Software Development Life Cycle models and

Syllabus: DevOps Lifecycle for Business Agility, DevOps, and Continuous Testing. DevOps influence on Architecture: Introducing software architecture, The monolithic scenario, Architecture rules of thumb, The separation of concerns, Handling database migrations, Microservices, and the data tier, DevOps, architecture, and resilience.

Software Development Life Cycle models

- **Agile**
- **Lean**
- **Waterfall**
- **Iterative**
- **Spiral**
- **DevOps**

Each of these approaches varies in some ways from the others, but all have a common purpose: to help teams deliver high-quality software as quickly and cost-effectively as possible.

1. Agile

The Agile model first emerged in 2001 and has since become the de facto industry standard. Some businesses value the Agile methodology so much that they apply it to other types of projects, including nontech initiatives.

In the Agile model, fast failure is a good thing. This approach produces ongoing release cycles, each featuring small, incremental changes from the previous release. At each iteration, the product is tested. The Agile model helps teams identify and address small issues on projects before they evolve into more significant problems, and it engages business stakeholders to give feedback throughout the development process.

As part of their embrace of this methodology, many teams also apply an Agile framework known as Scrum to help structure more complex development projects. Scrum teams work in sprints, which usually last two to four weeks, to complete assigned tasks. Daily Scrum meetings help the whole team monitor progress throughout the project. And the ScrumMaster is tasked with keeping the team focused on its goal.

2. Lean

The Lean model for software development is inspired by "lean" manufacturing practices and principles. The seven Lean principles (in this order) are:

- eliminate waste

- amplify learning
- decide as late as possible
- deliver as fast as possible
- empower the team
- build in integrity and
- see the whole.

The Lean process is about working only on what must be worked on at the time, so there's no room for multitasking. Project teams are also focused on finding opportunities to cut waste at every turn throughout the SDLC process, from dropping unnecessary meetings to reducing documentation.

The Agile model is actually a Lean method for the SDLC, but with some notable differences. One is how each prioritizes customer satisfaction: Agile makes it the top priority from the outset, creating a flexible process where project teams can respond quickly to stakeholder feedback throughout the SDLC. Lean, meanwhile, emphasizes the elimination of waste as a way to create more overall value for customers — which, in turn, helps to enhance satisfaction.

3. Waterfall

Some experts argue that the Waterfall model was never meant to be a process model for real projects. Regardless, Waterfall is widely considered the oldest of the structured SDLC methodologies. It's also a very straightforward approach: finish one phase, then move on to the next. No going back. Each stage relies on information from the previous stage and has its own project plan.

The downside of Waterfall is its rigidity. Sure, it's easy to understand and simple to manage. But early delays can throw off the entire project timeline. With little room for revisions once a stage is completed, problems can't be fixed until you get to the maintenance stage. This model doesn't work well if flexibility is needed or if the project is long-term and ongoing.

Even more rigid is the related Verification and Validation model — or V-shaped model. This linear development methodology sprang from the Waterfall approach. It's characterized by a corresponding testing phase for each development stage. Like Waterfall, each stage begins only after the previous one has ended. This SDLC model can be useful, provided your project has no unknown requirements.

4. Iterative

The Iterative model is repetition incarnate. Instead of starting with fully known requirements, project teams implement a set of software requirements, then test, evaluate and pinpoint further requirements. A new version of the software is produced with each phase, or iteration. Rinse and repeat until the complete system is ready.

Advantages of the Iterative model over other common SDLC methodologies is that it produces a working version of the project early in the process and makes it less expensive to implement changes. One disadvantage: Repetitive processes can consume resources quickly.

One example of an Iterative model is the Rational Unified Process (RUP), developed by IBM's Rational Software division. RUP is a process product, designed to enhance team productivity for a wide range of projects and organizations.

RUP divides the development process into four phases:

- Inception, when the idea for a project is set
- Elaboration, when the project is further defined and resources are evaluated
- Construction, when the project is developed and completed
- Transition, when the product is released

Each phase of the project involves business modeling, analysis and design, implementation, testing, and deployment.

5. Spiral

One of the most flexible SDLC methodologies, Spiral takes a cue from the Iterative model and its repetition. The project passes through four phases (planning, risk analysis, engineering and evaluation) over and over in a figurative spiral until completed, allowing for multiple rounds of refinement.

The Spiral model is typically used for large projects. It enables development teams to build a highly customized product and incorporate user feedback early on. Another benefit of this SDLC model is risk management. Each iteration starts by looking ahead to potential risks and figuring out how best to avoid or mitigate them.

6. DevOps

The DevOps methodology is a relative newcomer to the SDLC scene. It emerged from two trends: the application of Agile and Lean practices to operations work, and the general shift in business toward seeing the value of collaboration between development and operations staff at all stages of the SDLC process.

DevOps Unit 2

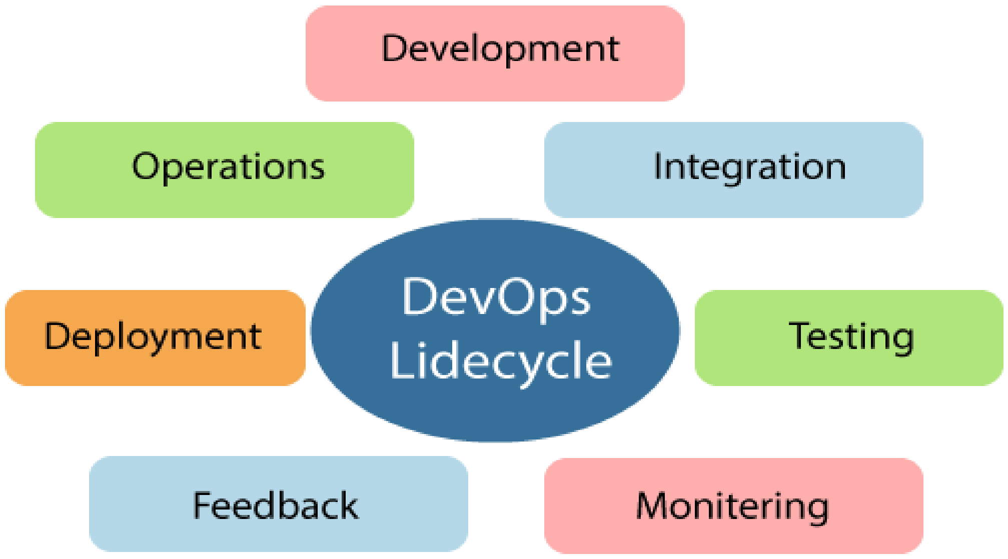
In a DevOps model, Developers and Operations teams work together closely — and sometimes as one team — to accelerate innovation and the deployment of higher-quality and more reliable software products and functionalities. Updates to products are small but frequent. Discipline, continuous feedback and process improvement, and automation of manual development processes are all hallmarks of the DevOps model.

