Integration Technical Conference 2019

A06: Practical Considerations for moving to a more Agile Integration Architecture

Using App Connect with Containers

Andy Garratt, IBM UK
andyg@uk.ibm.com

IBM **Cloud**

IBM

Define goals, targets and success factors

Agile Integration Architecture is great!

So is App Connect V11!.

So are Clouds, Containers, Kubernetes, Service Meshes, CI/CD etc

They can help you do a lot – but what do you ACTUALLY want to do?

Defining “Agile”

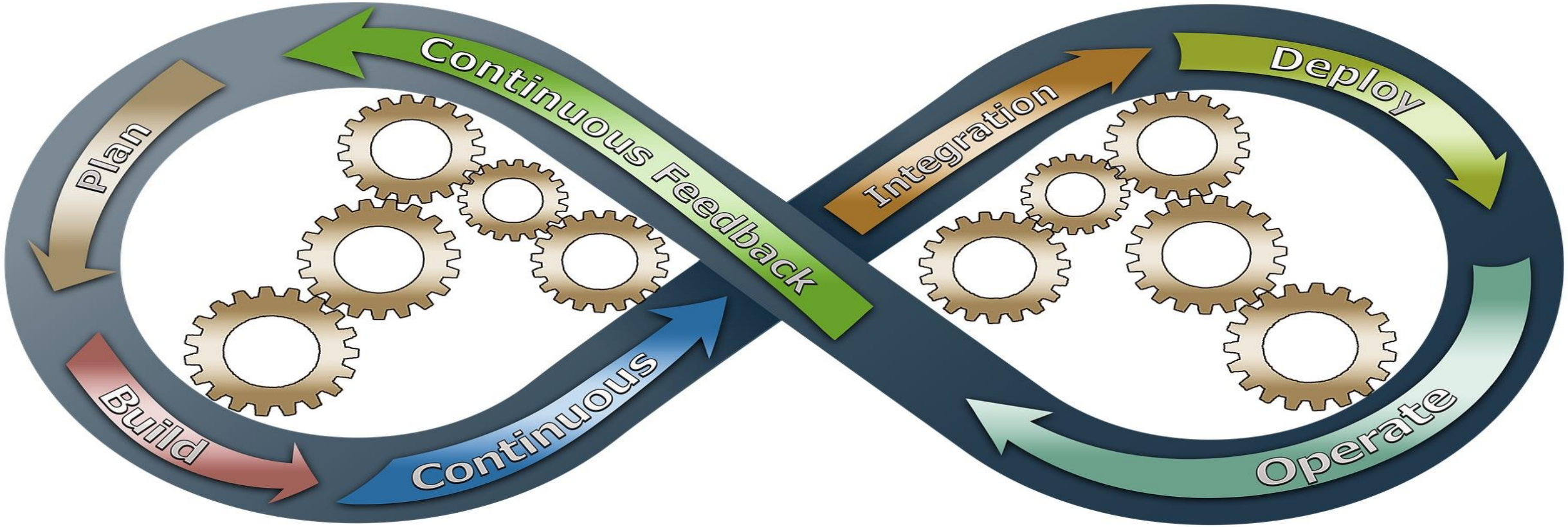
– Individuals and interactions
over processes and tools?

– Working software
over comprehensive documentation?

– Customer collaboration
over contract negotiation?

– Responding to change
over following a plan?

“Our highest priority is to satisfy the customer through early and *continuous delivery* of valuable software”



“Deliver working software frequently, from a couple of ~~weeks~~ hours to a couple of months, *with a preference to the shorter timescale*.”

Risk Free Deployment

"We want to be able to upgrade and/or patch without months of retro-testing"

"We want to deploy or patch an API or a service independently of all other services"

"We want to deploy and patch 'live' without impacting our users"

Cloud / Location Independence and portability

"We want to be able to run our integrations wherever makes most sense. We also want to move them around"

"Minimum/Zero dependencies – embrace the container concept"

"Just another container. Deploy integrations as microservices"

Embrace DevOps

"Everything is built using a fully-automated pipeline"
"Each integration has independent pipelines"

"What gets built in DEV is what's deployed to PROD"

"Developer testing is on a multi-instance cluster. No more 'it works on my laptop'"

Escape the "backlog trap"

"If they want to build integrations with their teams, that's fine"

"We want to be more responsive to our API consumers and be able to build faster"

"Re-use by copy – look at NPM / git copy+fork approach

Agile Integration Architecture

Modernizing integration to enable ***business*** *agility*

Fine grained deployment

Architecture & design

→ Improve build independence and production velocity (***deployment*** *agility*)

Decentralized Ownership

People & Process

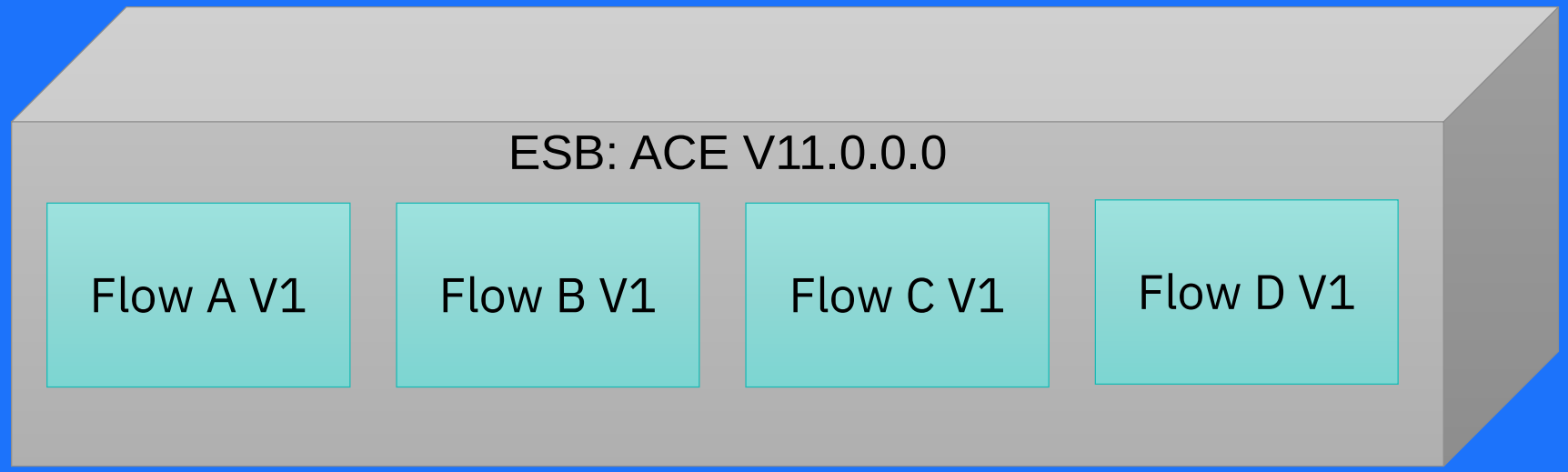
→ Accelerate agility and innovation (***development*** *agility*)

