Financial Industry – Containers, VMs, Cloud Native – an Overview of the approach

- Industry / Market Perspective
- FS’s key IT Challenges
- What does the future look like
- Build / Operate / Support
- How do you get there

FS Objectives and Challenges

Objectives

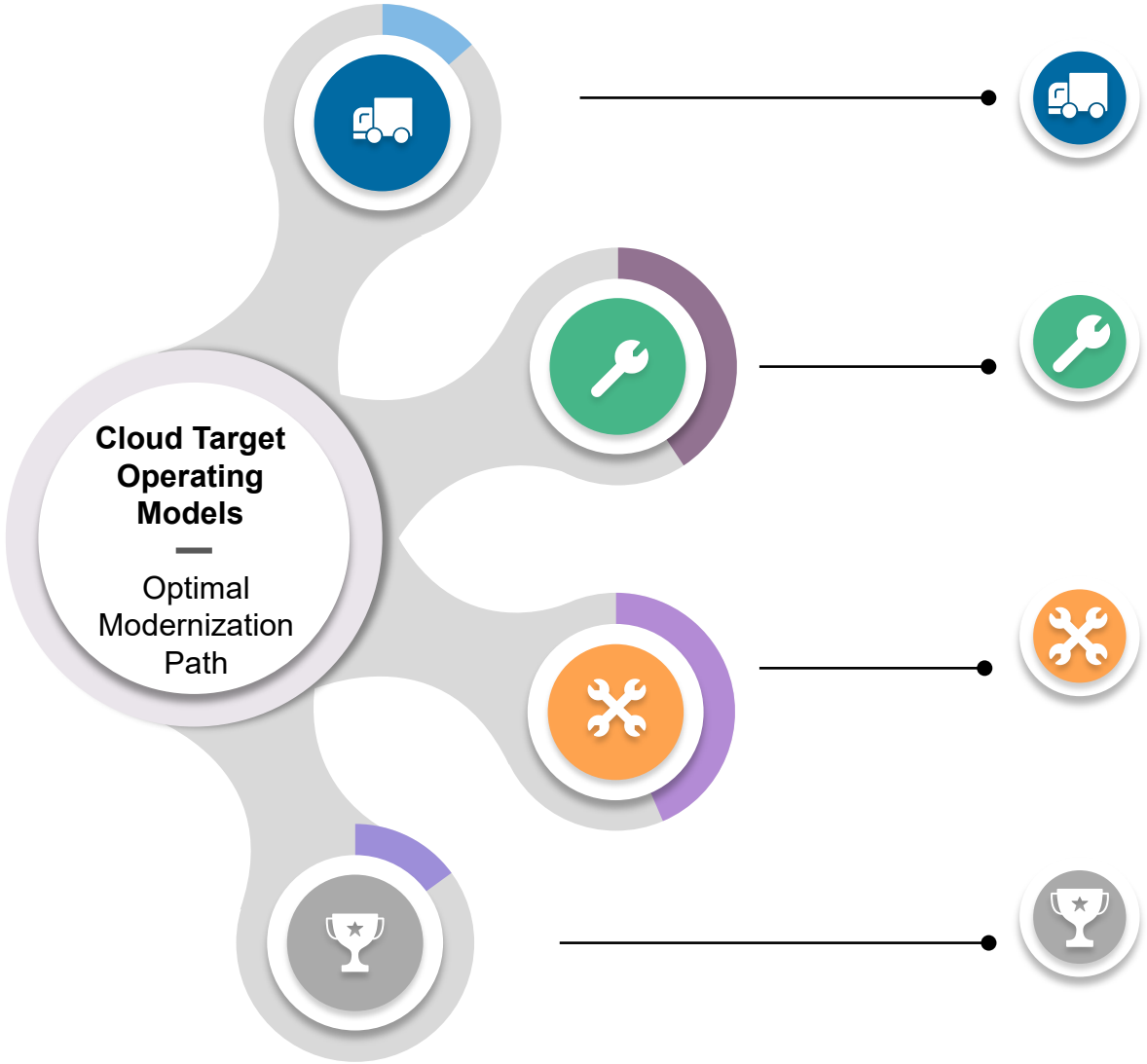
- Increase speed to market and enable delivery of features/functionality to production in very short timeframe and increase scalability
- Reduce cost through transformation & automation

Current State Challenges

- Complexities of bi-modal operations - need to enable and accelerate transformation to cloud while supporting legacy operations and optimizing cost of operations
- Complex environment with tightly coupled dependencies
- Lack of ability to quickly diagnose issues in the environment (often requires manual troubleshooting and consumes lot of time and resources from multiple teams)
- Lack of full automation of build-deploy-test-release cycle
- Increased workload due to new cloud based technology platforms while lack of reduction in demand for legacy environment support
- Need for better governance to enable adoption / readiness and management of modernized platforms

Application Categorization for Cloud Modernization

Cloud modernization paths allow tradeoffs between upfront investment and long term expectations at the workload level



Lift and Shift
Replication of the current systems and applications from the current environment to the new environment. The lowest risk migration but the lowest value and cost savings.
Future state support model limited by constraints in the legacy application architecture.

