Dilemma of Speed vs. Scale in Software System Development

Best Practices from Industry Leaders

by

Kshitij Kumar

B.E., Metallurgical Engineering, Indian Institute of Technology, Roorkee, India, 1999

M.B.A., University of Delaware, Newark, DE, USA, 2005

Submitted to the System Design and Management Program in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Engineering and Management at the

Massachusetts Institute of Technology

February 2017

Signature of Author Signature redacted

Kshitij Kumar

Fellow, System Design and Management Program

January 10, 2017

Certified By Signature redacted

Michael A M Davies

Senior Lecturer, System Design and Management

Thesis Supervisor

Accepted By Signature redacted

Warren P. Seering

Weber-Shaughness Professor of Mechanical Engineering and Engineering Systems

Massachusetts Institute of Technology

This page intentionally left blank

1. Abstract

Many startup organizations face a dilemma as they scale up and their systems grow more complex. This dilemma is between the speed of releases i.e. agility and the scalability of their systems, reflected in the performance, stability and maintainability of their systems as they become larger. A startup is typically very nimble and releases new features and updates to its product very quickly. However, as a startup grows bigger the frequency of releases typically tends to go down. A similar phenomenon is observed in case of incumbent organizations, those that are old, large and complex, and that already have systems at scale; they have systems that perform at scale, and are stable and maintainable, but the pace of development is slow, and it find it hard to speed up their release cycles. Through the study of organizations that have successfully reconciled the required and coveted scalability along with speed as these organizations moved from being small startups to become larger, this study demonstrates that speed vs scale is a dilemma for startups that can be reconciled as they scale up, because there are a set of practices such as modular architecture choices, minimizing work in progress by adopting frequent deployments, automation in testing and utilizing innovative management techniques, which can enable startup organizations to scale up their system and maintain high speed. Although the study of incumbents as they try to speed up while maintaining their scalability is outside the scope of this work, this study also presents a hypothesis for further investigation that it may also be possible for incumbent organizations to speed up at large scale, by embracing these practices.

Thesis Supervisor: Michael Davies

Title: Senior Lecturer, System Design and Management

2. Acknowledgements

First of all, I would like to thank my thesis advisor, Michael Davies, who, despite his very busy schedule, always took time to guide me throughout the process, starting from the selecting and refining the thesis topic, research questions, analysis and approach to the final completion. It has been an enlightening journey and I have learned a lot from Michael.

To Pat Hale, for giving me the opportunity to come to MIT and his constant support, Bill Foley and SDM staff for their help all along.

To several faculty members and researchers at MIT School of Engineering, MIT Sloan School of Management, Harvard Business School and Harvard Kennedy School of Government, whom I have associated with and learned from in last few years. I have benefited immensely from their wisdom.

To my friends, my class mates, my colleagues and practitioners in industry who have taken time out of their busy personal and professional lives to hold countless discussions that helped me clarify and solidify many ideas.

And finally to the people whom I don't name here (you know who you are), I know that I could not have made it without your support.

Contents

Table of Figures

3. Motivation

Prior to joining the System Design and Management program at MIT, I had worked for about 12 years in software services and software product development projects in various capacities, ranging from engineering to product management and project management. Over these years, I had worked or consulted with more than 10 software development organizations or the organizations that had information services as backbone of their business processes and functions. These organizations ranged from early stage startups to Fortune 100 companies.

Over my professional career I had observed that usually there was a perception that the quality of the product or architecture suffers if the releases are rushed and vice versa. In the initial phase of a startup, the release cycles are fast, as the need to validate the idea and get early feedback is paramount. Based on this validation, either further investment rounds are raised or the team pivots to a different idea.

However, as the system matures and scales, usually the release cycles slow down. There is an understanding that the development needs to become more deliberate to ensure the quality and scalability of the system. QA also now needs to spend an enormous amount of time to ensure that bugs don't slip into production. All this slows the pace of development.

However, surprisingly even after this, typically architecture suffers, technical debt creeps in and these further slow the release cycles as it becomes progressively harder to add new features or manage the growing code base. Fixing one area could cause unintended effects in other parts of the system. Eventually, this all has an adverse effect of the morale of the employees who find themselves spending a lot of time is fixing issues all over the place instead of putting in new features.

While many organizations and practitioners held this perception of a dichotomy between speed and scale, after my discussion with Michael Davies, I realized that there are some organizations, such as such as Netflix, IMVU, Spotify or Facebook, that have been able to continue to scale and be fast, even as they grow enormously. These organizations have scaled very fast over last decade and are still growing. These organizations have demonstrated that the dilemma between the speed of development and the scale of the systems may be a dilemma that can be reconciled, and there might be some repeatable practices that help achieve speed with scale. This is why I wanted to look deeper into the practices of these organizations to find out a set of best practices that other organizations can use to maintain speed and quality architecture as they scale up and grow bigger. I hope that this study can provide a preliminary set of best practices that may help continue to release quickly as startup companies grow and their systems scale. I also hope that this study acts as a starting point to further investigation to see if these practices remain valid for incumbents which are trying to speed up while maintaining their scale.

4. Thesis Work Plan

The goal of this work is to study the software development practices and related organizational practices at Facebook, Spotify, IMVU and Netflix and delineate the lessons learnt to produce a set of best practices that could prove useful to organization as they try to scale up or as they try to become more agile. I have employed document/research survey, case studies, subject matter expert interviews and comparative analysis to conduct research, gather and analyze information and determine conclusion. The thesis is divided into three major parts:

- Case Studies
- Analysis and discussion of effective practices
- Conclusion

5. Case Studies

This section consists of the four case studies of organizations that have successfully scaled up while maintaining speed. Each case study investigates one organization for practices which enable that organization to reconcile the dilemma between the speed of development and the scale of the systems. The following are the four case studies in this section.

- Case Study 1- Facebook
- Case Study 2- IMVU
- Case Study 3- Netflix
- Case Study 4- Spotify

5.1 Case Study 1- Facebook

5.1.1 About Facebook

Facebook is an online platform for social media and social networking based in Menlo Park, California, United States. The Facebook website was launched in early 2004 and today, Facebook is available on a large range of devices from desktops, laptops, tablets, and smartphones as a website or as an application over the Internet and mobile networks. As of September 2016 Facebook had 15,724 employees, 1.79 billion monthly active users, with 1.66 billion mobile monthly active users, of which 84.9% of daily active users are from outside the US and Canada. [1]

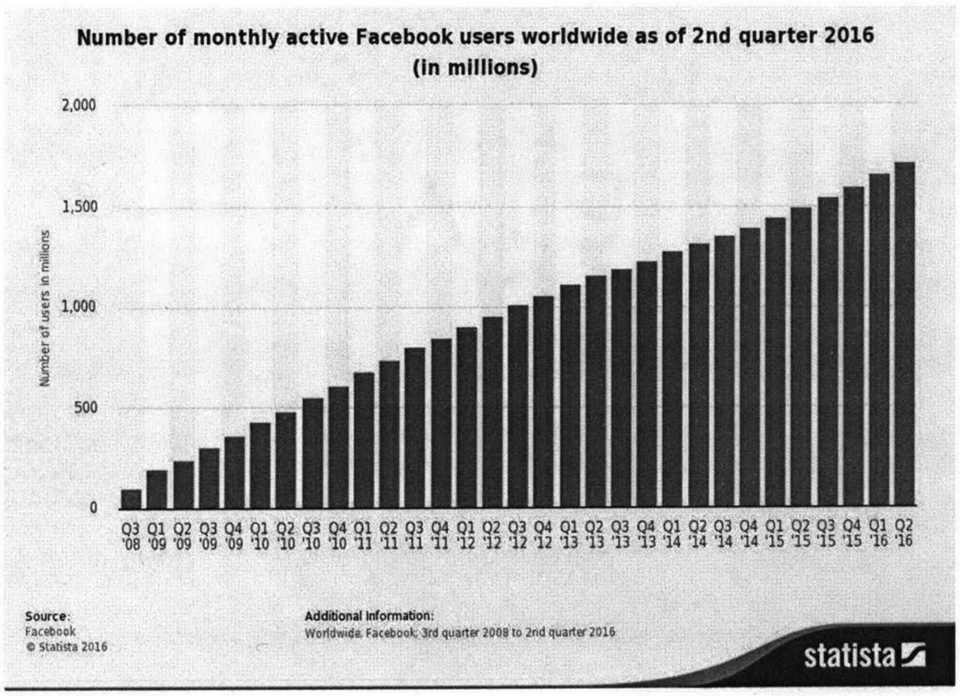


Figure 1 Growth of Monthly Active Facebook Users worldwide 2008-2016 [2]

5.1.2 Continuous Bite Size Deployments

The most important development characteristics of Facebook are speed with growth [3]. At Facebook the front end of the website is continuously developed and updated by engineers, many hundreds of time a day, involving thousands of files. This is true even at the large scale of the system. This is evident from the charts below that show that as the code base size of Facebook has grown faster than linearly, the commits per month have also kept pace and are now in thousands per month.

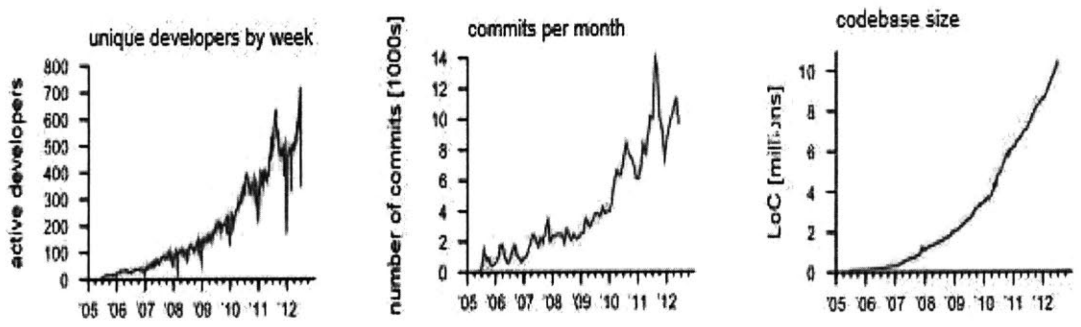


Figure 2 Illustration of code commit speed and codebase size at Facebook form 2005-2012 [3]

Facebook uses continuous deployment because of the benefits of deploying small and frequently. [3] One of the main benefit of many frequent and small deployments is that it not only reduces the likelihood of something major going wrong with deployments, it also makes it relatively harmless to pull out the small chunk of offending code. Since each deployment push only includes a small amount of code, it becomes easy to quickly debug and fix the problem. This is especially true since not only the changes are small but also because the engineer who had developed and deployed the code has done it fairly recently and hence it would be still fresh in the mind of that engineer so making the change or fixing the bug in that piece of code becomes far easier.

Also, using small, frequent and independent pushes reduce the possibility of creating unnecessary dependence in the codebase. This allows for a system with very low level of technical debt accumulation, if any. These low levels of technical debt accumulation help create a system that is not only architecturally robust, but is also easy and quick to modify, maintain and add new features to.

Another major advantage of doing faster and smaller releases at Facebook is that it facilitates almost real-time user feedback on the deployed features and Facebook can course correct as it moves along. This is because the small changes to a new feature or a change can go live quickly, so these changes can be validated quickly via A/B testing. This saves time, effort and cost spent in developing a feature that customers may not want. [3]

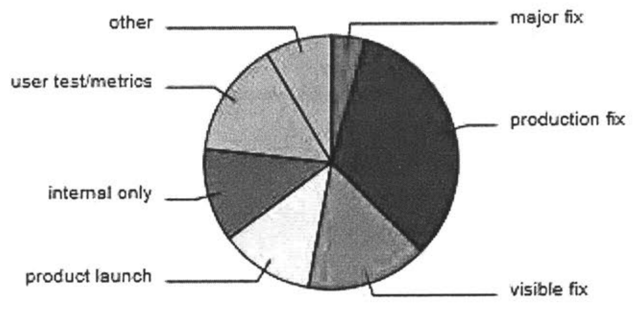


Figure 3 Reasons for daily push at Facebook [3]

At Facebook engineers work out of a single stable trunk of code, and there are no long living branches since the code is deployed in the main branch quickly. This makes the merging the code to the main branch (using Subversion) easy. This practice also encourages the engineers to push code rapidly into main line and this shortens the time between successive commits by the engineer, thereby speeding up releases. [3]

Facebook's strong commitment to frequent deployment is evident from the fact that it trains new Facebook employees in the initial six-week boot camp about the practice of frequent deployment. In fact, new developers at Facebook are encouraged to commit and release new code into the code base, as soon as possible to overcome the fear of releasing in production. This promotes innovation as developers develop an ability and mindset of fearlessness towards frequent deployment. [3]

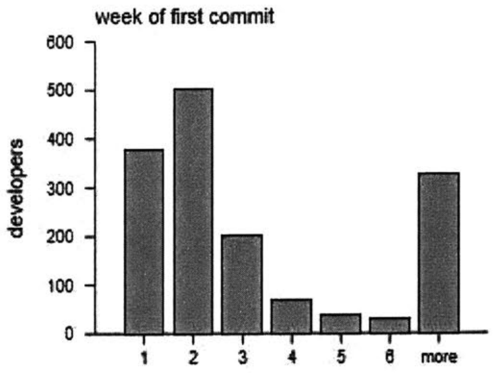


Figure 4 Time to first commit for Facebook boot campers [3]

It is evident from the chart above that there are a significant number of new engineers in the boot camp who commit code within first 3 weeks of the boot camp. This experience, while scary sometimes, inculcates the culture and expectation of frequent commits, and does so especially for those engineers who come from an organization with slower and more deliberate releases.

One of the challenges faced at Facebook in continuous delivery was to deliver features that require large changes in the code base. Facebook mitigated this problem using a practice called "branch by abstraction" [3]. In this practice Facebook deploys large changes that are too big to be deployed via continuous deployment by breaking the large change needed into many smaller chunks, and then hide these changes behind abstraction.

An example provided by Feitelson et al. [3], in their paper Development and Deployment at Facebook is as below:

"Consider the delicate issue of migrating data from an existing store to a new one. This can be broken down as follows:

1. Encapsulate access to the data in an appropriate data type.

2. Modify the implementation to store data in both the old and the new stores.

3. Bulk migrate existing data from the old store to the new store. This is done in the background in parallel to writing new data to both stores.

4. Modify the implementation to read from both stores and compare the obtained data.

5. When convinced that the new store is operating as intended, switch to using the new store exclusively (the old store may be maintained for some time to safeguard against unforeseen problems).

Facebook has used this process to transparently migrate database tables containing hundreds of billions of rows to new storage formats.

In addition, deploying new code does not necessarily imply that it is immediately available to users. Facebook uses a tool called "Gatekeeper" to control which users see which features of the code. Thus it is possible for engineers to incrementally deploy and test partial implementations of a new service without exposing them to end users".

5.1.3 Process of Continuous Deployment at Facebook and Pipeline Tooling

At Facebook, every unit of code which is a potential candidate for weekly or daily push to deployment, has to go through a continuous deployment pipeline. This pipeline is fitted with a set of tools that help make the process of deployment not only easier and less time consuming for the engineers, but these tools also help to rid the process of inadvertent human mistakes. These tools also provide feedback on the state of the deployment process and system and this alerts the team if something has a potential to cause issues so that it can be fixed before the problem actually arises.

The process of weekly push happens in a set of iterative steps/stages. Please see the diagram below and the detailed steps that follow for an illustration of the push process as per Feitelson et al. [3]

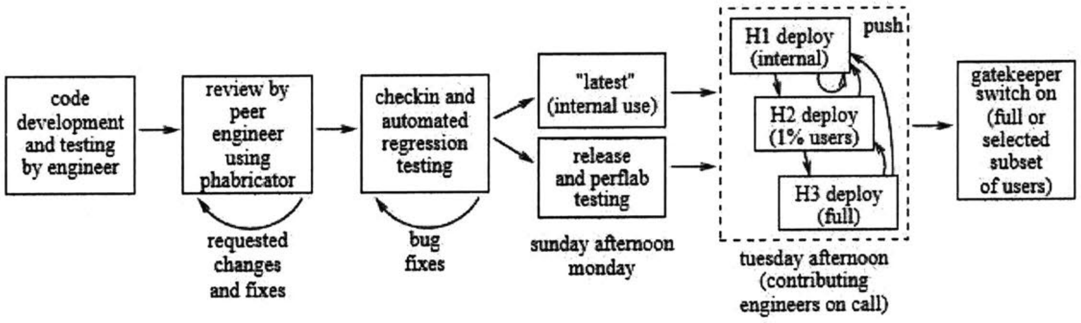

code development and testing by engineer
review by peer engineer using phabricator
requested changes and fixes
checkin and automated regression testing
bug fixes
"latest" (internal use)
release and perflab testing
sunday afternoon monday
H1 deploy (internal)
push
H2 deploy (1% users)
H3 deploy (full)
tuesday afternoon (contributing engineers on call)
gatekeeper switch on (full or selected subset of users)

Figure 5 Facebook's version of their deployment pipeline [3]

5.1.3.1 Development, Unit Test and Peer review

At Facebook, the engineers own the responsibility to unit test their code and get it peer reviewed before pushing it further down the deployment pipeline. Peer review is a process in which a different engineer reviews the code. Facebook uses a code review tool called Phabricator to help with these code reviews. Phabricator allows the developer to not only see current and historic code but also facilitates communication around proposed code changes. This also enables bug tracking to fix the problems discovered and also document the issues on wiki.