Cloud native infrastructure

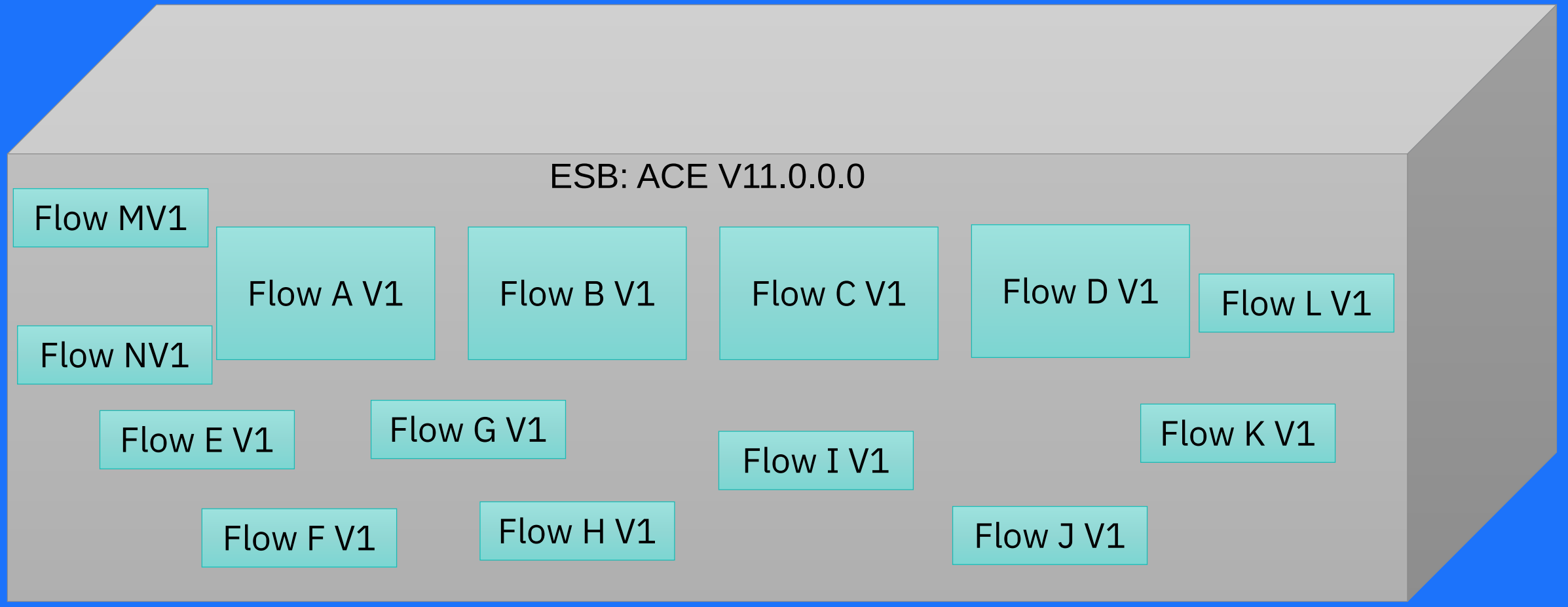
Infrastructure & Technology

→ Dynamic scalability and inherent resilience (***operational*** *agility*)

