Lessons Learned as Solution Arch

I think I’ll continue to update this as I learn. I don’t know much now but that will change over time. I have to start somewhere.

I could research the topic and make a nice little sanitized blog but that’s not coming from me and my experiences and would not be a unique contribution. If you are interested in this topic then you need other sources of information than this running blog post, to be well rounded.

## Lesson: If you give stupid requirements you will get stupid code

Have enough respect for your programmers to make sure they get real requirements. Not requirements like, “Make the new application do exactly what the old application does”.

Requirements should be of good quality but if they are not of good quality at least make some attempt at making them better than the example above.

One fun side effect is a bug in the old app will be recreated in the new app. That’s fun.

## LESSON: Strive for World Class quality

Many programmers doubt themselves or are plain lazy. If your development shop is small they will suggest you are too small to deliver world class software. If your organization is larger there will be other excuses, “We are not NASA”.

I have one word for that “Bull shit”. Ok, that’s two works but you get my meaning.

One developer can deliver world class software. One hundred developers can do the same. We are all professionals and should be improving our craft. I expect no less from a lawyer in a law first of one as I expect from a law firm of one hundred.

Check this out: <https://www.youtube.com/watch?v=EE-t5J7hnHE>

My view of a World Class Software organization is different than this speaker. He does the measure of the software organization to the performance of a company. It’s important to hear different views and form your own opinion.

From the video: The most self-critical organizations have the best quality and the least critical organizations have the least quality. Steve tells a story about a company trying to improve their estimation process. At the end Steve couldn’t find anything wrong and asked what their success rate was. They said 97% success but they wanted to figure out how to get the last 3%.

Teams can function effectively until around 7 members. Steve discusses this around 58:00

## LESSON: Have a cadence

A cadence in everything you do is essential to staying fresh and keeping systems running smoothly.

What I mean by cadence is keep regularly scheduled intervals for certain tasks.

Examples:

Deployments should be on a cadence. Over time this creates a comfortable deployment story and likely leads to more automation. Users become accustomed to change and have an expectation on when to see new features and improvements.

Checking for 3rd party updates should be on a cadence. It’s far cheaper to make many small changes than a few large changes. I have a scheduled task, in my calendar, to initiate a search for all new components. For the developer this could include NuGet packages, NPM packages, JS libraries, IDE updates, other developer tool updates such as source control tools, etc. This list can go on but you get the point. If you are worried about the frequency of change and the impact that has on code quality then you probably don’t have a good build/automated test story.

If you don’t deliberately maintain some kind of cadence then you’ll find yourself re-writing code because the gap between changes are to great or you’ll simply become stagnate as a development shop.

## LESSON: Show commitment, make across the board change

If your team makes a decision to change something then make a project out of it and change everything the same way.

If you are moving toward a new server version then don’t change the server for one application. Update all your servers.

If you are changing the namespace of new projects then take the time to change the namespace of all your projects. Even if a better technique is discovered don’t implement it until you’re committed to changing the technique in all your code. It’s better to do a thing the same way everywhere so if a fix or better approach is discovered it’s easier to change.

As a result you’ll develop a culture that is used to touching all the code, ideas are more thoroughly vetted, code is not left behind, and making a change many times now is cheaper than making the same change many times over time.

This may cause up front work but will significantly reduce “technical debt” in the long run.

## LESSON: Don’t be proud of your complex environment

Only you will be impressed by an environment so complex that only you understand it and you become the indispensable man.

Grave yards are full of indispensable men.

True genius is in simplicity. Taking a complex situation and massaging it down toward simplicity is a true mark of genius and value add.

If a highly complex environment is a 5 then work hard to make it a 2 because technology evolution will bring it right back up to a 5 quickly. If you aren’t constantly working toward a 2 then that 5 becomes a 7 and soon a 9. Suddenly you have an environment so complex that the smallest change is too expensive to deliver.

If you cannot leave your company to be promoted to another position because of the complexity you’ve created or allowed, then you’ve failed. Try again or get out of the leadership business.

## LESSON: Don’t frustrate innovation

Technology is released at such a rapid pace it’s almost impossible to keep up on a single stack. It’s impossible to keep up on all stacks.

Other developers in your organization will be turned onto technologies you’ve summarily ruled out and don’t fit into your long term plans. Possibly other developers are innovating with technologies you’ve chosen but you’re learning all the technologies and that other programmer is only learning the one technology. You find yourself wishing they would slow down.

Don’t frustrate innovation. Don’t hold back your developers.   
  
It’s far less expensive to course correct than create innovators. Let innovators innovate. Don’t frustrate them because you aren’t ready.