



Office of the CTO Perspective:
Application Enabled Infrastructure –
Practical Cloud Native and the Rise of Application Platforms

Contents

1. Bottom Line	3
2. Technology Overview	5
2.1 Overview	5
2.2 Microservices Primer	8
3. VMware Perspective	9
3.1 Cloud Native Application Platforms Perception in the Enterprise – the Beginnings of Application Enabled Infrastructure Movement	9
3.2 The Era of Application Platforms is Born	12
3.2.1 Application Platform Architect in a Nutshell	14
3.3 What Do CIOs Mean When They Say, “We want to be out of the infrastructure business?”	15
4. Recommendations	15
4.1 HCR with Service mesh	17
5. Conclusion	19
6. Appendix 1 – Brief Look at Microservices Structure.....	20
7. Acknowledgement.....	22

1. Bottom Line

The cloud-native movement has entered the beginning of its maturity phase, where its initial stance of ignoring the existing enterprise has been re-adjusted by IT reality. Its original stance of building only stateless systems is no longer feasible—at some point these systems must be able to manage state as well. The movement is a leading force behind the Application Enabled Infrastructure and Application Platforms thinking that are critical for IT to embrace as part of an effective cloud strategy.

- Fifty percent of customers we interact with have some sort of application platform challenge. Furthermore, in both private and public cloud, customers are provisioning more hardware than what is required—in some cases twice what is needed.
- VMware can help the industry by building a true Hybrid Cloud Runtime (HCR). HCR is a connective tissue between private and public cloud, capable of delivering service mesh and application optimization functionality at runtime—making it truly aware of the nature of the application workload. This awareness delivered by HCR will help guarantee better response times and reduce the need to provision excessive hardware.
- Microservices patterns in many modernization projects are suffering from the fact there is no comprehensive platform-wide perspective. Some platforms only optimize the cloud-native aspect and ignore the rest. Even cloud-native platforms are being complemented with a service mesh layer to fix many of the latency and scalability issues.
- CIOs are misled into thinking the private cloud is part of their problem. In fact, many that have transitioned to the public cloud are now re-evaluating their application platform strategy. In many cases, excessive hardware provisioning is causing a migration from public to private cloud, where the lack of understanding and abstracting the nature of the application platforms is core to this problem.
- Abstracting the application platform is becoming a critical pattern in the industry. A simple rudimentary Kubernetes cluster does not constitute an application platform.
- Abstracting new personas for Application Platform Architect (APA) and Site Reliability Engineer (SRE), and providing SREaaS across our products long-term, will help customers make this transition within their own private and public clouds.
- Computer language runtimes were designed to optimize within one runtime for any deep method/function call stack, often making the most sought-after areas of the code more “hot” or readily available, all within one memory space. However, microservices and Functions-as-a-Service (FaaS) paradigms are making applications more highly distributed, thus less likely to fully leverage such optimization mechanisms. Specifically, these optimizations are no longer as effective as they have no notion of the distributed nature of the application. Essentially the network must now provide a layer that behaves like a distributed runtime compiler constantly optimizing across the network of inter-communicating services. We can call the first rudimentary version of this layer the service mesh.

- We should build application platforms, not only generic infrastructure layers. When we coined SDDC, what we really meant was a Software Defined Application Platform (SDAP), because almost all generic hardware designs are taken by our customers and customized for a set of application platforms. We should alleviate our customers from this burden and provide blueprints that certify our various products across specific application platform use cases.

2. Technology Overview

2.1 Overview

We often meet customers that have migrated to the public cloud only to later determine that some of their critical legacy application patterns have transitioned to a public cloud implementation, and they are now paying higher costs due to this design flaw. Regardless of cloud location, what really matters is how well you have abstracted the application platform nature of your enterprise workloads. If you don't understand your application workloads in terms of scalability, performance, reliability, security, and overall management, then you are simply shifting the problem from one cloud to another.

From our perspective as we interact with customers, we continue to see customers struggle with building good application platforms. Often various challenges exist with stability, deployment, and scalability—certainly more than fifty percent of customers we interact with have some sort of application platform challenge. More so, we continue to see that, whether in a private or public cloud, customers are provisioning more hardware than what is required—in some cases twice what is needed. So why is this happening? Movement after movement has attempted to solve this phenomenon of excessive hardware provisioning in the last two decades, but we still find ourselves in the same old situation where an application fails to perform and the old response of throwing hardware at the problem is used.

IT practitioners are bringing their old habits to new problems. The key to this problem is deeply rooted in the knowledge gap that exists between development and operations organizations. In this paper, we talk about the notion of the **application platform** and its teachings to close the gap that exists between developers and infrastructure architects. At the most fundamental level you can think of application platforms as an abstraction of three major parts: **1) application code logic; 2) application runtime where the code runs; and 3) infrastructure abstractions such as CaaS, K8s, and fundamental IaaS.**

The current cloud-native movement can be viewed as an example of application platform thinking. For example, cloud-native practices are about developers marching into infrastructure practices, using infrastructure-as-code techniques to fully automate current manual steps of provisioning the Infrastructure-as-a-Service (IaaS), and Container-as-a-Service (CaaS). No doubt, CaaS is a big part of this as a key abstraction. The infrastructure team can view this as a threat or embrace it as a drive toward the next level of maturity. From VMware's perspective, we must be at the forefront of this shift and build products for this new persona that is forming in the industry.

Application Platforms are more than just the cloud-native movement. It's a school of thought on how to run practical enterprise application platforms for the cloud era, inclusive of all application types that exist in a typical enterprise. Cloud-native advocates often lead with "you have to refactor your code," but there are many other ways to modernize without having to refactor code; for example refactoring the application runtime that is encapsulating the application code. You can have a multipronged application modernization strategy where, in some cases, it makes sense to re-write your

applications, whereas in other cases it may be cost prohibitive. This is especially true in cases where you attempt to migrate one perceived isolated and well-contained application, then quickly find there is no such thing in the enterprise.

