z/labs | software defined mainframe®

White Paper

A Guide to Making Mainframe Applications Cloud-Native

Contents

Executive Summary

By Mark Cresswell
Chief Executive Officer,
LzLabs

For many industries the Rubicon of cloud provisioning, as the default deployment choice for commercial applications, has been crossed. Alternate deployment models now need far greater justification than does the cloud.

And the cloud is not just for new applications; many initiatives are underway to transform legacy applications (that currently run in a variety of traditional environments) such that they can take advantage of the cloud. Google's Anthos is a great example of such a program.

Google is but the tip of the iceberg. Other technology players in the cloud computing environment, are contributing software infrastructure along an open-source model, to support this exponentially growing enthusiasm for cloud deployment.

These players have formed an alliance, under the auspices of the Linux Foundation, known as the Cloud-Native Computing Foundation (CNCF). We discuss the CNCF in more detail in the next chapter, but at a high-level, it is a remarkably focused, and fit-for-purpose collection of well-supported technology and expertise designed to make all the benefits of cloud-computing realisable.

From increased development agility, to horizontal scalability, service-meshes, micro-services, and security; if you are looking to take advantage of the cloud, you'll need to become cloud-native.

However, even with all this innovation, the cloud still seems stubbornly out of reach for those legacy applications that run on mainframes. This paper explores what it means for an application to be cloud-native and discusses the friction points associated with making mainframe applications cloud-native. The paper also details how mainframe applications, when coupled with the correct virtualization and container models can not only be easily made to fit into a cloud environment but are actually very good candidates to be provisioned as cloud-native.

What are Cloud-Native Applications?

The term cloud-native is gaining traction as a way of encapsulating the many beneficial characteristics of applications which exploit the unique elements of cloud-computing. More than just a marketing term, cloud-native has a robust definition that, in large part, is promoted by the Cloud-Native Computing Foundation.