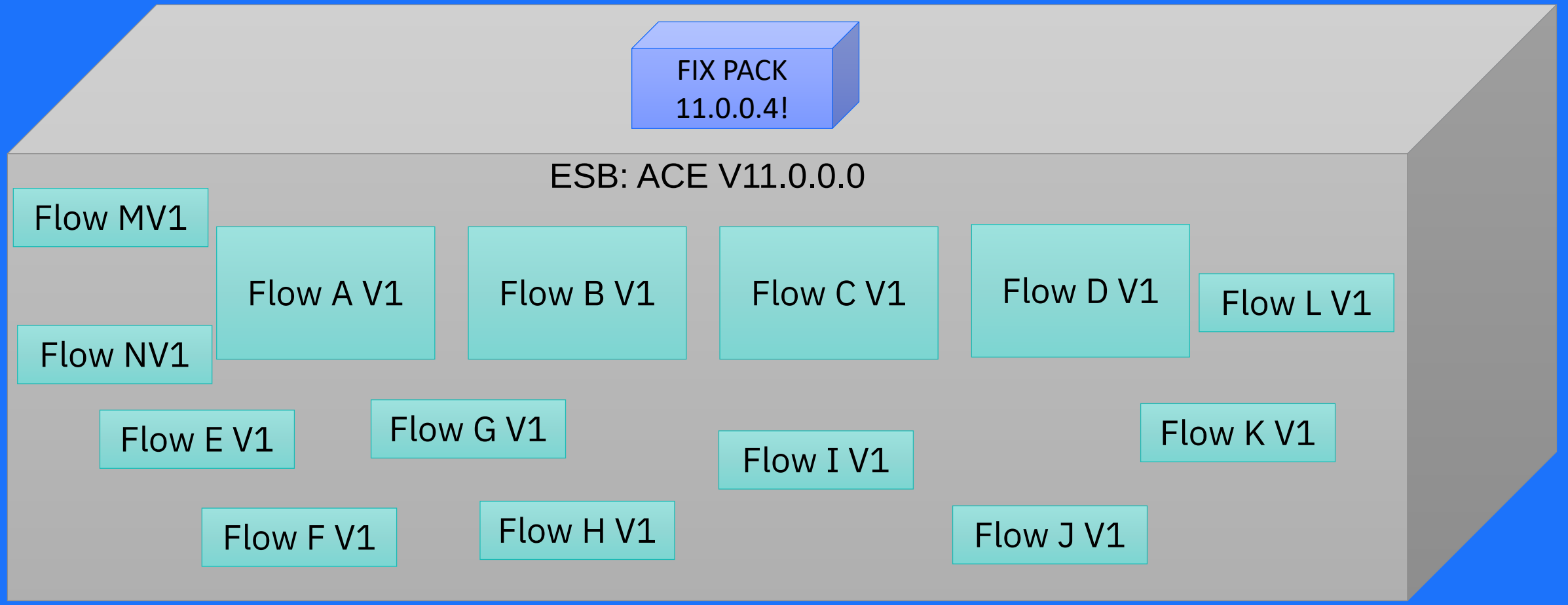
Importance of Independence



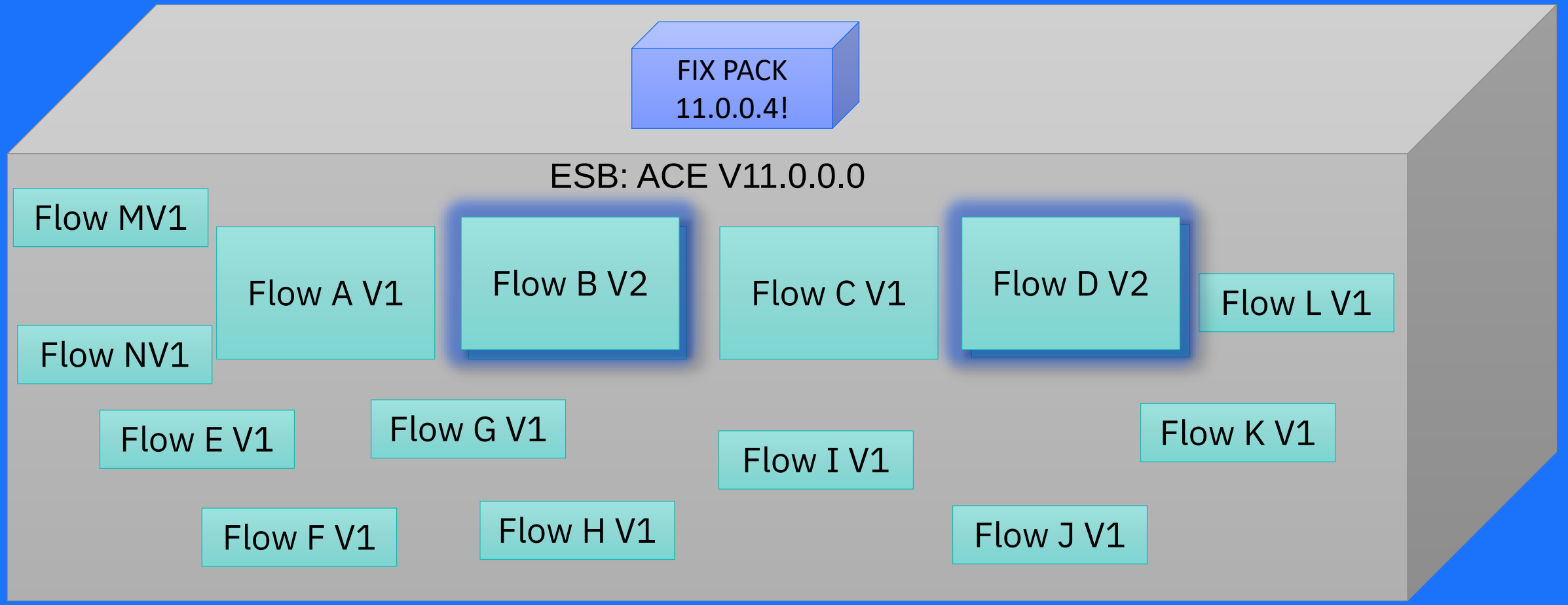
Importance of Independence



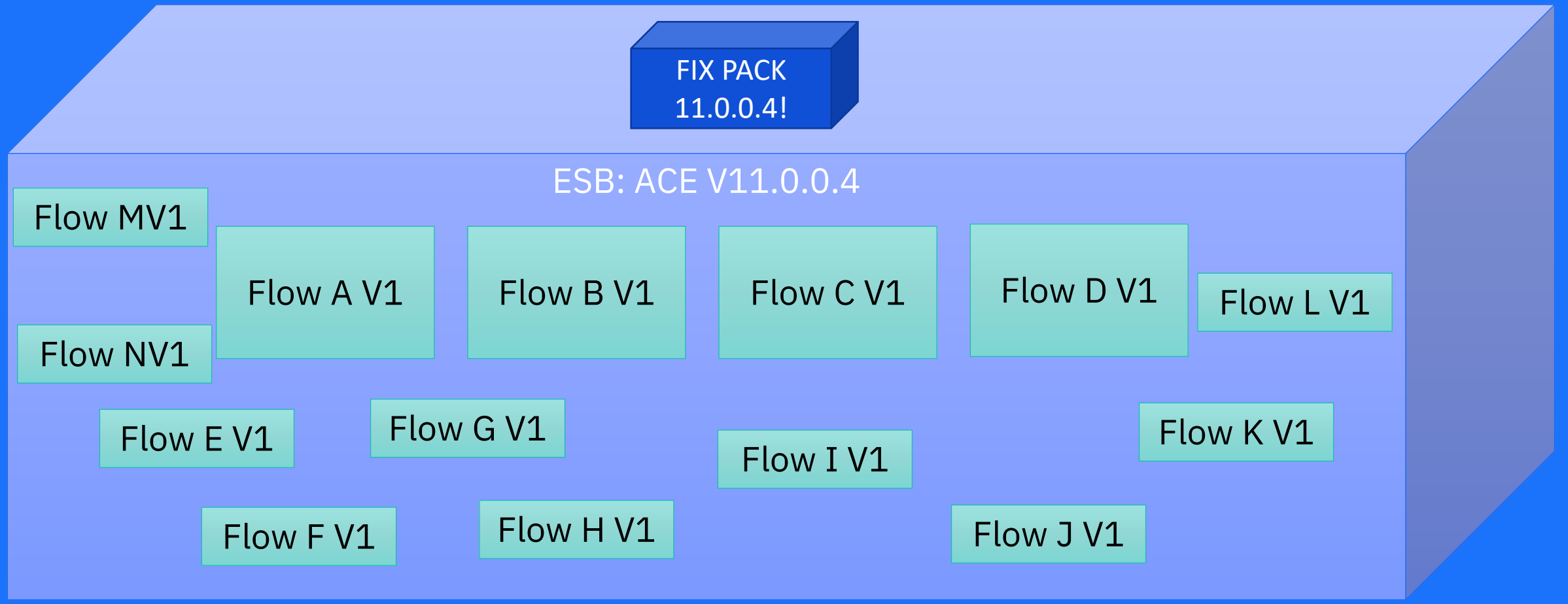
Importance of Independence



