Testing Services in Production

Keith Stobie
Keith.Stobie@microsoft.com

Abstract

There are many benefits to be realized by Testing Services in Production when the risks are properly mitigated. Testing in production finds problems at a scale that most groups can't afford to duplicate with a test environment. Using production systems for testing is critical to business success of an effective software service. This paper describes and demonstrates several different approaches to using production systems for testing including: when each approach is appropriate, what prerequisites are needed, and how each approach would be used.

Monitoring of services, Controlled Experiments, and production data are well-known forms of testing. You can also use tracers to follow service flow, do destructive testing (killing services, networks, etc.), and even do load, capacity, performance, and stress testing in production. In short, almost all kinds of testing can be done in production, but how do you mitigate the risk? Testing in production allows customers to benefit (and occasionally to suffer) from the most current advances. Throttling requests or work, exposure control, incremental rollout, and especially superb monitoring are all needed to control production testing risk.

Biography

Keith Stobie is a Principal Software Development Engineer in Test working as a Knowledge Engineer in the Engineering Excellence group at Microsoft. Previously he was Test Architect for Bing Infrastructure where he planned, designed, and reviewed software architecture and tests. Keith worked in the Protocol Engineering Team on Protocol Quality Assurance Process including model-based testing (MBT) to develop test framework, harnessing, and model patterns. With twenty-five years of distributed systems testing experience Keith's interests are in testing methodology, tools technology, and quality process. Keith has a BS in computer science from Cornell University.
ASQ Certified Software Quality Engineer, ASTQB Foundation Level
Member: ACM, IEEE, ASQ.

1. Introduction

Web Services and the cloud are becoming the pervasive way to deliver software functionality to users. As users become dependent on cloud services, they require high assurance about reliability and availability of the service. News reports of failures in public services are frequent (Infosec 2011) (Nguyen 2010). A growing consensus on major web services prescribes testing in production as effective risk mitigation (Ciancutti n.d.) (Hamilton 2007) (Johnson n.d.) (Michelsen n.d.) (Rigor 2011) (Soasta, Inc. 2010). There are still a few holdouts that question testing in production (Rathi 2011).

To make Testing in Production relatively safe, as Hamilton (Hamilton 2007) noted,

> "The following rules must be followed:
> - the production system has to have sufficient redundancy that, in the event of catastrophic new service failure, state can be quickly recovered,
> - data corruption or state-related failures have to be extremely unlikely (functional testing must first be passing),
> - errors must be detected and the engineering team (rather than operations) must be monitoring system health of the code in test, and
> - it must be possible to quickly roll back all changes and this rollback must be tested before going into production."

To keep production metrics correct you must treat data carefully as given in section 1.1. In order to minimize risk while doing production tests you need exposure control as given in section 1.2

Section 2 discusses various approaches to production tests starting from simple Asserts in 2.1 and basic Monitoring in 2.2 to rather traditional testing offline in production environments in 2.3. Controlled experiments as discussed in section 2.4 began as a design comparison technique but can be leveraged by test teams into production testing of new versions, etc. More challenging, after mastering the previous approaches, is capacity testing the live site (2.5), data injection via tracers (2.6) and fault injection (2.7).,

1.1 Isolate Test Data from Production Data

If you are running tests in production and, like most production sites, you are logging for analysis what happens on the live site, you must consider how your tests can potentially perturb the live site data. Some sites may argue that production tests are such a small fraction of their overall workload (say < .01%) that they don't care about the perturbation. Validate if this is really true for the service. Any billing for services can't include tests (except for test billing).

Most services will want to flag and or separate production tests from real production work. Many ways exist to do this. A simple way for HTTP-based work is to include an HTTP Header parameter string, e.g., `Pragma: test`, or an additional parameter string, e.g., `&traffictype=test`.

Then you have to modify your log processing routines to separate and potentially just drop the test data. Failure to remove the test work can have several negative repercussions including:

1) Viewers of the live site metrics may get a false picture (more activity or more errors than are really there)
2) Billing based on live site metrics could be skewed

Some services use a location setting for the browser that does not correspond to any real country or region. Instead of reporting the traffic as coming from a user in India it is reported as coming from a user

in “test.” When they process data to bill their customers, the production tests are automatically filtered out.

Creating user observable test data in production could create a detrimental user experience. Several methods have been used to avoid exposure. Some go for obscurity, e.g., using a language that doesn’t exist. For example, instead of en-US or zh-CN, they might use xx-XX. Other services filter out test data in their front ends. For example, they won’t return test data to any request not coming from a domain inside the production or test environment.

