Building Lean Continuous Integration and Delivery Pipelines by Applying DevOps Principles: A Case Study at Varidesk

Vidroha Debroy
Varidesk Inc.
Coppell, Texas
USA
vidroha.debroy@varidesk.com

Senecca Miller
Varidesk Inc.
Coppell, Texas
USA
senecca.miller@varidesk.com

Lance Brimble
Varidesk Inc.
Coppell, Texas
USA
lance.brimble@varidesk.com

ABSTRACT
Continuous Integration (CI) and Continuous Delivery (CD) are widely considered to be best practices in software development. Studies have shown however, that adopting these practices can be challenging and there are many barriers that engineers may face, such as – overly long build times, lack of support for desired workflows, issues with configuration, etc. At Varidesk, we recently began shifting our primary web application (from a monolithic) to a micro-services-based architecture and also adapted our software development practices to aim for more effective CI/CD. In doing so, we also ran into some of the same afore-mentioned barriers. In this paper we focus on two specific challenges that we faced – long wait times for builds/releases to be queued and completed, and the lack of support for tooling, especially from a cross-cloud perspective. We then present the solutions that we came up with, which involved re-thinking DevOps as it applied to us, and re-building our own CI/CD pipelines based on DevOps-supporting approaches such as containerization, infrastructure-as-code, and orchestration. Our re-designed pipelines have led us to see speed increases, in terms of total build/release time, in the range of ***330x-1110x*** and have enabled us to seamlessly move from a single-cloud to a multi-cloud environment, with no architectural changes to any apps.

CCS CONCEPTS
• **Software and its engineering → Agile software development**; Software configuration management and version control systems

KEYWORDS
Continuous Integration; Continuous Deployment; Continuous Delivery; DevOps; Software Build; Software Release

ACM Reference format:

Vidroha Debroy, Senecca Miller, and Lance Brimble. 2018. Building Lean Continuous Integration and Delivery Pipelines by Applying DevOps Principles: A Case Study at Varidesk. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (*ESEC/FSE '18*), November 4–9, 2018, Lake Buena Vista, FL, USA. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275528

1 INTRODUCTION
Continuous Integration (CI) – which automates software build tasks such as compilation and testing – is rapidly increasing in popularity. When combined with Continuous Delivery (CD[1]) – which can automate the deployment of software – the overall time between when code is completed, and made available to end users/customers, is greatly reduced. Reports show that CI/CD can help companies, for example: Flickr to deploy software more than 10 times per day [[1]], and Facebook to increase the frequency of mobile deployments [[10]]. There are also additional benefits such as shorter feedback loops, and in the case of HP's LaserJet Firmware division – "development costs per program went down 78%" [[5]]. Having CI/CD pipelines is thus, considered to be a best practice and in fact, an essential practice from the perspective of DevOps [[3]]. The State of DevOps report also finds these practices to be consistently indicative of high IT performance [[8]].

Varidesk offers a multitude of active workspace solutions for home and office spaces, and our customers include organizations such as Microsoft, Verizon, State Farm, and Walmart, to name just a few [[11]]. Varidesk has a multi-national presence and offers businesses and general consumers localized products, and several purchase options, via our Website. The current complete web application is built on top of a Commercial Off-The-Shelf (COTS) Content Management System (CMS), which also handles many responsibilities pertaining to eCommerce, and communication with our Enterprise Resource Planning (ERP) system; and therefore, our current web application is somewhat monolithic in terms of its design. Recognizing the value of CI/CD, we have had a pipeline for this web application in place right from the start, but given the application's monolithic nature, the pipeline itself is also simple – because even though we have multiple environments such as *development*, *staging* and *production*, in the end it has always been just one app, and this has made builds/releases and the supporting infrastructure easy to change, update and maintain.

We recently decided to move towards a micro-services-based architecture; decomposing our web application into many smaller, and more focused services. This however, meant that each service should be build-able and deliverable independent of

[1] CD is often also used to denote Continuous Deployment. We direct readers interested in learning more about the *subtle distinction between Continuous Delivery and Continuous Deployment* to the post at [[2]].

the other, which meant individual integration and delivery pipelines, while also retaining the ability to build and release everything *en masse*. Consequently, the number of build and release definitions would increase significantly, and the infrastructure that was utilized may no longer be sufficient. At the same time, we also decided to have our new system on a multi-cloud environment (which wasn't the case before), meaning that our pipeline(s) had to support this, and not make any cloud-specific assumptions. This created a number of challenges – as we were not able to build and release speedily, and at scale, across multiple clouds (further discussed in Section 2) using the existing infrastructure and tooling we had in place.

