# Improving Resiliency of Distributed Applications Using a Validation & Verification Platform

For enterprises, having a platform and tooling for applying chaos, or inducing disruptions to validate and verify distributed applications, is an effective way for avoiding breaches of SLAs and ensuring robust uptime.

## Executive Summary

In current times, many modern applications are distributed in nature. It is commonplace to see various software components of the application – e.g., user interface, integration, data storage and application services – deployed in multiple physical servers/virtual machines/containers. These repositories are often available in the cloud, including public, private or a hybrid connection to the enterprise data center.

Architects can pick and choose from the compute units, storage units and other platform services into deployment architecture to run applications. However, applications deployed in such distributed environments that maintain SLAs become an architectural challenge. Actually, the following assumptions may not be applicable, leading to a breach of SLAs in the production environment:

- The network is reliable and secure.
- Latency is nil.
- The network is homogeneous.
- Network bandwidth is infinite.
- Cloud infrastructure is 100% available.

Ensuring the application is available and is resilient in the face of events – such as a failure in compute and dependent services – requires efforts that are different and more involved than the traditional ones required for private data centers. In a way, the distributed-environment issues mentioned above are considered the new "normal." Hence, dealing with these issues entails deployment planning, design effort, platform support and validating the application's behavior during various events. For example, the application may provide a product list from its cache when the product database is unavailable for some reason.

One very effective approach is to enlist the help of a validation and verification system (sometimes popularly termed a "chaos platform"). Such a system allows architects (with knowledge of the application architecture and deployment environment) to create plans to stress and disrupt application components and carefully monitor system behaviors.

The enterprise application on which validation is to be performed can be termed the "victim application" (see Figure 1); such an application may have multiple components and services, where executing such validation is important. The

### Validation Platform Overview

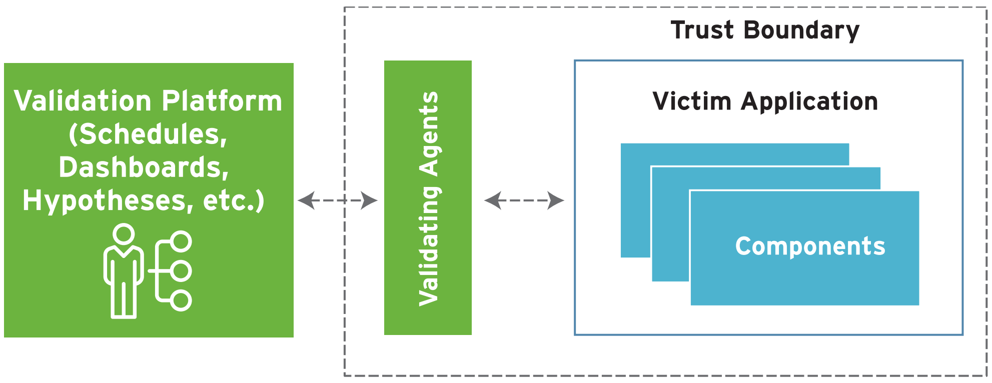

Validation Platform (Schedules, Dashboards, Hypotheses, etc.)
Validating Agents
Trust Boundary
Victim Application
Components