1.2 Exposure Control

Beyond exposure of test data, a variety of approaches is used to control exposure of new versions or test versions. Common practice is to make it available to a limited audience on a limited set of resources and gradually expand both (Eliot 2009). For example, only enough machines and network to run a minimal instance are used initially and then only accessible to a restricted set of people, e.g., employees and then friends and family or external crowd source testers. The initial instance has many names such as slice of production, canary, etc. After a proving period, incremental rollout of the new version is allowed into more production resources in one data center and across many data centers until, when proven, it becomes the default for all data centers. Similarly user exposure may be controlled to Beta users, specific regions, or specific countries.

2. Test in Production Approaches

Test in Production presumes a relatively stable system already verified using standard unit-testing, component testing, and system test techniques. Live sites must be verified for their processes as well including standard operating procedures, incident handling procedures, and escalation procedures. Many organizations and standards cover live site procedures.

Depending on the architecture and maturity of the web service, some approaches are easier to adopt than others. Offline testing in a production environment is decades old and should be easily accomplishable. Asserts are the easiest to adopt and observational monitors are critical to the running of any service. Experiments, A/B testing, are actually a standard marketing approach (MarketingExperiments n.d.) and many third parties (e.g. (Google n.d.)) can help a site do this with no instrumentation changes to the web service except providing the A and B versions. Self-Test is more elaborate, but is a standard coding exercise not unique to services. Synthetic transactions are usually the first test in production approach that requires careful thinking. Tracers are like synthetic transactions but coming from a data perspective instead of a user perspective. Most advanced are Live performance, capacity, and stress tests and also Live fault tolerance tests.

2.1 Asserts

The most basic aspect of Testing in Production is to leave asserts that don’t significantly impact performance in the production code. Services need to have multiple instances of the actual code running both for scaling and availability. You must have no single point of failure in your system (hardware, software, or environment). Your code should be reasonably well tested before going into production, so an assert firing should be a relatively rare event and affect only one or a few instances. Given multiple instances and independent Heisenberg (Gray 1985) asserts firing, asserts are your first line of defense in detecting and recording rare problems via crash dumps or virtual machine images. Through analysis of the asserts fired you can make your system even more robust.

2.1.1 Self-Test

More advanced than simple asserts, is program self-test, just like what is done in hardware. Self-Test generally requires you already have asserts in place and is appropriate for long running services or for background verification of persistent data. Many programs are written with a low priority thread for doing verification or self-test for when the program is not actively used. During these idle times, the verification thread will make various checks such as walk and verify data structures. Similarly many services have low priority background processes that do nothing but constantly look for passive failures (e.g., disk sector becomes unreadable) or data corruption. For example, when you have data replicated three times, you can constantly compare the replicas and, upon disagreement, generally choose by vote the most common two as correct, and repair the third.

2.2 Monitoring

A critical aspect to running a web service is monitoring. Without monitoring, a service is being run blind and it is dangerous to even launch such a service. Monitors which only observe the system are effectively Test Oracles indicating if the service is successful or failing. Monitors measure system and service behavior and raise appropriate alerts or alarms depending on the measurements. A prerequisite or co-requisite for monitoring is an effective alerting and alarming infrastructure. Monitoring requires system capacity that must be planned. Efficiency of the monitors and whether they can be tuned for more detail or less detail will also affect the required resources needed. Observational monitors can be simple measurements like CPU usage above a threshold, e.g., 80%, or memory usage above a threshold. They can be time averages such as average response time over the last five minutes being less than one second. The response time measurement could be based on real user requests and responses. Monitors can be multi-level, as shown in Table 1 where ninety percent of requests must have response time less than two hundred milliseconds.

Table 1. Response Time Threshold observing monitor

% of requests	Threshold
50%	100ms
90%	200ms
99%	300ms

For monitors to be effective, testing of services in at least some non-live environments should be set identical to the production environment to indicate if the thresholds fire too often (false positives) or not enough (false negatives). The frequencies with which measurements are made also affect the performance and responsiveness of the system.

An observational monitor could measure availability by observing how often an error or no response is provided for a request.

