

SOFTWARE ENGINEERING

UNIT – 5

CHAPTER – 1

RISK MANAGEMENT

I. INTRODUCTION

Risk management in software engineering is the process of identifying, analyzing, and mitigating potential problems or uncertainties that could affect the successful development and deployment of a software project. In simple terms, it's about being aware of the possible challenges and problems that may arise during the software development process and taking steps to deal with them effectively.

The key steps in risk management:

1. **Identify Risks:** Recognize potential issues that may impact the project. This could include technical challenges, changes in requirements, resource constraints, or external factors.
2. **Assess Risks:** Evaluate the likelihood and potential impact of each identified risk. Determine which risks are most critical and require immediate attention.
3. **Plan for Risk Mitigation:** Develop strategies to minimize the impact of identified risks. This could involve changing project plans, allocating additional resources, or creating contingency plans.
4. **Monitor and Control Risks:** Keep an eye on the project throughout its lifecycle. If new risks emerge or existing ones evolve, adjust the risk management plan accordingly.
5. **Communicate:** Maintain open communication within the project team and stakeholders about potential risks, their status, and the strategies in place to address them.

By actively managing risks, software development teams can increase the likelihood of completing projects successfully, on time, and within budget. It's a proactive approach to handling uncertainties and challenges that may arise during the complex process of building software.

II. REACTIVE VS PROACTIVE RISK STRATEGIES:

Reactive and proactive risk strategies refer to two different approaches in dealing with potential issues or uncertainties that may arise during a project.

Reactive Risk Strategy: This approach involves addressing problems as they arise. When an issue or risk becomes apparent during the software development process, the team responds to it at that moment. Reactive is like firefighting: You react to a fire (problem) when it occurs, trying to minimize damage after it has started.

Example: Imagine a situation where a key team member unexpectedly leaves the project. In a reactive strategy, the team would respond by quickly finding a replacement or redistributing tasks to cover the gap.

Proactive Risk Strategy: This approach involves identifying and addressing potential issues before they become actual problems. The team anticipates possible risks and develops strategies to prevent or minimize their impact. Proactive is like fire prevention: You take steps beforehand to reduce the chances of a fire (problem) happening, or you have a plan in place to quickly control it if it does.

Example: Consider a project where there's a risk of delays due to a potential lack of availability of a specific technology. In a proactive strategy, the team might plan ahead by securing alternative technologies or developing a contingency plan to avoid delays.

In software engineering, a combination of both strategies is often used. While you can't predict every possible issue, being proactive in identifying and planning for potential risks can significantly reduce the likelihood and impact of problems during the development process.

III. SOFTWARE RISKS:

Software risks in software engineering are like potential problems or uncertainties that might pop up during the process of creating computer programs or applications. Just like how you might encounter obstacles or unexpected challenges when building something, software projects can face issues that may affect their success.

Understanding and managing these risks is important in software engineering to increase the chances of a successful and smooth development process. It involves identifying possible problems early on, figuring out how likely they are to happen, and coming up with plans to either prevent them or deal with them effectively if they do occur.

These risks can come in various forms:

1. **Project risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.
2. **Technical risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.
3. **Business risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.
4. **Known risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.
5. **Predictable risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.
6. **Unpredictable risk:** This is a broad term for any potential problem that could affect the success of the entire software project.
Example: Delays in project timeline, budget overruns, or unexpected changes in project requirements.

Risk Characteristics:

In software engineering, risk characteristics refer to different aspects or qualities of potential problems or uncertainties that might affect a software project.

The key risk characteristics are:

1. **Probability:** How likely is it that a particular problem or issue will happen?
Example: If there's a risk of a computer crashing during a software test, the probability would be how often that crash is expected to occur.
2. **Impact:** If a risk happens, how much will it affect the project?
Example: If a crucial team member leaves, the impact could be high because it might slow down the project.
3. **Time Sensitivity:** Does the risk need to be addressed quickly, or can it wait?
Example: If a critical software component has a high risk of failure, addressing it promptly is time-sensitive to avoid delays.
4. **Dependencies:** Are there other factors or events that are connected to this risk?
Example: If a software component relies on a third-party service, the risk involves not only the component but also the reliability of that service.
5. **Mitigation Difficulty:** How hard is it to lessen the impact or likelihood of the risk?
Example: If a risk involves a rare software bug, it might be challenging to mitigate because fixing rare bugs is often more difficult.
6. **Uncertainty:** How well can we predict or understand this risk?
Example: If a new technology is being used in the project, there might be more uncertainty about potential technical risks compared to using a well-established technology.

Understanding these characteristics helps the software development team make informed decisions about which risks to prioritize and how to manage them effectively during the project.

IV. RISK IDENTIFICATION:

Risk identification in software engineering is like making a list of possible problems or uncertainties that might show up during the creation of a computer program. It's about recognizing and understanding things that could go wrong.

Imagine you're planning a big party. Before the day arrives, you might think about what could possibly go wrong: bad weather, guests running late, or not having enough snacks. Similarly, in software engineering, teams think about potential issues before they happen.

There are two distinct types of risks:

In the context of risk management, generic risks and product-specific risks refer to different categories of potential issues that a project may encounter.

1. **Generic Risks:** These are risks that are common to many types of projects, irrespective of the specific nature of the product being developed. Generic risks are often associated with the overall project management and development processes.

Examples:

- a. Lack of communication among team members.
- b. Budget overruns.
- c. Delays in project schedule.
- d. Changes in project requirements.

2. **Product-Specific Risks:** These are risks that are unique to the characteristics and requirements of the specific product or software being developed. They are more closely tied to the nature of the project and the features of the product itself.

Examples:

- a. Integration issues with specific hardware.
- b. Difficulty in implementing unique and innovative features.
- c. Compatibility problems with certain software environments.
- d. Unanticipated user interface challenges.

Generic Risks are like common challenges that many projects might face, similar to how most construction projects might face weather-related delays. Product-Specific Risks are more like challenges that are specific to the unique aspects of what you're building, like a custom-designed

house having unique construction requirements. Both types of risks need to be considered and managed during the course of a project. While generic risks are generally applicable to various projects, product-specific risks require a more tailored approach based on the unique characteristics of the software or product being developed.