In fact, many applications will have dependencies on other components, leading to the phenomenon we like to call “The Root Extraction Factor” (TREF) of typical enterprise grade applications, where applications grow dependencies organically over a period within a wide spectrum of the organization—ultimately complicating the public cloud migration strategy. These dependencies are often service-to-service integrations, but are also data integrations, where separate databases are interconnected. This creates an anchor for any migration strategy and often complicates it.

In some cases, you can effectively refactor—or build from the ground up—which is essentially what all other cloud-native vendors promote. In other cases you can more intelligently scale your application platforms with techniques to tune the application runtimes to better utilize the compute space. We know the majority of applications in a practical enterprise that can’t afford to be re-written fall into this last category of tuning application runtimes. This latter use case can be unique to VMware because we effectively have all the information from compute, network, and storage to deliver on this promise of optimizing the application platform without code change.

This has been our core strength since inception, and our customers have grown accustomed to the fact they can deploy on vSphere without any code change. The notion of being able to transform the application runtime without needing to make code changes, and building intelligent control plane layers that can deliver such features, also helps with current drawbacks of microservice implementations. Many microservice implementations suffer from extended latency issues as they massively scale out across the network, switching from what used to be an in-memory call within one monolithic application to a distributed call across the network. We can think of this special control plane as a multi-cloud runtime that is application aware; we like to call it HCR. More on this later, but first let’s understand a little more about cloud-native movement and its close affinity to microservices.

From early inceptions of the cloud-native movement there has been a drive toward microservices-based architectures, with advocacy for the ability to rapidly deploy code. (You will hear references to “code velocity,” making development teams independent from each other and reducing the old complexity of assembling one big monolithic code deployment unit.) The container movement was also adopted during this phase. But containers’ ability to abstract and package application runtimes alone did not solve the scalability and performance issues.

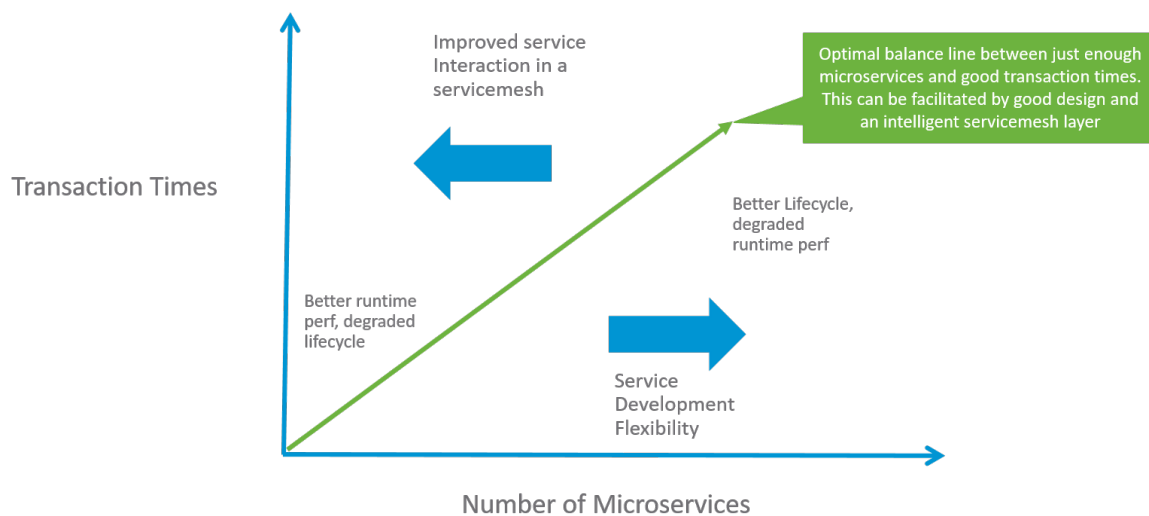
Next came the notion of dealing with container scheduling with Kubernetes and others. But container scheduling has no ability to look and understand application runtimes. Consequently, these systems continue to suffer from scalability and performance challenges. Essentially, with microservices what used to be an in-memory call is now several network-hops away, with many implementations overloading the load balancer due to new traffic chatter on the application platform.

This is an industry dilemma that is giving rise to yet another layer that is perceived to handle the ability to minimize the cost of the network hop between microservices calls. Projects like Linkerd (<https://linkerd.io/>) and Istio (<https://istio.io/>), referred to as service mesh layers, are layers that VMware can easily implement due to the ample visibility we have within the application platform, across three vectors of compute, network and storage. New paradigms cause new shifts in complexity. Hence, microservices introduce excessive chatter on the network, giving rise to the concept of service mesh, a load balancer control plane layer for inter-microservices communication, providing predictable Quality-of-Service (QoS) across communication between microservices.

2.2 Microservices Primer

Before we proceed further with this paper, it is important to understand some of the fundamental concepts behind a microservice architecture. There are many rules, but at the core of a microservice is business logic that has a well-published public interface that clarifies its usage. Microservice definitions are not dictated by number of lines of code. They are governed by the domain knowledge of your business, the lifecycle of the service, how many times it changes, and how to minimize the impact of change to other services in the application platform. The mapping between your business domains to microservices is a definitive exercise that needs to happen when designing your cloud-native application platform. This exercise would speed the rate at which business logic can be introduced into the application platform without massive disruption to other running microservices. Some might refer to this phenomenon as “developer velocity,” or “code velocity,” where it is all about rapid and fast deployment of new business logic. Others refer to it as “deferred integration”, because services discover and call each other (“integrate”) when the calling development team is ready, there is no developer-to-developer communication contention over having to wait to integrate code to release. Figure 1 shows a summary between the two design vectors that are perpendicular to each other, that of lifecycle flexibility vs runtime performance. For a further look at microservices, refer to Appendix 1 – Brief Look at Microservices Structure.