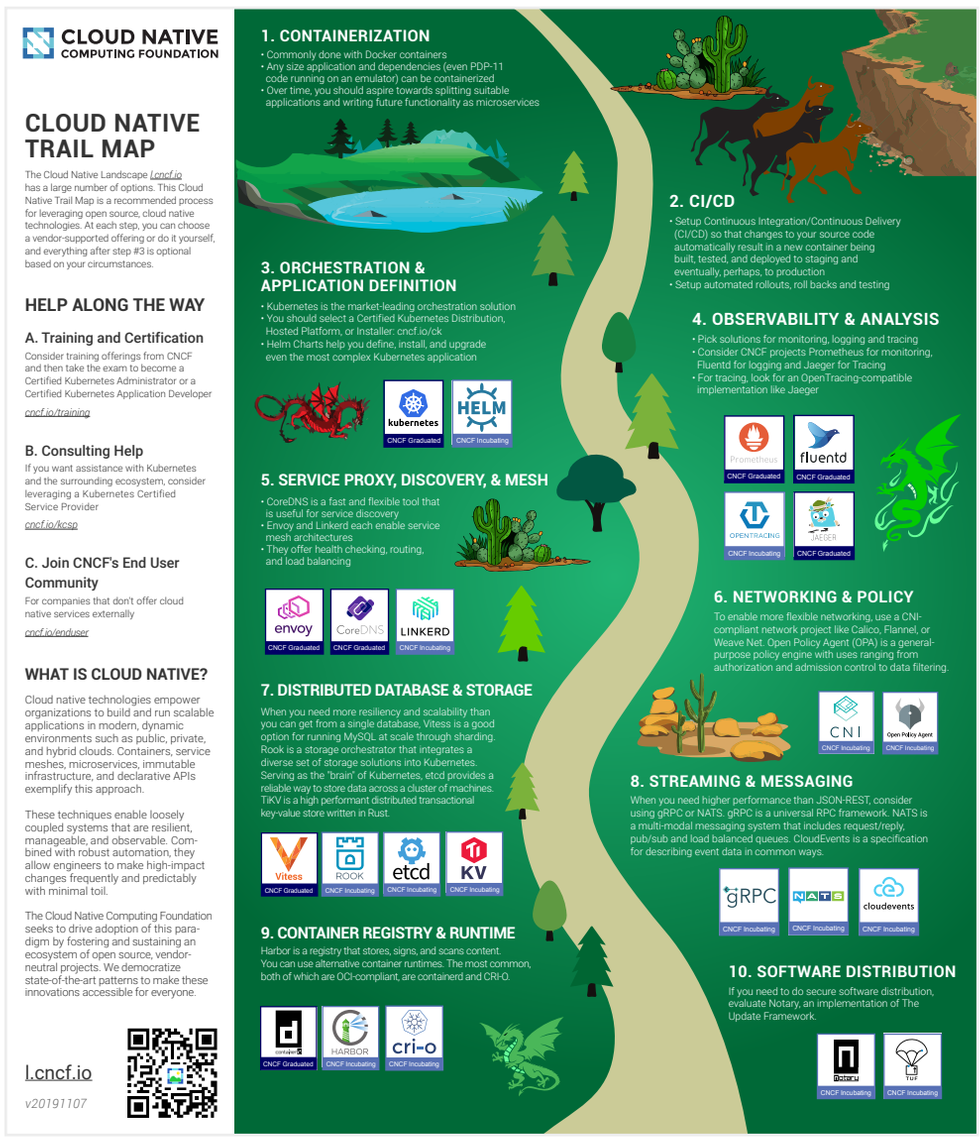
Cloud Native Computing Foundation

Launched by the Linux Foundation in 2015, the Cloud Native Computing Foundation (CNCF) is one of the fastest growing forums for the discussion, development and delivery of vendor-neutral, open-source, high-quality technology to accelerate the adoption of cloud computing using a cloud-native model. CNCF represents the groundswell of support for the utopian ideal that all the best applications are implemented using a cloud-native model.

Before discussing why mainframe applications are seen by many as being the antithesis of cloud-native, it's worth looking at what it means to be a cloud-native application.

Cloud-native is a separate and distinct notion to that of simply running in the cloud. Crucially, "cloud" in cloud-native has nothing to do with where the application runs. The term concerns itself with development and deployment technology that has its origins among the public-cloud vendors, but has now gone mainstream; regardless of the final deployment destination of the application. For an application to be cloud-native, it will exploit technologies, development patterns and methodologies rarely found in traditional, monolithic applications.

Cloud Native Trail Map



Source: Cloud Native Computing Foundation

Cloud-Native Application Characteristics

The core characteristics of cloud-native applications are as follows:

DevOps

This characteristic of cloud-native applications is listed first for a very good reason. The implementation of processes and technologies that fall under the umbrella term of DevOps is fundamental to cloud-native applications. Highly collaborative and agile development processes, with short development cycles of small incremental improvements, coupled with efficient and automated testing at scale, has been the impetus behind many cloud-native technologies. The so-called DevOps toolchain that facilitates these optimized development cycles feeds into other components that underpin cloud-native applications.

Continuous Integration & Continuous Deployment (CI/CD)

It's worth specifically calling out CI/CD as it relates to cloud-native applications in the context of DevOps. CI/CD is not a new concept, but in conjunction with other features of cloud computing, it has the power to fundamentally change the way organisations think about application roll-out. By developing small components of an application that can, upon the merge back into a source-code repository, be automatically integrated with other components and extensively tested, defects can be detected far more rapidly. Validating changes in this way ensures deployments proceed quickly and with the assurance of quality.

With smaller, independently developed and deployed components, organizations can employ novel methods of roll-out such as component-level A/B testing. New features can be "Canary Tested"[1] wherein they are made available in containers only to a select group of users.

