# Message Passing Interface (MPI)

<https://curc.readthedocs.io/en/latest/programming/MPI-C.html#example>

**MPI\_Isend(&buffer, count, datatype, dest, tag, comm, &request)**

&request is of type MPI\_Request (fill it up with an identifier)

**MPI\_Irecv(&buffer, count, datatype, source, tag, comm, &request);**

**MPI\_Wait(&request, &status)**

**MPI\_Waitall (count, &array\_of\_requests, &array\_of\_status):** It waits for the all calls to complete

**MPI Reduce and Allreduce**

<https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/>

**MPI Scatter, Gather, and Allgather**

<https://mpitutorial.com/tutorials/mpi-scatter-gather-and-allgather/>

**Simple collective communication**

<https://enccs.github.io/intermediate-mpi/collective-communication-pt1/#allreduce>

**MPI Collectives**

Operations that are performed by all the ranks together.

**Synchronization operation** (all processes to synchronize at a point before proceeding), **Data Movement** (across all the processes like broadcast) and **Collective Computation** (add up array elements across all the processes e.g. reduce operation)

**For Synchronization:**

**MPI\_Barrier(comm)** - MPI\_COMM\_WORLD. It’s a blocking call.

For example: we have three processes (p0, p1, p2) with their respective computation to do. MPI\_Barrier is going to wait for the processes to reach the MPI\_Barrier point and once all the processes enter the MPI\_Barrier point, then only all the three processes can get out of it.

**For Data Movement:**

Broadcast, Scatters, Gather, All to All transmission of data across the communicator.

**MPI\_Bcast(&buffer, count, datatype, root, comm);**

**For Collective Computation or Collective Operations (or Global Reduction)**

One process from the communicator collects data from each process and performs an operation on that data to compute a result.

**int MPI\_Reduce(const void \*sendbuf, void \*recvbuf, int count, MPI\_Datatype datatype, MPI\_Op op, int root, MPI\_Comm comm)**

# AKKA Model

A **toolkit and runtime** for building highly concurrent, distributed, and resilient systems in **Java** and Scala.

Not any microkernel or anything that we run our application on top of. It is just a regular dependency.

**CONCEPTUAL MODEL** of concurrent computation

**WHAT IS ACTOR??**

a small, independent unit of computation that can perform tasks, communicate with other actors, and react to messages it receives.

**ACTOR** - fundamental concepts that help to model and design systems where multiple tasks or computations can happen concurrently.

Actor encapsulates **State** and **Behaviour**.

In Akka, an actor is created using the **ActorSystem.actorOf** method.

An "actor system" is a collection of actors that work together.

In Akka, an **ActorSystem** is the starting point of any Akka application that we write.

SIMPLIFIED ANALOGY OF ACTORS IN A RESTAURANT SCENARIO

Each cook in a kitchen can be viewed as an actor.

They have their own tasks **(state) →** like chopping vegetables or grilling meat

How do they communicate?

They communicate by passing orders and updates (messages).

Each cook can work independently, allowing the kitchen to handle multiple orders simultaneously without chaos.

**Key Concepts:**

**Independence:**

Imagine each actor as an independent entity, like a miniature computer program or a tiny worker. Actors work concurrently, meaning they can operate simultaneously without interfering with each other.

**Encapsulation:**

An actor encapsulates both state and behavior. This means it has its **own internal data (state)** and a **set of actions it can perform (behavior)**. The internal state is private to the actor, and it can only be accessed or modified through messages.

**Message Passing:**

Actors communicate by sending and receiving messages. This is their primary means of interaction. When one actor wants to instruct or inquire about something from another actor, it sends a message. Messages are a way for actors to exchange information and trigger actions.

**Concurrency and Scalability:**

The actor model is particularly useful for building concurrent and scalable systems. Because actors are independent, they can operate concurrently without the need for complex synchronization mechanisms. This makes it easier to design systems that can efficiently utilize multi-core processors and scale horizontally.