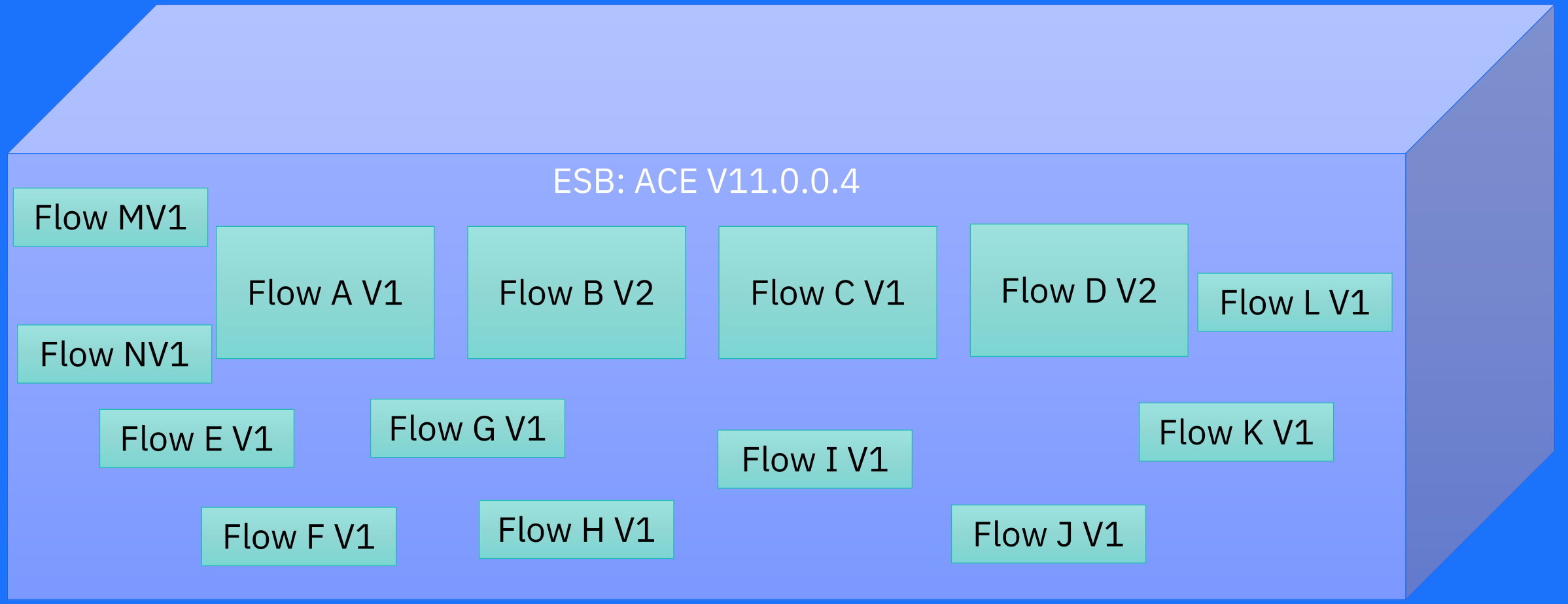
Importance of Independence



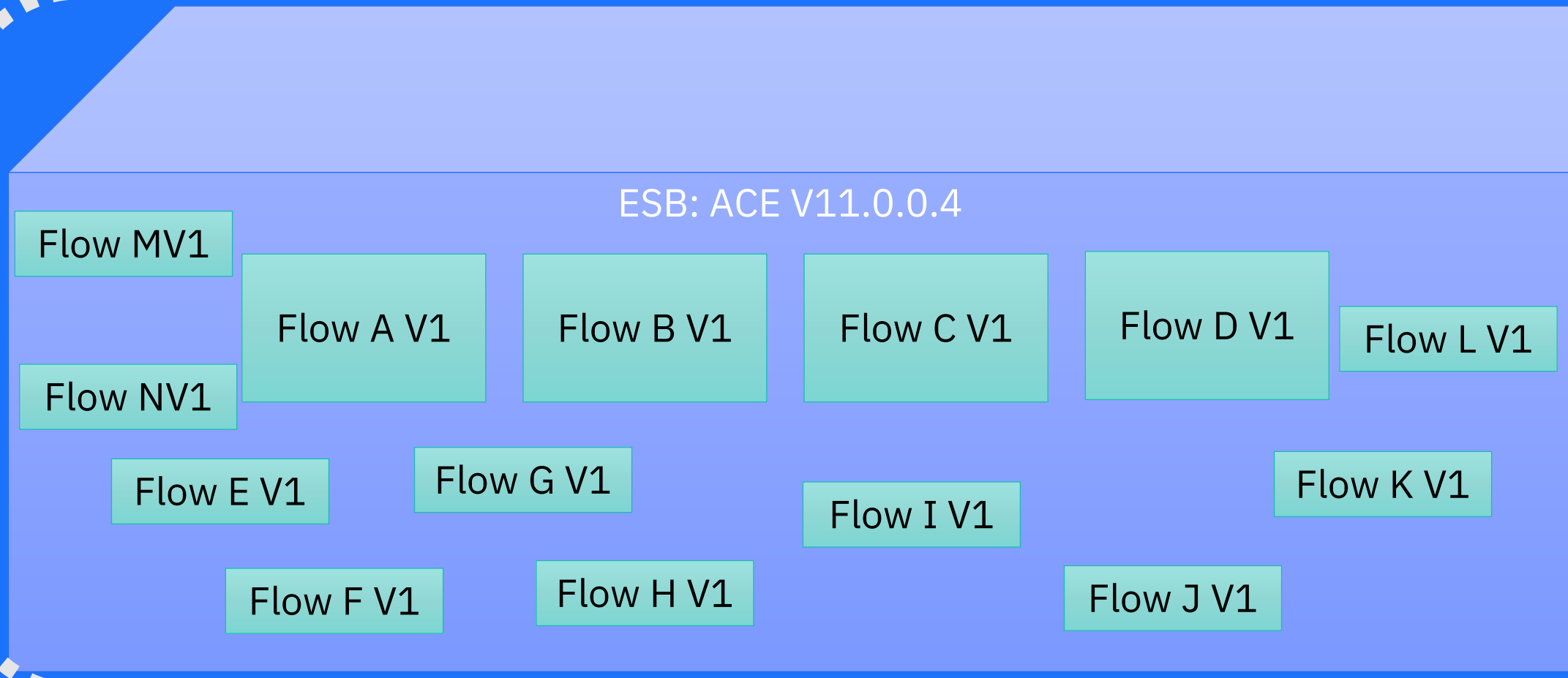
Importance of Independence



Importance of Independence

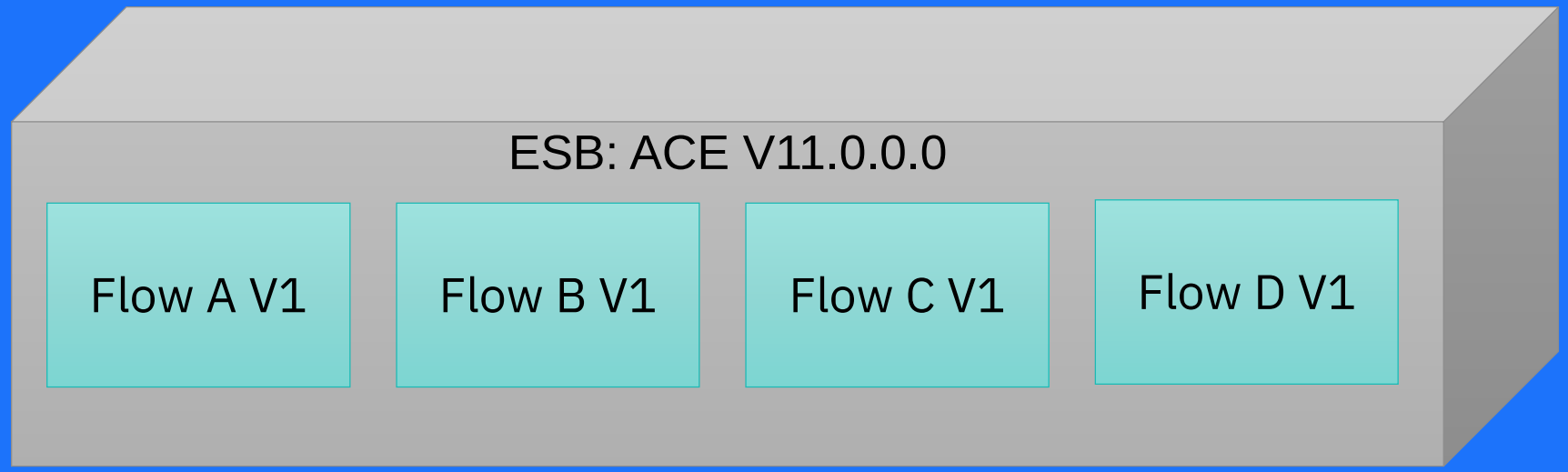


Importance of Independence

