How to Deploy Your Product When You Are Not 100% Confident it Will Work as Advertised

At some point in every software engineer’s career you will be asked, no forced to push code to production even though you and your team know it’s not ready. In this post I want to discuss my experiences deploying products when you are not 100% sure everything will work as expected.

## Background

While working for a large insurance company I was part of the typical big-co enterprise IT environment. One of the projects I led was to replace a legacy back end system and vendor with a new system and vendor. Sounds simple enough right. The project started off like most projects with a decent amount of research into the existing system to see how it worked. The problem we were solving was not particularly difficult from a technical perspective. We were simply taking requests sent to us from a customer facing application and sending the requests to a new 3rd party vendor, then sending the result back. For any decent software team this is easy, and thus I was not worried.

## Very Little Worry

After spending a sprint (2 weeks) diving into the existing legacy system, I quickly realized that there was a decent amount of business logic peppered thought the system. My first concern was that the business logic was not centralized into its own area, rather it was everywhere. In controller endpoints, in the data layer, and everywhere in-between. I figured ok let’s identify all the business logic, pull it out and centralize it somewhere. Only later in the project did I realize there was more business logic, but it was not in the legacy system I was replacing.

## Realizing Business Rules Can Be Complicated

Building a working replacement for the legacy system was not hard at all. At the time my team followed corporate development best practices, which meant we replaced a horrible legacy monolith with a well architected monolith. During this time frame, mid to late 2010’s the enterprise was just starting to venture into the microservice and Kubernetes world. Since all the new fancy stuff was not ready for prime-time, monolith and IIS deploy it was. Don’t fall into the trap of equating monolith with bad. The replacement system was built using the latest .Net core C# framework available at the time, TDD and SOLID principles. These architectural decisions will pay huge dividends later.

Within a few sprints we had an MVP consisting of a few core services up and running, at which point we decided to turn it loose on the team which will eventually consume our endpoints. Better to get your product in front of the customer as soon as possible even if its missing limbs. At the end of one of our sprints we deployed to QA, then went home feeling good about ourselves. When they started using it we realized there might be a problem. When the client called our system, they sent a request to one of our newly built web services, which we then sent to our fancy new 3rd party vendor for processing. In this request there is a payload containing insurance agent/agency data. This data consisted of a chunk of JSON which contained all the agent’s information gathered by the upstream system. The connection from the upstream system, to our system, to the vendor was working great, except the vendor kept returning errors. These were not technical errors, thought we had a bit of them in testing. No, these errors were telling us there is something wrong with the data we just sent. Remember back in the beginning when I reversed engineered the old legacy system only to find business logic everywhere. Apparently, there was more business logic, unfortunately this time I had no idea where it was!

For those of you not familiar how the insurance industry works in the United States let me explain it. See the U.S. insurance industry is highly regulated at both the federal and state level. This means if an insurance agent wants to sell insurance they must deal with a lot of federal and state regulations. One example is that they need a specific type of license to sell a particular insurance product in a particular state. “must deal with a lot of federal and state regulations”. In the world of software development this translates into lots and lots of business logic. Each state has its own set of rules governing what and how you can sell insurance products. These rules can vary by multiple factors including:

* State you wish to sell insurance in.
* Type of insurance license you hold.
* Type of insurance product you wish to sell.
* Type of agent you are. Are you an individual agent, or are you an agency?

The above rules created enough permutations that one could not possibly test for all of them in the given time frame.

## Project Issues and my Anxiety

Now that we were seeing many of our requests failing due to not sending the correct data, my team and I started to get concerned. After several meetings with business people we realized our old legacy 3rd party vendor housed many of the business rules we were missing. This was one of the values this vendor brought to the table. The business people who commissioned this project i.e. get rid of this vendor and build it in house had no idea this vendor abstracted away so many of the business rules. With us loosing this vendor we lost a lot of custom coding that went with it. Remember how I was worried that the business logic was everywhere and apparently, we did not have all of it in the legacy application. Now I finally knew where all the business logic went. To make matters worse, the business terminated the contract with the vendor and let the vendor know 90 days from the date of termination. This was a HUGE issue for my team! We now had a hard deadline as we would lose this vendor in 90 days, and this vendor had zero motivation to help us as they just lost our business!