Amazon Web Services describes DevOps as the combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity, evolving and improving products at a faster pace than organizations using traditional software development and infrastructure management processes. So like many SDLC models, DevOps is not only an approach to planning and executing work, but also a philosophy that demands a nontraditional mindset in an organization.

Choosing the right SDLC methodology for your software development project requires careful thought. But keep in mind that a model for planning and guiding your project is only one ingredient for success. Even more important is assembling a solid team of skilled talent committed to moving the project forward through every unexpected challenge or setback.

DevOps Lifecycle

DevOps defines an agile relationship between operations and Development. It is a process that is practiced by the development team and operational engineers together from beginning to the final stage of the product.



Learning DevOps is not complete without understanding the DevOps lifecycle phases. The DevOps lifecycle includes seven phases as given below:

DevOps Unit 2

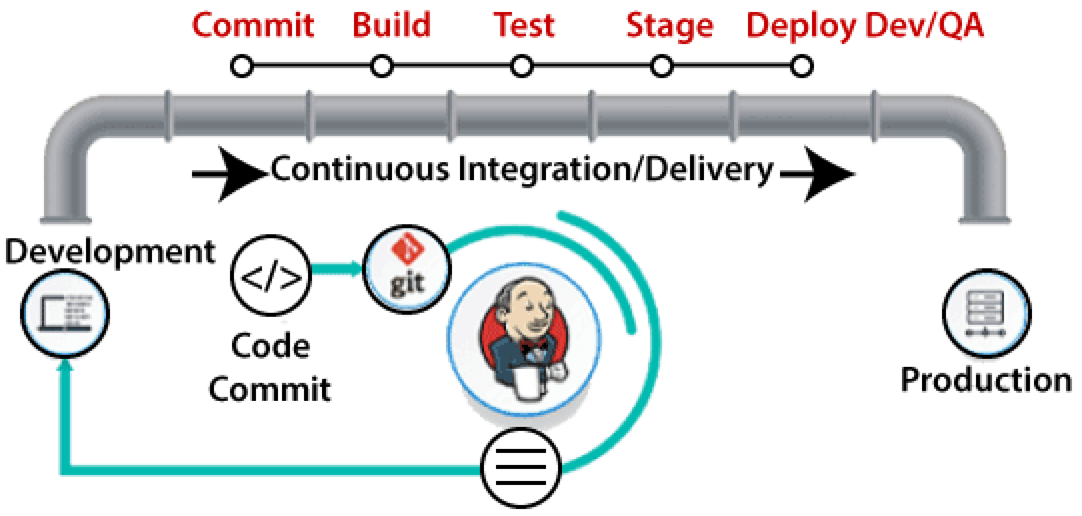
1) Continuous Development

This phase involves the planning and coding of the software. The vision of the project is decided during the planning phase. And the developers begin developing the code for the application. There are no DevOps tools that are required for planning, but there are several tools for maintaining the code.

2) Continuous Integration

This stage is the heart of the entire DevOps lifecycle. It is a software development practice in which the developers require to commit changes to the source code more frequently. This may be on a daily or weekly basis. Then every commit is built, and this allows early detection of problems if they are present. Building code is not only involved compilation, but it also includes **unit testing, integration testing, code review**, and **packaging**.

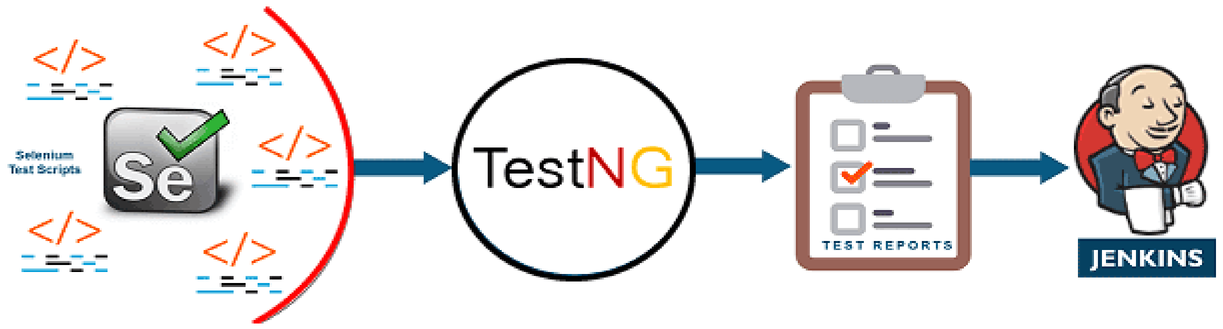
The code supporting new functionality is continuously integrated with the existing code. Therefore, there is continuous development of software. The updated code needs to be integrated continuously and smoothly with the systems to reflect changes to the end-users.



Jenkins is a popular tool used in this phase. Whenever there is a change in the Git repository, then Jenkins fetches the updated code and prepares a build of that code, which is an executable file in the form of war or jar. Then this build is forwarded to the test server or the production server.

3) Continuous Testing

This phase, where the developed software is continuously testing for bugs. For constant testing, automation testing tools such as **TestNG, JUnit, Selenium**, etc are used. These tools allow QAs to test multiple code-bases thoroughly in parallel to ensure that there is no flaw in the functionality. In this phase, **Docker** Containers can be used for simulating the test environment.



Selenium does the automation testing, and TestNG generates the reports. This entire testing phase can automate with the help of a Continuous Integration tool called **Jenkins**.

Automation testing saves a lot of time and effort for executing the tests instead of doing this manually. Apart from that, report generation is a big plus. The task of evaluating the test cases that failed in a test suite gets simpler. Also, we can schedule the execution of the test cases at predefined times. After testing, the code is continuously integrated with the existing code.

4) Continuous Monitoring

Monitoring is a phase that involves all the operational factors of the entire DevOps process, where important information about the use of the software is recorded and carefully processed to find out trends and identify problem areas. Usually, the monitoring is integrated within the operational capabilities of the software application.

It may occur in the form of documentation files or maybe produce large-scale data about the application parameters when it is in a continuous use position. The system errors such as server not reachable, low memory, etc are resolved in this phase. It maintains the security and availability of the service.