<https://arcagarwal.medium.com/introduction-to-akka-actors-5ec3ff032f4b>

The Actor hierarchy

**| /user**

|

| system.actorOf(props, “my-actor”)

|

**+**

**– – – – /user/my-actor**

|

| getContext()

| .actorOf(...m “my-child’)

|

**++**

|

|

| **/user/my-actor/my-child**

|

<https://getakka.net/articles/intro/what-problems-does-actor-model-solve.html>

Actors are implemented as light weight threads. But beyond lightweight threads, Actors also allow an explicit message passing interface, which threads lack. No shared memory in actors.

<https://medium.com/globant/akka-actors-in-java-4cdf1677dd0f>

To implement the Actor Model, there are a few fundamental rules to follow:

1. All computation is performed within an actor
2. Actors can communicate only through messages
3. In response to a message, an actor can:

* Change its state or behavior
* Send messages to other actors
* Create a finite number of child actors

<https://www.oreilly.com/library/view/applied-akka-patterns/9781491934876/ch01.html>

An actor ***Receives different types of messages*** and acts on them by:

* Sending messages to other actors
* Changing (mutating) it’s own state
* Changing it’s behavior (how it will react on following message)
* Creating more actors

Actors are lightweight, and easy to create thousands or even millions of them as they require fewer resources than threads.

The actors are isolated and do not have a shared memory. They have a state and the only way to change the state is by receiving the message. Every actor has its own mailbox which is similar to a message queue. Messages are stored in the actors’ mailboxes until they are processed.

Actors receive their messages into their mailbox and process messages from their mailboxes one at a time.

Actors after creation are waiting for messages to arrive. Actors communicate with each other only through messages. Messages are sent to actor’s mailboxes (inboxes) and processed in **FIFO order**. Messages are simple, immutable data structures that can be easily sent over the network.

Actors process only one message at a time. Actors are decoupled. They work asynchronously and they don’t need to wait for a response from other actors.

Actors have addresses. Actor can only communicate with another actor whose address it has.

**ADDRESS != IDENTITY**

Self-healing systems

**Pros**

Easy to Scale

Fault Tolerance

Geographical Distribution

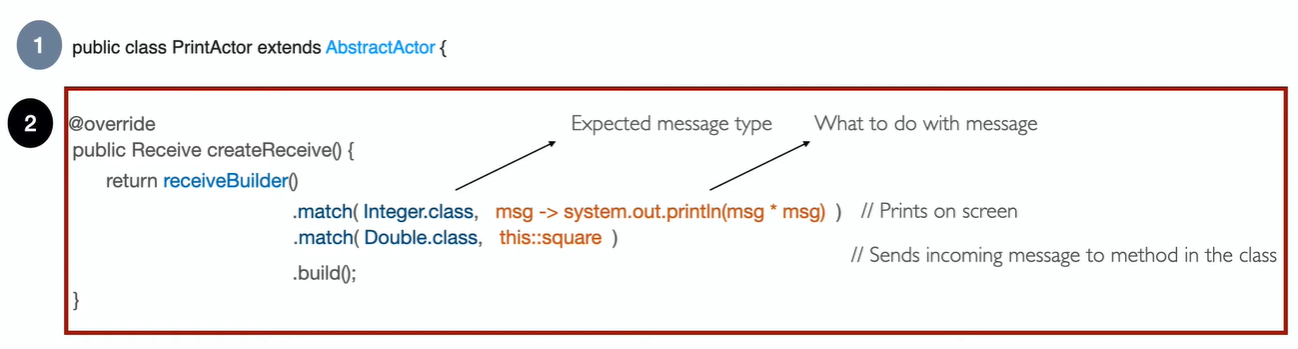
No shared state

**Cons**

Actors are susceptible to deadlocks

Actor’s mailbox overflowing

<https://developer.lightbend.com/guides/akka-quickstart-java/index.html>



**receiveBuilder()** gives us a **match()** method where we can check the incoming message type and define the behavior based on the message type.

# 

# Apache Spark

## Introduction to Apache Spark

Apache Spark is the most popular and most widely adopted data processing framework in a data lake.

