Components of Devops :-

**1. Automated Delivery Pipeline**

**The Pipeline**

First, let’s talk about the ***“pipeline”*** part of this terminology. We want to create a process that defines what needs to happen in what order from the moment new code is published to source control to the last step of making that code available to customers in production. Assuming we have 3 deployment environments (Development, System Integration and Production), then a typical delivery pipeline would have the following steps:

1. Developer pushes code to source control.
2. A build is triggered that will compile the source code and run tests to make sure everything is in order.
3. An artifact is created, given a unique version number, and published to an artifact repository.
4. Deploy the latest artifact to the Development environment at a defined schedule. The schedule could be every hour, 3 times a day, after the successful publishing of an artifact, or whatever suits you.
5. Deploy to the System Integration environment the artifact that was last deployed to the previous environment (i.e. Development environment) at a defined schedule. This schedule can be different from Development environment schedule.
6. Deploy to the Production environment the artifact that was last deployed to the previous environment (i.e. System Integration environment) at a defined schedule. This schedule can be different from the schedule of the other environments.

It’s worth noting that when each step is successfully executed, only then should we trigger the next step. Moreover, when an artifact has been successfully deployed to an environment, only then would it be a candidate for deployment to the next environment. This allows the team to verify an artifact is working appropriately in an environment before advancing it to the next.

**2. Automation**

The second aspect of an Automated Delivery Pipeline is the fact it needs to be ***automated***. Other than the first step when a developer pushes new code to source control, every other step should be automatically triggered and executed. In order to achieve automation, the following points need to be in place:

* Source code should be stored in a ***version control system (CVS)*** like Git.
* We need to use a ***continuous integration server*** that supports automatic triggers and scheduling like [Jenkins](https://jenkins-ci.org/), [TeamCity](https://www.jetbrains.com/teamcity/), [Atlassian Bamboo](https://www.atlassian.com/software/bamboo/), etc.
* The process of compiling and running tests should be automated using a ***build tool*** like [Maven](https://maven.apache.org/), [Gradle](http://gradle.org/), [SBT](http://www.scala-sbt.org/) or [Gulp](http://gulpjs.com/). The build tool also needs to be supported by our build system. For example, we could create a build job on Jenkins that utilizes Maven to compile, test and create an artifact from our source code.
* Once a developer pushes new code, that build job needs to be triggered either by polling or pushing. ***Polling*** means that the job is frequently checking source control to see if any new code has been pushed. On the other hand, ***pushing trigger*** means that the source control system will notify the build job whenever new code has been pushed.
* Every time we build a new artifact, it needs to be given a unique ***version number*** which is greater than any previously generated artifact. For example, if we use a major.minor.patch system of versioning and the last artifact was 2.1.2 then the next artifact should be given 2.1.3, 2.2.0 or 3.0.0 depending on which part we want to increment. I highly recommend you read [Semantic Versioning](http://semver.org/) for guidelines around incrementing MAJOR, MINOR, or PATCH version.
* Deployment to an environment should be on an ***automated schedule***. Most build systems like Jenkins support setting a cron schedule. However, most release management processes would stipulate that deployment to Production environment must be triggered manually in order to have accountability and minimize impact on customers. Hence, it’s common to make the Production trigger manual.
* If there are any ***database scripts*** that need to be executed, these scripts should be treated as source code in the sense that they need to be in version control and their execution should be automated as part of the deployment process. Database scripts of this type are usually called migration scripts. [Liquibase](http://www.liquibase.org/) and [Flyway](http://flywaydb.org/) are excellent examples of tools that can assist with database migration definition and execution.

**3. Configuration Management**

How do we ensure that the application is only talking to resources specific to the environment in which it’s running? Configuration management is the answer.

First step of configuration management is to ***externalizing those configuration*** concerns from source code into a configuration file (e.g. properties file for Java or an App.config file for .NET applications). For example, if we’re developing a Java application then instead of putting the URL of the database as a string in a Java class, the Java class that needs the URL will have to fetch it from a properties file or a system environment variable.