## How to Plan for the Go Live

So now you have some idea of what I was dealing with. For starters we had the following issues.

* *Hard deadline little more than 3 months away.*
* *Old vendor who would not deal with us anymore.*
* *Unknown amount of business rules to implement.*

This is the part of the post where I lay out my plan to get my team through this. With a hard deadline approaching my team and I met to discuss a plan of action.

###### Test as Much as You Can

We knew we could not create every possible combination of business case, thus we decided to focus on testing the scenarios which would have the biggest impact. One thing we knew was how much volume of business we did in each state. For example, we did far more business in California then say Idaho. We also knew what products we sold the most of (Life and Health). Armed with that information we decided to create test scenarios for the states and products which would have the biggest impact.

* *Identify areas of your product which must work day 1.*
* *Create test scenarios which cover areas which have the biggest impact on your business.*
* *Test and address any issues.*
* *Repeat and work your way down the list.*

And that’s what we did.

###### Go Early

One of the things which kept me up at night was the fact that we had to have everything working by the hard deadline. I am not a fan of deploying code to production and hoping, no praying everything works perfectly right out of the gate. In my experience nothing goes as planned. You will have many unforeseen issues. I felt I had 2 options.

1. Test as much as you possibly can and plan on working 24/7 starting from the night of the deployment until all issues are resolved.
2. Go early.

Option 1 is typical in large corporate IT shops. I’ve heard them called “release parties”, but trust me this is one party you don’t want to be invited to. What usually happens in this situation is you have some deploy time, say 9pm. You deploy and start watching for any issues to popup. Issues will pop up, and you will be working nights and weekends for a while until you get the system stable. Depending on how many issues, and how complex the system you might not survive.

Option 2 seems a little crazy on the surface. But Ron if you can’t test the system fully in the given time frame why in the world would you try and deploy early? With this we caught a break as the old system was still operational. Remember all the crazy business rules which can vary by state etc. With a little work we could route traffic to the new endpoint based on the state sent in the client request. If California was our biggest state we can turn them on as soon as we were comfortable with our test results. I cannot understate how big this was for us! We had a way route traffic to the new system one state at a time while the old system was processing all the other states. Going early has the following advantages:

* *Go early if you can process a small subset of traffic.*
* *Going early allows you to iron out any deploy infrastructure issues right away.*
* *Going early allows you to get a feel for how the system will function in production (under a reduce load).*
* *Going early will hopefully expose any architectural and design flaws, thus giving you time to find a resolution without the stress of keeping a production system up and running.*
* *Going early can score you political points with business people and higher ups.*
* *Going early will expose the users to the system earlier, thus when the deadline arrives you users should be ready.*

We decided to deploy 2 months early. Another advantage we had was we used Agile method to develop the product. This had the advantage of rapid iteration. With Agile, we were used to deploying every 2 weeks, and with a little work we were able to deploy multiple times a day.

###### Preparing for the Go Live

Deploying mission critical software has always filled me with dread. No one likes the anxiety that comes with all the pressure to deliver. I cannot imagine the pressures game developers face as a deadlines approach. At some point in my career I came up with what I am calling my *Go Live Routine*. This is how I mentally and physically prepare for the pressure of the go live day, and how I have learned to manage prolonged periods of intense work.

Get Good Sleep

I cannot overstate how important this is. Without sleep you will have little chance of performing at your peak capacity. For the go live day and all subsequent days I went to bed every night at 9pm, rising at 6am. Experience has taught me that when I go to bed after 10:30pm, I am foggy in the morning. Your body is different, do what works for you.

Practice Mindfulness

After awaking at 6am I did 40 minutes of yoga and 20 minutes of quiet meditation. This prepared my mind and more importantly, my emotions for the days chaos to come. I am convinced that I without this I would not have been able to cope with the intense stress.

Take a Break

At some point during the day I put down whatever I was doing and I went for a 30-minute walk. This allowed enough time to flush out thoughts and step back to see the bigger picture. I could tell that after just 30 minutes away my mind was working better.