A literature survey also revealed that CI itself (and CD too) has received little attention from the research community [[6]], and that despite the general adoption of these practices, many unanswered questions still exist along with barriers that software engineers have to face when using CI and associated tools [[5]]. In fact, a number of the barriers, discovered by studies such as [[5]] via survey, are precisely the ones that we faced. Ultimately, we were able to get past these barriers and build out lightweight and flexible CI/CD pipelines by the application of DevOps principles to the pipelines themselves. More concretely, we leveraged ideas such as containerization and orchestration to create the infrastructure that supports these pipelines. This was novel from our perspective because our initial goals had only been to apply these ideas to manage the apps themselves at runtime. Given the successes that we have seen that we directly owe to this strategy, we wanted to share our findings with the larger community, with the hope that they will be beneficial to academia and industry alike. Specifically, the contributions made by this paper are:

- We independently confirm the findings of last year's research study [[5]] – while we were not part of the authors' original survey, we acknowledge that we faced many of the same barriers identified in the study, when using CI.
- We focus on 2 barriers: long wait times for builds/releases to be queued/completed, and the lack of support for tooling[2]; and share solutions that we applied, providing as much transparency and detail as we can.
- We issue a call for a closer collaboration between industry and academia on researching CI and CD and increasing the knowledge-base. As pointed out in [[6]], with no data and many unanswered questions – decisions are often made based on folklore. By sharing our experiences, we want to do our part to address this.

The remainder of this paper is organized as follows. Section 2 presents background information followed by Section 3 which provides technical details on the changes we made to our CI/CD pipelines per DevOps principles. Section 4 details our results and observations, followed by our conclusions[3] in Section 5.

[2] The study in [[5]] identified several barriers, some distinct but related. For example: 'lack of support for the desired workflow' and 'lack of tool integration'. We generalize these together as (lack of tool support) it allows for brevity, without sacrificing content or clarity for our paper.

[3] As shown in studies such as [[5]], [[6]] and [[10]] (all within the last 2 years), there has been very little work done in this area. Furthermore, there has been almost no industry-academia collaboration, and thus, we have no dedicated section on Related Work.

2 BACKGROUND

Herein we detail our existing CI setup and infrastructure (that worked for our monolithic app); clarify the goals with respect to the new micro-services; and discuss the reasons why the existing setup would be problematic toward achieving our goals.

2.1 Existing/Prior Setup

Build/Release Definitions: Visual Studio Team Services (VSTS), an offering by Microsoft, has been our one-stop-shop for source/version control, task planning and bug tracking, as well as the repository for our build and release definitions. Each definition consists of one or more *tasks* (each task in turn corresponding to a step in the CI or CD process) which are made available in VSTS when creating/editing the definitions. The intent of the tasks is to encapsulate the description for each desired step in terms of not just the action to be taken; but also setting up pre-requisites; making sure that a minimum version of dependencies was available and such. A reference list of common Build/Release tasks available for use with VSTS is available at: https://docs.microsoft.com/en-us/vsts/pipelines/tasks/?view=vsts. Tasks that an organization develops on its own can also be uploaded to the VSTS marketplace to share with others or make available to just the organization's VSTS account.

For our monolithic web app, build/release definitions using only standard VSTS tasks had been sufficient.

Build/Release Infrastructure: Build steps (compilation, testing, etc.) and release steps (deployment, app restart, etc.) need to take place on some infrastructure (be it a physical machine or a virtual machine somewhere) and Microsoft abstracts this away to the concept of an agent – "*installable software that runs one build or deployment job at a time*". Microsoft then offers the ability to use their hosted agents – which is where they provide the machine-power and take care of maintenance and upgrades for a certain price point; or the ability to self-host agents where organizations can install the agent software on their own Linux, macOS or Windows machines. From a pricing perspective, running on one's own machines was most appropriate for us, and thus, we opted to use self-hosted agents instead of the Microsoft-hosted agent. Note, that each VSTS account does come with the ability to build and release jobs for free for up to 4 hours per month, and so we did leverage this, but only to the extent that it was free.

