Testing Cloud Hosted Solution in Production

Arvind Srinivasa Babu (arvind.srinivasa.babu@intel.com)
Sajeed Mayabba Kunhi (sajeed.mayabba.kunhi@intel.com)
Joshy A. Jose (joshy.a.jose@intel.com)

Abstract

The evolution of software from a simple application running in an isolated system to a multi-node solution in a network and now to a geographically distributed cloud solution with hundreds or thousands of nodes, has brought about new testing challenges. Testing strategies which work well in one model are not always applicable to others.

While many of the tests for a cloud-hosted solution can be done by replicating a subset of the actual production environment, some tests need a production-like environment to be effective. This could be challenging if the hosted solution is complex or large. Though one might successfully replicate the production environment for testing, it is very difficult or too expensive to replicate end user behaviour, as the sample set of users in cloud could be very large.

One option to overcome the given problem is to perform the testing in a live production environment. This paper attempts to bring forth the challenges and possible suggestions for testing cloud hosted solutions in a live production environment.

Biography

Sajeed Kunhi is a senior engineering manager with Intel with nearly twenty years of software engineering experience. He is highly passionate about process automation, product quality and DevOps principals. His expertise in software development, technology domain and managerial experience has helped in building optimized software development lifecycle.

Joshy Jose is a Software Architect at Intel, with more than sixteen years of experience designing and implementing software. He has been instrumental in improving development and testing processes incrementally over the years. His areas of interest span security management, device drivers, system programming and mobile application development.

Arvind Srinivasa Babu is a Technical Lead at Intel, with more than six years of experience designing and implementing software. He has been performing the role of a scrum master and strives for continuous improvement in testing and development. His areas of interest span over network, user interface interaction, system programming and mobile application development.

Copies may not be made or distributed for commercial use

1 Introduction

When we look at validating the quality of software, the term “testing” covers a very broad domain. There are various stages of testing and strategies which are formulated and employed before a solution goes to a world wide release. A product is made up of many modules and features, which can serve as an isolated standalone solution or scale onto multiple machines that are geographically distributed.

When a solution is cloud based, there are several phases of testing and deployment strategies which ensure a quality release, but very often the quality comes into question when a production environment causes an issue in the solution that has slipped through various testing phases. Issues can be attributed to infrastructure differences, load on server and performance at various loads. The goal of every phase in testing is to qualify the solution from the smallest testable unit within the software through validation of end-to-end solution. Each phase has an associated cost attached to it and as the test strategy moves through these different phases, a complexity arises where factors in production cannot be replicated in a test environment. These factors are unpredictable, and cannot be assessed during design of the software. Testing in a production environment would allow such data to be collected and be refactored into design.

In this paper, we will refer to existing strategies and phases of testing as *“Traditional testing.”* It includes unit testing, integration testing, system testing, automation, mocking etc. But not all the tests that are covered in a test strategy, would prepare a product/solution from being ready for deployment in a cloud production environment. Most cloud solutions have a commitment to have an uptime of 99.99% or higher. If a solution fails to deploy through traditional testing and deployment strategies, we would be forced to break an agile cycle to either fix the issues or rollback the production environment to its last known state. This adds to the cost in maintaining a production environment with latest version of the solution. There is a significant difference between quality inferred from traditional test environments and production environments. We will explore such differences and how to overcome these problems in upcoming sections

First we will give a brief overview on the different teams and their roles in delivering cloud solutions for deployment in a traditional product lifecycle and the challenges involved in it. In section 3 we will discuss the differences between testing in traditional test environments and production environments and why it is essential to test in production. We will also suggest essential best practices and how they influence a solution to achieve a successful production testing strategy. In section 4 we will detail how production testing can be achieved. In section 5 we will discuss how telemetry can be leveraged to quantify and collect metrics and use them to build better solutions. In section 6 we will discuss risks involved with production testing and how to mitigate them.

2 Traditional Product Life Cycle for Cloud Deployment

2.1 Traditional validation of cloud solutions

Traditional testing as mentioned earlier is a collective term used for various testing methodologies, phases and strategies that are put in place for a deployment of a cloud solution. These methodologies include unit testing, feature testing, integration testing, system testing, acceptance tests, user acceptance test, feature verification tests, smoke tests and automation. These validation methodologies have their own test environment which is an approximation of known production environment parameters. Quality inferred from the results of these methodologies are an approximation of how well the solution might work in the actual production environment. These phases of validation or methodologies are targeted to infer quality with a particular scope and are often disconnected with each other. Once the solution is put through this traditional cycle of testing, it is qualified as a high quality release build which is taken up for deployment.

A cloud-hosted solution could be developed by a single engineering team or multiple engineering teams depending on the complexity of the solution that is offered. Each team that takes ownership of one or more components within the solution may employ different validation strategies that works well within the team’s delivery process. When such complex solutions are integrated during this traditional validation phase, a lot of time is invested to ensure that the integration between different components do not affect the overall quality of the solution.

Copies may not be made or distributed for commercial use

We will briefly provide insight on how one such complex solution is integrated from different teams, how the solution validation is carried out before being deployed into production and the challenges faced with this approach.

2.2 Roles involved in traditional cloud validation

In a typical cloud deployment of a complex solution, there are three major roles or teams tasked with taking the solution from inception to deployment.

- Engineering
- System Validation
- IT Operations

A component within a complex solution or the solution itself is designed, developed and tested by an **Engineering** team. This team will be responsible for maintaining the components codebase, bug fixes and validation of the feature. They will reach out to other components' engineering teams and develop interdependent relationships. Each component within the cloud solution will have its own measure of quality validation process as defined by the Engineering team. If the quality confidence is accepted by stakeholders, the component is released to a System Validation team. The validation of the component by the Engineering team will be performed in a test environment that suits the component and that environment may not scale up to a production environment.