2.2.1 Synthetic transactions

A synthetic transaction is a monitor which actively provides input (test data). It may also actively verify the response or rely on existing observational monitors as the Test Oracle. The simplest example is a ping of network system to see if it responds at all. Sometimes availability measures are made based on synthetic transaction pings, but these can be unreliable indicators of actual availability such as you might get from an availability observational monitor.

Almost any functional test can be constructed into a synthetic transaction monitor. Deciding on which tests to run and how often are critical factors in the success of using them to prevent major service issues from arising or to detect issues quickly. Unlike functional tests, many times the test can't rely on a

specific state of the service while it is Live. Thus synthetic monitors may need to use the Delta Assertion pattern (Meszaros 2007).

Some synthetic transactions should always be in use to verify basic service operation. The larger the consequence of failure of a particular test, the more frequent it should be run. A good reporting infrastructure makes synthetic transaction results more valuable. Trends can be observed such as whether some tests only fail at a particular time or under a particular load.

2.3 Offline Testing in Production Environment

For decades services had the luxury of not being available all the time. It was acceptable to have schedule unavailability for some services, for example stock markets. During these offline times, the production environment could be used to conduct tests, but these were not while the system was live.

Similarly, today many services are either provisioned for a maximum load that rarely occurs (e.g., Tax filing day or Black Monday - the biggest online selling day of the year, etc.) or use cloud services that allow dynamic capacity to be purchased on demand. For maximally provisioned services, it is usually possible to schedule taking some of the resources offline during known low usage periods and use the production resources offline for testing. Similarly for dynamically purchased capacity, you can purchase production capacity for testing as easily as for live usage.

Testing offline in a production environment helps by having an environment as nearly identical to the live environment as possible, but still it isn't the live environment.

2.4 Controlled Experiments – A/B Testing

A/B testing is seeing how users interact with two different implementations and, based on some measure of goodness, selecting the better one. For A/B tests, the characteristic is to have production code of equivalent quality so the focus on the experiment is on distinguishing user preference for the designed differences between the implementations.