Figure 1

## Steps in Verification & Validation Process

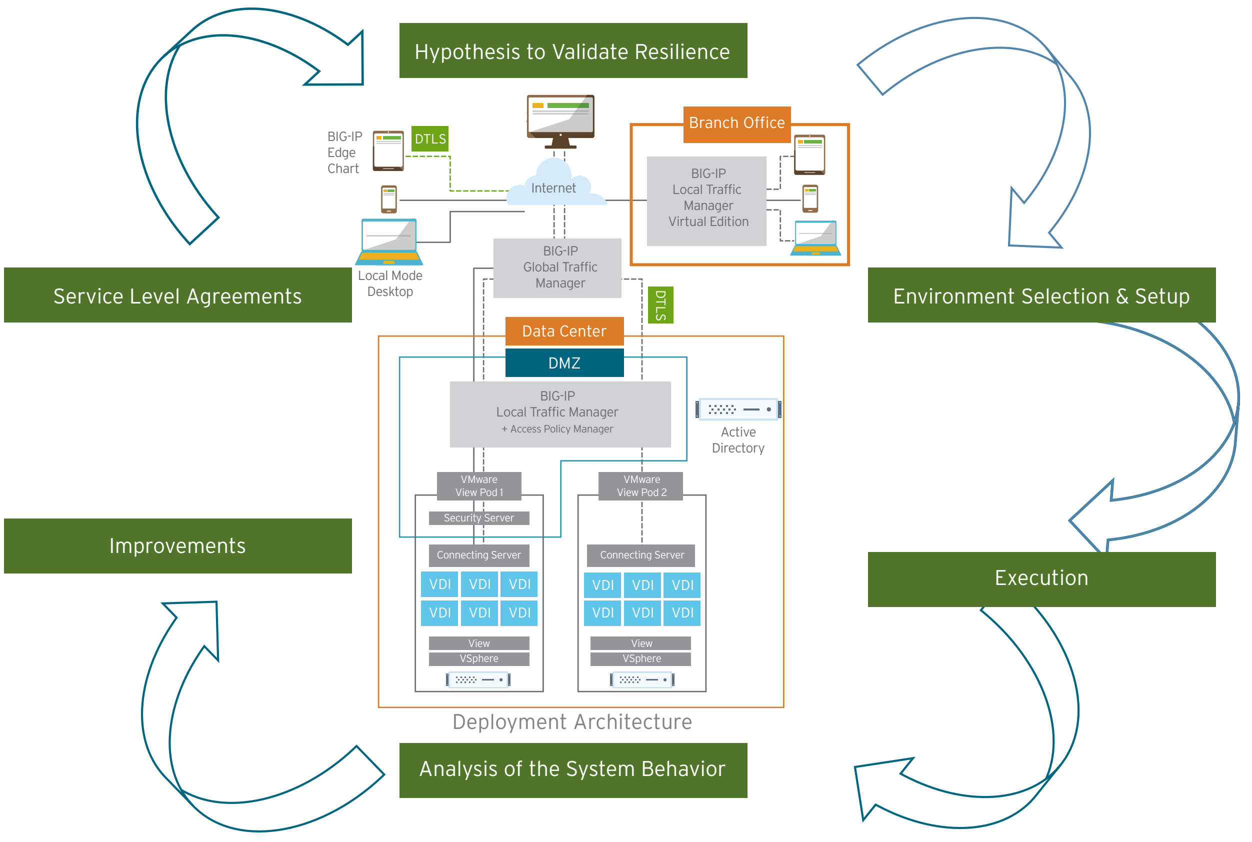


Figure 2

application may be deployed in any data center, including the cloud. In a secured enterprise environment, the validation platform can deploy agents within trusted boundaries, accept commands from the platform and send statistics to it.

A validation and verification system can typically consider the following elements:

- Understanding the SLAs expected from the victim application (see Figure 2).
- Formulating a hypothesis that automatically puts stress on various system components.
- Executing the plans on the selected deployment environment (e.g., production, staging, QA, etc.).
- Carefully monitoring the results of the execution and using the knowledge to improve the deployment architecture and, if required, recalibrating the SLAs.

This white paper elaborates the challenges in validating a typical distributed deployment architecture and its enterprise context, and offers an approach to mitigating such problems by using a platform to institutionalize the process.

Because failure cannot be avoided in the case of distributed applications, such applications should be prepared to handle and survive failure situations. In other words, these applications should become more resilient.