Splunk,Nagios,Sensu tool is used for monitoring

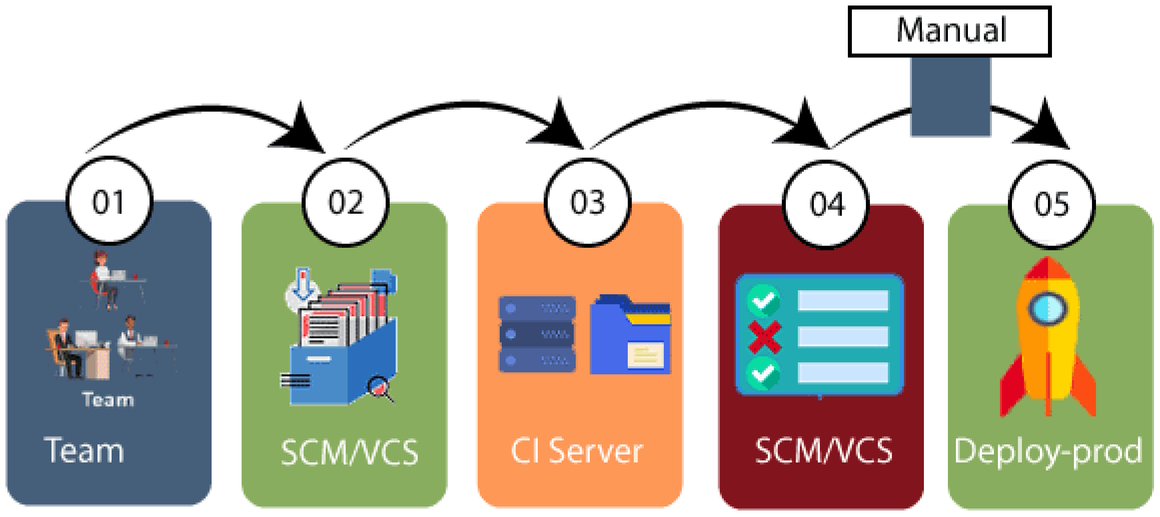
5) Continuous Feedback

The application development is consistently improved by analyzing the results from the operations of the software. This is carried out by placing the critical phase of constant feedback between the operations and the development of the next version of the current software application.

The continuity is the essential factor in the DevOps as it removes the unnecessary steps which are required to take a software application from development, using it to find out its issues and then producing a better version. It kills the efficiency that may be possible with the app and reduce the number of interested customers.

6) Continuous Deployment

In this phase, the code is deployed to the production servers. Also, it is essential to ensure that the code is correctly used on all the servers.



The new code is deployed continuously, and configuration management tools play an essential role in executing tasks frequently and quickly. Here are some popular tools which are used in this phase, such as **Chef, Puppet, Ansible**, and **SaltStack**.

Containerization tools are also playing an essential role in the deployment phase. **Vagrant** and **Docker** are popular tools that are used for this purpose. These tools help to produce consistency across development, staging, testing, and production environment. They also help in scaling up and scaling down instances softly.

Containerization tools help to maintain consistency across the environments where the application is tested, developed, and deployed. There is no chance of errors or failure in the production environment as they package and replicate the same dependencies and packages used in the testing, development, and staging environment. It makes the application easy to run on different computers.

DevOps influence on Architecture

Introducing software architecture

DevOps Model

The DevOps model goes through several phases governed by cross-discipline teams. Those phases are as follows:

Planning,Identify,andTrack Using the latest in project management tools and agile practices, track ideas and workflows visually. This gives all important stakeholders a clear pathway to prioritization and better results. With better oversight, project managers can ensure teams are on the right track and aware of potential obstacles and pitfalls. All applicable teams can better work together to solve any problems in the development process.

Development Phase Version control systems help developers continuously code, ensuring one patch connects seamlessly with the master branch. Each complete feature triggers the developer to submit a request that, if approved, allows the changes to replace existing code. Development is ongoing.

Testing Phase After a build is completed in development, it is sent to QA testing. Catching bugs is important to the user experience, in DevOps bug testing happens early and often. Practices like continuous integration allow developers to use automation to build and test as a cornerstone of continuous development.

Deployment Phase In the deployment phase, most businesses strive to achieve continuous delivery. This means enterprises have mastered the art of manual deployment. After bugs have been detected and resolved, and the user experience has been perfected, a final team is responsible for the manual deployment. By contrast, continuous deployment is a DevOps approach that automates deployment after QA testing has been completed.

Management Phase During the post-deployment management phase, organizations monitor and maintain the DevOps architecture in place. This is achieved by reading and interpreting data from users, ensuring security, availability and more.

Benefits of DevOps Architecture

A properly implemented DevOps approach comes with a number of benefits. These include the following that we selected to highlight:

- **Decrease Cost** Of primary concern for businesses is operational cost, DevOps helps organizations keep their costs low. Because efficiency gets a boost with DevOps practices, software production increases and businesses see decreases in overall cost for production.

- **Increased Productivity and Release Time** With shorter development cycles and streamlined processes, teams are more productive and software is deployed more quickly.

- **Customers are Served** User experience, and by design, user feedback is important to the DevOps process. By gathering information from clients and acting on it, those who practice DevOps ensure that clients wants and needs get honored, and customer satisfaction reaches new highs

.

- **It Gets More Efficient with Time** DevOps simplifies the development lifecycle, which in previous iterations had been increasingly complex. This ensures greater efficiency throughout a DevOps organization, as does the fact that gathering requirements also gets easier. In DevOps, requirements gathering are a streamlined process, a culture of accountability, collaboration and transparency makes requirements gathering a smooth going team effort where no stone is left unturned.

The monolithic scenario

Monolithic software is designed to be self-contained, wherein the program's components or functions are tightly coupled rather than loosely coupled, like in modular software programs. In a monolithic architecture, each component and its associated components must all be present for code to be executed or compiled and for the software to run.

Monolithic applications are single-tiered, which means multiple components are combined into one large application. Consequently, they tend to have large codebases, which can be cumbersome to manage over time.

Furthermore, if one program component must be updated, other elements may also require rewriting, and the whole application has to be recompiled and tested. The process can be time-consuming and may limit the agility and speed of software development teams. Despite these issues, the approach is still in use because it does offer some advantages. Also, many early applications were developed as monolithic software, so the approach cannot be completely disregarded when those applications are still in use and require updates.

What is monolithic architecture?

A monolithic architecture is the traditional unified model for the design of a software program. Monolithic, in this context, means "composed all in one piece." According to the Cambridge dictionary, the adjective monolithic also means both "*too large"* and *"unable to be changed.*"

Benefits of monolithic architecture

There are benefits to monolithic architectures, which is why many applications are still created using this development paradigm. For one, monolithic programs may have better throughput than modular applications. They may also be easier to test and debug because, with fewer elements, there are fewer testing variables and scenarios that come into play.

