Quarantine Dream

How Software Development Practices are Changing by Using Functionally-Isolated Microservices, DevSecOps, and Event-Driven Architectures

Prepared by MTSI

Tyler Darbyshire
Ben Winks

MODERN TECHNOLOGY SOLUTIONS, INC.

Table of Contents

INTRODUCTION 3

ACCEPTING CHANGE 4

AGILE 4
DEVSECOPS – A BRIEF CONVERSATION 5
CULT OF SECURITY 5
NO SUBSTITUTIONS, PLEASE 5
MAKING CHICKEN SALAD…OUT OF CHICKEN 6

SOFTWARE GUMBO 7

FUN-SIZED SERVICES 8

WHERE THE MICROSERVICES ARE 8
BREAKING IT UP 9
RETURN OF THE PARTY LINE 9
TOWER OF BABEL 9

CONTAINER ORCHESTRATION 11

MANAGING MICROSERVICES IN OPERATIONS 13

PUTTING A BOW ON IT 14

Introduction

The other day I was having a conversation with a coworker when a relic of the past thrust itself into the middle of my morning coffee. We were working through an issue with the way cloud-based virtual machine's report resource usage, and one of the required packages had failed. Upon reading through the logs, it was discovered that the package had to have Python 2.7 as the default interpreter for the system. It got me thinking, *when was the last time I started a new project in Python 2.7*? The longer I thought about it, memories of my then co-worker ripping our office apart because another had converted a project to use Python 3 without updating the code's syntax flooded my mind...and that was from 2016.

What made this so flabbergasting? Merely the required usage of Python 2.7? No, it goes further than that. Seeing almost anything outside of the context of its time has a similar effect. Think about seeing the Pyramids in Egypt – on the surface, they are each just a bunch rocks stacked together - at least to the average mammal, right? They are a physical *wayback* machine, a gaze into our historical past, into a by-gone time. Same thing with a classic car – it sticks out as notable because it is largely out of place...time has moved on, things have changed all around it, yet here it is, knocking about. This pattern holds true for software – imagine your IT Director sending a memo via *fax*, saying "*we've made the decision to go back to Windows XP – it's been around for a long time and we did good work using it. I'll be handing out AOL CDs after lunch as well, they should have about 1000 hours on them! Happy surfing!*" It wouldn't make sense to use old tech, so why do we use old development practices?

> "*You know, when I started, we didn't have the internet – we didn't use search engines to find information online – we rifled through the pages of the closest language manual to find the answer we needed, hoping those pages weren't stuck together from the morning Danish or irrecoverably stained from coffee spilled in the frustration of the moment. Or get this - we'd have punch cards that we'd spread out and drew a line on them so if we dropped them we could put them back in the correct order…things were different back then."*

Imagine not only choosing, but openly advocating for the use of old practices and technologies like that today – that may sound ridiculous to you...it *should* because it *is*. How does one even work without Google today, let alone the rest of the internet? There are so many skills and technologies developers need to know today, let alone general knowledge, compared to just a couple decades ago. It would be nearly impossible – perhaps *literally* impossible - for developers to retain all that *granular* and *precise* knowledge without a developmental *Library of Alexandria* ever at the ready. Given the speed with which new technologies, tools, frameworks, etc., come out, how can you know all the exact syntax and API features for new tech? You *can't*. *Johnny Number 5* would struggle with all that input! Just imagine having a conversation like this today: no mention of containers, eschewing modern languages like Python or Go in favor of something like Fortran? Preposterous! For the vast majority of use-cases that makes no sense! So why would we do this with development practices and ideologies? *We shouldn't*.

We should be using modern, efficient approaches to developing products – this paper is focused on reaching developers and breaking down why these modern approaches, like DevSecOps, Agile, creating microservices, and orchestrating them in production are better than old habits.

Accepting Change

> *How do I know what paradigms to follow? What about my tech stack? Practices? There's so much to choose from - how do I pick, especially when so many people still practice Waterfall? What makes Agile so much better when I've been told by some it isn't?*

This can seem daunting at first, especially given the menu of options on offer – but if you take it piece by piece, one step at a time, *cliché of choice,* things begin to clear up. Our primary job is to get the customer a *valid* and *verified* product that fits what they ***actually need***, not what we think they may want or something that's destined to become *shelfware*. What's the best way to achieve this? Thinking about it from this perspective, a development philosophy, Agile Development, becomes...well, *obvious*. Once we get there, we can then pair that up with another iterative, secure practice – DevSecOps.