A **System Validation** team is responsible for bringing different releases of components from Engineering teams together to form the solution. This team validates whether components are compatible with each other and are conforming to cloud performance standards. Any issues within this validation phase will require back and forth communication with the respective component's Engineering team. Once the System Validation team approves the various components, the solution is rolled up for deployment. The test environment employed here will be a mock-up or scaled down version of the actual production environment.

A typical **IT Operations** team manages the cloud infrastructure and deployments. This team will pick up the deployment packages validated by the System Validation team and starts the deployment. A typical deployment process will include ensuring the database connections are closed, redirecting users to a downtime notification page or zones within the cloud infrastructure where deployment is scheduled for a later time, all backups are performed , etc. and then proceed to deploy the new version. Depending on the maturity of the solution and deployment methods that the solution will support, downtime will be incurred. These are scheduled downtimes, and the timespan of such downtimes will vary depending on how complex the deployment is and how many issues arise during deployment.

2.3 Challenges

When the **IT Operations** team starts deployment, they ensure all sufficient precautions like backup, downtime notifications etc. are performed. If the operations team faces issues with the deployment, the typical response would be to either rollback the production environment to its last backed-up state or a special team (possibly including members of the Engineering, System Validation and IT Operations team) is formed that fixes issues as they surface and ensures the deployment goes through, possibly incurring additional downtime due to additional validation cycles on the fixes provided.

Engineering and System Validation teams have their own delivery process, which can ensure a deliverable is on a daily, weekly, bi-weekly or monthly basis. Each deliverable can bring in new features or fixes addressed for issues discovered from previous deployment, but if there is no mature deployment pipeline, the identified issues that are existing within production cannot be addressed in a timely manner. If the different components employ different processes, delivery might differ on the complexity of each component and System Validation will incur a lot of time and cost in validation of different components at different times to deliver a stable deployable release.

These challenges may not be restricted to this traditional validation methodology but may span different solutions. There could be other strategies employed before a cloud hosted solution is deployed into production and it depends on the stability and complexity of the solution itself.

3 Production Testing

3.1 What is Production Testing?

Production testing is a validation phase that can be adopted for a cloud based solution, where certain types of tests are executed and collection of quality related information happens in the actual production environment. These tests will be executed in a planned and systematic manner as part of deployment and post-deployment which will help minimize failure rates and optimal utilization of cloud resources. Production testing is not intended to replace traditional validation strategies but provide an additional drive towards a higher quality and reliable release.

3.2 Why add Production Testing?

Solutions that plan to be deployed on cloud or are already being deployed to cloud, need to have a mechanism that allows the different teams to rapidly respond to issues that arise from within the production environment. Issues can arise from within a solution's component or within the production environment which is very difficult to simulate in traditional validation methodologies. Quality inferred from production environments intend to carry more weight than quality information generated from a traditional test environment. A test environment is traditionally modelled based on an approximation of known production environment parameters and are generally not optimal for quality inference with scalability tests or load tests. The quality of the solution inferred is directly proportional to how close traditional test environments are modelled to production environments.

3.2.1 Differences between traditional testing and production environments

There are many aspects that vary between traditional test environments and production environments, a few are discussed in the following sub-sections.

3.2.1.1 Infrastructure

Infrastructure can refer to memory distribution, hardware differences, network topology, etc. The cost involved in replicating a production environment is directly dependent on the complexity of the infrastructure of the production environment. This cost factor affects how traditional environments are modelled and cost incurring infrastructure items like hardware, memory or network topology is scaled down or overlooked.

Network topology and its role in environments can be a crucial differentiator between production and local test environments. Test environments within a traditional lifecycle are often setup within local networks that usually have a very high throughput. A production environment can employ a complex network topology and can have different network throughput between different components of the hosted solution. A latency ping between a database server and web application in a cloud solution can be very high in a production environment compared with a local test environment.

There are multiple infrastructure differences that can be cited between a traditional test environment and a production environment, but the quality inferred from these environments have significant differences in terms on performance due to the varying hardware, faster networks etc.

3.2.1.2 Quality data

Cloud-based solutions are designed and validated for deployment in production environments based on data regarding quality collected across various testing phases. A traditional testing methodology gives only partial information on how well the solution is performing and whether the solution is working well within the parameters of a local test environment. For example, unit testing only provides quality data as to how much code functional coverage and conditional coverage is achieved, it does not give a measure of how well different modules of the solutions are integrated with each other. As the solution goes through various phases of testing, all the data regarding quality is a measure of how good the solution works in an approximate production-like environment.

Engineering teams do not actively collect quality information from production environments and even if little quality data is collected, it is overlooked due to the round trip involved from Engineering to

Copies may not be made or distributed for commercial use

production. The data collected here would be invaluable to the product engineering team. The quality data obtain from live production environment will better position the solution to tackle real issues proactively.

3.2.1.3 Load simulation

A part of traditional testing is to load the solution with number of requests to evaluate the performance under load. These requests originate from a single network and do not give an accurate reflection of the actual performance. A production environment will actually provide the performance data accurately based on how users have loaded it.