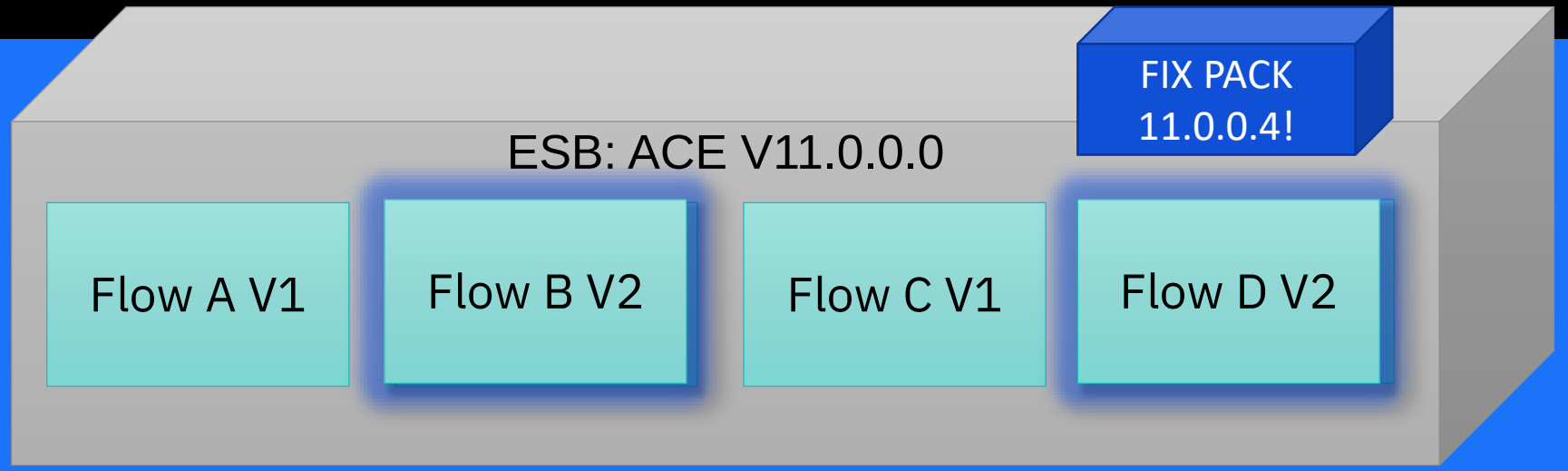


(Retro) Test Boundary

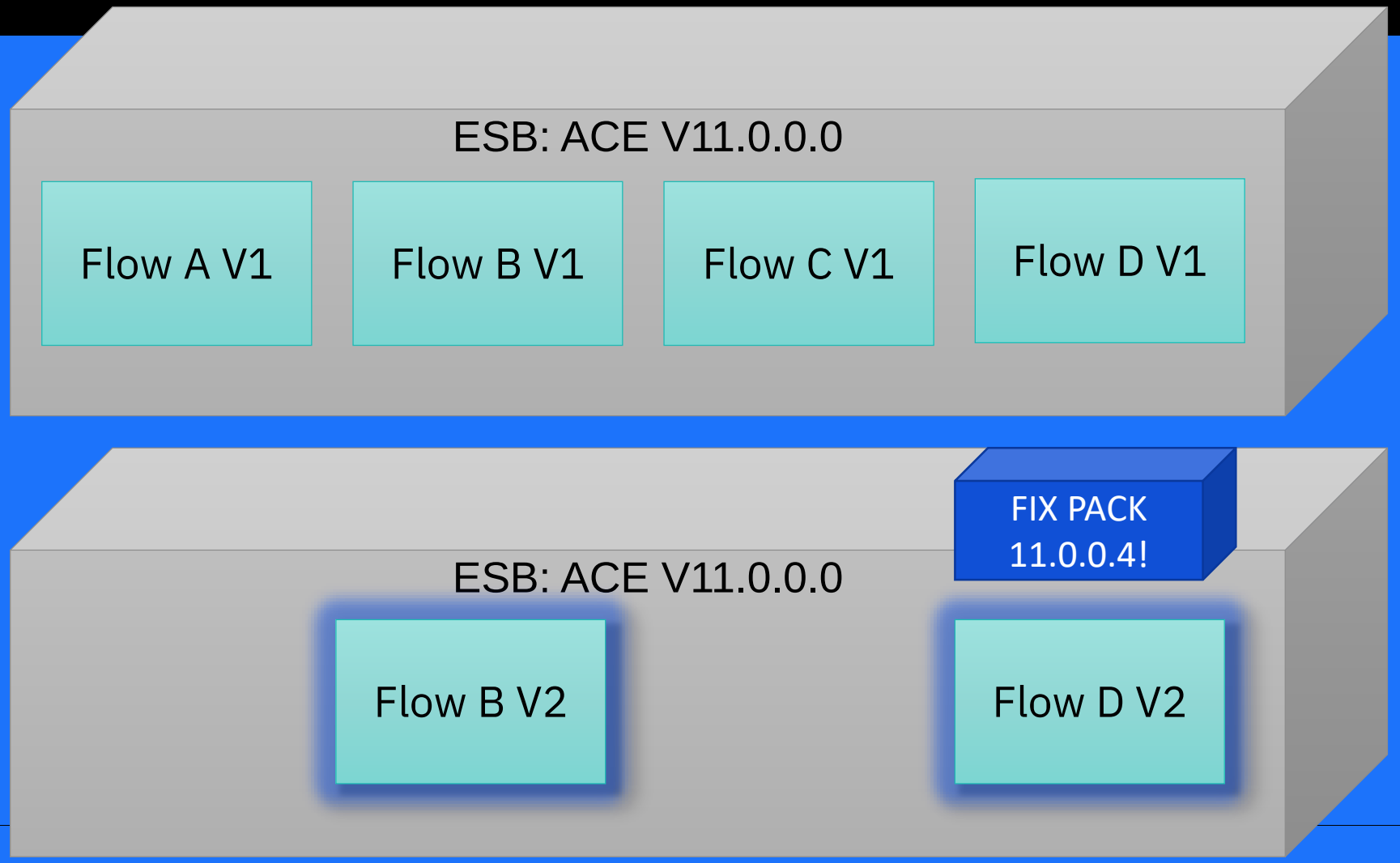
Importance of Independence



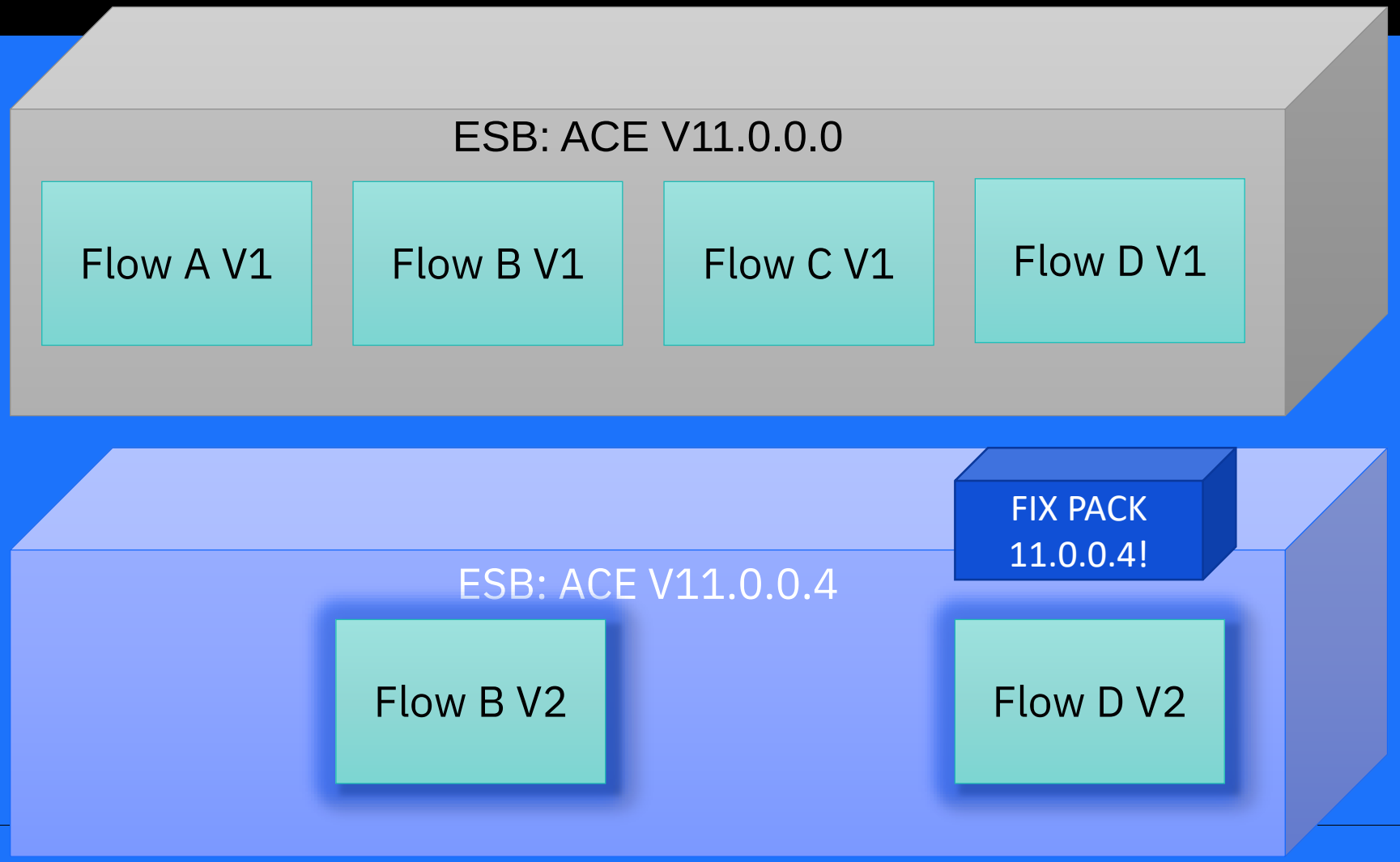
Importance of Independence



