WHITE PAPER

DevOps for legacy systems – The demand of the changing applications landscape

Abstract

The pace of business is getting faster as game-changers like digitization, cloud computing and big data take over the business world. Enterprises are looking to deploy new features rapidly, resulting in frequent application releases as opposed to the earlier one-time release scenario. To support these frequent releases, businesses need agility and continuous delivery. The DevOps (a blend of 'development and 'operations') approach introduces a collaborative working style in the development and operations teams, leading to rapid and continuous delivery. However, while implementing this methodology for legacy systems, businesses can encounter various challenges. In this paper we outline some of the technical issues in adopting the DevOps methodology for legacy systems and ways to achieve continuous delivery and successfully handle frequent releases.

Infosys
be more

With the advent of the web-scale IT model, increasing competition is driving businesses to introduce differentiating features at a rapid pace. Multivariate testing provides the ability to quickly test different variations of a feature with actual end-users, and choose the one that the customer likes the best. Consequently, the time taken for a feature to be implemented – from ideation to production deployment – is shrinking fast. This has resulted in more frequent application releases and the change has drastically impacted the dynamics of the development lifecycle.

The traditional approach where a development team develops a feature and then passes it on to a separate operations team may not address the needs of the frequent-release scenario. The delays involved in acknowledging, testing and deploying the application in the traditional manner increase the time to market of the feature.

DevOps methodology offers a solution

DevOps (a blend of 'development' and 'operations') is a practice that encourages collaboration between the development and operations teams. Breaking the barriers provides the teams with a holistic view of processes and constraints involved in the workflow of both the teams. The approach provides an understanding that helps design applications for rapid delivery.

The core tenet of the DevOps practice is Continuous Delivery (CD). CD is a set of processes that allows automated deployment and verification of an application across a set of environments. Automation not only reduces manual errors but also allows for quick, reliable and repeatable deployment of rapidly developed code.

When teams adopt the DevOps methodology for a greenfield project, they may find it easier to enable CD than for a legacy system project. While designing and developing a greenfield project, architects and developers start afresh and have the opportunity to take into consideration the requirements of CD. For instance, the developers can write code that is easy to unit test, testers can incrementally build automated test packs, and architects can design the application in a host-agnostic manner. On the other hand, in case of legacy systems, which have evolved over a period of time without any consideration of automation, the adoption of the DevOps approach may result in large-scale refactoring or redesign. It may prove to be a significant challenge to automate the vast amount of legacy code and processes.

DevOps roadmap for continuous delivery

A typical DevOps roadmap involves building a CD pipeline with the supporting capabilities (shown in Figure 1 below). Let us discuss the challenges at each stage in the light of a legacy system project and look at a few suggestions to overcome these.