Now, before we move onto the brief overview, we need to address the final "question" – there exists a not-insignificant community of developers and management who are firmly steadfast in their already-learned behaviors. *Something something old dog, something something new tricks?* Needing to adapt to modern workflows and new practices is not exactly appealing to those who have been working in the same paradigms for decades. "*Why do we have to change now – we've been doing this for years and it's worked fine.*" That kind of thinking would have us still riding around on horseback, taking Conestoga Wagons across the plains, and writing letters to one another on parchment by the melting wax of the ever-trusty candle.

Agile

In order to get to a place where we are able to make this dream of quarantined functional units, **ahem** microservices, a reality, we require a rock-solid foundational process from which to build. This is where Agile Development comes in. Having touched on Agile Development processes in our first whitepaper, let's not spend too much time here, but it's important to make sure we all have a common understanding before diving into the rest of the conversation.

The Agile workflow makes it easy to group top-level project requirements into smaller, similarly scoped groups that are easy to manage for both developers and management. Requirements can be broken down into functional units that form *Epics*. These longer-form milestones can be used to track the development of, for example, one of the microservices that form the system as a whole. Epics are comprised of various *stories* – these are single-tasks that are broken down in such a way that they can be accomplished during the current Sprint. The development team works stories in parallel. This is one example of how the Agile process flows, not a set-in-stone gospel that must be followed without deviation. To discuss current status and potential issues, project teams meet daily for a scrum, a short meeting that's *supposed* to be less than 15 minutes. This meeting is important to keep everyone, including the security and ops teams, apprised of potential issues needing rectified. Full disclosure - Agile is a *bit* more complex than what has been discussed here. This paper's intention is not to be about Agile specifically – the brief discission brings clarity to how Agile development, DevSecOps, and microservice implementation go hand in hand with each other.

DevSecOps – a brief conversation

Just like this is not a paper with a specific focus on Agile Development, it is equally not specifically about DevSecOps, but rather a discussion of a larger developmental *buzz-word* Venn-diagram. This paper is a conversation tailored to software development in general, but it would not be complete without a little more general process overhead.

DevSecOps (DSO) is as important to the whole as Agile, as is the developing of microservices and organizing them in a structured way – it is just not a primary focus of this particular piece of literature. DSO is a modern solution that pairs excellently with the Agile workflows. By including Security and Operations professionals into the daily scrums, they can be made aware of issues that arise, help with the automated workflows and continuous integration/continuous deployment pipelines that are being used, and, most importantly, they provide necessary insight to the process where developers can often be ignorant of issues until it is too late.

Cult of security

We need to be radical about security. We need to be working in a cult of security, if you will. Everyone in the company, from the interns to the highest levels of management sitting on *mahogany row* need to buy-in – no more just talking about it, *be about it*. Why should every single employee have stake in this culture and participate? Security should no longer be a responsibility pushed to people that have security in their title, or pushed off to the development team, etc. *EVERYONE* should be thinking about security because it benefits all of us in the end. For software, consider security throughout the whole development process: during modeling, planning, software approvals, development, writing tests, reviews, meetings, you get the point.

No substitutions, please

In addition, to say or imply that DSO is simply a pipeline grossly undersells the larger significance – the true benefits emerge through the cultural and procedural shift of baking security into the underlying processes. Yes, pipelines automate a lot of security – tests, scans, secure builds, etc. – but the instilled idea that all processes need to be secure down to the IDE we code in is *incessantly* important. Pairing this with the constant input and assistance from security and ops teammates, you have a sure-fire recipe for success – you just have to execute and not cut corners, lest the gains be lost due to negligence and complacency.

Combining automation with rampant and aggressive use of standardization practices, we really begin to see the benefits of being able to trust the process. This trusted process will lead to trusted products which then in-turn lead to decreased time from the lab to the range and production - a positive feedback loop. By using approved operating systems with approved software libraries and tools, you severely reduce discrepancies between environments. Similarly, by scanning, testing, building, and hardening code committed to a repo in the same, standardized way following each commit, you can ensure that the artifacts produced will be born of the same, approved, trusted process. No more ‘*it works on my machine’* or ‘*how did you build this?’*

Making chicken salad...out of chicken

DSO and Agile are important. The pairing of DSO with Agile Development lets you take advantage of the iterative nature they share. This repetitive, cyclical flow allows for continuous evaluation and evolution of the developing project. Equally important, though, is *what* is being developed, not just *how*. Making sure that we are designing modernized solutions that make use of cutting-edge architectures and deliverables like microservices and Event Driven Architectures (EDA) is crucial moving forward. Microservices can incentivize the creation of genericized solutions that enable the reuse of components – write it once and reuse it. Taking advantage of quarantined, isolated functionality, we can write services that perform AN function, AN *single* function ("an" is for emphasis here, ignore the grammatically incorrect usage, thanks!) rather than making a *Swiss army knife* service that does everything terribly. We can then start stringing microservices together using an EDA which can truly allow the whole to be greater than the sum of its disparate parts.