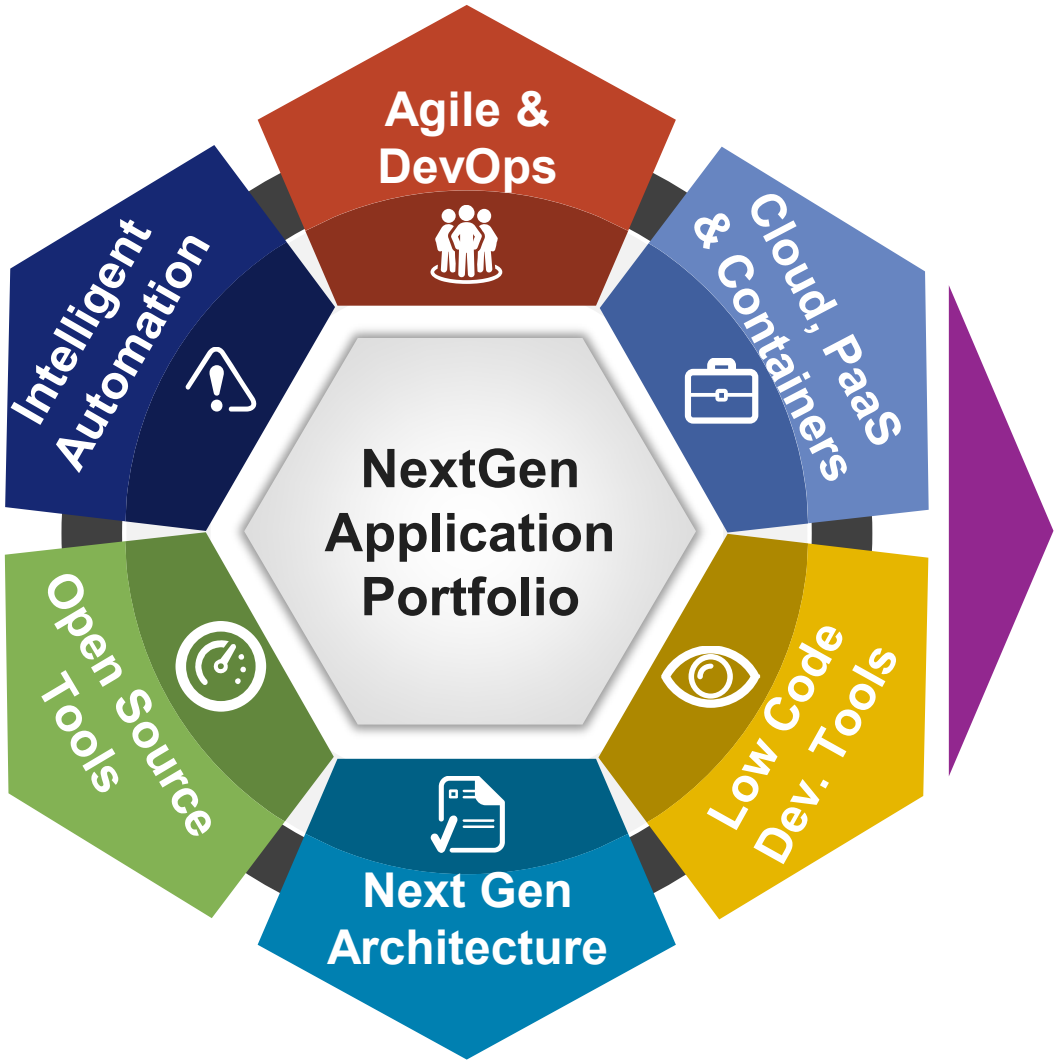
Cloud Infrastructure Ready
A lift and shift based migration which incorporates cloud functionality not requiring application changes. These functions may include DBaaS, DR, and Backup.
Shifting of operating and support efforts/costs to the cloud.

Cloud Services Ready
Migration of applications into a cloud environment taking advantage of application aware cloud services. This migration requires moderate application modifications to support autoscaling, automated provisioning, cache, queue and notification services.
Development and deployment services are fully managed in the cloud and integrated up to the application layer.

Cloud Native
Migration to native public cloud services. This requires significant application modernization to take advantage of serverless and micro-services but offers the highest value and long-term cost savings.
Mature DevOps automation, Self-healing, & telemetry is need to ensure support costs do not increase due to the increased number of components.

Six Industry Forces

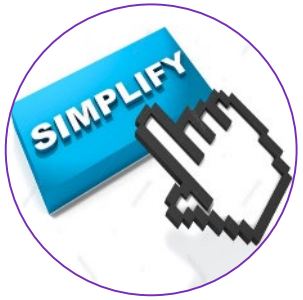
Impacting Next Generation Enterprise Architecture



Enabling

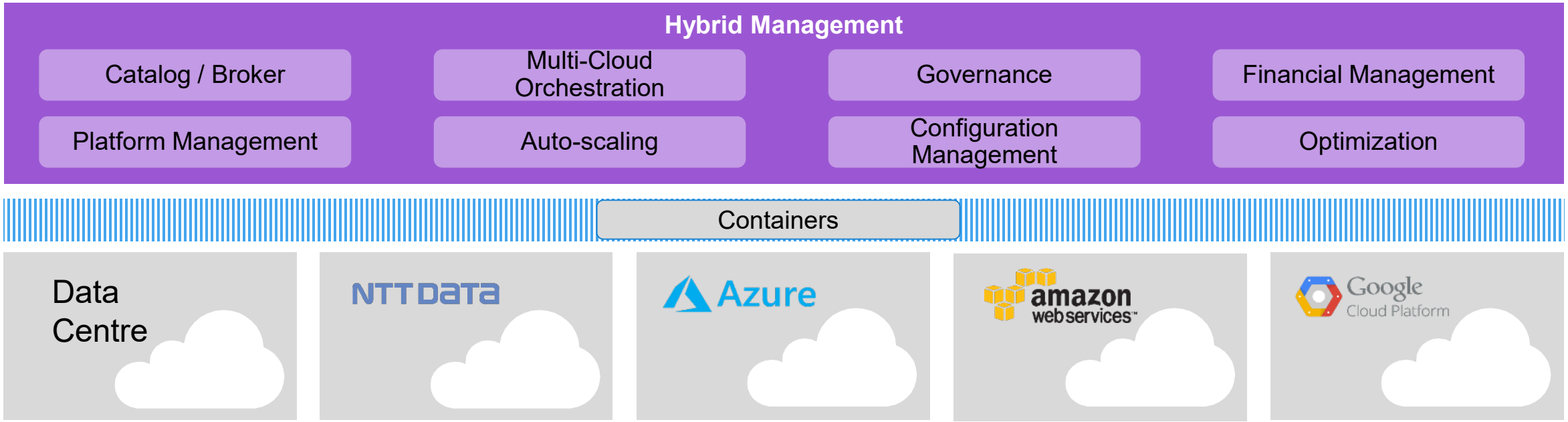
- Low-cost horizontal scalability
- Lower software costs
- Daily/weekly software releases
- Business & IT agility
- Improved business & IT automation
- Faster time to market