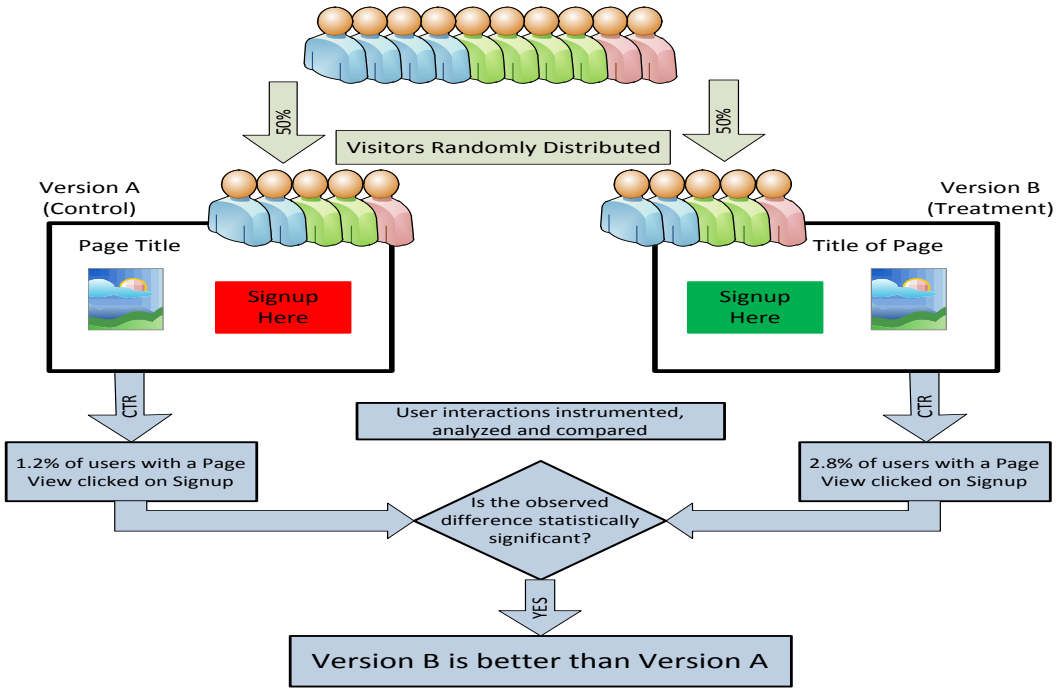


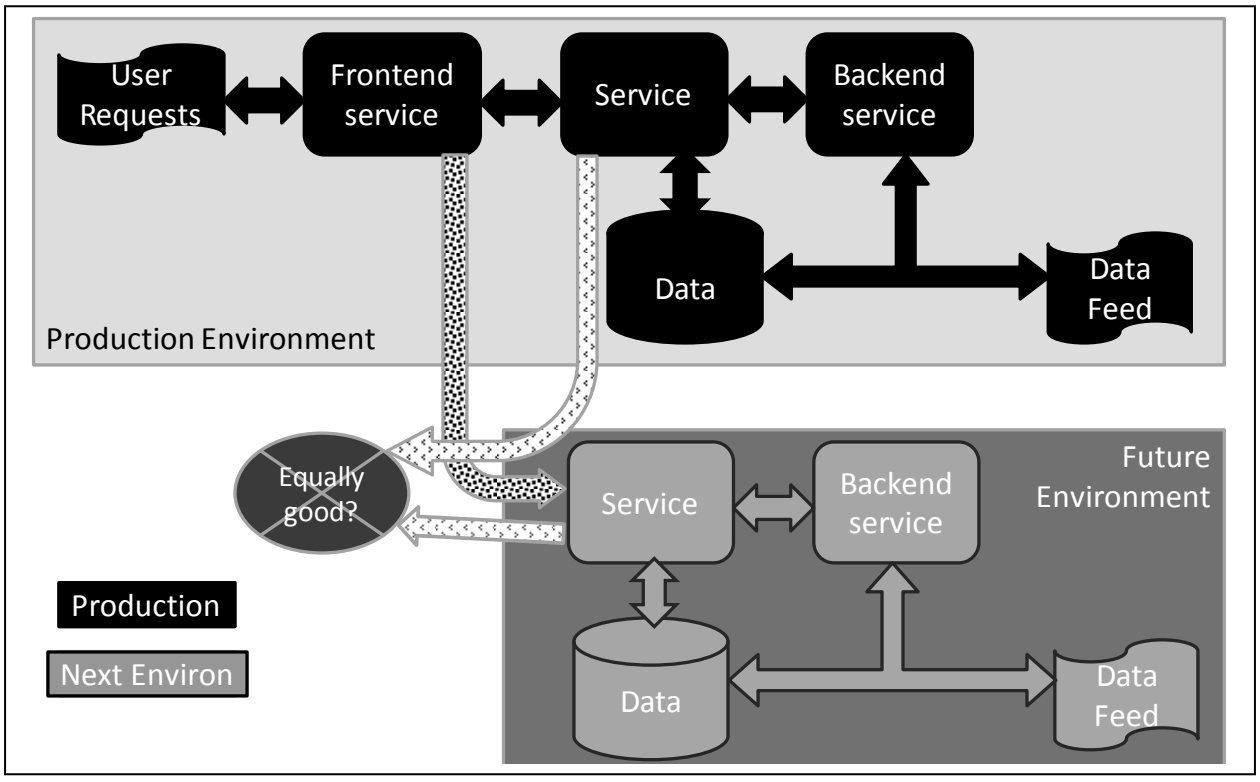
Figure courtesy of Seth Elliot from his SASQAG (Seattle Area Software Quality Assurance Group) April 2011 Talk Testing in Production - Your Key to Engaging Customers

(CTR = Click Thru Rate)

You can also consider mixed-mode testing (have a current version and next version of the same service running) for A/B testing by looking for no statistically relevant differences. Depending on the features or changes in the new version, you may expect to see one or more statistically relevant improvements. You can be looking specifically at CPU usage, Disk usage, I/O, etc. You also may look at the traditional user engagement metrics (Click-through rate, time on page, etc.), as a decrease in these for new version might indicate a possible problem (i.e., if the service is not responding in a timely fashion, it can cause users to click away from the page). You can then decide, based on the data, if the new version is an improvement worthy of complete roll out and replacement of the current version.

A/B testing requires enough instrumentation to tell if you are doing better or worse than before and especially the ability to react quickly when it shows the new treatment is worse than the old control. Typically the treatment is discontinued as soon as it is shown to be statistically significantly worse. To obtain statistical significance there must be sufficient usage of the both the A and B (or control and treatment) versions. When the new treatment is shown to be better for a sufficiently long period of time, it can replace the control treatment as the standard version.

An easier method is to just shadow the production environment with a new environment and fork the requests and compare the replies and other data to see if the new environment performs equally or better.



