Automatic Prevention of Accidents in Production

Chang-Seo Park
Google LLC
Mountain View, CA, USA
parkcs@google.com

ABSTRACT

We present a framework for automatically testing functional correctness of back-end servers. We created a pre-production environment where traffic between servers can be reconfigured dynamically. Production requests are sampled and replayed in our framework so that we can cover many corner cases of the system without having the developer manually write test cases. We also describe how to handle mutate requests and support checking the validity of back-end rewrites.

ACM Reference Format:
Chang-Seo Park. 2020. Automatic Prevention of Accidents in Production. In *AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3387903.3389318

1 INTRODUCTION

Faults in production can be costly. Revenue is lost while the error is unmitigated in production and many engineer-hours can be lost while diagnosing and fixing the issue. If it is a potential recurring issue, even more engineer-hours need to be put in to prevent a similar issue from happening again.

Production errors are hard to catch pre-release even with tests. With the popularity of agile methods increasing and more emphasis being put on testing (e.g., Test Driven Development), most software is well-tested with unit tests that aim to reach certain coverage metrics and integration tests for testing interaction between systems. However, these tests may not reflect the actual circumstances of the real production environment for the following (but not limited to) reasons:

(1) Tests are using incorrect versions of dependent back-end services. For a large system with multiple binaries, each binary may be running at different versions and have different release schedules. Even if a test for a specific feature is passing, it may actually be broken in production because of version mismatches (e.g., a required feature is not yet deployed to production).
(2) The system-under-test (SUT) is not configured identically to the production system (e.g., different flags for features or experiments).
(3) Tests are running with incorrect or unrealistic data. The test data may not exercise the system thoroughly, leading to untested code paths.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
AST '20, October 7–8, 2020, Seoul, Republic of Korea
© 2020 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7957-1/20/05.
https://doi.org/10.1145/3387903.3389318

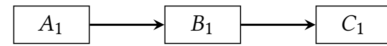

Figure 1: Production environment for a three-tier service

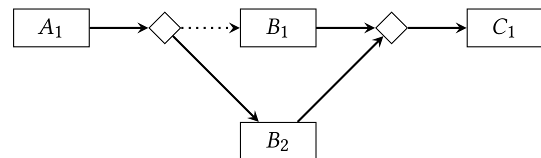

Figure 2: Pre-production environment with routers can divert traffic to different versions of a particular server

To compensate for these limitations, we would like to have accurate *release tests* that can determine if a new release of a server will keep the system running as expected and not break. Take the example three-tier service in Figure 1. If we want to release a new version of server B, say B_2, then we can start up a new environment with the servers A and C running at version 1, and server B at version 2. Server A in the test environment has to be wired correctly to server B_2. We would have to spin up separate environments for releasing server A and C because the wiring would have to be different.

Instead of spinning up a separate environment for every server each time release testing is necessary, we provide a pre-production environment that is a faithful reproduction of the production system, including configuration and data, to overcome the limits listed above.

2 PRE-PRODUCTION ENVIRONMENT

The pre-production environment is a faithful replica of the production system, using the same configuration flags and data sources. To help with testing new release candidates for each server in the system, we put a router in front of each server that can dynamically control where to send the traffic to. In Figure 2, the pre-production environment consists of the servers A, B, C running at their production versions (A_1, B_1, C_1 in this case) and the routers in front of server B and C. In case we want to test a new version of server B (B_2 in the figure), we can tell the router to divert traffic to server B_2 for server B's release test traffic.

By default, requests that go to the pre-production environment go through servers that are the same versions as in production, including data sources. For read only requests, this allows tests to work with real data to examine the system based on real usage. However, it is still up to the developer to write good release tests that

cover a large fraction of use cases that can confidently determine whether the new release is ready for production.

In the next section, we describe how to leverage production traffic to come up with new test cases automatically. For requests that mutate data, we need extra infrastructure, which we will cover in section 4.1.

3 AUTOMATED RELEASE TESTING WITH SAMPLED REQUESTS

Release testing requires a good set of tests to exercise the production system thoroughly. The system will only be tested on the parts that the tests invoke. We need to cover the public API of the system with variations in the method parameters. The tests would be in the form of a request and expected response pair.