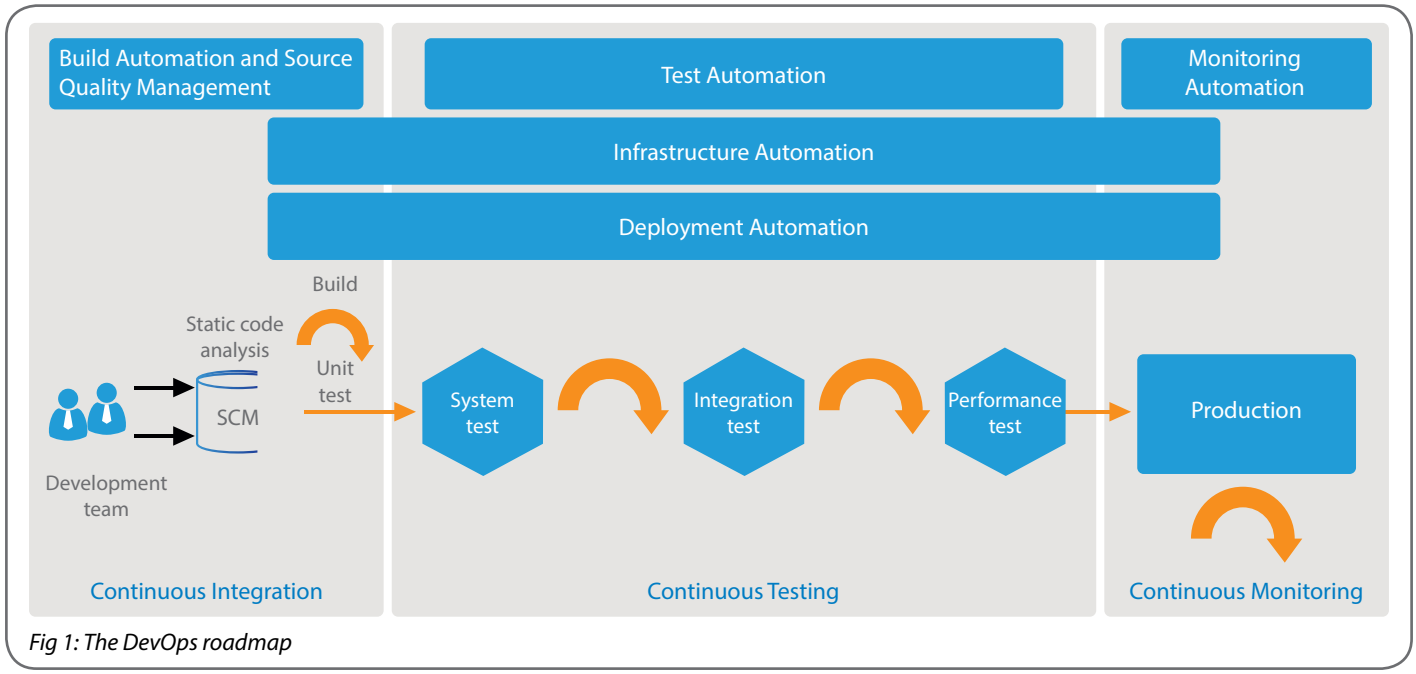


Fig 1: The DevOps roadmap

The adoption of the DevOps methodology and CD in a legacy system project involves three key aspects:

Standardization

Standardization here refers to identifying the small variations in processes that have entered the legacy system over time, and modifying the systems and/or processes to remove these variations. For instance, only one parameterized build script, deployment script and test script must be used for a particular application type (web application, web service, etc.). This script can be reused for any new or existing application by modifying the parameters. At a higher level of abstraction, the infrastructure necessary for the systems must also be standardized. For example, standard scripts must be used to create a web server, application server or database server.

Automation

Once the processes are standardized, they are ready to be automated. Automation involves the use of tools to configure and trigger various scripts. It removes errors introduced by manual intervention and accelerates the processes by eliminating manual involvement and makes them repeatable (so that a process can be configured to activate at regular intervals or on-demand) and reliable (with better alerting and monitoring via the scheduling tool). Ensuring standardization before automation is important since the process of standardization reduces the number of processes to be automated and maintained. Automating a disorganized process can in fact increase the effort required to maintain the various flavors of scripts.

Shifting Left

Identifying processes that can be performed earlier in the development lifecycle rather than at a later point in time is referred to as 'shifting left'. For instance, instead of running a regression test or performance test after the development and system test, a subset of the regression suite or performance test suite can be run during system test in order to catch integration problems and performance issues much earlier in the lifecycle. This can give the development team ample time to work on the issues and takes away the pressure of fixing these in a hurry before the release. This can prevent the team from deferring such issues to the next release. The process of 'shifting left' also reduces the cost of fixing bugs since it is more cost-effective to fix a bug in development than in production.

The use of the same deployment scripts across environments all the way through production ensures that the deployment scripts have been thoroughly tested and bugs have been addressed in lower environments. Enabling a standardized and automated process for build, deployment and testing helps shift left.