## FACTORS ADVERSELY IMPACTING APPLICATION AVAILABILITY

Because failure cannot be avoided in the case of distributed applications, such applications should be prepared to handle and survive failure situations. In other words, these applications should become more resilient[1] to failure. The applications should be designed and verified to be able to handle unexpected situations without the user's knowledge or with a graceful degradation of service. To do that, the focus should be on minimizing the mean time to repair (MTTR).[2] Some of the events that can lead to compromised availability and reliability include the following:

- **Server overload**: A local overload in one cluster may lead to its servers crashing; in response, the load balancing controller sends requests to other clusters, thereby overloading their servers and leading to a service-wide overload failure.
- **Resource hogging:** Running out of a resource can result in higher latency, elevated error rates or the substitution of lower-quality/stale results. Depending on what server resource becomes exhausted and how the server is built, resource exhaustion can render the server less efficient or cause it to crash, prompting the load balancer to distribute the resource problems to other servers. When this happens, the rate of successfully handled requests can drop and possibly send the cluster/service into a cascaded failure state. Some reasons for this happening are as follows:
  - » **CPU**: Thread starvation, long queue, too many in-flight requests, timeouts at client.
  - » **Memory**: Increased rate of garbage collection, cache miss, too many in-flight requests.
  - » **Threads**: Too many, frequent switches, starvation.
  - » **File descriptors**: Running out of file descriptors.
  - » **Disk capacity**: Exceeding the limit.
- **Cascading failures**: Resource exhaustion can lead to server crashes. Once a couple of servers crash on overload, the load on the remaining servers increases, causing them to crash as well. The problem tends to snowball, and soon all the servers begin to crash  It is often difficult to escape this scenario because as soon as servers come back online they are bombarded with an extremely high rate of requests and fail almost immediately.
- **Network unavailability or unacceptable latency**, leading to SLA breach of the connected elements.

- **Dependency on third-party services**, leading to either unavailability of dependent services or serving data of degraded quality.
- **Data center availability**: It is not rare for a data center belonging to a region to go off in the cloud. In that scenario, the services can be unavailable, jeopardizing the entire application's availability.
- **Microservices**: In applications based on microservice architecture, there are hundreds of interconnected microservices, often developed using different technology stacks. Having a number of different components in the system requires different approaches to keep it up and running, compared to a monolith.

## CHAIN IS WEAKER THAN THE WEAKEST LINK

As mentioned above, improving application availability and reliability is often about striking the right balance between component design and the utilization of the underlying platform/infrastructural capabilities. Popular design patterns – e.g., transient fault handling and circuit breakers – along with platform capabilities provided by popular cloud vendors such as AWS, Azure, Bluemix, etc. can be utilized to improve application availability.[3]

Here are some representative examples:

- Durable queues and asynchronous communications result in applications that are loosely coupled, thereby raising the chances that one failure will not result in cascaded failures.
- Availability sets ensure that should a planned or unplanned maintenance event or failure occur, at least one VM instance will be available for use.
- By placing all "web tier" applications into a single availability set, it becomes straightforward to reboot or upgrade the entire tier at one time.
- Workloads can be placed on geographically separate data centers, and advanced routing mechanisms – e.g., Azure Traffic Manager or AWS Route 53 – can be used to switch operations from the primary data center to the backup in the event of a catastrophic failure in the primary center.

Improving application availability and reliability is often about striking the right balance between component design and the utilization of the underlying platform/infrastructural capabilities.

No single component can guarantee 100% uptime (and even the most expensive hardware eventually fails), which leads to the conclusion that applications have to be architected in such a way that individual components can fail without affecting the availability of the entire system.

No single component can guarantee 100% uptime (and even the most expensive hardware eventually fails), which leads to the conclusion that applications have to be architected in such a way that individual components can fail without affecting the availability of the entire system. Implementation of such applications will demand adhering to some best practices, such as isolation (isolating parts of the system, bulkhead implementation), stateless (avoiding storing states), idempotency (loose coupling between participating components), self-containment (loose coupling between deployment units), circuit breakers (isolating dysfunctional downstream systems), avoiding single points of failure, etc. In effect, overall applications have to be stronger than the weakest links.