[https://spark.apache.org](https://spark.apache.org/docs/latest/api/java/index.html)

**Think of Apache Spark like a basketball game!**

There are 4 pieces:

* **the driver who is the coach**
* **the data which is the basketball**
* **the executors who are the players**
* **the query plan which is the play**

When a Spark job initially starts, the coach reads the play and tells the executors where to get the ball.

Only when a shot is taken (I.e. data is output somewhere), does the basketball do anything that matters. (in tech speak: Spark is lazily executed)

Things can break though:

**a player is asked to juggle 40 basketballs and fails.**

This happens in Spark when there is data skew. Executors fail and you need to manage skew better. An option here is to have the coach distribute the basketballs more equitable with random numbers. (this is a technique called salting the GROUP BY)

**the play is too large and too complicated and the coaches head explodes**

This is driver OOM that usually happens when half way through a play, a player starts throwing basketballs at the coach and asking him what they should do next. This is when you call collect(). The coach can handle an extremely light amount of basketballs only!

**the play takes too long and the shot clock (I.e. SLA) kills the job**

Sometimes you don’t have enough initial players on the team for the job to run successfully. A good strategy here is to bump up spark.sql.shuffle.partitions because until basketball each team can have more than 5 players!

Apache Spark is an open source, wide range data processing engine with revealing development API’s, that qualify data workers to accomplish streaming in spark, machine learning or SQL workloads which demand repeated access to data sets. It is designed in such a way that it can perform batch processing (processing of the previously collected job in a single batch) and stream processing (deal with streaming data). It is a general purpose, cluster computing platform.

Spark is designed in such a way that it integrates with all the Big data tools. For example, Spark can access any Hadoop data source and can run on Hadoop clusters. Spark extends Hadoop MapReduce to the next level which includes iterative queries and stream processing.

MapReduce is a programming paradigm that allows scalability across thousands of servers in a Hadoop cluster. Spark is highly accessible and offers simple APIs in Python, Java, Scala, and R.

There is a common belief that Apache Spark is an extension of Hadoop, which is not true. Spark is independent of Hadoop because it has its own cluster management system. It uses Hadoop for storage purposes only.

<https://nag-9-s.gitbook.io/spark-notes/apache-spark-concepts-key-terms-and-keywords>

<https://medium.com/@sujathamudadla1213/what-is-skewed-join-in-apache-spark-946228d89baf>

<https://spark.apache.org/docs/latest/api/java/index.html>

<https://spark.apache.org/docs/latest/building-spark.html>

<https://medium.com/@amitjoshi7/spark-architecture-a-deep-dive-2480ef45f0be>

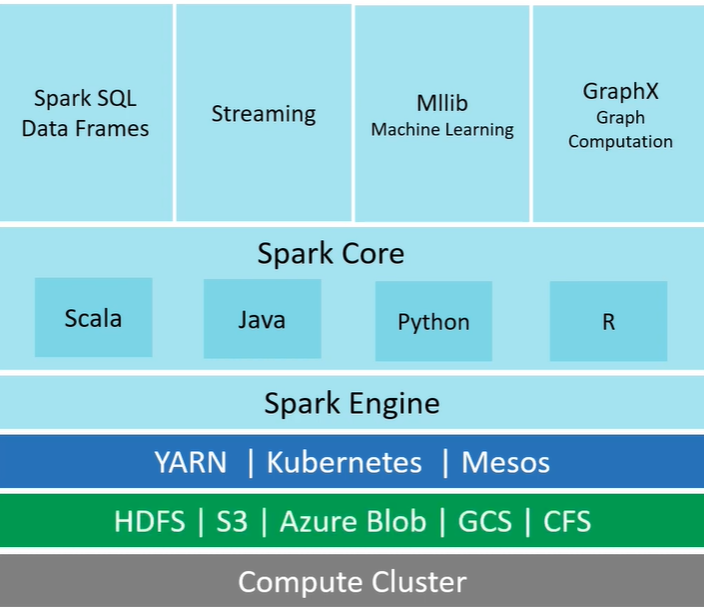
<https://www.encora.com/insights/apache-spark-architecture>

The main feature of the Apache Spark is it’s **in-memory cluster computing** that increases the processing speed of the