Impact



Most organizations are utilizing hybrid clouds with on premise and public clouds

Multi-cloud operations require the ability to manage and run environments across a range of cloud platforms



Leveraging the benefits of various platform options

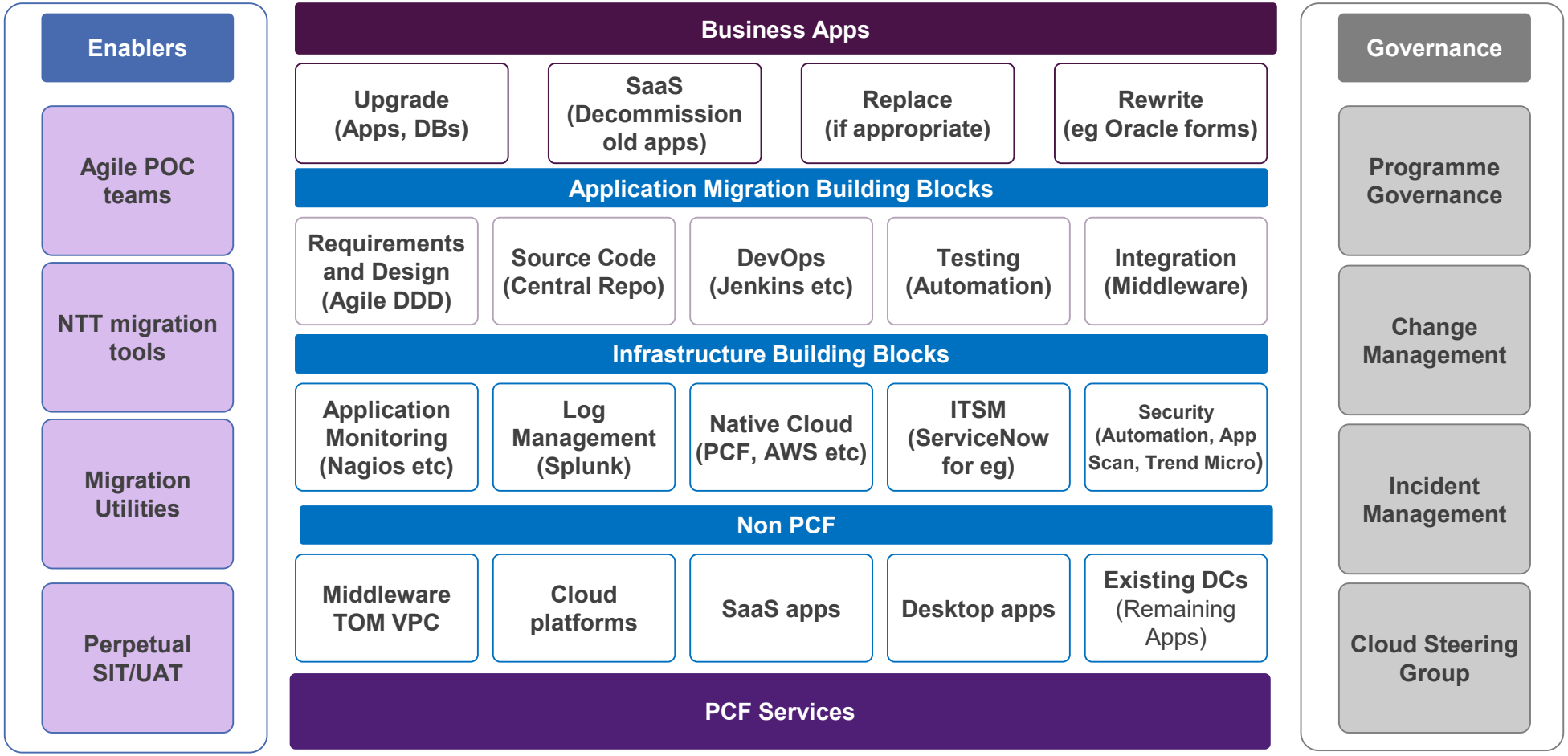
- Auto-scaling and enablement of burst capacity
- Expanded selection of location, functionality, platform ecosystems
- Benefits of onsite, private, and various public platforms available to meet the needs of specific workloads

Platform Modernization

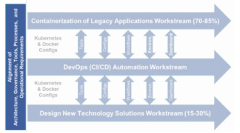
Key Assumptions Underpinning our Approach

- Many FS firms, incrementally migrating to a cloud platform (s)
  - Eg 15-30% of applications will be new/re-engineered using new technology standards
  - Eg 70-85% of applications will be containerized using PKS
- Two software delivery pipelines/technologies will need to be supported for several years
  - Legacy (existing) pipeline
  - Next Gen pipeline based on containers
- A detailed modernization roadmap designed based on app needs / dependencies
- Incremental transition to supporting a container, native-cloud delivery pipeline (Kubernetes and Docker e.g.)
- A consolidated DevOps(CI/CD) pipeline designed and implemented

Platform Modernization: Example of a complete picture

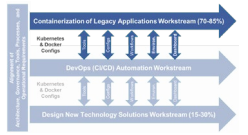


Roadmap for New technologies: General considerations



TRM IT Considerations	Comments
Architecture Principles	ARB, Sign offs, CRs, Design Pattern Book, IT standards, COE and Agile standards
Current Estate	Documented, Known, Dependencies mapped
Migration Plan	SMART (specific, measurable, actionable, realistic, time-bound)
Ranked Applications	Based on what can be containerised now vs what needs to be refactored
Ranked migration of apps	Simplest, fewer dependencies, not business critical should go first
HLD and EA are iterative	Built before but updated during the process
Domain Driven Design	Business Logic, Workflows comes from domain users (eg refactoring)
Security	Main principle built into HLD, includes AD, LDAP, PAM, OS, Perimeter etc.
Automate Testing	Payback is 5:1