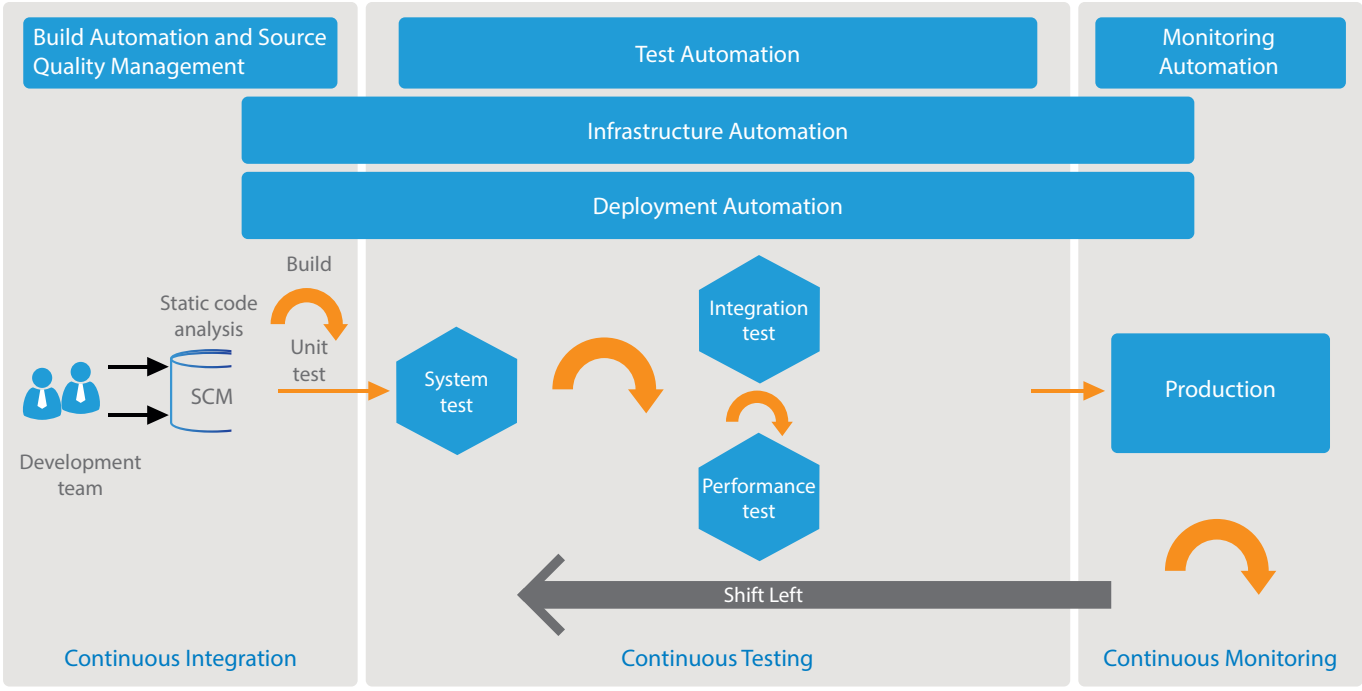
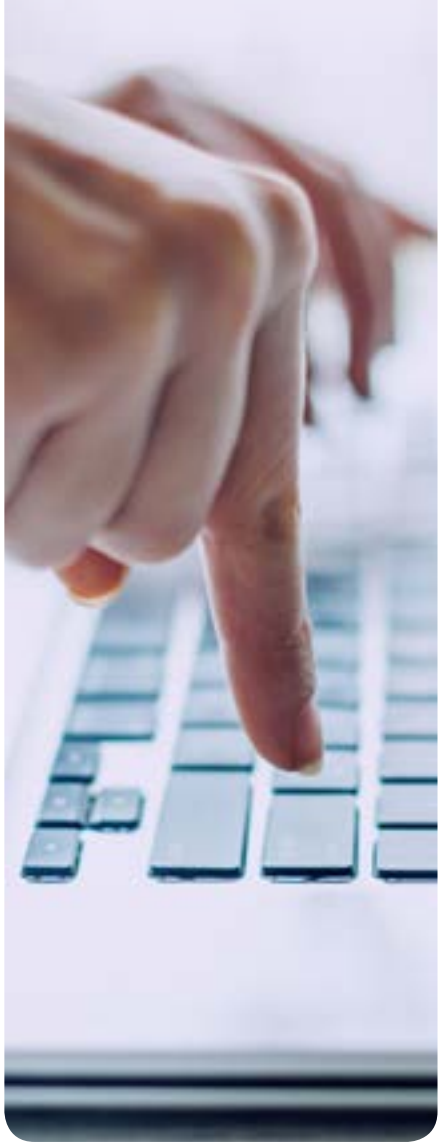


Fig 2: 'Shifting left' – a key aspect of DevOps for legacy systems

Successfully adopting DevOps for legacy systems

Let us look at some suggestions to address the challenges of DevOps for legacy systems. These are ways specific to the various stages involved in a typical CD pipeline.