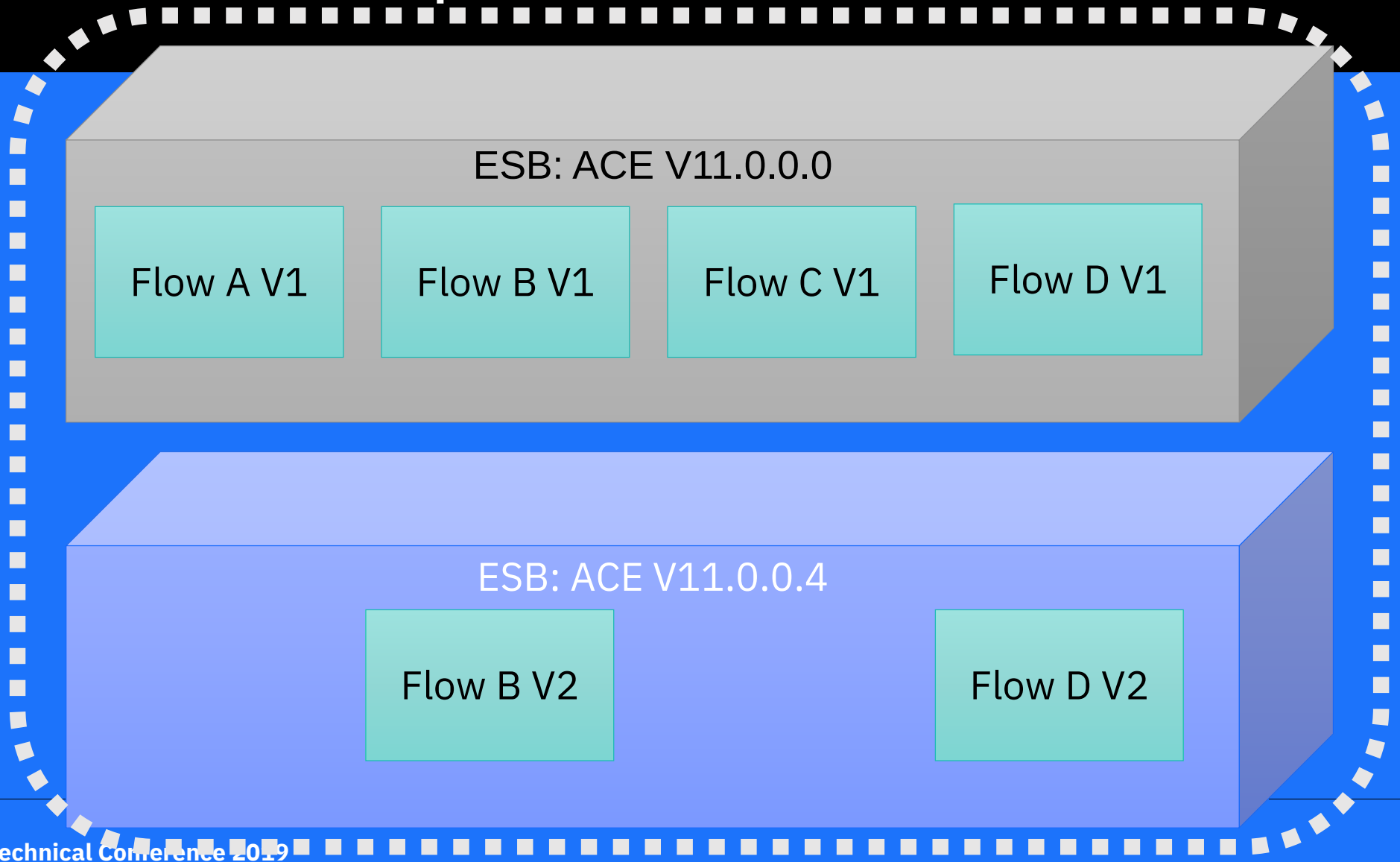
Importance of Independence



Importance of Independence

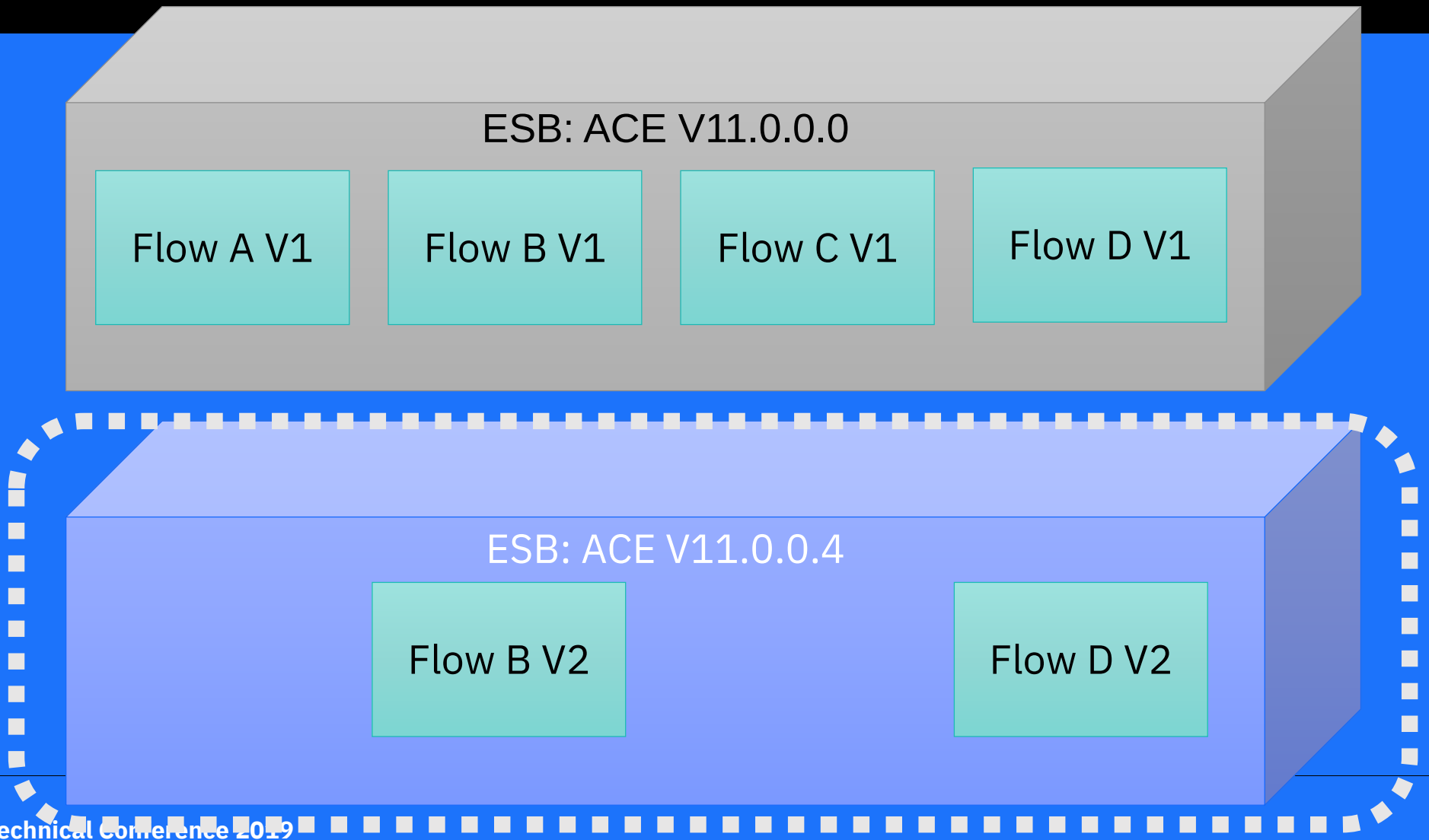


Importance of Independence

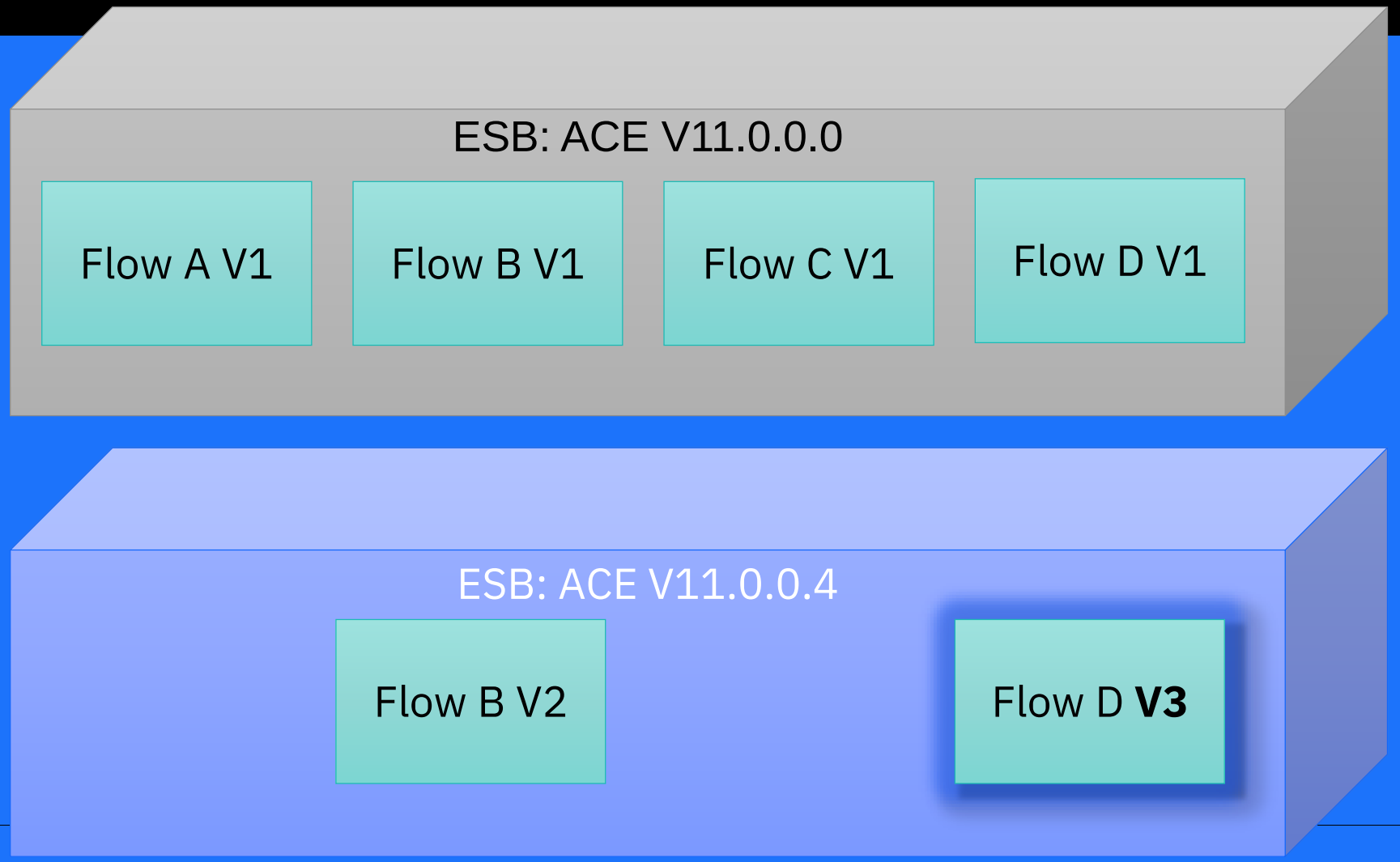


(Retro) Test Boundary??