2.5 Live Capacity Testing (fake traffic)

Live Capacity Testing is done only after your planning indicates you should be able to handle the capacity and smaller scale tests have been done to provide some level of credence to the planning models. However, planning models and the full scale are rarely identical and thus, after small tests and capacity model indicate it should work; the service should verify it actually does work.

The first goal of a live capacity test is to not impact users. This goal must remain paramount by constant monitoring of Service Level Agreements (SLAs) and throttling back or discontinuing the test even before an SLA is broken. Two major safety mechanisms are test auto-throttling and product throttling. In test

auto-throttling the load generation tool consumes real time monitors and will automatically pause or stop the load as the test approaches SLA or Key Performance Indicator (KPI) thresholds. With product throttling, the servers have technology to throttle capacity traffic if latencies increase significantly. Throttling can be by discarding low value traffic (e.g., Bot traffic). Throttling can also occur (but less desirably) by producing lower quality results (e.g., older cached data, instead of the freshest data).

A goal of live capacity testing is to verify business continuity scenarios in production at or beyond existing observed peak load while meeting SLA (uTest 2011). Typical components of an SLA include availability (request completion), request latency (e.g., average and percentiles), as well as requests per second. For an example goal, let N be the recently measured peak of requests per second. Assume you want to allow for projected growth of fifteen percent (15%) and you want to accomplish all of this during a Business Continuity (BC) event when one of your four Data Centers (DC) is down. With four DCs and one DC down, each remaining DC must have 1.33 capacity needed, so capacity test traffic would be 1.15 * 1.33 = 1.53N

Live capacity testing allows physical asset verification (network gear, DC power, etc.), validation of the service in new datacenters, and optimizing utilization of existing equipment via proper balancing. In prior tests, Bing has observed overheating when everything ran full throttle, or exceed maximum power input for the DC from utilities.

A way to do capacity testing is having a controller drive each of the clients across all DCs. The load is generated within each datacenter to avoid cross-DC network utilization (unless that is what needs to be tested). The controller pulls data from monitoring system, auto-throttling based on SLA. Are you providing the traffic using your products' APIs or also via emulated users using browsers? A capacity test passes when traffic goals are reached within SLA. You also need to consider the impact of your capacity test on partners that your service uses. Do you want to fake the partner traffic or really incur potential costs of capacity testing your partners also?

In planning a capacity test, first determine the overall goal. Examples goals include:

- Set targets based on your service history, predicted growth, required headroom buffer, and Business Continuity goals.
- Verify capacity of default experience
- Verify capacity of ancillary and partner services.
- Verify capacity of new features
- Verify a Business Continuity scenario
- Verify for a period of time, e.g., 4 hours or 24 hours

Second, determine the correct mix of traffic. This is non-trivial as there are many dimensions to consider. Examples of traffic types to mix:

- default traffic,
- Experimental traffic, e.g., future traffic for new feature or service
- markets, e.g., North American, European, Asian
- Geographic, e.g., Central US, Southern US
- patterns, (e.g., Distributed Denial of Service , DDOS, attacks, or typical calling patterns within the service)
- distribution,(e.g.,. cheap operations (like query) vs. expensive (like update))
- temporal (traffic is time dependent)
- cache hit ratio (traffic does or does not hit cache)

Final setup step is to distribute necessary data such as previous sanitized or masked logs of users' interactions with the service, or distribute example inputs to capacity clients.

Before running a capacity test, notify Designated Responsible Individuals (DRI) and capacity partners, ensure site is in a good state, and establish communication, for example via instant messaging or conference call. Then start load generation tool and ramp traffic slowly (until shown peak load is handled, so can throttle back with less risk if peak load not currently handled within SLA). Eventually try different load ramping methods such as spikes or steps. As stated previously extensive monitoring is critical. Stop the test when limits are exceeded, goals are met, or time runs out.