Assessing Overall Project Risks:

The process of assessing overall project risks in software engineering:

1. **Make a List of Potential Problems:** Think about things that could go wrong during the project, like technical challenges, changes in what the software needs to do, or running out of resources.
2. **Group Similar Problems Together:** Put similar problems into categories, like technical issues, business concerns, or operational challenges. This makes it easier to deal with them.
3. **Figure Out How Likely and How Bad Each Problem Could Be:** For each problem, decide how likely it is to happen and how much it could mess up the project if it does. This helps prioritize which problems to focus on.
4. **Focus on the Big Problems First:** Look at the list and figure out which problems are the most important to tackle. These are the ones that could cause the most trouble or delays.
5. **Come Up with Plans to Deal with the Problems:** For the big problems, think of ways to either stop them from happening or reduce their impact. This could involve changing plans, getting more help, or having backup strategies.
6. **Think About How Problems Might Affect Each Other:** Consider how one problem might lead to another. Addressing one problem might help or hurt another, so it's important to look at the connections between problems.
7. **Talk to Everyone Involved:** Get input from the people working on the project and anyone else who has a stake in its success. Different perspectives can help identify risks you might have missed.
8. **Write Down and Share Your Plans:** Keep a record of the problems you've identified, how likely they are, and what you plan to do about them. Share this information with the team and others involved in the project.

9. **Keep an Eye Out for Changes:** As the project goes on, watch for new problems or changes in the importance of existing ones. Adjust your plans as needed to handle the evolving situation.

By going through these steps, the project team can be better prepared to handle challenges, make smart decisions, and increase the chances of the project being successful. It's like making a game plan for a big journey, thinking about the possible bumps in the road and having ways to smooth them out.

Risk Components and Drivers:

Risk components and drivers in software engineering:

1. **Risk Components:** Imagine you're building a house with different parts like walls, roof, and foundation. In software, these parts are like different aspects of the project, such as the complexity of the technology, available resources, and project requirements. Each of these aspects can be a source of potential challenges or risks.
2. **Risk Drivers:** Now, think of external factors that can influence how difficult or likely these challenges are. These factors, like changes in market trends, new laws, or how well users accept the software, are the drivers. They push or affect the challenges, making some more important or likely to happen than others.

Risk Components are the pieces inside the project that might cause problems. Risk Drivers are the outside forces that can make some problems more likely or impactful than others. By understanding both the inside parts (components) and the outside forces (drivers), the project team can better prepare for and deal with potential challenges during the software development journey. It's like knowing the different parts of a game and understanding how external factors can make some aspects of the game more interesting or challenging.

V. RISK PROJECTION:

Risk projection, also known as risk estimation in software engineering, is like making an educated guess about how likely and how severe potential problems might be during a project.

1. **Likelihood:** Think about how probable it is for a certain issue to happen. Is it something that might occur often, rarely, or somewhere in between?
Example: If there's a risk of a computer bug, estimating the likelihood is like guessing how often that bug might show up.
2. **Impact:** Consider how much trouble or difficulty a problem might cause if it actually happens. Is it a small hiccup, a big setback, or something in between?
Example: If there's a risk of running out of time, estimating the impact is like figuring out how much of a delay it might cause.
3. **Combined Assessment:** Put together your guesses about likelihood and impact. This gives you an overall assessment of the risk—how likely it is to happen and how much it might affect the project.
Example: If there's a risk of a team member getting sick, estimating the risk involves thinking about how likely it is for someone to get sick and how much it would disrupt the project if it happens.

The goal of risk projection is to make informed predictions about potential issues so that the project team can plan and prepare accordingly. It's like looking at a map before a journey and trying to predict which parts might have more traffic or roadblocks.

Risk Table:

A risk table in software engineering is like a handy chart that helps the project team keep track of potential problems and plan how to deal with them.

How it works:

1. **List of Risks:** Imagine you have a list of things that might go wrong during your software project. These could be technical issues, changes in plans, or unexpected events.

2. **Likelihood and Impact:** For each risk, think about how likely it is to happen (high, medium, low) and how much trouble it might cause if it does (high, medium, low).
3. **Putting it in the Table:** Create a table with rows for each risk and columns for likelihood and impact. Then, fill in the cells with your estimates (like using "H" for high, "M" for medium, and "L" for low).
4. **Color Codes or Scores:** You might also use colors or scores to make it even simpler. For example, use green for low risk, yellow for medium, and red for high.
5. **Prioritizing and Planning:** The table helps you see which risks are the most serious (maybe they are both likely to happen and have a big impact). These are the ones you want to focus on first and come up with plans to handle.

In the end, it's like having a visual guide that quickly shows which parts of your project need extra attention and preparation. It's a tool to help you be ready for potential bumps in the road during your software development journey.

A risk table in software engineering typically looks like a grid or a table, where each row represents a specific risk, and each column provides information about that risk.

Example of what a risk table might look like:

Imagine you're making a list of things that might cause trouble during your software project. You want to be prepared, so you create a table like this:

Risk ID	Risk Description	Likelihood	Impact	Priority
R1	Technical bug in critical module	High	High	High
R2	Change in project requirements	Medium	High	Medium
R3	Key team member leaves	Low	High	Medium
R4	Delays in third-party software	High	Medium	Medium
R5	Server downtime during launch	Medium	High	High

- a. Risk ID: A special number for each problem.
- b. Risk Description: A short explanation of what might go wrong.
- c. Likelihood: How likely it is for the problem to happen (Low, Medium, High).
- d. Impact: How much trouble it could cause (Low, Medium, High).

- e. Priority: A mix of how likely and how much trouble, showing which problems are most important to deal with first (Low, Medium, High).

This table helps you see and understand the potential issues in a simple way, so you can focus on the most important ones and plan ahead to handle them. It's like making a to-do list for possible challenges during your software project.

VI. RISK REFINEMENT:

Risk refinement in software engineering is like taking a closer look at the possible problems we found when we started our project. It's about going deeper, thinking harder, and making really detailed plans to handle any challenges that might come up. Imagine it as being extra prepared by understanding everything about the risks and having clear steps on how to deal with them from the beginning to the end of our project journey. It's like being a super detective for our software adventure, making sure we're ready for anything that could happen.