Second step is to determine which configuration file to use. There are two approaches for this:

1. **At Run Time:** When the application is starting up, it will determine the environment in which it’s running and load the appropriate configuration file. Hence, this approach requires a separate configuration file for each environment. Ashraf Sarhan wrote a [blog](http://examples.javacodegeeks.com/enterprise-java/spring/load-environment-configurations-and-properties-with-spring-example/) that walks through an example of using [Spring](https://spring.io/) profile to implement this approach.
2. **At Deployment Time:** The second approach will instead write the configuration file at deploy time. Depending on to which environment we’re deploying, the deployment script will write the configuration file with the appropriate values. An example of the second approach is [Octopus Deploy](https://octopus.com/) which is described in [their documentation](http://docs.octopusdeploy.com/display/OD/Configure+Environment-specific+configuration+transforms+while+keeping+sensitive+values+in+Octopus).

**4. Regular Integration**

Integration in this context is simply to deploy our application to an environment where it will interact with other applications and components in the ecosystem. Having a regular schedule of integrating your application is essential in achieving a feedback cycle. When our integration cycle is tighter, so would our feedback cycle.

The constant enemy of tight integration cycles is manual processes. Hence, every time we want to make the cycle tighter, we will need to automate a process or step that was manual. Examples of manual steps that need to be automated are:

* **Deploying the application:** Automated Delivery Pipeline already helped us with this aspect by requiring to schedule an automated script that can fetch the deployable artifact and deploy it.
* **Testing:** We can only deploy the application to Production once it has been fully tested and certified in the lower environments. Hence, when we automate as much testing as possible, then we can reduce the time needed for new code to be deployed to production. One way of achieving this is by automating end-to-end testing while following a Test Pyramid approach like Martin Fowler describes in his [article](http://martinfowler.com/bliki/TestPyramid.html).
* **Reporting issues:** Instead of waiting for users or testers to report issues, we need to have an automated process to detecting problems and potentially solving them before our users notice them. Automated monitoring and health checks, which we will cover next, can help with this.
* **Troubleshooting:** This generally involves digging through log files which can be time-consuming when you have an application deployed on several machines. This is where log aggregators like [Logstash](https://www.elastic.co/products/logstash), [Graylog](https://www.graylog.org/), and [Splunk](http://www.splunk.com/) help us make the process faster and easier by providing a central place where we can query the logs and see what’s going on.

**5. Automated Monitoring & Health Checks**

Since DevOps involves operational duties, we want to know about problems before they are reported or noticed by users so we can solve them before they impact our users.

Minimally, our health checks should include:

* Checking that all applications are ***reachable and responsive***. For example, we can write an endpoint on our web application that returns the name of the server, IP address, current time date and/or version of the application. That way, when the application returns those values, we know it received our request and was able to process it and we can conclude that the application is alive.
* Checking that ***CPU Utilization*** is not too crazy.
* Checking that ***memory*** consumption is reasonable.
* Checking that there is ample free ***disk storage.***

We need to periodically and automatically check for those facts on each of our applications or resources and when one of them is not behaving appropriately, we need to be alerted automatically by email or text so we address them. Example of tools that assist with this are [HP SiteScope](http://www8.hp.com/us/en/software-solutions/sitescope-application-monitoring/), [CloudFlare](https://www.cloudflare.com/), [AWS CloudWatch](https://aws.amazon.com/cloudwatch/), [AppDynamics](http://www.appdynamics.com/), and more recently [Graylog Alerts](https://www.graylog.org/alerting-notifications-graylog/).

Once you have automated monitoring, it can open the door for more exciting opportunities like auto-scaling your application. For example, if our automated monitoring process detected a spike in CPU and network traffic for an extended duration, then we can hook that up to another automated scaling process that will spin up more instances of the application that will help deal with this extra load. Another example is that if the increase in network traffic was actually malicious like a [DoS attack](https://en.wikipedia.org/wiki/Denial-of-service_attack), then we can react by investigating and potentially blacklisting the suspicious IP addresses.

**6. The Firefighter Role**

When the development team takes on the “ops” duties as part of implementing DevOps, it could lead to the team having to deal with one issue the whole week or half a dozen of issues per day. These issues are interruptions to the team’s development activities which can result in reduced velocity and loss of concentration.

