DevOps and Test Automation

How to implement effortless automation across the release pipeline
Table of contents

3 Introduction

4 DevOps and automation

6 Creating a continuous delivery pipeline
   7 Local development
   8 Development
   8 Test
   9 Production

10 Automating the continuous delivery pipeline
   11 Regression testing
   12 Early regression / developer verification
   13 Feature regression / individual developer verification
   13 Smoke testing
   14 Business verification

15 Other possibilities for DevOps automation

16 Conclusion
DevOps—the practice of bridging software development and software operations—is a set of practices used to release high quality software into production.

DevOps personnel facilitate collaboration and transparency between the different roles involved in the production of software: Development, Testing, and Operations. This includes defining the release pipeline, building or adopting the right tools for the team, and automating as much of the release pipeline as possible.

Automation is a prerequisite for success with DevOps. Particularly test automation is a key ingredient when it comes to providing fast and accurate feedback to testers and developers, as it enables quick reactions to errors, bugs, and changing requirements.

This paper suggests a framework for defining four different environments across the release pipeline and five critical points at which automation can be implemented. With successful implementation, a seamless pipeline with continuous development and continuous testing can be achieved.
A DevOps professional is someone who understands all the processes involved within software development and testing, as well as the process of bringing the software from local developer computers to a production environment.

The integrated approach of DevOps was introduced as a professional field to counter the lack of agility in software production causing low quality and unstable end-products. One purpose of DevOps is to provide developers with insights into the world of the operations personnel and vice versa.

Historically, developers did not have much insight into the IT environments where their software product would be running. This meant that the developers only experienced their product in local, optimized environments without considering complicating factors, such as firewalls, load balancers, monitoring requirements, and fallback mechanisms.

Similarly, operations personnel would not have sufficient insight into the software product. These people are measured on how well they can make the given software run and are evaluated against key performance indicators such as ‘availability’ and ‘stability’ of the product. If a piece of software released by development is perceived by Operations to have a negative impact on these performance indicators, they can refuse to put the software into production or request more documentation which, in the end, lengthens the product’s time to market.
This lack of transparency between the two teams can lead to conflicts of interest. Furthermore, the shared sense of responsibility for the product’s success is at risk; the teams blame each other for preventing them from doing the job.

In any software production department, one or more DevOps managers should be appointed to reduce these risks. The main objective for the DevOps people is to support the creation of a release pipeline, which is the route a piece of code or feature must travel from a developer’s local computer into production.

There are several best practices to apply in the building of the pipeline. These include Continuous Delivery, Continuous Deployment, and Continuous Integration. A Continuous Delivery (CD) pipeline will serve as the example in this paper. Common for these approaches is that they all aim to mitigate risk involved when releasing software into production. The release pipeline relies heavily on automation, and this is the crux of the matter:

If a release pipeline’s efficiency is dependent on automation, then on a practical level, achieving success with DevOps stands or falls with how well the development department works with automation and which tools are at their disposal.
Creating a continuous delivery pipeline

The ideal release pipeline:

- **Reduces risk of bugs** and increases product quality
- **Increases the agility** of the product’s development
- **Increases productivity and reduces the stress** and last-minute panic often associated with product releases

With the right approach, it is possible to build such a pipeline. This section outlines a DevOps method for this purpose, while the next section identifies five critical points of the pipeline where automation can significantly boost productivity, agility, and product quality.

A standard release pipeline consists of the main elements or environments listed below. Depending on the software and compliance requirements, environments might be added or removed from the pipeline, either permanently or by demand.

- **Local development environments**
  - The individual developers’ personal computers

- **Development server**
  - Or Continuous Integration (CI) server

- **Test servers**
  - For functional UI testing of the product

- **Production environment**
A release pipeline ensures that no code slips into the production environment without being exposed by the so-called ‘gates’. A gate is a set of requirements that must be met for a build (a piece of code) to pass to the next stage, or environment, of the pipeline.

For example, a gate could set the following requirements:

- The code should build without errors
- All unit tests must pass
- All functional UI tests must pass

**Local development**

This environment is the developer’s local computer. This is where the code is produced and added to the source control system. The piece of code can be a new feature or module to the product, or it can be a fix or change to some existing code.
**Development**

This is where the code is built on a build server and deployed to a CI server. After development, the code is ‘checked in’ at the Development Gate. To pass this gate, the following requirements must be met:

- The code can build
- The code must have passed all unit tests
- The code must have passed a local verification by the individual developer

The checks of the Development Gate are critical because the Development environment is designed to imitate the Production environment. This means that if the code works in Development, it is likely to work in Production.