Writing tests for every corner case is tedious and time consuming. Furthermore, specifying the expected response for each given request is not an easy task. It would be helpful if we can fully or even partially automate the effort of writing tests.

One way to cope with the problem is to generate the test cases. We can have regression tests set up from the previous N days worth of requests. The current live system would act as the baseline and be used for the expected responses.

Combining the pre-production system with request sampling, we have an automated system for testing whether a new release candidate is fit for production. By sampling from production traffic, we have test cases that will thoroughly cover common use cases. With smarter sampling to bias towards rarer methods and parameter values, we can also increase coverage for the uncommon cases.

We call our system $(AP)^3$: Automatic Prevention of Accidents in Production for AP[1]. When any part of the server stack is ready for a new release, we run the set of sampled requests twice in the pre-production environment: once where the requests go to the live versions of all servers and again while diverting traffic to the new release candidate but keeping the other servers at their live versions. We collect the differences, filter out noise, and create a report that helps the developer determine whether the release candidate is working properly.

4 EXTENSIONS TO THE SYSTEM

4.1 Handling Mutate Queries

Testing for mutates beyond unit and functional testing is difficult for the following reasons:

(1) **Data setup**: Some mutations require the state of an account to be in a particular state for the mutation to be valid.
(2) **Destructive writes**: Mutations may have destructive effects that cannot be easily undone, so we need to be careful about any changes (especially to the production database).
(3) **Repeatability**: Because of the above two, it is not easy to run the same mutation test multiple times. Running the same test multiple times is crucial for version tests or latency tests.

In $(AP)^3$, we address the above difficulties by diverting traffic to a recorder / replayer as needed. For a sampled mutate request, we first run in capture mode to create a test case that can be repeatedly run in testing mode.

[1]AP stands for Advertiser Platform, but $(AP)^3$ can be used for any other product.

In capture mode, we collect all the data that is required for replaying a mutate request. A typical mutate request will read the current state from the data store and possibly make some external calls before writing back to the data store. We capture all the read requests and responses (or mock them if they are irrelevant) made to the external servers and data stores for replaying later. We use a data store that supports reading at an earlier timestamp, equal to when the original query was made, such that we can faithfully recreate the state when the original mutate request was made. Since we are replaying a mutate request and don't want the changes to be made again, we block the write request to the data store such that no data is altered in capture mode.

In testing mode, we replay the mutate request by using the recorded data during capture mode. Any requests that were mocked will be mocked in the same way, and the requests that were captured will be responded with the recorded responses. The final write request to the data store is also blocked but saved for comparison. For version testing, we send the same request to the live version and the release candidate and then compare the two write requests for equivalence.

4.2 Back-end Refactoring and Rewrites

Oftentimes, back-end code needs to be refactored or rewritten with changes in the API. We need to make sure that from the client's perspective, the responses from the new system do not differ unintentionally from the old system. Automatic validation of the new system against the old would make code migrations easier. It lessens the burden of the developer to write tests and check for differences, and it could do a better job of being more thorough than manually written test cases.

Comparing the two systems is very similar to version testing, except that the APIs may be different. We cannot send the same requests to the old and new systems without a translation layer. With a translator that can convert requests from the old system to the new system and convert back the responses from the new system to the old system, we can use $(AP)^3$ to automatically check the new system for functional parity.

5 FUTURE WORK

In $(AP)^3$, we use the live version of the system under test for the expected responses of the requests. This is a good approximation of the expected responses, but does not work well for bug fixes or new features. For these cases, we could use predicates for specifying expected results and still take advantage of the wide coverage brought by production request sampling. We may even be able to automatically infer invariants as in Korat [1].

Another issue with version tests is when there all too many differences in the results. Some of these changes are intentional (bug fixes) and some of them are irrelevant and can be ignored. We can help the developers who need to look at long reports by using machine learning to detect anomalies and only show the differences that need to be checked manually.

REFERENCES

[1] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. 2002. Korat: Automated Testing Based on Java Predicates. *SIGSOFT Softw. Eng. Notes* 27, 4 (July 2002), 123–133. https://doi.org/10.1145/566171.566191