Infrastructure automation

Infrastructure automation is the process of identifying the infrastructure requirements of the various systems, and creating an automated way of provisioning these infrastructure needs. While infrastructure automation may seem like an optional step, for a true CD pipeline, the team must implement this step to achieve the benefits of complete automation.

Infrastructure automation also promotes efficient use of available hardware or cloud-based machines. The team can spin up a system test environment only when it is needed, and spin it down when it is not needed, allowing some other system to use the infrastructure. In a cloud-based model, this can even translate into direct cost savings. It allows speedy recovery in case of a disaster. The team can spin up new machines in an alternate data center in case of a disaster in the main data center. Manual infrastructure provisioning takes longer and involves the risk of affecting day-to-day business operations by increasing the mean time to recovery of the failed systems.

One of the key challenges of an enterprise with legacy systems is identifying the current infrastructure landscape. Often, each system has its own hardware requirements and is provisioned manually on a need basis. A web server, for instance, could be a Windows 2003 Server with IIS 6.0, 4GB RAM and 4 cores for one system and Windows 2008 Server R2 with IIS 7.5, 8GB RAM and 4 cores for another.

For legacy systems, it is always recommended to standardize the infrastructure requirements into categories, such as web server, application server and database server with small, medium and large instances, based on capacity. These standardized infrastructure needs can then be scripted and automated using tools like Chef[1], Puppet[2], Ansible[3], or Salt[4]. In addition to provisioning, these tools also facilitate maintenance. For example, adding a new file server role on all application server instances is as simple as modifying the base script for application server and running the tool to update the existing instances.

Continuous integration and build automation

Continuous integration (CI) is the process of periodically (mostly on every check-in to the source control) checking out code from the source control system and building it on a clean development environment (typically the build server). In addition to compiling the source code, CI also runs static code analysis tools, unit tests and measures the code coverage. This ensures that the changes being checked in by various members in a team are compatible and that the code works as a whole.

Mostly, in legacy systems CI processes are not followed at all, and even when they are followed, they are ad-hoc. There are legacy systems that do not use version control, or use ancient version control systems not amenable to CI. Branching is another area where legacy systems are deficient, and the choice of a version control system has a direct bearing on the branching strategy adopted. Since the system often evolves over time, there may not be a known configuration for the build server or a standard way of compiling the sources. Non-standard build processes and branching strategies hinder the adoption of CI.

Instead of trying to introduce CI into a legacy system in one step, it is helpful to slowly introduce components of CI. For example, as a first step, the source code can be moved to a modern version control system. SVN[5] and Git[6], for instance, are FOSS central and distributed version control systems respectively that provide good compatibility with other development tools. Simultaneously, the minimum requirements for the build server (compiler version, tools and OS version)

can be identified. Then a standardized build management tool such as MSBuild[7] for .NET or Maven[8] for Java can be adopted. Once a version control system, build server and the necessary tools are in place, there are many build automation tools (both commercial and open source – Bamboo[9], Build Master[10], Jenkins[11] and TFS Build[12]) that can be used to set up automated builds.

Source code quality management

Project Quality Metrics (or PQMs) are metrics that help identify the overall quality of the source code. Some of the key PQMs are Lines of Code, Code Coverage, Complexity, Comments, Package Tangle Index and Dependency Matrix. Measuring and following these metrics helps track the trends in source code quality. For instance, an increasing trend in Lines of Code and a decreasing trend in Code Coverage indicate that the code is being checked in without unit tests. This also allows the team to identify potential areas of concern. For example, a piece of code with no code coverage is a high risk since it has a high probability of failing in unanticipated ways.