More requirements can be added to the Development Gate. In any case, if any of the checks fail, the check-in should be rolled back and the move to Test should be cancelled.

**Test**

This environment is for product tests of all kinds. It can contain multiple server setups to accommodate for various test types: functional tests, performance tests, load tests, etc.

To pass the Test gate, the following requirements must be met:

- The code must have passed all automated functional UI tests (regression testing)
- The product must have passed a verification of its visual appearance
- The server log files must have been inspected after regression tests
- Performance data must have met an acceptable benchmark
Again, more requirements can be added to this gate. If any of the checks fail, the check-in should be rolled back and the move to Production should be cancelled.

Production

This is where the final software is produced. There is no specific gate following this stage, but there are still checks associated with this final step: Smoke testing is for identifying any bugs before the end-user does. Obviously, this improves the end-user experience as the product is better received.
Automating the continuous delivery pipeline

The effectiveness of a CD release pipeline depends on how fast the results of the Development and Test Gate checks can be fed back to relevant stakeholders. For example, if a unit test fails in Development, the developer who did the check-in should be notified immediately. Or, if the automated UI tests in Test fail, an alert could appear on a dashboard monitored by the responsible team.

The faster the feedback mechanism, the more likely it is to find a quick solution to the given issue. If a developer is notified weeks after error detection in a checked-in piece of code, he or she has most likely moved on to other tasks and projects, and the context and relevant requirements of the given code is not on top of his or her mind. The developer would have to spend time readjusting to the right context, resulting in significant productivity loss.

The answer to how to speed up the feedback mechanism in the release pipeline is automation. Ideally, all the automation needs of the product delivery process should be manageable by using one tool that enhances the work of all the roles involved in the pipeline; from testers to product and business owners.

With the Leapwork Automation Platform as an example, what follows is a walkthrough of five junctures in the release cycle where automation can be implemented.
Regression testing

The primary area of automated testing is regression testing. This helps ensure that changes to a product base do not create bugs in the existing product. Regression testing usually takes place in the Test environment.

By relying on Leapwork, and without having to write a single line of code, the test team can automate and maintain the test cases required for passing the Test Gate. The cases can run on a schedule as often as needed or on an ad hoc basis.

The Leapwork Automation Platform comes with native plugins to the common build and release services such as Jenkins, TFS, Atlassian Bamboo, and TeamCity. This makes it very easy to trigger test cases to run as part of a release plan. Furthermore, Leapwork returns all test case results in JUnit format, which is natively supported by all platforms, enabling an overview of all test cases directly via any release platform an organization is using.

Leapwork comes with native plugins for the most common DevOps tools and pipeline orchestrators
It is very useful to have individual test cases be part of more than one schedule. This way, it is possible to test minor parts of the products often. At the same time you can have a schedule containing regression tests of all product features which is executed every night or close to a release.

Automation makes feedback instantly available. After test execution, all case results are available in the Leapwork Automation Platform, and it is possible to filter them by projects, schedules, and more. The results can also be emailed to designated stakeholders depending on the case outcome. Finally, test results can be automatically pushed to whichever bug management system is used (e.g. TFS, Jira, or HP Quality Center).

**Early regression / developer verification**

Running regression tests against the Development environment serves two purposes:

- **Making sure that the build is testable**, that is, ensuring that it makes sense to push to Test.
- **Providing feedback to developers about whether the Development environment is reliable**; if the code works here it can be assumed that it also works in Test and Production.

Early regression testing only covers a small part of the total regression tests available, for two reasons. Firstly, there is an upper limit to the number of early regression tests that can be executed per day because there is a higher frequency of deployment, or check-ins of code, to Development. Secondly, verification at this stage is usually kept at a limited number of basic test cases focusing on critical features.
The schedules for early regression testing are usually triggered as part of the same process that generates the build and deploys the code to the Development server. This is easily done with Leapwork’s scheduling capabilities.

