Best practices for a DevOps approach with IBM System z

Highlights

Why DevOps?

- Align portfolio decisions with business goals
- Achieve innovative ideas through collaborative development
- Deliver innovation by automating processes and eliminating waste
- Enable continuous integration with rapid testing
- Improve processes and products using feedback from internal and external customers

Today's enterprise applications are complex. While they provide users with a modern interface that can fit literally into the palm of their hands, applications still need to access back-end systems and operational databases, as well as integrate with third-party application programming interfaces (APIs), to provide the business functionality customers demand.

DevOps is an approach for software delivery based on "lean" and "agile" principles, in which all stakeholders—from line of business to development, quality assurance and operations—collaborate to deliver software more efficiently based on a continuous feedback loop. Adopting DevOps capabilities and principles can result in applications that are more efficient and effective, with continuous process improvement, while helping ensure that the changes and enhancements to the software are based on real customer feedback.

These concepts are not new, but are based on principles made popular by thought leaders in the *lean manufacturing* movement, such as Dr. William E. Deming, who proposed the PDCA cycle—Plan, Do, Change, Act (or Adjust)—to continuously improve manufacturing quality. Based on this core approach, the lean manufacturing movement aims to both continuously improve the product being manufactured and reduce waste in the manufacturing process. DevOps extends these basic principles from the lean and agile approaches to the entire software delivery lifecycle.

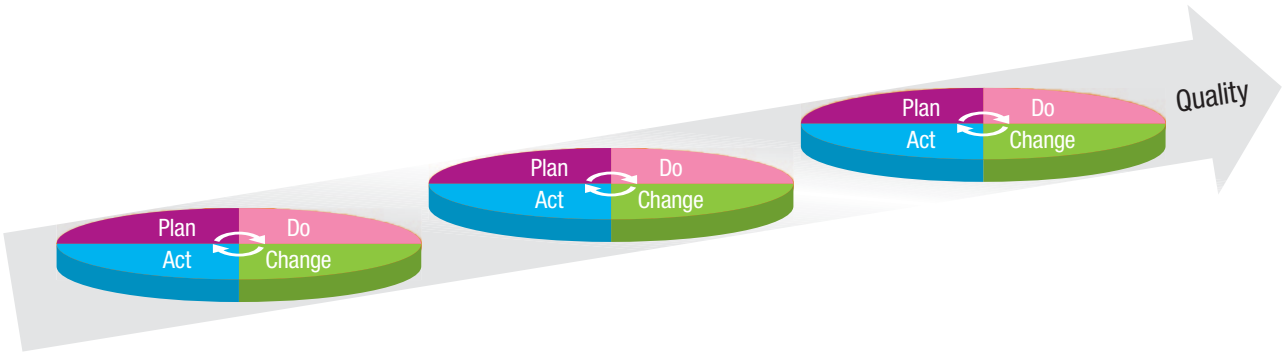

Figure 1. The Plan-Do-Change-Act cycle

The evolution of DevOps

The DevOps approach grew, as its name suggests, from the desire of organizations and teams to break down the barriers and link disconnected processes between development (Dev) and operations (Ops) teams. A lack of communication and trust between these organizations frequently created bottlenecks in the software delivery lifecycle. The result was a complex, disruptive and painful software deployment process—for an undertaking that was carried out as infrequently as possible. What is more, this lack of communication also made obtaining feedback from customers who used the software and conveying this information back to development and lines of business very cumbersome.

DevOps principles, as a result, are geared toward removing the bottlenecks between development and operations while strengthening communication and trust among various software delivery teams. The goal is to deliver small sets of features or updates to production and get them into the hands of customers more frequently, and then to obtain feedback and respond more quickly—all while improving the software delivery processes to reduce overhead, waste and rework.

As the DevOps approach matured, its principles began to extend across the entire software delivery lifecycle, enabling feedback to pass to business owners and lines of business. The DevOps approach also allowed the act/adjust portion of the PDCA cycle to influence not just the functionality of software, but also the release plans, requirements, development and testing environments, and the process of delivering the software—ultimately enabling the entire PDCA cycle to be applied to end-to-end software delivery. This maturity has allowed DevOps principles to be applied not only to systems-of-engagement applications, but also to more traditional systems-of-record applications, as organizations struggle to keep the balance between innovation and stability in the applications they deliver to customers in rapidly evolving and competitive markets.