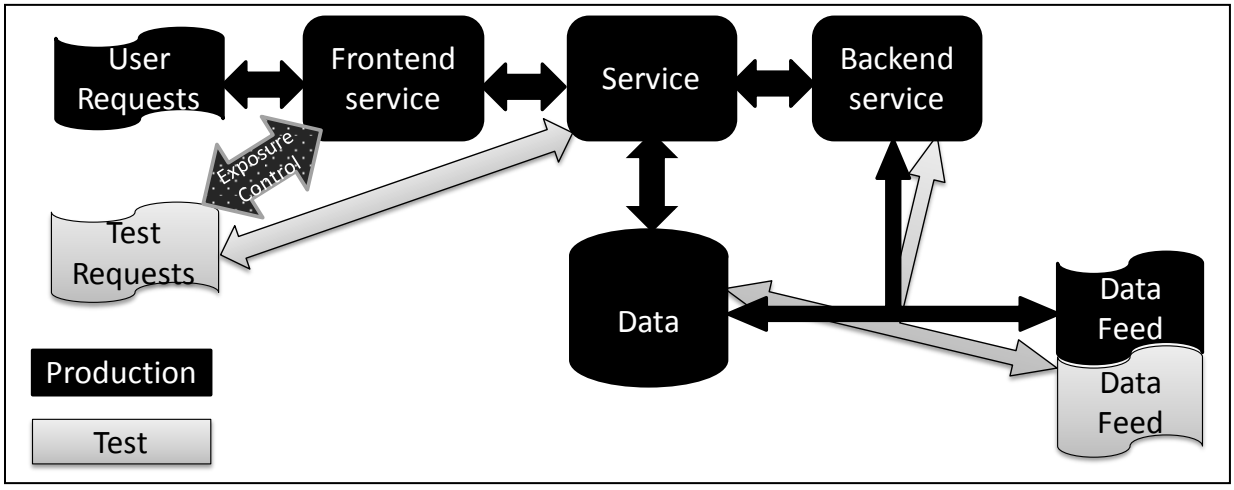
2.6 Tracers

Another method of Testing in Production is verifying correct data flow through the system. While data flow testing of the service in a test environment may show it should work, there is always the possibility of operational and configuration error such that it doesn't work in production. Thus, constant monitoring via injection of test data into back-end data feeds is another critical factor in verifying correct operation of a service. Like synthetic transactions, the data must be marked or handled such that it doesn't impact billing or services metrics related to real traffic.

With observation of the traces at various points throughout the data flow, any data dropped or misrouted instead of correctly transferred can be quickly detected.

For example, in a shopping service you could inject test products, test offers to sell products, and test reviews of products. Monitor injected feeds to see if the product is found in the product authority database. Monitor to see if the offers to sell and review feeds get classified correctly. Monitor if the product matching service matches product offers and reviews to products in the system. Finally, monitor if the user sees product when they search for it; if review search pages display reviews from the feed; and if product details page for a given product contains the matched offer and review information.

In testing tracer data you must consider whether you want or need to control the exposure. Decide if test products can be seen by the general public or only special logins, network addresses, etc.



2.7 Fault Injections

All large systems will have failures. They may be due to hardware failure, software failure, or environmental failure. While failure prevention is encouraged, it is never enough. Thus the system must be resilient enough to recover from failure, and recovery oriented computing (Brown 2001) should be considered.

Fault injection and resiliency testing should be done first in test labs, but that is not sufficient. Again, configuration, operations, specific hardware, etc. may come into play to make live production unique and different from a production-like environment.

For testing in production, we need to verify that the production system is indeed resilient to failure by deliberately inducing failures where the system is relatively reliable. Many services find that some failures, e.g., disk drive failure, are so common (multiple daily occurrences in a large data farm), that no special fault injection is needed. The failure recovery is frequently tested due to the unreliability of certain aspects of the system.

For the relatively reliable system parts, fault injection should occur. It can be planned, but ideally the system can handle random failures as is done by Netflix's Chaos Monkey (Ciancutti n.d.). You may limit the faults to those seen previously and based on previous frequency as a fault operational profile initially. Eventually you should be able to test for all fault types, although frequency of each one remains a critical decision. Commercial devices exist for inducing network data corruption, network latency, and network routing issues. Similarly you can have various types of controlled power failure. Via software you can cause loss of a process, a machine, a service, or potentially the entire DC.

Fault Injection may cause other unexpected failures or potentially a cascade of failures (Amazon 2011). Verifying you can control failure cascades and diagnose unexpected failures are just additional service capabilities you must have that Fault Injection helps verify.

3. Conclusions

Starting from Offline testing in a production environment, most services have added monitoring including asserts, and usually at least limited synthetic transactions. A/B testing experiments have provided proven value that more sites are adopting. But as numerous service outages have shown, more must be done. Architecting the service for Self-Test and making virtually all tests available as Synthetic transactions is required. Part of architecting a service for test in production is dealing with test data in the live environment, whether from synthetic transactions, tracers, or other means. Finally, all type of testing including performance, capacity, resiliency and stress tests must be engineered to work while the live system is running through a combination of service code, monitoring code, and test code.