5.1.3.2 Testing

Once unit tests and code reviews are successful, a set of regression tests are run on the code being pushed. These are automated tests that simulate user interactions with the system and use tools such as Watir and Selenium(WebDriver). This is done along with the Facebook's practice of dogfooding (using their own system), which uncovers any possible edge case scenario that would not have been caught by the automated testes. Also, apart from the testing the functionality and code behavior, Facebook uses a monitoring tool called "Perflab" to see how the performance on the production servers will be affected by the new code change. This information is important as even small performance issues, if left unaddressed, accumulate into significant performance issues. Using Perflab, engineers are alerted of such potential problems in advance, and take corrective action to either fix the code or remove it from the push.

5.1.3.3 Deployment

Once the testing (unit, regression, dogfooding and performance) has been completed, the code is deployed on a set of Facebooks internal servers (H1 in the diagram above) that are only accessible to Facebook engineers. This enables one more round of testing before going live. Once the deployment and testing at H1 succeeds without problems the code is now deployed to a set of few thousand servers (H2 in the diagram above) that expose the changes to a small subset of real world users. This is done so that if any problems are discovered at this stage, then the exposure will be limited. Once the deployment to H2 goes smoothly, then the code is finally deployed to the whole set of servers (H3 in the diagram above), and exposed to the full set of intended real world users. If something goes wrong at any of the stages the code is rolled back to the previous stage and the offending code is pulled out along with any other code that is dependent; rare, because of small pushes.

Once the deployment to H3 is successful, the code is propagated across all the servers using BitTorrent technology. While the new code is being deployed, the existing version keeps running till the deployment completes and then the code is switched to the new version, this is done to avoid any downtime during deployment.

In addition, Facebook uses internal IRC chat to communicate with all the developers whose code is deployed as a part of that particular push. As a policy, the developer must be available on the IRC at the time of deployment or his/her changes will be pulled out and not deployed. This is done so that the engineer is available to assist in case of any unforeseen issues that may arise in his or her part of code.

After the deployment is complete, Facebook continues to monitor the health of the system to detect any signs of problems. Facebook uses tools such as Claspin [4] to monitor system health. In addition, Facebook uses other methods such as tweet analysis to detect any problem that may have escaped the earlier quality checks.

5.1.3.4 Exposing deployed code in production in a controlled fashion

Since the code that is deployed contains all the changes (and possible A/B tests) the changes would need to be deployed selectively for the tests to be effective. For example, if a test is intended to check the effectiveness of a new feature on those American teens aged between 17-19 who live in the Northeastern United States, then that piece of code should be deployed in such a way that only intended users see that feature. Facebook manages this selective exposure of deployed code via a tool called Gatekeeper. Gatekeeper not only allows engineers to do dark launches, where even though the code is deployed, it has the interfaces turned off so that it is not visible to the users.

Gatekeeper also enables engineers to selectively turn off or on pieces of code that may be causing problems. [3]

5.1.4 Alternative ways to QA/Dogfooding

To ensure the quality of the platform without an extensive QA department is a challenge. This is why Facebook has a QA strategy that includes approaches from the usual unit testing and peer reviews, to automated testing, to innovative dogfooding techniques and bounty programs to catch bugs. These QA methods ensure that Facebook's systems are sturdy and meet customer needs as intended.

As per Feitelson et al. [3], at Facebook, engineers own the responsibility for unit testing their own code before it is considered a candidate for daily or weekly push to deployment. Once the code is unit tested and found ok, it is peer reviewed with another engineer to ensure that the code is free of bugs. This practice of peer review encourages the engineers to be extra careful in writing their code and maintain higher quality as someone else is going to look at it, Also the fresh perspective of a new person sometimes helps discover issues in a developed feature that remain invisible to the engineer who developed that feature.

Beyond the unit test and peer reviews, Facebook maintains a suite of automated regressions tests that are run frequently to discover bugs. Facebook uses tools such as Watir and Selenium to simulate user interaction with the system and test it. This test harness is continuously improved as new code is written or new items are needed, or are discovered to be tested.

Even after the unit tests, peer review and extensive automated testing, there might be some edge cases that may slip past these gates. In order to catch these bugs Facebook needs real humans to test the platform. To achieve this Facebook turns its employees into users. Facebook deploys the latest code as internal build that is used by its employees. It is called 'dogfooding'. i.e. eating your own dogfood [5] . This internal build is used like a regular Facebook platform by Facebook employees as users. This use by the employees of Facebook tests the platform extensively and in as realistic condition as possible. This internal Facebook build and testing environment also has an integrated bug reporting tool to identify and track issues easily. This way Facebook also performs extensive human driven testing of the system without the use of an extensive QA department. [6]

Facebook also involves its real users in testing and finding bugs and usefulness of newly deployed features in its platform. One of the ways is performing A/B testing. Facebook uses tools such as Gatekeeper to release selectively to segments and perform A/B testing. Since Facebook has a very large user base and it is a platform on which the deployment or modification is fully controlled by Facebook itself, it becomes easy to conduct A/B testing and gather user response and feedback data to see if the feature is useful or not. [3]

Finally, Facebook also uses crowdsourced testing in the form of Bug Bounty program. In this program members of general public can find a report bug or security hole in Facebook platform and gain monetary rewards and recognition. Per Facebook:

"Facebook Security's Bug Bounty program provides recognition and compensation to security researchers practicing responsible disclosure." [7]

Facebook issued "White Hat" [8] debit cards to the users/researchers who found security flaws and reloaded these cards with rewards when they discovered new bugs. These cards also act as a symbol of recognition because finding these flaws and bugs were rare and prestigious. Per Ryan McGeehan, a manager at Facebook security team, said in an interview with CNET:

"Researchers who find bugs and security improvements are rare, and we value them and have to find ways to reward them, having this exclusive black card is another way to recognize them. They can show up at a conference and show this card and say 'I did special work for Facebook.'" [8]

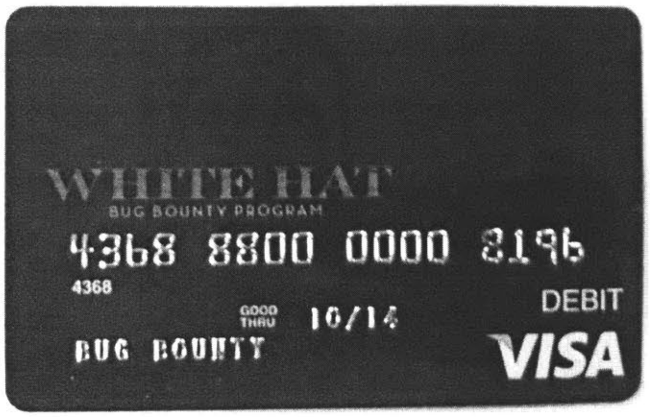


Figure 6 Facebook "White Hat Bug Bounty Card" [8]

Facebook received more than 9,000 bug reports through this channel in the first half of 2016 (Jan-June 2016), and paid a total of $611,741 to 149 researchers for this period. Facebook had paid more than $5 million to more than 900 researchers cumulative over the five years' period since this program started in 2011. [9] This is an innovative and useful way of catching bugs that may have slipped past all the quality control measures Facebook has implemented.

5.1.5 Culture of Accountability and DevOps

As per Feitelson et al. [3] Facebook encourages a culture of accountability and personal responsibility. As discussed in the Innovative QA practices section of this case, there is no extensive QA team at Facebook and Engineers are responsible for each piece of code they write. Although there are tools and system in place to help, such as automated testing and various tools that monitor and catch problems before they happen, or practices such as peer review and dogfooding, but still the final responsibility of the work lies with the person who wrote it. Engineer's unit test their piece, follow the code as it flows through the pipeline to the deployment and then they support the code after it has been deployed in production for any issues that may arise. This is similar to the DevOps paradigm where development and operations work together to support the system as opposed to traditional waterfall model in which developers develop and toss it over the wall to QA, QA tests and its goes into productions and then onwards operations support it. In the Facebook paradigm, the developer's accountability to make sure that their code runs smoothly, motivates developers to create great quality code and prevents serious bugs slipping into the production.

Another example of Facebook's culture of accountability and personal responsibility, as described by Ryan Paul in his article about his discussion with Chuck Rossi, the head of Facebook's release engineering team, is the use of "Karma Score" at Facebook [10]. During deployments, every engineer whose work is being deployed as a part of the push should be personally available on IRC chat to resolve any unforeseen issues. If an engineer's work breaks the build and needs to be taken out of the build, that engineer gets a bad karma score. These karma scores are used as a heads-up metric during code review and push process. If an engineer with low karma score pushes code to be deployed, his or her low karma score alerts relevant people to pay more attention during the code review to make sure the code is bug free and ready to be moved to the next stage. These practices have inculcated a culture of paying attention to the long term quality of the code and not just pushing the code to the next stage. This helps in creating a high quality system that is stable at scale.

5.1.6 Summary

Facebook has grown at a very high speed form its founding in 2004 to about 1.79 billion monthly active users all over the world as of Sep 2016. Growth of this kind presents unique challenges from a system and organization perspective. Facebook has managed this massive growth with a set of practices such as a culture of accountability and empowerment, frequent and small deployments, automation of its deployment pipeline, staged releases and innovative ways to test its releases. In addition to the automated testing, A/B testing etc., Facebook employs "dogfooding', eating your own dogfood" technique, i.e. employees using pre-production Facebook platform with new release candidate features, internally across Facebook. This uncovers problems that users may have faced if the release was deployed in production. Beyond dogfooding, Facebook uses the world at large as its testers through its innovative "Bug-Bounty" program to catch bugs that may slip into production its number of checks. In summary, Facebook is following the following set of practices that enables it to maintain growth with speed.

- Frequent and, small deployments
- Automation of delivery pipeline-tooling
- Controlled releases-use of Gatekeeper to stage releases
- Innovative Quality Assurance- Dogfooding, Bug-Bounty program
- Culture of accountability and empowerment - new engineers encourages to commit quickly, use of karma score

5.2 Case Study 2 - IMVU

5.2.1 About IMVU

IMVU, Inc., is an online metaverse and social entertainment platform. IMVU's platform enables its members to meet people, communicate, and play games by creating and using virtual 3D avatars . IMVU was founded in 2004 by Eric Reis and is referred to as the original lean startup and its founder Eric Reis is known as one of the founding champions of lean methodology.

As of Dec 2016, IMVU has more than 50 million registered members, 10 million unique visitors per month and three million monthly active users. IMVU has a virtual goods catalog of more than six million items with new additions of approximately 7,000 new items every day - most of these items are created by IMVU members. IMVU is located in in Redwood City, CA, and has 120 full-time employees. [11]

5.2.2 Culture of Lean Startup-Past and Present

IMVU is known as one of the original and leading practitioners of the lean startup methodology championed by its founder Eric Ries. When IMVU had started, it found a product market fit by doing small incremental feature releases of Minimal Viable Product (MVP) and continuously iterating. However, IMVU soon faced significant challenges as it tried to scale up. While the tradeoff between faster releases and less focus on architecture quality was not a big issue in the beginning stages, as time went on this pileup of technical debt started hindering the ability to quickly iterate. Revenue was flattening while the cash burn rate was going up, and employee morale was also dropping. While at IMVU anyone could quickly push a new feature in the production to validate with real customers, it was soon observed that the product started to look like a "bucket of bolts", comprised of many unfinished and abandoned features. Initially as IMVU started to grow after hitting a product market fit, IMVU had allocated 7 different teams that took care of various infrastructure and product related maintenance work. However, these teams had competing metrics, which while good for individual teams, were suboptimal for the whole organization. Also since each of these teams were small (1-3 engineers) it meant that usually only the things that were immediate, got done and each team did not have enough bandwidth to worry about the piling up of technical debt. Another issue was that since everyone is empowered to push a new feature, there was no accountability or ownership of these features and that was why the product became a mass of unfinished features i.e. "a bucket of bolts". [12]

To combat this situation, IMVU put in place a more robust product management team, which constituted several product owners and a VP of Product. Product owners were the owners of the product areas and made decisions on what went in those areas. Overall alignment of the product areas with IMVU's goals was ensured by the VP of Product. IMVU also made a switch to using scrum approaches with 2-3 weeks' sprints, to plan incremental development and delivery. This mitigated problem of the pileup of unfinished features. [12]

In addition, IMVU also learned and developed a lot of new processes that help to scale up and keep technical debt down. For example, technical debt was not put under the rug, and soon as a potential technical debt emerged in the technical review meetings, it was put in the next sprint to be resolved. This was called technical debt feedback loop. Also, while product team determined what percentage of resources will be allocated to a specific team, it was the engineering team that determined who were the actual people that were going to do the work. [12] This ensured that the right people are allocated to the right technical problems and further helped prevent accumulation of technical debt.

Finally, IMVU kept a dedicated project team which worked on remediating technical debt and also instituted a concept of technical debt "tax". This tax was applied on features that needed to change something in a piece of code that was particularly bad, but usually remained untouched, and hence was not prioritized to be refactored. Overall, IMVU has regularly spent 20% of engineering resourced on tests and processes, including a dedicated person per feature team to ensure that code is cleaned up, thereby preventing buildup of technical debt. [12]

5.2.3 Continuous Deployment

Per Eric Ries [13] at IMVU, the code is integrated and deployed frequently. Engineers are encouraged to commit early and frequently. Per Eric Reis, the final objective that continuous deployment tries to achieve is to help organizations minimize wastefulness in their processes, reduce the batch size of pushes and facilitate an increased tempo in their work. While there is a perceived dilemma between risking releasing imperfect code or waiting longer to double check everything, IMVU believes that this is a false dilemma. So, while IMVU accepts that they occasionally do negatively impact customers due to the speed of delivery of new features, and sometimes inadvertently a bug passes into the production system, or introduces some regression error, IMVU accepts these risks because there are many benefits such as learning from early customer responses thereby enabling IMVU to make adjustments quickly before the problem/issue becomes big and potentially costly.

Also, in order to keep a check on the overzealousness of frequent releases at the cost of quality, and at the same time keep a check on the tendency to spend enormous amount of time perfecting the quality of code and thus releasing slowly, at IMVU the development teams are divided into Cowboy and Quality factions, [13] Cowboys favor speed, while the Quality faction wants to take time to focus on quality. While this looks like a partisan situation, this actually works effectively at IMVU as it keeps both sides of pro-quick-release cowboys and pro-double-check-everything quality factions in check and generates the right questions. Also, continuous deployment in this scenario encourages learning on both sides of the divide. This enables a healthy debate and each individual gets an opportunity to learn.

Also, at IMVU there are tools and systems in place to keep a check on the shipping-too-soon tendency. Systems such as Cluster Immune system, continuous integration system, and '5 Whys' are there to keep a check on Cowboys and provide feedback. Because of the quick feedback, Cowboys learn the right level of testing, which enables them to release fast without being too fast and sacrificing quality, and avoiding this sacrifice in quality avoids creating technical debt that could eventually slow the speed.