This paper introduces DevOps and explores the nuances and special considerations for adopting the DevOps approach for enterprise systems developed and/or hosted on IBM® System z® mainframes. While the principles of DevOps do not change based on the introduction of a platform such as IBM z/OS®, they do require special consideration and effort in their implementation. This is especially true for tooling, as System z typically has had its own tooling that was designed long before modern development practices. Many z/OS-based environments still rely on traditional waterfall development processes and green-screen tools that have been in use for more than 20 years.

Given the true multi-platform nature of today's enterprises, with the presence of mobile, cloud, distributed, and IBM i and/or System z applications—all of which need to be created, integrated, deployed and operated—the need for the efficiencies, streamlining and collaboration that DevOps provides is becoming a key competitive differentiator.

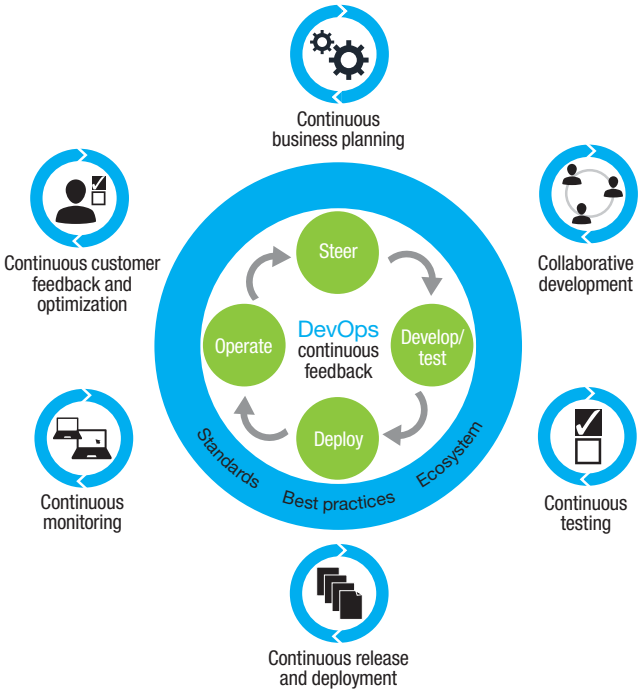

Figure 2. Enabling feedback with DevOps principles

The capabilities of DevOps

Putting into place the principles of DevOps requires adopting a set of capabilities that enable rapid, efficient and effective deployment of—and feedback regarding—the software being delivered. This requires that cross-functional teams including line-of-business stakeholders, architects, developers, quality assurance personnel, integrators and operations staff collaborate on and execute one coherent release plan.

The capabilities required for success include:

- Continuous business planning
- Collaborative development
- Continuous testing
- Continuous release and deployment
- Continuous monitoring
- Continuous customer feedback and improvement

