Legacy Systems & DevOps

Toni Karppi and Fredrik Åhs

April 2019

Contents

1 Introduction **3**

2 Legacy Systems **3**
2.1 Tightly coupled environment 3
2.2 Lack of standards 4
2.3 Late quality control 4
2.4 Little or no testing 5
2.5 Sparse release schedule 5

3 Mitigation of Problems **6**
3.1 Continuous integration 6
3.1.1 Version control 6
3.1.2 Build automation 7
3.1.3 Test automation 7
3.2 Microservices 7
3.3 Self-service provisioning 7
3.4 Process and culture 8

4 Conclusion **8**

1 Introduction

The role of safety and maintainability has not always been as central to the system development process as it is today. This fact has lead to the development of systems that are tightly coupled to their environments, error-prone, and hard to change without breaking things. Unfortunately, there are still many such systems running in production today. In this essay we will refer to these systems as "legacy".

On the other hand, we have DevOps practices that are focused on reliability, fast delivery, and automation. It would be nice if we could port over some of these methodologies to legacy systems, as this would lessen the maintainability burdens on these projects. Because of the sheer complexity of some legacy systems, it may be very difficult – or even impossible – to adopt the full DevOps mindset and methodologies in these projects. This does not however mean that we should not consider implementing these methodologies at all.

In this essay we will try to highlight some methods that have been used in industry to lessen the burden on maintaining legacy systems. All the methods we will present will at most only have minor impact on the actual implementation of the systems themselves. The focus will be on improving the quality of the project, while still keeping the core parts of the project intact.

2 Legacy Systems

The Merriam-Webster dictionary defines *legacy* as:

> "of, relating to, or being a previous or outdated computer system" (merriam-webster.com, 2019).

This definition is pretty general, so it can be difficult to say exactly what a legacy system should look like. So instead of focusing on the systems as a whole, we'll be focusing on properties that legacy systems tend to have. In the rest of this section we'll talk about some properties of legacy systems that may make it difficult to maintain them.

2.1 Tightly coupled environment

Legacy systems are often very tightly integrated to their environments (Ismail, 2017). They often have specific requirements of what operating system

the programs should be running on; and also what libraries and which versions of these libraries should be installed on the production machine. For example, a project may require that an outdated version of a library needs to be present when the project is being compiled, and this library is only known to work on Windows XP.

When taking into account that all developers working on the project must satisfy these requirements, it's clear that strict platform dependence has some real side effects. This restricts the freedom of choice for the developers, which may hinder their productivity. It may also be a lot more expensive to pay for support for tools that are not widely used anymore.

Having strict version requirements can also introduce security vulnerabilities. It may not always be possible to reliably upgrade packages or other software that the system depends on. This is because it is often difficult to verify that nothing has broken after an upgrade.

2.2 Lack of standards

Many legacy systems have been around for decades, and they've seen supporting tools come and go during their years. Different tools and scripts end up being used to build are otherwise manage the project, which leads to inconsistent and fragmented methods for managing the project (Rao, 2018). For example, there may be multiple different build scripts or tools that are supported, since those were used in the project once upon a time. Or there may be code in the system for implementing support for some obscure compiler.

All these things increase the complexity of the the system, and may lead to some hard to find bugs, or just slowing the development in general. Fixing such bugs may require having the original developer of that feature work on that problem, since they may be the only ones who know the particularities of that code. However, that is not always possible – the person or people who added or used these particularities may no longer be working on the project.

2.3 Late quality control

When working on these types of systems, it is common for developers to write code for a project, and when the project is ready for release, it is sent over to the operations team for quality assurance. This can be problematic, since it has been shown that the cost of fixing a problem increases exponentially with how late in the development stage the problem is detected (Planning, 2002). The QA process is really late in the development process, which means there

is an unnecessary waste of resources for fixing bugs that may have been able to be detected much earlier using modern development practices.