Figure 1 - Microservices Architecture Impedance between Lifecycle and Usage



3. VMware Perspective

3.1 Cloud Native Application Platforms Perception in the Enterprise – the Beginnings of Application Enabled Infrastructure Movement

The first trend we saw was Cloud Native Applications (CNA) go through its maturity curve, starting from a theoretical and impractical standing of “everything must be stateless,” moving to “stateless meets stateful” somewhere in the middle to make practical enterprise-wide application platforms. There is varied opinion with regard to this, from the early days of pure cloud-native where certain evangelists touted stateless-only CNA, to allowing for certain stateful services to interact with stateless services where the CNA platform is the norm.

Essentially, we find CNA has entered its practical enterprise use cases phase. This transition occurred over a period of three years. Clearly, the early evangelists were experimenting with various models, but then got bounded by enterprise realities. Today it is quite common to talk about stateless and stateful services within a CNA platform, and this is undoubtedly the most practical approach.

The second trend was early CNA evangelists advocating for everything in the enterprise to be re-coded to adhere to strict CNA. Some evangelists even advocated microservices everywhere as an absolute must. However, this had limited practical use in the enterprise, as there is a real cost associated with being able to re-code certain parts of a system. Therefore, the trend matured to “re-write certain parts as CNA, and let the other existing parts take advantage of the modernized platforms where possible,” as opposed to the novel early days of CNA message of “re-write everything.” This is a dual story of “re-write certain parts as CNA, and let the other existing parts take advantage of the modernized platforms where possible.”

Many in the enterprise are starting to embrace this stance. This approach helps drive an organic growth where eventually migrating everything to a CNA platform becomes a possibility. A much more phased adoption-driven approach is preferable, rather than migrating all at once.

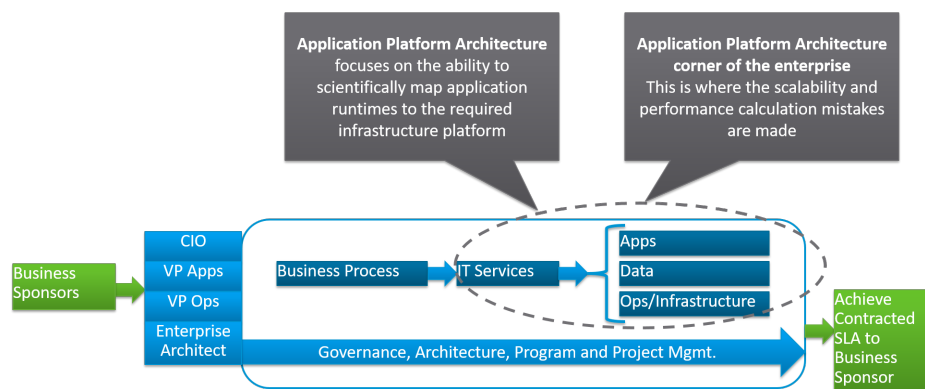
The other aspect of CNA in the industry, from our perspective, is that we continue to see customers struggle with building great application platforms. Often various challenges exist with stability, deployment, and difficult scalability use cases. Certainly more than fifty percent of customers we interact with have some sort of application platform challenge. More so, we continue to see in both private and public cloud, customers are provisioning more hardware than is required—in some cases 2.x to 3.x—more hardware provisioned than what’s needed. Why is this happening?

Numerous movements have attempted to solve this phenomenon of excessive hardware provisioning in the last two decades, but we still find ourselves in the same old situation: an application fails to perform and the response of throwing hardware at the problem continues to happen. IT practitioners are bringing their old habits to new problems. However, ***the key to this problem is deeply rooted in the knowledge gap that exists between development and operations organizations.***

In Figure 2, we show a typical organization with business sponsors interacting with leaders of the technology platform, and eventually mapping across the entire organization to deliver a specific application service. For example, a request comes in from the business sponsors; the technology leadership will do an initial assessment for feasibility, and then hand it over to the team that will do the build-out. Eventually these requested business processes map to some IT services. The IT services, in turn, map to application components, data, operations process, and infrastructure needed. ***It is in this last step of mapping application components and their runtime to Ops/infra where most of the miscommunication happens. This is the main area in all of IT where the problem of “50% of apps missing their SLA and 2.x more hardware is provisioned” happens.***

It is an area where developers speak one language and the operations team hears another, and vice versa. It is a practice seeded in the old school of IT and this needs to change. We find the change can happen via the cloud-native movement where we see platform architects being able to speak a common language between developers and Ops. This is where we start to refer to these new kinds of engineers, as platform architect/application platform architects—like SRE.

Figure 2 - Organizations of a typical enterprise applications platform group



To understand the background where this mapping breaks down, it is important to acknowledge the various artifacts involved. The process begins with developers writing code then packaging it, as shown in Figure 3. The package example in this case is “.war” (web archive), so it’s likely the application runtime shown here is an application server like Tomcat. Then the application runtimes in turn are deployed on virtual machines. ***These application runtimes play a critical role in the execution and scalability of an application.*** Most will typically have a configuration file or set of properties that allow you to scale the number of threads, the amount of memory allocated, and the garbage collection algorithm. Most developers will trivialize the importance of the application runtime and their scalability properties. In fact, in many cases, they commoditize it and think that at literally zero cost you can continue to scale-out instances without penalty. Of course, there is a penalty.