Apache Spark is based on two main architectures:

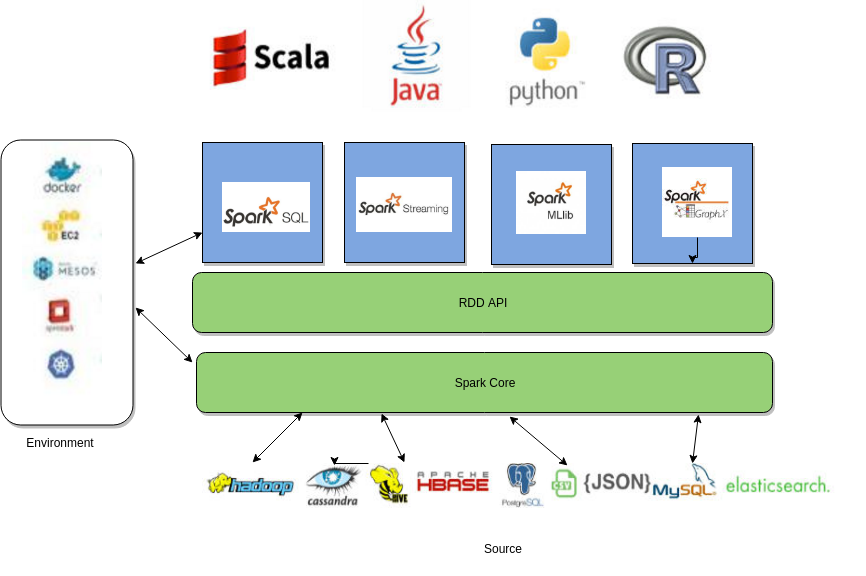
1. Resilient Distributed Datasets (RDD)
2. Directed Acyclic Graph (DAG)

## Spark Ecosystem



<https://spark.apache.org/docs/latest/cluster-overview.html>

Spark can be used for various purposes, such as real-time processing, machine learning, graph processing, and so on. Spark consists of different independent components which can be used depending on use cases. The following figures give a brief idea about Spark's ecosystem:



### Map Reduce

Map - Split, Key-Value Pair

Reduce - Shuffle, Reduce

**Map → Key-Pair → Shuffle → Reduce**

1. Distributed File System
2. No Data Movement
3. Key-Value Structure
4. Machine Failures (Re-perform the operation)

<https://towardsdatascience.com/apache-spark-101-3f961c89b8c5>

<https://spark.apache.org/docs/latest/api/scala/org/apache/spark/rdd/RDD.html>

<https://stackoverflow.com/questions/34202967/xmlns-xmlnsxsi-xsischemalocation-and-targetnamespace>

<https://www.youtube.com/watch?v=MAJ0aW5g17c>

**Maven Repository:** [https://mvnrepository.com/](https://mvnrepository.com/artifact/org.apache.spark/spark-core)

In in-memory computation, the data is kept in random access memory(RAM) instead of some slow disk drives and is processed in parallel. Using this we can detect a pattern, analyze large data. This has become popular because it reduces the cost of memory. So, in-memory processing is economical for applications. The two main columns of in-memory computation are -

1. RAM storage
2. Parallel distributed processing.

<https://medium.com/nerd-for-tech/exploring-big-data-with-apache-spark-introduction-and-key-components-a6872c581ce6>

## Workshop Explanation

SparkConf conf = new SparkConf().setMaster("local") .setAppName("wordCount");

JavaSparkContext sc = new JavaSparkContext(conf);

Next we are reading the input file using RDD's. RDD's are essentially blob's of text that you read from various sources and you can transform them into whatever you want using various operations. Here we are reading the input file from our local file system. If you want to read from HDFS, then replace the **file:///** with **hdfs:///**

**String inputFile = "file:///home/dsp/Desktop/sparkExamples/sample\_testing/resources/inputFile";**

**JavaRDD<String> input = sc.textFile(inputFile);**

Then we have our first transformation operation on the input RDD we have created in the above step.