2.4 Little or no testing

It is not uncommon to find that there are only a few test cases – or even none at all (Rao, 2018). This problem is related to the problem "late quality control", but this problem is more fundamental. When the development of a project is just getting started, it is often not that difficult to manually try out some inputs to check if the program works; but as the complexity of a software system grows – the effort required to get assurance of reliability using manual testing becomes too high to be viable. This can make the quality assurance process very cumbersome, and it will often lead to several bugs making their way into production, at which case the cost of fixing a bug is at its highest.

2.5 Sparse release schedule

Legacy systems tend to have fairly long periods between releases. This is a natural consequence of having the developers and operations teams be departmentalized and isolated from each other.

One consequence of this is that it may take a long time before a security vulnerability or some other bug gets patched. This may have serious consequences, as once a vulnerability becomes known, there's a high probability that it will get exploited. If the vulnerability is serious enough, this may force an unplanned release. This may not be an easy task, since deployment in legacy systems can be a very tricky process. One reason for why deployment is so difficult in legacy systems is because of the number of introduced features (and other changes) between two deployments. Unplanned releases may have an effect on the expected release schedule, where the planned releases get their release dates pushed to a future date; or the planned releases may get some features stripped because of lack of development time.

Another consequence of having a sparse release schedule is that the feedback loop is very poor. By having long delays between releases, it can be difficult to know if a newly added feature had a positive effect or not. There might exist a better implementation of the new feature, but there's no way of verifying this. The argument that a developer should know what is the right thing to do doesn't really work in this case, since there is a lot of subtle psychology mixed in the decisions for design and implementation. For example, the color of a button can have a real impact on the sales of

some product (Porter, 2011), and it's not really clear what this color should be using intuition alone.

3 Mitigation of Problems

It is fair to assume that these problems with legacy systems were not intended — or, at least not intended to be problems — but are likely either artifacts of the development practices under which the system was developed, or just due to system incompatibility. With that in mind, what steps can be taken to minimize the risk that the newly started projects of today will be the legacy systems of tomorrow? And how can DevOps be adopted for legacy systems that are still in development?

3.1 Continuous integration

Essentially, continuous integration (CI) is the practice of frequent integrating of development branches into the mainline of some version control system in order to avoid integration problems that might arise when integrations are far between. In order to minimize the risk that breaking changes are integrated into the mainline, the source code (with the new integration) should typically be automatically checked out and built on all supported platforms (most likely on a build server) to ensure compatibility. The same automation should also verify that unit tests passes and may additionally measure code coverage, run static code analysis and confirm that the code adheres to any style guides used. Enabling continuous integration for any non-trivial legacy systems will most likely be unfeasible to do in one step and should instead be done in several smaller steps.

3.1.1 Version control

Ideally, the first one would be to move or introduce the system to modern version control. Version control are obviously a requirement for continuous integration, and there are plenty of both centralized- and distributed version control systems available, out of which Subversion (SVN) and Git are the most widely adopted. And out of those two, data seems to indicate that Git are more popular (stackexchange.com, 2019). Whether to use SVN or Git depends on the needs of the organization but choosing either should be a safe bet due to widespread usage.

3.1.2 Build automation

While in the process of updating version control for the system, the build requirements can be identified and the build processes can be moved into a standardized build management tools such as Maven or Gradle which opens up for build automation.

3.1.3 Test automation

Test automation in legacy systems can prove a significant challenge since test coverage usually is very low or non-existent. By working with the business team, some core test scenarios can most likely be identified which should be prioritized by, for example, business critically. These can then be implemented in order of priority and then serve as large scale regression tests. Whenever integration breaks something, more regression tests can be added to cover that specific scenario. Over time, with the addition of unit tests to all new additions, code coverage will gradually increase.