3.3 Setting up for Production Testing

A solution that is validated in a traditional testing methodology has various gaps when it moves from one phase to another. The following are some best practices that influence a cloud-hosted solution to be "*Production Test Ready.*"

3.3.1 Continuous Integration

This is an **Engineering** centric strategy for how developers integrate code into a mainline branch continuously. This is partially implemented in solutions that use branching mechanisms or where code is not checked in for a few days until it is logically complete. Such practices lead to a problems when a large block of code developed across days or weeks gets integrated into the mainline suddenly thereby introducing lower code coverage, higher validation load, development process overheads and unfinished stories in agile processes. Following Continuous Integration within the team's delivery process can ensure that tasks are not planned for more than few hours and that the code developed gets checked into the mainline several times during the day. Though this is not directly related to testing in a production environment, it is an essential process that will help in a Continuous Delivery/Deployment pipeline. Advantages of following Continuous Integration is that at any given integration, only a small manageable, testable amount of change will merge into the main codebase that can be validated with minimal effort.

3.3.2 Continuous Delivery

Continuous Delivery follows the same idea of discrete, well defined work items as Continuous Integration. Traditional testing methodologies will have some level of automation that are either triggered manually or on a semi-automatic basis depending on when validation is required. Production testing would require a completed automated system by which small changes can be validated. Continuous Deployment is a continuous process, new code might require new automation tests written, and invalid tests can be removed. Planning for such activities within a solution's process cycle is critical for a successful Continuous Delivery pipeline.

3.3.3 Continuous Deployment

Continuous Deployment and Continuous Delivery are significantly different depending on the complexity of the solution. The team can either stop at Continuous Delivery or also employ a Continuous Deployment. The only difference between the two is that Continuous Delivery aims at producing high quality release that can be deployed manually anytime and is a required minimum when a solution wants to be ready for production testing. Continuous Deployment performs the deployment automatically once the solution/product has reached the end of Continuous Delivery process. This is a significant breakthrough for a solution and is extremely challenging to achieve, but when implemented successfully it will have a significant effect in maintaining minimal or zero downtime. We will explore different deployment methods when we discuss on setting up the production testing pipeline.

3.3.4 Features

Typically solutions offer various technologies as features or feature sets but a common practice within the Engineering process is that these features or feature-sets are targeted to address big functionality changes to completion that span multiple iterations. Such features carry a high risk of introducing new issues in production environments since a big change has skipped the **Continuous Integration** pipeline. When a solution wants to go to cloud, it must ensure that any and all features must be scoped

Copies may not be made or distributed for commercial use

into minute functionality that can be delivered in few hours. Existing feature enhancements should be broken down into something that can be delivered in a short span.

Designing features that address a small part of functionality can go a long way in minimizing issues within the production environment, allowing teams to respond to issues in a systematic manner. This also allows the team to track changes effectively.

3.3.5 Feature Flags

A feature flag or toggle is a valuable product design mechanism through which an entire feature can be disabled or enabled. These feature flags can be implemented at various levels, as part of UI features, deployment process and as part of the solution to target specific users. Feature flags can be used as part of production tests to measure a feature's performance in production and decisions to disable the feature can be taken either at end of deployment or when the solution is actively serving users.

Delays and disruptions in deployment strategies have a huge impact in production environments uptime. Feature flags reduce the risk of increased downtime by disabling features that perform poorly in the entire deployment process. Features can be released based on these toggles, but these toggles need to be removed once a feature is enabled for all users to avoid overhead of managing feature toggles that accumulate over time.

3.3.6 Incorporating Telemetry across the pipeline

3.3.6.1 Why Telemetry is essential for production testing?

Telemetry refers to the automated process of collecting information within a sourced environment. The information collected can be anything that the module has access to, and it is essential that telemetry needs to be built across the production testing pipeline. Data collected and collated can provide better data regarding the quality of the solution from Continuous Integration all the way until the solution is deployed and active. Analysis of this data and incorporating the feedback within the solution will help the solution be proactively ready to tackle production issues.

3.3.6.2 Privacy and Legality

There are laws and regulations in many countries that determine and guard what personally identifiable information is collected about their citizens. It requires clarity on the type of data collected and how the data is stored and utilized. Solutions should carefully consider collection of information which must conform to these rules and regulations. We will explore more on how data can be anonymized to respect privacy in upcoming sections.

4 Production Testing Pipeline

4.1 Overview

We explored the advantages of implementing different steps within a solution in the previous section. These concepts are better achievable by implementing a pipeline comprising of a well-defined process from Engineering all the way until the solution is deployed by IT Operations. Figure 1 depicts a suggested flow of how a cloud-hosted solution can be production tested.