Feature regression / individual developer verification

The third type of regression testing that is easily automated with Leapwork is feature verification on the individual developer’s local installation. With Leapwork, teams of testers, developers, and DevOps professionals can share and collaborate on test cases, and everyone can run them on their individual PCs if needed. The results of local runs provide valuable feedback to the entire team.

Smoke testing

“Errors will occur.” This is a well-known and accepted truth in software production. So, during a release cycle, it is never a question of “if errors will occur”, but rather how the organization will react to them. Enter the discipline of smoke testing. This is about running automated test cases against the Production environment to ensure that the software indeed runs as intended.

Tests done at this stage are the closest thing to real user interaction. An example of a test case could be to log in to a web application and then validate the login.

The test cases used in smoke testing are solely used for this purpose. The outcomes of the tests are not allowed to dictate any drastic changes to the product. Smoke tests should be able to run both frequently and quickly, and they should not be allowed to push the production system to its limits.
Business verification

When the owner of the system under test—the product/business owner—needs to configure and set up the system for end-users, the Leapwork Automation Platform can be used to ensure the validity of the system from a business perspective.

Example of Leapwork Automation Platform flow

The short learning curve of the Leapwork Automation Platform and its flowchart-based approach to automation makes it an ideal tool for business owners, managers, and non-technical specialists. These people can design and execute automated test cases to monitor that the business-critical aspects of the product works.
Other possibilities for DevOps automation

The release pipeline can be supported in various ways by automating repetitive DevOps tasks, for example daily baselining of databases. This is often a complicated and time-consuming procedure, consisting of the following steps:

- Get database back-up file
- Copy DB file to local folder
- Attach DB file to local DB folder
- Open DB file
- Run script to anonymize data
- Deploy DB file to Development and Test environments
- Restart servicer farms
- Verify that system is running
- Deploy DB file to Development and Test environments
- Restart servicer farms
- Verify that system is running

This process can be automated with Leapwork. Designing the automation case would take less than an hour, and from then on it can be scheduled for daily, automated runs. This comes with a range of benefits:

- Potential maintenance of a case is kept to an absolute minimum
- The case execution is handled centrally by one platform with a designated service account
- The individual steps of the process can be easily distributed to remote machines
- Teams can collaborate on cases which can change hands without an extensive handover
Conclusion

To supercharge your DevOps and support a highly efficient release cycle, start by mapping out your release pipeline and automate where relevant with the Leapwork Automation Platform.

With Leapwork Automation Platform automation can be implemented through every stage of the release pipeline:

- **Local development**: Feature regression / individual developer verification
- **Development**: Early regression / developer verification
- **Test**: Regression testing
- **Production**: Smoke testing, Business verification
You can use the illustration above as a template, or you can map out the process from scratch by following these steps:

1. **Identify all stages** – or environments – of your release
2. **Identify the gates and requirements** necessary in a build’s journey to production
3. **Identify the feedback mechanism** required for quick error detection and solving
4. **List all the operational procedures, services, and actions** involved with a release cycle
5. **Assign the items of step 4 to the relevant environments** and gates
6. **Identify which parts of steps 4 and 5 can**, and should be, automated to increase productivity

**A tool for supplementing DevOps with automation must**

- Support every stage of the release pipeline
- Be usable by testers, developers, operations personnel, and management alike
- Be able to run in any kind of infrastructure
- Keep maintenance of automation cases at an absolute minimum
- Not rely on users to write code, no matter how complex their automation needs are
- Have a very short learning curve to not disrupt release cycles
- Not take time away from testers’ primary responsibility; challenging the end-product
- Make it very easy to collaborate on automation cases
Combine DevOps with the Leapwork Automation Platform and reap the following benefits:

- **Do more product releases** per year without increasing development spending
- **Heighten and maintain product quality** by exponentially increasing the number of tests across the release pipeline
- **Drastically reduce the time from error detection to handling** and solution
- **Improve communications**, documentation, and feedback between teams
- **Rely on just one tool** for regression testing, smoke testing, business verification, and any automated monitoring needs
- **Never waste time typing or reading code**, whether it is for designing, executing, troubleshooting, or reconfiguring automation cases
- **Automate in days**, not weeks or months – and thereby avoid disrupting release plans

---

**Improve productivity, become more agile**

Tell us your use case and let us show you what Leapwork can do, or start a trial to experience the simplicity of our universal, visual automation language immediately.