Finding More Details: Imagine you have a list of potential problems for your software project. Now, you want to look closely at each problem, almost like using a magnifying glass to see more details.

Understanding How Likely and How Bad: For each problem, you're not just guessing anymore. You're really thinking hard about how likely it is to happen and how much trouble it might cause if it does.

Breaking Big Problems into Smaller Pieces: Some problems are big and complicated. During risk refinement, it's like breaking those big problems into smaller, easier-to-handle parts. It's like dealing with one piece at a time.

Making Step-by-Step Plans: Instead of just saying, "Oh, that might be a problem," you're making detailed plans. It's like having a step-by-step guide for what to do if a particular problem shows up.

Keeping Everything Up to Date: As you learn more and make plans, you're updating your information. It's like keeping a diary that has the latest details about each problem and what you're doing to prevent or fix them.

Talking and Working Together: This isn't something you do alone. You talk to your team, share what you've found, and work together to make sure everyone is ready for the possible problems.

So, risk refinement is like being a detective in your project, examining each potential problem closely, making detailed plans, and staying updated. It's all about being super prepared for anything that might happen while building your software.

VII. RISK MITIGATION, MONITORING AND MANAGEMENT [RMMM]:

"Risk Mitigation, Monitoring, and Management" (RMMM) in software engineering:

1. **Risk Mitigation:** This is like taking steps to prevent or reduce the impact of potential problems before they happen.
Example: If there's a risk of running out of time, mitigation could involve planning tasks carefully, setting realistic deadlines, and having extra time for unexpected issues.
2. **Risk Monitoring:** It's about keeping a watchful eye on the project to see if any of the expected problems are starting to show up.
Example: If you identified a risk of technology glitches, monitoring would involve regularly checking the software for any signs of technical issues.
3. **Risk Management:** This is the overall process of identifying, assessing, and dealing with potential problems throughout the project.
Example: It's like having a plan for everything - knowing what could go wrong, figuring out how bad it could be, and deciding what to do about it.

Mitigation is like putting on a superhero cape before any problems happen, trying to stop them in their tracks.

Monitoring is like keeping your superhero senses on high alert during the project, ready to jump in if any trouble starts brewing.

Management is like being the wise leader who has a game plan for the whole journey, making sure the team is ready to face whatever challenges come their way.

RMMM in software engineering is like being the superhero of your project. It's about gearing up in advance to stop problems, staying vigilant for any signs of trouble, and having a well-thought-out plan to lead your team through the exciting journey of creating software. It's superhero-level preparation to ensure a smooth and successful adventure in software development.

VIII. RMMM – PLAN:

The RMMM plan, or Risk Mitigation, Monitoring, and Management plan, is a strategic document in software engineering that outlines how a project team will identify, assess, and address potential risks throughout the development process.

Its components are:

1. **Risk Identification:** Recognizing and listing potential problems that could affect the project.
Example: Identifying the risk of software bugs, changes in requirements, or resource constraints.
2. **Risk Analysis and Assessment:** Evaluating the likelihood and impact of each identified risk.
Example: Estimating how likely it is for a bug to occur and how much it could delay the project.
3. **Risk Mitigation:** Planning and implementing actions to either prevent risks from happening or reduce their impact.
Example: Creating a strategy to test the software thoroughly to minimize the chances of bugs.
4. **Risk Monitoring:** Regularly observing and checking for signs of potential risks as the project progresses.
Example: Continuously reviewing the project timeline to ensure it stays on track.
5. **Contingency Planning:** Developing backup plans for dealing with risks if they materialize.
Example: Having a contingency plan in case a key team member unexpectedly leaves the project.

6. **Documentation:** Recording details about identified risks, their assessments, and the strategies in place to manage them.

Example: Keeping a log that describes each risk, its potential impact, and the steps taken to address it.

7. **Communication and Reporting:** Regularly updating and informing relevant stakeholders about the status of identified risks and the actions being taken.

Example: Providing status reports to team members and project sponsors about the progress in mitigating risks.

8. **Review and Adaptation:** Periodically revisiting and adjusting the RMMM plan based on changes in the project or the identification of new risks.

Example: Updating the plan when there are changes in project requirements or technology.

In simple terms, an RMMM plan is like a roadmap that guides the project team in dealing with potential challenges and uncertainties. It helps them be proactive in preventing issues, keeps them vigilant in monitoring the project's progress, and provides a structured approach to manage risks effectively throughout the software development journey.

Risk information sheet

A risk information sheet in a Risk Mitigation, Monitoring, and Management (RMMM) plan is like a cheat sheet for each potential problem (risk) in your software project. It's a quick reference guide that gives you key details about a risk so you can deal with it effectively.

Example:

Risk Information Sheet contains:

1. **Name of the Risk:** This is like giving a name to the potential problem.
Example: "Bug Surge Risk"
2. **Description of the Risk:** Briefly explain what the potential problem is about.
Example: "The risk of a sudden increase in software bugs affecting system stability."
3. **Likelihood of the Risk:** How likely is it that this problem will happen?

Example: "High likelihood - there's a good chance we might face more bugs due to the complexity of the code."

4. **Impact of the Risk:** If the problem happens, how much trouble will it cause?

Example: "Medium impact - it could slow down development, but it won't completely halt the project."

5. **Mitigation Plan:** What actions are you taking to prevent or lessen the impact of the problem?

Example: "Regular code reviews and testing to catch bugs early; team training on best practices to reduce the likelihood of coding errors."

6. **Monitoring Approach:** How are you keeping an eye on this problem as the project goes on?

Example: "Weekly code inspections, automated testing, and tracking bug reports to catch any surges in issues."

7. **Contingency Plan:** What's your backup plan if the problem occurs despite your efforts?

Example: "Have a dedicated team ready to address urgent bug fixes and allocate extra time in the schedule for debugging."

By having a risk information sheet for each potential problem, you're like a superhero with a set of quick reference cards. When a problem arises, you can quickly grab the right card and know exactly what to do. It's a simple but powerful tool to keep your project sailing smoothly.