Organizations such as ours that use VSTS can therefore, support their CI/CD by using agents, and since each agent runs one job at a time – the more agents there are, the more builds/releases can be run simultaneously. Each agent can be grouped into a *pool* and whenever builds/releases are queued, an available agent from the pool can pick it up to process. Microsoft does support installing multiple agents on the same machine, but they state that there are cases where much efficiency is not gained; and that users may run into problems if concurrent build processes are using common dependencies such as NPM packages – for example, one build might update a dependency while another build is in the middle of using it, which could cause unreliable

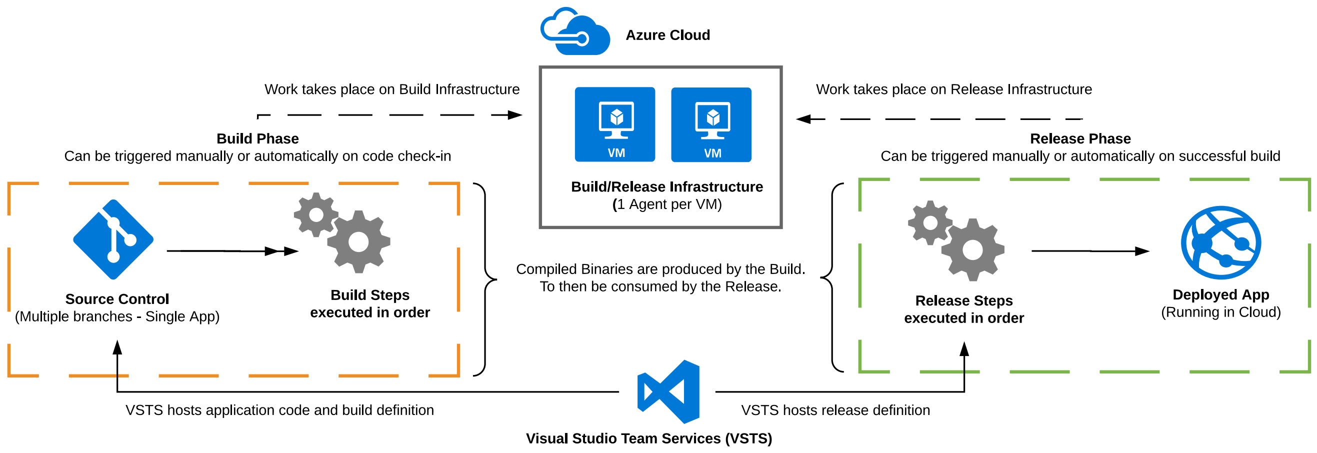

Azure Cloud
Work takes place on Build Infrastructure
Work takes place on Release Infrastructure
VM
VM
Build/Release Infrastructure
(1 Agent per VM)
Build Phase
Can be triggered manually or automatically on code check-in
Release Phase
Can be triggered manually or automatically on successful build
Source Control
(Multiple branches - Single App)
Build Steps
executed in order
Compiled Binaries are produced by the Build.
To then be consumed by the Release.
Release Steps
executed in order
Deployed App
(Running in Cloud)
VSTS hosts application code and build definition
VSTS hosts release definition
Visual Studio Team Services (VSTS)

Figure 1 The existing CI/CD pipeline and infrastructure used to support the monolithic web app

results and errors [[8]] . This applies to us as our current web app uses (and new micro-services will use) NPM packages. From an infrastructure perspective, this equated to one agent per machine for us and the setup we chose to go with was to install agents on Virtual Machines (VMs) that were hosted in the cloud (in our case Microsoft Azure[4]). This would mean we could manage our VMs easily via the Azure User Interface (UI) and could also disable/enable them at will, without having to physically maintain any infrastructure. We were also able to leverage other Azure features such as virtual networks and policies to allow/limit access to/from these machines. In the interests of transparency, we reveal that we used 2 VMs with one agent each, and each VM was sized at “Standard A2” with 2 vpus and 3.5 GB of memory, running Windows Server (2012).

For our monolithic web app, a build infrastructure of 2 VMs, sized at A2 (running an agent each), had been sufficient.

