CLEARED
For Open Publication
Oct 09, 2018
Department of Defense
OFFICE OF PREPUBLICATION AND SECURITY REVIEW

Is Your Development Environment Holding You Back? A DIB Guide for the Acquisition Community

Version 0.2, last modified 3 Oct 2018

A strong software development team is marked by some common attributes, including the use of practices, processes, and various tools.

An effective team starts with clear goals. The entire software team should have a clear sense of the project's goals and the value they seek to provide "the client." The goals should be translated into specific objectives, which may be measured in terms of agreed-upon key performance indicators (KPIs) or other frameworks. An effective development environment is one designed to deliver value towards those goals. (This KPI-driven paradigm should *not* be seen as an invitation to reprise an extended debate about requirements.)

Technical practices and processes that enable a development environment to deliver value towards those goals include:

- Organization through discrete "user stories" that can be broken down into smaller components and continually prioritized by the product owner
- Relatively short "sprints" (often two weeks), each ending in a retrospective, that enable measurement and learning throughout the process
- Blameless post-mortems that allow for maximum learning and speedy recovery from failures
- Automated testing, security, and deployment
- Testing (including user testing) and security should be shifted to the left and be part of the day-to-day operations within the development teams
- Continuous integration, in which developers integrate code into a shared repository several times a day, and check-ins are then verified by an automated build for early problem detection
- Continuous delivery or continuous deployment, in which the software is seamlessly deployed into staging and production environments
- Trunk-based development, in which team members work in small batches and develop off of trunk or master, rather than long-lived feature branches
- Version control for all production artifacts including open source and third party libraries
- Infrastructure as code: version control for all configuration, networking requirements, container orchestration files, continuous integration/continuous delivery (CI/CD) pipeline files
- Ability to execute A/B testing and canary deployments
- Ability to get rapid and continuous user feedback and to test new features with users throughout the development process

Effective teams will practice continuous delivery, in which teams deploy software in short cycles, ensuring that the software can be reliably released at any time. Continuous deployment can be measured by a team's ability to achieve the following outcomes:

- Teams can deploy on-demand to production or to end users throughout the software delivery lifecycle.
- Fast feedback on the quality and deployability of the system is available to everyone on the team, and people make acting on this feedback their highest priority.

Specific measures that will help you gauge if your development environment is working as it should include development frequency; lead time for changes; time to restore service after outage; and change failure rate (rollback deployed code). These questions and data, borrowed from the 2017 State of DevOps Report from DORA, can help assess where your teams stand:

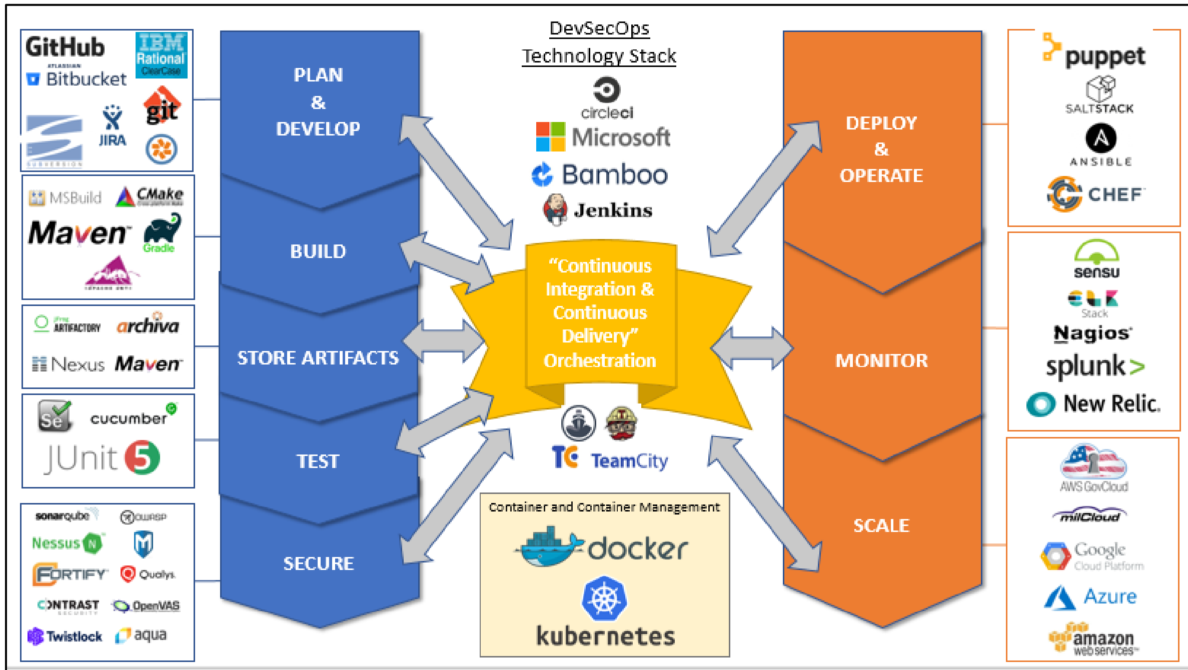
	High performance	**Medium performance**	**Low performance**
Deployment frequency How often does your organization deploy code?	On demand (multiple deploys per day)	Between once per week and once per month	Between once per week and once per month
Lead time for changes What is your lead time for changes (i.e., how long does it take to go from code-commit to code successfully running in production)?	Less than one hour	Between one week and one month	Between one week and one month*
Mean time to recover (MTTR) How long does it generally take to restore service when a service incident occurs (e.g., unplanned outage, service impairment)?	Less than one hour	Less than one day	Between one week and one day
Change failure rate What percentage of changes results either in degraded service or subsequently requires remediation (e.g., leads to service impairment, service outage, requires a hotfix, rollback, fix forward, patch)?	0-15%	0-15%	31-45%

* Low performers were lower on average (at a statistically significant level), but had the same median as the medium performers (2017 DevOps Report)

There is ***no exact set of tools*** that indicate that your development environment is working as it should, but the use of some tools will often indicate that the practices and processes above are in place. You commonly see effective software teams using:

- An issue tracker, like Jira or Pivotal Tracker
- Continuous integration and/or continuous integration/continuous delivery (CI/CD) tools, like Jenkins, Circle CI, or Travis CI

- Automated build tools, like Maven, Grable, Cmake, and Apache Ant
- Automated testing tools, like Selenium, Cucumber, J-Unit
- A centralized artifacts repository, like Nexus, Artifactory, or Maven
- Automated security tools for static and dynamic code analysis and container security, like Sonarqube, OWASP ZAP, Fortify, Nessus, Twistlock, Aqua, and more.
- Automation tools, like Chef, Ansible, or Puppet
- Automated code review tools, like Code Climate
- Automated monitoring tools, like Nagios, Splunk, New Relic, and ELK
- Container and container orchestration tools like Docker, Docker Swarm, Kubernetes, and more



Warning signs that you may have screwed up your development environment include:

- If teams cannot effectively track progress towards defined goals and objectives roughly every two weeks
- If teams cannot rapidly deploy various environments that mirror production to test their code such as in development, QA, and staging
- If teams cannot have real-time feedback regarding their code building, passing tests, and passing security scans
- If it takes months for end users to be able to see changes and provide feedback
- If teams cannot rapidly roll-back to previous versions or perform rolling-update to new versions without downtime
- If recovering from incidents results in significant drama or the assignment of blame
- If having code ready to deploy is a big event (it should happen routinely and without drama)
- If changes to the software frequently result in breaking it
- If developers are not empowered to change the code or build new functionality based on user feedback, or to change their process based on what they learn