3.2 Microservices

Legacy system tends to be monolithic by nature but further development of the system doesn't necessarily have to follow that path. Instead, when adding features, these can be developed as microservices. That is, small modular components, interacting with each other to form a larger application (Avidan, 2019). Here, the legacy system continues to serve data and doing what it always has done, while new features can be written using modern tools and languages without the overhead of navigating legacy code. But enabling modern development tools for legacy systems are mostly a biproduct of using microservices. Since services are loosely coupled each can be developed semi-individually, the real benefit comes from the increased deployability and modifiability, meaning the simpler and faster deployment procedures and easier development due to the decoupled nature of each service (Chen, 2018).

3.3 Self-service provisioning

Outside CI build automation, developers might need to run or test some part of the legacy system with a outdated but tightly coupled environment. Instead of having the developer to work within the outdated environment directly, virtualization is most likely used. However, virtual environemnts are not always readily available to developers but are traditionally managed by

the IT department and the process of getting access to the correct environment might be cumbersome and unnecessarily time consuming (Schindler, 2019).

With the use of tools such as Ansible or Terraform (and, arguably, Docker and other container platforms), configuring and deploying virtual resources are as simple as writing a configuration file (Woods, 2019). Enabling developers to a self-service virtual environment reduces the need to run outdated operating systems on development machines while it facilitates easily reproducible build and runtime environments.

3.4 Process and culture

In the end, teams working with legacy systems are likely to be accustomed to a work flow where features are developed in isolation for extended period of times and releases are few and far between. Adopting the technologies and practices of DevOps is probably impossible without an organization that supports it. Therefore, perhaps before anything else, the organization should shift towards more agile methods.

4 Conclusion

Adopting DevOps for legacy systems can prove to be challenging indeed. But a lot of the problems associated with legacy systems can — at least partially — be mitigated by agile and DevOps and,in the long run, even bring legacy system up to speed by migrating to modern technologies such as microservices (Chen, 2018). The suggestions above are but a few aspects of what DevOps can bring to legacy systems and obviously, what works for one system might not for another.

However, before deciding on what new technology to introduce, one should make sure that the organization can sufficiently support it.

References

Avidan, Zeev (2019). *How to Bring Legacy Systems to DevOps Speed*. URL: `https://thenewstack.io/how-to-bring-legacy-systems-to-devops-speed/` (visited on 04/30/2019).

Chen, Lianping (2018). "Microservices: Architecting for Continuous Delivery and DevOps". eng. In: *2018 IEEE International Conference on Software Architecture (ICSA)*. IEEE, pp. 39–397. ISBN: 9781538663981.

Ismail, Nick (2017). *Containing the legacy application challenge of Windows XP.* URL: https://www.information-age.com/containing-legacy-application-challenge-windows-xp-123469046/ (visited on 04/30/2019).

merriam-webster.com (2019). *Definition of legacy (Entry 2 of 2).* URL: https://www.merriam-webster.com/dictionary/legacy (visited on 04/30/2019).

Planning, Strategic (2002). "The economic impacts of inadequate infrastructure for software testing". In: *National Institute of Standards and Technology.*

Porter, Joshua (2011). *The Button Color A/B Test: Red Beats Green.* URL: https://blog.hubspot.com/blog/tabid/6307/bid/20566/the-button-color-a-b-test-red-beats-green.aspx (visited on 04/30/2019).

Rao, Gangadhar Hari (2018). *DevOps for Legacy Systems – The Demand of the Change Applications Landscape.* URL: https://www.infosys.com/IT-services/application-development-maintenance/white-papers/Documents/devops-legacy-systems.pdf (visited on 04/30/2019).

Schindler, Esther (2019). *Self-service provisioning 101.* URL: https://www.hpe.com/us/en/insights/articles/self-service-provisioning-101-1705.html (visited on 04/30/2019).

stackexchange.com (2019). *Are there any statistics that show the popularity of Git versus SVN?* URL: https://softwareengineering.stackexchange.com/a/136207 (visited on 04/30/2019).

Woods, Emily (2019). *Infrastructure as Code, Part One.* URL: https://crate.io/a/infrastructure-as-code-part-one/ (visited on 04/30/2019).