Risk Name	Bug Surge Risk
Description	The risk of a sudden increase in software bugs affecting system stability.
Likelihood	High
Impact	Medium
Mitigation Plan	Regular code reviews and testing; team training on best practices.
Monitoring	Weekly code inspections, automated testing, and tracking bug reports.
Contingency Plan	Dedicated team for urgent bug fixes; extra time in the schedule for debugging.

In this example:

- a. Risk Name: A short name for the potential problem.
- b. Description: A brief explanation of what the risk is about.
- c. Likelihood: How likely it is for the problem to happen (High, Medium, Low).
- d. Impact: How much trouble it could cause if it happens (High, Medium, Low).
- e. Mitigation Plan: Actions taken to prevent or lessen the impact of the problem.
- f. Monitoring: How the team keeps an eye on the risk during the project.
- g. Contingency Plan: What to do if the problem occurs despite mitigation efforts.

This simple table helps the project team have a quick and clear reference for each identified risk, making it easier to manage and address potential challenges during the software development journey.

SOFTWARE ENGINEERING

UNIT – 5

CHAPTER – 2

INTRODUCTION TO AWS CLOUD AND ITS SERVICES

I. INTRODUCTION TO CLOUD COMPUTING

Cloud computing is a technology paradigm that involves the delivery of computing services, including storage, processing power, and software applications, over the internet. Instead of relying on local servers or personal computers to handle applications, users can access and utilize resources hosted on remote servers. Cloud computing enables on-demand access to a shared pool of configurable computing resources, allowing users to scale their usage up or down based on their requirements. This model offers flexibility, efficiency, and cost-effectiveness, as users pay for the resources, they consume rather than investing in and maintaining their own infrastructure. Key characteristics of cloud computing include on-demand self-service, broad network access, resource pooling, rapid elasticity, and measured service.

Cloud computing is like having a virtual space on the internet where you can store and access your data and run applications, making it more convenient and flexible than keeping everything on a single physical computer

Imagine you have a computer, and you store all your pictures, documents, and software on it. Now, what if you could access all of that stuff not just from your computer, but from any computer or device, anywhere in the world? That's a bit like what cloud computing does.

Instead of keeping everything on your own computer, cloud computing lets you use the internet to store and access data, and run applications. It's like renting space on the internet to store your files and run programs. This means you can access your information and use software without being tied to a specific device. The "cloud" refers to the internet, and the services you use are hosted on servers (powerful computers) in data centers around the world.

Features of cloud computing

1. **Availability:** Availability in cloud computing refers to the idea that your data and applications are accessible whenever you need them.

Example: Think of it like electricity. You don't have to worry about generating electricity at home; you simply plug in and use it. Similarly, with cloud computing, your data and applications are "always on," like a reliable utility service.

2. **Scalability:** Scalability is about the ability to easily adjust the amount of resources (like storage or processing power) you're using based on your needs.

Example: Imagine you're hosting a website. If suddenly many people start visiting, you want your website to handle the increased traffic smoothly. Scalability in the cloud is like being able to easily add more tables to a restaurant when it gets busy and removing them when it's quieter.

3. **Pay-as-you-go:** Pay-as-you-go means that you only pay for the computing resources you actually use, just like you pay for utilities like water or electricity based on your consumption.

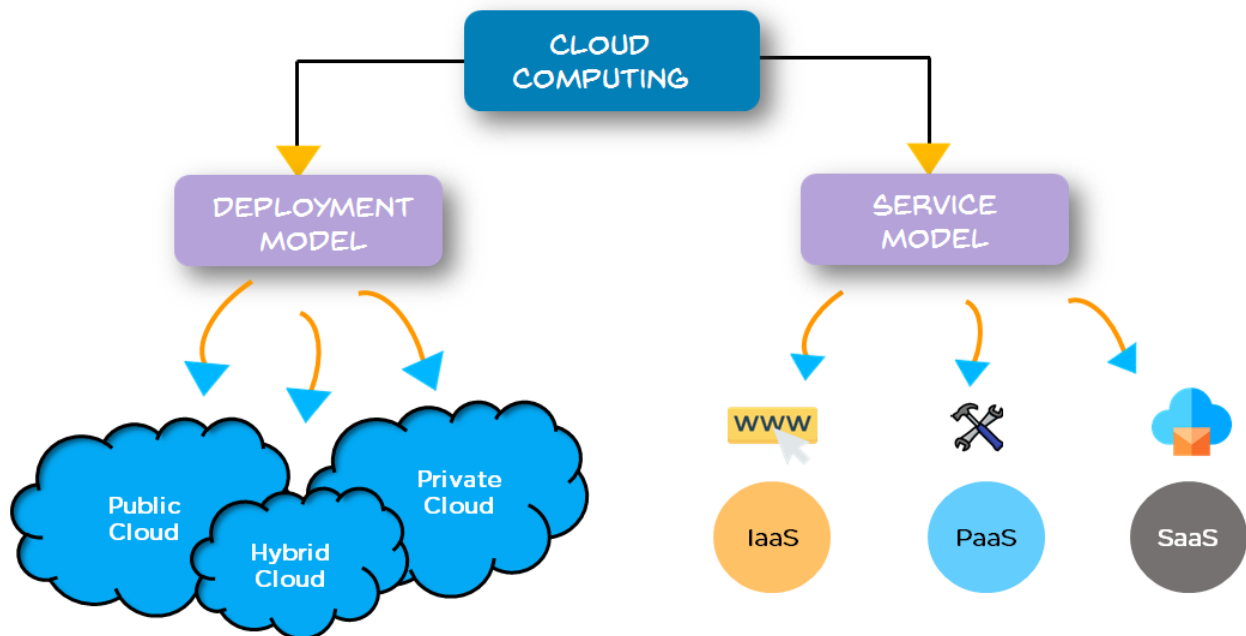
Example: Suppose you're renting a car. With pay-as-you-go, you're charged for the miles you drive and the time you have the car. In cloud computing, you're billed based on how much storage, processing power, or other resources you use. If you use more, you pay more; if you use less, you pay less.

Availability ensures your data is always accessible, scalability allows you to easily adjust resources to meet demand, and pay-as-you-go ensures you only pay for what you use, making cloud computing flexible, efficient, and cost-effective.

