Our Road to Continuous Delivery @ Tango

Amit Mathur

amit@tango.me

Abstract

Tango is a free mobile communications service with 300+ million registered members available on iPhone, iPad, Android phones and tablets. Tango's large user base, broad platform support, frequent releases, and organizational design pose challenges to achieving quality.

As a follow on from last year, this paper discusses specifically the investments that we made in test and deployment automation to enable the transformation from a traditional software delivery model to a continuous delivery model. This new model has enabled us to continue to have agility despite a rapid growth of customers, employees, and functionality.

Continuously releasing software may sound great in theory but is riddled with challenges. This paper will help you learn the tools and techniques we have used during this evolution as well as lessons we learned along the way.

Biography

Amit has over 15 years of experience in software quality assurance as a tester, test architect, test manager, and Director of Technology at Motorola, Veritas, Symantec, Microsoft, and now Tango. He has presented in numerous testing conferences including CAST, Star West, and PNSQC.

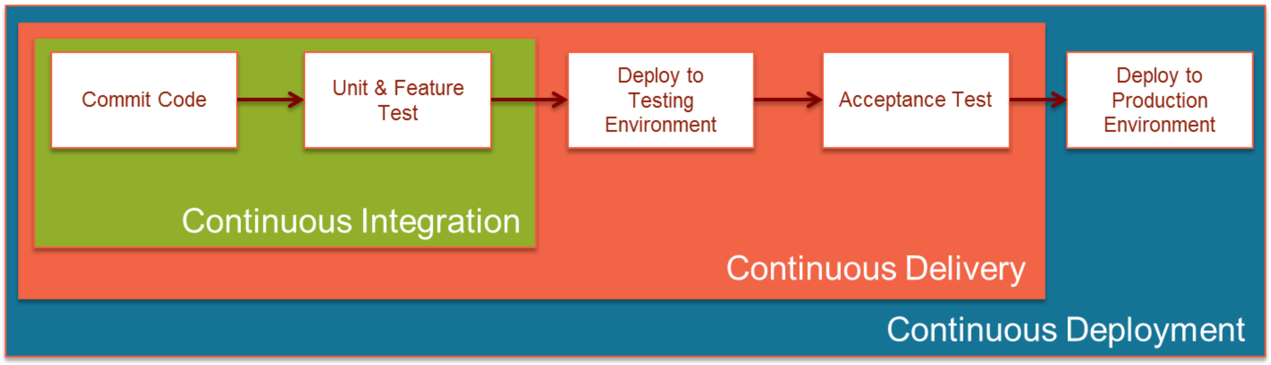
Amit holds a B.S. in Computer Science from the University of Illinois at Urbana Champaign.

1 Introduction

Mobile communications is hot and Tango is at the center of the mobile social communications race. It has been described as more of a sprint than a marathon. This has led us at Tango to converge towards a monthly release cycle.

As if releasing every month isn't challenging enough, we also try hard to give engineering teams a lot of autonomy to get their features out which poses interesting challenges when we attempt to integrate all the pieces together.

