

Microstructures

And other Velocity Drivers

Paul Pogonoski

 Copyright ©2022 Paul Pogonoski. All Rights Reserved.

Table of Contents

[Introduction 3](#_Toc118810794)

[Velocity Enablement and Gain 7](#_Toc118810795)

[Removing the Funnel to the centralized service teams 7](#_Toc118810796)

[Development teams responsible for the whole Lifecycle of anything they produce 9](#_Toc118810797)

[Microstructures 10](#_Toc118810798)

[Microstructure Ownership and Responsibility 14](#_Toc118810799)

[Restructuring the DevOps Team 16](#_Toc118810800)

[1. Creation and Management of Shared Services 18](#_Toc118810801)

[2. Creation and Management of Value-added Utilities 19](#_Toc118810802)

[3. DevOps Advisory Services 22](#_Toc118810803)

[4. Shared Production Support 24](#_Toc118810804)

[Implementing a Velocity Enabled Organization 25](#_Toc118810805)

[IaC Management 25](#_Toc118810806)

[Pipeline Management 26](#_Toc118810807)

[Build Artefact Management 26](#_Toc118810808)

[Cloud Security Changes 27](#_Toc118810809)

[Change Management Processes 27](#_Toc118810810)

[Monitoring and Alerting 27](#_Toc118810811)

[On-call Management 28](#_Toc118810812)

[Supporting the Lower Environments 29](#_Toc118810813)

[Shared Support Dry Runs 30](#_Toc118810814)

[Review and Test the impact on DR 30](#_Toc118810815)

## Introduction

I came into DevOps almost by accident. I’ve worked in Software design and delivery for most of my professional life, which started too long ago that I wish to detail. In will say, however, that I started as a COBOL programmer building a Sales Order Entry system before going on to write Assembler for the Operating system, all of which was on an IBM System 360 Mainframe. Then I worked with Minicomputers, PC’s, Client-Servers, CORBA, Web Services, SOA, API’s and Microservices using languages like C, C++, Java, Python, JavaScript and Node.js, and Golang; all governed by Waterfall, and Agile. Is that the summarized history of modern computing? Possibly, but it is a summary of my pre-DevOps career.

However, in 2015 I was offered and opportunity, by my then employer, to move to the UK as part of a plan to sell the company to a huge international services organization that had, one year before, bought the sister company of ours. The sister company had built a cloud design product that was agnostic and also supported bare metal. Back in 2015 this was of great interest. Our company had built a very flexible and powerful API Management provider. Back in 2015 this was also of great interest, and the two had synergies that made them attractive.

The plan failed, for reasons I won’t detail, so it left me in a position of being in the UK and needing to find a place in this huge international services organization, that I had joined as part of the sales plan. The obvious path was to become the EMEA expert in the cloud design/delivery product. So, I moved from Software Product build to Infrastructure and Cloud Professional Services.

Just Prior to this I had had some exposure to Cloud via AWS, where I migrated the API management Demo installation from CD’s distributed by post across the world, to an AWS AMI that could be used to create a temporary EC2 VM whenever a demo was necessary.

As luck would have it, while working for a client in South Africa in 2013, consulting on my company’s SOA and API Management solution for a very large Pay TV operation down there, the client asked me to brief them on this new approach to delivering Software called “DevOps”. So, my introduction to “The Phoenix Project” was made.

So, moving into working with the Cloud Management Product wasn’t entirely a green field for me.

Such is the way I do things; I began to re-read “The Phoenix Project” and began to see subtleties and lessons lost on me on the first read. I also consumed document after document and attended presentations on DevOps from various sources.

I became convinced that DevOps was the next, obvious, step from Agile in the evolution of computing. The problem is, apparently, I was the only one!

Even while at ThoughtWorks their focus was on Software Development and not Delivery. There was great work coming from Jez Humble, John Allspaw, and Kief Morris, but those at the top of the ThoughtWorks tree thought this was orthogonal to software improvement and velocity. It was still just about the infrastructure that software run on. Over time I have learnt to dislike the word “infrastructure” a great deal as it was a convenient way of justifying that DevOps had no impact on software development and delivery.

My most recent years my work has been one of delivering into public clouds, as these bundled services have become more mature and accepted by most businesses. The fundamental differentiator between a public cloud and any other cloud variant is that all services can be utilized and managed via an API. This knowledge has led to an epiphany for me:

Public Clouds have no infrastructure (at least none that the user sees). They are all a bundle of Software Services that you access via API’s and utilize to run your software.

So, if public clouds don’t have infrastructure, only software interfaces, what is the difference between using them and using SQL to utilize a RDBMS? Similarly, what is the difference between using them and a Ping Identity API, developers interact with to provide Authentication and Authorization for their services?

I pose these questions as a mechanism to reset your thinking in readiness for the ideas I’m about to put forward. Collectively I call them Velocity Enablement and Gain, or just Velocity Enablement.  
Velocity Enablement is a loose collection of practices that I term Velocity Drivers. As the title of this book suggests, Microstructures are one of these Velocity Drivers, and probably the most important. Others are:

* Structuring a DevOps team to be responsible for:
  + Shared Cloud Services
  + Velocity Enablement
  + DevOps Advisory Services
* Having Dev teams take responsibility for their own Microstructures
* Shared Production Support

We’ll take a detailed look at these Velocity Drivers in the remainder of this book.

Before we do that, Let’s explore why this book is concentrating on Velocity and Velocity Enablement and Gain.

**Velocity** is a metric for work done, which is often used in agile software development. [[1]](#footnote-1) It is fair to say that it has had a chequered history and that there may be better metrics [[2]](#footnote-2), however using the term, if not the metric, is still useful in conveying a sense of the ability to deliver required business ability. So, with this in mind, why is this book emphasizing it? To be honest, as a practicing engineer, I’m always more concerned with Security, Resilience, Performance, Scalability, and Maintainability.

The reality is, in today’s world, it is very expensive to create business value from computing. It may be necessary in some cases, but if the cost of delivering that value can be minimised a business to use those methods (hence the dream of No-Code[[3]](#footnote-3) systems becoming fashionable again). Therefore, regardless of the measure, if Velocity is increased more is done in the same amount of time, or, as I like to look at it, *waste is eliminated*.