Examples of companies using Test in Production surviving major failures is growing and conversely there are too many examples of services not handling major failures and not fully testing in production.

Summary Table

Approach	Risk of Causing issues	Exposure Control	Data Control
Asserts	Medium	Optional	N/A
Monitoring	Low	N/A	N/A
Offline	Low	N/A	N/A
Controlled Experiment	Medium	Required	Useful
Live Capacity	High	Useful	Required
Tracers	Medium	Useful	Useful
Fault Injection	High	Optional	Useful

3.1 Acknowledgements

Greg Veith has exemplified how to do Live Capacity Testing. Ken Johnston and Seth Eliot continue to expand my understanding of Test in Production.

References

(MarketingExperiments n.d.)Amazon. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region.* 2011. http://aws.amazon.com/message/65648/ (accessed 6 21, 2011).

Brown, A. and Patterson, D. A. "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)." *High Performance Transaction Processing Symposium.* Asilomar, CA, 2001.

Ciancutti, J. *5 Lessons We've Learned Using AWS.* n.d. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html (accessed 6 1, 2011).

Eliot, S. *Feeling TiPsy…Testing in Production Success and Horror Stories.* 12 14, 2009. http://blogs.msdn.com/b/seliot/archive/2009/12/14/feeling-tipsy-testing-in-production-success-and-horror-stories.aspx (accessed 6 1, 2011).

Google. *Advanced A/B Testing.* n.d. http://www.google.com/support/websiteoptimizer/bin/answer.py?hl=en&answer=62999 (accessed 6 21, 2011).

Gray, J. "Why do computers stop and what can be done about it?" *Technical Report 85.7, PN87614, Tandem Computers, Cupertino.* 1985. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf (accessed 6 1, 2011).

Hamilton, J. "On Designing and Deploying Internet-Scale Services." *Proceedings of the 21st Large Installation System Administration Conference (LISA '07).* http://www.usenix.org/event/lisa07/tech/full_papers/hamilton/hamilton_html/ (accessed 2011/06/01), 2007. 231-242.

Infosec. *Amazon AWS goes down, takes several others along with it.* 04 21, 2011. http://infosecindia.com/2011/04/21/amazon-aws-goes-down-takes-several-others-along-with-it/ (accessed 06 21, 2011).

Johnson, K.N. *Performance testing in the production environment.* n.d. http://searchsoftwarequality.techtarget.com/answer/Performance-testing-in-the-production-environment (accessed 6 1, 2011).

MarketingExperiments . *A/B Split Testing.* n.d. http://www.marketingexperiments.com/improving-website-conversion/ab-split-testing.html (accessed 06 21, 2011).

Meszaros, G. "Delta Assertion." In *xUnit Test Patterns*, by G. Meszaros, http://xunitpatterns.com/Delta%20Assertion.html (accessed 2011/06/21). Meszaros, G.: Addison-Wesley, 2007.

Michelsen, J. *SOA Testing Best Practices: Complete, Collaborative, and Continuous.* n.d. http://advice.cio.com/john_michelsen/soa_testing_best_practices_complete_collaborative_and_continuous (accessed 05 1, 2011).

Nguyen, C. *Facebook Website Down?* 12 2010. http://www.ubergizmo.com/2010/12/facebook-website-down/ (accessed 6 21, 2011).

Rathi, M. *The Support Authority: Why testing in production is a common and costly technical malpractice.* 02 02, 2011. http://www.ibm.com/developerworks/websphere/techjournal/1102_supauth/1102_supauth.html?ca=drs- (accessed 06 01, 2011).

Rigor. *Continuous Deployment - Functional Testing in Production.* 1 7, 2011. http://rigor.com/blog/continuous-deployment-functional-testing-in-production.html (accessed 6 1, 2011).

Soasta, Inc. "CloudTest® Strategy and Approach." 9 27, 2010. http://www.cloudconnectevent.com/downloads/SOASTA_TestingInProduction_WhitePaper__v1.0.pdf (accessed 06 01, 2011).

uTest. *Testing the Limits With SOASTA's Dan Bartow – Part I.* 01 2011. http://blog.utest.com/testing-the-limits-with-soastas-dan-bartow-part-i/2011/01/ (accessed 6 1, 2011).