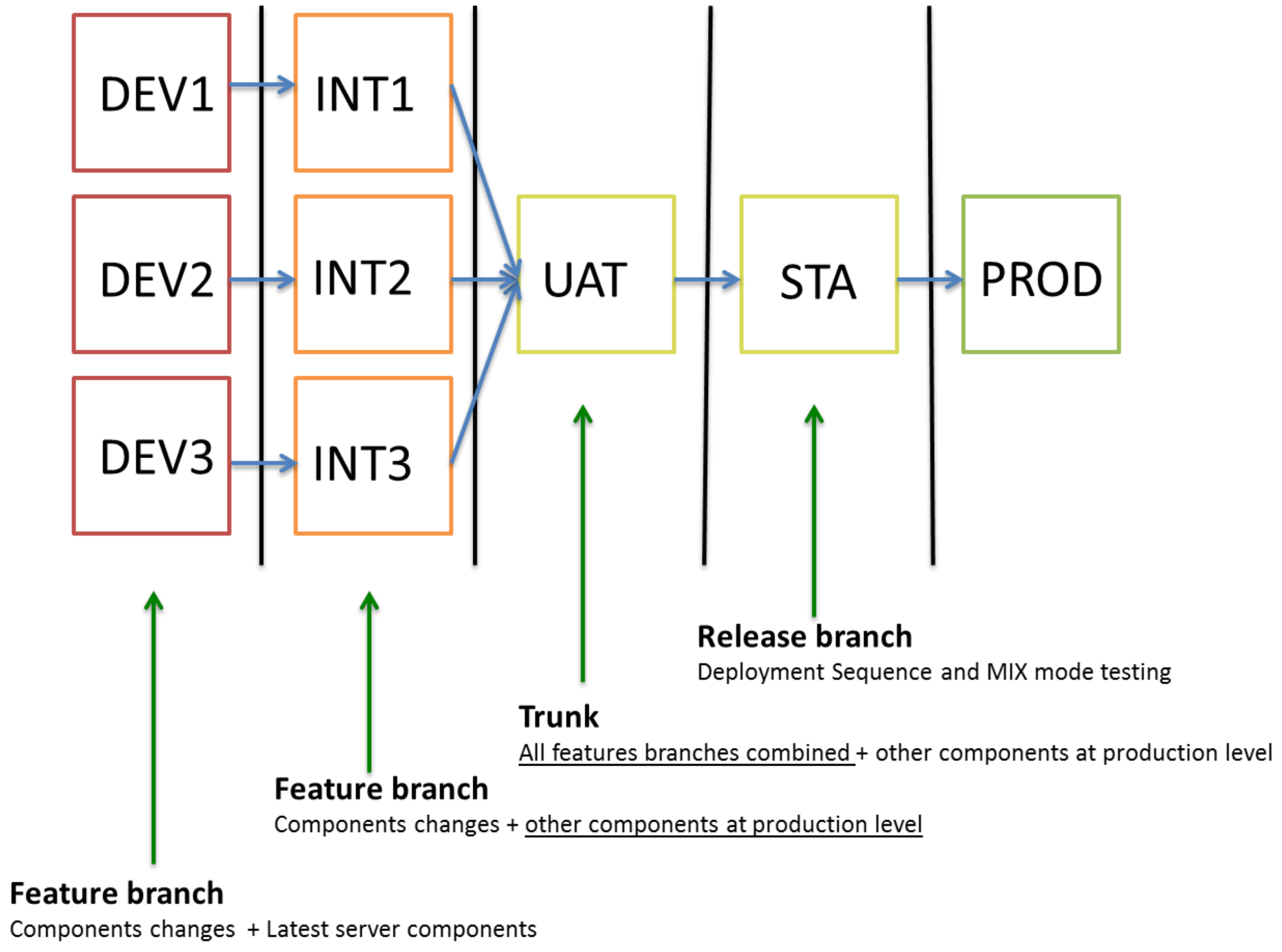
2 Continuous Delivery



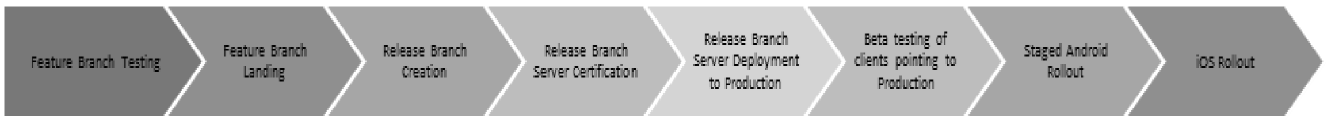
Tango has enjoyed the benefits of continuous integration (automated unit and feature tests) after commits for many years. To continue to increase our velocity we shifted focus to automating our deployment to our integration testing environments and automatically executing acceptance tests.

Our next step will be Continuous Deployment all the way to production! It has been an exciting ride and the rest of the paper will discuss how we implemented the systems the benefits achieved by each.