A great deal more can be said about CI/CD, but for the purposes of this paper we can stop at the fact that it relies on the dynamic provisioning of compute resources to automatically integrate and extensively test the smallest of incremental developments. In this respect cloud computing has been a game-changer for CI/CD. For evidence of the impact to development velocity of a commitment to CI/CD, Amazon's deployment rate takes some beating.[2]

1 What is canary (canary test, canary development)? - TechTarget - https://whatis.techtarget.com/definition/canary-canary-testing

2 How Amazon handles a new software deployment every second - ZDNet - https://www.zdnet.com/article/how-amazon-handles-a-new-software-deployment-every-second/

Service Architecture

At their core, cloud-native applications are based on independently deployable components. In practice this means that, even though these components are part of a larger system, they can be upgraded and scaled independently of other components with which they interact. For this deployment flexibility to be plausible and valuable, the interfaces between the components of the applications must be stable, and the components must be materially smaller than the application as a whole.

These requirements have given rise to the term Micro-Services Architecture (MSA). In reality, MSA is an extension of both an API-driven model for development and the CI/CD idea that supports rapid development, testing and deployment of smaller application components. When implemented using containers and infrastructure software endorsed by the CNCF, the management overhead of handling many more independent components within an application is reduced.

Naturally Orchestrated and Containerized

A cloud-native application will not be tied to any hardware or software infrastructure, other than by choice. Using containers ensures this separation and also forms the basis of more advanced deployment options such as a server-less environment for applications.

Containerization, as a core component of cloud infrastructure, has been around for a long time in internet terms. As far back as 2014, Google was deploying thousands of containers per second.[3] As containers continue to proliferate in support of cloud-native application development, they need run-time "management". One element of this management requirement, often referred to as orchestration, has fallen to a core component of cloud-native infrastructure called Kubernetes. Kubernetes and its ecosystem of technologies and tools ensures that containers are correctly load-balanced across a cluster, and that the services they provide to other parts of the application are easy to find. Other capabilities such as restarting failed containers; ensuring that any application affinities are honoured; and storage management and authentication are tightly woven into the facilities Kubernetes provides to containers.

Containers also provide an excellent way of creating and deploying Microservices; the smaller units of application functionality described previously. Developers who exploit MSA leverage containers, among other reasons, to ensure they need not concern themselves with infrastructure issues, which leads perfectly into a discussion on APIs.

API Driven

The notion of APIs is not new, and indeed their use in modern development is now commonplace. But, in the context of cloud-native, APIs take on a special significance. APIs are used within a cloud-native application to decouple the provider of a service from its consumer in a way that cascades throughout the entire development and deployment process.

3 EVERYTHING at Google runs in a container - The Register - https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/

Stated differently, in cloud-native environments the implementation of the API is a question the developer of the consumer component need not answer. It is enough to know that its implementation exists somewhere within the cloud-infrastructure, and that it can be found using standard techniques. In many cases this may be no more significant than inclusion of dependencies within a container image, as provided by the DevOps Toolchain, or it may be a RESTful or gRPC service available via service discovery infrastructure facilitated by Kubernetes. This is in stark contrast to the inter-program interfaces of legacy implementations, which require programmers to statically link APIs to their applications and are typically tightly coupled and brittle.

An important collateral benefit of a commitment to robust APIs, is that it enables an organization to embrace "digital" more easily. APIs do not just enable components of an application to interact more easily; they can also be revenue-generating in their own right. The APIs made available to third-parties are driving digital initiatives that would be difficult to secure within monolithic applications. In a microservice-based cloud application, the cloud platform enables rich security policies and proxies to allow specific services to be safely exposed to external applications. This notion of driving revenue directly from APIs, is often referred to as the API economy. A thorough explanation of the trend is provided by a Forbes article from 2017.[4]

Stateless Applications

In pure architectural terms, cloud-native levels of separation can introduce complexity not found in monolithic stateful applications.

Of course, state must be maintained somewhere for most enterprise applications. However, cloud-native applications are designed to ensure that state is not inadvertently maintained within the application during its execution or shared at the application level, other than through those elements specifically identified as being stateful, such as Kubernetes StatefulSet deployments, databases and queues; and only then in ways that do not constrain other elements of cloud-native design. When appropriately managed, state provides significant benefits in terms of horizontal scaling, recovery and service architectures.

In pure architectural terms, cloud-native levels of separation can introduce complexity not found in monolithic stateful applications. However, open-source frameworks and tools, specifically those endorsed by organisations such as CNCF, have made such designs easier to implement and manage. The fundamentals of this management are discussed later.