On the other side of divide, the obvious disadvantages of waiting too long to release, acts as a check on the Quality factions. Because the longer the engineer waits to release, the larger the conflicts become and harder it gets to reconcile these conflicts. Not only that, waiting too long increases the risk of delivering something that may no longer be what customers really want because of elapsed time and changing customer needs over time. Also, because it is harder to do quick tests by putting changes in the front of customer quickly and getting immediate feedback, this makes deployed features vulnerable to failure in meeting the customer needs. [13]

Finally, continuous deployment at IMVU has a strong positive effect on employee morale by reducing the overhead effort of doing a release. Each engineer gets the freedom to work as per their own release schedule. Engineers can check-in code and deploy as soon as it is ready to deploy. There is a high level of individual autonomy, and little need for coordination, approvals etc. It is pretty satisfying to the engineers to be free from these tasks and be able to focus on creating features. This boosts morale, which has a positive effect on work quality. [13]

To support the continuous deployment approach, IMVU has a set of tools and practices that help it to release frequently. Per Eric Ries [14] one of the main tool is BuildBot. BuildBot is a continuous Integration server that acts as a central place to perform testing from unit tests to integrations tests before each time a push is committed. IMVU also has source control systems such as CVS, Subversion and Perforce, that are used to check and enforce coding standards on the code commits. If a piece of code is found unsuitable to deploy, it is rejected and a message is sent to the engineer about the error. IMVU monitors even the code that had been deployed using alert system such as Nagios. Nagios alerts engineers in case an anomaly appears in the system, after it was deployed. Nagios can even stop the production pipeline and halt code commits until the analysis, such as '5 Whys' has been done and the issue has been fixed. [14]

5.2.4 Testing- Test Driven Development and Automation

According to James Birchler from IMVU [15], IMVU uses a combination of Test Driven development strategy with comprehensive automated testing along a small team of human QA engineers who test the edge cases not caught by the automated tests. IMVU also relies on its ability to do A/B tests quickly and effectively to improve the quality of the platform. This, when conducted in conjunction with root cause analysis practice such as '5 Whys', not only catches existing bugs, but also prevents future issues by resolving underlying causes.

Per Birchler [15] and Fitz [16], at IMVU, engineers test everything they build. In fact, before a feature is coded, the engineer writes a specific test for that feature and they are required to unit test the feature before the automated tests are run. IMVU maintains a comprehensive set of automated tests that are used to test code reedy to be deployed, in agreement with the continuous deployment paradigm. IMVU has a strong culture of automated testing. IMVU simulates automated user interaction with the platform using automation tools such as Selenium, that go through the platform and runs various possible use cases and underlying expected behaviors of system. IMVU has built a supporting infrastructure with tools such as BuildBot, to this end. This along with the cluster immune system ensures that every piece of code that would be deployed to the production goes through the extensive set of automated tests and also continuously monitors in-case any indication of anomaly in the system, raises an alert to the team about the bug in the system. This set of tests are continuously curated to ensure that the set remains effective and covers as much ground as possible.

Also, since automated testing, however extensive, has its limitations and cannot catch all the issues. This is why IMVU also has a small team of experienced QA engineers who manually test the system to identify any issue that slipped past the automated tests. These QA engineers are experienced in IMVU platform and use their knowledge of system to use the system in the way a user would and try to find out any potential issue. These QA engineers work closely with the engineers when the feature is being developed and this helps to prevent bugs even before the code is ready for deployment. Also, IMVU encourages engineers to perform demonstrations to another human, and this prevent bugs that may creep in due to oversight. Also, every time a QA engineer finds a test case that could be automated and was not in the automated test suite, that test is added to the test suite. IMVU makes sure that it happens by introducing a formal step in the SCRUM process to perform this curation of automated test suite. [15]

Finally, IMVU uses A/B tests to get user feedback and continuously fix any issue that users encounter of use this feedback to steer product in a way that would stick the best with the users. Since IMVU uses the continuous deployment paradigm of releasing small and often, doing A/B testing become easy and very effective, as pulling a small piece of code out that was deployed very recently, is far easier than pulling out a big chunk of code that took months to develop. [15]

5.2.5 Continuously Learning Culture

One if the aspects of IMVU's culture is learning, and then sharing that learning. One of the methods of learning at IMVU is conducting deep root cause analyses to issues and converting the analysis results into knowledge and sharing it to the whole organization. Per Eric Ries [17] whenever something breaks at IMVU, to get to the root of the problem, IMVU follows a technique called '5 Whys' from Taiichi Ohne, co-inventor of the Toyota Production System. In this technique, the idea behind this technique is to ask a series of question to delve deeper into the multiple layers of the problem. This approach uncovers much deeper issues, than the ones discovered by a shallower root cause analysis. '5-Whys' approach also allows to remediate issue and discover causes at different levels. As a result, the problem is not only solved at the surface level but also some deep rooted issues that could have caused other issues down the line or down the system emerge and get proactively addressed.

Below is one example given by Eric Reis in case of the root cause analysis of site outage: [17]

"Let's say you notice that your website is down. Obviously, your first priority is to get it back up. But as soon as the crisis is past, you have the discipline to have a post-mortem in which you start asking why:

1. *why was the website down? The CPU utilization on all our front-end servers went to 100%*
2. *why did the CPU usage spike? A new bit of code contained an infinite loop!*
3. *why did that code get written? So-and-so made a mistake*

4. *why did his mistake get checked in? He didn't write a unit test for the feature*
5. *why didn't he write a unit test? He's a new employee, and he was not properly trained in TDD"*

The '5 Whys' analysis has discovered a deeper problem at 5 different levels. Now the issue can be fixed at 5 different levels. The deeper the level the more far reaching the remediation can affect.

Remediation for the '5 Whys' analysis example as per Eric Reis: [17]

1. *"bring the site back up*
2. *remove the bad code*
3. *help so-and-so understand why his code doesn't work as written*
4. *train so-and-so in the principles of TDD*
5. *change the new engineer orientation to include TDD"*

In order to conduct an effective '5 Whys' analysis, IMVU has one dedicated person called '5 Whys Master', [17] who is in charge of leading this analysis. As soon as a new problem is discovered, the people associated with the problem are gathered in a room and '5 Whys' master leads the discussion to analyze the issue via '5 Whys' techniques. Once the analysis is complete, this analysis is announced to the whole organizations to ensure that the whole company gets educated about the problem and what this analysis discovered.

According to Eric Reis, [17] it was found that a small set of the problems that keep recurring, are usually responsible for most of the work done on remediation efforts across organization. Using '5 Whys' technique is a very cost effective way to prevent these small set, but potentially very costly, problems that suck up most resources.

Another important aspect of IMVU learning culture is learning by doing [18]. Learning by doing is very effective in breaking past the resistance to change and ingrained habits learned from past experiences. One example of this approach is how IMVU teaches the value of continuous deployment to new engineers who may not be fully convinced about the effectiveness of continuous deployment and remain resistant to change. If a new engineer's favors working on a separate branch of code, and insists on only integrating at the end and not frequently deploying despite the convincing efforts regarding the virtues of continuous delivery from IMVU, Eric Reis mentions that IMVU lets them try their way. He calls it "code bouncing". [18] As the new engineer completes his work on his separate branch of code and then tries to check in this large batch of changed code into the main branch, he/she faces a set of integration conflicts, which require collaborating with other engineers to resolve them. In the process of resolving these conflicts, time passes and new changes by other developers across organization get checked into the main branch of code in which the engineer is trying to merge. So new conflicts appear because for these new changes in the main branch. While resolving these new conflicts, more changes happen in the main branch. This process repeatedly continues until the engineer is somehow finally able to resolve all the conflicts after a lot of effort or gives up and asks for a code freeze of the main branch so that no new code is checked in while the engineer merges his code. This harrowing experience impresses the value of small and quick deployments to the engineer in a far more effective manner than a lecture in the virtues of continuous deployment.

5.2.6 Summary

In spite of being the original Lean Startup company and having the founding champion of the Lean Startup movement as one of its founders, IMVU faced big challenges in maintaining speed as it scaled, and the pursuit of speed had left IMVU with a system that resembled a "bucket of bolts" i.e. a mass of unfinished and un-aligned features, with a discouraged workforce. However, IMVU has come a long way from the "bucket of bolts" system that could not scale as IMVU grew, to a streamlined system that scales at speed. Over the years IMVU has adopted practices that have enabled IMVU to circumvent the choice between speed and scale. The following are the main reasons why IMVU is able to scale while maintaining speed.

- Practice of lean startup/feedback driven process
- Continuous deployment
- Deployment and Testing Process Automation
- Test driven development
- A culture of learning-asking '5 Whys'
- Proactive mitigation of technical debt

These practices do not stand alone, for example automation of deployment and testing process is vital for the implementation continuous delivery pipeline. The practice of lean startup and a culture of introspecting and continuously learning go hand in hand. Overall, these practices, create a synergy at IMVU that enables it to scale at speed.

5.3 Case Study 3- Netflix

5.3.1 About Netflix

Netflix is a popular movie streaming site that accounts for about one third of peak Internet [19] traffic in North America. Netflix was founded by Reed Hastings and Marc Randolph on August 29, 1997 as a DVD by mail business. As of Oct 2016, Netflix is a multinational entertainment company available in over 190 countries with 86 million paid subscribers worldwide, including more than 48 million of these subscribers in the United States. [20]

Netflix has experienced the challenge of growing extremely fast in the short time of 18 years, of which only 9 years have been with streaming technology. Netflix had started with a monolithic code base and has transformed itself into a system that is comprised of many micro services which has facilitated flexibility, scalability and quick releases. This has enabled Netflix to grow very fast in a few years. Below chart shows the growth of Netflix subscribers base during 2010 to early 2016. [21]

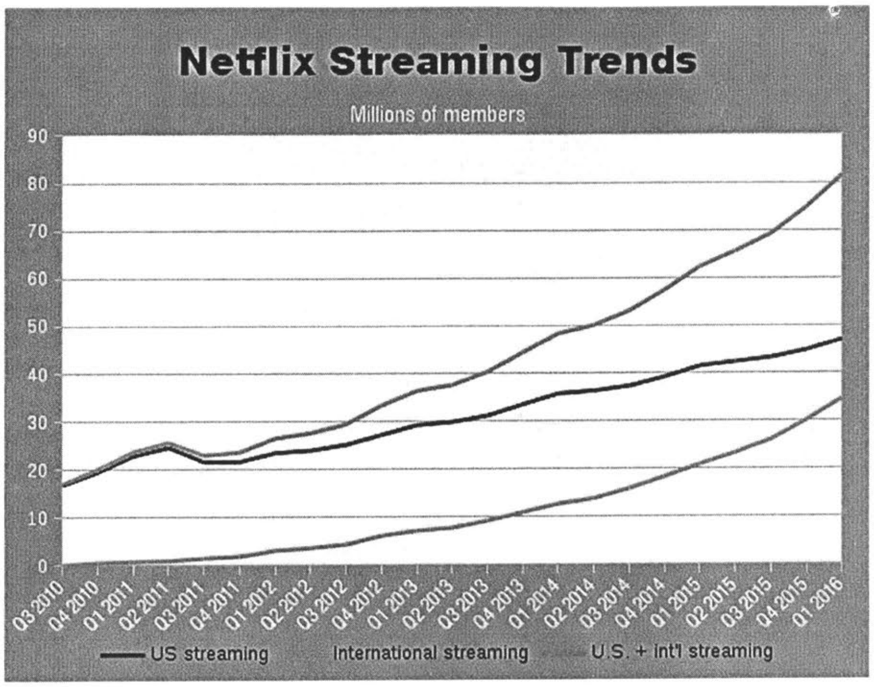


Figure 7 Netflix Subscriber Growth from 2010-2016 [21]

5.3.2 Scalable Architecture

Starting in 2009 Netflix started to move to a modular microservices architecture. The modular microservices architecture is one of the reasons why Netflix has been able to scale very fast without creating enormous technical debt in their systems. As of mid-2015 Netflix used about 500+ micro services and handles about two billion API requests per day. [22] Each of these microservices are individual and independent product features, and they can be modified, updated or debugged independent of the rest of the system. The release schedule of each of these microservices is also independent of the release schedule of other microservices across Netflix. Each of these microservices are managed by a separate team that works independently and makes their own decisions on the release schedule and frequency of releases. [23] Also, since 2009, Netflix has been following a continuous delivery model and this model works very well with microservices architecture. The use of continuous delivery with microservices in Docker containers has enabled Netflix to deploy an update in production in seconds. [23]

Prior to the use of microservices architecture Netflix had a monolithic architecture. This monolithic architecture was prone to failures and was very hard to maintain. In this monolithic architecture at Netflix, the code base was becoming a spaghetti of different subsystems and features and a single error or bug in on small part of the system could bring down the whole of the website. Every time something went wrong with the system all the engineers at Netflix had to be on alert as it was not readily obvious which system originated the error. This monolithic architecture was not only costly and wasteful, but also prone to failures as engineers sometimes inadvertently wrote a piece of code that broke something in the other parts of the system. [22]

The maintainability of the monolithic system is one of the reasons why Netflix moved away from the monolithic architecture to a loosely coupled, independently deployed and maintained microservices architecture. Netflix started with moving non-customer facing applications, such as movie encoding to AWS cloud based microservice and then started to move customer facing pieces such as device configurations, account signups, programming selections etc. Towards the end of 2010, the move of the entire customer facing pieces to AWS was complete, and by the end of 2011 Netflix competed the final transition to microservice architecture and the move of its entire system to the cloud. [22]

Also, the nature of Netflix's business needed a system that would be up and running 24x7x365 and also a system that could scale at speed. Over the past decade and especially since 2009 Netflix has grown now only in terms of user base but also in terms of the platforms it supported. Netflix is supported not only on major operating systems and browsers, but also on gaming consoles such as Wii and Xbox. In addition, this challenge becomes even more pronounced as these devices change rapidly and new devices and form factors such as tablets and smartphones enter the market very fast. For this reason, Netflix needed its system to scale fast, be flexible and reliable. In order to be able to flexibly scale at speed, Netflix chose microservices architecture on cloud. On cloud, Netflix can have flexible capacity and increase/decrease its capacity in a matter of minutes' vs changing the number of servers in a datacenter. Its microservice architecture with continuous deployment has enabled Netflix to very rapidly deploy its features and platforms. This is possible because each microservice is deployed independently by an independent team on an independent schedule. This avoids lot of wasted time in syncing and coordinating changes and schedules, minimizes conflicts and mistakes and keeps up the team morale. All these together became a partial enabler of scale with speed at Netflix. [22]

5.3.3 Continuous deployment/delivery

At Netflix the culture is to move fast and if it fails, then fail small. Since Netflix is in an inventive market space, Netflix believes that fixing some errors as a cost of doing things in a better way, is better than too much focus on preventing errors at the cost of slowing things down. This is why Netflix in 2009 adopted a continuous delivery model to rapidly deliver small size pushes. Netflix moves code through to production smoothly with minimal manual work but also provides insight into the process so that problems can be identified earlier. [24] [25] Netflix has been able to accomplish this without an extensive quality assurance department and without special release engineers. Netflix is able to achieve this because of an automated pipeline on top of Amazon's AWS, that enables individual teams to deploy their piece to the infrastructure as and when they are ready. This allows not only savings in time and effort by engineers in coordinating the deployment efforts, but it also allows for quick feedback from users in almost real time instead of having the implemented features sit on the shelf. Another benefit is that by delivering small increments at a faster rate, it enables engineers to quickly fix any problems as they arise, since the changes are small, and so easy to roll back, and also since the changes are still fresh in the minds of the person who made them. [24]

5.3.4 Philosophy behind CI/CD at Netflix

Per Ben Schmaus, in order to maintain a balance between speed and stability at scale, Netflix has two fundamental principles behind its continuous delivery design: [24]

1. Automation: Netflix has automated most of the pieces of the continuous delivery pipeline which could be effectively done by an automated system. This automation has the advantage of freeing humans to do other more value added tasks such as develop new features. Automation also helps drive down error that can be caused by human oversight or fatigue and provides repeatability and consistency.
2. Insight: Insight means the ability to know the state and health of the system at various stages. From tracing the code in the deployment pipeline, to the running health of various application on infrastructure. This continuous feedback enables Netflix to identify, isolate and take action to fix a problem even before it becomes an issue.