3 Release Process


DEV1
DEV2
DEV3
INT1
INT2
INT3
UAT
STA
PROD
Release branch
Deployment Sequence and MIX mode testing
Trunk
All features branches combined + other components at production level
Feature branch
Components changes + other components at production level
Feature branch
Components changes + Latest server components

Our high level goal is to release a new Tango client (iOS and Android) every month. To achieve this goal we believe that the best approach is to conduct as much testing in production as possible. For this to be feasible, we need to accelerate getting our server infrastructure into production quickly.


Feature Branch Testing
Feature Branch Landing
Release Branch Creation
Release Branch Server Certification
Release Branch Server Deployment to Production
Beta testing of clients pointing to Production
Staged Android Rollout
iOS Rollout

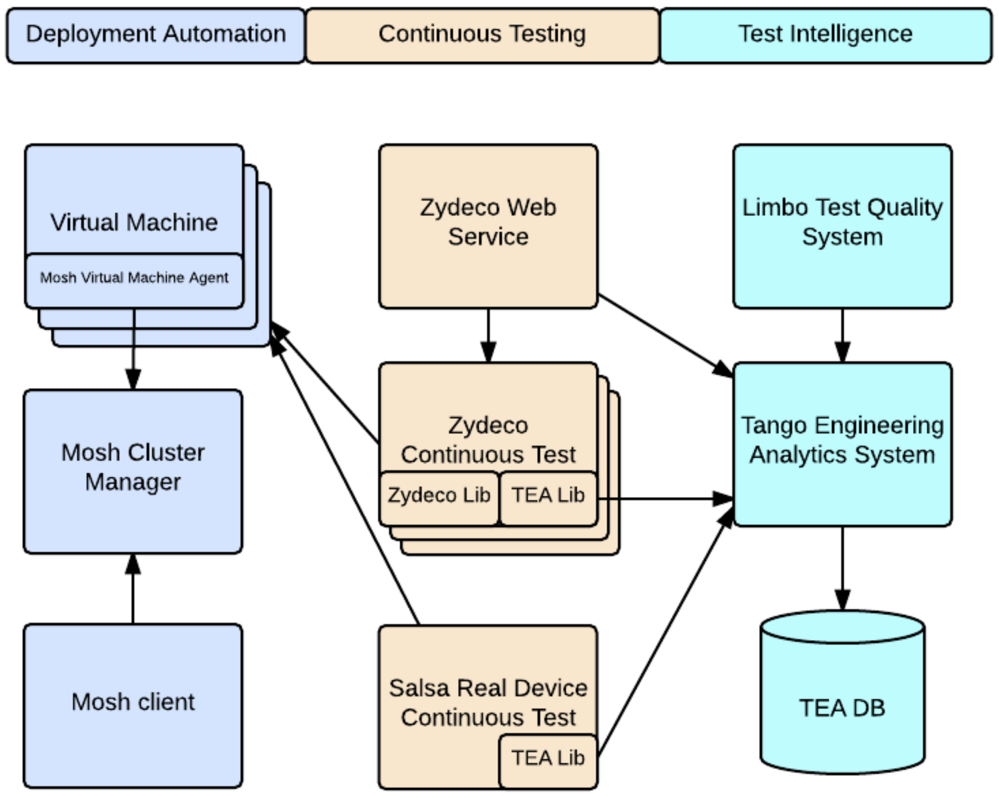
Here is the high level release process:

- Test feature branches
  - Feature branches contain all of the client and server changes for a new Tango feature. The feature branch is tested in a development environment.
- Integrate changes to trunk
  - Once the features are verified in the feature branches the branch is approved to integrate changes to trunk. Once landed to trunk, the changes are deployed to an integration

environment and tested together. All of the features that are targeted for a given release land to trunk in the same way.