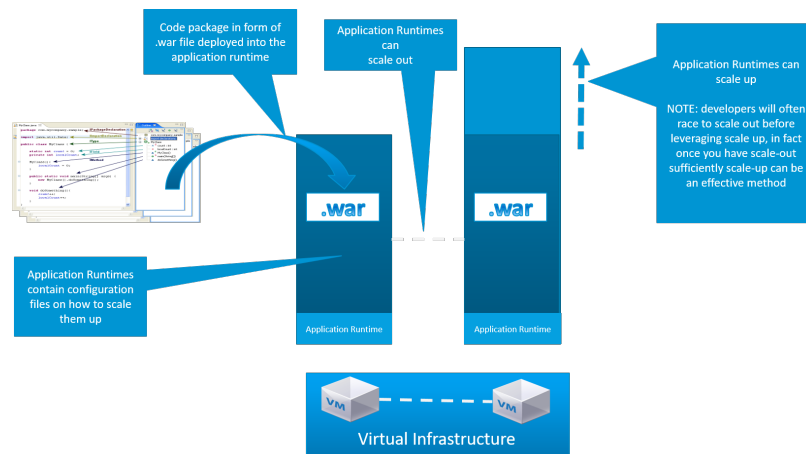
On the other hand, operations engineers do not know these application runtimes exist, or know very little about them. Therefore, we have a situation where developers are half-owning the application runtime and operations engineers are responsible for their SLA.

Indeed, this is not a good thing. This miscommunication is what is causing half the world's applications to continue to suffer in terms of scalability and performance, even after 2.x more hardware is being provisioned than what is needed.

As you can see in Figure 3, you can both scale-up and scale-out these processes ("processes" here is synonymous with application runtime). Obviously, scale-out requires the additional licensing and management cost, while scale-up can be a cost-effective solution to provide more execution capacity for minimal overhead. We highlight this because we are talking about application runtime configurations and their importance.

Many times we find Ops will add more resources to a virtual machine (VM) but the application runtime properties are not changed to reflect the addition of new compute resources. In this case, you have a situation where there is ample compute space at the VM level but the application runtime is not leveraging it. This is where compute space bloat happens (essentially the problem of 2.x more hardware being provisioned). The poor performance is not from the compute space bloat, but from the fact the application runtime was not scaled-up through changing its own configurations to indicate it can have more threads and more memory. Many systems will further respond to this poor performance by excessively scaling out the instances, leading to severely underutilized infrastructure and poor application performance.

Figure 3 - Development Artifacts and the Application Runtime

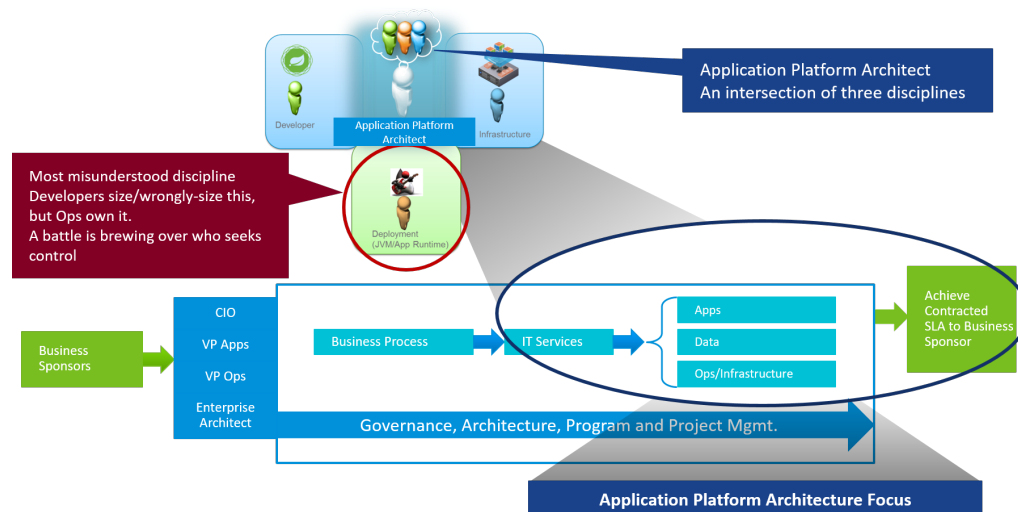


Clearly, from the above discussion of hardware bloat, we are on an unsustainable course. CNA is trying to address this challenge by helping to build new types of engineers that are multi-disciplinary: engineers that close the gap between developer and operations knowledge of an application platform. First, let us inspect more closely what this new type of engineer

3.2 The Era of Application Platforms is Born

In Figure 4, the IT enterprise is superimposed with the notion of an **Application Platform Architect (APA)**. An APA is someone who has done extensive software development, has worked in how to scale-up and scale-out the application runtime, and has a solid understanding of infrastructure platforms. The APA is someone who has worked critically across the scalability and performance concerns of the application and associated application platform.

Figure 4 - Application Platform Architects a New Persona for CNA Era



As an example to better understand what an APA is, Company X is in the business of selling a trading platform as a service. In their latest iteration they have multiple components: a messaging system, custom distributed Java-based services, and a critical GemFire data platform (in-memory database). The core of this platform is the in-memory database. GemFire is a Java-based distributed database, in this case running on VMware vSphere. Hence, you need someone with a decent understanding of these three skills, or at least two of the three. When the platform was first released, developers were always involved during the operation of this platform. This placed a huge strain on the development efforts; however, later they managed to train and hire APAs, essentially re-skilling their existing DevOps team. Their existing DevOps team had great skills in automated deployments of application platforms, but lacked the skillset in terms of scalability and reliability techniques. In a few cases, we trained their existing engineers to become good APAs. But in other cases, they could hire SREs as the closest persona to the notion of what an APA does. A popular saying in that team was "...we put SRE back into DevOps." This is to indicate that the founding principles of DevOps were SRE, but over the years DevOps digressed into meaning anything and everything for everyone, and more about CI/CD rather than its core founding principle of reliability. Many DevOps engineers they interviewed did not have scalability and performance backgrounds like SREs—an artifact of where the industry had influenced DevOps in the direction of certain vendors with CI/CD tools.