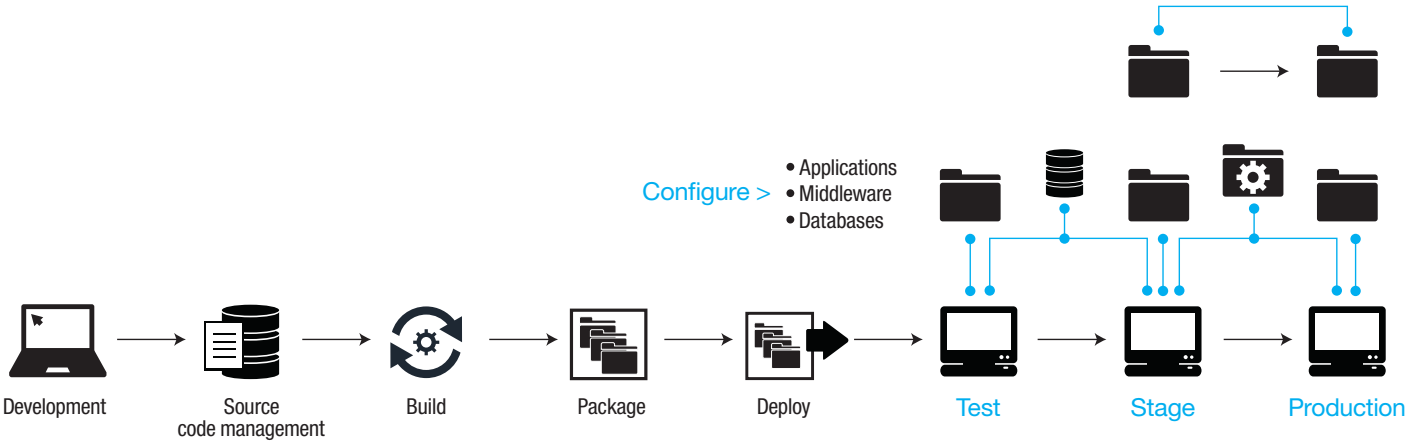

Figure 3. The delivery pipeline

This paper will focus on three capabilities of DevOps: *collaborative development*, *continuous testing*, and *continuous release and deployment*. These practices are inherently interconnected and interdependent. Together they form what is known as a "delivery pipeline" (Figure 3). This paper will describe the components of the pipeline, with the definitions of continuous integration, continuous delivery and continuous testing.

Collaborative development

Delivering software applications or systems today involves multiple teams of developers working on separate components of the application. Typically the completed application would need to interact with other applications or services to perform its functions. Some of these external applications or services may be systems-of-record applications that exist in the enterprise or they may be external third-party services. There is, as a result, an inherent need for developers to integrate their work with components built by other development teams and with other applications and services.

This need makes integration an essential and complex task in the software development lifecycle. The process is commonly referred to as *continuous integration*, and it is a key part of collaborative development. In previous development processes, integration was a secondary set of tasks conducted after the components, or sometimes the complete application, were built. This sequence was inherently costly and unpredictable, as the incompatibilities and defects that tend to be discovered only during integration were discovered late in the development process. The result, typically, was a significant increase in rework and risk.

The "agile" movement introduced a logical step to help reduce this risk by integrating components continuously (or as continuously as possible). In this step, developers integrate their work with the rest of the development team regularly (at least daily) and test the integrated work. In the case of enterprise systems, which span multiple platforms, applications or services, developers also integrate with other systems and services as often as possible.

These steps to integrate results can lead to early discovery and exposure of integration risks. In the case of enterprise systems, they can also expose known and unknown dependencies related to both technology and scheduling that may be at risk. As these practices have matured, some organizations have adopted continuous integration practices that developers follow every time they check in code. In the most mature organizations, continuous integration has led to capabilities for continuous delivery in which the code and components are not only integrated, but are also delivered to a production-like environment for testing and verification.

In many enterprises, mainframe systems based on the z/OS platform still support the core of their enterprise applications, both for application hosting and for data storage. In these environments, continuous integration of enterprise applications is inherently dependent on System z—either as the development and/or integration platform or as the platform that houses applications and services that applications in development will be integrated with.

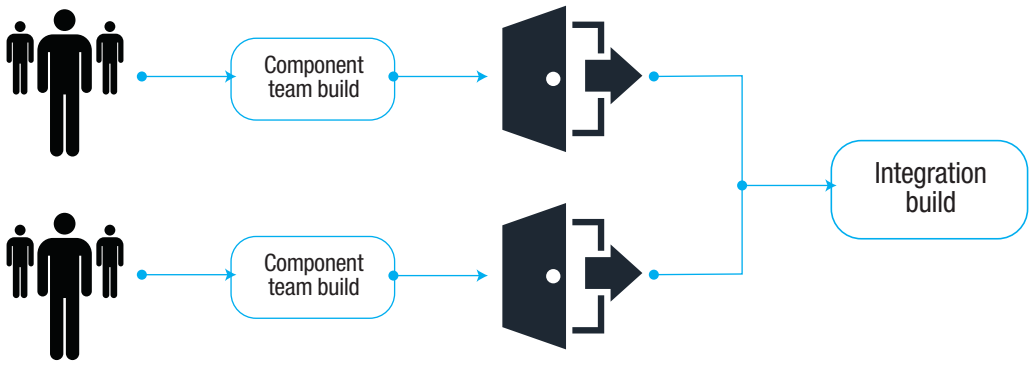

Figure 4. Collaborative development with continuous integration

Agile development and System z

The demands placed by business and customers on development organizations have driven the wide adoption by development teams of agile development practices. These practices are aimed at reducing the gap between the business (or the customers) and the development teams. They work primarily in three ways:

- By breaking the development effort into small chunks of work that can be completed in time-bound iterations. This allows developers to identify and resolve risk earlier than when they undertake entire projects or larger portions of projects.
- By including contact with the end user or a surrogate representing the user into the development iterations. This helps give developers a better understanding of the user's needs and allows for changing needs to be more quickly accommodated.
- By releasing software at the end of every iteration. This allows developers to demonstrate regularly what they have built in order to obtain user feedback.

As described, continuous integration is one of the tenets of agile development. It allows for developers to integrate their software components with components being developed by others—either internally or externally—on a regular basis, to allow for early identification of risks.

While agile development started in distributed systems development, it is now also being adopted in the System z development community. This has led to new challenges for developers on System z as mainframe usage models are not designed for on-demand availability for test resources. This is due both to constrained availability and cost, and to the limitations of existing tools that were not designed to support these new

development methodologies. Continuous integration requires that developers integrate their work as they complete it or at least once a day. This becomes an issue on the System z platform as the mainframe may not be available when the developers need it. This is an issue not only for systems where the development or integration platform is z/OS, but also for any system where the solution under development needs to integrate with a component or service running on System z.

Practices of continuous integration

Martin Fowler, a signatory of what is known as the "Agile Manifesto" and a thought leader in the development of continuous integration processes, has broken down the concept into 10 practices. In this section we explain these practices and explore how they are impacted when System z development is part of the development environment:

1. Maintain a single-source repository

Whether in managing code or any file, it is critical to use version management tools to manage the source base that allows multi-user access and streaming, or branching and merging, and that allows multiple developers in distributed locations to work on the same set of files. With any multi-platform development effort—and especially with System z—using a common, cross-platform, single-source repository becomes even more important. If such a repository is not implemented across platforms, any platform left isolated (System z, for example) would not be able to participate in continuous integration practices. Integration with any work conducted on the isolated platform would become an after-effort, waterfall-style integration.

This transition to a modern source-code repository represents a significant change for mainframe development teams that have been using the same capability for years. However, a single source-code management (SCM) tool is critical to allow the management of all artifacts, help break down the silos and remove a key bottleneck.

2. Automate the build

Automating the build is what makes continuous integration continuous. For System z builds, automation can become a challenge—as the availability of the System z environment and the cost of accessing it can both become issues. Availability certainly becomes an issue during production and business operation hours. An advantage for z/OS, however, is that the build traditionally is already completely automated. Additionally, it should be possible to coordinate the build across multiple platforms, when required.

3. Make your build self-testing

Just as builds need to be automated, so does the testing. The goal of continuous integration is not only to integrate the work of teams, it is also to see if the application or system being built is functioning and performing as expected. This requires that a suite of automated test scripts be built for unit-test level and for the component and application level. In true continuous integration, developers should be able to start an integration build by kicking off the right test suite when they commit the code. This process requires that the build scripts include the capability to build the software if needed, provision the test server, provision the test environment, deploy the built software to the test server, set up the test data, and run the right test scripts.

All of this can be a particular challenge for z/OS development, but it must be addressed. The requirement to have the environments to do the build, deploy it, and do the automated testing at any time helps improve the quality of the final code. This requires availability of system resources, the willingness to run large numbers of automated tests on a regular basis and the development of the automated tests.

4. Ensure that everyone commits to the mainline every day

The goal of having every developer, across all components and all development environments, commit their code to the mainline of their development streams every day is to help ensure that integrations remain as simple as possible. For z/OS development today many users work independently on their code changes until the final audit, which is when they realize their work is impacted by the work of other developers. This can lead to delays in releasing functions or to last-minute changes that have not been properly tested being deployed into production. Regular integration of code can help ensure that these dependencies are identified sooner so the development team can handle them in a timely manner and without time constraints.

5. Ensure that every commit builds the mainline on an integration machine

This is a second part of Practice 4. Making sure that every commit is built and that automated regression tests are run can help ensure that problems are found and resolved earlier in the development cycle.