- Create release branch
  - Once all of the features for a given release have been landed to trunk and tested in the integration environment, a release branch is created.
- Certify release branch
  - The release branch is then deployed into a staging environment for final certification.
- Deploy release branch to production
  - Once certified the release branch is deployed to production.
- Beta testing of clients pointing to production
  - Once the necessary services are present in production, new clients can now point to production. Tango employees are encouraged to update their clients to the new version for early testing.
  - After a period of internal testing, we release clients to select beta testers outside of Tango.
- Staged Android rollout
  - Android allows us to have a staged rollout for the client. We steadily increase the % of customers that will receive the new client and monitor for any adverse impact to our core business metrics as well as crashes.
- iOS rollout
  - Finally, once Android is 100% rolled out, we release our iOS client. We do this as we want to maintain parity between the client versions and iOS does not allow us to do staged rollouts.

4 High Level System Overview



At a high level there are three different pillars that support our continuous delivery efforts. First and foremost there is deployment automation. We have a **deployment automation** system that can automatically deploy various topologies onto AWS called Mosh.

The second pillar is **continuous testing**. We have several test frameworks that we use to ensure that the code is of high enough quality to release to production.

The third pillar is **Test Intelligence**. This is important as it allows us to determine the state of our tests to ensure that test failures are analyzed in a timely manner.

5 Deployment Automation

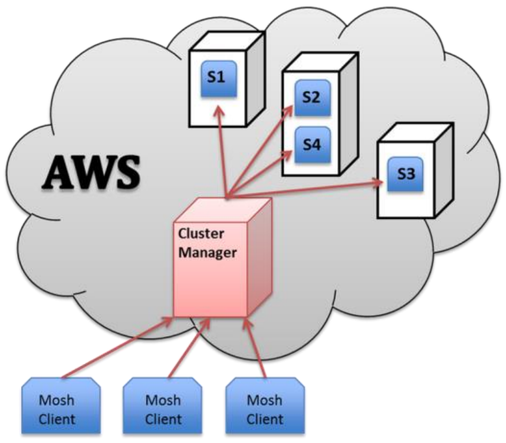
5.1 Mosh

One of the key challenges that we faced as we evolved as a company was that there were multiple different ways that servers were deployed to dev, test, and production environments. This difference was not only inefficient, it also was the cause of many escapes to production as the environment drift surfaced issues in production that were not found in dev and test environments. Traditionally, all of the dev, test, and production servers were hosted in a datacenter. There were many manual steps required to provision virtual machines, install application dependencies, and update application code and configuration.

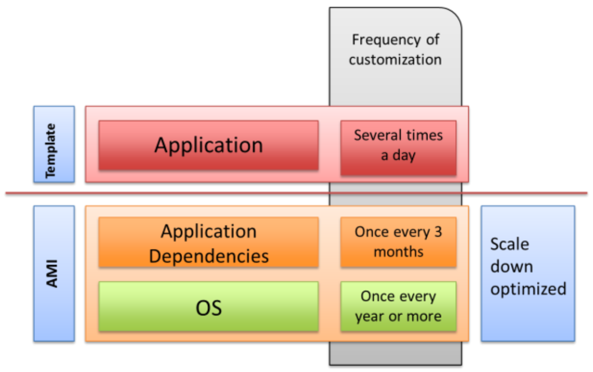
If we were going to release code quicker to production we realized that something had to be done to bring order out of chaos. We decided to build a system that would help us automate these steps. We named this system *mosh* to acknowledge the reality of the state of deployments at that time—like navigating through a mosh pit. Also, we like to name things after different kinds of dances.

5.2 Mosh Layers

Mosh is a distributed system which lives in the Amazon cloud that has a client, cluster manager, and virtual machine agents that run on all managed virtual machines. The client is how the operator interacts with the system. The Cluster Manager is the *brains* of the system and decides where applications should be allocated. The Virtual Machine Agent runs on each virtual machine is the component responsible for updating application builds and updating application configurations.