Types of cloud Computing

Cloud computing is of two types:

1. Cloud deployments
2. Cloud services



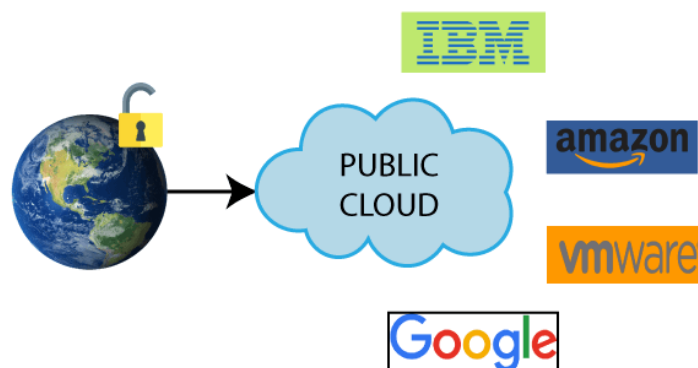
CLOUD DEPLOYMENTS

Cloud deployment refers to how and where you choose to use and manage your computer-related stuff, like storing data or running programs, using the internet.

The various cloud deployments are:

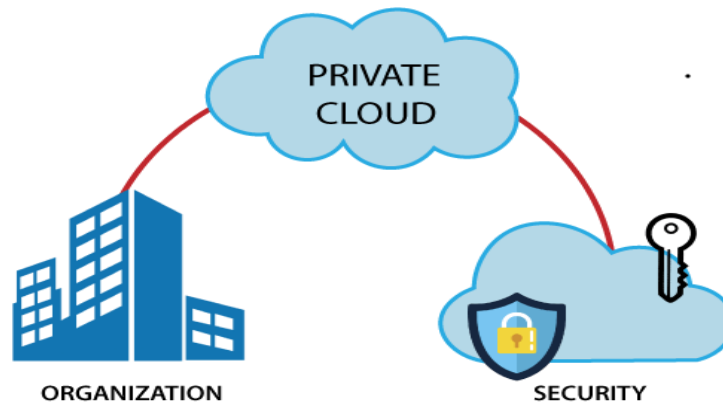
1. **Public Cloud:** Public clouds are like giant data centers owned by companies that rent out computing resources (like servers, storage, and networks) to anyone over the internet.

Example: Imagine a massive library where anyone can borrow books. In a public cloud, companies and individuals "borrow" computing resources from a shared pool offered by a cloud provider like Amazon, Microsoft, or Google.



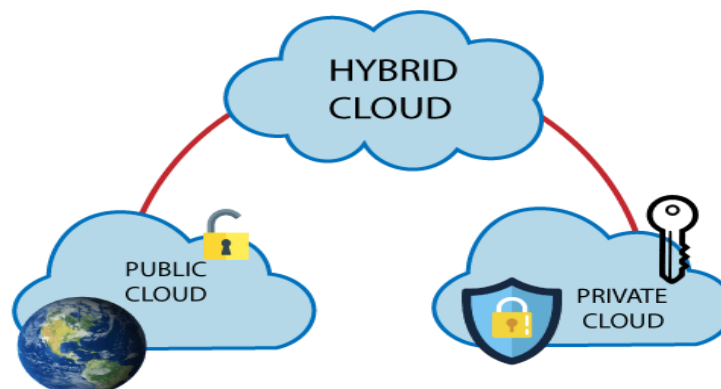
2. **Private Cloud:** Private clouds are like exclusive clubs where the computing resources are used only by one organization. It can be managed by the organization itself or a third-party provider.

Example: Think of a private library owned by a school. Only students and staff have access to the books. Similarly, in a private cloud, the computing resources are dedicated to a single organization.



3. **Hybrid Cloud:** Hybrid clouds combine elements of both public and private clouds. Organizations use a mix of on-premises infrastructure, private cloud, and public cloud services, and these environments can work together.

Example: Consider a bookstore that sells books online and in a physical store. The online part uses a public cloud for scalability, while the in-store inventory system is managed privately. The two systems work together, forming a hybrid setup.



II. CLOUD SERVICES

Cloud services are like digital tools and resources that you can use over the internet. It's a bit like borrowing things you need for your computer work without having to actually own or store them on your own device.

Cloud computing services are divided into three classes:

- (a) Infrastructure as a Service,
- (b) Platform as a Service, and
- (c) Software as a Service.

(a) **Infrastructure as a Service (IaaS):** Think of IaaS as renting the basic building blocks of computing over the internet. You get the fundamental components like virtual machines, storage space, and networking resources.

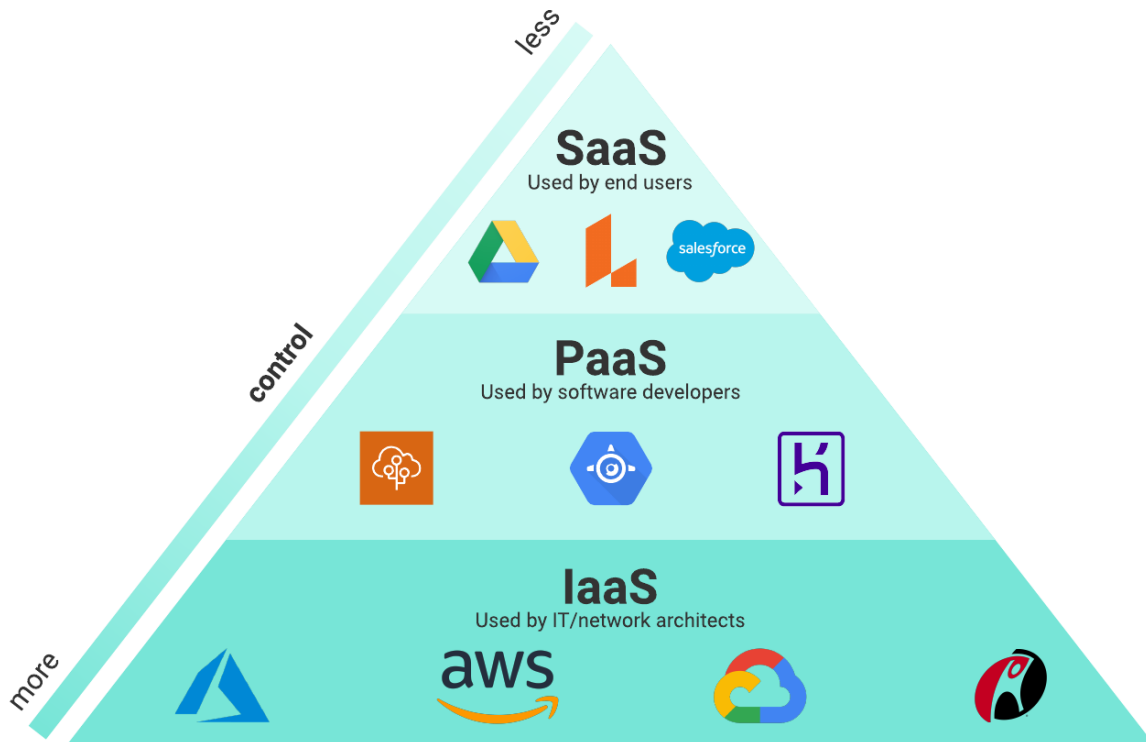
Example: Imagine you need a computer for a short period. Instead of buying a new one, you rent a virtual computer online with all the necessary resources to run your programs and store your data.