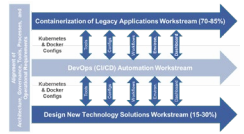
Roadmap for legacy technologies



Migration Considerations in 4 areas

Area	Principles	Comments
Middleware	Understand current estate App by App API TOM Design via POCs	Design Pattern Book API Design Skill sets
Dependency	Between Middleware and Applications DB Connectors, Interfaces, Schedulers, Events	Mapping out dependency on app per app basis
Database	Persistence vs Stateless SQL vs NoSQL Mapping out file movements	Polyglot architecture
Containers	Microservices Decoupling Disposable	Most legacy stateful not stateless Requires automation/ops

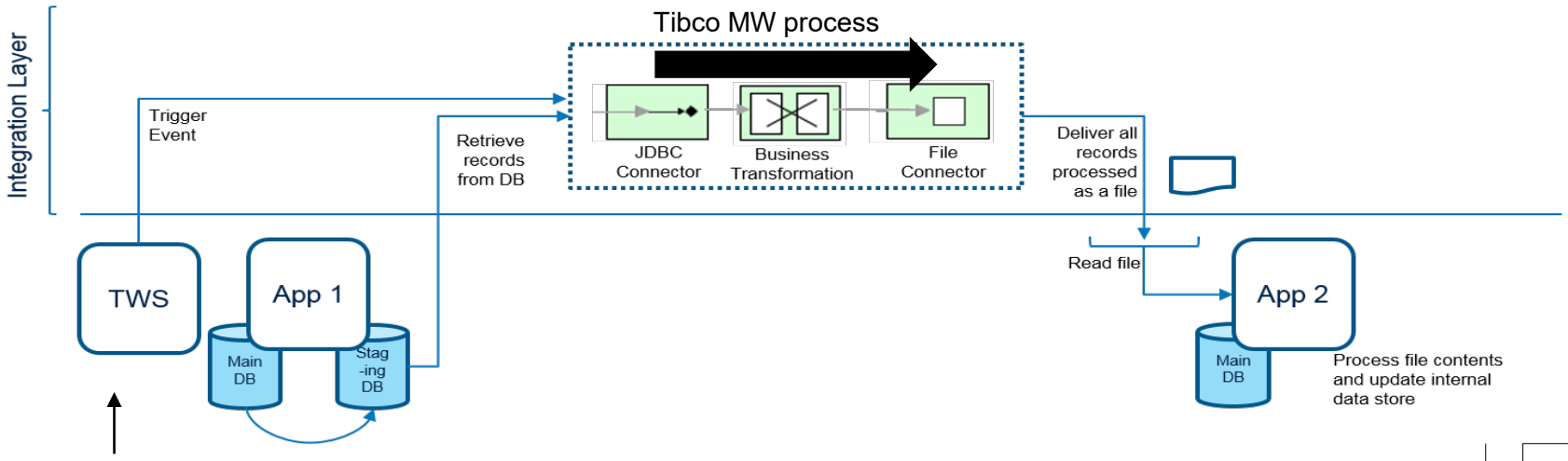
Migration Considerations: Containers



Design is Important	Comments	Legacy, New Build
PCF 12 Factor Guidelines for Cloud Native	Best practices	Could be for both, usually references New Cloud Native Builds
Stateless vs current Stateful	Disposable vs. Persistence & Dependent	For both (legacy is likely stateful, impacts design)
Compute-Network-Storage configs	Containers per host, runtime issues	Part of design, both
Orchestration	Discovery, Registration	For both
Network Constraints	Overlays	Usually references legacy
Storage Considerations	Container as a Storage, Read vs Writes	References legacy (though new builds would also consider these factors)
Monitoring, Logging	Splunk, Grafana	Both
Security	Hardened images, CI/CD, Containers share the same kernel, not isolated	Both (through processes may differ), Security risk with Containers
Scalability is in the code	Containers don’t scale	New Builds (not refactoring legacy)
VMs, CD in the Architecture	Are valid depending on the use case	VMs-Legacy; Ci/CD - Both

Roadmap for legacy technologies

Middleware example: Moving from legacy to iPaaS



An external job schedule, e.g. TWS or something similar, triggers a Tibco service or there is a poll schedule configured as part of the flow configuration. Within the Middleware service:

-A JDBC connector connects to the source application's database and identifies the new/updates records to process.

-The service validates the data and transforms the data to the format and schema which the target application is expecting

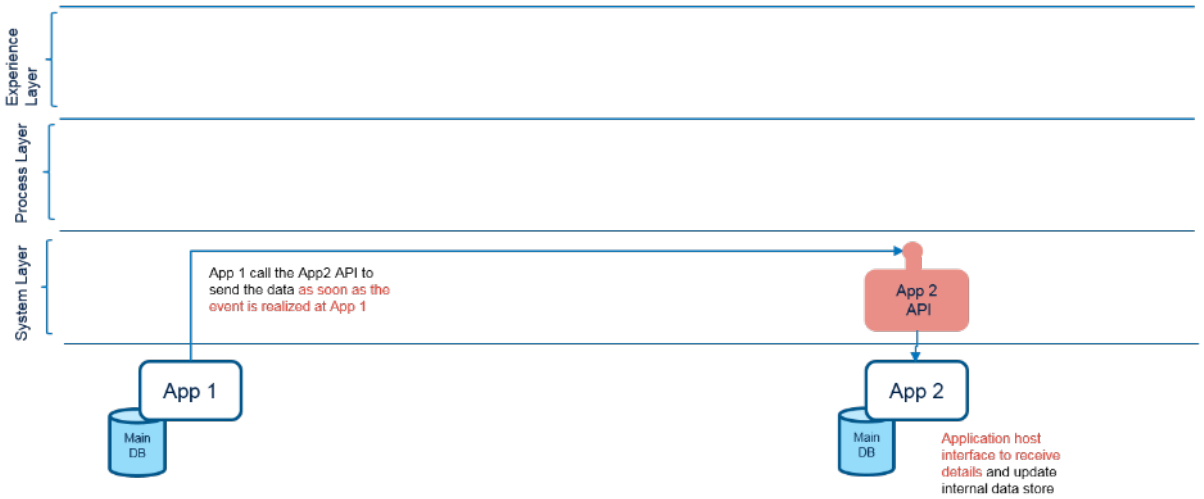
-The transformed data is then placed in the file buffer

-MW service delivers the file to the file location at which the target application will process the file and updates its data store

This process is moved to a System level API-led design, with DB and file integration. No business logic transformation. See next slide.
Note: there are other variations on this pattern

Roadmap for legacy technologies

Middleware example: iPaaS replacing legacy



There are different types of APIs
Need to standardize design approach

This process is moved to a System level API-led design, with DB and file integration

- Target application(App2) should have features to host the system API.
- Source application(App1) should have a feature to call the App2 API to post data event as soon as the change event is realised in App 1.
- This App2 API will validate the message, which it received from source system, and call the interface exposed by App2.