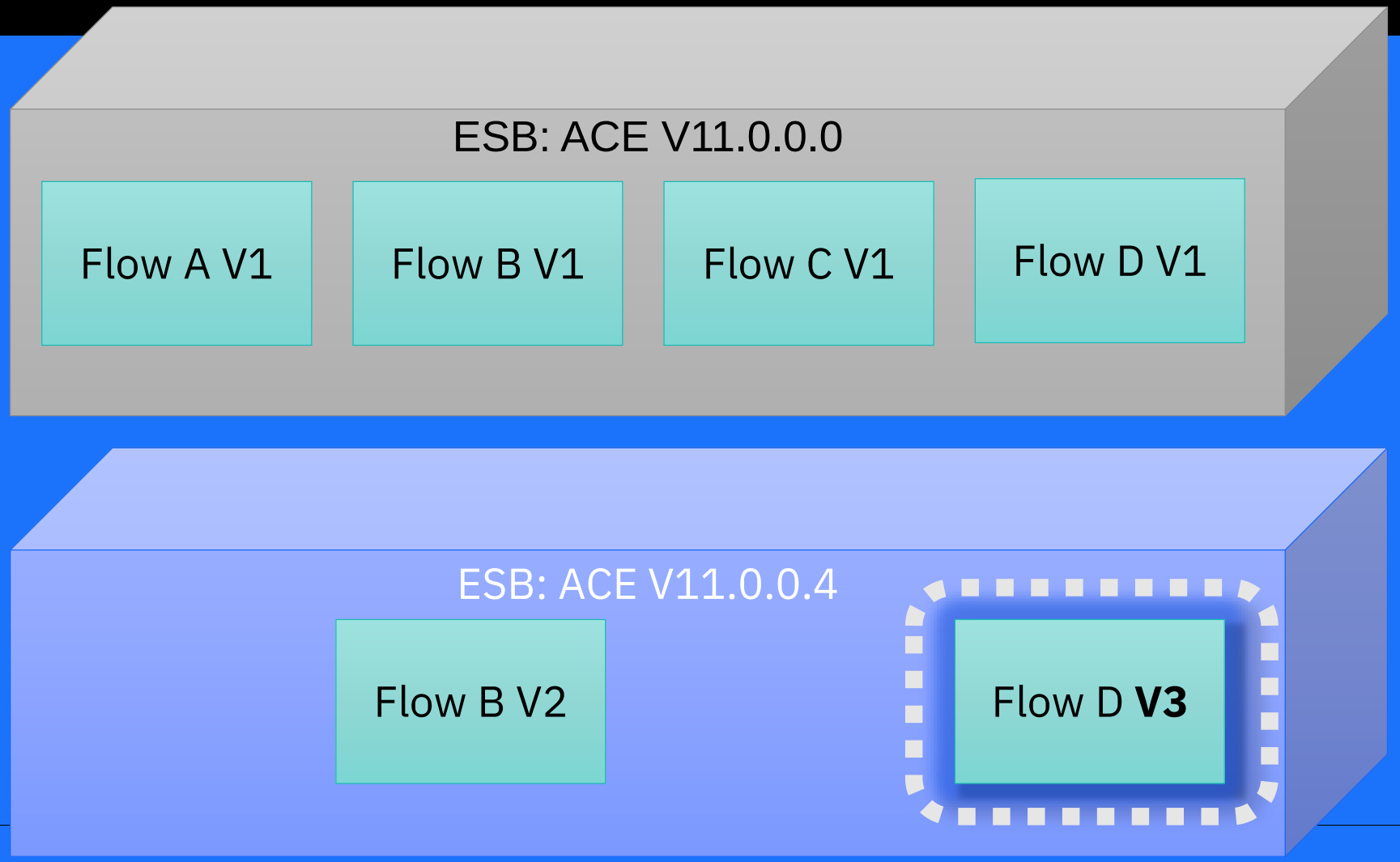
Importance of Independence



Importance of Independence

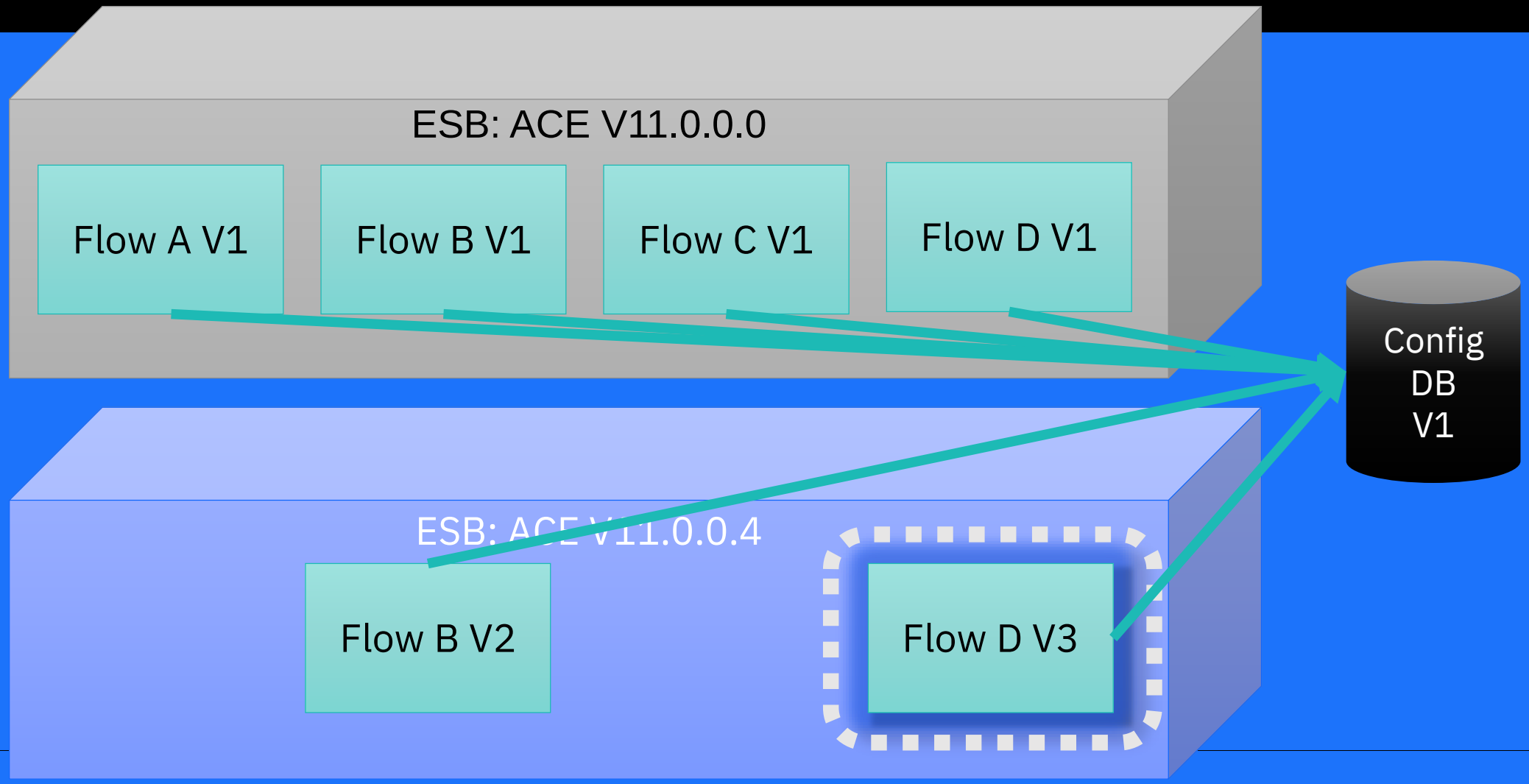


Importance of Independence



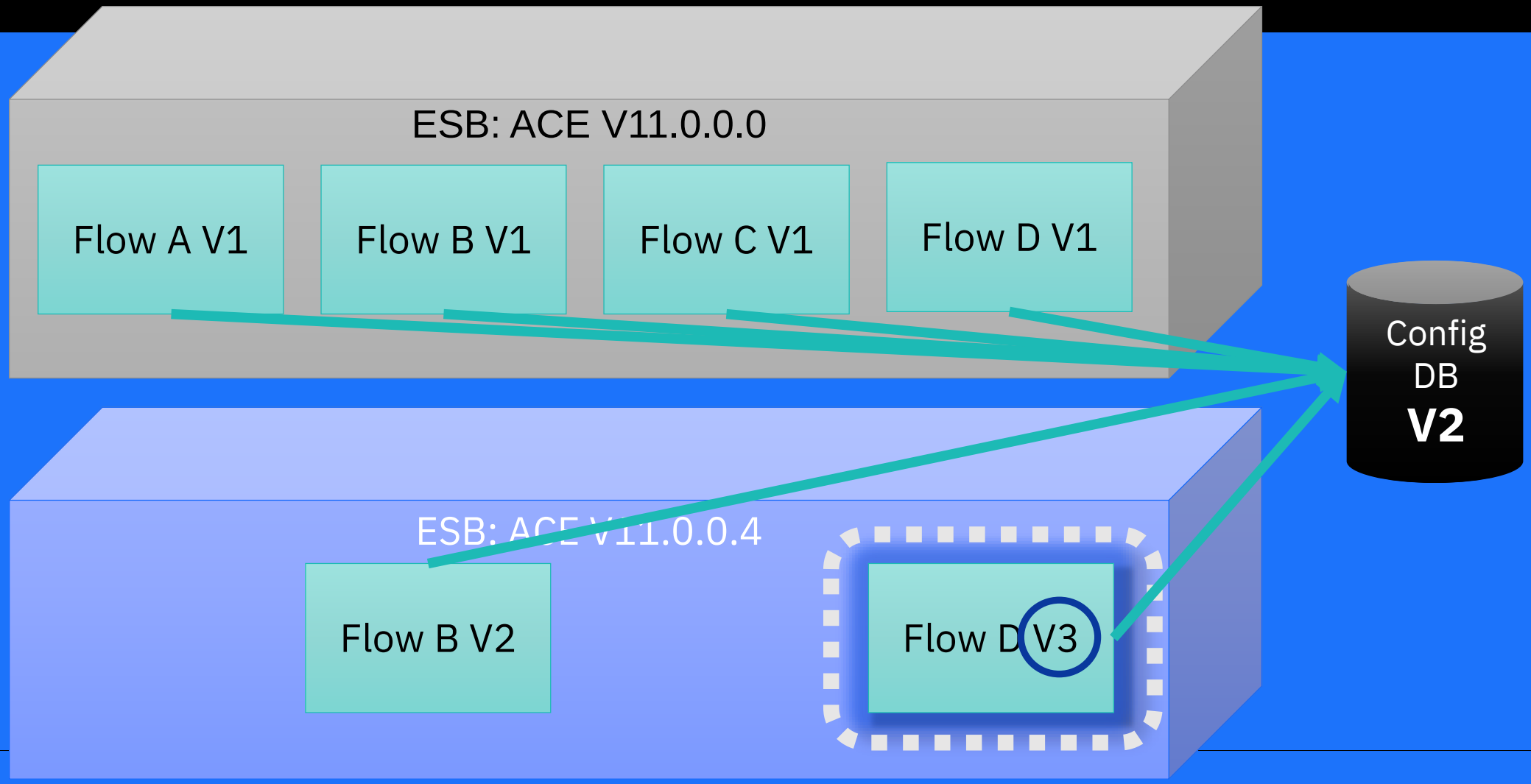
(Retro) Test Boundary