Figure 1 visualizes our setup and shows the various processes. We show a simplified flow with the understanding that this varies from environment to environment (for example, there are more quality gates in upper environments as we get closer to production). We also maintain focus on the apps, pointing out that other aspects also come into play (for example, creating load balancing rules, virtual networks, etc.). In our case (and we assume it would be for any organization), it was cost-effective to use the same infrastructure to execute both builds and releases; though we point out that this was not a necessity.

2.2 Desired Goals

Herein we outline the desired goals that accompanied those we had for our updated micro-service-based system. The reasons why we decided to switch to micro-services are beyond the scope of this paper – though it is worth mentioning that our old monolithic system was getting harder to extend, and we were attracted by the micro-service architecture in terms of the ability to scale and deploy units of the architecture independently. When making such a big architectural change however, there is a corresponding desire to also update/augment in other areas simultaneously, and we focus on those areas that are pertinent to CI/CD and app run-time. Specifically, we wanted to:

[4] https://azure.microsoft.com/

1. Build, scale and deploy each service independently.
2. Define the above in terms of code; i.e., having the services define what their run-time requirements were in a prescriptive fashion (addressing dependencies along the way).
3. Adopt DevOps principles such as using containers and orchestration to maintain fluidity of scale, automatic app re-launch in case of a crash, injection of secret values on app launch via environment variables, and so on. For brevity, we do not explain these concepts herein, assuming that the reader is familiar with them. In the interests of transparency however, we share that our goal was to have every build produce a *Docker*[5] Image which would then run as a container. The various containers would be managed using *Kubernetes*[6], which is an open-source container-orchestration system.
4. Allow our apps to run cross-cloud in the interests of reliability and cost-effectiveness. For example, while we strived for cross-region support within Cloud providers, we also needed to be prepared for outages across an entire Cloud provider. While we had only been running in Azure earlier, we now also wanted to be running in Google Cloud[7], with Amazon Web Services (AWS)[8] under possible future consideration.

2.3 Problems Faced

Long Wait Times for Builds/Releases: At least one of the problems becomes evident when we directly apply the existing CI/CD setup as-is to support the desired goals, as shown in Figure 2. Since every micro-service has its own build and release definition – and per the desired app decomposition, each service can be worked on by different developers completely independent of one another – the number of builds and releases increases greatly, and the build infrastructure (the 2 VMs highlighted in the red box) now becomes a bottle-neck.

If both agents were busy with a current build or release, then any more requests would just get queued, and the queue itself