Here is a list of what the APAs/SREs at Company X do on a day-to-day basis:

1. They own the performance and reliability of the trading platform.
2. They are paired with multi-discipline senior engineers from development and infrastructure. They form close relationships with them and learn what they need to learn daily. This is important because often they find themselves chasing multiple organizations and disciplines to resolve a specific challenge with the platform.
3. The trading platform is made of 40 VMs and various software components within them. As a result, they have fully automated the creation and tearing down of these environments. They have many environments like these, so the SREs fully automated the creation and tearing down of the trading platform. This was the minimum necessary to support their agile high-iteration development shop with multiple requests per day for creating such a trading platform.
4. They look at the most feasible way to scale the platform, whether it is a certain amount of healthy scale-out or a combination of scale-up and scale-out strategy. Feasible scalability that meets adequate performance is a daily pursuit for them, because they profit as a company by charging a fee for transactions on their volume-driven trading platform.
5. Their platform has high stringency on reliability and predictable runtime execution of the transactions, regardless of volume. In fact, if they miss the agreed-upon transaction response time, they pay their customers a penalty (the stock exchanges of the world that use this trading platform are their customers).
6. If there is a performance challenge, they are responsible for investigating it from the code and infrastructure platform perspective. Often, with the ability to troubleshoot by taking thread dumps, SAR reports, heap dumps, and many other metrics, they can narrow down the cause of the issue and organize multi-faceted team meetings to address the concern, usually in a war room style if needed. The SRE team would own most production problems, but if they need help will organize wider teams to be involved.
7. They work with software engineers in architecting the runtime for new applications, helping them understand the capabilities of the platform, as well as what is feasible and what is not from a scale-up and scale-out perspective. For example, which components will need affinity or anti-affinity rules. (Affinity means some components need to run next to each other and other components that are redundant pairs must not run on the same VM/physical host; i.e. anti affinity.)
8. They create various platform and infrastructure abstraction layers and infrastructure/platform APIs to help developers with writing smart cloud-native era applications. For example, applications that can understand and remedy themselves if there was a certain platform failure event.
9. They create very advanced metrics that can show the transactional volume versus cost of each transaction, and continue to improve the application platform as a

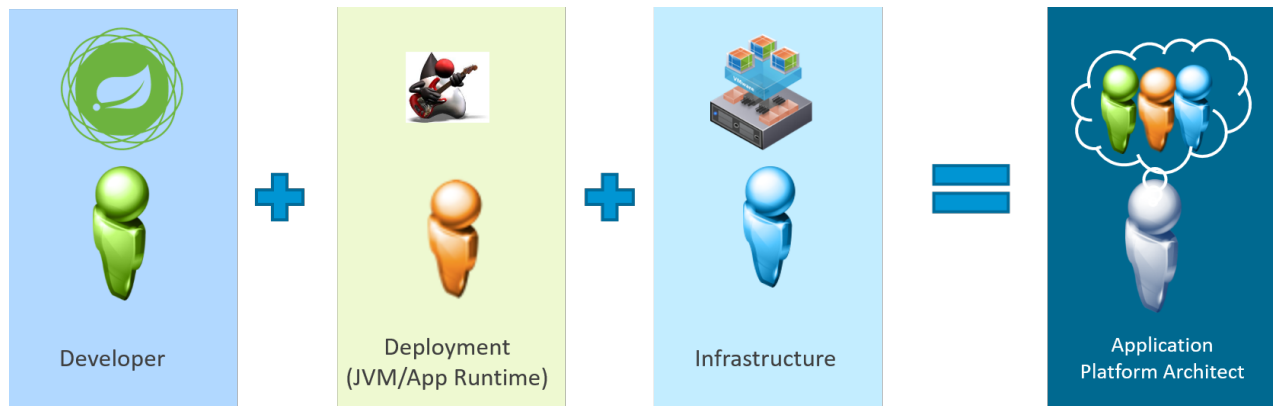
result. They also have further metrics on hours spent on administration, chasing and fixing reliability issues. (Trading platform business analysts use this rich platform data to help further analyze cost vs. profit of the business.)

Let's highlight further what the key skills are in each of these categories: in the case of a development background, it is critical that the APAs have had a significant software engineering background at some point in their career, have written code for major applications/services, and have become super-versed in at least one of the modern computer languages such as Java, JavaScript, Node.js, Ruby, Scala, Python, or Go. It is also critical that they have not only worked as pure software engineers developing narrow product features, but have worked in an area where multiple components/platform layers are involved—typically ones found in every enterprise IT shop. Systems they've worked on should include load balancers, web servers, application servers, middleware, messaging systems, in-memory databases, traditional relational databases, operating systems, containers, and virtualization. Most critically, they have been able to deal with common production reliability and scalability issues as software engineers. The best software engineers are ones that write and learn how to best run and scale their code in production environments.

3.2.1 Application Platform Architect in a Nutshell

In Figure 5, we show the three main categories of skill sets needed for the new role of Application Platform Architect: development background, deployment, a background in having deployed and scaled applications in production, and a good grounding in the infrastructures stacks on which the application platform relies.

Figure 5 - Application Platform Architect Combines Three Skillsets



In the case of a Deployment/App runtime background, this is a critical piece of the puzzle, and the hardest skill set to find. In fact, most organizations don't even know that this needs to exist as a skill. Over the past decade at VMware we have developed a specialized technical workshop for coaching teams on application platform thinking. We typically run this workshop with a mixed audience of SREs, VMware Architects, Platforms/Infra architects, developers, and application owners, all at the same time. This is by design, because we want to identify the gaps in interpretations in each layer and address them

holistically in real time. The workshop takes on the agenda of material that we present in this paper, but also has a deep-dive interactive session where we review an existing application platform and make design improvement recommendations. ***When the collective audience sees how our cross-application developer, application runtime, and infrastructure discussion is conducted to lead to a better design, most customers walk away saying that they would like to build an Application Platforms team within their organization.***

3.3 What Do CIOs Mean When They Say, “We want to be out of the infrastructure business?”

It’s not that CIOs want to be out of the infrastructure business, but primarily they want to avoid the earlier mentioned problem of missing the SLA half the time and spending 2.x on hardware. It’s an unsustainable model they want to resolve. ***If you do not fully understand the scalability paradigm of your application, then going from a private cloud implementation to a public cloud implementation will not solve the underlying infrastructure bloat due to a poor scalability pattern.*** Note: scalability pattern refers to the understanding of when to scale-up and when to scale-out, and by how many instances you need to scale to make the application platform meet the SLA.