*Improving daily work is even more important than doing daily work [[4]](#footnote-4)*

Therefore, improving ways of working, changing ways of working, or eliminating ways of working, can return back to the Dev team time and resources they can use to say:

* Improve the scope and detail of Performance Testing, or for some organizations, start conducting Performance testing
* Start, or do more Code Reviews and/or Walkthroughs
* Introduce, or improve, Vulnerability Testing
* Review the design for completeness and correctness
* Introduce Threat Modelling sessions

All these activities are seen as de-rigor for a modern, high performing, development team. However, the reality is that there are not that many modern, high performing, development teams in existence, even in “Poster Child” organizations like Google, Apple, Meta (Facebook), Spotify, etc. They also have a common goal, to reduce the Unplanned Work of Downtime, bugs, Performance issues and Security Breaches.

*Unplanned Work is what prevents you from doing planned, or productive work. Like matter and antimatter, in the presence of unplanned work, all planned work ignites with incandescent fury, incinerating everything around it. Like Phoenix. [[5]](#footnote-5)*

This is why I use the term Velocity Improvement and Gain. It’s a metaphor, a pointer, to an organization’s ability to create High Performing Teams [[6]](#footnote-6).

It is also important for me to state that Velocity Enablement and Gain is only practical in a Public Cloud. It’s Cloud Native, if you like. The reason I say this is that Velocity Enablement relies on the ability to break up Cloud Services into bundles that support a Microservice – the Microstructure. It is often not possible to do this with Data Centre Infrastructure, and Private Clouds are more often than not a renaming of and outsourced and shared Data Centre, making them as unsuitable as a physical Data Centre. For this reason, I will not be talking about anything other than Public Cloud Services in this book, not Infrastructure.

I have only one ask of you, the reader, going forward, and that is to keep an open mind, if you have issues with anything I propose then please ask yourself why? and keep on being enquiring and looking for “better ways”.

## Velocity Enablement and Gain

Velocity Enablement and Gain, or just Velocity Enablement, or VE, is a loose collection of practices called Velocity Drivers:

* Microstructures
* Structuring a DevOps team to be responsible for:
  + Shared Cloud Services
  + Velocity Enablement
  + DevOps Advisory Services
* Having Dev teams take responsibility for their own Microstructures
* Shared Production Support

All of these have the combined effect of

1. having the entire Application Lifecycle of anything utilized in the Production Environment managed by the Development team that created it;
2. and removing the funnel effect that occurs for any centralized “service” team, what most organizations have as there DevOps team(s).

When this happens your software delivery organization is freed up to maximize development and delivery velocity. Hence the use of the term Velocity Enablement.

By now you’re are probably realizing that VE looks like a somewhat radical re-alignment of what’s seen as the current responsibilities in software development. It’s not that radical, as I aim to convince you of. It certainly doesn’t require re-organizing teams, or introducing new teams, or even taking on new methodologies. What it is, however, is a very practical (This has been successfully used, to greater or lesser extents, in the most recent companies I’ve worked for) means of achieving “Shifting Left” as some vendors and consultants have labeled DevOps.

Let’s take the above points in reverse order.

### Removing the Funnel to the centralized service teams

Centralized teams have become the de-facto model for creating and managing service teams, like Desktop support, Networks, DevOps, Production Support, and so on ad-nauseum.

Given Conway’s Law[[7]](#footnote-7) this is not surprising, and a sensible approach – pre-cloud. Before the cloud organizations had their own datacenters or outsourced the provisioning and maintenance of hardware. Hardware, whether bare-metal or Virtual Machine based, was physical, finite, and required specialist resources to manage. It required constant monitoring, maintenance, and resource planning. It spawned ITIL and ITSM as a means to better understand the state of your constant monitoring, maintenance, and resource planning of your hardware. Therefore, centralizing teams responsible for this critical infrastructure was not only seen as instinctive but best practice.

Unfortunately, when organizations move to public clouds, even if they refactored their applications to use Microservices, they retained the centralized structure for the teams managing the cloud services. Consequently, the software-based services of the cloud we still seen as infrastructure – with all the connotations that that word brings (infrastructure is centralized, it’s bespoke, it’s slow to design and implement because it required specialized skills), even when the notion of DevOps became popular. DevOps teams were, effectively, renamed operations teams.

So why is this an issue for organizations that use Public Clouds? Let’s look at “wait time” as defined in The Phoenix Project:

The wait time for a given resource is the percentage that resource is busy, divided by the percentage that resource is idle. So, if a resource is fifty percent utilized, the wait time is 50/50, or 1 unit. If the resource is ninety percent utilized, the wait time is 90/10, or nine times longer. And if the resource is ninety-nine percent utilized the wait time is 99 time longer.[[8]](#footnote-8)

So, if the centralized team is “the resource” then there is a natural funnel that builds up a queue to the team that naturally slows down access to it the more it’s utilized.

Having a centralized DevOps team that Development teams rely on for the provision of cloud services is serious and inevitable retardant to Development and Delivery Velocity.

### Development teams responsible for the whole Lifecycle of anything they produce

OK, this may seem radical, and you will have to have a conversation with the teams about the perceived “extra work” of Production Support, but it is about time we discussed the “Elephant in the Room”. That is, why don’t development teams support the software they produce after hours?

There are 3 categories of failure to systems when in Production:

1. Infrastructure failure – networks, storage, Database, or compute resource
2. Software failure – catastrophic logic bug, situation not supported, fundamental design limitation
3. Partial, or Temporal, Infrastructure failure causing Software Failure – temporary loss of network, DNS error, Memory loss, load balancer swap

Two out of those 3 reasons involve Software. Also, the first category should be eliminated if software is designed correctly for a Public Cloud. Yet IT departments still organize around the datacenter model where there was a significant risk of infrastructure failure.

I’ve endeavored to find any study that analyses systems failures in commercial IT and looks at the above categories to try to break-down and understand the current risks in failures in cloud computing. Alas I have found nothing. Perhaps this is tied to the purveying assumption in the computing industry based on its non-cloud history and experience?