The following diagram illustrates the flow of a feature form the inception to global deployment at Netflix: [23]

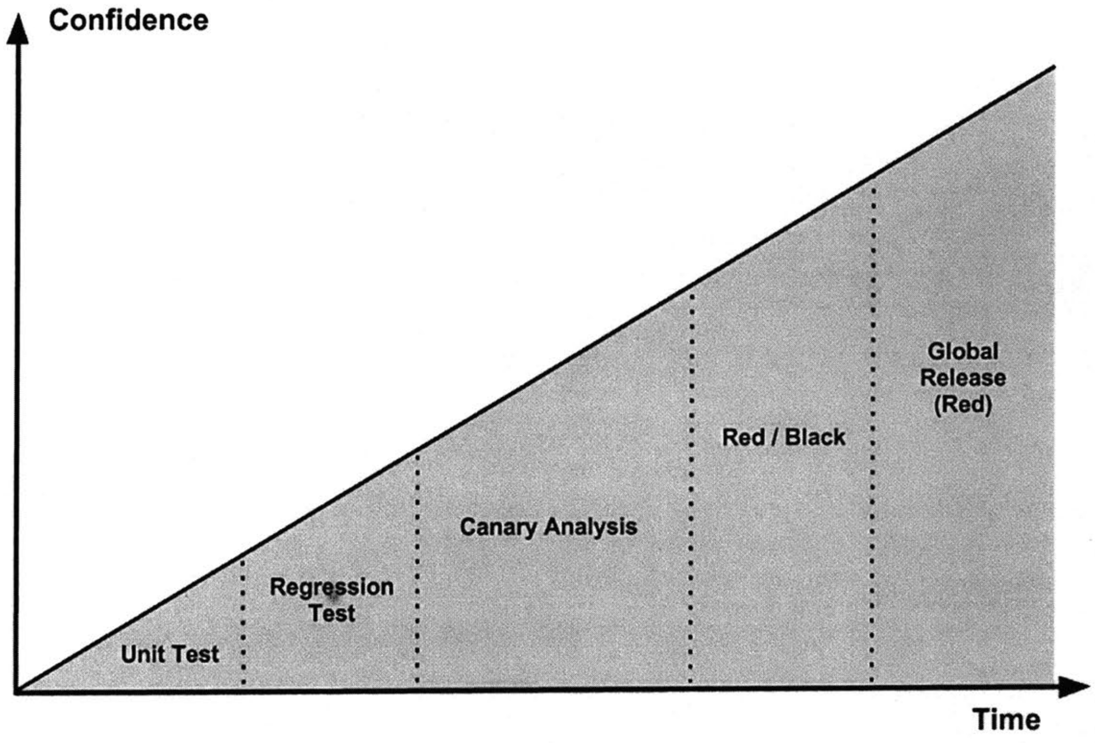


Figure 8 Feature from form Inception to Global deployment [23]

5.3.5 Code Base/Branches at Netflix

According to Ben Schmaus, Netflix maintains three branches with automated pipeline for different environments that serve different purposes. The branch to which the code should be deployed is chosen based on the type of change and its production readiness. [24]

Test Branch: The code is deployed in this branch if it is a kind of change that required several iterations of development/deployment/testing cycles and/or requires coordination from different teams to get it right. Once the code is ready it is merged manually into the release branch by the developer.

Release Branch: The code deployed to the Release Branch is weekly deployment candidates. The code deployed to the release branch is generally deployable, but this branch may become unstable when an integration test is run. Once fully ready for deployment, an engineer manually kicks off the deployment and the deployment is fully automated from that point onwards. Any commits to the release branch is backward put into the test branch as well.

Prod Branch: This branch acts as base for daily push. Changes from the release branch deployments are marched into the prod branch. A "ready for production deployment" change is also put directly into the prod branch and these deployments to the prod branch are auto-deployed to a canary cluster for a limited release to a small user section. If everything is found to be OK, then the code deployed on global infrastructure. The changes to prod branch are also added automatically to release and test branch as well.

5.3.6 Automation and Tooling of the Pipeline and Testing

Netflix has an automated deployment pipeline. According to Zef Hemel the automated pipeline works as follows: [23]

At Netflix, engineers push the code that is ready to be deployed into a code repository. From this code repository, a tool names Jenkins picks it up and builds to create an application package. It also creates a virtual machine image, AMI, including other components that Netflix servers need to run. Once the machine Image is produced it is registered and the deployed on the Netflix infrastructure. On its infrastructure, Netflix has multiple clusters of servers. Each of these clusters contains least 3 EC2 instances. These multiple instances and clusters are maintained in order to create availability spread and speed across multiple zones and also build redundancy to protect against outages. In order to deploy on this infrastructure of multiple clusters, Netflix has created a tool called Asgard. With the help of this tool virtual machine image can be created on new clusters. Also, another reason why Netflix moved away from manual deployment to the use of Asgard, is because Netflix needs to support multiple markets regions and so needs to deploy at many AWS regions. [24]

While new images are being loaded by Asgard, the old clusters are also maintained in parallel to ensure a backup in case something goes wrong. To automate this Netflix has created a service registry called Eureka. Eureka is the service that starts to direct all traffic to the new clusters. The old cluster is maintained overnight running in parallel to the new cluster and if no problems are observed in the new cluster the old cluster is removed and the new cluster stays. Otherwise the traffic moves back to the old cluster and new cluster is rolled back [23]

Finally, the insight and communication of the state of the deployment pipeline is important at Netflix so that the engineers are well aware of what's running in production and plan their availability and work accordingly. However, it is not a great idea to keep the whole team distracted with messages if they are not relevant for the current deployment. This is why Netflix uses an automated bot that sends messages on the team chat boards about the state of the deployment. When someone needs information about any of the pushes current of prior they just need to access this bot that maintains all the history of recent deployment pushes. This saves engineers time and avoids unnecessary distractions for the engineers who are not related to that particular push. [24]

The following diagram illustrates the deployment pipeline at Netflix: [24]

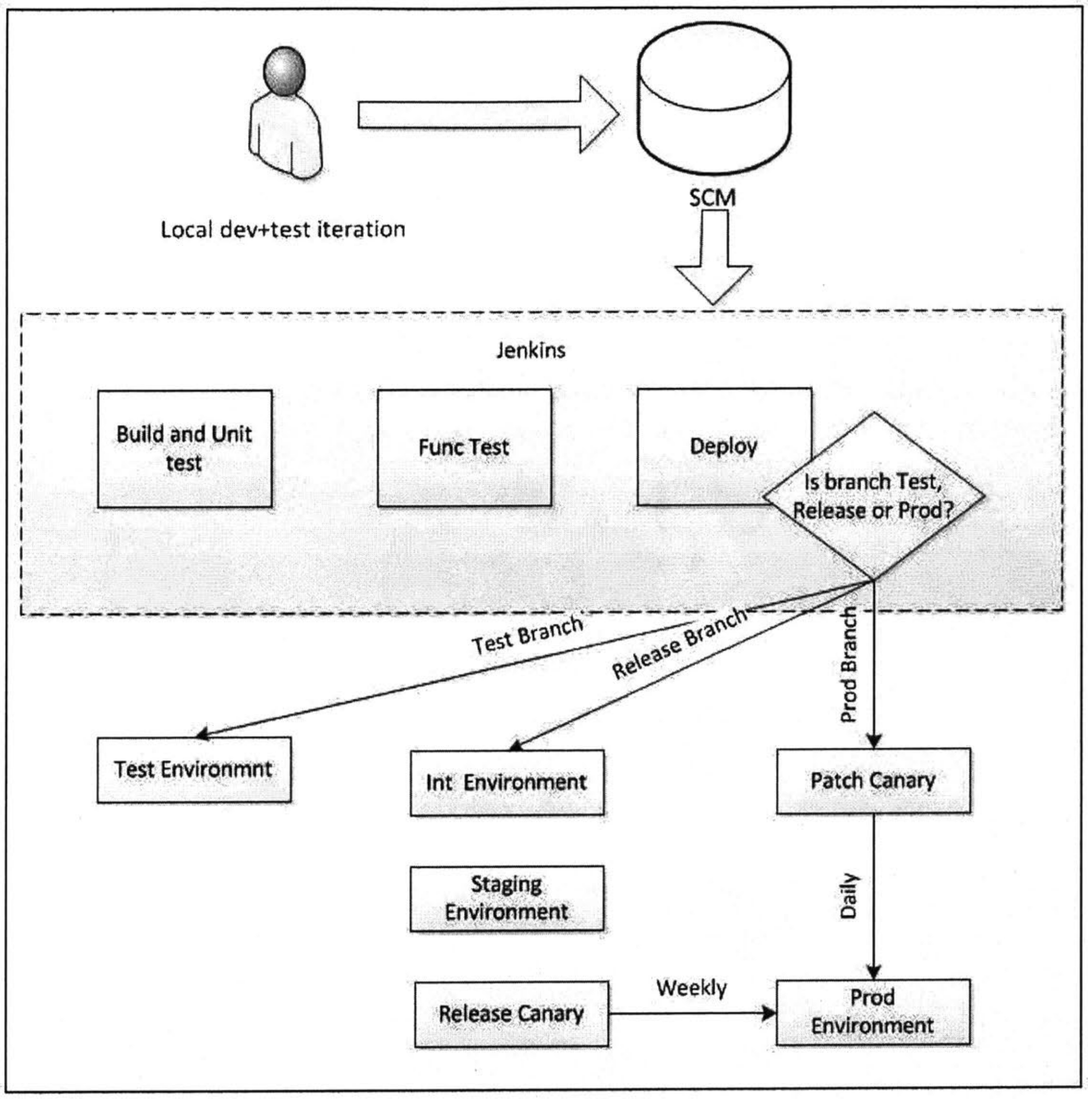


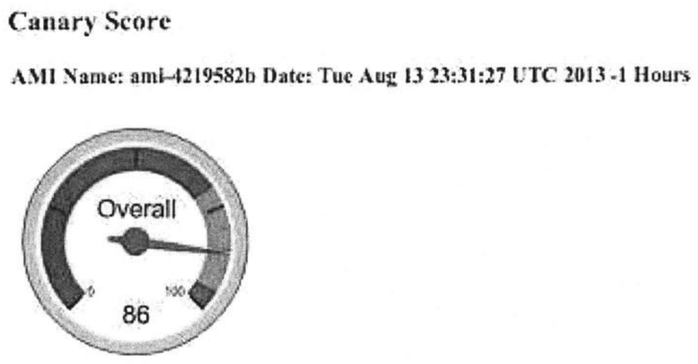
Figure 9 Illustration of the deployment pipeline at Netflix [24]

Additionally, according to Zef Hemel [23], in order to create and encourage failure resiliency in the systems Netflix regularly uses forced failures in the system. These software agents that introduce these failures are called "The Simian Army". These software "monkeys" of simian army introduce random failures in the system, like "Latency Monkeys" introduce latency in systems or

"Chaos Monkeys" introduce server failures etc. These failures keep the team on their toes and encourage a culture of resilience to failure in the organization, which is reflected in the failure resistant systems.

Besides the use of automated tests and A/B testing Netflix also uses canary analysis [24] to test its production code. The idea of canary analysis is that the code is deploys to a small subsection of the production infrastructure and testes there. If no problem arises then the code is gradually moves to the larger section of infrastructure and eventually deployed fully. The basic idea is to fail on a small section instead of failing on entire fleet of servers in the infrastructure.

Also Netflix has moved away from manually analyzing results from the canary deployment to a set of automatically generated metrics that compares the baseline metric with the canary metric on thousands of parameters and that alert the humans if something stands out and found that code is not ready for larger production deployment. This, although not a fool-proof system, does give Netflix a sense of the production readiness of the code. The Canary Analyzer analyzes each canary and creates a report. The Canary Analyzer has a confidence score and if the confidence score is high enough for a canary, that canary is automatically deployed across all regions. Below is an example of a canary report. This canary report indicates a not-high-enough score of less than 95 and hence this canary is not suitable for deployment across all AWS regions. This canary is pulled out and analyzed to discover the issues that may cause problem. [24]


Canary Score

AMI Name: ami-4219582b Date: Tue Aug 13 23:31:27 UTC 2013 -1 Hours

Figure 10 Sample Canary Analyzer Report at Netflix [24]

5.3.7 Culture/innovative mgmt. practices/processes and DevOps

Reed Hastings does not endorse the idea that processes by themselves will be a panacea for create great organizations and great products. According to Reed Hastings, [25] there are good processes and there are bad processes. Good processes help the team become faster and more efficient. One example of good process, according to Reed Hastings, is effectively communicating with the team about the code updates. while bad process creates barriers to speed and efficiency by being too bureaucratic of cautious. One example of bad process is multi-level approval process for projects, or for simple things such as a banner ad. Also, according to Reed Hastings, Netflix keeps curating its processes and keeps getting rid of bad processes as they creep in.

5.3.8 Culture of Freedom and Responsibility

Netflix has maintained a culture of "Freedom and Responsibility" [25] Employees at Netflix are empowered to make decisions and act in the best interest of the organizations as long as they follow broad level of guiding principles. One example of Netflix's "Freedom and Responsibility" culture is that at Netflix nobody tracks vacations and employees can take vacations as needed, as long as they are producing great work.

This paradigm of "Freedom and Responsibility" [25] is aligned with Netflix's effective use of DevOps [23]. At Netflix engineers support what they code and if something goes wrong while the engineer is not only responsible to correct it, but also to investigate what went wrong so that that failure is not repeated again in future. To achieve this, Netflix uses '5 Whys' method and this method not only uncovers the first level reason of the current issue, but also uncovers the deeper issues that may be the real cause of the issue. This not only prevent the obvious issue but also fixes the underlying cause that may have given rise to different issues later.

Another important practice at Netflix is the paradigm of "Context, Not Control". Netflix encourage its managers to provide adequate context and expectations and let the employees take control of how the job will be done. The control fosters a sense of accountability in the employee and this also translates into better quality of work.

Also, Netflix follows a model of "Highly Aligned, Loosely Coupled" [25] model of corporate team work. What it means is that while there is a high level of alignment in strategy, goals and values, however, there minimal effort is expended in cross-functional meetings unless these are to align on strategy and goals. Also 'loosely coupled' means that there is existing trust between teams about tactics that allows them to move fast

The above model is supported by Netflix's focus on hiring and retaining highest performers, because, according to Reed Hastings, [25] they make fewer errors and are 10x more effective than average performer. This is why at Netflix employees producing A level work with B level effort are rewarded and promoted, while employees producing B level work with A level effort are given a generous severance package. One thing to note is that while Netflix does reward performance generously and sub-par performers are let go, Netflix does not rank its employees in performance tiers to avoid unnecessary competition and promote cooperation. To attract and retain great employees, Netflix pays the top of the market salaries to its employees. Netflix tries to see what the employee can make elsewhere, and what would they pay to a replacement if the employee leaves, and what would they pay the person to retain him/her. Netflix makes sure that it pays higher than what the employee could get anywhere else. Also Netflix does not have any fixed budgets for compensation and increases compensation as the employee's market value increases to remain the top of the market payer. This practice has helped attract exceptional talent and maintain talent density as Netflix has grown. [25]

5.3.9 Summary

Netflix had originally started with a monolithic architecture which was very hard to scale. It was a monolithic spaghetti of code for different features and one break in part could have ripple effect in other parts. Since 2009 Netflix started a transformation that made it a highly scalable system while being very agile. Netflix has moved to a highly modular architecture using independent and decoupled microservices. It was also the same time when Netflix moved to the paradigm if continuous delivery. Netflix uses and delivery pipeline to automate various facets of deployment and testing. This automaton is also used to continuously gather and analyze data related to the health of system and take proactive action to prevent issues before they manifest. Netflix also uses canary technique to gradually test and deploy in stages.

Another major aspect of Netflix is its focus on high caliber of its employees. Netflix makes a lot of effort to hire and retain the top talent by paying above market rate and only keeps the best talent. Netflix has a culture of freedom and responsibility, in which the high caliber employees are empowered to make decisions, keeping in mind the broader context and alignment with high level strategy. Engineers at Netflix are fully responsible for what they write, from coding to supporting it in the production. This is similar to the DevOps philosophy. To support this, Netflix needs high caliber employees. This is why Netflix makes a lot of effort to hire and retain the top talent by paying above market rate and only keeps the best talent. In summary the following practices have transformed Netflix in a highly scalable system that is able to maintain massive growth at a high speed.