At the beginning of the software development lifecycle, it is usually easier to go with the monolithic architecture since development can be simpler during the early stages. A single codebase also simplifies logging, configuration management, application performance monitoring and other development concerns. Deployment can also be easier by copying the packaged application to a server. Finally, multiple copies of the application can be placed behind a load balancer to scale it horizontally.

That said, the monolithic approach is usually better for simple, lightweight applications. For more complex applications with frequent expected code changes or evolving scalability requirements, this approach is not suitable.

Drawbacks of monolithic architecture

Generally, monolithic architectures suffer from drawbacks that can delay application development and deployment. These drawbacks become especially significant when the product's complexity increases or when the development team grows in size.

The code base of monolithic applications can be difficult to understand because they may be extensive, which can make it difficult for new developers to modify the code to meet changing business or technical requirements. As requirements evolve or become more complex, it becomes difficult to correctly implement changes without hampering the quality of the code and affecting the overall operation of the application.

Following each update to a monolithic application, developers must compile the entire codebase and redeploy the full application rather than just the part that was updated. This makes continuous or regular deployments difficult, which then affects the application's and team's agility.

The application's size can also increase startup time and add to delays. In some cases, different parts of the application may have conflicting resource requirements. This makes it harder to find the resources required to scale the application.

Architecture Rules of Thumb

1. **There is always a bottleneck.** Even in a serverless system or one you think will "infinitely" scale, pressure will always be created elsewhere. For example, if your API scales, does your database also scale? If your database scales, does your email system? In modern cloud systems, there are so many components that scalability is not always the goal. Throttling systems are sometimes the best choice.
2. **Your data model is linked to the scalability of your application.** If your table design is garbage, your queries will be cumbersome, so accessing data will be slow. When

designing a database (NoSQL or SQL), carefully consider your access pattern and what data you will have to filter. For example, with DynamoDB, you need to consider what "Key" you will have to retrieve data. If that field is not set as the primary or sort key, it will force you to use a scan rather than a faster query.

3. **Scalability is mainly linked with cost. When you get to a large scale, consider systems where this relationship does not track linearly.** If, like many, you have systems on RDS and ECS; these will scale nicely. But the downside is that as you scale, you will pay directly for that increased capacity. It's common for these workloads to cost $50,000 per month at scale. The solution is to migrate these workloads to serverless systems proactively.
4. **Favour systems that require little tuning to make fast.** The days of configuring your own servers are over. AWS, GCP and Azure all provide fantastic systems that don't need expert knowledge to achieve outstanding performance.
5. **Use infrastructure as code.** Terraform makes it easy to build repeatable and version-controlled infrastructure. It creates an ethos of collaboration and reduces errors by defining them in code rather than "missing" a critical checkbox.
6. **Use a PaaS if you're at less than 100k MAUs.** With Heroku, Fly and Render, there is no need to spend hours configuring AWS and messing around with your application build process. Platform-as-a-service should be leveraged to deploy quickly and focus on the product.
7. **Outsource systems outside of the market you are in. Don't roll your own CMS or Auth, even if it costs you tonnes.** If you go to the pricing page of many third-party systems, for enterprise-scale, the cost is insane - think $10,000 a month for an authentication system! "I could make that in a week," you think. That may be true, but it doesn't consider the long-term maintenance and the time you cannot spend on your core product. Where possible, buy off the shelf.
8. **You have three levers, quality, cost and time. You have to balance them accordingly.** You have, at best, 100 "points" to distribute between the three. Of course, you always want to maintain quality, so the other levers to pull are time and cost.
9. **Design your APIs as open-source contracts.** Leveraging tools such as OpenAPI/Swagger (not a sponsor, just a fan!) allows you to create "contracts" between your front-end and back-end teams. This reduces bugs by having the shape of the request and responses agreed upon ahead of time.
10. **Start with a simple system first (Gall's law).** Galls' law states, "all complex systems that work evolved from simpler systems that worked. If you want to build a complex system that works, build a simpler system first, and then improve it over time.". You should avoid going after shiny technology when creating a new software architecture. Focus on simple, proven systems. S3 for your static website, ECS for your API, RDS for your database, etc. After that, you can chop and change your workload to add these fancy technologies into the mix.

The Separation of Concerns

Separation of concerns is a software architecture design pattern/principle for separating an application into distinct sections, so each section addresses a separate concern. At its essence, Separation of concerns is about order. The overall goal of separation of concerns is to establish a well-organized system where each part fulfills a meaningful and intuitive role while maximizing its ability to adapt to change.

How is separation of concerns achieved

Separation of concerns in software architecture is achieved by the establishment of boundaries. A boundary is any logical or physical constraint which delineates a given set of responsibilities. Some examples of boundaries would include the use of methods, objects, components, and services to define core behavior within an application; projects, solutions, and folder hierarchies for source organization; application layers and tiers for processing organization.

Separation of concerns - advantages

Separation of Concerns implemented in software architecture would have several advantages:

1. Lack of duplication and singularity of purpose of the individual components render the overall system easier to maintain.
2. The system becomes more stable as a byproduct of the increased maintainability.
3. The strategies required to ensure that each component only concerns itself with a single set of cohesive responsibilities often result in natural extensibility points.
4. The decoupling which results from requiring components to focus on a single purpose leads to components which are more easily reused in other systems, or different contexts within the same system.
5. The increase in maintainability and extensibility can have a major impact on the marketability and adoption rate of the system.

There are several flavors of Separation of Concerns. Horizontal Separation, Vertical Separation, Data Separation and Aspect Separation. In this article, we will restrict ourselves to Horizontal and Aspect separation of concern.

Handling database migrations

Introduction

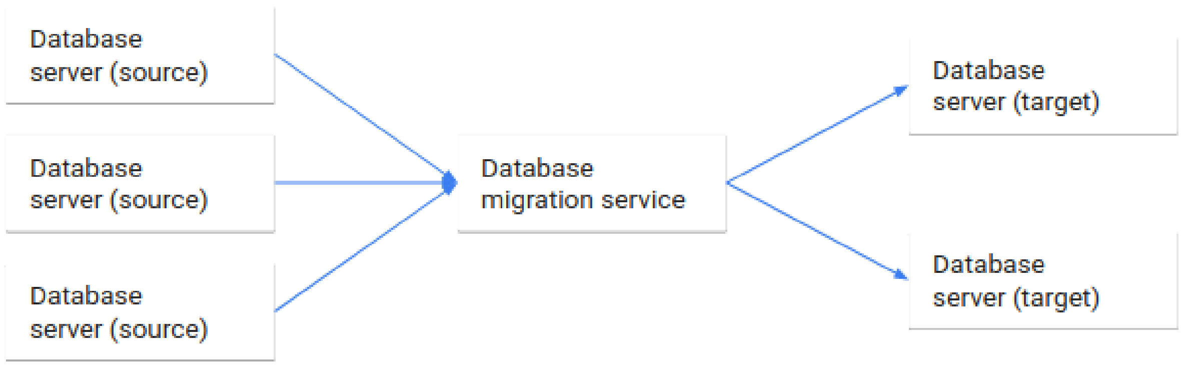
Database schemas define the structure and interrelations of data managed by relational databases. While it is important to develop a well-thought out schema at the beginning of your projects, evolving requirements make changes to your initial schema difficult or impossible to avoid. And since the schema manages the shape and boundaries of your data, changes must be carefully applied to match the expectations of the applications that use it and avoid losing data currently held by the database system.