Anecdotally, over the past year supporting a modern distributed solution on a public cloud, there was a single Sev-1 issue due to a temporary cloud service failure, compared to 3 Sev-1 issues due to software failures. In every case of the software errors the DevOps team were the first responders and took at least 30min to 1 hour to determine that a development team representative was needed to address the problem.

Anecdotally, again, in the 6 months before the last 12 months supporting the same solution, there were no Sev-1 issues due to a cloud provider service, but there were 6 Sev-1 issues due to software failures.

I accept that what I have outlined is a poor sample size for anything other than a high-level pointer, but a difference between the first 6 months and the last 12 months is that the company I worked for decided to make the developer teams responsible for supporting their code in Production using a “Shared Support” model (more on this later in this book). This was gradual and, to-date, they still don’t support the code out of hours, but the impact was noticeable and discussed when describing the Shared Production Support Model.

## Microstructures

Microstructures a key Velocity Driver. In fact, the key Velocity Driver in that their existence influence the remainder of the Velocity Drivers.

So, what are they?

Firstly, I wish to make a clarification. I will be using the word “infrastructure” in this chapter. This is not, in anyway, an approval (tacit or otherwise) of the use of the word, especially when a public cloud is involved. I’ve already, and will continue to remind the reader, that the use of the word is no longer reasonable in the world of the public cloud. However, I’m using it originally for two reasons:

1. I’m trying to be agnostic and not cloud specific
2. I use part of the word in the new term, Microstructure. Thus, wishing to make the subconscious link.

As its name suggests, a Microstructure is an analogy of a Microservice, in that they are the subsets of the total infrastructure that used to support the total solution. And just like Microservices, they can be grouped, or defined, by the functional domain they support. However, luckily, unlike Microservices, they are more easily identified and grouped.

Simply put, a Microstructure is the grouping of the infrastructure that supports a Microservice.

So, if a Microservice is a group of containers running in a Kubernetes instance, then candidate infrastructure components would be (being as Agnostic as possible):

* The Kubernetes Namespace definition
* The Kubernetes Pod(s) definition
* The Kubernetes Service definition
* The Kubernetes Ingress definition
* Any Kubernetes Secrets or Env Vars
* The Load Balancer for the Service Ingress
* The Firewall rules
* The Database definition

Now, some of these would be less likely candidates if the Kubernetes was not a cloud service. But if it was on AS, say, then all of them would be in the Microstructure for the Microservice:

* The EKS Namespace Terraform resource
* The EKS Pod(s) Terraform resources
* The EKS Service Terraform resource
* The EKS Ingress Terraform resource
* The AWS Secrets Manager Secret Terraform resources
* The AWS SSM Parameter Store Terraform resources
* The AWS ALB Terraform resource
* Any AWS S3 buckets Terraform resource needed for the Microservice
* The AWS Security Group Terraform resource
* The AWS IAM Roles Terraform resource needed for the Microservice
* The AWS RDS Cluster and DB Terraform resources

All of these would be defined in the same single GitHub (or equivalent) repo, and most likely the same repo as the Microservice code.

The CD pipeline would then be able to incorporate a conditional Microstructure pipeline that could be invoked to create, modify, or destroy the Microstructure.

At this point you are probably asking:

1. So why these components? For instance, why wouldn’t there be a single ALB, or SG, or RDS cluster; and what about the EKS definition?
2. Hmmm, who’s responsible for the conditional Microstructure pipeline? That is, aren’t you in danger of having different parts of the pipeline owned by different teams?

Two very good questions, so let’s take them one at a time, starting with question 1.

These components were chosen because they can be uniquely, and singularly, tied to the Microservice in this instance of using the AWS public cloud. Any cloud service that is shared cannot be considered in the Microstructure and is a Shared Service and the responsibility of the DevOps team (more on this later).

Further, making the ALB, SG, and RDS cluster unique and separate per Microservice has very little impact of pricing, and my cost lest then a single, large, shared instance of them, because:

* SG’s are free
* An ALB is changed per instance, per hour, but the lifecycle management costs are reduced
* An RDS instance may well be cheaper as costs are defined by RDS instance type, and these will be clearly much smaller and cheaper than a very large cluster for everyone. Also, the lifecycle management costs are reduced as you can upgrade/change without effecting other services.
* In lower environments these resources can be removed or shutdown when not in use, further saving money and allowing greater flexibility
* When the Microservice is removed from service, then the supporting Microstructure can be similarly removed with a single instruction to Terraform, making the lifecycle cost of the Microservice much less because management of it is simpler.
* Deploying the Microservice into different regions is significantly simplified by adjusting the pipeline to run in that region.

This is a concrete example of the impact of replacing infrastructure with cloud services. They are cheap and simple to define and deploy, and the notion of a large, shared, complex, and non-modifiable, shared service, or infrastructure, needs to be retired, unless when it makes sense (more later).

Now for the second question.

If your organisation does not require a development team to own and run its own CI/CD pipelines, then why?

CI and CD pipelines should be separate and distinct. The CI pipeline responsible for the build, test, and packaging for the software artefact. Most likely a container image placed into an artefact repository. The CD pipeline may then, but not always, be initiated to take that, and possibly other artefacts, to deploy the Microservice into the required environment. This separation allows for the same artefact(s) to be deployed many times in many places, under different conditions. It also allows for multiple development streams (maybe branch oriented or may be trunk-based) to create their own artefacts without impacting the CD pipeline.

This separation may seem the perfect reason to make the Dev team responsible for the CI pipeline, and the DevOps team responsible for the CD pipeline. Then adding the conditional Microstructure pipeline to the CD pipeline would seem to make this separation even more sensible. But it shouldn’t happen, especially if deploying to a public cloud.

As already mentioned, the Microstructure definitions are likely in the same repo as the Microservice code, because they are, in fact, symbiotic. Because of this the Dev Team should own and be responsible for the CD pipeline. They are responsible for the entire operational ecosystem of the Microservice, except for the overarching Shared Service of the EKS cluster, the VPC the cluster is in, the network in that VPC, and the account the VPC is in.