Note: there are other variations on this pattern

DevOps / Automation Containers

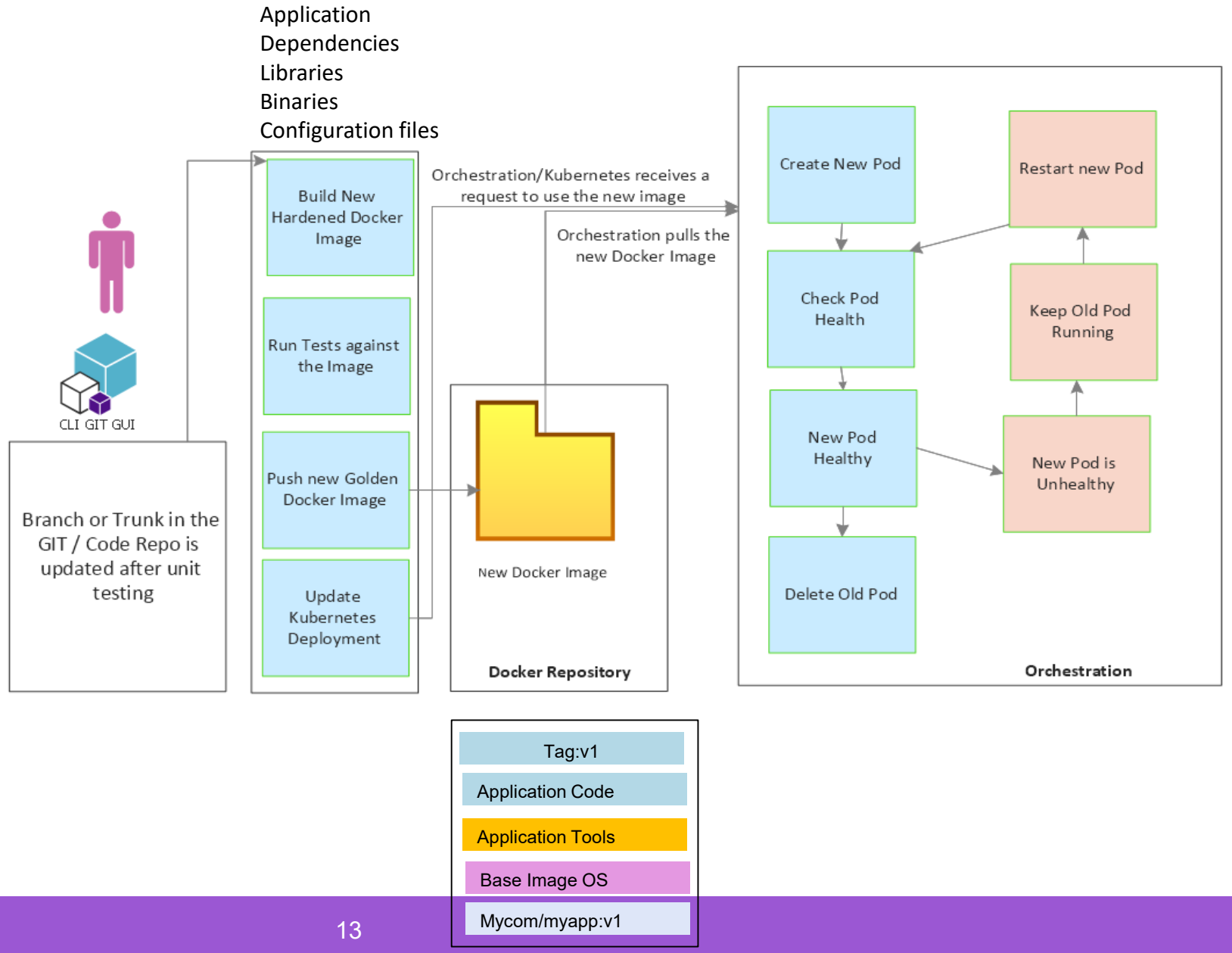
- Example CMP Nirmata

1. Source:
   - Check in the source code .java files etc.
   - Unit tests

2. Build:
   - Compile the code
   - Unit tests
   - Make & Tag Container images

3. Production / Orchestration
   - Automated
   - Microservices deployment
   - Blue Green, updating applications

4. Deployment Automation
   - CI for image building, SAST, DAST
   - CD for container deployments



DevOps / API & Containers (microservices)