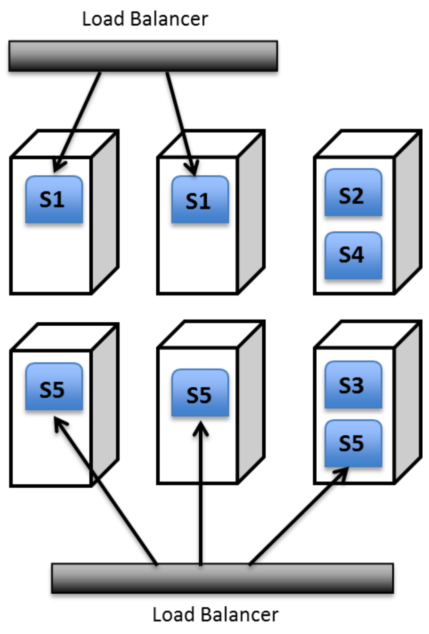
Mosh has two key layers. The first layer contains the operating system and application dependencies. The second layer is the application and application configurations. For us the first layer does not change that often so we decided to include both parts into the base virtual machine image that we use (AMI).



5.3 Balancing environment requirements with cost

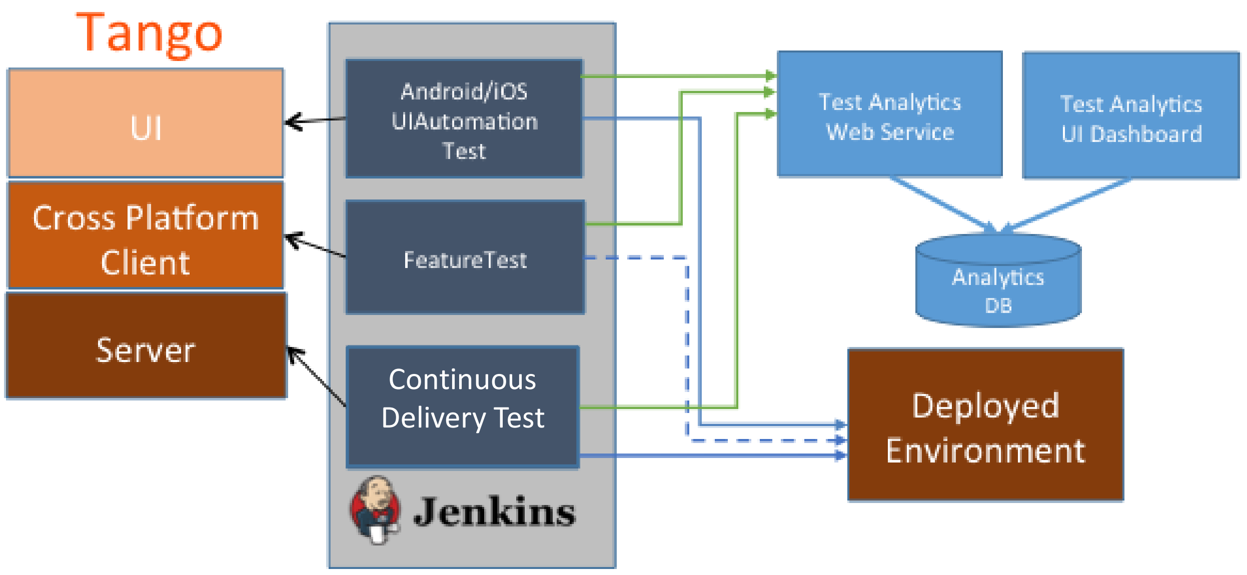
Mosh is designed to allocate services both in a scale out mode and a scale down topologies. Scale out mode (service S1 and S5 below) allows services to have multiple instances allocated behind a load balancer. Mosh also allows services to be deployed in a scaled down mode (service S2, S3, and S4 below).

The reason why mosh supports both topologies is that different environments have different requirements. Our production environment has a requirement for all the server components to be scaled out. Certain testing environments have a requirement for some components to be scaled out to allow for mixed mode (testing n and n+1 server builds in parallel) testing but not all of them. Development environments do not require scale out. The ability to support different topologies allows us to balance our testing needs with cost.



