes cluster. In this sense, you don't have to create different clusters per each tenant (you could, but for another approach). Also, by using ResourceQuota you can limit the resources used per namespace and avoid creating noise to the other tenants. Another point to consider is that if you would like to isolate your namespaces, you need to include Kubernetes Network policies because by default the networking is open, and can communicate across namespaces and containers.

Here is a comparison of *Amazon ECS vs Kubernetes.* You can also visit our youtube channel and watch a video that compares and declares which is *the Best Container.* Alternatively, –if you have a SaaS enterprise–, I'll recommend better to control your microservice via Amazon EKS or Kubernetes since it allows you to have more granular changes.



Kubernetes Multi-tenant SaaS Architecture diagram



Pros:

- Same pros as the microservices architecture with Amazon ECS
- Exceptional in-depth custom SaaS configuration.
- Used more by enterprise SaaS companies.

Cons:

- The classic, a higher learning curve vs Amazon ECS.
- A Re-architecture of your SaaS application.

* A simple Multi-tenant architecture with Kubernetes and siloed by Kubernetes Namespaces.

4.4 Serverless Multi tenant SaaS Architecture

The dream of any AWS architect is to create a Multi tenant SaaS architecture with a Serverless approach. That's a dream that can come true as a DevOps or SaaS architect, but it especially adds a fair amount of complexity as a tradeoff. Additionally, it requires a reasonable amount of time of collaboration with your dev team, extensive application of code changes, and a transformative Serverless mindset. Given said that, anyhow in a few years, it will be **the ultimate solution, and all depends on the talent, capabilities, and use case.**

Serverless is disrupting the IT stack, and you still on-premises? Go through this paper I created a few months ago which shows more details about the serverless ecosystem.

Pros:

- Truly based on consumption by milisecs/secs.
- End-to-end serverless application.
- A microservice per each event call.
- No need to worry about scalability, downtime, or availability. It is all built-in in the serverless ecosystem.
- Serverless simplifies much more the deployments per function.

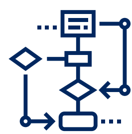
Cons:

- You need to refactor the entire application into a Serverless ecosystem
- It adds much more complexity than previous architectures.

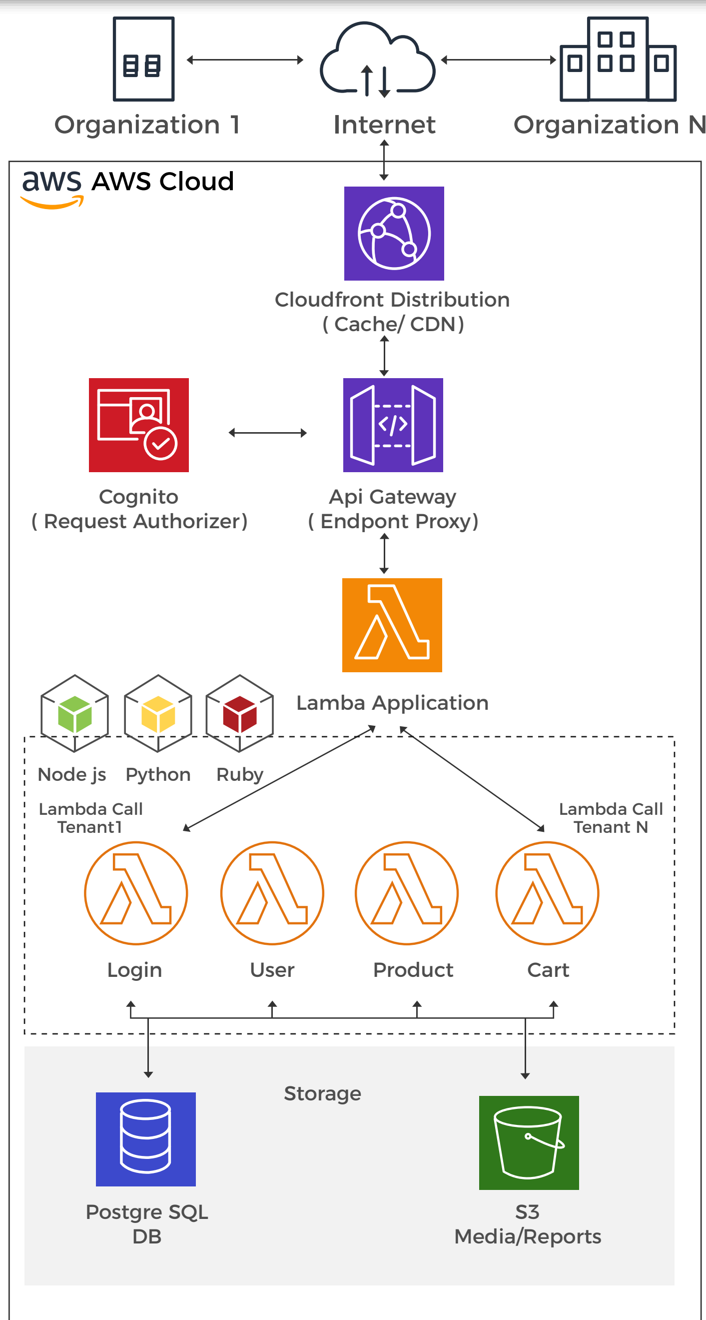
A Serverless SaaS architecture enables applications to obtain more agility, resilience, and fewer development efforts, a truly NoOps ecosystem.