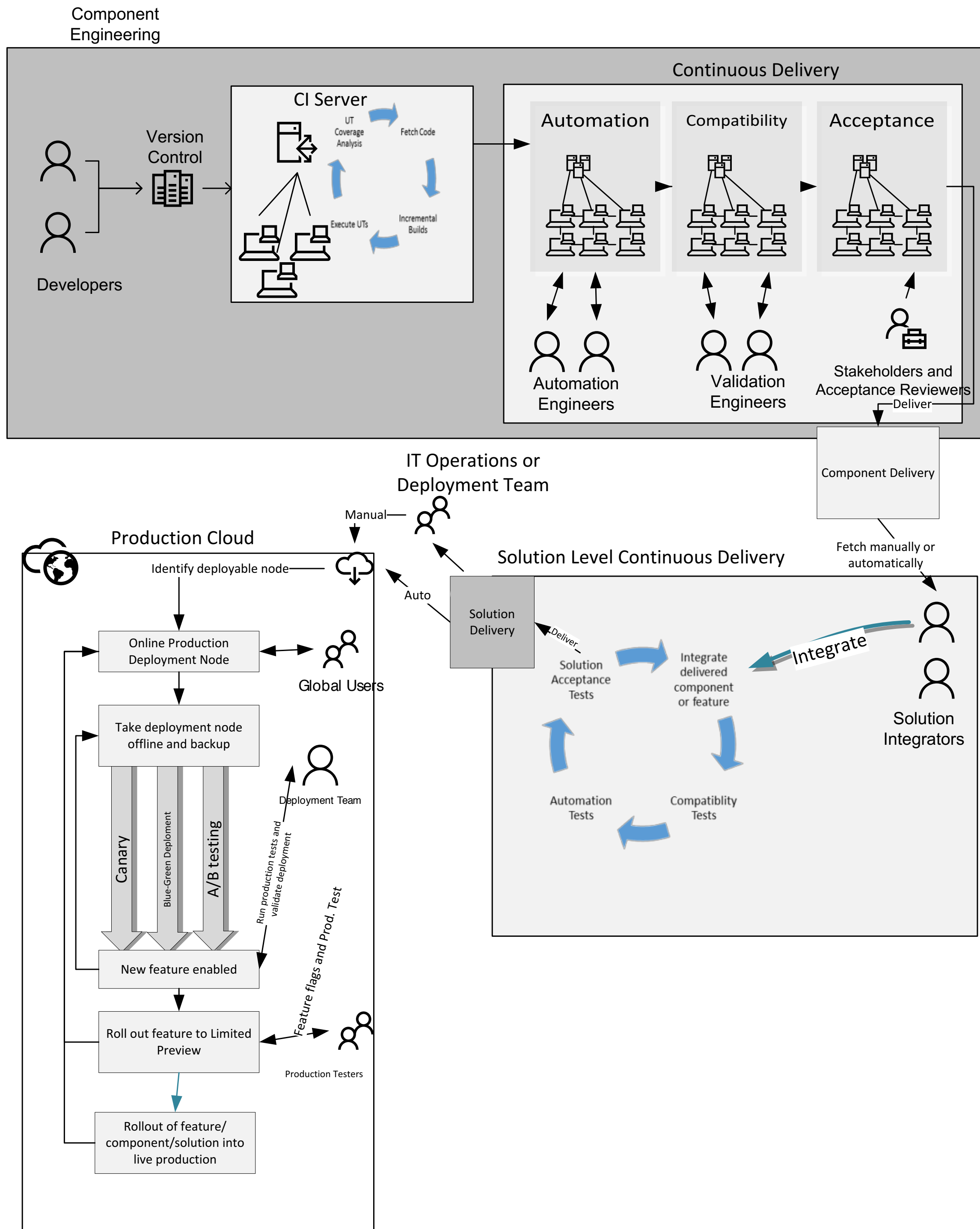

Figure 1 Production Testing Pipeline, from component engineering to deployment and production testing

4.2 Continuous Integration

Continuous Integration (CI), as mentioned earlier, is an Engineering-centric strategy to maintain a shared codebase among different developers. Features should be broken into testable units that can be completed in a few hours and committed in the mainline codebase continuously. Automated Unit Tests should be written and executed in this phase. Code reviews can be controlled before the code is committed with an approval process as mandatory requirement for code-commit. CI servers can be configured to fail this phase if sufficient coverage is not met or number of tests falls below the acceptance criteria. If sufficient coverage is met in the Continuous Integration phase, it ensures that the code which was integrated during several times during the course of the day has not broken an existing test and new code has sufficient coverage. There is no set metric for code coverage in this phase and is entirely dependent on the design architecture and complexity of the code that was written and what can be achieved through unit tests.

The Continuous Integration pipeline need not be restricted to just individual production deployable modules or components but also to automation suites that support qualifying solutions and their components within the Continuous Delivery pipeline. Figure 2 represents a typical Continuous Integration pipeline within a components Engineering team.

Telemetry can be incorporated in this phase to collect coverage reports for the Engineering team to spend cycles in delivering incremental code with acceptable code coverage.

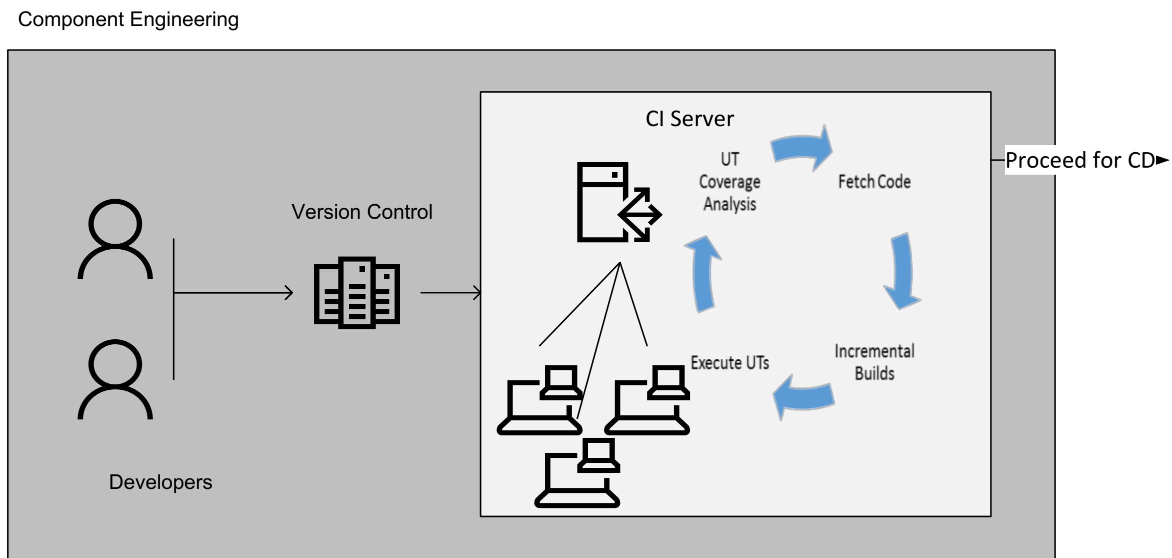

Figure 2 Continuous Integration within a component engineering before moving to component's Continuous Delivery