What are database migrations?

Database migrations, also known as schema migrations, database schema migrations, or simply migrations, are controlled sets of changes developed to modify the structure of the objects within a relational database. Migrations help transition database schemas from their current state to a new desired state, whether that involves adding tables and columns, removing elements, splitting fields, or changing types and constraints.

Migrations manage incremental, often reversible, changes to data structures in a programmatic way. The goals of database migration software are to make database changes repeatable, shareable, and testable without loss of data. Generally, migration software produces artifacts that describe the exact set of operations required to transform a database from a known state to the new state. These can be checked into and managed by normal version control software to track changes and share among team members.

While preventing data loss is generally one of the goals of migration software, changes that drop or destructively modify structures that currently house data can result in deletion. To cope with this, migration is often a supervised process involving inspecting the resulting change scripts and making any modifications necessary to preserve important information.



What are the advantages of migration tools?

Migrations are helpful because they allow database schemas to evolve as requirements change. They help developers plan, validate, and safely apply schema changes to their environments. These compartmentalized changes are defined on a granular level and describe the transformations that must take place to move between various "versions" of the database.

In general, migration systems create artifacts or files that can be shared, applied to multiple database systems, and stored in version control. This helps construct a history of modifications to the database that can be closely tied to accompanying code changes in the client applications.

The database schema and the application's assumptions about that structure can evolve in tandem.

Some other benefits include being allowed (and sometimes required) to manually tweak the process by separating the generation of the list of operations from the execution of them. Each change can be audited, tested, and modified to ensure that the correct results are obtained while still relying on automation for the majority of the process.

State based migration

State based migration software creates artifacts that describe how to recreate the desired database state from scratch. The files that it produces can be applied to an empty relational database system to bring it fully up to date.

After the artifacts describing the desired state are created, the actual migration involves comparing the generated files against the current state of the database. This process allows the software to analyze the difference between the two states and generate a new file or files to bring the current database schema in line with the schema described by the files. These change operations are then applied to the database to reach the goal state.

What to keep in mind with state based migrations

Like almost all migrations, state based migration files must be carefully examined by knowledgeable developers to oversee the process. Both the files describing the desired final state and the files that outline the operations to bring the current database into compliance must be reviewed to ensure that the transformations will not lead to data loss. For example, if the generated operations attempt to rename a table by deleting the current one and recreating it with its new name, a knowledgable human must recognize this and intervene to prevent data loss.

State based migrations can feel rather clumsy if there are frequent major changes to the database schema that require this type of manual intervention. Because of this overhead, this technique is often better suited for scenarios where the schema is well-thought out ahead of time with fundamental changes occurring infrequently.

However, state based migrations do have the advantage of producing files that fully describe the database state in a single context. This can help new developers onboard more quickly and works well with workflows in version control systems since conflicting changes introduced by code branches can be resolved easily.

Change based migrations

The major alternative to state based migrations is a change based migration system. Change based migrations also produce files that alter the existing structures in a database to arrive at the desired state. Rather than discovering the differences between the desired database state and the current one, this approach builds off of a known database state to define the operations to bring it into the new state. Successive migration files are produced to modify the database further, creating a series of change files that can reproduce the final database state when applied consecutively.

Because change based migrations work by outlining the operations required from a known database state to the desired one, an unbroken chain of migration files is necessary from the initial starting point. This system requires an initial state, which may be an empty database system or a files describing the starting structure, the files describing the operations that take the schema through each transformation, and a defined order which the migration files must be applied.

What to keep in mind with change based migrations

Change based migrations trace the provenance of the database schema design back to the original structure through the series of transformation scripts that it creates. This can help illustrate the evolution of the database structure, but is less helpful for understanding the complete state of the database at any one point since the changes described in each file modify the structure produced by the last migration file.

Since the previous state is so important to change based systems, the system often uses a database within the database system itself to track which migration files have been applied. This helps the software understand what state the system is currently in without having to analyze the current structure and compare it against the desired state, known only by compiling the entire series of migration files.

The disadvantage of this approach is that the current state of the database isn't described in the code base after the initial point. Each migration file builds off of the previous one, so while the changes are nicely compartmentalized, the entire database state at any one point is much harder to reason about. Furthermore, because the order of operations is so important, it can be more difficult to resolve conflicts produced by developers making conflicting changes.

Change based systems, however, do have the advantage of allowing for quick, iterative changes to the database structure. Instead of the time intensive process of analyzing the current state of the database, comparing it to the desired state, creating files to perform the necessary operations, and applying them to the database, change based systems assume the current state of the database based on the previous changes. This generally makes changes more light weight, but does make out of band changes to the database especially dangerous since migrations can leave the target systems in an undefined state.