This is a very practical example of what most people think is “shifting left”. However, the initial definition, as described in The Phoenix Project, was about moving testing, and proving out the solution, as soon in the lifecycle of the application/Microservice. Regardless of which definition you subscribe to it still works because the Dev team is relying on no-one but themselves in making sure what they are delivering is a resilient, error free, scalable and performant as the Product Manager’s SLA’s require (more later).

So, what of infrastructure that is shared, that isn’t in a Microstructure? These services will be managed by the DevOps team as shared services they offer to the development teams. More on this in the chapter on the structuring of the DevOps team.

I said at the start of this chapter that I considered Microstructures the key Velocity Driver. How does separating and bundling infrastructure, or cloud services, increase Development and Delivery Velocity?

Delivery Velocity seems more self-evident, in that the Dev team is not waiting for a separate team to deploy the infrastructure before they can deploy their code, especially when that separate, centralised, team is fully, or near fully, utilized. On top of that, the infrastructure may be part of some larger monolithic infrastructure, meaning that there would be wait time scheduling changes. So, separation allows the Dev Team to take ownership and modify as needed (more next chapter).

Development Velocity does seem a harder claim to make. So, if the Dev Team owns the Microstructure in Production, it would naturally flow that they also owned it in the lower environs. If the team owned it in the lower environs, then they could create, modify, or remove as they needed, not waiting on the separate team as above, thus compounding the impact. Thus, design changes can be easily accommodated as issues are found in an iterative, agile, step delivery of the code. Indeed, the Microstructure would be built up the same way, with an initial MVP then additions as the code dictated. This would result in much greater velocity.

What I’ve just demonstrated is that Microstructures are clearly Velocity Drivers, just like their software counterparts, but it is the key driver because it is the catalyst for other Velocity Drivers. In this case the Dev teams become the owners of the Microstructures.

## Microstructure Ownership and Responsibility

At this point, I hope that you are beginning to understand why Microstructures is considered the key Velocity Driver. As with everything in Velocity Enablement, Microstructures are not a radical re-engineering of your cloud Architecture, they require no new technology or skills, just an adjustment in thinking re responsibility and ownership.

Microstructures gift a significant amount of agility and velocity to the Dev teams, as well as a significant amount of resources are freed up in the DevOps team for them to take on new challenges (more on this later).

Microstructures also, over time, gift operational and runtime knowledge to the Dev teams. Thus, making the critical Non-Functional aspects of their code transparent, removing the fear, or distrust, that opaqueness brings and much easier to understand and deliver.

There may be those amongst you arguing that the SRE movement does this. I would say that it doesn’t, for the following reasons:

1. Just like with DevOps, the industry did not take time to understand where the term came from and what it meant in that context. Instead it applied its own definition based on its existing prism of the world, thus making SRE’s more like existing positions with existing skill sets. If those organisations that employ SRE’s used them as per the original definition, then It would look similar to the DevOps team I will propose in the DevOps team structure chapter, but
2. SRE’s take the responsibility for reliability away from Developers and assume it themselves, in that centralised, funnel generator, SRE team.
3. If not in a centralised team, SRE’s end up being embedded in the Dev team and becoming the “infra Guy/Girl” of the team. Again, taking the responsibility for reliability away from Developers.

The key point about Microstructures is that the Dev team now becomes responsible for the complete lifecycle of the code they produce, with modern tools and technology making this responsibility much easier and attainable.

Even assuming Dev teams want to take on this additional responsibility, and that they understand and accept that this will allow them to have net more time to activities they currently don’t have time for, how will they attain the skills and knowledge that allows them to do this?

The answer to this question becomes clear in the next chapter. However, in summary The DevOps team(s) will restructure, both to take advantage of the time and resources they have been given back, but also to allow them to advise and mentor the Dev teams so that they will, in combination of the teams taking on formal training on IaC and CasC tools and practices, attain these abilities over a short amount of time.

The addition of new value-added utilities will save them time and effort, and the need to develop and manage Pipelines, IaC, and CasC themselves.

The advice and mentoring received from the DevOps team(s) via their Advisory Service, will considerably shorten any learning time while they “Learn as they Do”.

The support and mentoring received via participating in Paired Support on incidences, will, again, considerably shorten any learning time while they “Learn as they Do”. Especially with respect to problem detection and solving in a high-pressure situation; as well as Root Cause Analysis post incident.

To repeat some salient points in respect to the changes Velocity Enablement requires of Dev and DevOps teams:

* It makes existing teams take on new responsibilities via the redistribution of responsibility based on the Agile Application Lifecycle
* It does not change make-up of the teams wrt to their experience and skills
* It does require the learning of skills that will increase an individual’s value
* It does not abandon the teams to learn those skills by themselves
* It makes teams share the burden of Production support
* It creates a “virtuous circle” of improvement based on lessons learnt from Production incidents, creating increasing MTBF (Mean Time Between Failures) and increasing improving MTTR (Mean Time to Recovery)
* It creates time to be able to apply the lessons learnt into the improved Development practices

## Restructuring the DevOps Team

We briefly touched on SRE’s in the previous chapter, lets describe what an SRE is:

So, what exactly is Site Reliability Engineering (SRE)? We admit that it’s not a particularly clear name for what we do—pretty much every site reliability engineer at Google gets asked what exactly that is, and what they actually do, on a regular basis.

Unpacking the term, a little, first and foremost, SREs are engineers. We apply the principles of computer science and engineering to the design and development of computing systems: generally, large distributed ones. Sometimes, our task is writing the software for those systems alongside our product development counterparts; sometimes, our task is building all the additional pieces those systems need, like backups or load balancing, ideally so they can be reused across systems; and sometimes, our task is figuring out how to apply existing solutions to new problems.

Next, we focus on system reliability. Ben Treynor Sloss, Google’s VP for 24/7 Operations, originator of the term SRE, claims that reliability is the most fundamental feature of any product: a system isn’t very useful if nobody can use it! Because reliability2 is so critical, SREs are focused on finding ways to improve the design and operation of systems to make them more scalable, more reliable, and more efficient. However, we expend effort in this direction only up to a point: when systems are “reliable enough,” we instead invest our efforts in adding features or building new products.