(b) **Platform as a Service (PaaS):** PaaS is like renting not just the infrastructure but also a ready-made platform for building and running applications. It provides tools and services that make it easier for developers to create and manage their software.

Example: It's like having a fully equipped kitchen (the platform) where you can cook without worrying about buying pots, pans, and utensils (the infrastructure). Developers can focus on building their applications without dealing with the underlying complexities.

(c) **Software as a Service (SaaS):** SaaS is like using software directly from the internet without installing anything on your computer. You access fully functional applications without having to manage the software, servers, or infrastructure.

Example: Instead of buying and installing a word processing software on your computer, you use an online word processor. You log in, type your document, and save it—all without worrying about software updates or storage.



IaaS: Renting basic computing resources like virtual machines and storage.

PaaS: Renting a platform with tools to make it easier for developers to build and manage applications.

SaaS: Using fully functional software directly from the internet without installing anything.

Each class represents a level of abstraction, offering users different levels of control and responsibility over their computing environment. It's like choosing how much you want to do yourself and how much you want the cloud service provider to handle for you.

III. UNDERSTAND AND CREATE CLOUD INFRASTRUCTURE USING AWS

Cloud infrastructure in AWS is like renting and using computer resources, storage, and services over the internet instead of having physical machines in your own office or data center. Imagine you need a computer to run a website, store files, or process data. Instead of buying and maintaining a physical computer, you can go to AWS, where they have a bunch of powerful virtual

computers ready for you. You pick the size and type you need, and you only pay for the time you use it.

Similarly, if you want to store a lot of pictures, videos, or any data, you don't need to buy and manage your own hard drives. AWS provides a massive online storage space where you can store as much as you want.

AWS also offers various tools for networking, security, databases, and more. It's like having a toolkit that lets you build and run your digital projects without worrying about the physical stuff—it's all in the cloud. You access and manage these resources through the internet, and AWS takes care of the behind-the-scenes work, like keeping the servers running and secure.

AWS services

Some key AWS services:

1. **Amazon EC2 (Elastic Compute Cloud):** Rent a virtual computer in the cloud. It's like booking a computer online when you need it, and you can choose how powerful it is.
2. **Amazon S3 (Simple Storage Service):** Store your files on the internet. It's like having a huge online storage room where you can keep all your pictures, documents, or videos.
3. **Amazon RDS (Relational Database Service):** Get a managed database in the cloud. It's like having a smart assistant organize your data in a way you can easily find and use it.
4. **Amazon VPC (Virtual Private Cloud):** Create your private section of the internet. It's like having your own fenced-off area in the digital world where you control who gets in.
5. **Amazon CloudWatch:** Keep an eye on your stuff in the cloud. It's like having a watchman who tells you if anything unusual is happening with your virtual computers or services.
6. **AWS Lambda:** Run your code without managing servers. It's like having a magic spell—your code runs whenever needed, and you don't have to worry about where it's happening.
7. **Amazon CloudFront:** Make your website or app load faster everywhere. It's like having delivery trucks strategically placed worldwide, so your digital content reaches users quickly.
8. **AWS IAM (Identity and Access Management):** Control who can do what in your cloud. It's like giving specific people or programs keys to different rooms in your digital house—you decide who can go where.

9. **AWS CloudFormation:** Create your digital world with a blueprint. It's like having a recipe to build everything you need in the cloud, and you can replicate it whenever you want.
10. **Amazon Sage Maker:** Teach your computer to learn and make decisions. It's like having a teacher for your computer, so it can get better at recognizing patterns or making predictions.

These services are like building blocks that you can mix and match to create your own digital playground in the cloud. You use what you need, and AWS takes care of the technical details behind the scenes.

The steps to create cloud infrastructure on AWS:

Creating cloud infrastructure on AWS involves several steps:

Step 1: Get Started

Sign Up: Go to AWS website and sign up for an account. Fill in your details and set up payment info (but don't worry, they have a free tier for beginners).

Step 2: Log In

Access Console: Log in to AWS using your newly created account.

Step 3: Explore Services

Check Services: Look around in the AWS Console. Focus on services like EC2, S3, and VPC for now.

Step 4: Launch a Virtual Server

Start an EC2 Instance: Go to EC2 Dashboard, click "Launch Instance." Follow the steps to choose a virtual server, set it up, and launch it.

Step 5: Set Up Storage

Create an S3 Bucket: Go to S3 Dashboard, click "Create Bucket." Choose a name, region, and create your virtual storage space.

Step 6: Configure Networking

Make a VPC: Go to VPC Dashboard, click "Create VPC." Set up your private section of the internet.

Step 7: Monitor Resources

Use CloudWatch: In the CloudWatch Dashboard, set up monitoring for your virtual stuff.

Step 8: Security and Access

Create IAM User: In the IAM Dashboard, click "Users," then "Add user." Make a user, give them permissions, and get security credentials.

Step 9: Deploy Your App (Optional)

Try Elastic Beanstalk: In the Elastic Beanstalk Dashboard, click "Create Application." Deploy your app without worrying about the technical details.