## VERIFICATION & VALIDATION: BACKGROUND

As we can see, it is important to constantly and continuously validate whether a distributed application is really able to withstand and recover from various types of failures in its running environment. A proven approach toward this is to apply principles of chaos engineering, which has already been done by organizations like Netflix.[4] Netflix has successfully applied advanced principles of chaos to constantly verify the resilience of its applications in a production environment, and it has been successful in running web-scale enterprise applications.

The principles of chaos engineering, when applied on a distributed application, are as follows:

- Build a hypothesis around steady state behavior.
- Simulate real-world events.
- Run experiments in production/staging/QA, etc.
- Apply automated tool-based failure simulation, which involves randomness and all possible combinations of failure such as CPU utilization nearing 100% along with network loss, or memory utilization nearing 100% with disk full, or some components of the target application not being available.
- Automate experiments to run continuously: Preferably, putting such validations in the DevOps pipeline followed by strong monitoring and metrics collection of the application under chaos, so that a quick restoration from an undesired state can be triggered if required.

### Enterprise Context

Netflix's scale and its velocity of releasing new features, its presence and its global distribution pose unique problems that led to the creation of the Chaos Monkey tool. However, this concept is equally applicable to the enterprise. Elements that are unique to enterprises are:

- **Controlled environment**: Enterprise IT services want to minimize the number of support scenarios that are intermittent or difficult to trace. They would like to identify system vulnerabilities early on to take corrective measures. They also want to be able to control when and how many instances, and on which environments, the verification and validation for availability need to run. The metrics collected from the execution should be reported, or in some cases fed to the monitoring system – e.g., System Center Operations Manager, Kibana, AppDynamics, etc.

- **Repeatability**: Enterprise IT would like to carefully plan the execution of chaos, and schedule it specifically to include/exclude enterprise workload schedules (e.g., not running chaos during data-feed time windows), execute it over and over again, and possibly integrate with their DevOps pipeline. They also want to configure the plan when new services or instances get added to the application's current deployment topology.

- **Secured**: Enterprise IT wants to administer which instances/services are chosen for chaos execution and the credentials/keys used to run the execution. They will need to isolate the instances that have exclusive access to the victim application to execute chaos. Most important, they will want to store the key files required to access the instances in their own safe custody.

- **Customizable and extensible**: Changing business needs may prompt an enterprise to introduce new dependencies such as in deployment. They may prompt the enterprise to completely shift to AWS cloud from the data center, shift some select workloads to cloud, introduce Linux or Windows instances, etc. In cloud, the enterprise may want to target chaos execution on IaaS or PaaS. Modern PaaS offerings from providers such as Azure provide system development kits (SDKs) specifically for this purpose. The enterprise might want to apply validation and verification encompassing all these scenarios.

- **Coverage**: Enterprises that own the applications will want to execute chaos in an orchestrated, planned way – rather than execute random chaos – on components that are aware of application flows, and tune/repeat the process to ensure coverage of the most vulnerable components.

Enterprises can target a platform with the capability to create hypotheses, import scripts and components, and verify credentials – and then use them to create schedules to automate the resiliency check.

## Participating Elements of Verification & Validation Platform

| Elements | Details |
|---|---|
| **VALIDATION PLATFORM** | |
| Dashboard | A dashboard to monitor and control the execution of the hypothesis by communicating with the appliance/agent. |
| Hypothesis & Topology Designer | Topology builder workbench that lets a designer create an application topology visually that represents the victim application, design the chaos plan and select target instances end points to execute chaos. It offers a set of plans consisting of target instances, scripts, schedules and credentials that are applied to instances and services to cause planned and controlled disruptions to measure application behavior. |
| Topology Metadata | The data store to hold metadata (e.g., JSON) created by the Hypothesis Builder workbench. |
| Agent/Appliance | A set of agents that act as a conduit between the victim application instances and controller services that initiate chaos via orchestrator and scripts. |
| Controller | Set of APIs that takes command and returns results to the dashboard by communicating with the agent/appliance, and can also be used in "headless" mode devoid of any UI. |
| **MONITORING SOLUTION** | • Not part of the platform, but any monitoring tool such as AppDynamics, DynaTrace, Kibana, Nagios or provider-specific CloudWatch. These can be used to monitor the statistics of events while chaos is applied. • Custom scripts that can collect data from instances and various logs to give a clear picture about the details of failures in validations, if any. |
| **VICTIM APPLICATION** | LOB application where chaos is applied. This will typically be instances, service end points in a data center or cloud platforms such as AWS Azure. |
| **LOAD GENERATOR** | Standard load generating tools like Visual Studio Test Manager or LoadRunner that execute recorded test scenarios. |