Eat Like an Athlete

I ate very light foods several times a day to keep my brain and body at peak performance, almost vegetarian. Salads, fruit, vegetables, ect… I have no actual proof that this helped but I do know that a big lunch kills my afternoon productivity. Better safe than sorry.

###### Go Live!!

After testing as much as we could we decided to go live 2 months early. We spent the last month testing as many permutations as we could. We were as ready as we will ever be. If you remember we prioritized testing our highest volume states first, but after discussions with business leaders we felt turning on our biggest state first might be too risky. Yes, we could always quickly revert and push all traffic back to the legacy system, but big IT is a very risk adverse and we decided to turn on a state with much lower volume of traffic. And so, at 9pm we deployed our code, ran some initial tests to make sure everything was working as expected, then went to bed and waited for the business to start using it the next morning.

The next morning, I knew would be intense, as a result I came up with a routine to help manage any issues which would arise. First thing in the morning perform the following:

1. Open corporate messaging. Catalog any issues, questions, or anything related to the product which just went live.
2. Open corporate e-mail. Catalog any issues, questions, or anything related to the product which just went live.
3. Meet with your team and business sponsor and go through the list, triage and prioritize all the issues, questions, and anything else. We now had a prioritized list of issues to work.

With the list in hand, we set about fixing the most critical bugs first. As soon as we had the items fixed we deployed to QA and our testers verified the issue was resolved. Then, we packaged up the fixes and deployed to production. Depending on how much work was involved we could get several issues fixed and deployed in a couple hours. After a deploy, the business sponsors worked with the users to ensure the issues were resolved. After the deploy we met again, re-cataloged the list with anything that came in since doing it in the last time and the cycle repeated itself. One thing I must emphasize, it is easy to get completely overwhelmed by all the issues, questions, ect… coming in from all the business sponsors, users, your manager, and anyone else. It is *critical* to your success and sanity that you manage the intake process and you *must* compartmentalize, and block out all distractions. I made this mistake more than once, and was overwhelmed trying to address everything all at once. For the first few days we were deploying 3-5 times a day. By the end of week 1 we were deploying 2-3 times a day. By week 2 things were getting noticeably better with about 1 deploy a day. After week 2 we were deploying a couple times a week. At the 1 month mark we were back to our regular 2-week sprint/deploy cadence. We still had issues but they were manageable and were part of our regular workflow.

In summary, this was not the largest go live I was apart of, but it was the largest where I was the team lead. Also, it was not a huge enterprise wide initiative, rather it was for one division within the organization, still large for my small team. In my tenure at big-co we had one massive integration project which was in fire mode for 6 months. This project chewed up numerous good people. It was as close to a death march project as I have seen.

###### Lessons Learned

After about 2 months the system was stable and we had time to reflect on everything. Reflecting or in Agile speak “having a retrospective” is a very important step in any sizable initiative. You will make mistakes, and there will be oversights. The purpose of the retrospective is to document them while they are still fresh in your mind or while you are still licking your wounds 😊. For me personally, this project was a great experience, and it validated several concepts which get tossed around a lot in the development community.

SOLID Works

Earlier in this post I mentioned we replaced a horrible monolith with a well architected monolith. One of the key architectural decisions we made when designing the new system was to follow SOLID principles. For some of you reading this you might say “yea no crap” following SOLID is the way to go, but I have encountered more than a few senior engineers who do NOT follow this. What usually happens is the system becomes an un-manageable mess, and eventually our team gets brought in to take over and clean it up. So, what exactly was the benefit of architecting with SOLID principles?

By decomposing all the major components into little single responsibility classes, we were able to isolate and test very specific pieces of the system. For example, in one instance we needed to handle an individual insurance agent different from an insurance agency. We wrapped the functionality in a nice little class with the unit tests. With this, we were able to prove the business rule worked. When you include the interfaces and unit tests, we ended up creating several hundred classes to encapsulate and isolate all the functionality. This might seem like overkill, but in the next paragraph I will explain how it save us on more than one occasion.