6. Keep the build fast

Virtually nothing impedes continuous integration more than a build that takes extremely long to run. z/OS builds are generally fast due to the standard practice of building only changed files. However, z/OS builds do need to be coordinated with distributed builds, and scheduling the appropriate time when z/OS resources are available can be an issue.

7. Test in a clone of the production environment

Testing in an environment that does not accurately represent the production system leaves a lot of risk in the system. The goal of this Practice 7, then, is to test in a clone of the production environment. It is not always possible, however, to create a clone of an entire System z or non-System z environment just for testing. Even harder is to create a clone environment with other workloads running on it.

Instead, what this practice requires is the creation of what is known as a “production-like” environment. In terms of specifications, this environment should be as close to the production environment as possible. It also should be subject to proper test data management. A test environment should not contain production data, as in many cases, that data needs to be masked. Proper test data management can also reduce the size and complexity of the test environment.

A complex system with multiple components—both pre-existing (such as other services and applications) and new components being developed—also creates challenges. All the components, services and systems that applications need in order to access and interact with each other may not be available for running tests. This may occur for multiple reasons—the component, service or system may not have been built yet, it may have been built but available only as a production system that cannot be tested with non-production data, or it may have a cost associated with its use. For systems hosted on System z, cost can become a major issue.

8. Make it easy for anyone to get the latest executable

Anyone associated with the project should have access to what is built and should be given a way to interact with it. This allows validation of what is being built against what was expected.

9. Make sure everyone can see what is happening

This is a communication- and collaboration-related best practice, rather than one related to continuous integration. However, its importance to teams practicing continuous integration cannot be discounted. Visibility to the progress of continuous-integration builds via a central portal or dashboard(s) can provide information across all practitioners. This can boost morale and help build the sense of working as a common team with a common goal. If challenges occur, visibility can provide the impetus for people to step in and help other practitioners or teams. Visibility via a common team portal is especially important for teams that are not collocated—but it is also key for collocated teams and for cross-platform teams that work on different components of a project.

10. Automate deployment

Continuous integration naturally leads to the concept and practice of *continuous delivery*—the process of automating the deployment of software to test, system test, staging and production environments.

Automated deployments are common in z/OS environments because SCM systems generally include build and deployment. However, most projects do not have enough z/OS resources for each team to deploy into a test environment at all times. Deployments also need to be coordinated with the distributed side of the infrastructure, which can present a challenge due to the lack of common tooling. An effective practice to meet these needs is that of *continuous release and deployment*.

Continuous release and deployment

The continuous release and deployment capabilities required to create a delivery pipeline are shown in Figure 3—with the core capability that automates the delivery pipeline being continuous delivery. The concept of continuous delivery stems from the need for rapid deployment that is driven by continuous integration. As continuous integration produces builds at a steady pace, these builds need to be rapidly progressed to other environments in the delivery pipeline. Builds need to be deployed to the test environment to perform tests, to the integration environment for integration builds and integration testing, and so on, all the way to production. Continuous delivery facilitates deployment of applications from one environment to the next, as and when deployment is needed.

Continuous delivery is a core part of continuous release and deployment that is defined as the ability to regularly deliver the application being developed to different environments in the delivery pipeline—for example development, quality assurance, integration test, user acceptance test, production or others (names may vary). Its goal is validation and potential release to customers in a repeatable, reliable, automated manner. Continuous delivery, however, is not as simple as just moving files around. It requires orchestrating the deployments of code, content, applications, middleware and environment configurations, and process changes.

Two key points are important to note with regard to continuous delivery:

- It does not mean deployment of every change out to production, a process commonly known as continuous deployment. Continuous delivery, instead, is not a process but rather a capability—to deploy to any environment, at any time, as needed.
- It does not always mean deploying a complete application. What is deployed may be the full application, one or many application components, application content, application or middleware configuration changes, or the environment to which the application is being deployed. Or it may be any combination of these.

Two of the 10 practices of continuous integration form the link to and the necessity for continuous delivery:

- Test in a clone of the production environment
- Automate deployment

While “test in a clone of the production environment” may be a testing practice, it also requires continuous delivery capabilities to deliver the new build to the clone test environment. This delivery may require provisioning the test environment and/or any virtualized instances of services and applications, as well as positioning the relevant test data in addition to the actual deployment of the application to the right test environment.