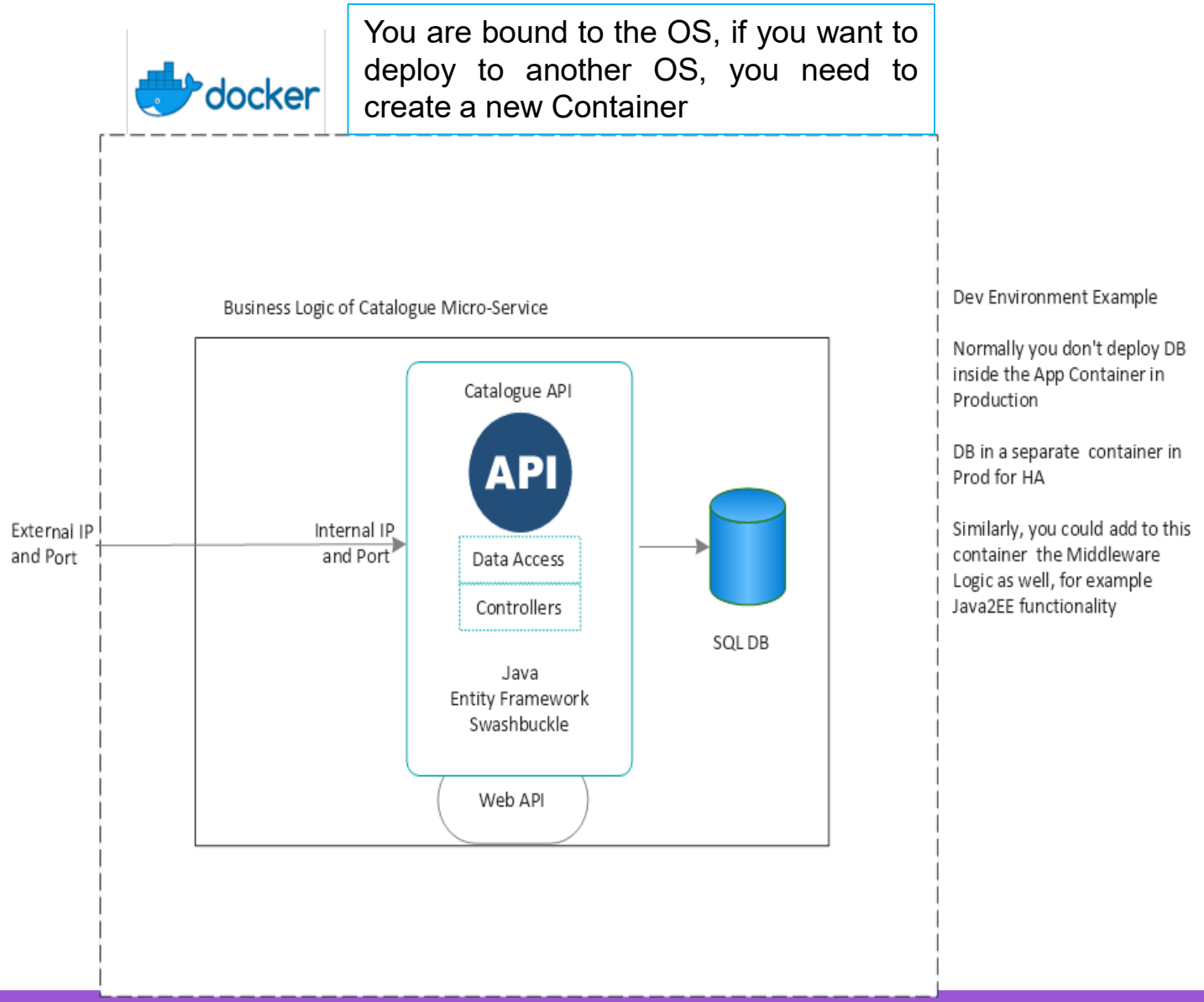
- Easier if a New Build; more difficult if Legacy based

Simple Example:

- I have a Container called ‘Catalog’ (of different loans)
- This service implements its functionality in a single .Java Web API project that includes classes for the data model, business logic, and data access code.
- Data is stored in a SQL Server DB

We need:

- Data Access API or ORM
- Generate meta data description via Swagger (or similar)
- DB connection stings, variables used by Docker
- Dependency injections