6 Continuous Testing

6.1 Overview



At a high level, we have three test frameworks with different goals. The UI automation frameworks are designed to test the UI layer and below (all layers) for end-to-end coverage. The Feature Test framework is designed to cover the client cross platform layer to server layer. The continuous delivery tests are designed to specifically target servers.

All frameworks are designed so that test cases can be annotated and results sent to a central analytics server to provide a high level view of all tests and the ability to drill down to specific test failures to reduce risk associated with product and test issues.

6.2 UI Automation

Our original UI automation strategy was to leverage Android and iOS UI automation to cover our new client UI during a release. The problem is that our UI changes very frequently (daily) and our automation ends up continuously broken. We decided that instead of spending tons of time during the release fixing the functional tests we would focus our efforts on two fronts:

1. Performance/Reliability – The performance and reliability of our clients are critical as they have a direct impact on the user perception of quality and their engagement. We realized that we simply could not do performance and reliability without automation since it requires running in loops and taking time measurements. Towards the end of the release, we run performance automation for core scenarios as part of the final sign off to ensure there is no performance regression for new clients.
2. Server Regression Validation – Although we get good targeted server coverage via ServerAPI tests and client/server coverage via the Feature Tests that can be targeted towards server environments, we still like to ensure that when we release new server code that our existing production clients do not break. This is accomplished by continuously running the current

production client release against our server environments to ensure that core user scenarios are working end to end. This does not suffer from the problem of client UI constantly changing since we are keeping clients fixed and changing the servers.

6.3 Feature Test Framework

The Feature Test Framework is a hermetic[1] system (self-contained server system with no network connection) utilized by developers to author end to end tests that can be run on their development machine. The system consists of a test client that is a light wrapper on top of the cross platform client layer that contains the bulk of the client code that gets released with the exception of the user interface layer.

In addition to unit tests, developers are required to author feature tests to ensure that they have end-to-end coverage. The key benefit of feature tests is that developers can understand the impact of changes throughout the ecosystem on their components. For example if they have a dependency on a downstream component that changed, they may witness their feature tests breaking as a result and quickly get the issue resolved with the offending party.

Finally, a subset of feature tests (indicated via the dashed line above) can point to deployed environments. They serve as tests for environment stability and some of them are turned into production server monitors. They serve as good monitors since they simulate user behaviors very well as they contain the key parts of the clients that users actually use.

6.4 Continuous Delivery Tests

Continuous Delivery Tests are test cases that specifically target direct server interactions. As our services are RESTful, the tests perform http requests and verify the correct http responses are received.

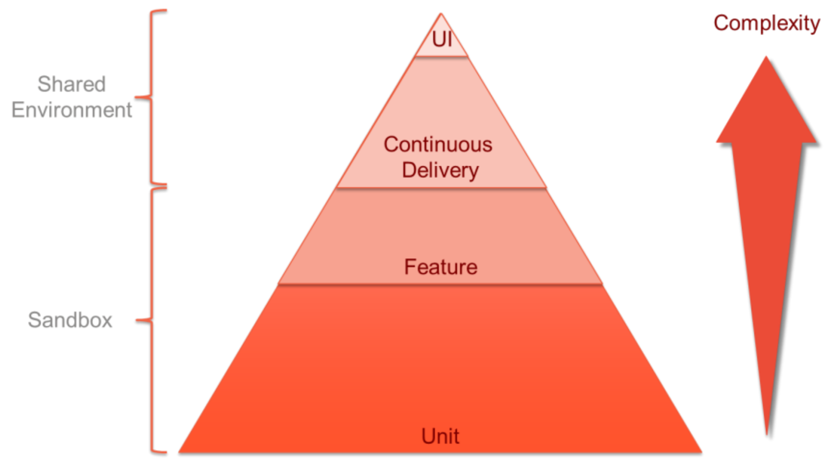
The idea behind having targeted server tests is that they can be developed and executed without the client being present. In particular, it allows us to target servers directly that only interact with other servers.