Once the above scalability patterns have been well understood, the next problem to tackle is the CI/CD, which allows for rapid testing and integration, and continues to build/deployment of the application platform. You will find that developers will refer to this as “developer velocity.” By this, we mean the ability to spin up a platform quickly and with predictable accuracy. Of course, the last element would be offering a development team autonomous agility to deliver services without being slowed down and/or burdened by a slower team. Hence, the rise of Microservices Architecture.

CIOs and CTOs of application platforms that have achieved developer velocity have moved the importance of their platform from being tier-1 to tier-0, with tier-0 meaning it is critical to the business, where without it the business would fail. This type of developer velocity, driven by the cloud-native movement, is moving CIOs from tactical CIOs that report to the CFO, to strategic CIOs that report to the CEO of their company.

4. Recommendations

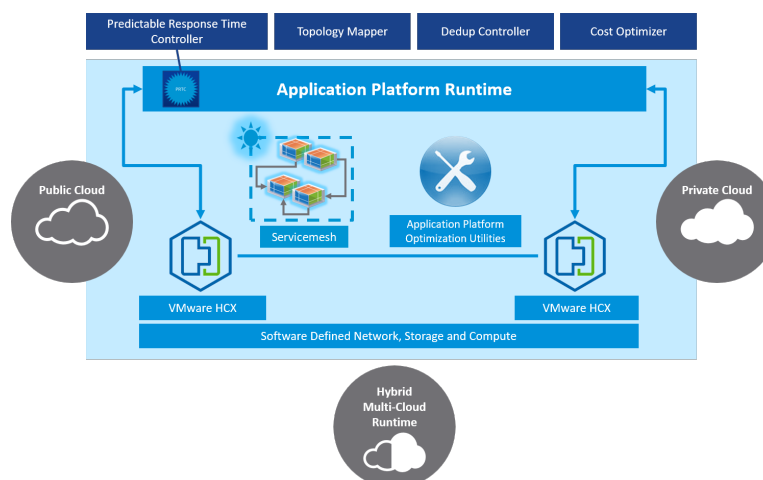
When we talked about SDDC a few years back, we were really saying that a Software Defined Application Platform (SDAP) is a specialized multi-cloud runtime common layer that is application-workload aware. This is a layer that understands application workload behavior, the cost of its placement on various clouds, the cost of its scale on a private or public cloud, and the needs of its data affinity or proximity to other services. It’s an intelligent control plane that is application-workload aware, which we believe is the essence of true HCR.

Keep in mind, if one doesn’t deliver on all of these vectors, they are simply building generic infrastructure layers that are blind to what application workloads need. To every set of generic hardware configured, it ends up being customized to suit a customer’s application workload, and while there can be many such instances of workloads, mostly

they can be categorized into a dozen use cases. There is a huge opportunity to leverage and take advantage of the phenomenon of common multi-cloud runtimes, where applications are starting to look more like networks, and as such network layers need to build specialized application-workload aware functionality to deliver value.

We are seeing the drive toward building specialized multi-cloud application platforms, but these platforms cannot simply be split in an ad-hoc fashion across two separate clouds. As one customer said, “putting a load balancer in front of a private and a public cloud does not constitute a hybrid/multi-cloud system. I can do that all day myself. What I need is a common runtime between the two.” Hence, one must think of a common stretched layer between these clouds. This common layer could be referred to as the HCR, which would be the main engine of the Application Platform runtime, with service mesh capabilities, application workload behavior simulation utilities, [VMware HCX¹](https://hcx.vmware.com)-like workload mobility services, software defined compute, network and storage, with distributed trace capabilities using our management products such as [Wavefront²](https://www.wavefront.com/) and others. One can think of these layers as SRE-as-a-Service (SREaaS), where there is an effort to fully automate anything that an SRE does to keep an application platform optimized, providing constant real-time analytics data, and making intelligent placement and optimization decisions. It may be that SREs have a manual job today, but their current job will be fully automated by these specialized layers—a layer like an HCR—an application platform runtime with service mesh at the core of its foundation, Figure 6. In this figure we also show some specialized controllers such as Predictable Response Time Controller (PRTC) for optimizing response time at specific scale across services, Topology Mapper to determine the call graph between services including placement on compute space, Dedup Controller that removes compute space fragmentation