At a high level, what are the new parts of this next-generation serverless SaaS architecture? Every call becomes an isolated tenant call, either going to a logical service (Lambda function) or going to the database data coming from the Amazon API Gateway as an entry point in the serverless SaaS application. Now that you have decou-

pled every logical service, the authentication and authorization module needs to be handled by a third-party service like Amazon Cognito, which will be the one to identify the tenant, user, tier, IAM tenant role, and bringing back an STS token with these aspects. Particularly API Gateway will route all tenant functions to the correct lambda functions matching the STS Token.



Multi tenant architecture example: For SaaS with Serverless.



5 Types of Multi tenant SaaS Architecture: Database layer

The multi-tenancy concept comes with different architecture layers. We have already advocated the multi-tenancy application layer and its variants. Now, it is time to explore multi-tenancy in the Database layer, which is another aspect to discover. Paradoxically, the easiest and cost-effective Multi-tenant database architecture is the pure and real database multitenancy.

As a next step, you need to evaluate what Multi tenant database architecture to pursue with tables, schemas, or a siloed database.

The following multi-tenant database architectures can be distinguished as:

5.1 Single database: A table per tenant (pure multi-tenancy and pooled model).

This database architecture is the common and the default solution by DevOps or software architects. It is very cost-effective when having a small startup or with a few dozen organizations. It consists of leveraging a table per each organization within a database schema. There are specific trade-offs for this architecture, including the sacrifice of data isolation, noise among tenants, and performance degradation -meaning that one tenant can overuse compute and ram resources from another. Lastly, every table name has its own tenantID, which is very straightforward to design and architect.

Alternative single-tenant database architecture: a Shared table in a single schema in a single schema in a single database. Perfect for DynamoDB. (We didn't cover this approach - FYI)

Pros:

- Lowest cost per tenant.
- The easiest architecture to scale your database. (However, you have always a limit).
- Great approach for hundreds/-thousands of tenants.

Cons:

- Hard to troubleshoot a single-tenant per table.
- Hard to backup and restore a single tenant per table.
- Reaching the single database limits, it becomes extremely difficult to control.
- Low tenant isolation.

5.2 Single Database: A schema per tenant (bridge model).

This Multi-tenant database approach is still very cost-effective and more secure than the pure tenancy (DB pooled model), since you are with a single database, with the exception of the database schema isolation per tenant. If you are concerned about data partitioning, this solution is slightly better than the previous one (Single/pooled DB, a table per tenant). Similarly, it is simple to manage across multiple schemas in your application code configuration. One important distinction to notice is that with more than 100 schemas or tenants within a database, it can provoke a lag in your database performance. Hence, it is recommended to split the database into two (add the second database as a replica). However, the best database tool for this approach is PostgreSQL, which supports multiple schemas without much complexity. And lastly, this strategy of –a schema per tenant– shares resources, compute, and storage across all its tenants. As a result, it provokes noisy tenants that utilize more resources than expected.