Horizontally Scaled

One of the fundamental benefits of a cloud computing deployment model is on-demand scaling of resources to support workload growth or spikes. This scaling model tends to be horizontal in nature insofar as the limits are based on the number of machine images available to meet workload demand, rather than the resources that can be made available to any one machine image. To take advantage of this model requires the application to be implemented using many of the patterns and tools previously mentioned.

4 2017 Is Quickly Becoming The Year Of The API Economy - Forbes - https://www.forbes.com/sites/louiscolumbus/2017/01/29/2017-is-quickly-becoming-the-year-of-the-api-economy/

Logging, Tracing and Monitoring

For all the many advantages of cloud-native implementations, one of the challenges is diagnosing issues as they arise. With application components, and the transactions they support, loosely coupled and distributed across many machine images, identifying the root-cause of failures or degradation can be difficult. Having said that, technologies have emerged that go a long way towards providing the level and class of instrumentation required. Indeed, three of the top-level open-source projects within CNCF: Prometheus, Jaeger and Fluentd, are entirely focused on instrumenting cloud-native applications. The result of this commitment has been an explosion in support for these projects.

Why Has it Been Difficult for Mainframe Applications to Become Cloud-Native?

It is conventional wisdom that mainframe legacy applications cannot participate in the new world of cloud-native computing; so much so that it is rare for commentators to question why this is. In reality there are only a few reasons why it is difficult and, as we shall see later, none of these reasons are insurmountable. These items, detailed in the following paragraphs can be summarized as follows:

- Instruction-set Architecture
- Mainframe Software Infrastructure
- Development, Testing & CI/CD
- Monolithic Applications
- Cloud-native Infrastructure
- Logging, Tracing & Monitoring

Instruction-Set Architecture

The biggest barrier to mainframe legacy application participation in the cloud-native movement is the simple fact that the applications will not run on the underlying cloud hardware, without significant refactoring and recompilation. LzLabs explores this restriction in more detail in its white paper the Why Source-code Dependency is a Challenge for Mainframe Workload Rehosting.[5] Mainframe applications are typically compiled into instruction-set architecture- specific machine-code. The mainframe instruction-set architecture is substantially different from the x86 platforms that underpin almost all cloud services.

5 Why Source Code Dependency is a Challenge for Mainframe Workload Rehosting - LzLabs - https://hubs.ly/H0nGMY40

Mainframe Software Infrastructure

Legacy mainframe applications rely on infrastructure software to manage batch and online activity, data access and many other legacy mainframe features. Components such as: CICS®, JES, IMS™, DB2® and VSAM are woven tightly into the applications. Like the applications themselves, this infrastructure software is also tied to the physical mainframe hardware; it will not run in a conventional x86 cloud environment.

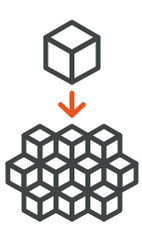
Development, Testing & CI/CD

The more the notion of cloud-native applications is understood, the more the subject of a highly-optimized, development pipeline emerges as central to that notion. Although it's no longer necessary to sign-on to a 3270 terminal to write COBOL programs, and tools like Jenkins can remotely schedule builds of mainframe legacy applications, all other aspects of a mainframe development pipeline are essentially as they were in the '80s and '90s.

In more practical terms, a mainframe development pipeline cannot support many of the rapid deployment features upon which cloud-native applications rely. For example, it's virtually impossible to spin up testing environments on mainframes without extensive planning. There is just no support for large-scale, container-driven integration testing after each merge of a code branch.

Monolithic systems and application architectures make it very difficult to employ strategies like Canary Testing to switch certain users to a newly built component for initial user-acceptance. And the idea of delivering small incremental enhancements on a daily basis is an anathema in this world. In general, these restrictions are the result of what have become known as monolithic applications.