4.3 Continuous Delivery

Continuous Delivery is an extension to Continuous Integration, which performs automated test runs to meet acceptance criteria for stories. System level acceptance tests are performed to achieve overall compatibility between interdependent components. This process is can be complicated and has a direct correlation with the complexity of the solution which is trying to achieve a continuous delivery pipeline. Large complex solutions brought to cloud may not have sufficient automation coverage and largely rely on black box testing. Such solutions should actively work on increasing automation coverage and devise a process that will ensure that CI delivered solution is tested by end of a sprint is of high quality and ready to ship.

One of the essential factors for Continuous Delivery is how solutions and their components are designed as features. Features, as mentioned earlier, allow different teams to have visibility to the changes made,

and provide feedback on failure as soon as possible. This also allows the feature developed to be validated and readily available to be deployed into production. Application release automation and build automation are some of the type of methodologies that allow and execute various parts of Continuous Deployment pipeline.

Continuous Delivery can be implemented both at a component level (for complex solutions) and at a solution level. For a solution level Continuous Delivery pipeline, the System Validation Team can integrate the different components delivered as part of the component level Continuous Delivery pipeline. A System level automation can be run to ensure compatibility between different components and proceed with a system level automation to validate end-to-end functionality and acceptance tests for the entire solution. The different Continuous Delivery pipelines will provide a high quality deployable package that can be deployed manually or can be deployed automatically using Continuous Deployment strategies.

Figure 3 represents a Continuous Delivery pipeline at both a component level and solution level.