Pros:

- Low development complexity.
- This pattern is best used for a few dozens of schemas.
- More secure vs single database (A table per tenant).
- You can customize specific schemas per tenant. A Different version per schema.
- Scales horizontally.

Cons:

- It doesn't comply with PCI/HIPAA/Fedramp regulations. However, if you dont need them, who cares?
- Can get slower by the fact that loading a specific schema can be an expensive operation.
- Medium tenant isolation.
- Updating a database structure would need an update to all schemas.

5.3 **A database server** (instance) **per tenant Siloed model** (expensive, but the best for isolation and security compliance).

This technique is significantly more costly than the rest of multi-tenant database architectures, but it complies with security regulations; the best for performance, scalability, and data isolation. This pattern uses one database server per tenant, it means if the SaaS app has 100 tenants, therefore there will be 100 database servers, extremely costly.

When PCI, HIPAA or SOC2 is needed, it is vital to utilize a database siloed model, or at least find a workaround with the correct IAM roles, the best container orchestration –either Kubernetes or Amazon ECS namespaces–, a VPC per tenant and encryption everywhere.

Cons:

- Low development complexity.
- High tenant and data isolation.
- Widely used and accepted by the customer.

Architect your SaaS App with AWS

Let's Start!

Cons:

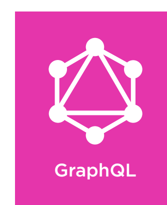
- Highest costs per tenant.
- Complex to manage N database servers (hard management).
- Hard to scale to more than 100 servers??
- Scales vertically.

Tools for a Multi tenant Database architecture:

Amazon RDS with PostgreSQL (best option).

DynamoDB (great option).

Amazon RDS with Mysql



GraphQL. As described previously, use it in front of any of these databases to increase speed on data retrieval, speed on development, and alternative to RESTful API, which helps to relieve requests from the backed servers to the client.

6 Application Code Changes.

Once you have selected your Multi tenant strategy in every layer, let's start considering what is needed to change in the code level, in terms of code changes. If you have decided to adopt Django (from Python) for your SaaS application. Then you need a few tweak changes to align your current application with your Multi-tenant architecture from the database and application layer. Fortunately, web application languages and frameworks are able to capture the URL or subdomain that is coming from the request. The ability to obtain this information (subdomain) at runtime is critical to handling dynamic subdomains for your Multi-tenant architecture. We won't cover in-depth what lines of codes we need to include in your Django application –or in any other framework–, but at least I'll let you know what items should be considered in this section.

7 In a nutshell for Python Django:

- **1** Add an app called tenant.py, a class for tenantAwareModel with multiple pool classes.
- **2** How to identify tenants? You need to give each tenant a subdomain; to do so, you need to modify a few DNS changes, Nginx/Apache tweaks, and add a utility method (utils.py). Now, whenever you have a request, you can use this method to get the tenant.
- **3** Determine how to extract the tenant utilizing the host header (subdomain).
- **4** Admin isolation.

Note: Previous code suggestions can change depending on the architecture.

8 Wildcard DNS Subdomain – URL based SaaS application.

Basically, every organization must have its own subdomain, and they are quite useful for identifying organizations. Per tenant, it is a unique dedicated space, environment, and custom application (at least logically); for example, '*org1.saas.com*', '*org2.saas.com*', and so on. This URL structure will dynamically provision your SaaS multi-tenant application, and this DNS change will facilitate the identification, authentication, and authorization of every tenant. However, another workaround is called path-based per tenant, which is not recommended, for example, '*app.saas.com/org1/...*', '*app.saas.com/org2...*', and so on.

So, the following is required in this particular section:

- A wildcard record should be in place in your DNS management records.
- This wildcard subdomain redirects all routes to your Multi-tenant architecture (either to the load balancer, application server or cluster end-point).
- Similarly, a CNAME record labeled (*) pointing to your '*app.saas.com*' or '*saas.com/login*'. An asterisk (*) means a wildcard to your app domain.
- As a final step, another (A) record pointing your '*app.saas.com*' domain to your amazon ECS cluster, ALB, or IP.