[5] https://www.docker.com/what-docker

[6] https://kubernetes.io/

[7] https://cloud.google.com/

[8] https://aws.amazon.com/

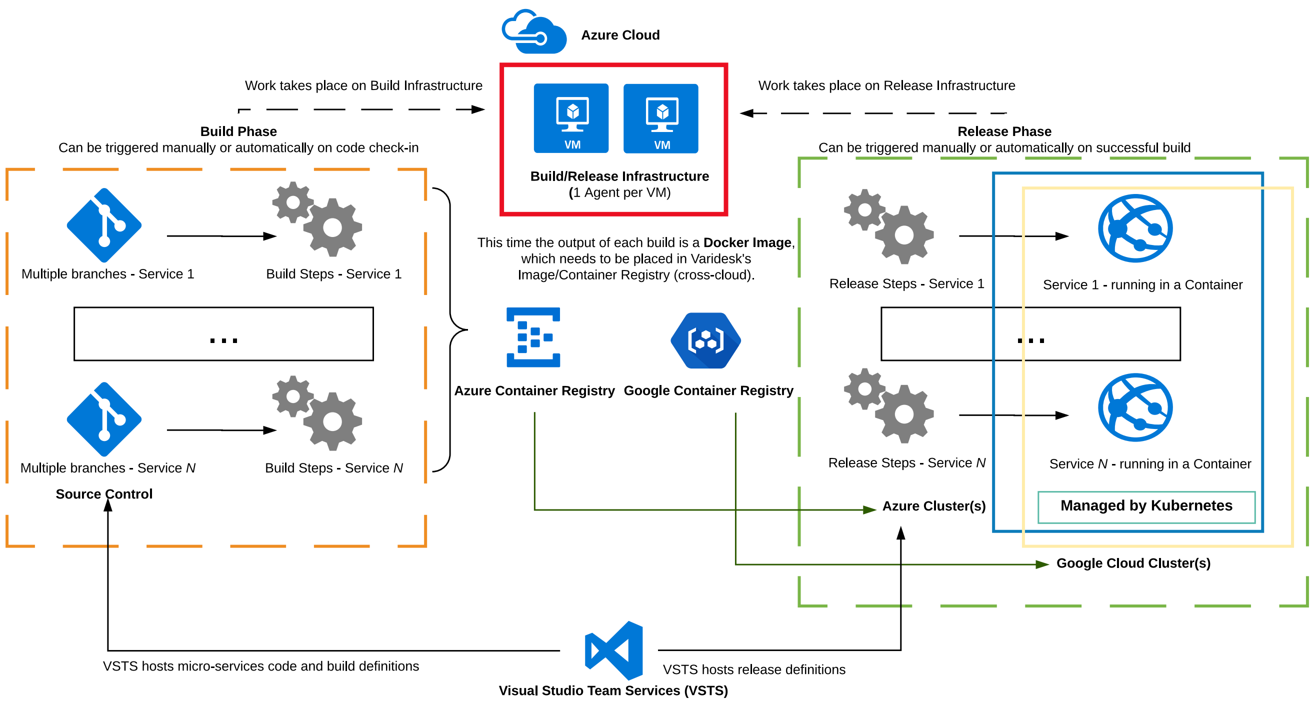

Azure Cloud
Work takes place on Build Infrastructure
Work takes place on Release Infrastructure
VM
VM
Build/Release Infrastructure
(1 Agent per VM)
Build Phase
Can be triggered manually or automatically on code check-in
Release Phase
Can be triggered manually or automatically on successful build
Multiple branches - Service 1
Build Steps - Service 1
...
Multiple branches - Service N
Build Steps - Service N
Source Control
This time the output of each build is a Docker Image, which needs to be placed in Varidesk's Image/Container Registry (cross-cloud).
Azure Container Registry
Google Container Registry
Release Steps - Service 1
Service 1 - running in a Container
...
Release Steps - Service N
Service N - running in a Container
Azure Cluster(s)
Managed by Kubernetes
Google Cloud Cluster(s)
VSTS hosts micro-services code and build definitions
VSTS hosts release definitions
Visual Studio Team Services (VSTS)

Figure 2 Applying the existing CI/CD setup directly to support the micro-service-based architecture

would start growing. Even though the Microsoft hosted agent could be leveraged (as 4 hours of time is available free of charge every month), it did very little to alleviate how much time was spent just waiting for builds and releases to start and then finish. It was especially frustrating for the Team all around, when builds would be scheduled but, would end up failing because of a Unit test failure or because a file had not been checked in, as such a build was not just blocking others till it had finished, but also would need to be re-built itself once the error had been fixed. We note that we had accounted for the potential need to scale our build/release infrastructure both vertically (adding more resources such as vcpus or RAM to each agent), and horizontally (adding more agents themselves), during our initial process planning. But given what we were now seeing in terms of the build/release frequency, should we scale to keep up, some basic calculations already put us an order of magnitude over the budget we had set for the build/release infrastructure.

It was not uncommon to see builds and releases queued for more than a couple of hours. And scaling to try and keep up was going to put us way beyond our budget.

It was not plausible for us to spend that much additional money just on CI/CD when all we were doing was trying to re-architect our web app. At the same time the problem was severe in terms of wall-clock time lost just waiting for a build/release. We even tried to maximize our time on the hosted agent – in one special case our Development Manager, at the time, reported waiting up to 13 hours for a build to be queued on the hosted agent!

Lack of Tool Support: When we began architecting our new solution and developing the CI/CD pipelines for it, we did some basic *proofs-of-concept* – including building out container images instead of just compiled binaries and pushing them to a container registry. Unfortunately, because we had not acquired the license for Google Cloud yet, we did our initial assessment completely based off the integration between VSTS and Azure. As discussed in Section 2.1, tasks encapsulate steps in the build and release definitions and there was very conveniently a task already available for use as shown in Figure 3. Unfortunately, it only became evident later that this task was *only meant to support Azure or Docker Container Registries*. The task itself was non-configurable with alternatives which meant we had a problem pushing to Google Container Registry (i.e., we could not directly achieve the flow shown in Figure 2).