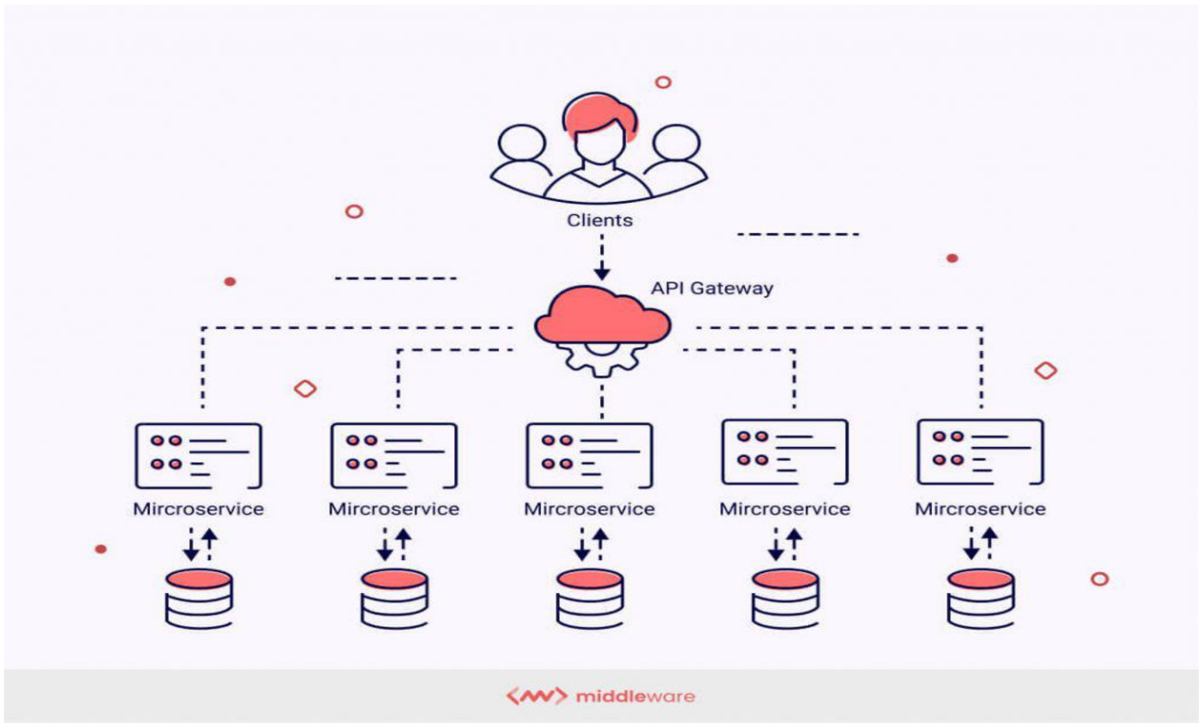
Micro services

Micro services, often referred to as Micro services architecture, is an architectural approach that involves dividing large applications into smaller, functional units capable of functioning and communicating independently.

This approach arose in response to the limitations of monolithic architecture. Because monoliths are large containers holding all software components of an application, they are severely limited: inflexible, unreliable, and often develop slowly.

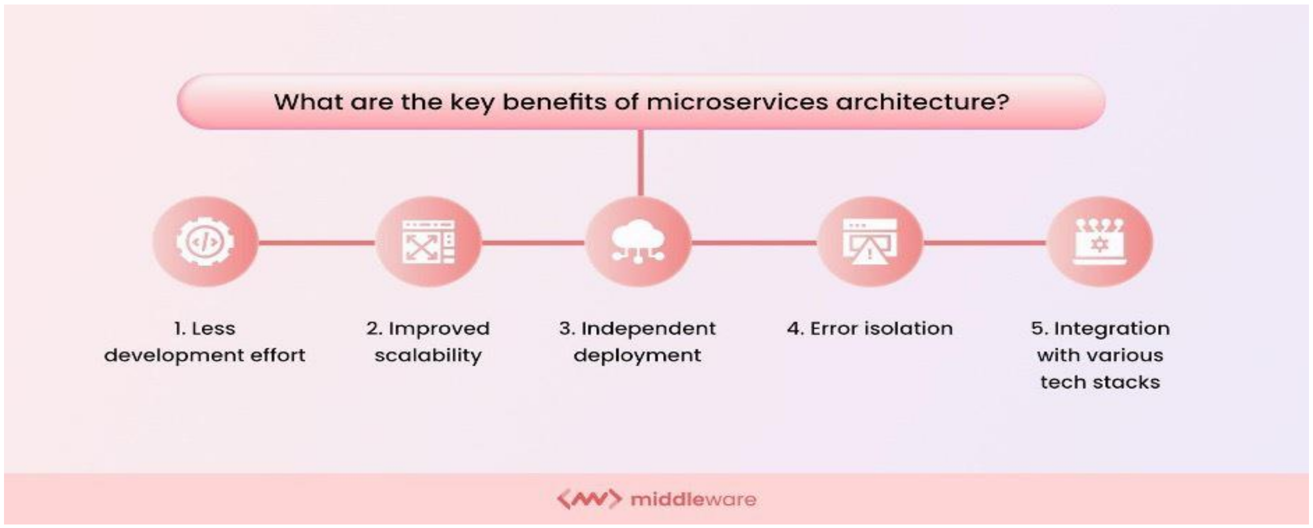
With micro services, however, each unit is independently deployable but can communicate with each other when necessary. Developers can now achieve the scalability, simplicity, and flexibility needed to create highly sophisticated software.

How does microservices architecture work?



The key benefits of microservices architecture

Microservices architecture presents developers and engineers with a number of benefits that monoliths cannot provide. Here are a few of the most notable.


What are the key benefits of microservices architecture?
1. Less development effort
2. Improved scalability
3. Independent deployment
4. Error isolation
5. Integration with various tech stacks
middleware

1. Less development effort

Smaller development teams can work in parallel on different components to update existing functionalities. This makes it significantly easier to identify hot services, scale independently from the rest of the application, and improve the application.

2. Improved scalability

Microservices launch individual services independently, developed in different languages or technologies; all tech stacks are compatible, allowing DevOps to choose any of the most efficient tech stacks without fearing if they will work well together. These small services work on relatively less infrastructure than monolithic applications by choosing the precise scalability of selected components per their requirements.

3. Independent deployment

Each microservice constituting an application needs to be a full stack. This enables microservices to be deployed independently at any point. Since microservices are granular in nature, development teams can work on one microservice, fix errors, then redeploy it without redeploying the entire application.

Microservice architecture is agile and thus does not need a congressional act to modify the program by adding or changing a line of code or adding or eliminating features. The software offers to streamline business structures through resilience improvisation and fault separation.

4. Error isolation

In monolithic applications, the failure of even a small component of the overall application can make it inaccessible. In some cases, determining the error could also be tedious. With microservices, isolating the problem-causing component is easy since the entire application is divided into standalone, fully functional software units. If errors occur, other non-related units will still continue to function.

5. Integration with various tech stacks

With microservices, developers have the freedom to pick the tech stack best suited for one particular microservice and its functions. Instead of opting for one standardized tech stack encompassing all of an application’s functions, they have complete control over their options.

What is the microservices architecture used for?

Put simply: microservices architecture makes app development quicker and more efficient. Agile deployment capabilities combined with the flexible application of different technologies drastically reduce the duration of the development cycle. The following are some of the most vital applications of microservices architecture.

Data processing

Since applications running on microservice architecture can handle more simultaneous requests, microservices can process large amounts of information in less time. This allows for faster and more efficient application performance.

Media content

Companies like Netflix and Amazon Prime Video handle billions of API requests daily. Services such as OTT platforms offering users massive media content will benefit from deploying a microservices architecture. Microservices will ensure that the plethora of requests for different subdomains worldwide is processed without delays or errors.