Practice 10, "automate deployment," is the core practice of continuous delivery—it is not possible to achieve continuous delivery without automation of the deployment process. Whether the goal is to deploy the complete application or only one component or configuration change, continuous delivery requires having tools and processes in place to deploy, as and when needed, to any environment in the delivery pipeline.

Practicing continuous delivery also tests the deployment process itself. It is not unusual for organizations to suffer severe issues when deploying an application to production. However, it is possible to uncover these issues early in the delivery lifecycle by automating the deployment process and validating it by deploying to production-like environments in pre-production.

Continuous delivery to System z

For System z applications, there are two common mechanisms or paths to continuous delivery:

- For organizations with a mature set of deployment tools and practices in place, deployment to the target logical partition (LPAR) may be carried out leveraging a legacy configuration management tool. While limited in their ability to deliver the full capability of continuous delivery, these tools can automate deployment to z/OS systems.
- Alternatively, an organization can utilize a specialized deployment automation tool that has full support for multi-platform deployments including z/OS. The IBM UrbanCode™ Deploy tool, for example, has a z/OS agent that can install natively on the target LPAR(s) to enable continuous delivery.

Organizations looking to limit utilization and maximize availability of System z test environments can utilize IBM tooling that allows for non-production instances of z/OS to run on distributed systems for development and testing. This technology, known as IBM Rational® Development and Test Environment for System z, is typically part of the delivery pipeline, providing non-production environments such as development or quality assurance. In such a scenario, the continuous delivery process would deliver the application to these pipeline environments and eventually to production on System z. IBM UrbanCode Deploy is a tool for automating application deployments that supports this configuration of environments.

Continuous testing

Continuous testing is the capability for testing the application, the environment and the delivery process at every stage of the delivery pipeline for the application being delivered. The items tested and the kinds of tests conducted can change depending on the stage of the delivery lifecycle.

Continuous testing is achieved by testing all aspects of the application and environment, including:

- Unit testing
- Functional testing
- Performance testing
- Integration testing
- System integration testing
- Security testing
- User acceptance testing