Software gumbo

Before the time of microservices, monolithic applications were the *meta* – that's just how things were done. The core concept of monolithic services was to throw every required feature into a single packaged service, not unlike a software *gumbo*. This service was only deployable and maintainable by a single person, or at best, a very small team of people that helped develop it and hid this knowledge away, disguised as job security. We've all worked with or heard of a *wizard* who has been tasked with maintaining the craziest spaghetti-code-laden project and only they know how this thing works. The office jokes about how "we can't fire Jeff because he wrote that thing, and I sure as hell won't learn it..." Please, stop it. No more of this.

Developing monolithic applications quickly became a horrid, breathtakingly terrible, error-prone process and was hugely reliant on those people to keep the ship off the ocean floor, let alone afloat. Additionally, when the service failed or crashed from a bug, the entire application was brought to its knees - no resiliency whatsoever. When a user experienced a bug in the service, it took time and multiple people to locate where the bug was introduced. All digging through the long procedural chain of components calling other components, passing pointers, manipulating data structures and databases, clearing and loading values into registers, shifting bits, making jumps, undoubtedly bad recursion, loops, loops, loops, loops, did I mention that components call components, bad enc©ↀing, and segmentation fau----. This mangled, tangled mess of overly complex, often unorganized code leaves us with the ultimate question of where was the bug actually introduced? Great question that begs for an answer, but sadly, it is not be easy to find. On top of all that, to make a bad thing **even** worse, the time it took to get a new release of the service with the bug-fix out to production took quite literally *five*-ever due to the amount of regression testing needed to ensure nothing else broke in the application or procedure chain, let alone the new authority to operate (ATO), etc.

Fun-sized services

Microservices, EDAs, containers, and other modern tools/best practices address most issues with monolithic applications. Development teams can scope related requirements into services that perform similar functionality. This will result in a system composed of multiple services, each responsible for a specific and related set of operations. No two services should provide the same functionality or address the same requirement, unless *explicitly* needed. A single service should not be providing functionality that is out of its requirement scope either. For example, a file parsing service should not be responsible for handling authentication requests – split those up, keep them separate. Monolithic applications tried to do everything in a single service – and that was a disaster.

Scoping requirements into smaller services allows for pieces of functionality and features to be developed faster and in parallel. Each service can be developed and released regardless of whether the entire system is finished or not. This new parallelized development results in customers getting features quicker as functional units can be delivered once completed, if desired. Due to the increased frequency of software deliveries (can be in days or weeks, not months or potentially years) and the use of Agile, customer input and feedback is more readily available. Gone are the days of waiting for the end of a contract to get customer eyes on the product, as well as feedback and requests, and then rushing to fix issues, get them tested, get them approved by security, and then redelivered.

If you eliminate the legacy approach of chaining functions or method calls together to accomplish a workflow, then updating each component or adding to the workflow becomes more straightforward. They are noticeably easier to maintain, organize, version control, and test. No more trying to test, scan, and re-approve the monolithic beast. We can focus on each service individually.

Generalization is another thing to keep in mind while planning and developing these services. Generalizing a service so that it is not “hardcoded” to any specific use-case yields future benefits and can save rework. Of course, not all services can be used for other efforts - contract funding, reuse rights, etc., can muddy the waters, but when possible, do it. The generalized services can be reused across other projects, which breaks down the old, siloed way of doing development. Why constantly recreate the wheel every time we need a wheel? Or more on subject – why create a new authentication service for every new system? Make one, reuse it, and continually update it, making use of your organization’s developers. Better yet, is there an opensource option that doesn’t require implementation at all? Food for thought :)

Where the Microservices Are

When a feature is complete and ready to deploy, there is not a need to depend on a single person that knows the secret incantation, the rain-dance for how to deploy it. Use containerization and orchestration technologies that are standardized, which allows us to deploy our single feature to production via automated workflows in a continuous deployment pipeline. This makes it easy to continually redeploy with a consistent result because of the removal of rampant human error and inconsistency by making liberal use of automation.