Website migration

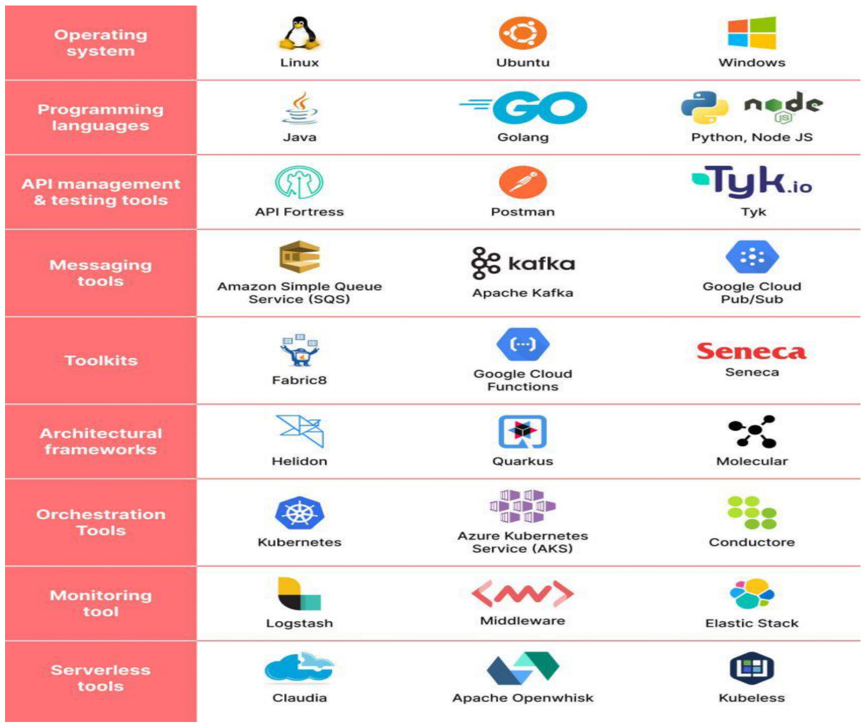
Website migration involves a substantial change and redevelopment of a website's major areas, such as its domain, structure, user interface, etc. Using microservices will help you avoid business-damaging downtime and ensure your migration plans execute smoothly without any hassles.

Transactions and invoices

Microservices are perfect for applications handling high payments and transaction volumes and generating invoices for the same. The failure of an application to process payments can cause huge losses for companies. With the help of microservices, the transaction functionality can be made more robust without changing the rest of the application.

Microservices tools

Building a microservices architecture requires a mix of tools and processes to perform the core building tasks and support the overall framework. Some of these tools are listed below.



Operating system	Linux	Ubuntu	Windows
Programming languages	Java	Golang	Python, Node JS
API management & testing tools	API Fortress	Postman	Tyk
Messaging tools	Amazon Simple Queue Service (SQS)	Apache Kafka	Google Cloud Pub/Sub
Toolkits	Fabric8	Google Cloud Functions	Seneca
Architectural frameworks	Helidon	Quarkus	Molecular
Orchestration Tools	Kubernetes	Azure Kubernetes Service (AKS)	Conductore
Monitoring tool	Logstash	Middleware	Elastic Stack
Serverless tools	Claudia	Apache Openwhisk	Kubeless

1. Operating system

The most basic tool required to build an application is an operating system (OS). One such operating system allows great flexibility in development and uses in Linux. It offers a largely self-contained environment for executing program codes and a series of options for large and small applications in terms of security, storage, and networking.

2. Programming languages

One of the benefits of using a microservices architecture is that you can use a variety of programming languages across applications for different services. Different programming languages have different utilities deployed based on the nature of the microservice.

3. API management and testing tools

The various services need to communicate when building an application using a microservices architecture. This is accomplished using application programming interfaces (APIs). For APIs to work optimally and desirably, they need to be constantly monitored, managed and tested, and API management and testing tools are essential for this.

4. Messaging tools

Messaging tools enable microservices to communicate both internally and externally. Rabbit MQ and Apache Kafka are examples of messaging tools deployed as part of a microservice system.

5. Toolkits

Toolkits in a microservices architecture are tools used to build and develop applications. Different toolkits are available to developers, and these kits fulfill different purposes. Fabric8 and Seneca are some examples of microservices toolkits.

6. Architectural frameworks

Microservices architectural frameworks offer convenient solutions for application development and usually contain a library of code and tools to help configure and deploy an application.

7. Orchestration tools

A container is a set of executables, codes, libraries, and files necessary to run a microservice. Container orchestration tools provide a framework to manage and optimize containers within microservices architecture systems.

8. Monitoring tools

Once a microservices application is up and running, you must constantly monitor it to ensure everything is working smoothly and as intended. Monitoring tools help developers stay on top of the application's work and avoid potential bugs or glitches.

9. Serverless tools Serverless tools further add flexibility and mobility to the various microservices within an application by eliminating server dependency. This helps in the easier rationalization and division of application tasks.

Microservices vs monolithic architecture

With monolithic architectures, all processes are tightly coupled and run as a single service. This means that if one process of the application experiences a spike in demand, the entire architecture must be scaled. Adding or improving a monolithic application's features becomes more complex as the code base grows. This complexity limits experimentation and makes it difficult to implement new ideas. Monolithic architectures add risk for application availability because many dependent and tightly coupled processes increase the impact of a single process failure.

With a microservices architecture, an application is built as independent components that run each application process as a service. These services communicate via a well-defined interface using lightweight APIs. Services are built for business capabilities and each service performs a single function. Because they are independently run, each service can be updated, deployed, and scaled to meet demand for specific functions of an application.

Data tier

The data tier in DevOps refers to the layer of the application architecture that is responsible for storing, retrieving, and processing data. The data tier is typically composed of databases, data warehouses, and data processing systems that manage large amounts of structured and unstructured data.

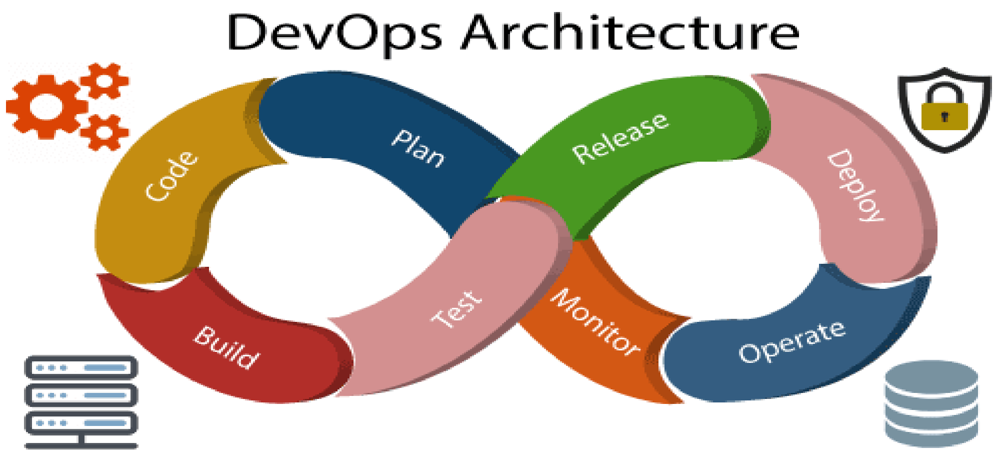
In DevOps, the data tier is considered an important aspect of the overall application architecture and is typically managed as part of the DevOps process. This includes:

1. Data management and migration: Ensuring that data is properly managed and migrated as part of the software delivery pipeline.
2. Data backup and recovery: Implementing data backup and recovery strategies to ensure that data can be recovered in case of failures or disruptions.
3. Data security: Implementing data security measures to protect sensitive information and comply with regulations.