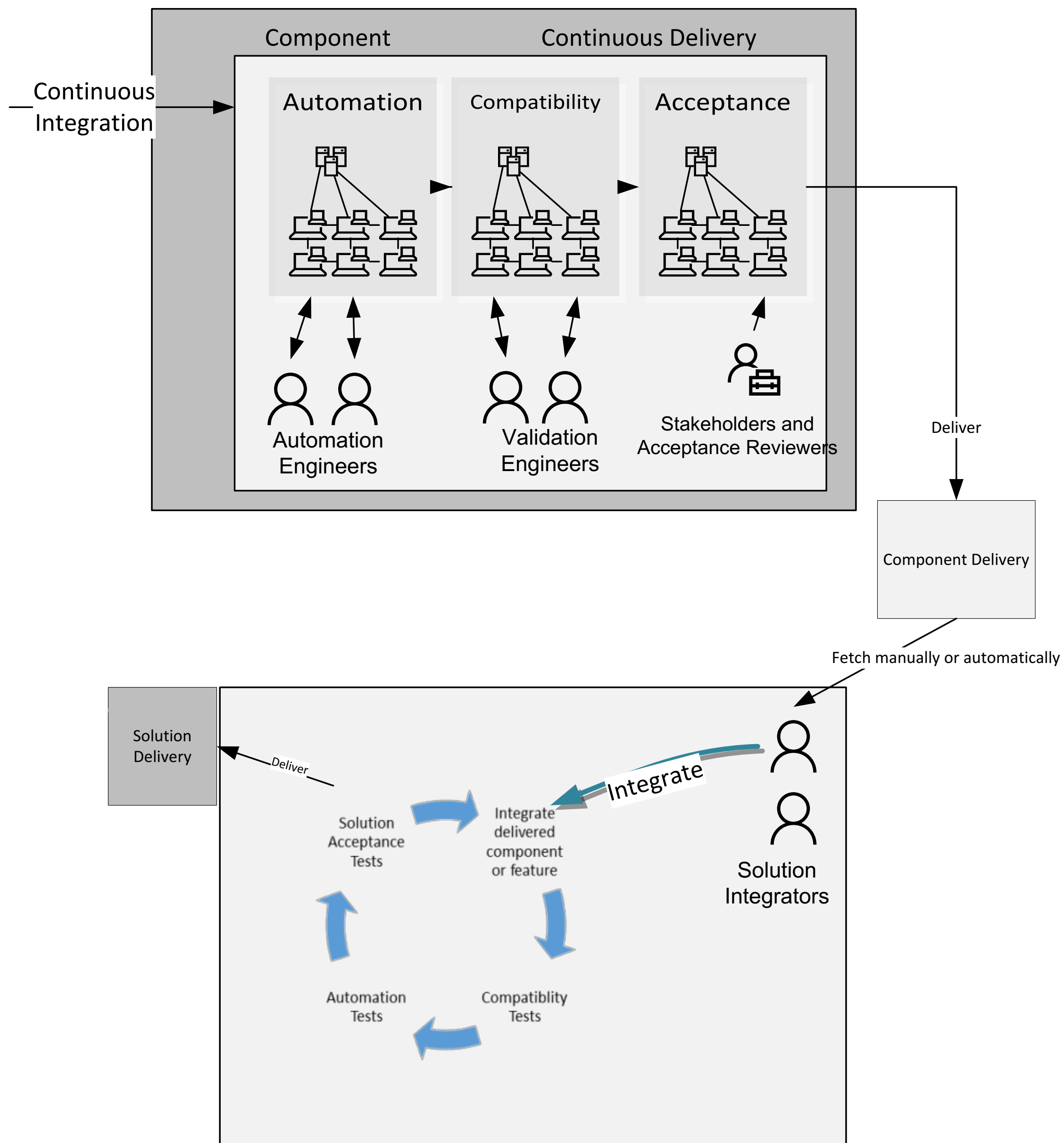

Figure 3 Continuous Delivery at Component Level and at Solution Level

Copies may not be made or distributed for commercial use

4.4 Continuous Deployment

Deployment is started once a solution is delivered as part of the Continuous Delivery. This process is currently manual for most cloud solutions and is done in a scheduled maintenance window. Continuous Deployment is a process that is elusive for most solutions, attributing to huge deliveries in traditional methodologies. When proper production testing pipeline is implemented, deployment can be triggered automatically where features or small incremental changes have gone through various automated tests and can be safely deployed into production.

Figure 4 is a representation of Continuous Deployment within the Production Testing Pipeline

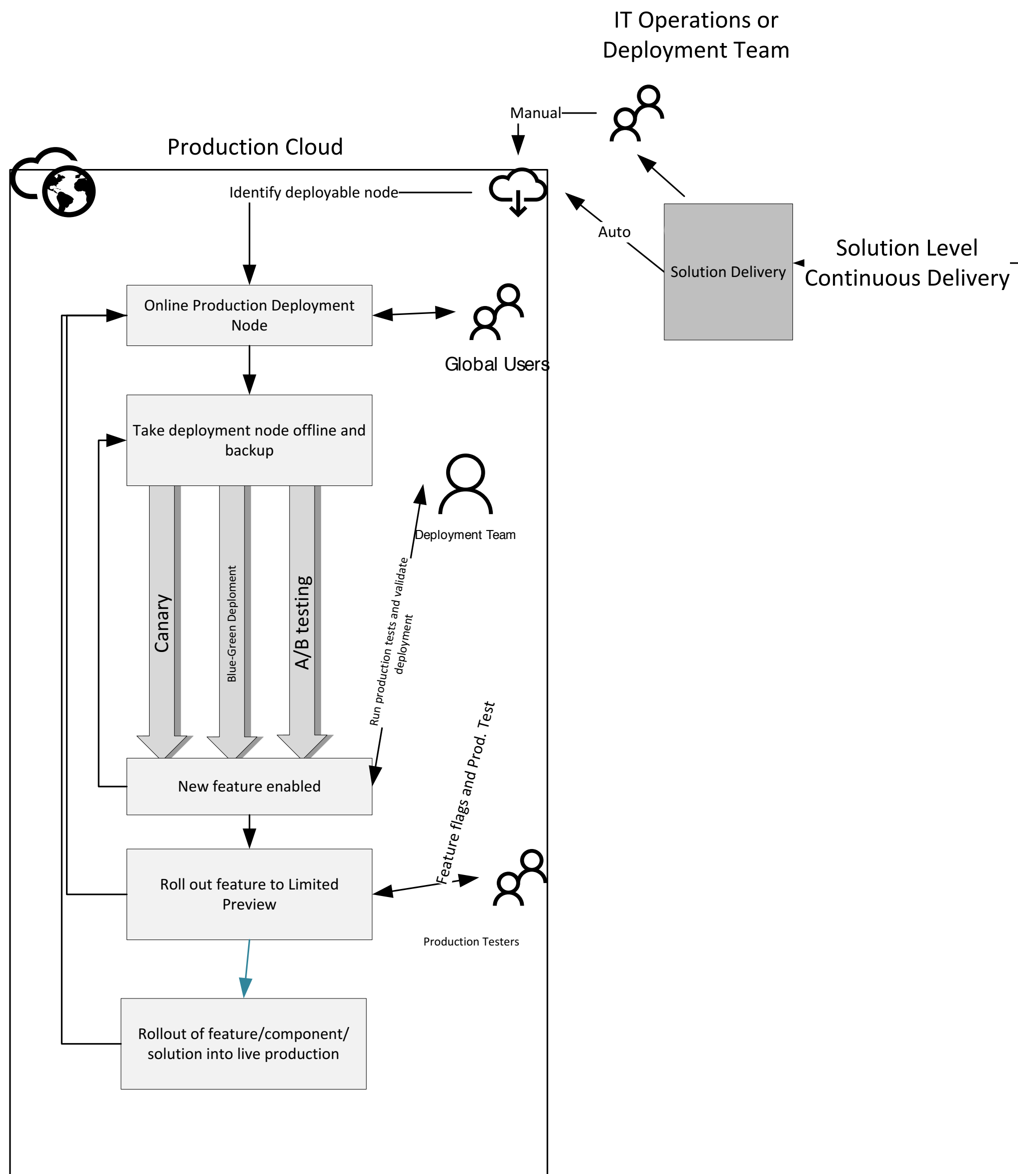

Figure 4 Continuous Deployment takes the solution and uses any of the below mentioned deployment strategies and uses feature flags to control features for production testers.

Next we will discuss strategies that can be employed to deploy the solution into production environment.

4.4.1 Canary Deployment

Canary deployment is done when a new version of the solution is rolled out. It starts with rolling out features on to a subset of the production infrastructure where no users are routed. A few production tests can run here to test whether the environment meets the prerequisite for the solution. Next step

would be to slowly route a small batch from the total set of users into the infrastructure holding the new version. If an issue is encountered with a new feature, feature flags can be used to disable the feature completely without impacting the rollout. If the rollout was successful, the users can be redirected to the updated portion of the infrastructure by a load balancer and other zones within the infrastructure can be updated until the solution is fully deployed in the entire production environment.