Monolithic Applications



Most mainframe applications were designed and built during an era when scalability was a vertical-only proposition – that of increasing processor, memory and I/O capacity on the same machine image. Consequently, little consideration was given to application design patterns that supported clusters of many machine images upon which the components of the application could be concurrently scheduled. Also, in the heyday of mainframe application development, very little infrastructure software existed to support rapid and "componentized" application development. The result was application implementations which we now refer to as monolithic, where virtually everything the application needed ran within the same application process – or address space in mainframe terms – and was often implemented within a single application module.

Mainframe applications are quintessentially monolithic, insofar as any changes often require an entire application to be deployed for each small change. As we have discussed, the testing landscape for mainframe applications does not lend itself to repeated and frequent scaling or regression testing. Consequently, such small changes are not routinely undertaken. Over time such restrictions have served to reinforce the monolithic nature

of mainframe applications; there is little point breaking them into smaller deployable components if the testing required to support a more modular and agile development pipeline is simply unavailable.

Even when monolithic mainframe applications make use of well-defined interfaces to communicate between software modules, the APIs tend to be implemented by programs that are statically linked into the main application. This use of APIs is incomparable, to even non-cloud-enabled applications on other platforms. The key point is that it is more difficult to refactor mainframe applications to make greater use of APIs than for legacy applications to be implemented on more modern platforms.

Furthermore, mainframe applications have similar dependencies on mainframe infrastructure software that is itself statically linked into the applications.

This tight coupling of application components and the operating environment itself, and the challenges in refactoring this particular kind of application, are central to why it is so difficult for these applications to become cloud-native.

Absence of Cloud-Native Infrastructure

Much of the technology that has emerged in recent years to support cloud-native applications is simply not available within the legacy environment of mainframes.

Much of the technology that has emerged in recent years to support cloud-native applications is simply not available within the legacy environment of mainframes.

Chief among the reasons why the mainframe environment is unable to run cloud-native applications is lack of support for containers that can run legacy applications, or any of the orchestration services upon which their exploitation depends. Without the ability to run application components using a containerized deployment model, many of the other cloud-native requirements become unachievable.

For example, even if you wished to deploy a newly minted application change, isolated from the rest of the environment and application components, in its own container for testing purposes, you would instead need to replicate potentially the entire application, data, and infrastructure.

Logging, Tracing & Monitoring

The mainframe environment is one of the most thoroughly instrumented computing architectures. The standards of reliability it has established over the decades are rooted in its default logging and monitoring capabilities. Furthermore, many independent software vendors have emerged to augment these features with even greater levels of precision and diagnosis. Yet, like many other aspects of legacy mainframe computing the lens through which this instrumentation works is that of monolithic applications, running across a limited number of machine instances; with a consistent set of infrastructure components. As granular and detailed as it is, it cannot provide (in functional terms) what's needed for its applications to become cloud-native.

Eliminating the Cloud-Native Friction with a Software Defined Mainframe

As the name suggests LzLabs Software Defined Mainframe® (LzSDM®) is a mainframe implemented in software. It is the ultimate expression of infrastructure as code, insofar as it eliminates any lock-in between a legacy mainframe application and the underlying hardware for which it was originally designed.

It is based on two different, but entirely complimentary concepts. These concepts are similar to the way Java operates across environments. And to aid in that analogy, we refer to the concepts in a similar way.

LzLabs Virtual Machine (LzVM)

LzVM is the component of LzSDM that eliminates the instruction-set issues of running customer mainframe applications, which exist only in binary form, on x86 computers; thereby opening up the entire world of cloud-native facilities. LzVM shares many of the core features of a typical Java Virtual Machine (JVM). It loads mainframe binary programs in the same way a JVM loads a Java byte-code program. LzVM also features optimizations such as Just-in-Time (JIT) compilation to ensure the programs work optimally on an x86 computer.

LzLabs Native Interface (LzNI)

As we saw in the previous section, there is no escaping the fact that legacy programs depend on services provided by software infrastructure that only runs on the legacy mainframe operating environment. The programs interact with these features in a variety of ways, each of which is analogous to what one might describe as an API. This proprietary software infrastructure is just not present on x86 computers running Linux.

LzSDM solves this problem by replacing the implementations of the APIs, which formerly locked a legacy application into mainframe software infrastructure, with implementations based on open-source components. These components deliver the same outcome for the application as it previously had on the mainframe.