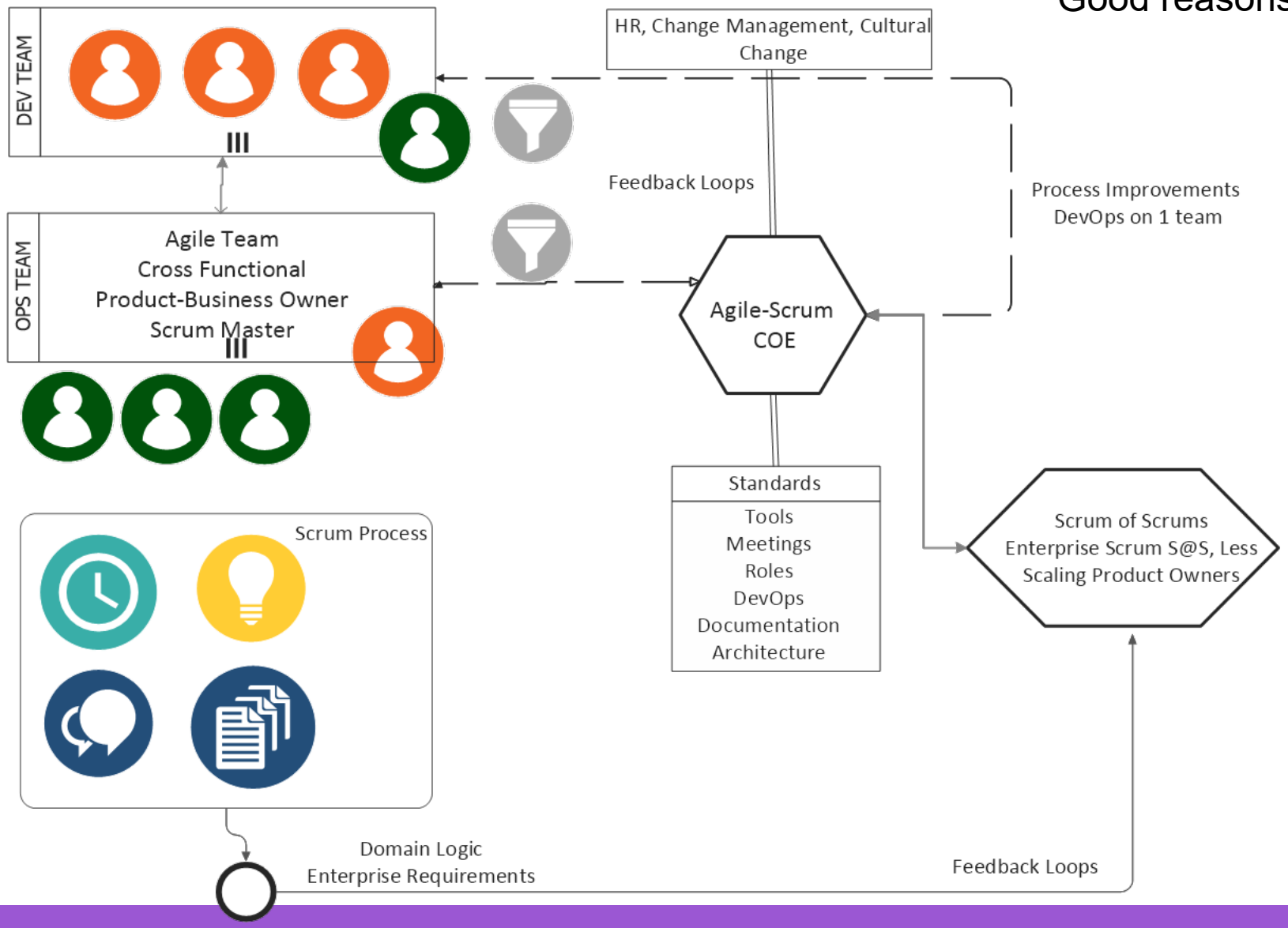


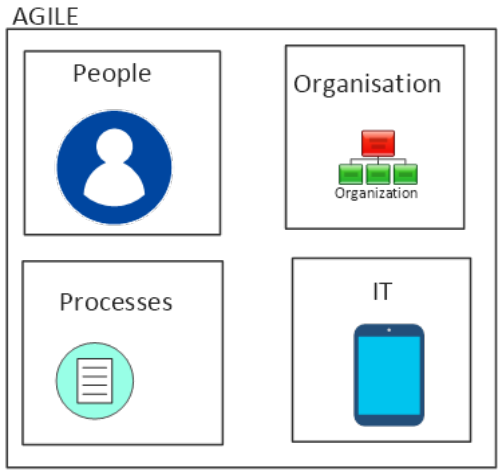
Operational Considerations

- Cluster management
  - Node management – Health, Scaling
  - Authentication and authorization – Single Sign On, Role Based Access controls
  - Configuration management – Platform configuration management
  - Platform security – cluster communication, securing OS
- Network management
  - Ingress traffic
  - Network policies for application access
  - DNS management
- Storage management
  - Classes of Storage – SSD, IOPS, Magnetic, Backups
  - Volume security – securing volumes, application level access
  - Secure data at rest and in transit
- Container Image management
  - Private secure registry with access to CD
  - Container scanning for vulnerability using Jfrog Xray, Black Duck Hub
- Logging
  - Log aggregation with ELK or splunk
- Container Monitoring
  - Use specialized monitoring tools like TwistLock, Sysdig, Rapid 7, Layered Insight
- Secrets Management
  - Use Vaults like HashCorp, CyberArk
  - Rotate certificates and credentials

Implications to Operating Model & Governance – Agile, Scrum COE

Good reasons to have C level sponsored COE





Necessary Capabilities – 3 Workstreams

Area	Principles	NTT can help with:
Middleware	Understand current estate App by App API TOM Design via POCs	-Discovery of what should be left on legacy, vs API-driven (new) -Design Pattern Book (assess, create) -API Design for new -Assess and help Resource Skill sets -HLD and iterative DDD
Dependency	Between Middleware and Applications DB Connectors, Interfaces, Schedulers, Events	-Mapping out dependency on app per app basis (related to above points) -HLD and iterative DDD
Database	Persistence vs Stateless SQL vs NoSQL Mapping out file movements	-Map out the Polyglot architecture -Data persistence vs stateless -HLD and iterative DDD
Containers	Microservices Decoupling Disposable	-Propose automation within this process on PCF (and outside of PCF) -HLD and iterative DDD -CTaaS if that is preferred

Capabilities – Workstreams Continued

Area	Principles	NTT can help with:
Containers	Test Automation, tooling	-Propose automation within this process on PCF (and outside of PCF) -Assess, help implement
Containers	Security	-Security model including Regulation requirements -CTaaS if MSP model preferred -Assess, help implement
Security in General	Configuration control, environmental consistency	-Based on security model -Automated, configured templates (e.g. serverless) -Assess, help implement
TOM Cloud, PCF etc	Best practices	-Resources as needed

Capabilities – General but impacting the Workstreams

Area	Principles	NTT can help with:
Monitoring	One single pane of glass IR, alerts, Resources consumed	-Tool assessment re agents, logs, automation e.g. Splunk aggregation to a Cloud panel -Assess, help implement
ITSM	ITSM integration with Environmental setup (e.g. ServiceNow-Jenkins-Docker-KN)	-ITSM modification and integration, to ensure consistent environmental parameters are set up -Approval process for the above within the ITSM -Assess, help implement

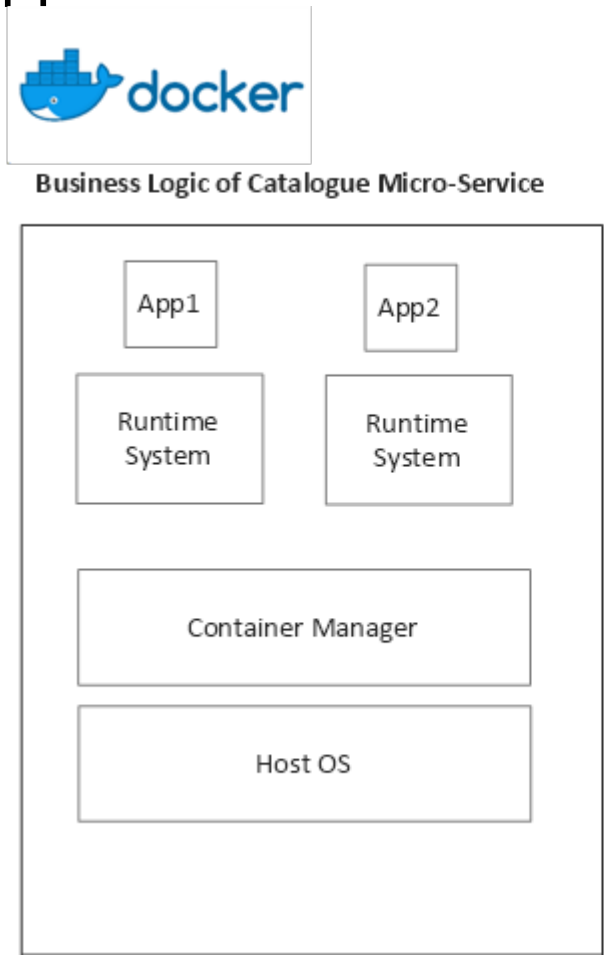
Appendix A: DevOps / Automation Containers – When to use Containers?