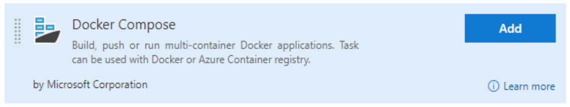

Docker Compose
Build, push or run multi-container Docker applications. Task can be used with Docker or Azure Container registry.
by Microsoft Corporation

Figure 3 Adding a docker compose task in a build

While searching for other options, we discovered work being done by Google, to develop a VSTS Extension for deploying to Google Cloud – but it is not available on the VSTS Marketplace as yet. In fact, the current official release is only at version 0.0.9 which suggests that it is far from being available for general consumption. A Github repository[9] exists for the project that is available; which means we could contribute some code ourselves, but that would also take time and effort that we had not budgeted for. Thus, we fully concur with claim that lack of tool support presents a barrier to the adoption of CI/CD [[5]].

At this point, it looked like due to lack of tool support – we were either going to have to give up Google Cloud as an option or create entirely new CI/CD pipelines to support it.

[9] https://github.com/GoogleCloudPlatform/google-cloud-tfs

3 DEVOPS TO THE RESCUE

It has been acknowledged that there is no uniformly agreed upon definition of *DevOps* [[3]] and that it can mean different (albeit related) ideas in different contexts. There are some common principles however that seem to generally apply – for example, embracing ideas such as containerization at a technical level [[4]], and working toward cross-collaboration among teams from a process perspective [[3]]. It had always been our intent to align ourselves with the DevOps rhythm while we were building out the micro-services; but the application of these very same principles also to the build/release pipelines was very beneficial to us.

3.1 Agents as Orchestrated Containers

Recall from the discussion in Section 2.1 that while we could concurrently perform multiple builds on the same VM by installing multiple agents on it – it could cause unreliable results and errors [[8]]. This has to do with the fact that each agent is really just an app, that when running on the same kernel (lowest level of the Operating System) along with another agent, has the ability to interfere with it. But containerization solves precisely this problem! What if instead of just installing agents on VMs, we had our agents run in containers on a virtualized operating system? The standard setup is contrasted with this approach in Figure 4 and it completely isolates one agent from another.

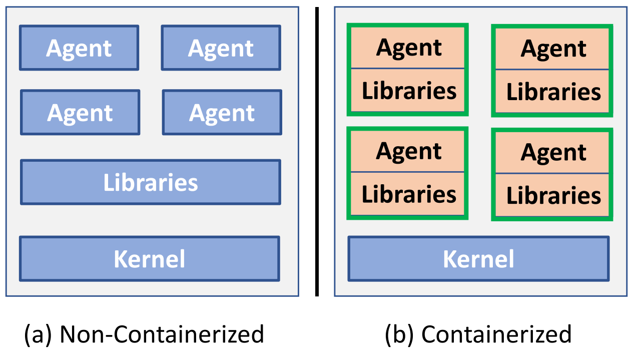


Figure 4. Non-Containerized vs containerized agents

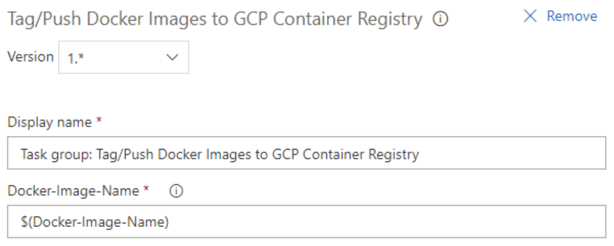
Continuing to think along the lines of these DevOps principles allowed us to realize additional benefits as well. We created our own custom Docker Image hosted in our own container registry that extended the standard Microsoft Agent Image with other useful tools (for example, both the Azure CLI tools and the Google Cloud CLI tools). Furthermore, any time we wanted to update the image with more tools, we would just update our own custom image and apply it to simply create new containers. This is very different from the traditional setup where if new tools or dependencies needed to be brought into the VMs, then they would have to be done on a VM-by-VM basis. Additionally, if Microsoft ever updated the Agent software then the currently running Agents would have to be un-installed one by one and then re-installed; whereas in our containerized case, we would just update the common Docker image once.