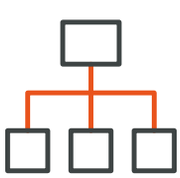
When LzVM encounters a request for a service provided by such an API, it hands that request to LzNI. LzNI uses underlying features on the Linux environment to satisfy the request, in a way that ensures the behavior of the application is preserved. Crucially, it is in LzNI where many of the capabilities of LzSDM, which enable mainframe applications to become cloud-native, reside.

In conjunction, these two facilities – the LzVM execution engine and services provided by LzNI – enable legacy mainframe applications to genuinely operate and appear as Linux applications. From all practical perspectives, there is now little architectural distinction between a legacy mainframe application running within LzVM and using LzNI on an x86 computer, and a Java application running within a JVM and using JNI on an x86 computer.

With the mental model of the runtime environment in place, it becomes easier to understand how the rest of the journey toward a cloud-native implementation of mainframe legacy applications unfolds.

Decomposition

Simply moving a monolithic legacy mainframe application to a cloud environment does not make it cloud-native. However, by using LzSDM, the journey down that path begins. This section of the paper builds on the fundamentals of LzVM and LzNI to illustrate how the path to cloud-native mainframe applications emerges.



APIs

The first thing that happens, when using LzSDM as the platform to move legacy mainframe applications into the cloud, is that all the statically-linked, platform-specific API implementations, which tie the applications to the legacy mainframe, are removed.

The removal of the existing API implementation is crucial on two levels. First, as mentioned in the previous section, the existing API implementation code will not run on an x86 cloud, so there is no point in its remaining present. Secondly, the existing APIs, and their implementation, perpetuate the monolithic nature of the application. Components

of the existing implementation code tend to be woven tightly into the applications via static linkage. The implementations dictate certain service models that cannot satisfy the requirements of a cloud-native application.

The various API implementations are replaced within LzSDM world by the LzNI capability described above, using a consistent and dynamic model. The implementation for these new APIs all follow the same basic pattern; the implementation is based on standard Linux facilities and cloud-native components, augmented with LzSDM code that ensures the applications requesting the APIs get the same result from the new API implementation as from the legacy implementations they replace.

What this implementation model allows for is the exploitation of cloud-native components in areas that provide immediate benefit as the application is migrated to the cloud. Three examples of this are presented to illustrate specifically what this means.

What this implementation model allows for is the exploitation of cloud-native components in areas that provide immediate benefit as the application is migrated to the cloud.

Firstly, LzSDM uses cloud-native IPC services such as gRPC[6] for communications between applications and system services, such as input/output services. LzSDM replaces the synchronous calls between an application and the mainframe I/O subsystem, with references to these modular gRPC services we have developed. In so doing, the erstwhile mainframe application will now benefit from all the inherent capabilities of load-balancing, tracing and granular authentication that gRPC provides in a cloud-native context.

Secondly, LzLabs has implemented distributed lock management within LzSDM on top of a cloud-native, scalable, distributed solution, etcd,[7] to serialise access to resources across instances of LzSDM within the cluster. etcd is a reliable, fast, key-value store for critical data items that need to be synchronised in a cluster. It is a core component of Kubernetes, used to manage state and configuration within a Kubernetes cluster. In conjunction with its container-friendly implementation it is a perfect tool to serialize access to resources. By using etcd, LzSDM enables customers to run mainframe applications in horizontally-scaled, server-less containers; with the same integrity as if they were running in a mainframe Sysplex.

Thirdly, is the implementation of databases within LzSDM. Regardless of which database API a mainframe application is using, within the implementation of the database API, LzSDM normalizes the syntax of the request into either Structured Query Language (SQL) or Cypher Query Language (CQL) and ensures all semantic inconsistencies between the intent of the original mainframe database request and the SQL or CQL implementation are compensated for. LzSDM then proceeds to use a standard cloud service database API to satisfy the request. This mechanism enables mainframe applications to take full advantage of Relational and Graph services provided by cloud-service providers.

gRPC, etcd and cloud database services are just three of a range of cloud-native components LzSDM relies on to transform the future potential of mainframe applications.