DNS Records entries:

```
*.saas.com  CNAME 'app.saas.com
app.saas.com  A  1.2.3.4   OR  app.saas.com A (alias) ▼
balancer.us-east-1.elb.amazonaws.com
```

Note: An (A) Alias record is when you are utilizing an ALB/ELB (Load Balancer) from AWS.

9 Web Server Setup - Nginx configuration to support Multi tenant SaaS applications.

Let's move down to your web server, specifically Nginx. In this stage, you will need to configure your Nginx.conf and server blocks (virtual hosts). Set up a wildcard vhost for your Nginx web server. Make sure it is an alias (ServerAlias) and a catch-all wildcard site. You don't have to create a sub-

domain VirtualHost in Nginx per tenant; instead, you need to set up a single wildcard VirtualHost for all your tenants. Naturally, the wildcard pattern will match your subdomains and route accordingly to the correct and unique patch of your SaaS app document root.Let's move down to your web server, specifically Nginx. In this stage, you will need to configure your Nginx.conf and server blocks (virtual hosts). Set up a wildcard vhost for your Nginx web server. Make sure it is an alias (ServerAlias) and a catch-all wildcard site. You don't have to create a subdomain VirtualHost in Nginx per tenant; instead, you need to set up a single wildcard VirtualHost for all your tenants. Naturally, the wildcard pattern will match your subdomains and route accordingly to the correct and unique patch of your SaaS app document root.

SSL Certificates. Just don't forget to deal with the certificates under your tenant subdomains. You would need to add them either in the Cloudfront CDN, Load balancer, or in your web server.

Note: This solution can be accomplished using Apache web server.

10 Follow the 12-factor methodology framework or die trying!

Following the 12-factor methodology represents the pure DevOps and cloud-native principles, including immutable infrastructure, dev/test and prod parity with Docker, CI/CD principles, stateless SaaS application, and more.

Willing to know more about the 12-factor methodology? This article deeply explains *how to adopt the 12-factor methodology* for any SaaS application on AWS.

11 What are the Multi-tenant SaaS architecture best practices?

11.1. SaaS architecture Best practices for your multi tenant application:

How is your SaaS application going to scale?

You should consider a strategy on how to scale your SaaS application. Here is a good Scaling strategy to follow:

1. Amazon AutoScaling, either with ec2 instances or microservices.
2. Database replication with Amazon RDS, Amazon Aurora or DynamoDB.
3. Application Load Balancer.
4. Including a CloudFront CDN for your static content.
5. Amazon S3 for all your static/media content.
6. Caching system including Redis/Memcached or its equivalent in the AWS cloud – Amazon ElastiCache.
7. Multi-availability zone set up for redundancy and availability.

More details *here* on how to scale a SaaS Application.

Build a Multitenant Architecture for SaaS

Let's Start!

Code Deployments with CI/CD (application updates across tenant apps). Another crucial aspect to consider is how to deploy your code releases across tenants and your multiple environments (dev, test, and prod). You will need a Continuous Integration and Continuous Delivery (CI/CD) process to streamline your code releases across all environments and tenants. If you follow-up on my previous best practices, it won't be difficult. The CI/CD practice is another world that your DevOps team needs to get familiar with, but with a team like us on ClickIT. CI/CD is just one of the five principles of DevOps practices, it is pretty lean for us to adopt it into your SaaS application. Ready to go?

What tools to embrace CI/CD?
CI/CD tools: Jenkins, CircleCi, or AWS Code pipelines (along with Codebuild and CodeDeploy).

DevOps Automation: Automate your new tenant creation process
How are you creating new tenants per each subscription? Identify the process of launching new tenants into your SaaS environment. You need to trigger a script to launch or attach the new Multi tenant environment to your existing Multi-tenant architecture, meaning to automate the setup of new tenants. Consider that it can be after your customer gets registered in your onboarding page, or you need to trigger the script manually.

Tools for the automation:

- Terraform (Recommended)
- Amazon CloudFormation (Trust on an *AWS CloudFormation certified team*).
- Ansible.