Here are the requirements for continuous delivery tests:

- Environment Mobility -- Should be able to run in any environment including localhost.
- Non-intrusive -- Should clean up all garbage created by the test and should not interfere with other test instances (including itself).
- Security -- Must not expose environment secrets as test could be run in production environment.

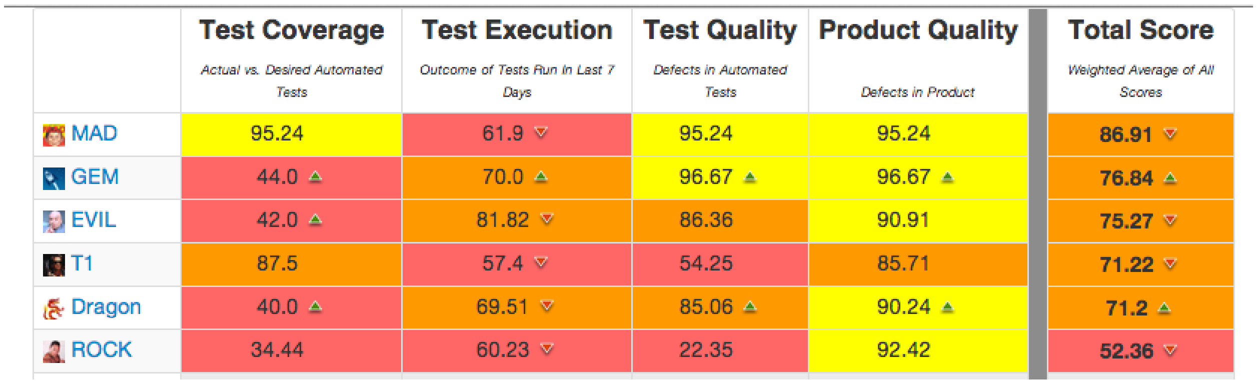
One of the key realizations we had when defining continuous delivery tests is that since they exercise a relatively small amount of the server code there need to be many more targeted tests that have been run already before these get run. We tell our development teams that they need to think in terms of the testing pyramid. Without solid unit and feature tests, the continuous delivery tests are not very useful since there is a high risk that something is broken that the continuous delivery tests did not exercise. When combined together they are very powerful as we get increased confidence as the code flows through different environments.

The testing pyramid below depicts visually the quantity of expected tests of each type. As we get higher in the pyramid the complexity of the test increases since it is exercising a larger area of the system. One key to continuous delivery is to have enough high quality tests that can be run against a shared environment instead of being constrained to run only in a sandbox.



7 Test Intelligence

7.1 Team Testing Dashboard



	Test Coverage *Actual vs. Desired Automated Tests*	Test Execution *Outcome of Tests Run In Last 7 Days*	Test Quality *Defects in Automated Tests*	Product Quality *Defects in Product*	Total Score *Weighted Average of All Scores*
MAD	95.24	61.9 ▽	95.24	95.24	**86.91** ▽
GEM	44.0 ▲	70.0 ▲	96.67 ▲	96.67 ▲	**76.84** ▲
EVIL	42.0 ▲	81.82 ▽	86.36	90.91	**75.27** ▽
T1	87.5	57.4 ▽	54.25	85.71	**71.22** ▽
Dragon	40.0 ▲	69.51 ▽	85.06 ▲	90.24 ▲	**71.2** ▲
ROCK	34.44	60.23 ▽	22.35	92.42	**52.36** ▽