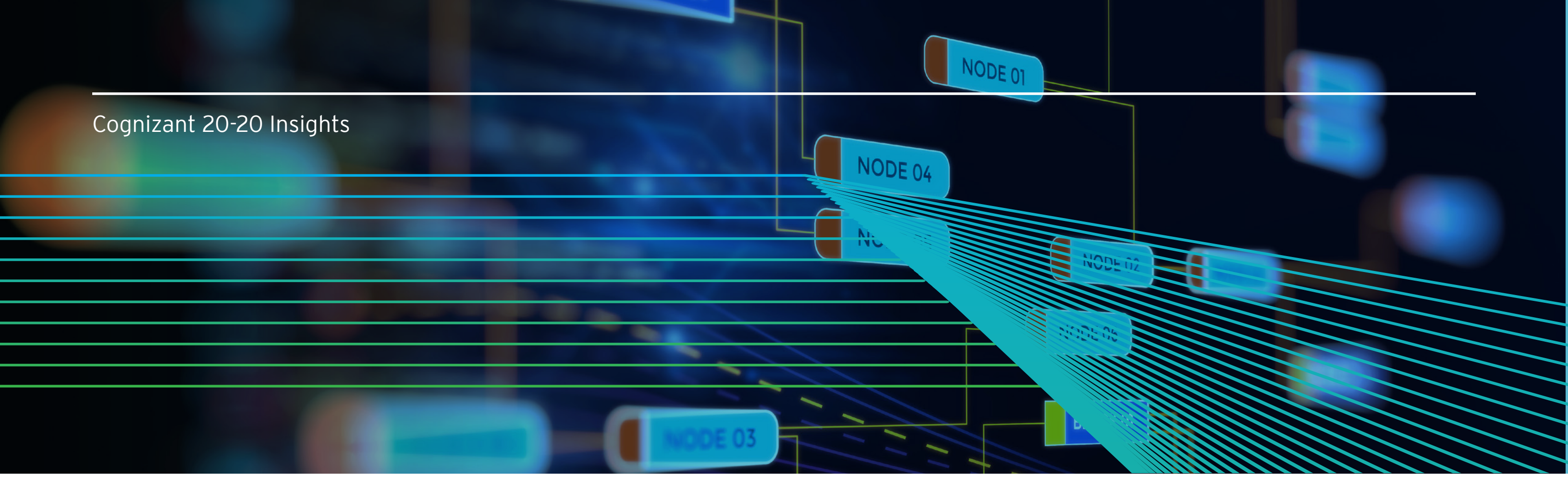
Figure 3

## VERIFICATION & VALIDATION: A PROPOSED METHOD

Encompassing the concepts of the preceding sections, enterprises can target a platform with the capability to create hypotheses, import scripts and components, and verify credentials – and then use them to create schedules to automate the resiliency check. It can also augment the existing release management to ensure resiliency of the deployed application. Hypothesis formulation is a key activity in this proposed solution.[5] The core building blocks around such platforms could be as shown in Figure 3.

The elements of a validation platform can be put together as follows:

- A web interface can be used to set up the hypothesis that is stored as metadata; it also contains schedules, credentials and scope data.



- An appliance sits at the deployment environment responsible for orchestrating validation scripts; it also communicates with the controller for reporting.
- The appliances/agents are secured with key files and SSH for IaaS, user credentials for PaaS.
- Controller acts as a channel for executing commands and metrics collection.