Finally, SREs are focused on operating services built atop our distributed computing systems, whether those services are planet-scale storage, email for hundreds of millions of users, or where Google began, web search. The “site” in our name originally referred to SRE’s role in keeping the google.com website running, though we now run many more services, many of which aren’t themselves websites—from internal infrastructure such as Bigtable to products for external developers such as the Google Cloud Platform.[[9]](#footnote-9)

So, an SRE:

1. Build or extend Computing Services (or systems) for use by product developers
2. Ensure reliability of those Services and the products that user them
3. Wrap these systems in useful interfaces for easier/better use or combination

These seem more akin to the public cloud engineers than to the SRE’s employed in most organisations these days. This is why, regardless of the model you use for an SRE, it’s not supportive of Velocity Enablement, and not the model used for a DevOps team organised to support Velocity Enablement.

The model the model used for a DevOps team organised to support Velocity Enablement is:

1. Creation and management of Shared Services
2. Creation and management of value-added utilities
3. IaC and other DevOps Advisory services
4. Shared Production Support

The proportion of time spent on each category of activity depends on the maturity and state of the organization the team belongs to, but can approximately say that one third is spent on category 2, with a further third spent on category 3, and the final third spend on categories 1 and 4. No category has a higher priority than any other. Although it is hoped that categories 1 and 4 do not need much time spend on them at all, especially of using a public cloud. Let’s examine these activities.

### Creation and Management of Shared Services

So Shared Services are simply the cloud services, or hardware infrastructure, that are not in a Microstructure because they have more than one user. They are usually services that run the code a Dev team builds, deploys, maintains, and, hopefully, supports (more on this later).

Let’s use our AWS model, defined in the Microstructures chapter as an example. The Shared Services would be:

* The AWS EKS service Terraform resource
* The AWS VPC Terraform resource that the EKS service was deployed in (including routes, subnets, and other networking components)
* The AWS Account Terraform resource that the VPC was defined in
* The AWS IAM users Terraform resource (unless using Azure AD user provisioning within the AWS SSO)
* The AWS AIM Shared Roles Terraform resource
* The AWS IAM Policies Terraform resource

As you can see, with the exception of the EKS service, these are all base services needed for any AWS-based cloud, but nothing to do with the delivery of software. It should be considered a red flag if the DevOps team is spending its resources on any service, or Hardware Infrastructure, that is used for software delivery, unless it is shared, as in the case of EKS.

So, let’s take a closer look at how the EKS service should be supported.

You should have noted that the Microstructure defined in the Microstructures chapter included the following:

* The Kubernetes Namespace definition
* The Kubernetes Pod(s) definition
* The Kubernetes Service definition
* The Kubernetes Ingress definition
* Any Kubernetes Secrets or Env Vars

Given the DevOps team is responsible for the EKS service, it’s a good question to ask why are these resources part of the Microservice a Dev team is responsible for? Instinctively, you might assume that DevOps should be responsible for them as part of the Shared Service, but this does not aid Velocity enablement and is the old, Infrastructure support team, thinking that we are moving away from.

Let’s ask another question, the same question we asked in the Microstructures chapter:

*Are these resources needed for the correct running of other resources in EKS?*

The answer is easily answered as ‘No’. Therefore, they should be part of a Microstructure for that Microservice and managed be the Dev team that owns the Microservice.

But you may ask, doesn’t this make managing the EKS, and, in particular, Security, harder for the DevOps team, and would that extra burden outweigh the benefits of having them in a Microstructure? This is a fair question, and one that should always be asked.

The answer, of course, is no. Kubernetes Namespaces were conceived to support this very use case. And, yes, allowing a Dev team to willy-nilly define their own Namespaces could lead to security vulnerabilities, unless the DevOps team provided and maintained the Terraform Module that did this in it’s own Terraform module repository (with via an Open Source Citizen instance, or via the Hashicorp repository). This is an example of a category 2 activity, as defined above (more on this next).

So, we can see that by applying a simple and logical question to the provisioning and maintenance of cloud services, and hardware infrastructure, the existence of both Microstructures and Shared Services come into existence, with their inherent Velocity Enablement for both the Dev teams and the DevOps team.

### Creation and Management of Value-added Utilities

Once the DevOps team is released from the responsibility of managing Microstructures, they can now utilize some of that time and resources to making life simpler, easier, and more secure for the Dev teams. They do this by creating utilities that, for example, automate repetitive tasks via Pipelines, IaC, CasC, and Bots, and create helper Terraform Modules and Inner Sourcing. Let’s look at some of these examples.

#### Terraform Modules

We have already hinted at these in the above section. Terraform [*Modules*](https://developer.hashicorp.com/terraform/language/modules)are containers for multiple resources that are used together. These produce very handy abstractions for repeatability and tidy code. Most of the time these modules only make sense within the context of the problem you are working on. Sometimes. However, it makes good sense to refactor one or more of these modules to be used by anyone. When you do this its best to use a private registry that is compliant with Hashicorp’s Registry API. The Open Source [Citizen registry](https://github.com/outsideris/citizen), or [Hashicorp’s Private registry](https://developer.hashicorp.com/terraform/cloud-docs/registry) are good for this purpose. Using a registry makes sharing and accessing the modules easy and is Hashicorp’s suggested best practice.

To what would be a good candidate for a shared module? Anything that is repeatedly used that requires authorisation to perform a task, like the Kubernetes Namespace creation requirement as mentioned above; or DB creation within a cluster, or route DNS domain names. In short, anything that a DevOps team might be asked to do themselves, because Security restrictions, but want it to be part of a Microstructures that the Dev teams might create.

#### Automated tasks

These may usually be tasks automated by one or more Terraform repos but can also be some configuration tasks via Ansible or Python. If Terraform is involved, these would be more complex workflows than can be encapsulated in a module.