4.4.2 Blue-Green Deployment

This kind of deployment is done where multiple copies of the solution are maintained inside the production environment. Select pods within such environment are marked as "Blue" and others as "Green". Blue environments are set to handle live traffic, and the Green environment would be brought offline to perform the deployment. Rapid quick tests can be executed in this window to ensure that the environment will start up properly. Once confidence on the new version is developed, users can be switched or routed to the new version (Green pods) which will go live and the same process will now be deployed onto the Blue environment that will be taken offline. This kind of deployment can be performed where the solution needs to be deployed to indicate version changes. Performing tests in such a stage will help plan future deployments safely with minimal downtime.

4.4.3 A/B Deployment and testing

A/B testing is followed to determine a better visual user experience by providing the same information in different formats within the web page. In production testing, A/B testing can be implemented easily by incorporating both visual designs as features with feature flags based on continuous assessment from production environment across different targeted users. This testing initiative will help design better user experience on the user interface since the quality data is derived from the actual production environment from real users.

4.5 Testing in Production

So far, we discussed on different deployment mechanisms and running production tests within them. These tests are specifically designed to be run on production environment and are a collective effort of Engineering, System Validation and IT Operations team. Teams representing various components should collaborate and plan features. These features have feature flags incorporated to toggle them during production validation among certain targeted users. Based on their performance the feature flags are ensured to be always on and removed and rolled out to the public.

During deployment, features use deployment or build related feature flags to control deployment. Once features are enabled, they are targeted to certain types of users using permission flags. These features will remain in production and rolled out to all users based on their acceptance in production environment. Validation engineers become a subset of targeted users who arrive at rolling out to different release phases. Features can be enabled or disabled based on how the quality information derived from running these types of production tests within the pipeline.

4.6 Release Strategy

One important component in a production testing pipeline is having a proper release strategy. The release strategy involves formulating a release timeline from inception to deployment of every small code block that gets committed within the Continuous Integration pipeline. Withholding solution delivery over a period of time accumulates risk of deployment failures and increased downtime. If a code block is delivered as part of production pipeline, it needs to be deployed with production tests. Controlling the failure of features can be done with planned releases targeting specific users initially before being rolled out to all users.

4.7 Phased Releases

Phased releases are part of release strategy that should be implemented within the production environment so that new features are rolled out to smaller audiences initially and eventually to larger audiences. Examples of phased releases to consider are **Internal Release**, **Limited Preview Release**, and **Global Rollout**. There can be more intermediate release phases according to the requirements of production testing. The advantage of doing phased releases is that in production, features can always

move into next phase independent of when they were delivered. For example, Feature A and Feature B have entered a limited preview phase, after few cycles, Feature B's performance and quality confidence is far superior compared to Feature A, then Feature B can move onto a global rollout even though Feature A might have been delivered first.

4.7.1 Internal Release or Targeted Release

This is a limited release targeted to users within the organization or early adopters. Typically, new features that have passed the various tests in production environment are put to test here to see their performance and functionality. If features have passed through this release, the next step is to roll it out as a limited preview. Feature flags can be used here to allow validation and automation engineers to hook up automation in live production.

4.7.2 Limited Preview Release

A limited preview release phase rolls out the features into a larger audience for example, to external beta adopters. Feedback from the users can be engaged to improve upon the feature. If there are no issues and confidence on quality of the feature is high, it can be rolled out to a much larger audience or the public. Features that have passed internal release can stay in limited preview while new features are rolled out to the public. The whole idea is to measure feedback and improve a solution.

4.7.3 Global Release

A feature that has passed through internal release, and limited preview can now be considered to be released to the public. This is the final release phase where everyone in the system will be able to use the feature. A feature that has achieve this milestone has gone through various rigorous testing phases in traditional test environments as well in production environment with smaller audience. A feature that has been promoted to this milestone indicates that it is of a very high quality.

5 Quantifying Telemetry Data

We will briefly look on how telemetry can be used to quantify metrics and utilize the data for delivering better quality releases for Production Testing.

5.1 Design as a feature with feature flag

One of the features that can be built within the solution is a telemetry feature. Telemetry can raise flags across the industry with concerns of what information is being collected, but the data collected here would help a long way in helping Engineering teams plan better features. Design and integration of this module is of great importance, core modules that do the low level work can integrate with telemetry feature, publish information of their activity which allows telemetry feature to translate into metric sets.

5.2 Quality and Performance metrics

5.2.1 Quantifying a metric

A metric is a standard measure of degree of a property that the solution possesses, e.g. quality metrics, software metrics, infrastructure metrics, etc. Metrics collected from production environments have huge value as they represent the actual data on how well the solution is performing in real time. The metrics derived from traditional testing are different from the metrics derived from production testing as the data collected from production can be viewed as sensitive by customers/clients and would require multiple levels of privacy being built in.

When a cloud-hosted solution is deployed live, key metrics to collect include amount of warnings and errors generated during deployment, downtime, uptime, requests served per time units, database access times, ping time and subnet distance between different components, round trip time between various network communication due to a request, file I/O's, resource consumption.

Metrics are currently collected only when there is an issue or they are mostly ignored. Having this information would go a long way in providing better performing solution in future releases. Integrating

metric analysis within a product lifecycle will go a long way in anticipating production issues and provide better features proactively.