1. Decoupled and modular architecture-microservices
2. Culture of freedom and Responsibility-DevOps paradigm
3. Focus on hiring and retaining highest quality employees
4. Continuous delivery with Automation of delivery pipeline
5. Monitor and get insights by gathering and analyzing data from its pipeline
6. Gradual and staged deployment-canary analysis

5.4 Case Study 4 - Spotify

5.4.1 About Spotify

Spotify is a music streaming service that provides copyright protected music and other content from various artists and media companies. On Spotify, users can browse or search music by artist, album, genre, playlist, and record label. Spotify was launched in 2008 by the Swedish startup Spotify AB. [26] As of June 2016, Spotify had more than 100 million active users, [27] of which 30 million were paying subscribers, this had increased to 40 million by September 2016. [28] Also as of May 2016 Spotify had a headcount of 1,610 full-time employees. Spotify is growing very fast as it added 300 more staffers to its payroll in 2015-16, compared to the year prior. [29]

5.4.2 Continuous Delivery, tooling and testing at Spotify

Spotify follows a frequent release model via continuous Integration and continuous delivery pipelines. Since releasing big and slow creates multiple conflicts at the time of merge as by the time big release is ready to be merged, the main code branch changes with the other changes to it. As a result, developers need to spend considerable amount of time and effort to resolve those conflicts before releasing. This makes releasing harder and messier and lowers motivation and hence pushes release frequency even lower this creates a vicious cycle of slower and slower releases. On the other hand, releasing often creates less dependencies and less conflicts so it becomes easier to release. This makes releasing easier and hence developers are motivated to release more often and this creates a virtuous cycle of faster releases. The following diagram exemplifies the Spotify's philosophy about frequent releases. [30] [31]

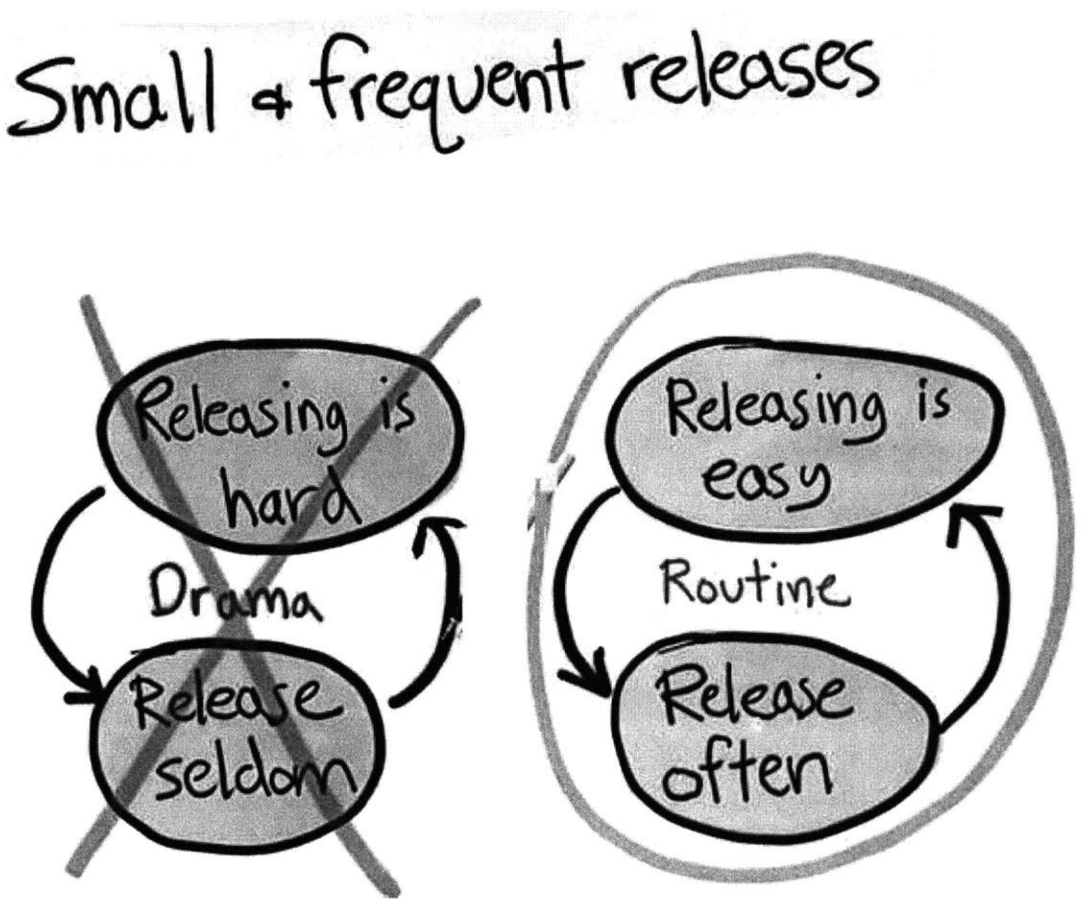


Figure 11 Continuous releases at Spotify [30], [31]

Also, Spotify uses a concept of release trains. Release trains contain a set of features that are released regularly. Since these releases happen frequently a feature that gets developed gets on a release train. If a feature is not fully ready for the release, it still gets on the release train (but remains hidden with feature toggle) so that dependencies or any integration issues are discovered as early as possible. The diagram below describes how release trains work at Spotify. [30] [31]

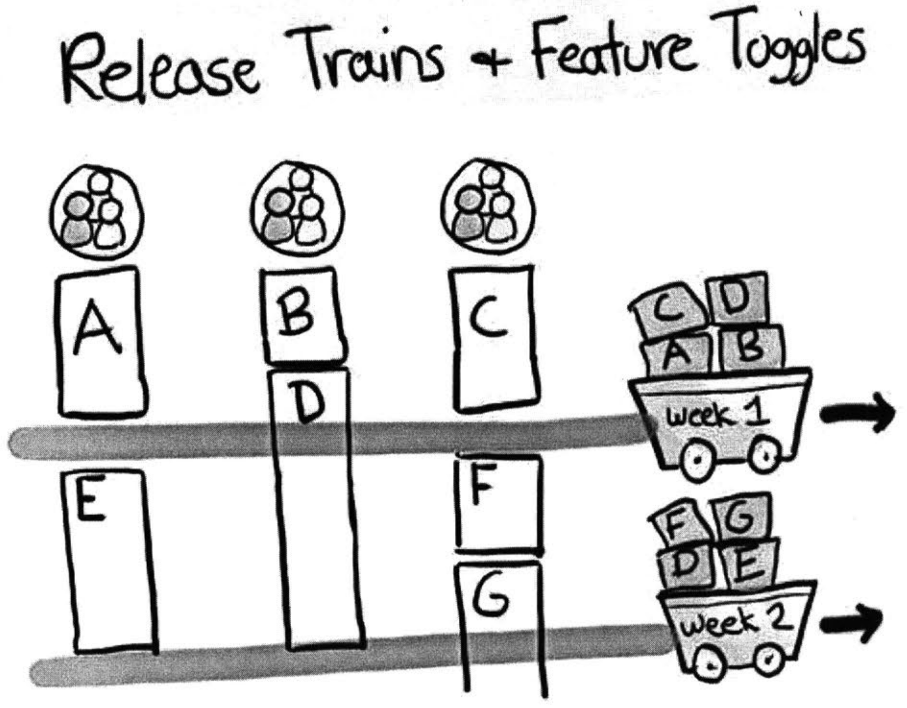


Figure 12 Concept of Release Trains and Feature toggle at Spotify [30], [31]

5.4.3 Tooling the continuous delivery pipeline and automated testing

Per Matt Linander, Technical Product Owner at Spotify, [32], Spotify maintains and uses a set of tools to create the continuous delivery pipeline and to automate testing. To this end, Spotify runs a large number (>100) of backend services using Docker and Helios in production. Most of these services are delivered via Continuous Integration/Continuous Delivery pipelines. A Typical pipeline path at Spotify looks like the following: [32]

As soon as an engineer, when done with the coding, merges the changed code to the master branch, the service starts to get built by a Jenkins instance, launched when it gets notified of the change. A Docker-maven plugin is used to create a Docker image for these services that are written in Java. After this a set of integration tests are run to test the changes. Once the testing is successful the change is moved further in the deployment pipeline. This step is taken over by Helios which deploys machine by machine and finally a check is performed.

This is one of the several ways in which the deployment happened at Spotify. There are variations that depends on each team's preferences.

5.4.4 Service oriented Architecture at Spotify

Spotify follows a micro services architecture paradigm to scale up quickly. Kevin Goldsmith, Vice President of Engineering at Spotify, talked at the GOTO Berlin 2015 conference about how Spotify uses micro services to be innovative and build a scalable system. [33] He argued that micro services are easier to deploy, test and monitor than monolithic systems. Per Kevin, for an enterprise to be able to adopt fast and keep innovating in a highly dynamic and competitive marketspace, it needs an architecture that can scale fast. [34] Also, Spotify has a large platform that supports more than 100 million active users with about 40 million paid users (as of September 2016) and has than 1,600+ employees who maintain these systems that run some very complex business rules. [27] [29] This requires a system that can scale its components independent of each other. To this end, Spotify strives to minimize possible dependencies in its product, and microservices are very effective for creating a system of independently deployed and managed parts. [33] [34] These microservices are unit of functionality and are designed in a way that these units have little dependency on each other. This is why Spotify chose to build on a micro service architecture with autonomous and full-stack teams which are in charge of one or more microservices. This is very effective in avoiding the occurrence of synchronization hell between teams. Also the sheer number of developers pushing changes to the same product makes it very important to minimize these dependencies as much as possible. [35]

Per Kevin Goldsmith, VP of Engineering at Spotify at the GOTO Berlin 2015 conference: [33]

"We've been doing micro services at Spotify for years. We do it on a pretty large scale. We do it with thousands and thousands of running instances. We have been incredibly happy with it because we have scaled stuff up. We can rewrite our services at will - which we do, rather than continue to refactor them or to add more and more technical data over time. We just rewrite them when we get to a scaling inflection point. We do this kind of stuff all the time because it's really easy with this kind of architecture, and its working incredibly well for us. So if you are trying to convince somebody at your company, point to Spotify, point to Netflix, point to other companies and say: This is really working for them, they're super happy with it."

This micro services architecture has allowed for a lot of resilience in the Spotify system. [33] [34] This modularity has enabled Spotify to continue operating seamlessly even when some of its services are down. In addition, at Spotify, there is an inbuilt assumption and hence preparedness about the possibility of failures in the service design itself so even if individual services fail it may not affect user experience very much.

However, for microservices architecture to be effective the documentation and discovery of these microservices needs to be effective. In order to create good documentation and discovery tools for microservices, Spotify has built and uses a system discovery and documentation tool that documents and displays all the microservices that are available. In addition, calling various microservices frequently may cause latency issues, so to combat this latency issue Spotify has built "view aggregation" services that collect data in order to populate client view on the server itself. This strategy view decreases client calls to the server and hence effectively reduces latency. [34]

5.4.5 Processes

Spotify uses autonomous teams (called squads) that have full-stack technical capabilities and also are independent and can act freely. The microservices architecture works very well with this structure since microservices are independent by design and can be worked on or can be modified independently without any major effect on other microservices. At Spotify these microservices are owned by individual full-stack squads (full stack teams). Since each squad is full-stack, it consists of front-end developers, back-end developers, QA engineers, Product Owner and UI expert. These teams are made full stack because their mission is to independently and autonomously manage the microservice and hence they need to able to have all the necessary capabilities to do it independently or in essence act like a mini start-up managing and being responsible for an independent product. [35]

Per Kevin Goldsmith, Spotify's VP of Engineering at Spotify: [33]

"Developers deploy their services themselves and they are responsible for their own operations too. It's great when teams have operational responsibility. If they write crummy code, and they are the ones who have to wake up every night to deal with incidents, the code will be fixed very soon."

So in summary, Squads are responsible for deploying/owing and maintaining the operations of their microservice at Spotify. Spotify has a set of tools and set of practices that help these squads in this endeavor. In fact, the very paradigm of microservices supports this individual team ownership. Whenever possible, Spotify encourages breaking up larger systems into smaller microservices, each doing a specific task. For example, Spotify makes the ownership of microservices unambiguous, so that there is no confusion about who is doing what. This way there is little overlap between areas of responsibilities or missions of different teams. Spotify also has

Guilds that are across squads and these guilds help squad members with cross team learnings thereby bringing consistency to the practices across whole organization.

5.4.6 How Spotify keeps Architectural Integrity

Per Ivarsson et al. [36] at Spotify, Squads are full-stack teams that are independently in charge of a feature. So they may sometimes have to make changes to many different systems to update their feature. Since everyone can make updates to multiple system and no-one is focusing on the architecture of the whole system then the overall architecture can deteriorate. So in order to maintain architectural integrity Spotify creates a "go-to" person or a "System Owner" who is responsible for the architectural integrity of that system. Some systems have more than one system owner and many times if the system is critical from the operations perspective, then there can be two system owners, one being from the operations side and another from the development side. This becomes a Dev-Ops setup. This "System Owner" or "System Owners" are responsible for taking care of the architectural integrity of the system. However, these System Owners are not the typical system architects and architecture is not their primary job. They are a regular squad member or a chapter member or lead and have other regular duties. They do the architecture work for their system from time to time, on an as needed basis. Mostly their job as a System Owner is to keep watch and guide developers when they make modifications to their system to ensure that conflicts or technical debt is not created in their system. Basically these system owners take care of the stability, quality, prevention of technical debt and scalability of the system. In addition to the System Owner role Spotify also has a role of Chief Architect. The job of a Chief Architect is to coordinate among the different system to ensure good architectural practices are followed and are consistently implemented across the systems. Even though the Chief Architects are not enforcers, the individual squads have the final say on the decisions related to design, but the Chief Architect acts as a strong influencer to maintain good architecture across Spotify's systems. [36]

5.4.7 Organizations and Culture

Spotify has a culture of Autonomy with Alignment. In order to create alignment across organization there are rules however, these rules are not strictly enforced, the systems are designed in a way that it becomes very hard to not act in accordance with the rules. These rules are not arcane or arbitrary, but largely the set of best practices that have evolved over a period and are easy to pick up and follow. Teams are free to work their way, however it become much harder if they choose to do in a way that is not in alignment with the Spotify's prescribed way.

In addition, Spotify has a concept of Servant Leadership. At Spotify, Leaders do not tell other people what to do, instead leaders have to articulate what needs to be done and the parameters around the goal. The leader has to trust the team to be able to deliver and achieve the goals within the parameters. If the team does not deliver, usually it is because the goal was not articulated well or the parameters were not set adequately. This makes the leaders think hard to accurately and effectively articulate goals and the parameters. Following is an illustration of the concept of servant leadership at Spotify: [30] [31]

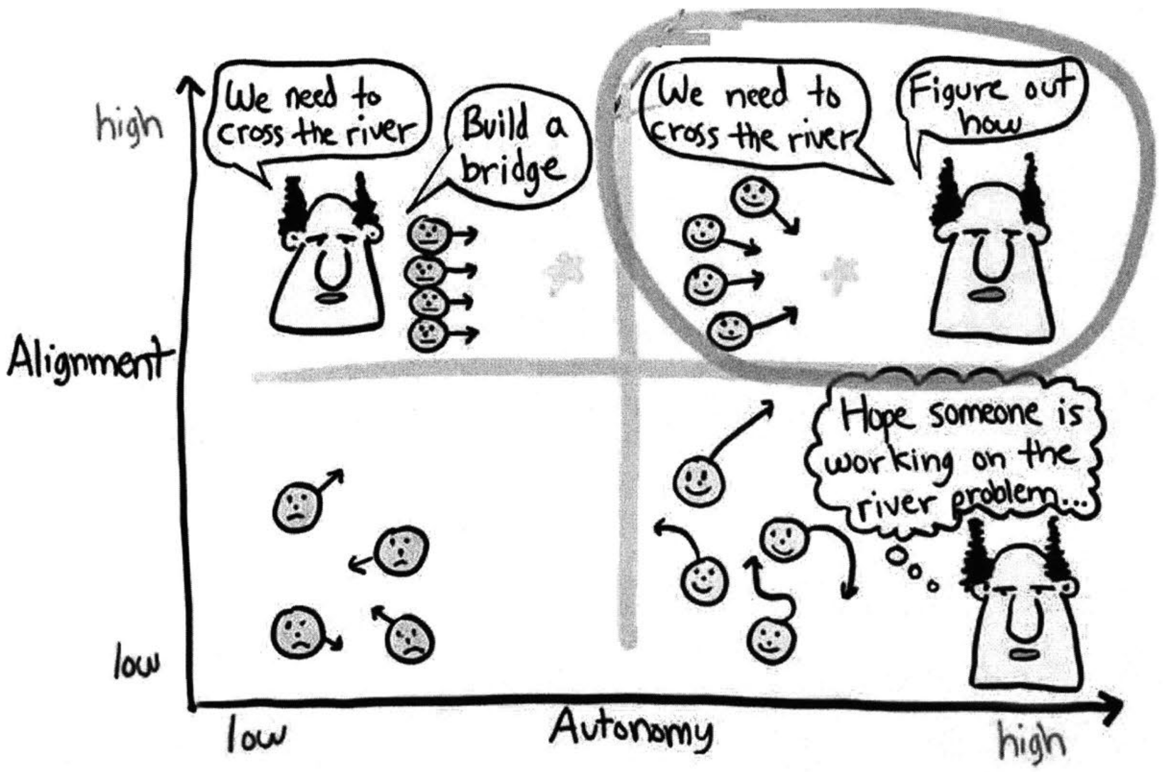

high
We need to cross the river
Build a bridge
We need to cross the river
Figure out how
Alignment
Hope someone is working on the river problem...
low
low
Autonomy
high

Figure 13 Concept of Alignment and Autonomy at Spotify [30], [31]

In order to implement and encourage the culture of autonomy with alignment and servant leadership, Spotify has organized its team structure in an innovative way, in terms of Squads, Chapters and Guilds along with various support roles as System Owners and Chief Architect. Per Ivarsson et al. Organization structure at Spotify can be illustrated as in the following diagram: [36]