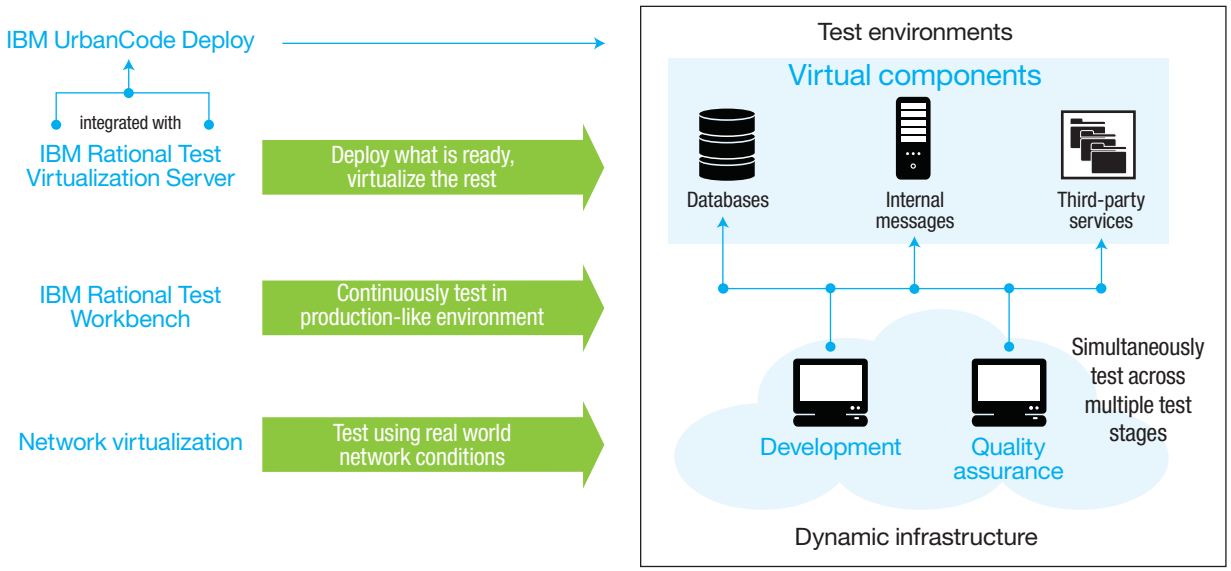

Figure 5. Deploy what is ready, virtualize the rest

Deploy what is ready, virtualize the rest

In continuous testing, the biggest challenge is that some of the applications, services and data sources that are required to perform some tests may not be available. Alternatively, even if they are available, the cost associated with using them may prohibit running tests on an ongoing basis. Furthermore, the costs of maintaining large test environments to serve all teams developing multiple applications in parallel also can be high.

The solution is to introduce the practice known as *test virtualization*. This practice replaces actual applications, services and data sources that the application must communicate and interact with during the test with virtual "stubs." These virtual instances make it possible to test applications for functionality, integration and performance without making the whole ecosystem available. This virtualization can be utilized to perform the myriad types of testing listed in the previous section. The IBM solution for test virtualization is IBM Rational Test Virtualization Server, which has the ability to virtualize multiple technologies, including IBM WebSphere® MQ and IBM CICS® applications on System z.

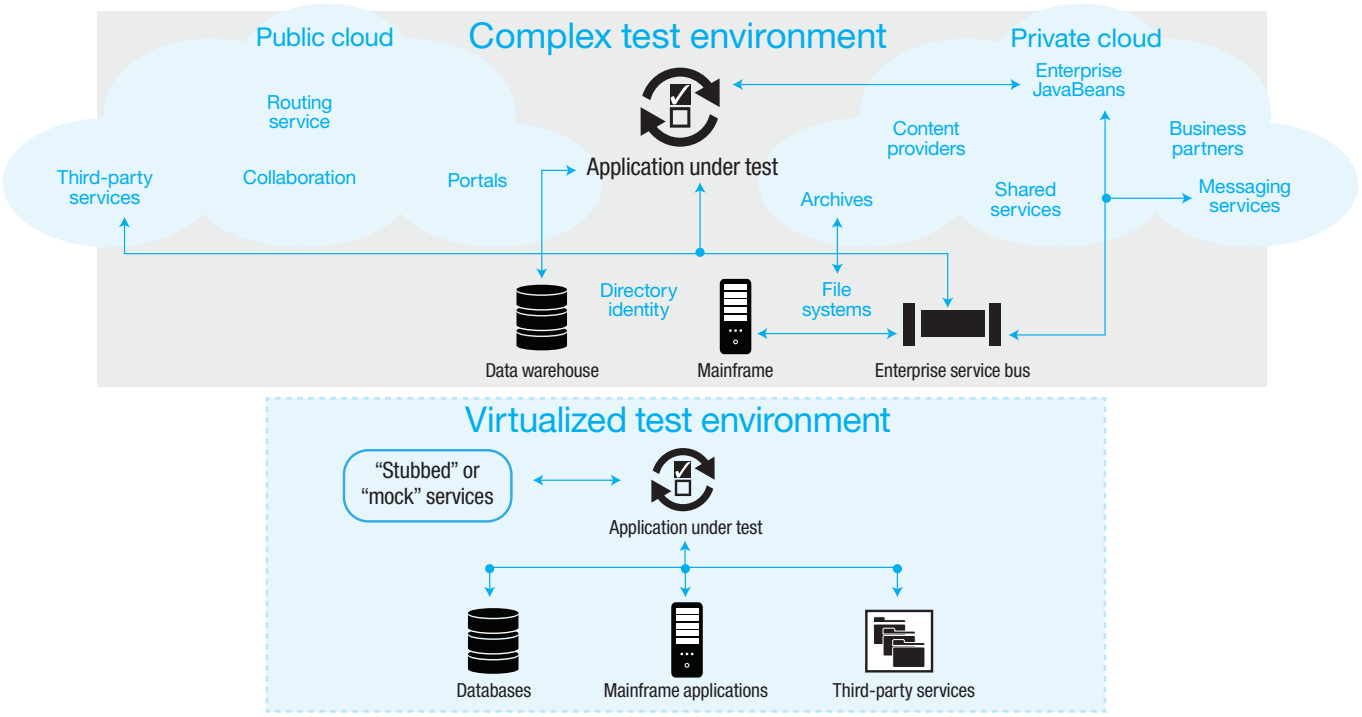

Figure 6. Test virtualization for continuous testing

Using IBM Rational Development and Test Environments for System z

Testing applications—including testing for z/OS—often encounters a lack of available resources. Generally a limited number of development and test LPARs must be shared across teams. In addition, many organizations carefully control the MIPS utilization of their test teams in order to develop capacity for production.