Flat Map is an inbuilt function that takes one input and can provide any number of outputs depending on the operations used inside it.

**JavaRDD <String> words = input.flatMap(l -> Arrays.asList(l.split(" ")));**

Here we are splitting the sentence on white space characters. So, the flatmap function here returns a list of all the words in the input document and that will be stored in the RDD named words.

Next, we have another transformation mapToPair that returns a Tuple of word and the number 1.

And, a Tuple is very similar to ordered pairs in the Cartesian coordinate system. Tuple2 looks like (x,y), where x is the Key. Similarly Tuple3 will be (x,y,z) and so on.

JavaPairRDD<String, Integer> pairs = words.mapToPair(w -> new Tuple2(w, 1));

As an example, the word you in the input will be mapped to (you,1) by mapToPair function. And, since the result is a pair, we have to store it in a JavaPairRDD which supports pairs.

And, then we are doing the final transformation on the pairs that will add up individual counts of each word.

JavaPairRDD <String, Integer> counts = pairs.reduceByKey((x, y) -> x + y);

**ReduceByKey** method groups all the Tuple pairs with the same key. We have the word 'you' repeated thrice and so we have (you,1) three times. Now, (you,1) , (you,1), (you,1) will become (you,3) \* because of \* the sum we are doing inside the function. And similarly for the other words.

Then finally we are performing an action on the RDD which is where the actual computation of all the above steps takes place. collect() will return all the elements in the RDD and we are printing that using **println**, giving us the output we want.

<https://stackoverflow.com/questions/54181592/how-to-solve-this-error-type-mismatch-cannot-convert-from-liststring-to-iter>

<https://www.freblogg.com/spark-word-count-with-java>

<https://www.digitalocean.com/community/tutorials/apache-spark-example-word-count-program-java>

## Format Code

<https://formatter.org/>

<https://www.tutorialspoint.com/online_java_formatter.htm>

# 

# Cloud Computing

Cloud computing is the use of computing resources (hardware and software) that are delivered as a service over a network (typically the Internet). The name comes from the use of a cloud-shaped symbol as an abstraction for the complex infrastructure it contains in system diagrams. Cloud computing entrusts remote services with a user's data, software and computation.

Cloud computing is a general term for anything that involves delivering hosted services over the Internet. Cloud computing providers offer their services according to three fundamental models: Infrastructure as a service (IaaS), platform as a service (PaaS), and software as a service (SaaS) where IaaS is the most basic and each higher model abstracts from the details of the lower models.

The most popular and biggest cloud service providers are **AWS (Amazon Web Services), Microsoft Azure, Google Cloud Platform, IBM Cloud, Salesforce, Oracle, Red Hat,** etc.

IaaS,PaaS and SaaS are all types of cloud computing service models that a cloud service provider can offer, and you can use them on a pay-as-you-go basis via the internet.

A cloud service has three distinct characteristics that differentiate it from traditional hosting.

* **It is sold on demand**, typically by the minute or the hour;
* **It is elastic** -- a user can have as much or as little of a service as they want at any given time; and
* **The service is fully managed by the provider** (the consumer needs nothing but a personal computer and Internet access). Significant innovations in virtualization and distributed computing, as well as improved access to high-speed Internet and a weak economy, have accelerated interest in cloud computing.

***A more tempered view of cloud computing considers it the delivery of computational resources from a location other than the one from which you are computing.***

<https://www.moneyandrobots.com/p/cloud-computing-is-just-like-making>

Let’s explore Google Cloud Platform as an example:

* **Google Compute Engine**, one of its products, is an **Infrastructure as a Service (IaaS)** offering that delivers high-performance, customizable virtual machines hosted in Google’s data centers.
* **Google App Engine**, on the other hand, is a **Platform as a Service (PaaS)** offering that empowers developers to create, deploy, and scale their web applications within Google-managed data centers.
* When it comes to **Software as a Service (SaaS)**, Google offers a plethora of SaaS products, many of which are widely recognized. Google Workspace, for instance, is a valuable Business-to-Business (B2B) product, while Gmail is practically universal, used by people around the world.