When to Containerise	Benefits	Comments
In House Applications	This helps in porting applications, ensuring environmental consistency and control and is faster – app, db, middleware can be separated into micro-services	Knowledge of the business logic and dependencies are necessary in order to 'split' the application into logical containers
Batch job premised Applications	Not real time, not heavy processing, fits well with Containers	Containers are about Abstraction, the more demanding the workload the fewer levels of Abstraction you want
Smaller Workloads and Apps	Easier to decompose and micro-service	ERP, CRM and 'big' legacy apps are difficult to containerise given tight coupling
If you want to automate, simplify deployments	Helps simplify a DevOps deployment tool chain, which no longer needs to differ based on the nature of the runtime artefact (e.g., PHP vs JVM, etc.). All runtime differences are encapsulated within the container	Not all business apps need a complicated CI/CD, CI/CD should be straightforward
Pressure to get out of a data centre e.g. underlying network is terrible or fails and you want to limit resources consumed or maximise node usage	AWS and other public Cloud platforms have world-class networking which immediately makes the app more reliable, responsive	Read-Write, Sync, DB update errors will shrink towards zero with a dark fibre/first class network, Containerising can be very quick and controlled though VMs can also be used of course though it costs more in resources
Old Farms with Old OS as part of a general migration out of the data centre	Example: Win 2003 OS or a Citrix farm (App V is itself a container). You cannot migrate 2003 OS to 2016 R2 – your only option with very old legacy is to wrap the application into a container and port that using some techniques, to a new OS version	Saves the app while you decide on decommissioning, or moving to a SaaS, or rewriting the app

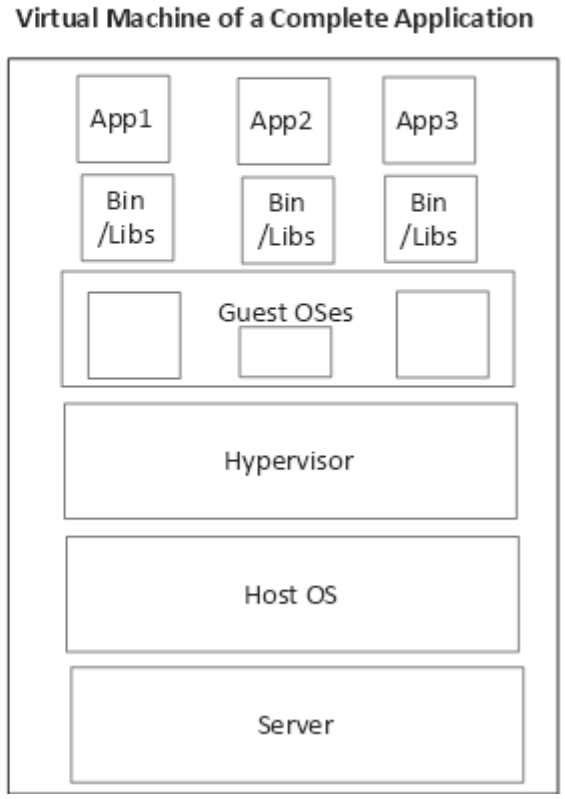
Appendix A: DevOps / Automation Containers – Benefits

Why Containerise?	Benefits
Portability	Containerization makes applications portable by virtualizing at the operating-system level, creating isolated, encapsulated systems that are kernel-based.
Lightweight	To port an application the entire operating system does not need to be encapsulated along with it. It is just the application that goes inside the container
Requires fewer Resources	Can run numerous containers at once without taking up a lot of space, whereas with VMs, doing the same requires many more GBs of space. With only 1 OS for the containers (i.e., one kernel) possible to run far more containers on a host than with full VMs
Rewriting code is not required	We don't need to re-write code for every platform the container is going to run on. This saves time, money, and effort (if different OS, need to create a new container)
Faster	Decomposing the App (if the team has the requisite knowledge!) should result in a 'micro-services' approach which brings benefits including improved performance and decoupled design
Continuous Integration and Continuous Deployment	Docker for eg provides a system for image versioning. The process is pretty simple: setup the build process to pull code from a repository, build it, package it into a Docker image, and push the newly created image into an image repository Then the deployment process (eg Kubernetes) can pull the new image from the repository, test the application, and deploy to the production servers. Since the Docker daemon is the same across all environments, your app will not fail on production, if it worked in Dev and Pre-Prod. Consistency is ensured.

Appendix A: VM vs Containers – Both can be valid



Bin/Libs can be shared or separate



VM = complete virtualization of physical servers into multiple VMs using hypervisor software systems

We can split a server into multiple logical or virtual servers, with each of the virtual sessions being a VM.

Hypervisor allows numerous applications to access the server compute resources

Not OS-bound as with a Container

VMs consume GBs more resources on average

Appendix A: VM vs Containers – Both can be valid

Why use VMs?	Benefits
Security	VMs have their own kernel and they don't use and share the kernel of the host OS, isolated from each other unlike Containers (this is a security risk with Containers)
Multi-OS	VMs residing on the same server can run different OSes. One VM can run Windows while the VM next door might be running Ubuntu. Containers virtualize underlying OS while VMs virtualize the underlying hardware
Dynamic allocation of resources	A hypervisor will take all spare CPU cycles from across all VMs and use them on the VMs that need and can benefit most from the additional processing power. Completely transparent to OS and applications running on the VMs
Portable, HA, DR, can move/copy VMs easily	In a properly configured cloud or hybrid-environment, each VM can be booted from any available physical server node in the server farm, datacenter, or across multiple datacenters. OS and applications within the VM are unaware of this capability
Monoliths	Might be easier and quicker to VM a monolith (tightly coupled, no traversing outside the App boundary), then to try to decompose into Containers where there is no real need to either create a micro-service, or engage in refactoring
Real time, heavy processing systems	Containers are abstract and put layers between server resources and the application. For heavy real time processing, VMs might be more appropriate, given that they are not as abstract