Typically, legacy systems have little or no unit tests, causing the code to be extremely fragile. In such cases changes made to one component can cause bugs in an unrelated component. Evolving coding standards and neglect (following the rule of the thumb: "don't fix it if it ain't broke") cause the code to accumulate a large number of static code rule violations. Compromises made during the design process – either due to lack of time or of technical maturity – add to the overall poor quality. This technical debt causes resistance when PQM measurements are introduced in legacy systems.

One way to measure and track PQMs in legacy systems is to start with a baseline number for the metrics and then ensure that any new code added or legacy code re-factored contributes to either maintaining the baseline or improving it. It could be mandated that all new code written must have unit tests and whenever a bug fix or feature is implemented in legacy code, unit tests are written for the legacy code.

By following the Boy Scout rule "Leave it better than you found it"[13], over time the legacy code can be re-factored and brought up to the current standards providing long-term benefits in readability and maintainability of the code.

Test automation

Test automation is the process of automating the execution of tests, publishing the results and measuring the code coverage. Running test cases helps in verifying the behavior of the code and measuring code coverage helps in determining the source code that needs additional testing, which means functional scenarios that are not being tested currently.

Legacy systems tend to have low code coverage due to few or no unit tests. Testing is typically done in higher environments and is manual. As more features are added to a legacy system, the manual testing effort increases drastically, eventually slowing down feature delivery. This problem is amplified when there are multiple teams working on the same code base.

Big-bang test automation for any non-trivial business application is an extremely difficult and time-consuming task since such a system would entail hundreds or thousands of use cases. Moreover, no system is static. There are bound to be additions of new features and bug fixes, further increasing the scope of test automation.

Test automation can be adopted in a legacy system by working with the business team to identify the core test scenarios, prioritizing these by using parameters such as business criticality and risk of failure and automating the top-priority ones first. This risk-based test suite can serve as the regression test suite that is always run to verify the build. This way the existing functionality is not broken due to the changes made in this release. When defects are identified, test cases should be written for these scenarios and added to the regression test suite. When new features are added, test cases for these should be scripted and included in the regression test suite based on business criticality and risk of failure.

Over time, the addition of risk-based and change-based test cases to the automated regression test suite covers the most critical business scenarios. Running such a suite strengthens the confidence of the business about the functional quality of the code.

Deployment automation

Deployment automation is the process of identifying and automating the movement of code to the environment where it is intended to be executed. This also involves automating the setup of the environment itself such as website setup, Secure Sockets Layer (SSL) configuration, etc. The ultimate goal of deployment automation is to make the release a non-event, as opposed to a huge, all hands procedure that needs the entire system to be offline for the period of deployment.

Most legacy systems do not have a standardized deployment process, and practice the 'done-coding,-now throw-it-over-the-wall-to-another-team-for-deployment' approach. Often, the various environments that the code needs to be deployed in through its lifecycle are inconsistently set up, due to which every environment needs a specific deployment procedure. Attempts to automate such disparate processes often result in a disorderly system that is impossible to maintain.

Deployment automation can be gradually adopted in legacy systems by following a step-by-step approach of consolidating the application inventory, standardizing the deployment process by application type and then automating the deployment process. It is important to reuse the same deployment script across environments and application types by 'parameterizing' the deployment script. A web application deployment script, for instance, should be able to deploy any web application as long as the package source and destination web server(s) are specified as parameters.

In order to prevent outages during deployment, feature toggles or active-passive deployment can be used. Feature toggles[14] allow a newly added feature to be turned on or off using a configuration switch. A feature could be deployed to production using a one-server-at-a-time approach with the toggle turned off. The toggle is turned on only when the deployment is complete to all servers and the feature is verified as 'ready for production'. Active-passive deployment involves segregating the servers into sets. A load balancer can control whether traffic is routed to a particular set of servers or to both the sets. The feature can be deployed during a non-peak time on one set of servers which are not servicing requests, and then brought into rotation. The same process is repeated on the second set of servers. This overlapped deployment process prevents a complete outage during deployment.