An approach that I like to employ is to define a ***Firefighter role*** and rotate it among the team. All issues that crop up are directed to the Firefighter and she is tasked with:

* Responding to the person(s) who reported the issue
* Gather all the facts about the issue
* Troubleshoot the issue and figure out the root cause
* Solve the issue herself and/or engage the appropriate personnel who can solve it
* Document the issue and solution in the end.
* Organize and conduct a “postmortem” meeting after solving the issue in order to keep the team and stakeholders informed, be transparent and improve the process, code or infrastructure to better handle similar issues in the future.
* Pull in additional help from the team if deemed necessary.

As a result, the firefighter is effectively minimizing the distractions and interruptions from the rest of the team. This role should be rotated among teammates. For example, each teammate takes on the role for a week or two during which this is their main responsibility. When there are no issues to deal with, the firefighter can join the development efforts.

Dealing with issues that crop up at testing environment or production are opportunities to get first-hand experience at what “extraordinary” situations the application has to deal with and then we can come up with some ideas on how to improve the application’s handling of those situations. Those discoveries will certainly help the application evolve and mature.

**7. Infrastructure as Code**

This could be the most recognizable aspect of DevOps but I intentionally list it at the end to emphasize that it’s not the only aspect.

Over the years, many good practices came out of developing software for business problems that are widely adopted but many of those practices are not employed to infrastructure as widely. Treating infrastructure as code does not only mean to write code for infrastructure but it also mean to apply the aforementioned best practices to infrastructure code:

* Put it in **version control** like Git, Subversion and Mercurial. This allows us to track changes, figure out why something was changed, revert back to a working version, etc.
* **Name your scripts properly and organize them** in packages/folders so that we can find scripts easily and know for what they were written.
* Always look for opportunities to **reuse code/scripts** and follow DRY principle (Don’t Repeat Yourself) as much as possible.
* If possible, **automate running the scripts** and remove as many manual steps as possible. In the beginning, documenting and versioning scripts is good enough but after a while we want the ability to run those scripts by clicking a button instead of copying and pasting chunks into a console to run them. Build tools like Jenkins could be your best friend for this task.
* If possible, **test your code** via automated tests. This could the trickiest part specially when we are writing scripts that provision a server or container.

There has been a steady growth in terms of tools that enables us to write, run and test infrastructure code and to mention just a few of them by categories:

* **Server Provisioning:** [Vagrant](https://www.vagrantup.com/), [Puppet](https://puppetlabs.com/), [Chef](https://www.chef.io/chef/)
* **Containerization:** [Docker](https://www.docker.com/), [Docker Compose](https://docs.docker.com/compose/)
* **Task Runners/Build Tools:** tools like [Gulp](http://gulpjs.com/), [Gradle](http://gradle.org/) and [Rake](http://rake.rubyforge.org/) were primarily created for compiling and testing code (e.g. implementing an Automated Delivery Pipeline) but I believe they can be a powerful addition to infrastructure code because:
  + They are very good at helping us design and build a set of re-usable tasks and functions that could be composed into a pipeline of tasks for executing shell scripts, compiling code, publishing artifacts, making API calls, etc.
  + They have plugins and extensions that be leveraged to do common infrastructure tasks like creating zip files, generating files from templates, interacting with databases, etc.

All of the previous DevOps components discussed in this post are not to be treated in a “all or nothing” manner. Instead, it’s better to take them one by one. We can pick one to understand, design and implement. Once we are satisfied, we can move on to the next one.

**Some of the key benefits that companies achieve if they practice continuous delivery include:**

* **Prevent risky releases –** Preventing a risky release is crucial as a release with bug could hamper both the user experience and profitability of the software. Continuous delivery makes deployment process swift and accurate mitigating the risks drastically thereby impacting downtime and performance.
* **Improve time to market –** The traditional software delivery cycle was long as development, testing and fixing bugs used to be sequential. With the help of [regression and pipeline automation](http://www.tothenew.com/devops-chef-puppet-docker), both dev and QA teams can really leverage Agile, work on the roadblocks and improve the build quality.
* **Improved user experience –** Automation helps developers to discover the regressions quickly and fix them continuously without delay. The build quality is superior to the delivery process with an automated deployment pipeline. This leads to better software, zero downtime and improved the [user experience](http://www.tothenew.com/experience-design).
* **Reduced costs** – Companies that invest in test automation, environment provisioning and automation of delivery pipeline prevent risky deployments and also reduce the costs of delivering incremental changes to the product by eliminating various fixes costs associated with shipping of releases.