Remember during the go live period when we had to triage and fix all the issues coming in. Because the bugs were encapsulated into little classes, identifying and fixing issues became trivial. This was a huge win for us! This allowed us to do the following:

* + Identify the bug with relative ease.
  + Fix the bug.
  + Update the unit tests to prove the fix is working as expected.
  + Re-deploy knowing we were NOT affecting any other part of the system.

When your classes do not follow the single responsibility principle, fixing a bug or adding a feature can be risky, as you are not 100% sure you will not introduce another bug somewhere else in the system. With our code we were very confident we could make rapid changes and not worry about destabilizing another unrelated part of the system.

By favoring composition over inheritance, we were able to add and modify functionality with ease. There were several points thought the project where we would encounter new state specific business rules we needed to add. For example, what type of insurance license was required (resident vs non-resident, active vs non-active) would vary by state. We knew we needed to retrieve the appropriate license so the contract was the same no matter what, but WHAT type of license we needed could vary by state. This was solved by creating specific LicenseRetrievers to support the functionality. We could now add new license retrievers with ease by having a factory instantiate the appropriate object based on the state. This was just one example of how our architecture allowed us to evolve the system quickly and safely.

###### Log Everything

Again, this seems like a no-brainer, but I cannot tell you how many systems I have seen where there is little or no logging. When you have little or non-existent logging, figuring out what the system is doing in production becomes exponentially harder. Our logging goal was simple, GET VISIBILITY INTO A RUNNING PRODUCTION SYSTEM. This broke down to the following:

* Identify the endpoint which is being called.
* Log the request data.
* Log the response data.
* Log any exceptions with full stack traces.
* Add verbose logging when needed.
* Try and log “Why” something broke.
* Try and offer suggestions as to how to fix.

Logging the endpoint and capturing the request/response data allowed us to identify and re-produce many scenarios with relative ease. We could take the request data, write a test and see it behave in a development environment. The following is how we would triage an issue:

* Use the log to identify the endpoint.
* Use the log to identify the request data.
* With the above 2 items, create a Postman request in a lower environment.
* Run the request and hopefully reproduce the issue in a lower environment.

Adding context to logging to try and explain “Why” something broke is something I realized after years of studying logs and trying to figure out “what does this log mean”? For example, let’s say you are troubleshooting an issue and you see a log that reads “Invalid input state”. To me this log is fairly useless. What does that mean exactly? My advice when writing log messages is to “*clearly identify what went wrong, and offer a suggestion as to a possible remedy.*” An improved way to write the previous log can be “The state ‘AA’ is not a valid U.S. state abbreviation. This request requires a valid US state abbreviation”. Notice I included the input data “AA” in the log so the person reading it can clearly identify what was sent. Now the person troubleshooting the issue can take one look at the log and say “Oh yea they sent an invalid state abbreviation” and let the end user or system know what the issue is. With proper logging, issues like this can be resolved in minutes not hours.

###### Build Dashboards and Reports

This is very similar to logging except where logs offer visibility for technical people, dashboards and reports allow visibility for end users. The basic rule is this. Build a dashboard to give your uses and stakeholders visibility into the system they are interacting with. This has several benefits:

* With a dashboard, the end users can proactively solve some issues. For example, if the previous error with the invalid state was displayed on a dashboard along with the request data, the user can clearly see that someone sent a request with an invalid U.S. state abbreviation. A developer may no longer be necessary to identify this issue.
* If some business process has several steps handled by various departments a dashboard can give real-time updates as to which step in the process the work is currently.
* With a dashboard, you can inform the users in real time various status information. If a service is down for maintenance the dashboard can let the end user know, and thus not wonder why something is not working.
* With a dashboard or admin UI you can empower the power users of your system to do some of the maintenance themselves. This has the potential to greatly reduce the amount of “technical maintenance” development teams must do after a product goes to production.
* Users can answer business questions themselves.

The basic goals behind creating dashboards and reports is to empower the user to find the information for themselves. In my experience logging, and dashboards are often overlooked aspects of a software development project. This is because the business does directly see these as beneficial. What seems to occur in my experience is that the development team is then on the hook for “technical manual labor”. This is non-development work related to maintaining the system.