Step 10: Explore More (Optional)

Play Around: Explore other AWS services based on your needs. Check out RDS for databases, Lambda for serverless computing, etc.

Step 11: Check Your Bill

Manage Costs: Go to the Billing Dashboard. Keep an eye on your usage and set up alerts to manage costs.

Step 12: Learn and Have Fun

Experiment: Read AWS guides and experiment with different services. Enjoy building your digital world in the cloud!

IV. DEPLOYING THE WEB APPLICATION

Steps for Deploying a Static web application Site with Docker and Nginx

1. Create and Push web application into GitHub
2. Create the virtual machine and connect to it.
3. Clone the web application from GitHub, Write the Dockerfile

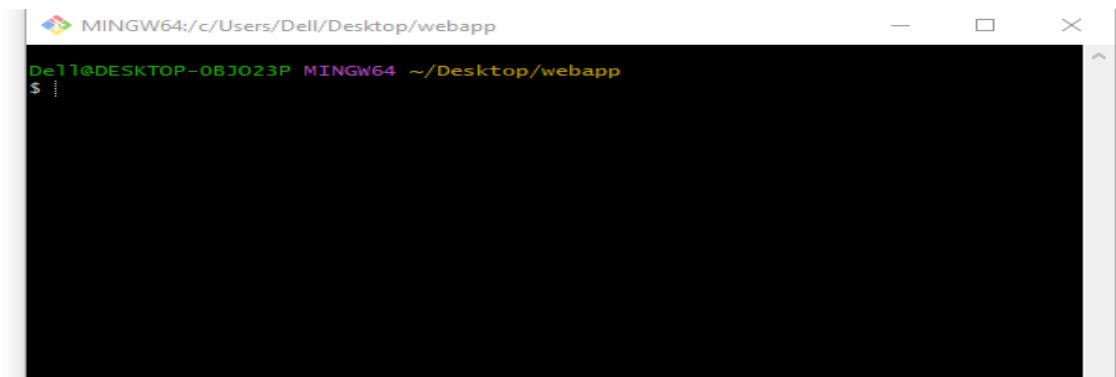
4. Create the image
5. Run the image and access it public ip of virtual machine

1. Create and push web application into GitHub

Make sure that you have your HTML files already in the current directory.

```
File Edit Format View Help
<html>
<head>
</head>
<body>
  <h1>Deploying web application</h1>
</body>
</html>
```

Now push the code to GitHub



Initialize the git repository, check the untracked files, add the files to staging area

```
Dell@DESKTOP-0BJ023P MINGW64 ~/Desktop/webapp
$ git init
Initialized empty Git repository in C:/Users/Dell/Desktop/webapp/.git/

Dell@DESKTOP-0BJ023P MINGW64 ~/Desktop/webapp (master)
$ git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        index.html

nothing added to commit but untracked files present (use "git add" to track)

Dell@DESKTOP-0BJ023P MINGW64 ~/Desktop/webapp (master)
$ git add .

Dell@DESKTOP-0BJ023P MINGW64 ~/Desktop/webapp (master)
```

Git commit is used to save the file in local repository

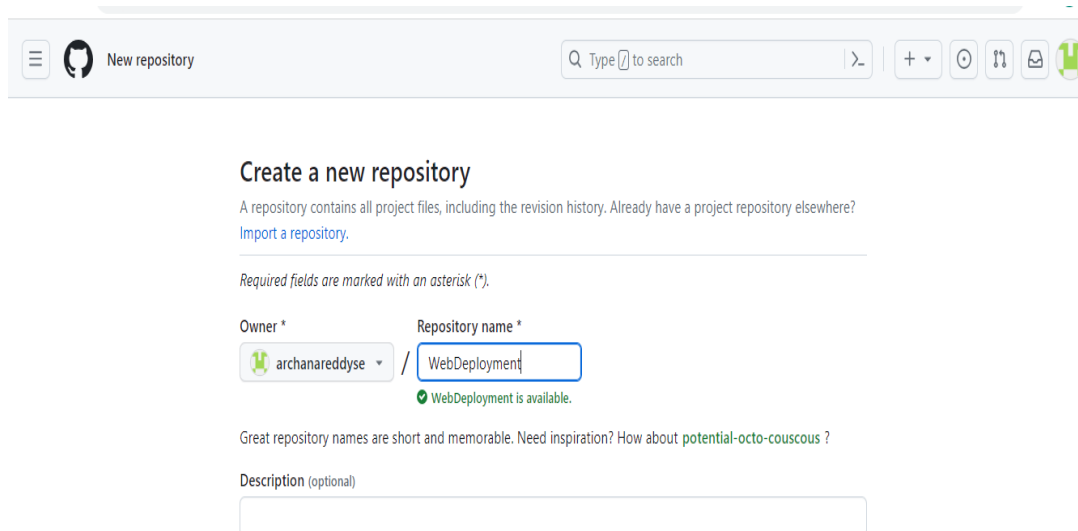
```
De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (main)
$ git commit -m "web deployment commit"
[main (root-commit) a55cdf5] web deployment commit
1 file changed, 7 insertions(+)
create mode 100644 index.html
```

Check whether git is connected to GitHub or not

```
MINGW64/c/Users/De1l/Desktop/webapp
De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (master)
$ git ls-remote
fatal: No remote configured to list refs from.

De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (master)
$ |
```

Create the repository in GitHub



Once created the repository, connect git bash to GitHub and check if connected.

```
De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (master)
$ git remote add origin https://github.com/archanareddyse/WebDeployment.git

De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (master)
$ |
```