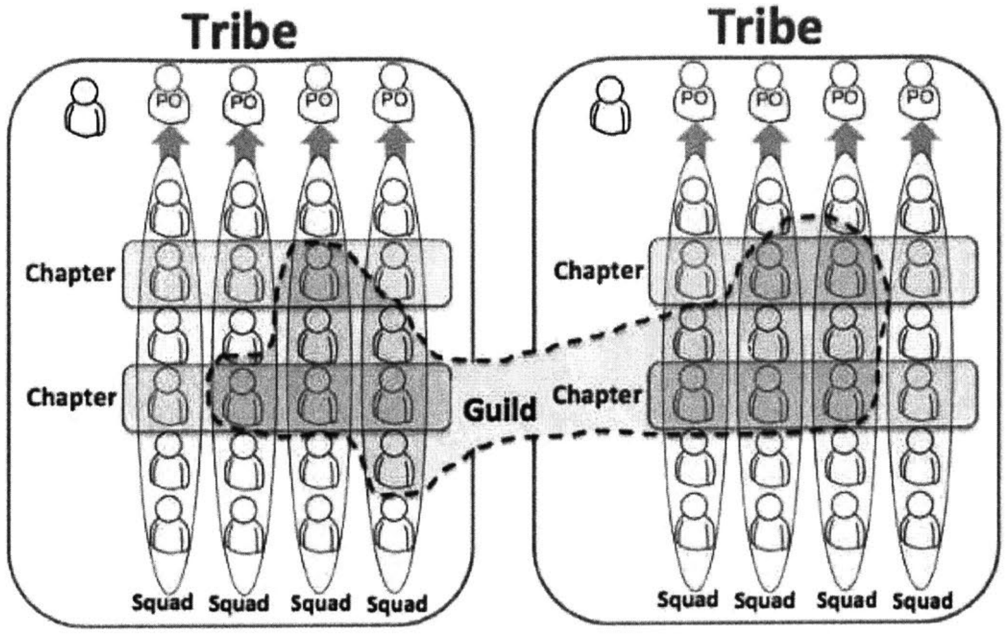


Figure 14 Spotify Organization [36]

5.4.7.1 Squad

A Squad is the smallest unit of the organizations at Spotify. A Squad is a mini-startup team designed to function with little dependence on other teams. A Squad is the owner of its part of the system and this squad decides how to design, develop, test and release for their part. These squads remain with their part of the system long term and hence develop a deep expertise in that area. These squads, although they do not have a formal leader, they do have all the necessary skills to act as a self-sufficient startup team capable of supporting the product independently. A typical squad typically has a Product Owner, an Agile Coach, Developers and Testers to complete the team. [36]

Spotify encourages squads to function according to the "Lean Startup" paradigm where the model is to "Think it, Build it, Ship it and Tweak it". [37] Squads deliver a minimal viable product (MVP) quickly and then get customer feedback and validate their learnings. Based on the validated learning squads tweak the MVP to further build a product that customer will like. This is an iterative process that relies on learnings from releasing early and often to the actual customers.

At Spotify, squads can also use 10% of their time on whatever project they want to. This time is usually called "Hack Day" or "Hack Week" depending on how the squad chooses to take it, and it helps the squad team to continue learning and be at the cutting edge of the technologies, which further improves the quality of their work. [36]

5.4.7.2 Tribes

Spotify groups the squads that are working in a related area together in the larger groups called tribes. These tribes are limited to the 100 people as it is believed that humans find it very hard to maintain social relationship beyond 100 people at any time. A Tribe is usually located in the same building and acts as a platform to facilitate communication between related squad teams, helping to share knowledge, new developments and help alert for and resolve possible conflicts. Also, since tribes consist of squads working in same or related area, and they work in the same physical location it proves to be effective in identifying potential conflicts by formal or informal interaction between members of different squads. [36]

5.4.7.3 Chapters and Guilds

While squads and tribes are focused on the delivery. Spotify also has cross tribe or cross organizations groups of the members of the same competency. For example, a Testers Chapter or a Testers Guild. A Testers Chapter is a group of testers within a tribe and helps testers get better in the testing discipline. In a chapter members share, problems, issues and lessons related to their competency. Guild is more loosely organized and has broader reach beyond a tribe and across organization. These competency groups focus on the competency, more that the delivery focused groups i.e. squad and tribe. This way Spotify maintains the high level of technical competencies of its employees, while still maintaining the focus on the delivery. [36]

5.4.8 Summary

Spotify has a very innovative organization structure, which emphasizes autonomy with alignment. Spotify's maintains speed by creating autonomous, self-contained, "mini startup" teams or squads. At the same time Spotify keeps a check on the alignment of these mini-startups with a supporting roles of product owners, system owners and chief architect. The use of modular architecture in the form of independent microservices works very well with the Spotify's organization structure with each team managing their own microservice independently. This modular architecture with the use of continuous delivery with frequent controlled releases and automated testing and deployment pipelines are also the factors that enables Spotify to scale with maintains quick releases. In summary, the following practices at Spotify help it achieve speed at scale.

- Culture of autonomy with alignment-use of innovative organization structure
- Small and frequent but controlled Releases-Use of continuous delivery with feature toggles
- Automation of release pipeline
- Use of modular architecture with microservices

6. Discussion

This section consists of the discussion of the repeatable best practices of the organizations based on their case studies in the previous section. These best practices are discussed in the context of their usefulness in enabling the organizations in this study to scale up their system and maintain high speed of releases. The following best practices are discussed:

- Best Practice 1: Managing software complexity and technical debt with scalable and decoupled architecture
- Best Practice 2: Test Driven Development
- Best Practice 3: Continuous Integration, Continuous Delivery, Continuous Deployment with Automation and Tooling of deployment pipeline
- Best Practice 4: Alternative QA/Automated Testing/Continuous Testing
- Best Practice 5: Innovative communication/organization structures and management practices

6.1 Best Practice 1: Managing software complexity and technical debt with scalable and decoupled architecture

In the initial stages of a startup product development, speed to market is of paramount importance and architecture takes a back seat. But as a product grows, and more features are added, and if architecture continues to take back seat then along with the rise in complexity, the internal dependencies rise. If it is not taken care of earlier in the life of the system, these dependencies become bottlenecks and the growth eventually slows down because of this accrued technical debt. The successful organizations in this study have mitigated this problem by adopting a modular service oriented architecture earlier in the life of the systems.

The use of modular architecture that is built of different independent parts leads to a very scalable and easy to maintain architecture, since in the service oriented architecture paradigm, different parts of the system are independent of each other and internal changes of one part does not impact the functioning of other part. This is why each part can be updated and maintained independently and on a separate schedule from the other parts of the system.

Jeff Bezos's "Bezos Manifesto" [38] is a prime example of the importance of Service oriented architecture (SOA) to maintain scalability and minimize the technical debt. Below is the text of Jeff Bezos manifesto/mandate to Amazon's employees. [38]

1. "All teams will henceforth expose their data and functionality through service interfaces.

2. Teams must communicate with each other through these interfaces.

3. There will be no other form of interprocess communication allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

4. It doesn't matter what technology they use. HTTP, Corba, Pubsub, custom protocols -- doesn't matter. Bezos doesn't care.

5. All service interfaces, without exception, must be designed from the ground up to be externalizable. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

6. Anyone who doesn't do this will be fired.

7. Thank you; have a nice day!"

Many of these organizations in this study such as Netflix, Amazon and Spotify have taken modular architecture even further by using microservices. A microservices architecture is a special case of the broader service oriented architecture paradigm. The microservices architecture paradigm, like the service oriented architecture paradigm, is used to build flexible and independently deployable software systems. As in the service oriented architecture, services in a microservice architecture are independent processes or functions that communicate with each other over a network to process a request. Also, just like service oriented architecture, the microservice architecture uses protocols that are technology-independent.

In the microservices architecture paradigm, services are usually smaller than the services in service oriented architecture. The microservices architecture can be loosely compared to a Lego model, with each Lego brick as an independent service. Since these microservices are individually and independently deployed and, with little dependencies on other microsystems and designed to perform single service, it becomes easier to service these microservices independently. The benefit of distributing different functionalities of the system into different smaller microservices is that it enhances the cohesion and decreases the coupling between different parts of the system. This makes it easier to change and add functions and qualities to the system at any time. The microservices architecture paradigm also allows for the architecture of an individual service to emerge through continuous refactoring and rewrites, and thereby reduces the need for a big up-front investment in design, and enables releasing software early and continuously.

For example, in 2009 Netflix started to move to a microservice architecture before this paradigm became commonplace. At the time Netflix had a monolith codebase which was a spaghetti of interdependence. One bug in on part of the system could bring down the whole system. This is when Netflix started to use microservices architecture of loosely coupled service blocks. This use of microservices helped Netflix service become much more resilient to failure and maintainable. Also Netflix adopted a model of continuous delivery model, which complements microservices architecture very well. Continuous delivery works well with microservices architecture because each microservice is a usually an individual and independent product feature that can be maintained/updated autonomously and on its own schedule, without impacting other microservices. Issues or release schedule problems with one microservice usually has no detrimental effect on the release schedule of any other microservice. Also with the use of microservices Netflix is able to deploy thousands of simultaneous instances of server in case of increased demand for services. Netflix can adapt and adjust its capacity up or down with AWS cloud very quickly, in minutes. In the past, same effort to increase capacity could have taken hours or days. With microservices individual components can scale at different rates and this allows multiple independent teams working on different pieces to have different release schedules. This increases the agility and effectivity of the development process at Netflix along with scale. [22], [24] Another example of use of decoupled service based architecture is use of microservices at Spotify. Spotify has grown very fast since its launch in September 2008 and as of June 2016, Spotify had more than 100 million active users. To be able to scale at this speed, Spotify needed to build the system that could scale components independently. This system, therefore, could not work with a single monolithic spaghetti code base and needed a way to decouple different components from each other, with minimal interdependencies, so that these components could scale individually. This is why Spotify chose an architecture paradigm based on microservices while putting autonomous full-stack squads (teams) in ownership of individual microservices in order to mitigate synchronization issues across the organization. [35]

Also as Kevin Goldsmith in his GOTO Berlin 2015 conference [33] mentioned that Spotify chose microservices because there were easy to scale. Any problems in a microservices based system can be identified and isolated easily and thereby fixed quickly with lower cost. These microservices are much easier to test because of smaller scope. Finally, microservices are easy to deploy as these can be deployed independently and quickly. The risk of failure is low and ease of rollback in case of failure is much more than a monolithic system. [35]

Spotify's use of microservices also gels well with its organizational structure where individual squads are responsible for individual components or microservices and have little dependence on other squads. To resolve any conflicts and maintain architecture integrity, Spotify has a role of "System Owner". Every system at Spotify has one or more system owner (s). These System owners coordinate and guide people to ensure that they don't create unnecessary conflicts in their code. System owners ensure that documentation is up to date, looks out for any buildup of technical debt, and monitors stability of the system. In addition to the System Owners role, Spotify also has Chief Architect role to ensure consistency in architecture across Spotify. The job of a Chief

Architect is to coordinate among the different system to ensure good architectural practices are followed and are consistently implemented across the systems. Even though the Chief Architects are not enforcers, as the individual squads have the final say on the decisions related to design, the Chief Architect acts as a strong influencer to maintain good architecture across Spotify's systems. [36]

6.2 Best Practice 2: Test Driven Development

Many of the organizations in this study follow parts of a Test Driven Development (TDD) methodology. Test Driven Development is a methodology that requires that every feature that is developed should have a test written for it before the development of the feature begins. This approach is different from the traditional waterfall approach where tests are written after the feature has been coded. Test Driven Development approach is similar to the user story driving approach used in many agile practices such as Scrum, in which acceptance criteria are written and development is based on the acceptance criteria. The main difference between agile and TDD approach is that in the agile model, the acceptance criteria is not a full blown test case, and a test case would subsequently need to be developed based on the acceptance criteria, and development can begin without the actual test case in place. However, in Test Driven Development the actual test is written using the user story or the use case as a base. This test is validated to ensure that it is an effective test case before the coding begins.

Per Agile Alliance*:* [39]

"Test-driven development" is a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules and expected benefits per Agile Alliance:

- *"Write a "single" unit test describing an aspect of the program*
- *Run the test, which should fail because the program lacks that feature*
- *Write "just enough" code, the simplest possible, to make the test pass*
- *"Refactor" the code until it conforms to the simplicity criteria*
- *Repeat, "accumulating" unit tests over time*

Expected Benefits

- *Many teams report significant reductions in defect rates, at the cost of a moderate increase in initial development effort*
- *The same teams tend to report that these overheads are more than offset by a reduction in effort in projects' final phases*
- *Although empirical research has so far failed to confirm this, veteran practitioners report that TDD leads to improved design qualities in the code, and more generally a higher degree of "internal" or technical quality, for instance improving the metrics of cohesion and coupling"*

In summary, the practice of Test Driven Development not only encourages a simpler, more robust design and enables easier debugging but also leads to a more modularized and scalable code. In the test driven development paradigm, developers think, write and test code in small independent units and later sew these pieces together. This practice results into smaller and better focused class designs along with looser coupling between parts of the system and cleaner interfaces between subsystems. This methodology also results into a set of automated test cases that can be run each time a new feature is deployed. This is even more effective since every piece of code is written in a way that makes it sufficient to pass a test case, these automated tests automatically cover every scenario or parts of codebase. This set of tests continues to get more robust as new tests get added to the set because engineers keep writing and adding test each time they develop a new feature.

IMVU uses this practice of test driven development and it has helped IMVU to create a set of thousands of automated and manual (to a lesser degree) test cases that are run each time a developer pushes a new feature in the main line. IMVU has a very small QA department. It's the responsibility of the individual developer to also write the test to verify the code. This builds an extensive suite of automated tests that gets more robust every time a new test is added. This provides extensive test coverage. However, this set of automated tests cannot cover every edge case or functionality. Some cases must be tested manually by intelligent and experienced humans. This is why even though IMVU has an extensive suite of automated tests and relies on automated tests primarily and heavily, IMVU also has a small QA team of smart and experienced engineers who identify these edge cases and test those. These QA engineer work closely with the developers as the feature is being developed. This provides the QA team with much deeper insights into what the feature is and what it is supposed to do. This makes the manual QA team much more effective than simply handing off the developed feature to the QA to test. Many bugs are found at this stage and fixed even before formal testing. Also, these edge cases when discovers are also added to the test suite, if possible. IMVU makes sure this happened by creating a task in the SCRUM for this. Another practice is the use of peer demo, in which the developer has to demonstrate his developed feature to another engineer. This forces manual testing and a second set of eyes even before the automated or manual testing is done.[15]

6.3 Best Practice 3: Continuous Integration, Continuous Delivery, Continuous Deployment with Automation and Tooling of deployment pipeline

Successful organizations such as Facebook, Netflix and IMVU encourage developers to integrate and deploy code quickly, frequently and incrementally. This is opposed to the prevailing paradigm of branching out codebase, coding a set of features and then merging the changes after few months when the whole feature set is done. This approach presents a challenge of large merge conflicts as the main line undergoes substantial changes by the time release branch is ready to be merged. Continuous integration solves this problem by reducing the need for large scale merges. Since the bite-size changes are integrated into the mainline quickly and frequently, the likelihood of integration conflicts and build failures goes down substantially when the developer's small code change or update push is integrated into the main line of code.

The following is a good description of Continuous Integration, Continuous Delivery and Continuous Deployment and their differences as per Michael Chletsos. [40]

"Continuous Integration is the practice of merging development work with a Master/Trunk/Mainline branch constantly so that you can test changes, and test that changes work with other changes. The idea here is to test your code as often as possible to catch issues early. Most of the work is done by automated tests, and this technique requires a unit test framework. Typically, there is a build server performing these tests, so developers can continue working while tests are being performed".

"Continuous Delivery is the continual delivery of code to an environment once the developer feels the code is ready to ship. This could be UAT or Staging or could be Production. But the idea is you are delivering code to a user base, whether it be QA or customers for continual review and inspection. This is similar to Continuous Integration, but it can feed business logic tests. Unit tests cannot catch all business logic, particularly design issues, so this stage or process can be used for

these needs. You may also be delivering code for Code Review. Code may be batched for release or not after the UAT or QA is done. The basis of Continuous Delivery is small batches of work continually fed to the next step will be consumed more easily and find more issues early on. This system is easier for the developer because issues are presented to the developer before the task has left their memory".

"Continuous Deployment is the deployment or release of code to Production as soon as it is ready. There is no large batching in Staging nor long UAT process that is directly before Production. Any testing is done prior to merging to the Mainline branch and is performed on Production-like environments. The Production branch is always stable and ready to be deployed by an automated process. The automated process is key because it should be able to be performed by anyone in a matter of minutes (preferably by the press of a button). After a deploy, logs must be inspected to determine if your key metrics such as revenue, user sign-up, response time or traffic, are affected.