With microservices, if, and likely *when,* a bug occurs, it is either because the service received bad or malformed data or there’s a problem in the service itself. Fortunately, the services are well-scoped so locating the bug is infinitely easier and usually takes a short amount of time.

Finding bugs in microservices can usually be done by a single person. Once fixed, the change is committed and goes through the same automated DSO pipeline process mentioned earlier. When it passes the pipeline it is deployed in place of the old, buggy version. Additionally, when bugs or failures happen in a containerized microservice, failure does not befall the system as a whole – the other microservices stay up and continue to be functional while the orchestrator restores the failed service to a healthy state. This provides far better resiliency than ever before, and it comes for free with the orchestration platform.

Breaking it up

Taking a step back, developers may question what the significant difference is between having a bunch of microservices issuing requests to each other and a monolithic application having a procedural chain of functions calling other functions. That is where the beauty of event driven architectures (EDA) comes into play. From a monolithic standpoint, functions must import other functions to call a function that initializes a new class and passes it to another function that returns a value to the function before that calls a different function with the returned value that calls the same function before again - wait...now I'm lost and smell Cajun food. Messy right? A chain of logic with so many dependent functions that are explicitly called can be disastrous, particularly when things go inevitably belly-up.

If you break some of these functions into their own services and eliminate the need to import and directly call them within a single, long-lived process, developers start to realize managing code and troubleshooting becomes simpler. But wait, if the services can't call each other directly, how do they interact? Enter message buses.

Return of the party line

Message bus technologies are a fundamental part to microservice interfacing and form the backbone of EDAs. The message bus is the glue that binds the quarantined functionality and provides a medium for their emergent properties to manifest. Services do not need to call each other directly, oftentimes they aren't even aware of the other service's existence. The input to nearly every single service should be an event that goes across a message bus that everything is connected to. The microservices will sit and listen for things of interest, not unlike an old party line. When a topic of interest arrives, the service is triggered, the message is consumed, data processed, actions taken, and a new message is output for someone else to handle. If the message topic isn't of interest, just ignore it and keep listening.

This streamlines service integration, enables parallelism, and increases efficiency. Additionally, migrating two completely separate microservice systems together is as simple as sharing the bus and the events on it, or creating custom adapters or shims for incoming and outgoing message types between the two systems. No massive refactor needed. Compare this with breaking down the complex procedural chain discussed earlier in order to shoe-horn the new functionality into its correct place.

Tower of Babel

An underappreciated boon brought about by using microservices is the ability to use the best language for the job. If you need a GUI that gets fed processed data, you can create a JavaScript WebApp, you can process the data using Python, C++, Go, Kotlin – it doesn't matter

since they are completely separate. No longer are you forced to write a Swing UI because you need some Java tooling on the backend. You can even have multiple services using various flavors of C++ if need be. It does not matter *because they are separate.* Use whatever language works best to convey the information or process the data. We are now able to integrate many more languages, frameworks, and technologies to best solve the problem at hand, not inefficiently *kinda* solve it by being handcuffed to using a single language.

Container orchestration

Time to take more handcuffs off – what if there is no need to care about the libraries and packages installed on the host system. Why not throw out the operating system with the bathwater too – no need to care about the host, but let's pray it is not Windows... Each service can have its own environmental dependencies, configurations, and even a *specific* operating system. Sounds like this would be a pain in the *tuchus* to maintain, especially for a bunch of services in operations. Thankfully we have containers to do just that. Containerization is a technology, briefly explained, that allows for building software into an isolated environment with its own operating system, file system, dependencies, and configurations. By committing to our secure practices from earlier, we can use images, from Platform One's Iron Bank for example, that are secured and hardened against known vulnerabilities. These form a great base layer that we can start adding a service into. By running these containers through the same pipelines with automated scanning and reporting jobs mentioned earlier, reports show that everything is still secure and ready for deployment. This helps make managing operational environments for services no sweat at all.

Now that we are not sweating and have containers deployed, how do you manage a bunch of service containers in an operational environment? Glad you asked! Container orchestrators, like Kubernetes, carry the heavy burden of networking, resource management, resiliency, data persistence and management, scalability, and so much more. There is a lot going on there, but thankfully, each service just needs a small configuration manifest that tells the orchestrator the desired state of the service and it accomplishes that for you. These configurations can define things like, what port does your service need to be mapped to, what domain name does it need to be available on, what file paths need to be persisted, how many replicas of the service are needed, etc. Container orchestrators also enable rolling updates, redundancy, and automatic container restarting to achieve the best resiliency possible.