6 gRPC - A high-performance, open source universal RPC framework - https://grpc.io/

7 etcd - https://etcd.io/

Progressive Refactoring

Sometimes called incremental modernization, progressive refactoring is the most effective method of reaching the promised land of a fully cloud-native implementation.

In its' report: Use Continuous Modernization to Build Digital Platforms From Legacy Applications[8], Gartner's headline recommendation reads "Legacy application portfolios are often viewed as a problem and subjected to large-scale rip-and-replace efforts. Application leaders should instead manage their portfolio as an asset, removing impediments and executing continuous business-driven modernization to provide optimum value".

At its most fundamental, LzSDM is primarily designed to make progressive refactoring toward a fully cloud-native implementation for mainframe applications a reality.

Before proceeding to explain how LzSDM supports progressive refactoring, an important point of clarification must be made. As we have seen, a core function of LzSDM enables mainframe binary programs to run on x86 cloud hardware without changes. Yet we are now discussing changing programs in the context of progressive refactoring; this seems like a contradiction.

Central to ability of LzSDM to enable progressive refactoring is the fact that by enabling mainframe binaries to run unchanged, it allows a refactoring program to focus on only those programs that need to be changed. Any other program dependency can remain in its binary form.

Stated differently, conventional mainframe applications may consist of hundreds, and in some cases thousands, of programs with a complex chain of interdependencies. It's likely that the initial scope of any progressive refactoring exercise will be limited to just a handful of those programs. LzSDM ensures that the final refactored program can continue to depend on the remaining programs, which will be in mainframe binary form, in a perfectly natural way. The alternative would be untenable; changing potentially thousands of programs just to support the refactoring of handful.

LzWorkbench™

Central to the ability of LzSDM to support progressive refactoring toward a fully cloud-native implementation is LzWorkbench. LzWorkbench is a set of plug-ins to Eclipse; LLVM-based compilers; a runtime environment for interoperability, and a debugging infrastructure. LzWorkbench is designed to make refactoring legacy mainframe applications an identical experience to working on conventionally modern applications.

LzWorkbench fits perfectly into a modern development pipeline. The fit is so perfect that the only difference between working on the applications, which originally ran on the mainframe, and any other application, is the language the program is written in.

This consistency means the entire scope of cloud-native services becomes available. With containerization of these applications a standard option, and the mechanism by which LzSDM can uncouple even tightly linked components of a monolithic application, the

8 Use Continuous Modernization to Build Digital Platforms From Legacy Applications (ID G00344837) - Gartner - https://www.gartner.com/en/documents/3846764/use-continuous-modernization-to-build-digital-platforms-0

DEVELOPMENT PIPELINE INTEGRATION

move to an architecture based on microservices is no more complicated for mainframe legacy applications than for any other application. These mainframe applications can now be enhanced according to exactly the same DevOps methodology as any of the other applications undergoing progressive refactoring.

Stateless Services

The preservation of state is obviously crucial for enterprise applications. However, the mechanism by which state is preserved has an impact on how far towards a cloud-native implementation the application can travel.

The question of state has two dimensions in respect of the way LzSDM enables legacy mainframe applications to become cloud-native.

The first of these dimensions relates to the services LzSDM provides to the applications. As previously mentioned, LzNI is the component of LzSDM that implements the features mainframe applications rely on to run successfully. LzNI features are implemented to the maximum extent possible using popular open-source libraries and specifically cloud-native projects. The net result is the state is never inadvertently stored between invocations of the applications, by the infrastructure itself. But when state is to be stored it is stored through entirely conventional means.

For example, LzSDM must provide a job management service, compatible with mainframe job submission, to enable a batch workload to function. This service is implemented entirely statelessly, and as such can be scaled like any other cloud-native workload, with incoming requests balanced using standard cloud-native load balancing. However, the batch jobs it manages, and any intermediate states of those jobs, are maintained transactionally within a shared PostgreSQL instance.