Finally, we used Kubernetes to manage our running agent-containers and thought of our 2 build VMs simply as a 2-node cluster. Kubernetes allowed us to automatically bring up a new container should one of the existing containers fail. It also allowed us to secret-ize sensitive data such as access tokens used to communicate with VSTS and the Azure and Google cloud container registries, such that values were injected at container initialization and never stored in the base image, in keeping with security best practices.

3.2 Breaking Down a VSTS Task

Addressing the problem of pushing images to Google Cloud's Container Registry was truly a cross-collaborative effort between Teams at Varidesk, another embodiment of DevOps Principles. At the time, due to the limitations of the Docker Compose task offered on VSTS (discussed in Section 2.3), we either had to develop a separate pipeline independent of VSTS, which would lead to redundancy; or try to add code to the task that had been started on by Google, which was far from ready; or in the worst case give up on the idea of pushing to Google Cloud, maybe switching to AWS. Based on brain-storming sessions between the Development, Cloud and Automation teams at Varidesk, considering the various alternatives, we decided to focus most on the VSTS Docker Compose task. The underlying idea here was that – this task already had the ability to push to Docker and Azure container registries; fundamentally pushing to any container registry should follow the same steps and if we could mimic those steps and then adjust to support Google Cloud, then we could create the missing logic ourselves.

And an interesting decision was made at this time – to not mimic the VSTS Docker Compose task as another custom VSTS task on our side, but rather to bring all the logic into a script that could be executed on the build/release agents. This directly plugged into the idea of using our own custom Docker Image for the container-ized version of the agents, because by using this approach with Kubernetes, we could mount volumes with the necessary script files to load them into the container file system at runtime; making our VSTS task to push to Google Cloud a very simple one – just *invoke the script passing in the necessary parameters.* Using the script which mimics the operations performed in the Docker Compose task, we could now easily push the image to Google Cloud and there were additional benefits described in Section 4. Our custom script was then exposed as a task in our own Build definitions and only needed one input – the name of the base Docker image (passed in via a parameter named 'Docker-Image-Name' at build time) as shown in Figure 5. The same task is used across all our service builds, just passing in a different image name for every build definition.



Tag/Push Docker Images to GCP Container Registry

Version 1.*

Display name *
Task group: Tag/Push Docker Images to GCP Container Registry

Docker-Image-Name *
$(Docker-Image-Name)

Figure 5 Custom task (as a script) to push to Google Cloud

4 RESULTS AND OBSERVATIONS

The ability to push to Google Cloud via VSTS is binary in nature and thus, we can simply say that we were able to do so with our changes to the pipeline, whereas we were unable to do so without it. We explored an alternative where we could listen in on check-ins made to source code by registering for Webhooks, but there was considerable lag and duplication of logic, and so re-thinking the VSTS task as basic scripting was very beneficial. It is also noteworthy that by decomposing the VSTS Docker task to a script – we de-coupled ourselves from a dependency on VSTS tasks. The same script that we used to push our images to Google Container Registry, can in fact be used to push to the Azure Container Registry (the only part that would change would be the authentication information, which is injected into the containers), and thus, we don't need to rely on the VSTS Docker task at all. Using this approach, we can also expand to AWS irrespective of whether they offer support for VSTS tasks or not. Interestingly enough, this current setup ports over nicely to other build tools such as Jenkins[10], which means that in a way – we are able to de-couple ourselves from VSTS (for the purposes of CI/CD) entirely. This makes our current CI/CD pipeline very robust.

To assess whether containerized-agents were helpful or not, we looked at data from 2220+ builds and 350+ releases of historical data (decomposition into micro-services had translated into 42 different build and release definitions at the time of writing of this paper). A reason for the difference between the number of builds and releases is that not every build completes successfully which prevents it from making to the release phase; and to fix the issue results in at least one other build (typically more). Furthermore, check-ins into branches always results in builds, but typically only incorporation into the master-branch/trunk results in a release.