To address both these issues, IBM Rational Development and Test Environments for System z provides a real z/OS environment running on an Intel-based Linux system. This environment includes a z/OS hardware emulation that enables running the true z/OS platform with middleware such as CICS, IBM Information Management System (IMS™), IBM DB2® and WebSphere, allowing greater flexibility for testing. Leveraging Rational Development and Test Environments for System z in conjunction with Rational Test Virtualization Server allows organizations to create complete end-to-end testing environments that include z/OS without utilizing the actual System z environments. This frees the System z mainframes for production use.

Conclusion

Enterprises today are deploying applications that are truly cross-platform—from mobile to mainframe. The DevOps approach to development uses "lean" principles to create an efficient and effective delivery pipeline that allows applications to be developed, tested and delivered as it helps raise the quality, increase the speed and reduce the costs of development.

Adopting a DevOps approach requires organizations to change in areas spanning people, processes and tools. To make these changes possible, the organization must adopt three core capabilities—*continuous integration*, *continuous delivery* and *continuous testing*. These capabilities are platform agnostic, but in order to adopt them in a multi-platform ecosystem, it is essential to adopt an integrated tool set that supports all the required platforms.

The mainframe running z/OS has been an outlier for most of the DevOps tooling available in the market. The tools from IBM Rational—specifically, IBM UrbanCode Deploy, IBM Rational Development and Test Environments for System z and IBM Rational Test Virtualization Server—support the System z platform. With their support for true multi-platform development that includes z/OS and distributed platforms, these solutions provide true capabilities to scale DevOps at the enterprise level—using one tool chain to support DevOps for applications, teams and platforms across the organization.

For more information

To learn more about IBM Rational solutions and DevOps, please contact your IBM representative or IBM Business Partner, or access the following resources:

IBM video:
DevOps for System z
https://www.youtube.com/watch?v=3ALmTUXhfk0

IBM article:
Multiplatform application deployment with UrbanCode Deploy
ibm.com/developerworks/rational/library/multi-platform-application-deployment-urbancode-deploy/index.html

IBM e-book:
DevOps for Dummies
https://www14.software.ibm.com/webapp/iwm/web/signup.do?source=swg-rtl-sd-wp&S_PKG=ov18162

IBM video:
Understanding DevOps
http://www.youtube.com/watch?v=HpZBnc07q9o

IBM web page:
"Enterprise modernization"
http://www-03.ibm.com/software/products/en/category/SWY00

IBM blog:
DevOps for Enterprise Systems News
http://devopsenterprisesystems.wordpress.com/

Additionally, IBM Global Financing can help you acquire the software capabilities that your business needs in the most cost-effective and strategic way possible. We'll partner with credit-qualified clients to customize a financing solution to suit your business and development goals, enable effective cash management, and improve your total cost of ownership. Fund your critical IT investment and propel your business forward with IBM Global Financing. For more information, visit: **ibm.com**/financing

About the authors

Rosalind Radcliffe
Distinguished Engineer, Rational Chief Architect for CLM and DevOps

Sanjeev Sharma
Executive IT Specialist, Rational Software Specialty Architect

IBM Software Group

IBM Corporation
Software Group
Route 100
Somers, NY 10589

Produced in the United States of America
July 2014

[1] Martin Fowler, "Continuous Integration," May 01, 2006. http://martinfowler.com/articles/continuousIntegration.html

[2] Sanjeev Sharma, *DevOps for Dummies*, John Wiley & Sons, Inc., 2014. https://www14.software.ibm.com/webapp/iwm/web/signup.do?source=swg-rtl-sd-wp&S_PKG=ov18162

Please Recycle

RAW14362-USEN-01