LzSDM also implements a form of workload separation, where services for directly managing files and other resources, which maintain state, can be containerized separately

from the processes that act upon them. In doing so, these services can be managed by Kubernetes as StatefulSets within the cluster.

Readers familiar with mainframe concepts will understand clearly the analogy with Resource Owning Regions and File Owning Regions. LzSDM enables such separation, within cloud-native infrastructure, without forcing code changes.

There are other examples, too numerous to mention, where LzSDM design has been informed by the needs of state management in a cloud-native world.

Secondly, mainframe applications were developed long before the cloud-native notion existed; consequently, they maintain state in a variety of ways. Although, rarely was the action considered a stateful programming decision when these applications were originally written.

This is a very hard problem to solve. In some, but by no means all, cases the storing of state within mainframe applications occurs through well-defined APIs. It is these APIs that have been reimplemented by LzNI. Wherever possible, LzLabs is augmenting these LzNI implementations with capabilities to ensure that any action that could create an affinity is shared in the cluster in a way that doesn't impact application performance or outcome, but does eliminate any state-oriented constraints on horizontal scaling.

Logging, Tracing & Monitoring

One of the drawbacks often cited for cloud-native implementations is complexity of topology. A monolithic legacy application can be visualized easily, has a manageable number of interoperability points and its runtime whereabouts are entirely predictable. A fully implemented cloud-native application is almost the exact opposite.

Fortunately, the cloud-native movement has provided a well-supported and implemented set of technologies to help manage the increased complexity of cloud-native applications. These technologies are implemented to a greater or lesser extent within all the LzNI implementations LzLabs has developed for LzSDM.

The Jaeger root-cause tracing mechanism is implemented throughout LzSDM. Using Jaeger ensures that any choice of visualisation tool, which supports the OpenTracing standard, can see how all the components of an application work across the cluster and where any degradation is occurring.

Similarly, the use of Prometheus within the LzNI implementations enables raw system performance clocks and counters to be aggregated across the cluster.

Since many of the other cloud-native technologies implemented within LzSDM contribute to this instrumentation infrastructure, the level of granularity dwarfs anything the management of these applications would have enjoyed in their original mainframe implementation.

Importantly, these management features, as implemented in LzSDM, go a long way to ensuring that legacy mainframe applications can be treated exactly the same as any other cloud-native application from the very important perspective of service-level management.

Summary

System-of-record data residing on mainframes is some of the most intrinsically valuable data to the enterprise. Yet the applications and services available to leverage that data are constrained in their agility by a platform that enjoys none of the innovation present in cloud-native computing.

The conventional wisdom is that these mainframe applications are virtually impossible to move to the cloud. However, when based on LzLabs Software Defined Mainframe (LzSDM), the movement to the cloud becomes far easier and with far less risk than previously imaginable.

Importantly, using LzSDM, it is not only possible to rehost mainframe legacy applications on cloud infrastructure, but it can be done in ways that instantly increase the application's opportunities to take advantage of cloud-native features, without requiring any changes to application source-code. Furthermore, the applications are now positioned perfectly for progressive refactoring towards a more complete cloud-native implementation.

About LzLabs

LzLabs is a software company that develops innovative solutions for enterprise computing customers, including its **LzLabs Software Defined Mainframe® (LzSDM®).** The company was founded in 2011 and is headquartered in Zürich, Switzerland. LzSDM® liberates and enables customer legacy applications to run unchanged on both Linux hardware and Cloud infrastructures. Thousands of mainframe transactions are processed per second, while maintaining enterprise requirements for reliability, availability, serviceability, and security. Our software solution provides unrivaled compatibility and exceptional performance, dramatically reducing IT costs. LzLabs' offices in Switzerland and the UK are home to highly-experienced mainframe experts and modern IT thought leaders from across the globe.

Contact Us

LzLabs GmbH
@LzLabsGmbH
info@lzlabs.com

LzLabs GmbH
Richtiarkade 16
CH-8304 Wallisellen,
Switzerland

+41 44 515 9880

Duke St., 7th Floor, Block C
Duke's Court Building
Woking, GU21 5BH
United Kingdom

+44 (0)1483 319185

lzlabs.com/products