Table 1 Comparing Queue Times for Builds/Releases

	Agents		
	Hosted	Basic	Containerized
Avg. Queue Time	1hr, 17 min	23.2 min	4.2 seconds

Table 1 presents the average time spent by a build or release in the queue, i.e., just waiting to be processed – where the 'Hosted' column represents using the VSTS Hosted agent; the 'Basic' column represents using non-containerized agents; and the 'Containerized' column represents our approach – with the same infrastructure capacity as 'Basic' (described in Section 2.1), with 4 containerized agents in all, managed by Kubernetes (on just 2 VMs). We cannot speak to the size/capacity of the hosted agent as that information is not provided by VSTS. The time spent in the queue for the Basic approach is about 330 times that of the Containerized approach, and similarly the queue time using the Hosted agent is 1,110 times that of the Containerized approach, which translates to significant time saved. Since all of the infrastructure is managed without any new cost incurred, yet the throughput is high, our CI/CD pipeline is very lean.

CI/CD pipelines are an essential DevOps Practice. But by applying DevOps principles to the construction of the CI/CD pipeline itself, we were able to make it very lean and robust.

5 CONCLUSIONS

Continuous Integration (CI) and Continuous Delivery (CD) are widely considered to be best practices. Yet little research has been done in this area and very recent studies have shown that there are gaps in the knowledge-base, and barriers that prevent adoption of CI/CD at organizations. We ran into some of these barriers – such as long build/release wait times and lack of tool support – at Varidesk while migrating from a monolithic web app to a more micro-services-based architecture and trying to go cross-cloud. Thus, we ratify the results from earlier studies in that such barriers do indeed exist. We were able to get around these barriers by applying DevOps principles such as containerization, orchestration and cross-collaboration among teams, to ultimately develop a lean and robust CI/CD pipeline which has shown to be very performant and suitable for our needs. In doing so, we also achieved de-coupling and portability – which are good software engineering principles. We describe our solutions in as much detail as we can, to support the development of the knowledge-base in the areas of CI/CD and encourage further collaboration between academia and industry.

Future work includes researching how to automatically scale the number of build/release agents up and down based on the current build/release load. While Kubernetes makes the scaling easy, we do not as yet have a way to query the load and respond in real-time, to add more (or even reduce the number of) agents. Our current customized Docker image for our build/release agent has proprietary information which we are removing; and the same applies to the current scripts that were developed in-house. Once done, we will make these freely available to the larger community.

[10] https://jenkins.io/

REFERENCES

[1] J. Allspaw and P. Hammond. 10+ deploys per day: Dev and ops cooperation at Flickr.
https://www.youtube.com/watch?v=LdOe18KhtT4
Accessed: May 28th, 2018.

[2] C. Caum, Puppet. Continuous Delivery Vs. Continuous Deployment: What's the Diff?
https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff
Accessed: June 10th, 2018.

[3] A. Dyck, R. Penners, and H. Lichter, "Towards definitions for release engineering and DevOps", in Proceedings of the 3rd IEEE/ACM Workshop on Release Engineering, pp. 3-3, Florence, Italy, July 2015.

[4] S. Guckenheimer, Microsoft Corp. "What is DevOps"
https://docs.microsoft.com/en-us/azure/devops/what-is-devops
Accessed: June 12th, 2018.

[5] M. Hilton, N. Nelson, T. Tunnell, D. Marinov and D. Dig, "Trade-offs in continuous integration: assurance, security, and flexibility", in Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE), pp. 197-207, Paderborn, Germany, September 2017.

[6] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig, "Usage, costs and benefits of continuous integration in open-source projects", in Proceedings of the 31st IEEE/ACM Intl. Conference on Automated Software Engineering (ASE), pp. 426-437, Singapore, September 2016.

[7] J. Humble. "Evidence and case studies"
https://continuousdelivery.com/evidence-case-studies/
Accessed: June 12th, 2018.

[8] Microsoft Corp. Build and Release Agents.
https://docs.microsoft.com/en-us/vsts/pipelines/agents/agents
Accessed: May 31st, 2018.

[9] Puppet and DevOps Research and Assessments (DORA). 2017 State of DevOps Report.
https://puppet.com/resources/whitepaper/state-of-devops-report.

[10] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor and M. Stumm, "Continuous deployment of mobile software at Facebook", in Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), pp 12-23, Seattle, WA, USA, November 2016.

[11] Varidesk Inc., Corporate Website – 'About Varidesk' at: https://www.varidesk.com/company/about-varidesk, 2017.