Note: Ensure you utilize Infrastructure As Code principles in this aspect.

My advice

If you are looking for a sophisticated DevOps team and widely known tool, go for Jenkins; otherwise, go for CircleCI. If you want to keep leveraging AWS technologies exclusively... Then go for AWS Code pipelines. But, if you're looking for compliance, banks, or regulated environments, Go for Gitlab.

How your architecture will be isolated from other tenants: Siloed compute and siloed storage

Just identify the next: Every layer of the SaaS application needs to be isolated. The customer workflow is touching multiple layers, pages, backend, networking, front-end, storage, and more bits – How is your isolation strategy?

Take in mind the next aspects:

- IAM Roles per function or micro services.
- Amazon S3 security policies.
- VPC isolation.
- Amazon ECS / Kubernetes Namespace isolation.
- Database isolation (tenant per table/schema/silo database)

Tenant compute capacity – Have you considered how many SaaS tenants can it support per environment?

Just think, you have 99 tenants, compute/database load is almost to the limits, do you have a ready environment to support the new tenants? What about the databases? You have a particular customer that wants an isolated Tenant environment for its SaaS application. How would you support an extra Tenant environment, separated from the rest of the multi-tenant architecture? Would you do it? What are the implications? Just consider a scenario for this aspect.

Tenant clean-up.

What are you doing with the tenants that are idle or not used anymore? Perhaps a clean-up process for any tenant that has been inactive for a prolonged period, or remove unused resources/tenants by hand, but you need a process or automation script.

12 Conclusions

Multi tenant architecture and SaaS applications under AWS… What a topic that we just discovered! Now you understand the whole Multi tenant SaaS architecture cycle from end-to-end, including server configuration, code, and what architecture pursues per every IT layer. As you can notice, there is no global solution for this ecosystem. There are multiple variants per each IT layer, either all fully multi-tenant, partially tenant or just silo tenants. It falls more on what you need, budget, complexity, and the expertise of your DevOps team.

I strongly recommend going for microservices (ECS/EKS), partially multi tenant SaaS in the app, and database layer. As well, include cloud-native principles, and finally, adopt the multi-tenant architecture best practices and considerations described in this article. That being said, brainstorm your SaaS architecture firstly by thinking on how to gain agility, cost-efficiency, IT labor costs, and leveraging a nearshore collaboration model (which adds another layer of cost-savings).

If you ever need a hand on how to architect your SaaS application, execute the whole AWS/DevOps projects and follow these principles, or just hire a DevOps engineer to fulfill your DevOps needs, just contact us. We help enterprises run successfully their DevOps embracement. Read more about the voice of our customers.

ClickIT is an AWS Select Partner with multiple AWS Certifications. Every engineer on ClickIT loads more than 10 DevOps projects based on SaaS architectures and cloud-native applications including PHP Laravel, React, Angular, NodeJS, Python, Go, Ruby, and Java. In the DevOps space, we work with any cloud provider (Azure, AWS, Digital Ocean, and Google Cloud), with any CI/CD, including Jenkins, CircleCI, bitbucket, and more.

In regard, Automation with Terraform

and CloudFormation is our best choice. And even better, most of our AWS/DevOps projects are following PCI, HIPAA, and SOC2 regulations. If you are a fintech, healthcare, or SaaS company, well, you know this type of requirement should be included in your processes. In case you're looking to learn more about DevOps practices and SaaS applications, don't hesitate and visit our YouTube Channel to learn more through videos.

Implement the migration of your host in the AWS

Contact Us!

About ClickIT

ClickIT is an experienced Cloud and DevOps Nearshore Solution Provider for 10 years. Our competencies are Financial Services, Healthcare, MarTech, Ecommerce, Big Data & Analytics and our Experience comes with startups and mid-large enterprises. We are AWS and GCP certified partners with an experience of helping more than 200 product and service-centric companies based out of the US with their cloud migration and DevOps initiatives.

aws partner network

Select Consulting Partner

- AWS CloudFormation Select Partner
- AWS Solutions Architect Associate
- AWS Developer Associate

ClickIT
SMART TECHNOLOGIES