Figure 6 - The HCR a Core Runtime for an Application Platform

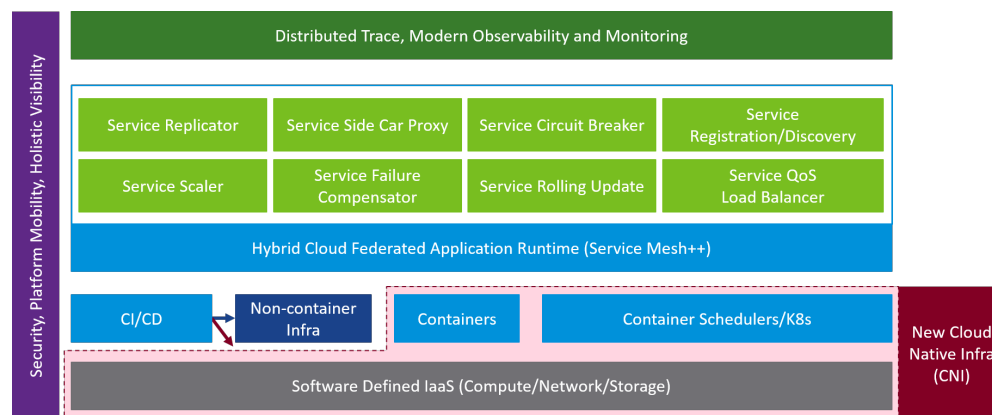


¹ <https://hcx.vmware.com>

² <https://www.wavefront.com/>

In Figure 7, we show a functional diagram for an application platform containing a Cloud Native Infrastructure (CNI) layer, CI/CD, service mesh, security and policy management, and Distributed Trace capabilities that will help drive SREaaS initiatives. This application runtime also applies to non cloud-native products for the enterprise, where we think a truly federated service mesh layer can marshal between cloud-native and non cloud-native services.

Figure 7 - Application Platforms Functional Diagram



4.1 HCR with Service mesh

Figure 7 shows a service mesh layer with key functions. (Note: in Figure 6 we also show a service mesh as a core function of HCR.) Some of these functions are well evangelized in the industry, and others are based on our empirical findings in terms of what we think a service mesh should remedy in a typical microservices cloud-native application platform. The following is a list of functions of a service mesh:

Service QoS Load Balancer: This is at the core of any service mesh implementation. With so many microservices implementations suffering from extended response times as the service mesh grows, an intelligent workload/service-aware load balancer is needed to help optimize routes and reduce the impact of latency. This can be critical in functionality for achieving the intended behavior with using “Predictable Response Times Across Service Routes” and “Service to Service Co-location and Service vMotion” patterns. At a minimum, the load balancer would provide traffic optimization for HTTP, gRPC, WebSocket, and TCP.

Service Sidecar Proxy: The reference to “sidecar” comes from the fact that it is a supporting service that attaches to a main service context. Just like a motorbike’s sidecar, it attaches to extend the motorbike’s capabilities. To create this notion of a highly intelligent workload/service-aware control layer such as a service mesh, one must be able to inject lots of service mesh introspection and optimization information, but in a non-hardcoded manner. Often developers are offered libraries that can enhance the performance of their service, but with the heavy requirement of hardcoding certain callouts within their code, often littering the primary business logic code with plumbing code. On

the other hand, the sidecar proxy concept alleviates this burden by intercepting calls and adding the needed optimization information to the call, all at runtime without any specific hardcoded code callouts.

Service mesh is gaining traction rapidly in the market because there are many microservice implementations that are failing to meet their performance, scale, and reliability objectives. It is important to focus on clear use case areas that would allow microservices implementers to gain control of their SLAs. Below we focus on new use cases, skipping over the common use cases that have already been written about such as load balancing, security, scale controller, redundancy, and sidecar proxy.

Service Replicator: The service mesh can easily create various availability zones and based on the injected configuration from user, can maintain anti affinity of services by replicating them to other zones. In most cases the replication can be simple. In others the user may provide service dependencies and would require a sub-branch of the service mesh to be replicated as a result. The main objective is that the replication behavior can be abstracted from the main code and housed in the metadata configuration the service mesh uses to apply the replication logic.

Service Circuit Breaker: Just like with electrical circuits, when a certain current level is reached a breaker is tripped and no more current can flow. In similar fashion, a microservice can be protected by a breaker pattern that encapsulates it with a service call breaker. The encapsulating breaker pattern detects any potential deterioration of SLA and breaks all calls to the base microservice by throwing an error, where an error handler in the system can have additional logic to act on the situation. In most cases rudimentary implementations simply cut off ensuing calls to the service. But in a more advanced functionality, they can have the ability to shut down the microservices and provide cascade shutdown behavior. This means if one service is shut down, its neighboring or related services are shut down as well. Finagle (<https://twitter.github.io/finagle/>) and Hystrix (<https://github.com/Netflix/Hystrix>) were some of the early examples of this pattern.

Service Discovery and Registration: In any large service deployment, as a service mesh grows, knowing which service is in the mesh is paramount; but equally important is the ability to have a well-known and entrusted service registration mechanism, where certain policies can be applied.

Service Scaler: This is a service mesh sidecar process that sits and listens to certain service execution metrics, then determines which services need to be scaled. At first this appears to be a simple problem but, in fact, it requires the ability to sample execution data over time, and heuristically apply changes and learns from them to improve the decision in the next sample. Many of the use cases at the service mesh layer would critically depend on Service Scaler functionality, particularly the following use cases: Achieving Predictable Response Times Across Service Routes, Service to Service Co-location and Service vMotion, and Service to Service Co-location and Service vMotion.

Service Failure Compensator: There are a lot of manual steps that a typical SRE/APA does to handle a myriad of day-to-day service failures. SREs can write many such handlers to plug into the service mesh to be called back by the compensator handler when a certain event happens. The essence of the mesh is that it is partly developed by a

vendor with a certain baseline, but then it grows quickly by many plugins/compensation handlers that the users plug in. This is where the huge value of the service mesh comes in, where many manual steps are captured and automated away.

Service Rolling Update: This function provides the ability to update service instances without causing interruption to current midflight transactions.

5. Conclusion

More developers are getting into the business of building specialized application platforms that are IaaS capable, creating purpose-built platforms to suit their workload needs. As developers are building new cloud-native platforms, they are quickly realizing that their systems aren't complete and lack true scale and performance, because a large part of the enterprise is not running on their cloud-native platforms. This is where VMware can help. Not only can VMware develop [cloud-native platforms](https://blogs.vmware.com/cloudnative/products/)³, it can also develop a holistic platform that encompasses all enterprise applications, cloud-native and others, with intelligent awareness of the nature of the application workload. This can take all the difficult performance and scalability concerns away from the developer and provide them purpose-built and highly configurable application runtime, such as HCR.