5.2.2 Categorizing Metrics

Grouping metrics is a critical component of telemetry. Metrics need to be grouped according to the anonymized information collected. In each telemetry session, the datasets need to include number of I/O's performed, database access, database query performance, number of requests served and average time between responses. Feature specific metrics can be built into during the development of the feature, which in turn can generate lot of metrics to better anticipate their performance.

The data should be grouped into sub-categories relevant to the solution. For example, network related information like ping time, subnet distance, and round trip time can be collated under network metric. This network metric can be individually collected for various component to component interactions (WebApp to DB and/or WebApp to file system etc.). The design of the data set would vary from solution to solution due to varying components in the design of the cloud hosted solutions.

Reporting all this information should not hamper the performance of the solution, which is why telemetry should be designed as a plug-and-play feature that can be turned on or off at any time.

5.2.3 Anonymizing metrics

Anonymizing metrics in a key strategy to be able to legally collect the data from within cloud solutions. All cloud performance metrics must include an identifier (custom-generated) that would uniquely identify different components for which metrics are calculated.

As pointed out in an earlier example, a ping time between a web solution and its production database can be a simple metric collected by anonymizing the components such as the WebApp and database which should be represented as unique identifiers. This would obfuscate the data where no client information would be included, including IP, DNS names, etc. A similar example can be made for subnet distance where only hop count can be collected between components and information on routers/IP/Subnets on the way need not be collected.

5.3 Utilizing metrics for continuous feedback

A feature's quality is determined on how well it works under different conditions under various scenarios. Through every phased release, a feature will be subjected to different loads, usages and tests. Metrics derived from this production testing phase will provide constant feedback to the engineering team to invest better to anticipate user expectations.

5.4 Designing data for anonymization

With production testing, collection of data is critical for better quality. Design of anonymized datasets would allow customers' information to remain private. These datasets could measure performance between different components which is part of the solution and would grant the merit of legally accessing the information.

Analysis of the anonymized dataset would require a thorough inspection of what are the network issues, performance issues on script executions, throughput of data, request/response speed ratio etc. The datasets which are categorized identify data valuable to engineering teams regarding the production infrastructure, load and performance of the solution without risking the privacy of the customers. Categorized grouping of data allows anonymized data to generate patterns where the solution is failing and addressing issues within the production pipeline.

6 Risks of Production Testing

6.1 Bugs, feature sizing and brand

Bugs are a fact of software life — with production or test environments. Early detection helps improve quality release over release. A bug that arises from production is reflects poorly and can show a gap in

Copies may not be made or distributed for commercial use

the testing strategy. It could be environmental or a product defect. Not every code path can be tested when there is release pressure. Features should be designed to be very incremental. A wrongly sized feature can be an oversight and has potential to break or perform badly. When a feature is marketed beforehand and fails to deploy, it can lead to bad publicity for the solution. Cloud hosted applications require rapid response and code gets checked in throughout the day. A bad release strategy, overzealous marketing, and continuous deployment of breaking features detracts customers and clients which can have lasting impacts on the company brand.

6.2 Deployment Failures and Rollbacks

Another risk that most cloud solutions run into are deployment failures and rollbacks. With feature flags, new features should be disabled, if there are incremental improvements are made to features that are already rolled out to global production, the feature should not be disabled. Proper deployment rollback needs to be performed, frequent deployment failure and rollbacks can significantly increase the downtime. Another important thing to note is following bad deployment practices like failing to ensure the database has no active transactions or failure to backup all necessary information can affect rollbacks or risk losing critical business information.

Another risk with deployment failures can arise from incorrectly managing feature flags that can potentially allow a feature to be released without proper validation. As multiple features and their corresponding feature flags are developed, these flags accumulate within configuration files over multiple releases. This adds the overhead to micro-manage each flag beyond their intended use at the time of production testing and global rollout. Once features are enabled in production and enter into the global release phase, it's associated feature flags that control the deployment should be removed from configuration files. Failing to do this increases the time production testing team and deployment team spend on identifying the correct feature flags to enable or disable during each deployment cycle.

6.3 Net Promoter Score

Net promoter score can be critical component to an organization to determine how well a solution is performing. A seamless deployment mechanism where downtime is not noticed by a user is an indication where the solution will likely have promoters. A cloud solution that has nothing new to offer over a period of time coupled with failed features and downtime has the potential which leads the detractors to simply give up on the cloud-based solutions. It is essential to balance testing strategies, features and production testing in line with traditional testing methodologies to focus on achieving a good net promoter score.

7 Conclusion

In this paper we have discussed on how quality confidence is impacted across traditional and production test environments. We have suggested some best practices for making a solution to be production-test ready by designing a solution as small incremental features, incorporating feature flags to help disable poorly performing features during various strategies of continuous deployment. We have also discussed setting up a production testing pipeline alongside existing traditional testing methodologies and releasing features independently in a phased manner. We have discussed in length about the benefits of incorporating telemetry within a solution as a feature, anonymizing datasets and grouping them into different categories and design a better solution proactively by including telemetry analytics within a product's lifecycle. We also called out some risks in production testing if proper controls are not implemented. Production testing is an extension to existing traditional testing methodologies and provide accurate performance and behavioural results in actual production environments. Utilizing quality related information from production tests will help a web-based solution to deliver better quality features, with minimum or zero downtime.

References

1. Canary deployments: http://docs.octopusdeploy.com/display/OD/Canary+deployments
2. A/B testing, Blue-Green, Canary Deployments: http://blog.christianposta.com/deploy/blue-green-deployments-a-b-testing-and-canary-releases/