**More Examples:**

**SaaS:** Gmail, Microsoft 365 (Office 365), Salesforce, Slack, Dropbox, Zoom, Adobe Creative Cloud, Zendesk, Shopify, etc.

**PaaS:** Heroku, Google App Engine, Microsoft Azure App Service, Red Hat OpenShift, IBM Cloud Foundry, Elastic Beanstalk, Docker, Firebase, etc.

**IaaS:** AWS Elastic Cloud Compute (EC2), Azure Virtual Machine, IBM Cloud, DigitalOcean, Alibaba Cloud, VMWare Cloud, etc.

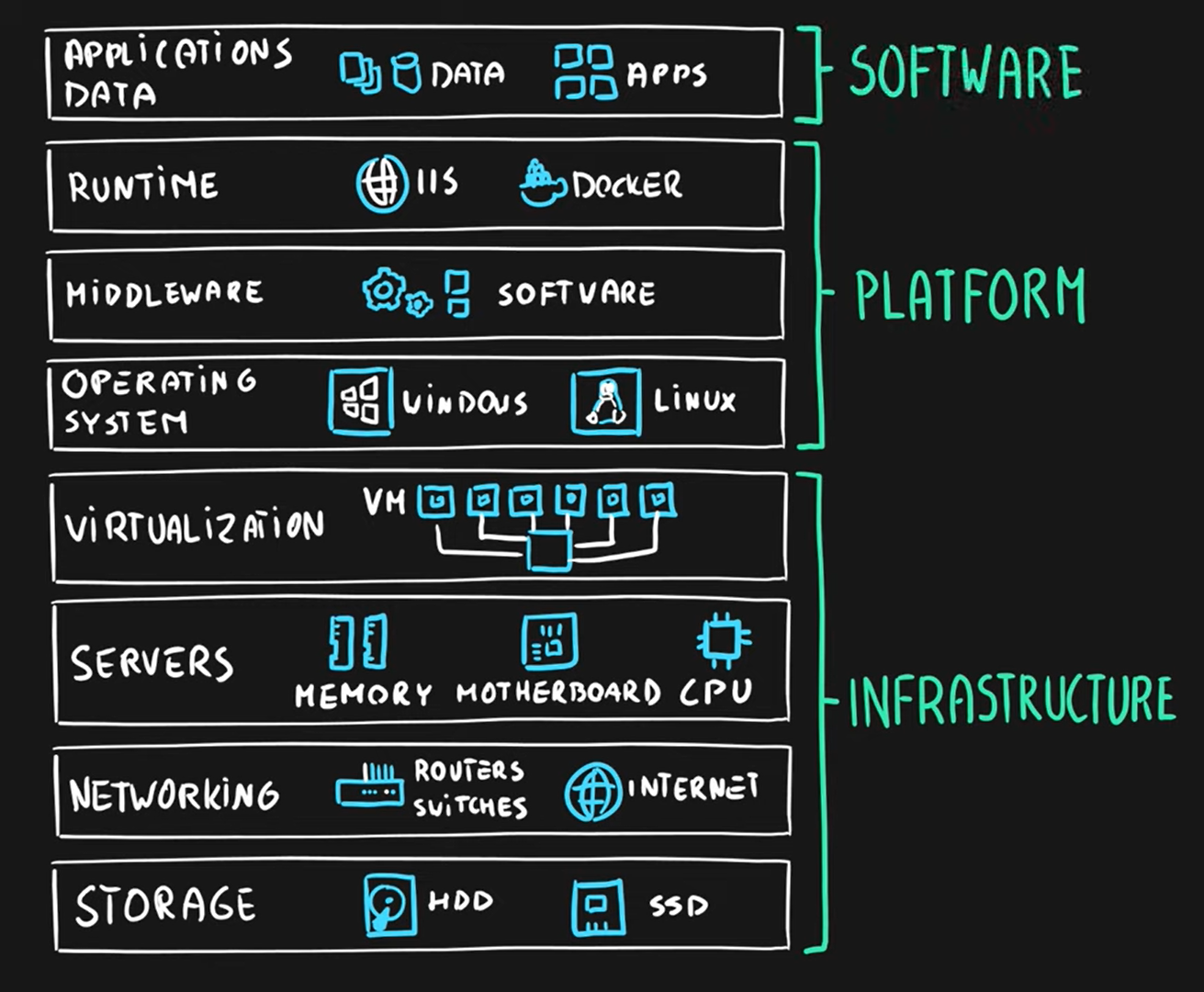
## Cloud: SaaS vs PaaS vs IaaS

<https://swfungineer.medium.com/csaas-vs-paas-vs-iaas-f388f4e79d0c>

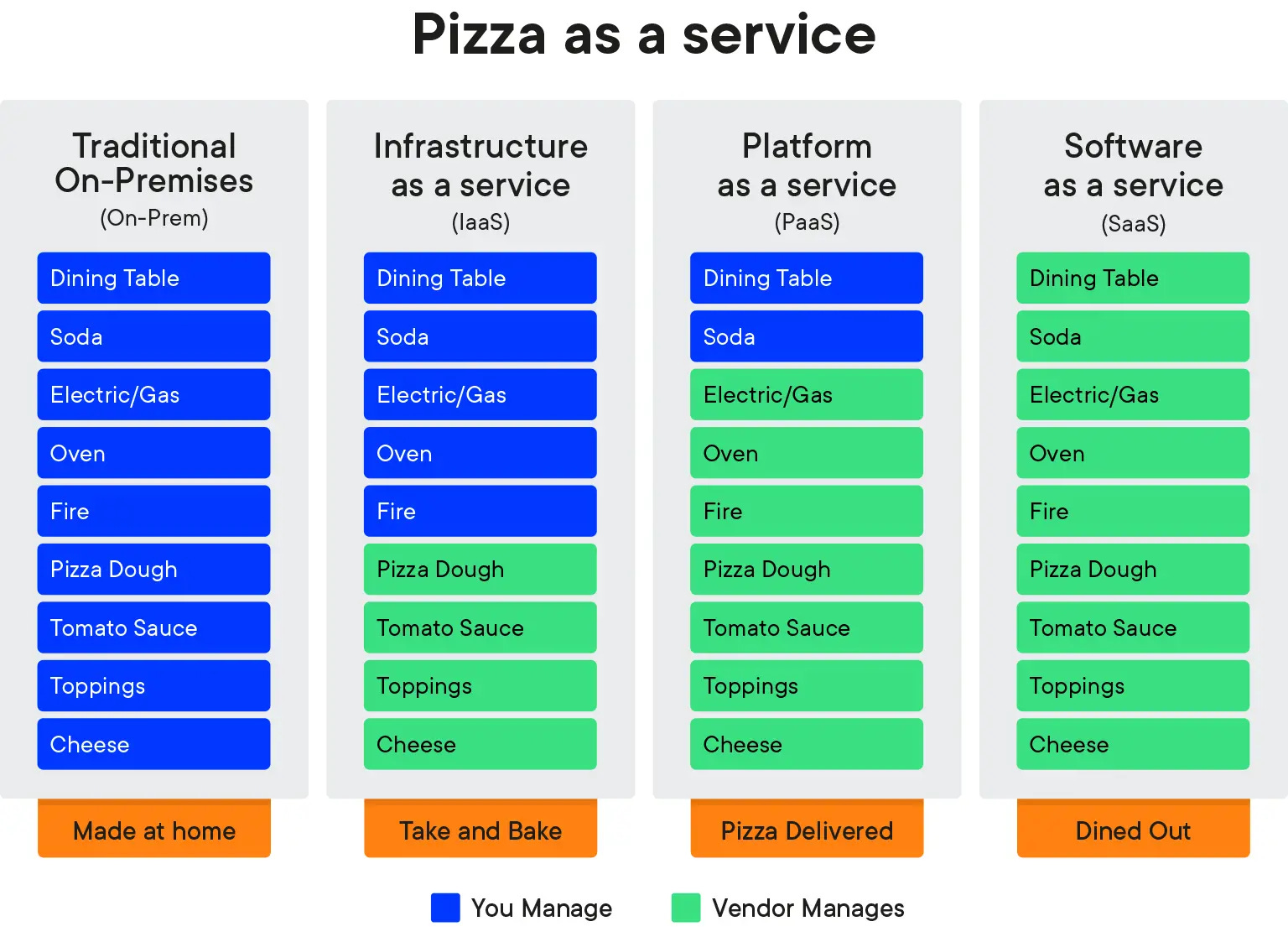
<https://medium.com/@fundingblogger/iaas-paas-saas-what-is-it-explained-with-examples-2f4ee34e7685>