³ <https://blogs.vmware.com/cloudnative/products/>

6. Appendix 1 – Brief Look at Microservices Structure

For example, in the XYZcars.com (you can get more details on XYZCars use case [here](#)) case that we are about to inspect, the company maintains the Account information and associated business logic within a service (not just Account entity manipulation, but business logic for account offerings), as shown in Figure 8, it is a business domain function that maps to several account entities within the organization. Additionally, the mapping between account information and its business function co-relation to other microservices is maintained as metadata in the application platform to help drive discussions about impact of change. Every microservice will have a service name, service port, public interface, and configuration/metadata.

NOTE: Microservices in an organization are essentially about splitting the monolithic business functionality into smaller pieces of self-contained logic, well-encapsulated so that changes don't cause ripple effect, this in turn speeds up the development velocity, which is what microservices are all about. Of course, ***this inevitably means that what use to be an in-memory call from one business function to another, it is now a network hop away. Naturally, this implies performance will be impacted, and often we find with microservices implementations there is an explosion of traffic that the load balancer wasn't used to handling***, and this is where the shift towards a new way of thinking is needed. We often say that complexity of systems has shifted with Microservices Architecture (MSA), from N number of in-memory calls to N number of network hops, naturally the control of this complexity is giving rise to specialized layers, such as Platform-as-a-Service (PaaS), application platforms, and Service Mesh, where the platform can act to co-locate, optimize routing (minimize the impact of excessive network hops), service registry, service discovery, auto-scale, rolling-restart, and deliver various other QoS functionality. Now of course, this also implies understanding the organizational structure of the enterprise and designing services that promote the disentanglement of complex communications, approval paths, cross team testing, and complex cross team deployments.

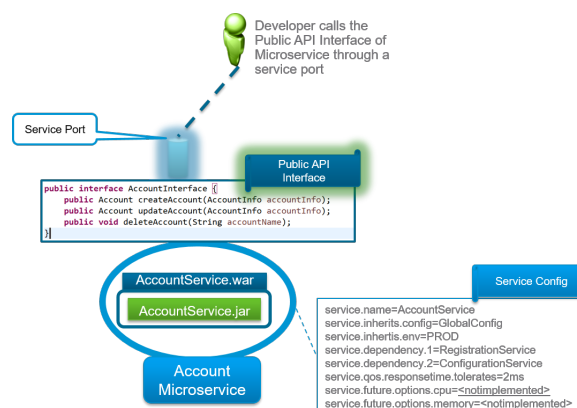
Main Attributes of Account Microservice are shown below, and illustrated in Figure 8 :

- **Service name** is AccountService, which encapsulates its business logic in a AccountService.jar file that in turn is encapsulated within a AccountService.war file. The war file is used to service it into a servlet container (like Tomcat for example), application service, or Spring Boot. This will allow it to be called via REST call.
- **Service Port**, this is the port the client will use to call the Account Service
- **Public Interface**, an interface is a contract that defines what APIs are available on the microservice, so that any client calling this can adhere to the contract. Contracts help facilitate rapid integration between clients and services, or between services. Now we are using the term “client” but a client could be a browser, a front-end, a middleware layer, or any other microservices calling into the account microservice. Typically, it will be the middleware layer facilitating communication to other calling systems. But the sheer fact that the service has a well-defined contract, other services that automatically discover the Account Service can

immediately understand how to use the Microservice without much regard for its internal details. This also binds the Service to adhere to a contract and ideally prevent breaking it in the future, so the calling service that uses the interface can have relative confidence that the contract won't be broken. Sometimes due to poor initial design a service interface can be changed and hence can break clients or other services that are calling the AccountService, but at least a quick refactoring of the code can discover which clients are calling the interface. Metadata and configuration information can play a big role in minimizing impact here.

- **NOTE:** if you are retrofitting your code from old Service Oriented Architecture (SOA) and Simple Object Access Protocol (SOAP) based constructs, do not proxy REST calls to SOAP to avoid minimizing impact on Client code, it is best to go through the pain of changing the service code layer to using REST natively/directly and remove all things “SOAP-y” from your code base.
- **Service Configuration**, this is where certain dynamic runtime properties can be placed, for example which environment the service will run in, production, or test, or what configurations it inherits from the Configuration Service, like the persistent database service, what dependencies it has on other services from re-deployment and lifecycle perspective. Some other features would be around QoS where the service configuration outlines what service execution time it is willing to tolerate, this in turn is registered with the application platform and then if exceeded an event handler is triggered to do appropriate remediation. Other advanced future options that can be thought about is how much CPU and Memory, other compute resources the service needs, but mostly people tend to ignore the usage of these properties and defer them to either container, or container scheduler.
- *The above attributes of a microservice are illustrated in Figure 8, shown is the AccountMicroservice hosting the business logic in AccountService.jar and then in turn this jar file is encapsulated within a war file to facilitate deployment in a servlet container that allows for REST communication, such as SpringBoot.*

Figure 8 - Main Attributes of a Microservice



7. Acknowledgement

The author would like to thank the following reviewers for their valuable feedback:

Greg Lavender, Senior VP and CTO Cloud Architecture OCTO

Pere Monclus, CTO NSBU

Tom Hite, Sr. Director of Professional Services Emerging Engineering

John Blake, Director Cloud Architecture OCTO

Roman Tarnavski, Principal Architect, APJ OCTO

Michael Francis, Principal Architect, PSO

Andrew Babakian, Principal SE

Mitesh Pancholy, Principal Architect, PSO

Niran Even Chen, Staff SE

Michael Gasch, Application Platform Architect OCTO (K8s Specialist)

Author: Emad Benjamin, Sr. Director and Chief Technologist of Application Platforms