One example that I’ve been involved in was a Terraform repo that managed the on-boarding/off-boarding of new engineering staff for AWS, RDS (PostgreSQL and MySQL) clusters, MongoDB clusters. Instead of doing these separate tasks manually using a number of different interfaces and having no record of who had access to what, the Terraform worked off a large JSON map that was the item of record of our engineering employees and their respective permissions. This repo took that JSON and created an individual Secrets Manager Secret per employee per Environment in JSON format. It used this to CRUD the AWS permissions in AIM, and later AWS SSO, while separate Terraform in separate repo’s took the same secret to define what was needed for RDS and MongoDB. All controlled by pipelines.

Another example of common modules would be the packaging of monitoring configurations, whether on the collection side, or the display side, and could include SLO modules if your organisation has adopted that approach.

Another example creating and managing Jenkins and its users, plugins, and pipelines via Ansible Configuration as code. Nothing for that Jenkins was defined via the UI console (which was read-only). Changes were made by the Dev team members to the limited CasC files and submitted a PR for the DevOps to vet and approve. Once approved the changes were merged and a pipeline run to CRUD the users, or Plugins, or Pipeline defs, depending on the change. Here was an example of Inner Sourcing the changes in the CasC to improve the velocity of Dev teams and removing that centralised funnel.

Taking these examples further, that could be wrapped in service bots running in Microsoft Teams or Slack, that interactively guides the users to input the requisite information then update the repositories (instead of a person making those changes as above), and the requisite pipeline(s) then initiated to actuate the necessary changes. Some call this “Bot Ops” but I worry about the old-school “Operations” connotation of this term.

I hope I’ve given enough examples to illustrate the types of activities involved here, and why I expect the combined work in this category of work to be at least one third of the work undertaken by a DevOps team that supports Velocity Enablement.

Too little of this work is currently undertaken by most DevOps teams at the moment because they are monopolised by support work, either because they are a centralised support team, or by the expectation that anything post Testing is the responsibility of DevOps – the old “Ops Wall”. Adopting Velocity Enablement, shifting responsibilities via Microstructures not only frees up the DevOps team to do this work it, to some extent, requires it.

### DevOps Advisory Services

While the creation of Microstructures may, or may not, necessitate the DevOps team to start building value-added utilities, it most definitely does necessitate the DevOps team to deliver Advisory Services.

It is just not practical to shift the responsibility of creating and supporting Microservices to the Dev teams without the DevOps team providing support for them in the form of Advisory Services. That is organizing with the teams planned sessions where a DevOps team member walks a Dev team through the organizations Cloud Architecture, or Standards, or Principles, or examples of IaC or CasC.

**Note, Advisory Services are not the creation or support of Microstructures on the team’s behalf, instead it’s the transfer of knowledge and skills to enable the ability of the Dev teams to create and support the Microservice(s) themselves. However, this does not include training for tools like Terraform, or Ansible, or Python. There are plenty of excellent online training for the Dev teams to engage themselves.**

So Advisory may include taking a team through the modules in a Terraform modules registry, what they do and how to use them. It might be carrying out a similar exercise for the code repo’s that support inner sourcing; or the Bots available to be used.

In some circumstances, it may include reviewing new Terraform code, or being engaged in the design for some new part of a Microservice. It should not include debugging IaC or CasC code except under exceptional circumstances, like an organization moving to a Velocity Enablement model.

Certainly, any output from building value-added utilities should lead to Advisory work, including the creation of Wiki pages and the advertisement on them.

You can see by the type of activities undertaken in the Creation and Management of Value-added Utilities chapter that Advisory Services are not intended to be SRE activities because the activities are about knowledge transfer. However, if an organization wanted to emphasize these types of activities, they must accept that they’ll be undertaken by the Dev teams as owners of the code and Microstructure(s) that support it. Therefore, including SRE activities in advisory would be about knowledge transfer on how to ensure resilience, shared modules that could be used, etc. That is knowledge transfer.

Further to this, Advisory Services are never provided by a DevOps team member allocated to a Dev Team. That is any time some advisory activity is provided to a team it can be safely assumed that different members of the DevOps team would provide it. This is for a number of reasons:

* DevOps don’t have the temptation to “go native” with a Dev team; which aids the following point.
* Stops the temptation of a Dev team to make the DevOps team member “the infra guy/girl”.
* Ensures the Dev teams are exposed to all DevOps team members.
* Ensures the Dev Teams are exposed to different skill and knowledge levels within the DevOps team.

### Shared Production Support

Shared Production Support, or just Shared Support, is very much like the Agile Development practice of Shared Programming, except in a Production Support context then a Development context. That is, like Pair Programming where two developers would periodically get together to address some difficult coding component of what the team may be building, and one takes the lead while the other observes and suggests, when there is a production incident involving a Microservice both the on-call representatives for both the Microservice owning Dev team and the DevOps team get together on a video call, with the most knowledgeable person taking the lead (or the Dev Team representative if this takes toom long to determine). They would both then support each other in following their support protocol and determining the best course of action to remove the issue.

This support model is logical for the following reasons:

* The issue could be the result of an issue with the Microservice, it’s Microstructure, a combination of Microservices/Microstructures, a Shared service, or a combination of all
* Modelling off Shared Programming results in a known and repeatable structure in which multiple teams can work efficiently
* Modern Monitoring and Alerting tools and practices can easily be aligned with it.
* It reinforces the distributed nature of responsibility needed for a modern distributed and cloud-native solution architecture.

## Implementing a Velocity Enabled Organization

Let’s assume that you have digested the arguments in this book and have decided to implement Velocity Enablement in your organisation. The first question you might have is “What is the best way to implement it?”.

It’s been stated a number of times That Velocity Enablement is about the re-assignment of responsibilities, rather than a total change and disruption to the organisation, but this still involves significant change.

The best advice is to treat this like a migration and make the changes one Microservice at a time. Plan out the change in responsibilities:

* IaC repo ownership
* IaC repo merge
* CI pipeline ownership
* Artefact ownership
* CD pipeline ownership
* Cloud security ownership/changes
* Change Management process changes
* Environment Monitoring and Alerting permissions and ownership (hopefully encapsulated in the TF repo changes)
* On-call Process changes
* On-call administration changes
* Lower environment CI/CD test runs, including cloud resource destruction and implementation
* Shared Support dry runs
* Review and test DR impacts of changes

I do not intend to go into detail on any of these points, as most should be well known to the reader, but will explain the impact from, or the impact to, Velocity Enablement per point.