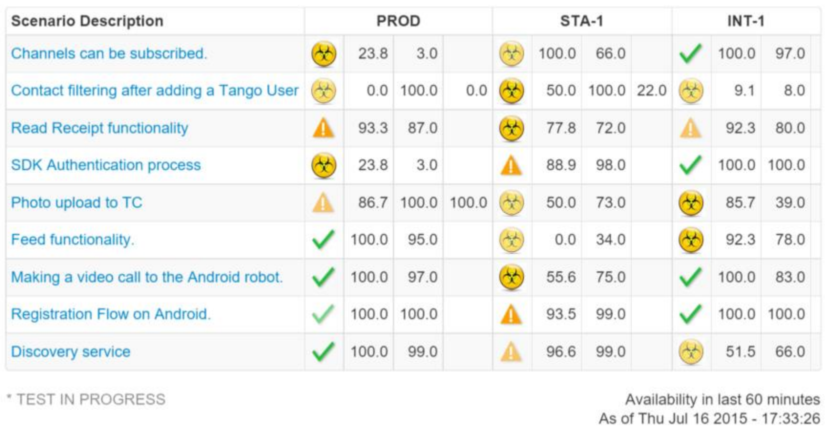
We want the vertical teams to 'own' the quality of what they built so we built a Test Testing Leader board which helps the entire company visualize where each team is at with their test automation (coverage, what was being run, product issues found, and test issues).

The idea is to gamify testing so that teams feel motivated to execute and keep their test automation clean. This encourages a bit of healthy competition between teams. All test automation is run continuously via our Jenkins continuous integration system and fed in real time to this dashboard.

One nice element of the dashboard is that it is updated in real time and provides a stock ticker like up/down symbol providing a continuous feedback loop to test teams.

7.2 Test Quarantine

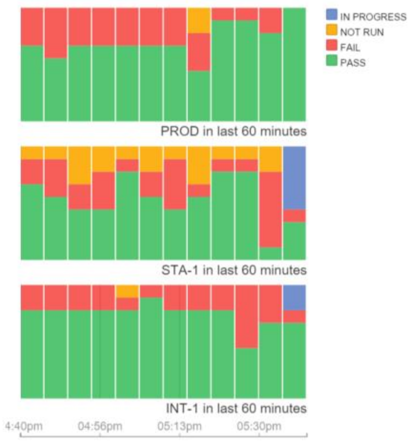
We also have a server component by component status that indicates the health of each component as well as a quarantining system that blocks certifications if tests are not analyzed of fixed in a timely manner. A test gets quarantined (depicted by the toxic icon below) when it has had a sufficient amount of unanalyzed failures or test issues. This allows us to distinguish between issues where we have a high confidence and those that we have lower confidence.



Scenario Description	PROD				STA-1				INT-1		
Channels can be subscribed.	☣	23.8	3.0		☣	100.0	66.0		✓	100.0	97.0
Contact filtering after adding a Tango User	☣	0.0	100.0	0.0	☣	50.0	100.0	22.0	☣	9.1	8.0
Read Receipt functionality	⚠	93.3	87.0		☣	77.8	72.0		⚠	92.3	80.0
SDK Authentication process	☣	23.8	3.0		⚠	88.9	98.0		✓	100.0	100.0
Photo upload to TC	⚠	86.7	100.0	100.0	☣	50.0	73.0		☣	85.7	39.0
Feed functionality.	✓	100.0	95.0		☣	0.0	34.0		☣	92.3	78.0
Making a video call to the Android robot.	✓	100.0	97.0		☣	55.6	75.0		✓	100.0	83.0
Registration Flow on Android.	✓	100.0	100.0		⚠	93.5	99.0		✓	100.0	100.0
Discovery service	✓	100.0	99.0		⚠	96.6	99.0		☣	51.5	66.0

* TEST IN PROGRESS

Availability in last 60 minutes
As of Thu Jul 16 2015 - 17:33:26



8 Conclusion

Based on these (and other) test investments, we have been able to scale up the company dramatically while steadily improving the quality of our client software and servers. We still have a lot more work to do to ensure that nasty bugs don't escape to production but are happy that we are fully focused on continuous delivery to help us achieve our goals.

References

1. Hermetic Servers, Chaitali Narla & Diego Salas, http://googletesting.blogspot.com/2012/10/hermetic-servers.html