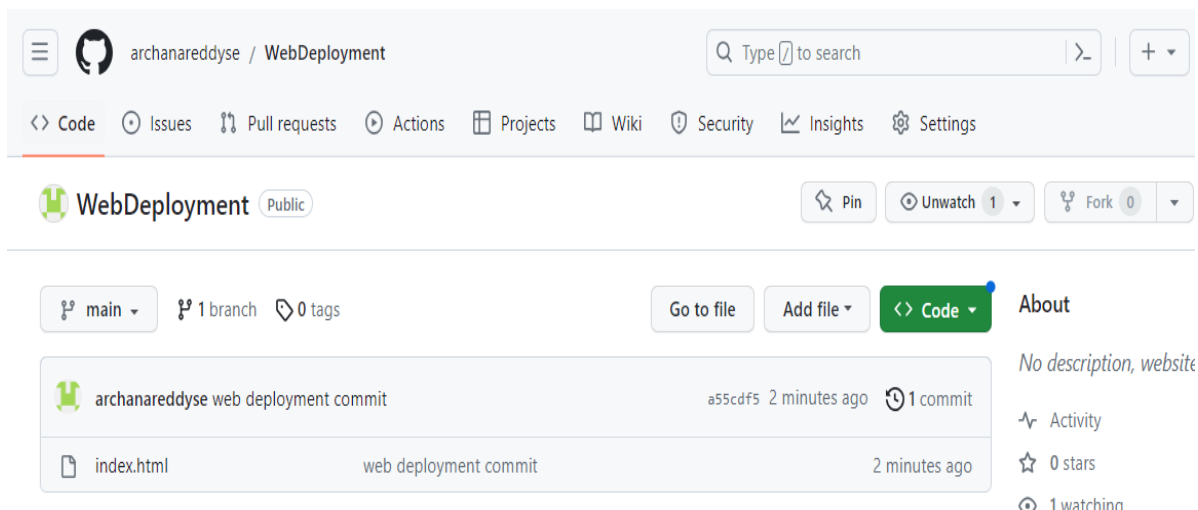
```
De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (main)
$ git ls-remote
From https://github.com/archanareddyse/WebDeployment.git
```

Now push files to GitHub

```
De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 285 bytes | 285.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/archanareddyse/WebDeployment.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.

De1l@DESKTOP-0BJO23P MINGW64 ~/Desktop/webapp (main)
$ |
```

Now we see the files are present in git hub



2. Steps to create the virtual machine and connecting to it.

Amazon Elastic Compute Cloud (Amazon EC2) provides scalable computing capacity in the Amazon Web Services (AWS) cloud. Using Amazon EC2 eliminates your need to invest in hardware up front, so you can develop and deploy applications faster.

Ex 1: Launch ubuntu instance

Step 1: Login to AWS

Step 2: Choose region which is near? (Asia pacific - Mumbai)

Step 3: Services -- EC2

(If any keypairs -- delete)

(If any security groups - delete, except default)

Services -- EC2 --- Launch Instance

Stage 1 --Name (Giving name to the machine) ubuntu

Stage 2 -- Select AMI (Note: Select free tier eligible) ubuntu server

Stage 3 -- Architecture as 64-bit

No of instances -- 1

Stage 4 -- t2. micro

Stage 5 -- Create a new keypair

Stage 6 -- Network Setting ----Create Security group -- (It deals with ports)

We have 0 to 65535 ports

Every port is dedicated to special purpose

Stage 7 -- Storage - 8GB (Observation - we have root - it is same as C Drive)

Stage 8 --- click on launch instance

Observation - One machines created.

We can use PowerShell /Gitbash /webconsole, to connect to ubuntu machine to connect to above terminals we need to go into the path of the keypair and paste the ssh -i command from the AWS console

```
PS C:\Users\Dell> cd downloads
PS C:\Users\Dell\downloads> ssh -i "webapplication.pem" ubuntu@ec2-13-232-50-228.ap-south-1.compute.amazonaws.com
Welcome to Ubuntu 22.04.2 LTS (GNU/Linux 5.19.0-1025-aws x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

System information as of Tue Sep 12 09:20:37 UTC 2023

System load:  0.02294921875   Processes:            99
Usage of /:   20.6% of 7.57GB   Users logged in:      0
Memory usage: 24%            IPv4 address for eth0: 172.31.2.3
Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.
```

Here we are connected with virtual machine of ubuntu.

3. Clone the code from GitHub, Write the Dockerfile

- install docker ---apt-get update
- apt-get install docker.io
- install nano -----apt-get update
- apt-get install nano

```
ubuntu@ip-172-31-2-3:~$ git clone https://github.com/archanareddyse/WebDeployment.git
Cloning into 'WebDeployment'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
ubuntu@ip-172-31-2-3:~$
```

4. Navigate to folder of web deployment and create Dockerfile

```
ubuntu@ip-172-31-2-3:~$ ls
WebDeployment
ubuntu@ip-172-31-2-3:~$ cd WebDeployment
ubuntu@ip-172-31-2-3:~/WebDeployment$ nano Dockerfile
ubuntu@ip-172-31-2-3:~/WebDeployment$
```

5. Dockerfile

```
ubuntu@ip-172-31-2-3:~/WebDeployment
GNU nano 6.2 Dockerfile *
FROM nginx:alpine
COPY . /usr/share/nginx/html
```

6. Create the image

Build the image by

docker build -t html- webapplication.

```
ubuntu@ip-172-31-2-3:~/WebDeployment$ sudo docker build -t html-webapplication .
DEPRECATED: The legacy builder is deprecated and will be removed in a future release.
            Install the buildx component to build images with BuildKit:
            https://docs.docker.com/go/buildx/

Sending build context to Docker daemon  59.39kB
Step 1/2 : FROM nginx:alpine
alpine: Pulling from library/nginx
7264a8db6415: Pull complete
518c62654cf0: Pull complete
d8c801465ddf: Pull complete
ac28ec6b1e86: Pull complete
eb8fb38efa48: Pull complete
e92e38a9a0eb: Pull complete
58663ac43ae7: Pull complete
2f545e207252: Pull complete
Digest: sha256:16164a43b5faec40adb521e98272edc528e74f31c1352719132b8f7e53418d70
Status: Downloaded newer image for nginx:alpine
--> 433dbc17191a
Step 2/2 : COPY . /usr/share/nginx/html
--> 3bea89c63e3c
Successfully built 3bea89c63e3c
Successfully tagged html-webapplication:latest
```

7. Run the image and access it with public ip of virtual machine

`docker run -d -p 80:80 html-webapplication:latest`

```
ubuntu@ip-172-31-2-3:~/WebDeployment$ sudo docker run -d -p 80:80 html-webapplication:latest
146c89da71dc57f3f7be22d4b4fd0bf32ed71e38aac9761208c6638b34519b05
ubuntu@ip-172-31-2-3:~/WebDeployment$
```

Accessing the app by public ip of virtual machine