The validations can run multiple iterations to find faults and cluster for the specified period of time. A scenario fails when the platform hits a single failure in cluster validation. As an example, consider a hypothesis that is set to run for one hour with three concurrent faults. The test will induce three faults and then validate the cluster health. The test will iterate through the previous step till the cluster becomes unhealthy or one hour passes. If the cluster becomes unhealthy in any iteration – i.e., it does not stabilize within a configured time – the test will fail. This indicates that something has gone wrong and needs further investigation.

## BENEFITS OF AUTOMATED RESILIENCY VALIDATION

Detecting the breach of SLAs in any environment, including production, is always a tricky affair and can lead to wasted hours and end-user dissatisfaction. We recommend a resilience validation platform based on the principles described in this paper. Such a platform has the advantages of being applied to many application topologies in the enterprise, using reusable hypotheses that can be run over and over again based on the enterprise workload. The platform is suitable for enterprise scenarios with a focus on security. The enterprise will have options to use its own monitoring tool and custom scripts to track the impact of hypothesis execution. The platform can have its own AppStore capability from where the agents to be deployed to enterprise instances can be downloaded. The greatest benefit of this could be utilizing the insights retrieved from the execution into making better resilient architecture that adheres to SLAs.

## REFERENCES

- http://techblog.netflix.com/2014/09/introducing-chaos-engineering.html
- http://techcrunch.com/2012/07/30/netflix-open-sources-chaos-monkey-a-tool-designed-to-cause-failure-so-you-can-make-a-stronger-cloud/
- http://www.ibm.com/developerworks/library/a-devops4/
- https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-controlled-chaos

## FOOTNOTES

[1] Application resiliency means the power to return to the original state after deviations, due to some influencing factors; these factors ultimately lead to application availability and reliability. Reliability means the application is able to serve end users or systems, with acceptable SLAs.

[2] Availability (A) = MTBF / (MTBF + MTTR), $A = Ax . Ay$ when components are connected in series, $A = 1 – (1 – Ax)^2$ when components are connected in parallel.

[3] Azure compute service comes with a 99.95% SLA; Azure SQL Database has a 99.9% SLA; and Azure Storage has a 99.90% SLA. Without any additional work, your application is by default guaranteed no more than 108 minutes of downtime in a month (out of 43,200 minutes).

[4] Netflix pioneered the concept of Chaos Monkey, a tool that randomly disables production instances to make sure the application can survive this common type of failure without any customer impact. Chaos Monkey is best run in the middle of a business day, in a carefully monitored environment with engineers standing by to address any problems.

[5] An example of the hypothesis could be: "To establish that overall workload is handled in the system with X% success with Y% degradation (e.g., only 50% of read operations returns HTTP 200 OK with less than 7 secs.) despite the database engine process, and then the web server process in the application server is killed."

## ABOUT THE AUTHORS

### Sandip Bandyopadhyay

**Principal Architect, Cognizant Technology Solutions**

Sandip Bandyopadhyay is a Principal Architect within Cognizant Technology Solutions, with more than 18 years of experience implementing applications in Java, AWS and various open source platforms such as node.js. Currently, he leads a team focusing on tools and assets development around microservices. Sandip holds a master's degree in physics from Calcutta University. He can be reached at Sandip.Bandyopadhyay@cognizant.com | LinkedIn: www.linkedin.com/in/sandip-bandyopadhyay-880971/.

### Moinak Bhattacharya

**Chief Architect, Cognizant Technology Solutions**

Moinak Bhattacharya is a Chief Architect within Cognizant Technology Solutions, with over 19 years of experience implementing applications and platforms in .net-based technologies, AWS, Azure and various open source environments around javascript. Currently, he leads a team focusing on tools and assets development around SaaS, microservices and cloud. Moinak holds a master's degree in computer applications from NIT Rourkela. He can be reached at Moinak.Bhattacharya@cognizant.com | LinkedIn: www.linkedin.com/in/moinak-bhattacharya-17730588/.