The following diagram is an example of the pipeline of Continuous Deployment per Michael Chletsos. [40]

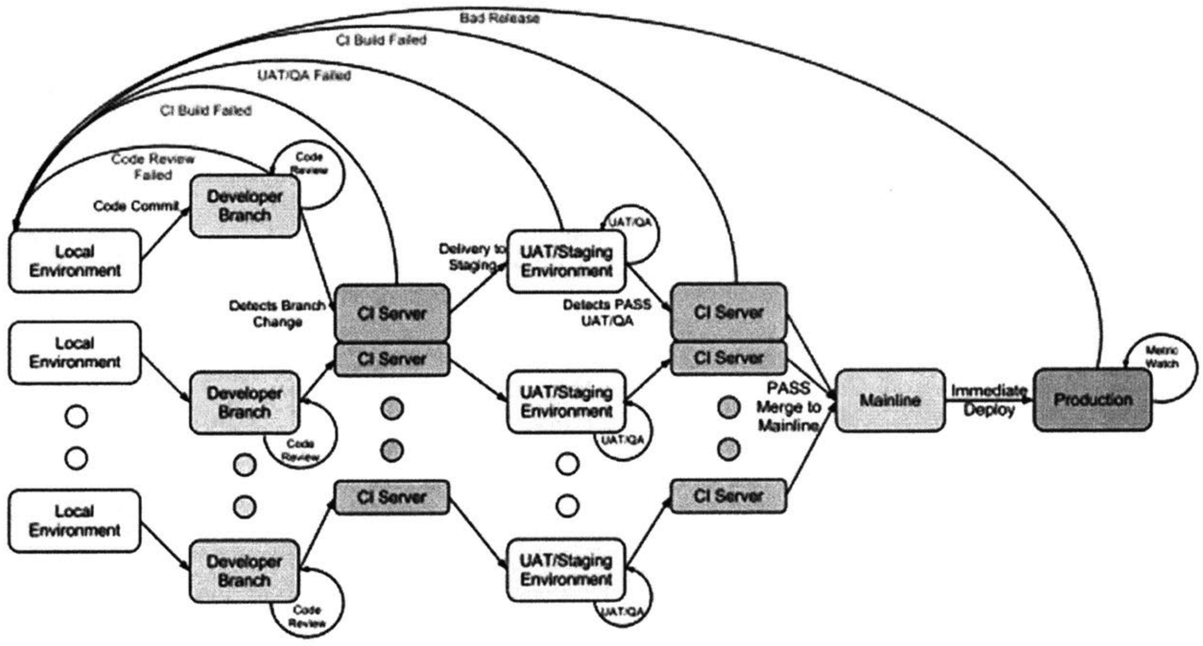


Figure 15 An illustration of a Continuous Deployment pipeline [40]

In the above workflow, once an engineer checks in code to the developer branch, this code is picked up and unit tested by continuous integration server., At this stage, based on the test results, it is determined if the change should be merged into staging environment, Once the code is deployed to staging environment, QA tests the environment, and decides to move it to production, and from here the continuous integration server can take over again and merges this code into the production environment.

The continuous deployment approach usually leads to a significant reduction in time, cost, and risk of deploying changes by pushing smaller, incremental and frequent updates into production.

This enables the frequent automated testing and ability to isolate, fix or roll back changes that are causing problems or are not well received by users. This also prevents wasted time, effort and resources used to develop a large set of features that will be eventually rolled back. This approach helps build the right product because frequent releases allow for the quick user feedback on the deployed features. As a result, it enables team to cut their losses on unwanted features and work only on the features that are proven through real user validations. This approach of incremental user validation helps to build an overall right product. Also, the risks related to the release process decreases significantly, and this process becomes more reliable. This reliability comes from the repeated use testing of the deployment process and associate scripts. This repeated use and testing helps to drive the bugs out from the deployment process and its associated scripts and this results into a more robust deployment process. Finally, as the releases become more frequent and more reliable, the need to push lot of code in each release decreases, this encourages smaller releases in which finding and fixing any bugs, whenever these occur, is much easier. This reduces the impact of these problems as these get fixed quickly. [41]

To enable continuous delivery and continuous integration, various facets of the delivery pipeline have to be automated. For example, automated testing. Every time the code is integrated into the main line, a whole set of automated unit and integration tests are run. Various processes such as pushing code through stages to final deployment are also automated with the help of a set of tools, such as Gatekeeper used by Facebook or Jenkins used by Netflix. These tools, such as Perflab and Claspin used by Facebook help monitor system health and identify issues early on. The use of the right automation tools helps minimize human errors and discover issues early on even when these issues are not obvious. Basically automation helps achieve two main objectives: first, minimizing manual steps, thereby reducing possibilities of human oversight or fatigue relater errors and bring consistent and reliability in the process; and second, automation enables the team to gather data from throughout the pipeline and this provides valuable insight that help improve product and processes.

One thing to note in continuous delivery is that even though it seems like the automation does most of the work, the accountability of the right code being deployed still lies with the engineer who developed the code. This automation simply makes their lives easier by assisting with the workflow.

For example, Facebook follows the philosophy of "Ship early and ship often". This philosophy has been a part of Facebook since the early days of Facebook when Mark Zuckerberg and other early engineers coded a lightweight process that allows to iterate quickly to get new features out. Today Facebook pushes new features, product improvements, and bug fixes to the live website every day. [42]

Continuous deployment at Facebook has some major benefits, Since the small changes to a new feature or a change can be live quickly, it can be validated quickly via A/B testing. Also, since in continuous deployment, each time, only a limited/small amount of new code in pushed through pipeline, this mitigates the risk of something major will go wrong. Since the changes are bitesize, these can be quickly rolled back. Smaller deployments are easier to debug since the changes are small and releasing often means that the code change would be still fresh in the engineer's mind who made the change. This drastically cuts down the debug time. Facebook trains new Facebook employees in the initial six-week boot camp about the practice of frequent deployment. In fact, Facebook enables and encourages its new developers to deploy new code within few weeks of starting in their positions, so that they are not hesitant to rapidly committing code into pipeline. This practice promotes innovation as developers develop an ability and mindset of fearlessness towards frequent deployment. [3]

In order to enable continuous deployment Facebook has a set of tools that automate the pipeline, for example, one of the tools that an engineer uses to know the state of a push by an engineer Facebook used bots. An engineer can query a bot to know the state of the push. The bot also keeps the engineer updated as the push is about to go live as the engineers would need to be available when this happens. The availability of engineer is required in case an unexpected problem arises. If the engineer is not available, the push is taken out to the deployment.

Facebook also uses Watir and Selenium as a test console to automate the testing of the push. This test console not only show which tests are broken, it also shows the drill down of when the test broke, and which change broke it so that the fix can be easier and quick.

Another important tool used by Facebook is Gatekeeper. Gatekeeper, as the name suggests, controls the push changes that get onto the live website. It also has the ability to control the deployment as to what segment of the users will see the changes. For example, via Gatekeeper Facebook can control the demographic such as East Coast users, any one between 15-35 years old etc. This gives Facebook an ability to do controlled releases and perform A/B testing on the new incremental changes. [3]

Another example of continuous deployment is Netflix. The goal at Netflix is to move fast and if fail, then fail fast and small. In order to do that Netflix has created an environment in which deployment to production is very easy and painless for engineers while maintaining a constant flow of information to the rest of the team about the state of the deployment. To this end, starting in 2009 Netflix has adopted a continuous delivery model, and deploys code to production more than a hundred times per day. Netflix achieves this rapid deployment without an extensive QA department or set of release engineers. Netflix has created a system which uses Platform as a Service model on Amazon AWS to enable each developer to deploy at their own schedule. [24]

Also in order to support continuous deployment Netflix has created infrastructure and delivery pipeline focusing on automation and insight in a way that make it easy to get the features in front of the users quickly instead of sitting on a developer's machine waiting to be deployed. Bite size delivery and deployment not only are easier to manage but also less risky as the amount of change and its impacts on the larger system is small. Also, this automated deployment pipeline exposes feedback from the pipeline about the code moving via the pipeline to production. This feedback enables Netflix to catch potential problem quicker and fix it before these finally land in production.

A third example is the use of Continuous deployment at IMVU. At IMVU, the code is deployed frequently and is integrated continuously. At IMVU, like Facebook, engineers are encouraged to commit early and frequently. Per Eric Reis, [18] the goal of continuous deployment is to minimize waste from the development process by bite sizing the work and also increase the cadence of the development teams. While there is a perceived dilemma between risking releasing imperfect code or waiting longer to double check everything, IMVU believes that this is a false dilemma. So, while IMVU accepts that they occasionally do negatively impact customers due to the rapid releases of features to the customers. Sometimes this manifests in putting out a bad feature, or introduces bugs in the production environment. But IMVU accepts these risks as a cost to the numerous benefits of releasing early and often.

At IMVU the development team members usually fall into two separate factions based on their inclination or preference of speed or quality. The factions are referred to as Cowboy and Quality factions. [18] Cowboys promote speed while quality faction wants to take time to focus on quality. While this looks like a partisan situation, it actually works for IMVU as it keeps both sides in check and ensures that it asks the right questions. Also, continuous deployment in this scenario facilitates learning for both the Cowboys, as well as the Quality factions, as this practice enables both sides to learn directly from the deployed code.

Also there are tools and systems in place to keep a check on the shipping-too-soon tendency. These systems such as a cluster immune system along with a continuous integration server, and the '5 Whys' practice are there to keep a check on Cowboys and provide feedback. However, because of the rapid feedback, Cowboys learn the type and extent of testing and validations that enables faster work. In effect, this practice and set of systems act like a check to the tendency of going too fast because they realize that quality problems ultimately cost time.

On the other side of the divide, the obvious disadvantages of waiting too long to release act as a check on the wait too long for Quality factions. Because the longer the engineer waits to release, the larger the conflicts become and harder it gets to reconcile these conflicts. Not only that, but waiting too long also increases the risk of delivering features or functions that may not be what customers really want, because it becomes harder to do quick tests by putting changes in the front of customer the longer the wait.

Also, continuous deployment at IMVU also has a great effect of employee morale as by reducing the overhead related to the release process. Since continuous deployment takes away the need for a lot of scheduling coordination efforts. An engineer can deploy as and when code is ready to be deployed and starts getting real feedback without depending on when the next release is scheduled. This cuts down on the level of traditional release management work, and this is usually welcomed by engineers, who get to work on their code instead of the coordination. It's much more satisfying to the engineers and boosts morale, which in turn improves quality.

6.4 Best Practice 4: Alternative QA/Automated Testing/Continuous Testing

Most of the successful organizations in this study have a small QA department. Prima-facie, this seems counterintuitive for a fast scaling organization, since QA cannot be skipped or avoided if high quality systems are to be created. On closer examination it is revealed that while the QA departments have been shrinking, the overall testing of the system has been increasing. This is achieved by automated testing. In most of these organizations in this study, there is a marked shift towards automated testing from traditional manual testing. These organizations have built a suite of automated tests that are executed every time something is pushed to the release. This set of automated test keeps getting better and more comprehensive as more testes are added to this set. Also since the testing is automated it can run continuously/frequently without incurring massive costs of manual testing.

Continuous automated testing can have many benefits over the manual testing, because it not only provides for an immediate feedback for the impact of the changes being pushed through the pipeline, but also ensures that each piece of code is tested and does not slip through the crack because of human oversight. Continuous automated testing also generates metrics that can be used to gain insights in the quality or speed/momentum trends which can be very useful to improve the efficiency of the development and deployment process.

Another departure from the traditional QA practice is that these organizations test their system continuously and use this generated data to dynamically improve the process itself. This is different than the traditional model of manually testing after development. To implement a continuous testing these organizations use a build as a part of the deployment pipeline. Using this build server with continuous automated testing not only runs the suite of integration tests, but also generates additional dynamic data related to the process which can be used to make the releases faster and improve the quality of code.

For example, at IMVU there is a strong emphasis on automated testing. Per James Birchler [15] IMVU uses a comprehensive set of automated tests as a part of continuous deployment process. This is supported with the infrastructure IMVU has built to support continuous deployment, This infrastructure includes systems such as BuildBot which is a continuous integration server that runs and monitors the automated tests at the time of each code push and a script that allows to safely deploy updates to the cluster. IMVU has also created a system called cluster immune system to continuously monitor any major regressions bugs that slip past the testing and create alerts. In case of an error or anomaly, the last push is undone and system is reverted back to the original state. In addition to this the practice of root cause analysis (5 Whys) to continuously improve the whole development and deployment process.

IMVU doesn't have an extensive Quality Assurance department entirely focused in testing. At IMVU it is the engineers themselves who write tests for everything they code. IMVU is able to do without an extensive manual Quality Assurance in the deploy process because of the scope, scale and thoroughness of automated tests. It's in thousands and continuously growing. Most of these tests are in-effect having browser automatically execute use cases and test behavior, using tools such as Selenium. Other than that tests are run to test classes, functions, web requests etc. IMVU heavily relies on these tests. This means that these tests must be reliable. This is why IMVU constantly curates these tests. But since all the bugs cannot be caught by the automated testes and some edge cases need to be tested manually IMVU does have a small but experienced QA team of engineers who test the system the way a real user would use it and test out the edge cases that could not be tested by the automated tests. Also these QA engineers improve the automated test suite by finding manual test that can be automated and thus added to the automated test suite. IMVU has formalized this step in their Scrum process to ensure this. Also IMVU encourages peer-review/demo practice that requires the requires developers to present their work to another person. This practice itself prevents bugs by making engineers more careful, since they would demonstrate it to another colleague, and also helps catch issues that slip because of the original engineer's familiarity-blindness of the issue.

Also, these QA engineers at IMVU work very closely with developers and keep testing in-flight features and provide feedback as the developers build their features. This helps to prevent bugs because by the time the feature formally reaches test engineers to test, it has already been tested multiple times and many of the potential bugs have already been resolved during this period.

However, despite all the automated and manual testing done prior to release, IMVU still relies heavily on automation and its infrastructure to do a rollout in a controlled manner to production, and there is an additional layer of protection as cluster immune system constantly monitors for regression bugs slipping through, thereby catching those bugs and reducing the risk of negative impact on real customers.

Finally, IMVU constantly listens for feedback coming in through various inbound channels such as community management and customer service teams. Engineering team has specifically assigned time to react to and fix any of the issues or bugs reported via this feedback mechanism so that these bugs definitely get fixed. [15]

So, in effect, IMVU commits early and commits often and every commit automatically runs all tests in an automated test suite that engineers keep updating. If the tests pass the code is deployed to the cluster and the website. Otherwise, it's back to the engineer who wrote the code to fix and try to redeploy. These tests suite takes only a few minutes to run then the code is pushed further downline eventually to the website. This means a new update is pushed to the website every few minutes. Also there is a manual validation of the edge cases that can't be caught by the automated suite. This combined with constantly improving test suite and regular A/B testing enables IMVU to deploy multiple times a day.

Another example is Netflix. Besides the use of automated tests and A/B testing Netflix also uses canary analysis to test its production code. The idea of canary analysis is that the code is deploys to a small subsection of the production infrastructure and is tested there. If no problem arises then the code is gradually moves to the larger section of infrastructure and eventually deployed fully. The basic idea is to fail on a small section instead of failing on entire fleet of servers in the infrastructure. [24]

Also Netflix have moved away from manually analyzing results from the canary deployment to a set of automatically generated metrics that compares the baseline metric with the canary metric on thousands of parameters and that alert the humans if something stands out and found that code is not ready for larger production deployment. This although not a fool-proof system, does give Netflix a sense of the production readiness of the code. [24]

Another practice used by Netflix to increase robustness of the system is the use of "The Simian Army" [23]. The software "monkeys" of this simian army are created to force random disruption in the system such as introducing random latency in system of disrupt servers. These random failures expose the vulnerabilities of system and encourages engineers at Netflix to build a robust and failure resistant system. This practice also fosters a culture of resilience in the organization to deal with the disruptions whenever they happen.

A third example is Facebook: One of the important practices that Facebook follows is the Dogfooding. Dogfooding is a recent industry term which means eating your own dogfood, i.e. using your own product. Facebook encourages its engineers to be users as well. Basically, Facebook employees use an experimental build based on the latest code all the time to access social network. This includes changes that haven't been rolled out to the main production trunk yet. This enables employees to dogfood its own changes and discover any issues early on even before the code is pushed out to the rest of the world. Also since this experimental site is open to all of the employees, the changes get tested more intensively as the users are employees and have more intimate knowledge of the system. This experimental site also has bug management tool inbuilt to report any issues easily and immediately.

Also, the nature of the online platform with its enormous user base and use of continuous deployment enables Facebook to extensively use user feedback mechanism. Facebook regularly runs live experimentation i.e. A/B testing on sections of its user base. If a new feature needs to be tested for its fit to the user base, this feature is deployed to a small subsection of the servers and expose to a specific segment (or set of segments) of users. This generates an enormous data set for Facebook to analyses and see if the feature is further roll-worthy.