Infrastructure automation is a key step that helps achieve complete deployment automation since it enables the elimination of environmental inconsistencies.

Monitoring automation

Monitoring is a key process that ensures that systems are functioning correctly and business continuity is intact. Automated monitoring allows proactive detection and resolution of disruptions, rather than reactive or passive issue detection. It involves monitoring system health (server's CPU, memory, disk usage, etc.) as well as application health – for instance, ensuring that the website is up and responding to user requests within the specified SLA.

Automated monitoring provides a close feedback loop with the development team, ensuring that the team is able to address the common cases of system failure and reduce the mean time to recovery.

Legacy systems typically undergo very little or no monitoring. Issues are detected due to customer or end-user complaints. The process of troubleshooting of the reported issues is time-consuming as there are insufficient logs to pinpoint the root cause. In cases where logging is a practice, miscalculated logging configuration causes too many or too few logs. Too many logs make it difficult to find the useful ones for resolving issues.

IT teams can enable legacy systems to adopt automated application monitoring by working with the business team to identify critical business exceptions which impact end-users. Once such exceptions are identified and automated, alerting can be set up using tools such as Nagios[15], Truesight[16], etc. The alerting mechanism should be concise, timely and targeted at the right group. For instance, an alert "Order submission errors – 50 errors in the last 30 minutes" could be sent to the order management development team, when such errors occur. Periodically analyzing the application logs can help identify gaps such as missing logs for a particular use case or too much logging for another, allowing the development team to address these gaps on an ongoing basis.

System health monitoring can be performed by using the features provided by the underlying platform such as perf counters on Windows and collecting and aggregating the metrics exposed in a dashboard. Off-the-shelf systems are also available for monitoring system health. These include SCOM[17] for Microsoft servers and Nagios for other platforms.

Application Performance Monitoring (APM) is another key component of monitoring that allows insight into the actual performance of the system in production. Off-the-shelf systems such as New Relic[18], dynaTrace[19], and AppDynamics[20] are available to help measure and track end-to-end performance metrics.

Conclusion

While we looked at technical challenges in detail, the successful adoption of the DevOps methodology for a legacy system is possible only if the teams working on legacy systems also change their processes and mindset towards Agile and CD. Often, these teams are accustomed to developing features for an extended period of time without deployment into any production-like environment. Their detailing of formal release notes and handovers to operations teams finally result in one-time difficult and complicated deployment. Having a CD pipeline capable of deploying features into production daily or hourly is effective only if feature development is quick and if the operations team is capable of accepting frequent updates from the delivery team.

About the author

Gangadhar Hari Rao has worked in large e-commerce transformational programs for Retail and CPG clients. He has more than 10 years of experience which spans across architecture definition and evaluation, technology strategy for Continuous Delivery and DevOps, implementation, and performance engineering.

References

1. https://www.getchef.com/chef/
2. http://puppetlabs.com/
3. http://www.ansible.com/home
4. http://www.saltstack.com/
5. https://subversion.apache.org/
6. http://git-scm.com/
7. http://msdn.microsoft.com/en-us/library/wea2sca5(v=vs.90).aspx
8. http://maven.apache.org/what-is-maven.html
9. https://www.atlassian.com/software/bamboo
10. http://inedo.com/buildmaster
11. http://jenkins-ci.org/
12. http://msdn.microsoft.com/en-us/library/ms181710(v=vs.90).aspx
13. http://programmer.97things.oreilly.com/wiki/index.php/The_Boy_Scout_Rule
14. http://martinfowler.com/bliki/FeatureToggle.html
15. http://www.nagios.org/
16. http://www.bmc.com/it-solutions/truesight-operations-management.html
17. http://www.microsoft.com/en-in/server-cloud/products/system-center-2012-r2/
18. http://newrelic.com/
19. http://www.compuware.com/en_us/application-performance-management.html
20. http://www.appdynamics.com/solutions/application-performance-management/

For more information, contact askus@infosys.com