What a lot of people fail to do with Kubernetes is think about security. Just because your services are deployed in Kubernetes does not mean they are secure. Other measures are needed to ensure that services are secure and communicate with trusted, known endpoints. Implement a zero-trust architecture everywhere in the DSO process, *especially* in clusters. No need to *trust but verify* –just do not trust at all. For communication to commence, services and endpoints are verified and then once verified, they're whitelisted. Sidecar containers in a service *pod* enable this. The word *container* can replace a *pod* in Kubernetes vernacular, but pods can consist of multiple containers if needed. The sidecar containers effectively act as an intermediary for all traffic, inbound and outbound. It determines those allowed to be communicated with and being that it lives in the same pod as the service, the two work in concert to enable this zero-trust architecture. Other benefits provided by sidecar containers are continuous security scanning, monitoring, alerting, logging, service-mesh integration, and network protection. Services should only be able to connect to each other if they are explicitly whitelisted to communicate in their deployment manifests. Service-meshes should be leveraged to assist with the management of sidecars and the services they are integrated with.

What's a service-mesh? Again – another great question. It is essentially a framework of technologies that are enabled by sidecar containers. They provide benefits like insight into connections and events between services. This helps identify where bugs are introduced in a processing workflow between microservices. Service-meshes isolate systems to better implement the zero-trust architecture, and when two systems integrate, service-meshes allow for minimal attack surfaces being exposed. How? Only the necessary connection(s) between

the two systems are opened and available for use. Service-meshes ease the challenge of managing dozens of microservices in operations by streamlining insight into the mesh. Information like how services are performing, which services are communicating, etc., is readily available via this constant, live monitoring. They also provide centralized logging, enhanced security with pod-to-pod encryption, they enable a zero-trust architecture, ease network management...need I go on? They are useful, highly beneficial, and we should be taking advantage of them.

Managing microservices in operations

At first thought, you would think managing dozens of services in operations would be more convoluted and much more difficult than a single monolithic application. Fortunately, that is not the case - there are tools and practices that have been developed to help with this. The service-meshes, sidecar tools, Kubernetes, and other orchestration and monitoring tools provide a view into nearly all levels of the production/operations environment. Hardware level, service level, data storage, networking, etc. – all of these layers are actively monitored, reporting and logging is available, and up-to-date status is always at hand. Operations professionals and site reliability engineers (SRE) use these management tools that are deployed in the operational environment to understand the ever-evolving state of the deployed services and the platform. If logs need to be analyzed by developers because there is a bug, there are tools that can be used to gather all logs from the sidecar services into a centralized location - this makes scouring through the logs so much easier as they are searchable. If a failure occurs, the sidecar service can send a message to an alerting tool that can notify SREs to make them immediately aware of the failure. If an SRE is curious about how much hardware resources a service has been using over time, these types of metrics can be gathered by a sidecar service and collected by a tool that can perform the needed analyses and display the metrics on a dashboard to aide in making a decision to scale the service up or down if it needs more or less resources, respectively, for example.

Matter of fact, insight into operational applications has never been this good – the breadth and depth of information at all levels of the production environment is astounding. Similar levels of monitoring and visibility in monolithic applications and legacy operational strategies are either outdated at best or nonexistent at worst. As technology becomes more robust and advanced, so do the support tools. Although more services get deployed into operations, the same level of insight is provided to understand how those services are performing, their heath, interactions, their logs, resource consumption, and more.

Putting a bow on it

The wonders of current technology help to build amazing solutions for current and future problems. When used correctly, technology and modern workflows contribute to the creation of truly astounding things. As time moves on, so should we. Kick the old ways to the curb. Ride the wave of modernization that is flowing through the DoD. Focus on security. Securing tools, practices, services, deployments, containers, etc., is the way forward and it is something that will be expected – it will be a deal-breaker if not done rather than a feather in the cap if it is.

Creating a slew of quarantined functional units – microservices – and linking them up by "seeing" and "sending" digital smoke signals can be a dream for development teams. Rather than having to build a massive monolith, specialized teams can be brought on to create specialized services. This opens the pool to more detailed expertise, deeper skillsets, and a more varied solution pool. This fosters rapid innovation and creative solutions that will help keep us safe and secure.