<https://avs431.medium.com/explain-it-to-me-like-i-am-a-5-year-old-cloud-delivery-models-iaas-paas-and-saas-with-use-77499c233fd3>

<https://m.oursky.com/saas-paas-and-iaas-explained-in-one-graphic-d56c3e6f4606>



**Pizza as a Service Analogy**



### Service Scope

**SaaS (Software as a Service):** Provides software applications from the cloud, and users access these applications. The software’s usage and maintenance are the responsibility of the provider.

**PaaS (Platform as a Service):** Offers a platform and tools for application development, allowing developers to write and deploy application code.

**IaaS (Infrastructure as a Service):** Supplies infrastructure resources like virtual machines, storage, and networking, with users managing the operating system and applications directly.

### Control Level

**SaaS:** Provides the lowest control level, with limited permissions for software configuration and management.

**PaaS:** Offers intermediate control, allowing management of application code and database settings, while infrastructure management remains the provider’s responsibility.

**IaaS:** Grants the highest control level, enabling users to control all aspects, from virtual servers and network configurations to operating systems and application management.

### Use Cases

**SaaS:** Typically suitable for using software applications like office tools, email, and collaboration software, eliminating the need to purchase software licenses.

**PaaS:** Useful for setting up development environments, application development, testing, and web application hosting.

**IaaS:** Used for meeting infrastructure requirements such as server virtualization, storage management, backup, and recovery.

## Public vs Private vs Hybrid

<https://medium.com/@mintholic1/public-private-and-hybrid-cloud-e87da3eb763f>

## Virtualization

<https://www.ibm.com/topics/virtualization>

<https://aws.amazon.com/what-is/virtualization/>

<https://aws.amazon.com/compare/the-difference-between-type-1-and-type-2-hypervisors/>

## AWS Lambda

Amazon Lambda is a serverless computing service, which comes under the Platform as a Service (PaaS) category. It abstracts the underlying infrastructure, allowing users to run and implement their codes.

## AWS CloudFront

Amazon S3 is designed for large-capacity, low-cost file storage in one specific geographical region.\* The storage and bandwidth costs are quite low.

Amazon CloudFront is a Content Delivery Network (CDN) which proxies and caches web data at edge locations as close to users as possible.

S3 buckets are regional. CloudFront is global. If you have a user across the globe, they will get the file from a nearby CloudFront edge location, and not from your region

CloudFront has more advanced URL signing too.

I believe you need CloudFront if you want a custom domain, with HTTPS.

You could replicate your S3 bucket to different regions, and use DNS to route different users to different buckets, as an alternative.

<https://www.reddit.com/r/aws/comments/pulahd/whats_the_benefits_of_using_cloudfront_with_s3/>

<https://stackoverflow.com/questions/3327425/when-to-use-amazon-cloudfront-or-s3>

## IIS

Internet Information Services, also known as IIS, is a Microsoft web server that runs on Windows operating system and is used to exchange static and dynamic web content with internet users. IIS can be used to host, deploy, and manage web applications using technologies such as ASP.NET and PHP.

## Firestore | Firebase - Google

Cloud Firestore is a massively scalable, cloud-hosted, NoSQL, realtime database. In order to structure your data, you define collections (similar to tables in SQL) which contain documents (similar to rows). Each document contains fields that contain the actual data. You can reference an individual document using its unique path, or you can query a collection for documents whose fields contain the data you’re looking for.

<https://firebase.google.com/docs/firestore>

## ECS Fargate

<https://aws.amazon.com/fargate/>