4. Data performance optimization: Optimizing data performance to ensure that applications and services perform well, even with large amounts of data.
5. Data integration: Integrating data from multiple sources to provide a unified view of data and support business decisions.

By integrating data management into the DevOps process, teams can ensure that data is properly managed and protected, and that data-driven applications and services perform well and deliver value to customers.

Devops architecture and resilience

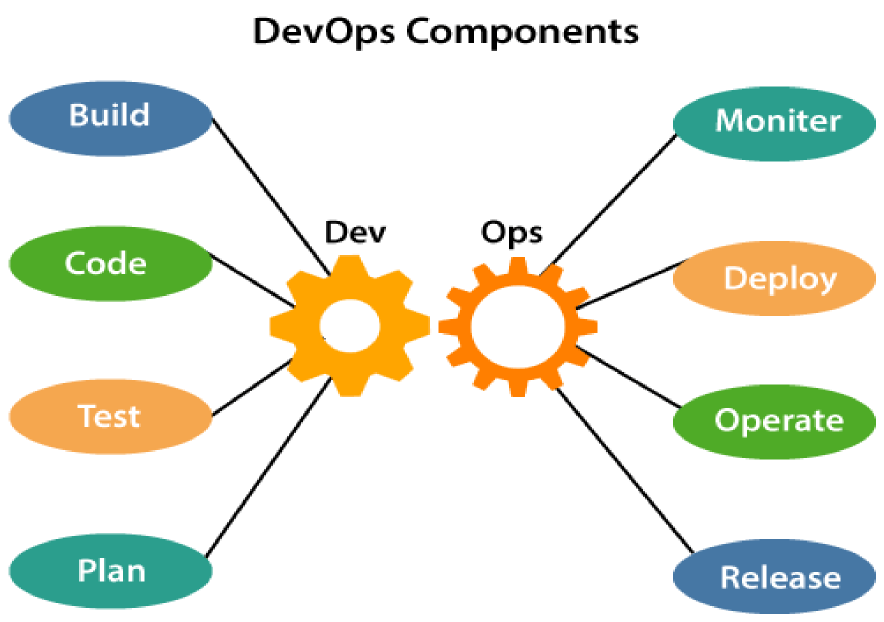


Development and operations both play essential roles in order to deliver applications. The deployment comprises analyzing the **requirements, designing, developing**, and **testing** of the software components or frameworks.

The operation consists of the administrative processes, services, and support for the software. When both the development and operations are combined with collaborating, then the DevOps architecture is the solution to fix the gap between deployment and operation terms; therefore, delivery can be faster.

DevOps architecture is used for the applications hosted on the cloud platform and large distributed applications. Agile Development is used in the DevOps architecture so that integration and delivery can be contiguous. When the development and operations team works separately from each other, then it is time-consuming to **design, test**, and **deploy**. And if the terms are not in sync with each other, then it may cause a delay in the delivery. So DevOps enables the teams to change their shortcomings and increases productivity.

Below are the various components that are used in the DevOps architecture



1) Build

Without DevOps, the cost of the consumption of the resources was evaluated based on the pre-defined individual usage with fixed hardware allocation. And with DevOps, the usage of cloud, sharing of resources comes into the picture, and the build is dependent upon the user's need, which is a mechanism to control the usage of resources or capacity.

2) Code

Many good practices such as Git enables the code to be used, which ensures writing the code for business, helps to track changes, getting notified about the reason behind the difference in the actual and the expected output, and if necessary reverting to the original code developed. The code can be appropriately arranged in **files, folders**, etc. And they can be reused.

3) Test

The application will be ready for production after testing. In the case of manual testing, it consumes more time in testing and moving the code to the output. The testing can be automated, which decreases the time for testing so that the time to deploy the code to production can be reduced as automating the running of the scripts will remove many manual steps.

4) Plan

DevOps use Agile methodology to plan the development. With the operations and development team in sync, it helps in organizing the work to plan accordingly to increase productivity.

5) Monitor

Continuous monitoring is used to identify any risk of failure. Also, it helps in tracking the system accurately so that the health of the application can be checked. The monitoring becomes more comfortable with services where the log data may get monitored through many third-party tools such as **Splunk**.

6) Deploy

Many systems can support the scheduler for automated deployment. The cloud management platform enables users to capture accurate insights and view the optimization scenario, analytics on trends by the deployment of dashboards.

7) Operate

DevOps changes the way traditional approach of developing and testing separately. The teams operate in a collaborative way where both the teams actively participate throughout the service lifecycle. The operation team interacts with developers, and they come up with a monitoring plan which serves the IT and business requirements.

8) Release

Deployment to an environment can be done by automation. But when the deployment is made to the production environment, it is done by manual triggering. Many processes involved in release management commonly used to do the deployment in the production environment manually to lessen the impact on the customers.

DevOps resilience

DevOps resilience refers to the ability of a DevOps system to withstand and recover from failures and disruptions. This means ensuring that the systems and processes used in DevOps are robust, scalable, and able to adapt to changing conditions. Some of the key components of DevOps resilience include:

1. Infrastructure automation: Automating infrastructure deployment, scaling, and management helps to ensure that systems are deployed consistently and are easier to manage in case of failures or disruptions.
2. Monitoring and logging: Monitoring systems, applications, and infrastructure in real-time and collecting logs can help detect and diagnose issues quickly, reducing downtime.
3. Disaster recovery: Having a well-designed disaster recovery plan and regularly testing it can help ensure that systems can quickly recover from disruptions.
4. Continuous testing: Continuously testing systems and applications can help identify and fix issues before they become critical.
5. High availability: Designing systems for high availability helps to ensure that systems remain up and running even in the event of failures or disruptions.

By focusing on these components, DevOps teams can create a resilient and adaptive DevOps system that is able to deliver high-quality applications and services, even in the face of failures and disruptions.