The above is apart from the mainstay of QA at Facebook which is a set of automated tests that are run regularly to avoid regressions bugs and to identify majority of the issues. These automated tests simulate the user interaction and identify if the website doesn't behave as expected. These tests also act as a sanity check against common errors so that engineers can focus on finding and fixing issues not easily uncovered by these automated testes.

Another innovation that Facebook uses to catch bugs that escapes its automates testing, A/B testing or Dogfooding, is the use of crowd sources QA. Facebook has a bug bounty program that rewards members of the general public, who report a bug in Facebook, with money and recognition. This program has been very successful and has encouraged smart people from across the world, especially, Russia, india, brazil and UK to report hundreds of bugs that went undetected.

6.5 Best Practice 5: Innovative communication/organization structures and management practices

The innovative architecture, design, deployment and QA practices at the organizations in this study, are also reflected in their organizational and communication structures, culture and management practices. This is in line with Conway's Law which states that the organizations designing systems are constrained to create designs which mirror the communication structures of these organizations. [43] For example, Spotify has a model of squads that function as mini-startups and follow the paradigm of "Think it, Build it, Ship it and Tweak it". [37] These squads along with their system owners are fully empowered (with alignment to the broader set of rules) to make decisions and are full-stack teams, however at the same time these squads are also fully responsible for their feature and system and it anything goes wrong, are supposed to fix it. This is a reflection of the independent microservices based architecture at Spotify. Since each of these microservices are decoupled and can be independent maintained, the organization structure of squads and the culture of mini-startups becomes an excellent fit to the microservices architecture.

This mirroring of architecture and design in culture is also present at Netflix which has a culture of "Freedom and Responsibility." [25] At Netflix employees are empowered to make decisions, and at the same time are also held responsible for what they write, and not only quickly fix it if anything goes wrong but also perform a detailed root-cause analysis of what went wrong and how to prevent it in future. This goes hand in hand with the heavily microservices driven architecture at Netflix which supports independence and also with the DevOps paradigm at Netflix.

One thing to note is that this autonomy is not arbitrary and these organizations do provide a set of best practices to be followed and a set of parameters under which to deliver so that autonomy doesn't create chaos. For example, Spotify balances autonomy with alignment [36] across organization. To achieve this balance, Spotify has a set of rules, or a set of best practices, which although are not strictly mandatory, are encouraged. The systems at Spotify are organized in such a way that it becomes hard to move away from these recommended rules. This way if engineers are still free to work the way they want, but they usually have a very good reason to do so. This maintains autonomy while keeping the chaos at bay. In addition, Spotify also maintains autonomy by having a concept of Servant Leadership. At Spotify, Leaders do not dictate what to do, instead leaders simply articulate what needs to be done and set the parameters around that goal. The leader then trusts the team to deliver the goals within the parameters. If the team does not deliver, usually it's considered that possibly the goal was not articulated well or that the parameters were not adequately defined. This not only makes the leaders clear and accountable but also make team accountable with autonomy.

Also, at all of these organization engineers are responsible for their piece of code from when it starts through its production deployment and beyond. This is very close to the DevOps paradigm even if it's not always called DevOps at some of these organizations. For example, Facebook's DevOps culture in which developer who writes the code is also the one who supports it. This is one of the basic premises of DevOps paradigm and this promotes a culture of accountability and prevents lower quality code, or buildup of technical debt. [42] Another example of DevOps setup is Spotify where not only the squads are fully responsible for their feature and system owner responsible for the architectural integrity of the system, for certain critical systems that are critical from the operations perspective, Spotify has more than one system owners, with one being from the operations side and another from the development side. This is a DevOps setup and helps keep the team that developed and owns the feature also as being responsible to maintain it.

Another characteristic of these organizations is the high quality of workforce. All of these organizations focus on hiring and retaining best quality of talent. For example, Netflix has a huge focus on attracting and retaining best talent and it does this by offering salaries higher than what the market would pay for the same position. In turn Netflix entrusts its engineers with expectation of high quality work. At Netflix, employees producing A level work with B level work are rewarded and promoted, while employees producing B level work with A level effort are given a generous severance package. Similarly, engineers at Facebook are expected to commit code within their first few weeks of starting at organization or team and quality is emphasized as evident form the use and tracking of engineers' 'karma score' which Facebook uses to track and maintain quality in work. A 'karma score' is assigned to each engineer and can go up and down based on the quality of work. Every time a code breaks build or is found buggy, the relevant developer loses some karma points. While these karma scores do not directly affect job security of the engineer, it does act as a red flag and causes closer reviews of the engineer's work.

Another practice is to organize and structure teams for optimal performance and facilitate better flow of information, continuously learning and enable coordination among teams. This is done via setup of discipline/competency groups that cuts across the teams that focuses on the improving skills related to the craft without sacrificing focus on delivery, or via sharing outcomes and learnings from the root cause analysis of failures across organization when something goes wrong and. For example, Spotify has the paradigm of tribes, squads and guilds. By organizing development team across organization in this innovative way, Spotify not only establishes ownership and accountability for the product with the squad, but also maintains cross team coordination via tribes, and promotes the expertise in various disciplines via guilds.

Another way the organization keeps on continuously learning is by conducting deep postmortems/reviews of issues when things go wrong to find out the root cause of the problem, and then sharing the results across organization. This root cause analysis not only prevents future issues, but also provides a great learning opportunity for the organization. However, care should be taken to avoid blame assignments during the root-cause analysis as it can derail the whole process. For example, at IMVU when something goes awry, the culture is to not view it as a crisis or look for someone to blame. They see it as a learning opportunity and use the root cause analysis technique such as '5 Whys', to learn about the underlying issues that manifested as the problem. Each defect or failure is seen as an investigation opportunity to learn the deficiencies of the process, and improve it. By continuously learning, tweaking and improving, the process and system become progressively more stable, robust and it prevent problems from occurring and this enables the releases to go faster over time.

By utilizing innovative ways to organize teams and aligning individual's incentives with the right objectives of the organization can go a long way towards maintaining fast release pace without sacrificing architecture and without building technical debt.

7. Conclusion

Even though the dilemma of scale vs speed seems real for many startup organizations, the organizations in this study proves that this is a dilemma that can be reconciled and overcome. It is possible to scale up while maintaining speed as organizations come out of the startup phase and move into more mature phase. Although the practices followed by these organizations are not identical, and cannot be said to fit a definite mainstream model or development methodology completely, these successful organizations do have a common theme of best practices that has emerged from this study.

Most of these practices can be categorized under three broad categories.

1. Adopt a highly modular and granular service oriented architecture
2. Minimize the work in progress by adopting continuous integration and continuous delivery and by automating testing and delivery pipeline
3. Maintain a culture of independence with accountability; this mirrors and supports the above architecture, design, deployment and QA practices at these organizations

While the practices followed by these organizations may not fit for every startup organization, these do serve as an important and useful set of recommendations to pick and choose from when trying to scale up while maintaining speed. One thing to keep in mind is that these best practices are not simply an implementation of certain tools or frameworks as these may require a change in not only the way engineers do their work, it also requires a paradigm shift for the whole organizations. However, if implemented correctly, these best practices could be very useful to scale at speed while maintaining a quality architecture. This not only reflects in the customer satisfaction and market share capture, but also helps keep employees happy and motivated.

Finally, there are some limitations of this study, which can be the basis for suggestions for further investigation.

First, the organizations in this study are largely B2C, web based startup organizations that have grown from a small system with few users to a large scale system with a big user base in last one or two decades. As a result, these organizations have been able to use practices such as continuous deployment, experimentations and innovative organization setups etc. from the outset. Further work can be done to see if these practices are effective even in the B2B or B2B2C kind of organizations or systems that are not fully deployed on organization controlled servers.

Second, this study is limited in scope for studying startup organizations that have scaled up while maintaining speed. Although it can be hypothesized that the best practices outlined in this study are useful even for the incumbent organizations that are already at large scale and are trying to speed up. This is demonstrated by organizations such as Netflix which have grown with a monolithic architecture only to hit a wall in terms of speed at larger scale, and then implemented these practices to re-architect its systems to bring back the speed while still growing. However, the study of incumbent organizations trying to move from scale to scale with speed was not the primary focus of this study. Further investigation can be made to confirm if these practices, as outlined in this study for startup organizations, remain valid for incumbents as well.

8. References

[1] Facebook, "Facebook-Company Info," 2016. [Online]. Available: https://newsroom.fb.com/company-info/.

[2] Statista.com, "Statista," 2016.

[3] E. F. a. K. L. B. D. G. Feitelson, "Development and deployment at Facebook," *IEEE Internet Comput. 17(4),* pp. pp. 8-17, Jul-Aug 2013.

[4] Sean Lynch-Facebook, "Monitoring cache with Claspin," September 2012. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/monitoring-cache-with-claspin/10151076705703920/.

[5] J. Mark, "How Proper "Dogfooding" Might Have Saved Facebook Home," 2013. [Online]. Available: https://www.fastcompany.com/3012670/how-proper-dogfooding-might-have-saved-facebook-home.

[6] Facebook Developers, "F8 2015 - Big Code: Developer Infrastructure at Facebook's Scale," March 2015. [Online]. Available: https://www.youtube.com/watch?v=X0VH78ye4yY.

[7] Facebook Bug Bounty, "Facebook Bugbounty," 2016. [Online]. Available: https://www.facebook.com/BugBounty/.

[8] E. Mills, "Facebook hands out White Hat debit cards to hackers," December 2011. [Online]. Available: https://www.cnet.com/news/facebook-hands-out-white-hat-debit-cards-to-hackers/.

[9] Facebook Bug Bounty, "Facebook Bug Bounty," October 2016. [Online]. Available: https://www.facebook.com/notes/facebook-bug-bounty/facebook-bug-bounty-5-million-paid-in-5-years/1419385021409053.

[10] R. Paul, "Exclusive: a behind-the-scenes look at Facebook release engineering," 2012. [Online]. Available: http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/.

[11] IMVU, "IMVU Faq," 2016. [Online]. Available: http://www.imvu.com/about/faq.php.

[12] J. B. T. F. Brett G. Durett, "But Does it scale: Evolution of Lean at IMVU at Startup Lessons Learned Conference 2010," 2010. [Online]. Available: https://www.youtube.com/watch?v=NAw7xK3uGpU&feature=youtu.be.

[13] E. Ries, "Startup Lessons Learned-Why Continuous Deployment?," June 2009. [Online]. Available: http://www.startuplessonslearned.com/2009/06/why-continuous-deployment.html.

[14] E. Ries, "Continuous deployment in 5 easy steps," March 2009. [Online]. Available: http://radar.oreilly.com/2009/03/continuous-deployment-5-eas.html.

[15] J. Birchler, "IMVU'S APPROACH TO INTEGRATING QUALITY ASSURANCE WITH CONTINUOUS DEPLOYMENT," April 2010. [Online]. Available: https://engineering.imvu.com/2010/04/09/imvus-approach-to-integrating-quality-assurance-with-continuous-deployment/.

[16] T. Fitz, "Continuous Deployment at IMVU: Doing the impossible fifty times a day," February 2009. [Online]. Available: http://timothyfitz.com/2009/02/10/continuous-deployment-at-imvu-doing-the-impossible-fifty-times-a-day/.

[17] E. Ries, "Five Whys," November 2008. [Online]. Available: http://www.startuplessonslearned.com/2008/11/five-whys.html.

[18] E. Ries, "Why continuous deployment," June 2009. [Online]. Available: http://www.startuplessonslearned.com/2009/06/why-continuous-deployment.html.

[19] Sandvine Report, "Global Internet Phenomena Report," Sandvine, 2016.

[20] Netflix, "Final Q3 Letter," 2016. [Online]. Available: http://files.shareholder.com/downloads/NFLX/2992434071x0x912075/700E14FD-12BE-4C3A-9283-9A975C7FE549/FINAL_Q3_Letter.pdf.

[21] A. Bylund, "Netflix, Inc. Subscriber Growth Explained in 4 Charts," April 2016. [Online]. Available: http://www.fool.com/investing/general/2016/04/26/netflix-inc-subscriber-growth-explained-in-4-chart.aspx.

[22] SmartBear Software, "Why You Can't Talk About Microservices Without Mentioning Netflix," December 2015. [Online]. Available: http://blog.smartbear.com/microservices/why-you-cant-talk-about-microservices-without-mentioning-netflix/.

[23] Z. Hemel, "How Netflix Deploys Code," June 2013. [Online]. Available: https://www.infoq.com/news/2013/06/netflix.

[24] B. Schmaus, "Deploying the Netflix API," August 2013. [Online]. Available: http://techblog.netflix.com/2013/08/deploying-netflix-api.html.

[25] R. Hastings, "Netflix Culture-Freedom and Responsibility," August 2009. [Online]. Available: http://www.slideshare.net/reed2001/culture-1798664/.

[26] Spotify Team, "Spotify News," October 2008. [Online]. Available: https://news.spotify.com/us/author/spotifyteam/.

[27] M. Murgia, "Spotify crosses 100m users," June 2016. [Online]. Available: http://www.telegraph.co.uk/technology/2016/06/20/spotify-crosses-100m-users/.

[28] J. Kahn, "Spotify hits 40 million paid subscriber milestone, outpacing Apple Music's growth," September 2016. [Online]. Available: https://9to5mac.com/2016/09/14/spotify-40-million-subscribers/.

[29] J. I. Wong, "Spotify's average salary keeps rising—even as its losses mount," May 2016 . [Online]. Available: https://qz.com/691188/spotifys-average-salary-keeps-rising-even-as-its-losses-mount/.

[30] M. Harasymczuk, "Spotify Engineering Culture part 1 (Agile Enterprise Transition with Scrum and Kanban)," April 2014. [Online]. Available: https://www.youtube.com/watch?v=Mpsn3WaI_4k&list=PL7ND_Jm7TtuTj4hi4aCQ1sJN3hL8a7LFw&index=6.

[31] M. Harasymczuk, "Spotify Engineering Culture part 2 (Agile Enterprise Transition with Scrum and Kanban)," October 2014. [Online]. Available: https://www.youtube.com/watch?v=X3rGdmoTjDc&index=6&list=PL7ND_Jm7TtuTj4hi4aCQ1sJN3hL8a7LFw.

[32] M. Linander, "What is a high level overview of how Spotify uses Docker and Helios in their code test deploy workflow," October 2015. [Online]. Available: https://www.quora.com/What-is-a-high-level-overview-of-how-Spotify-uses-Docker-and-Helios-in-their-code-test-deploy-workflow.

[33] K. Goldsmith, "GOTO 2015 - Microservices @ Spotify," December 2015. [Online]. Available: https://www.youtube.com/watch?v=7LGPeBgNFuU&list=PLEx5khR4g7PJwebFxZrJmgqw_7Dy4nAmq&index=2.

[34] B. Linders, "Microservices at Spotify," December 2015. [Online]. Available: https://www.infoq.com/news/2015/12/microservices-spotify.

[35] J. Vanian, "How Spotify is ahead of the pack in using containers," February 2015 . [Online]. Available: https://gigaom.com/2015/02/22/how-spotify-is-ahead-of-the-pack-in-using-containers/.

[36] H. K. &. A. Ivarsson, "Scaling Agile @ Spotify with Tribes, Squads, Chapters & Guilds," Henrik Kniberg & Anders Ivarsson, 2012.

[37] H. Kniberg, "How Spotify builds products," http://www.slideshare.net/ssuser6cf9c3/how-spotifybuildsproducts, 2013.

[38] R. Rowan, "Stevey's Google Platforms Rant," 2002. [Online]. Available: https://plus.google.com/+RipRowan/posts/eVeouesvaVX.

[39] Agile Alliance, "Agile Alliance glossary -TDD," 2016. [Online]. Available: https://www.agilealliance.org/glossary/tdd/.

[40] M. Chletsos, "Continuous Delivery vs Continuous Deployment vs Continuous Integration - Wait huh?," November 2012. [Online]. Available: https://blog.assembla.com/assemblablog/tabid/12618/bid/92411/continuous-delivery-vs-continuous-deployment-vs-continuous-integration-wait-huh.aspx.

[41] L. Chen, "Continuous Delivery: Huge Benefits, but Challenges Too," *IEEE Software*, pp. 50 - 54, Mar.-Apr. 2015.

[42] C. Rossi, "Release engineering and push karma," April 2012. [Online]. Available: https://www.facebook.com/notes/facebook-engineering/release-engineering-and-push-karma-chuck-rossi/10150660826788920.

[43] M. E. Conway, "How Do Committees Invent?," *Datamation magazine,* April 1968.