### IaC Management

This section assumes there is existing Terraform, or other IaC technologies, that manages the lifecycle of the AWS resources that will form a Microstructure or, rarely, Microstructures.

If there is no IaC that exists for the cloud resources or hardware infrastructure, then all work must cease until this is created. ***There should exist no Microstructure that is not managed by IaC***. If such existed, that is created manually via a console, or even the CLI, it would make it impossible to automate it via CD pipeline(s), and Disaster Recovery would clearly be of high risk of failing (more later).

Tasks that could be carried here are:

* The separation and refactoring of code from a monolithic repo
* Merging of multiple instances of separated code into the Microservice repo
* Elimination of duplicated IaC code
* Creation of shared modules

The end result should be that there is clear, unambiguous, and efficient IaC home in the Microservice repo, or a separate repo linked to the Microservice repo for the Microstructure.

### Pipeline Management

There is very likely to be a need to refactor the CD pipeline to at least add logic to support the running of the IaC in cases where the Microstructure is to be created, updated, or removed.

If there was significant refactoring of the IaC the CD pipeline will have to accommodate that and most likely the new Home for the IaC.

If there was a single pipeline responsible for both the building and testing of the runtime artefact, as well as the deployment of it, then it is strongly recommended that this be broken up into separate CI and CD pipelines, as discussed [here](#cicd).

If there are separate CI and CD pipelines, then most likely the ownership of the CD pipeline and permissions on who can run will have to be reviewed and changed.

### Build Artefact Management

Here you are likely to review permissions, most likely reducing or changing the persons/groups that had read access as the same group responsible for its creation is now responsible for its deployment.   
This might not be obvious, but if the Dev team is responsible for the lifecycle of the Microstructure in the CD pipeline, it makes no sense for, say, the DevOps team to be responsible for the Microservice’s deployment. That is, the Dev team is responsible for the full lifecycle of all runtime components of the Microservice. Greatly simplifying Deployment and support of the Microservice, and aiding Velocity Enablement.

### Cloud Security Changes

Here, again, the change in responsibility and ownership of the runtime components of the Microservice will necessitate the need to review and change runtime permissions for those components in the Cloud, or Hardware infrastructure.

This is very likely to mean a reduction, or simplification is needed, in a similar manner to the build artefact permission changes.

### Change Management Processes

The organisation may see some significant changes in this process. The reduction in the number of actors in the process – a singular team now responsible for all changes – could mean that complete automation is now possible.

It is, however, likely to add an additional check/validation as the DevOps team will go from the deployer to the validator – at least additionally as the organisation tries to get use to the change in responsibility. In the longer term there is no, real, reason that the DevOps team needs to be involved at all, unless the deployment utilises a Shared Service.

Please note that that addition of an additional check/validation does not mean the additional step, as an existence validation step could just include an additional team.

### Monitoring and Alerting

It should be pointed out, if it hasn’t occurred to the reader before this, that any Monitoring and Alerting logic/configuration should be considered part of the Microstructure.

Given that the Dev team is the only group of people that understand the Microservice’s design and resource requirements, it is the only group of people that can sensibly define the metrics that need monitoring and the thresholds at which those metrics represent an warning, or something worse, and what to do about it (it might not require human intervention at all. Instead an auto-repair utility may be initiated).

Thus, a review of ownership and permissions in what-ever software is used by the organisation uses is required to accommodate the impact of these and other changes.

### On-call Management

This subject is somewhat vexed, and for the same reason that Velocity Enablement seeks to address. That is, the notion that Dev Teams should be on-call and support any issues with the software they produce. Considering on-call duties highlights this issue fully and does need to be sensitively managed.

Let’s consider some implications:

* If DevOps are compensated for being on-call, will that compensation be reduced because of the reduced responsibility
* Will the Dev’s be compensated for being on-call now that they have increased responsibility?
* If DevOps aren’t compensated for being on-call, will they now be compensated if the Dev’s are?
* How does an organisation manage perception versus reality of being on-call with the Dev Teams? That is, the better quality the code delivered, the least likely an on-call issue will ever occur (excepting a cloud service or hardware infrastructure-based issue).
* Can you allow some Developers to opt-out of on-call responsibility, given their direct impact on causality of on-call issues?

I will not pose to have all the answers here, as Dev and DevOps teams are made up of individual Human Beings that have their own unique perspectives. However, good management is about leadership in adversity or difficulty, and on-call changes are necessary in order to fully extract the value of Velocity Enablement, and the reason for the use of Paired Support to help share knowledge and skills.

One approach to consider, is to have the Dev teams be on-call during office hours initially, say, for the first 3 months of a Microservice they have migrated to be managed by Velocity Enablement. There is no need to consider most of the issues mentioned above as there is minimal change to support. However, it does give the Dev team members time to adjust and become familiar with the realities of support (as opposed to the perception), as well as the ability to practice Shared Support.

### Supporting the Lower Environments

This might not be obvious but being responsible for a Microstructure is just not a single environment focus. All the lower environments must be considered. This may seem a simple statement, but the reality is that this highlights one of the difficult areas of management and coordination. That is, synchronising lower environment cloud service versions (or hardware infrastructure state) with the Microservice version/branch being developed or tested in the lower environments.

Currently, most organisations have significant difficulty in managing this issue for two reasons:

1. The cloud services are managed by a different team than the Microservice, which invariably means;
2. The cloud services are treated as a monolith making changes problematic, especially changes needed frequently.

However, having a Dev team manage the Microstructure that singularly supports a Microservice removes these issues completely.

Now a Microstructure version can simply match the version of the Microservice within the same branch, especially if housed within the same code repo.

This is more powerful than first thought, because the branch/version is an atomic entity, any issues arising out of the changes can only be because of changes in that version. There is no longer an invisible “fence” where blame may be thrown backwards and forwards like a tennis ball because of the lack of knowledge on the other side of the “fence”.

Secondly, the microstructure is now being developed and tested in the lower environments in the same Agile manager that the code is. It may be via different sprints (or whatever “Effort boxing” your adopted Agile Process and ceremonies use), and is probably better if it is (because the same resources can work on both the Microservice and Microstructure, and also issues introduced by changes are more easily identified an separated between the Software and the IaC changes).

When these changes are accounted for and allowed for in the migration planning then disruption is minimised, but the improvements brought about by the reduction in Unplanned Work due to the simplification of ownership in the lower environments become apparent earlier.

### Shared Support Dry Runs

Just as Pair Programming takes time and practice to begin to work well and deliver results, the same will be true with Shared Support. However, unlike Pair Programming, where not getting it right for some time will not have an impact on the businesses bottom line, it may well have a significant impact on a Businesses bottom line if delays and mis-understandings caused a live issue not to be solved within an agreed SLA with a customer, or customers.

To mitigate this risk, it is suggested that organised dry runs be organised in a lower environment. The planned activity should not be known to the Dev team(s) targeted, so as to simulate a live issue as much as possible. Likewise, re the DevOps personnel on-call at the time.

The use of tools like Netflix’s Chaos Monkey would be even better to use as a means of ensuring a failure for a dry run.

In addition, as mentioned above, the nature of being responsible for the runtime of the lower environments will improve co-operation and skills in issue resolution.

What the dry runs should have as objectives are that the personnel should:

* Develop a “muscle memory” on the process of engagement with all stakeholders
* Develop a familiarity with the runtime environments of the Microservices
* Develop knowledge of the skill strengths and weaknesses of the team members wrt time pressures
* Develop Trust between the teams and the team members

### Review and Test the impact on DR

Cloud Native Disaster Recovery is a subject all by itself. However, the greater all components in your cloud are defined via IaC and CasC the greater the likelihood of DR success because the greater the percentage of automation in the provisioning of your cloud.

Adopting a mantra of “nothing is deployed into the cloud unless it is defined by IaC/CasC and automatically deployable” and getting support from senior management to enforce this, will go a long way to ensuring your cloud is re-deployable, and thus helpful is supporting DR.

One area that needs reviewing, and that is often overlooked, is that of ensuring the Cloud Region is not hardcoded in any IaC but provided via an input variable from the pipeline.

An organisation often decides on using a single region when using a Public cloud. This makes sense as it simplifies the architecture and support of it. However, this often leads to that region being hardcoded in the IaC as it’s easier to do this, and nothing is likely to change….. right?

The problem is the mostly likely cause of a need to utilise a DR environment, in a Public Cloud, will be loss of a Cloud Region. This can and has happened. So, if the region is hardcoded in the IaC, no matter how automated, your DR will not be successful or be significantly delayed at best. Using the region from an input variable easily resolves this issue.

You could further automate/simplify this by DevOps creating a pipeline utility that returns the right region for a given cloud account/environment, and this is used for the value if the input variable. This is especially helpful when multiple regions are used, but also ensures consistency.

Therefore, the creation of Microstructures and the commensurate ownership and permission changes resulting from them will have an impact on the DR process and workflows, thus necessitating a detailed review of an organisations DR process when new Microstructures are created.

An advantage of introducing Microstructures to the DR process, is that once the Shared Services are rolled out (themselves reduced by the existence of Microstructures) the Microstructures will be deployed in parallel as their Microservices are deployed in parallel into the Shared Services. Thus improving the rollout velocity of the DR environment(s).

### 

1. https://en.wikipedia.org/wiki/Velocity\_(software\_development) [↑](#footnote-ref-1)
2. https://www.agilealliance.org/glossary/velocity/ [↑](#footnote-ref-2)
3. https://en.wikipedia.org/wiki/No-code\_development\_platform [↑](#footnote-ref-3)
4. ***The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win***  
   Authors: [Gene Kim](https://dl.acm.org/profile/89658608457), [Kevin Behr](https://dl.acm.org/profile/89658623857), [George Spafford](https://dl.acm.org/profile/81388590746)  
   Publisher: IT Revolution Press  
   **ISBN:**978-0-9882625-9-1  
   **Published:**10 January 2013 [↑](#footnote-ref-4)
5. ***The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win***  
   Authors: [Gene Kim](https://dl.acm.org/profile/89658608457), [Kevin Behr](https://dl.acm.org/profile/89658623857), [George Spafford](https://dl.acm.org/profile/81388590746)  
   Publisher: IT Revolution Press  
   **ISBN:**978-0-9882625-9-1  
   **Published:**10 January 2013 [↑](#footnote-ref-5)
6. https://en.wikipedia.org/wiki/High-performance\_teams [↑](#footnote-ref-6)
7. https://en.wikipedia.org/wiki/Conway%27s\_law [↑](#footnote-ref-7)
8. ***The Phoenix Project: A Novel about IT, DevOps, and Helping Your Business Win***  
   Authors: [Gene Kim](https://dl.acm.org/profile/89658608457), [Kevin Behr](https://dl.acm.org/profile/89658623857), [George Spafford](https://dl.acm.org/profile/81388590746)  
   Publisher: IT Revolution Press  
   **ISBN:**978-0-9882625-9-1  
   **Published:**10 January 2013

   [↑](#footnote-ref-8)
9. ***Site Reliability Engineering***  
   Authors: [Betsy Beyer](https://learning.oreilly.com/search/?query=author%3A%22Betsy%20Beyer%22&sort=relevance&highlight=true), [Chris Jones](https://learning.oreilly.com/search/?query=author%3A%22Chris%20Jones%22&sort=relevance&highlight=true), [Niall Richard Murphy](https://learning.oreilly.com/search/?query=author%3A%22Niall%20Richard%20Murphy%22&sort=relevance&highlight=true), [Jennifer Petoff](https://learning.oreilly.com/search/?query=author%3A%22Jennifer%20Petoff%22&sort=relevance&highlight=true)  
   Publisher: O'Reilly Media, Inc.  
   **ISBN: 978-1491929124**  
   **Published:** 23 March 2016 [↑](#footnote-ref-9)