Importance of Independence



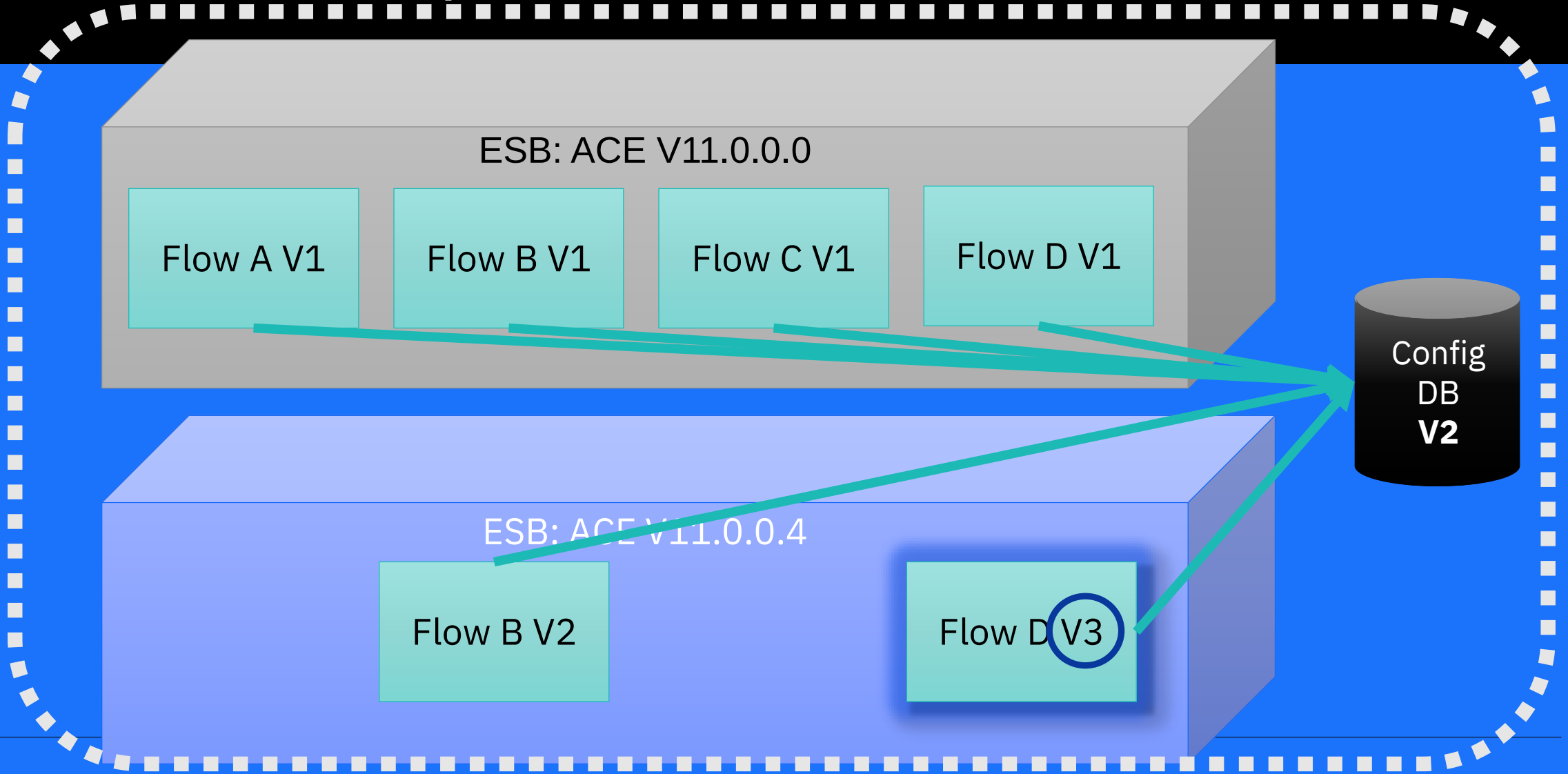
(Retro) Test Boundary

Importance of Independence



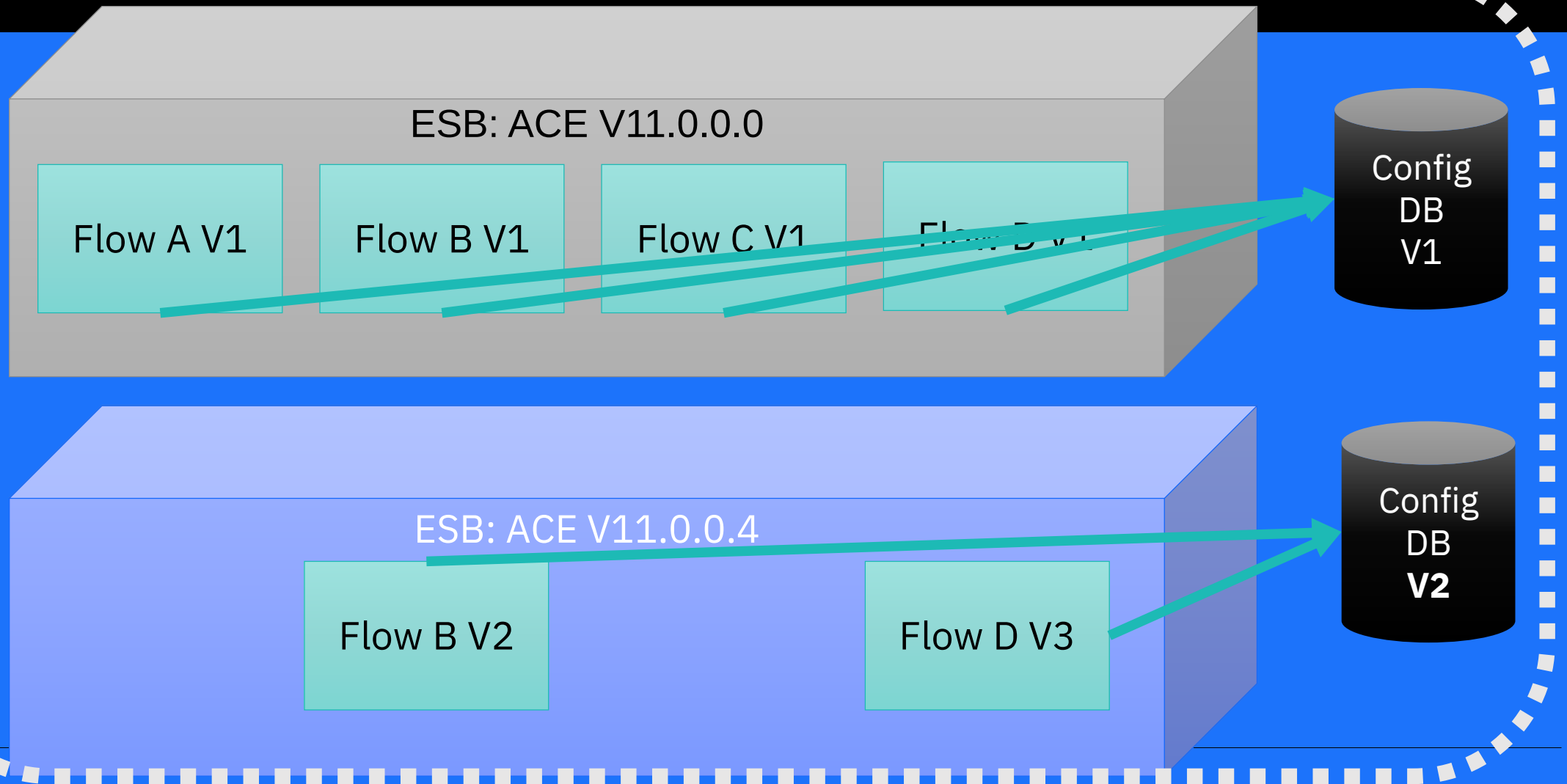
(Retro) Test Boundary

Importance of Independence



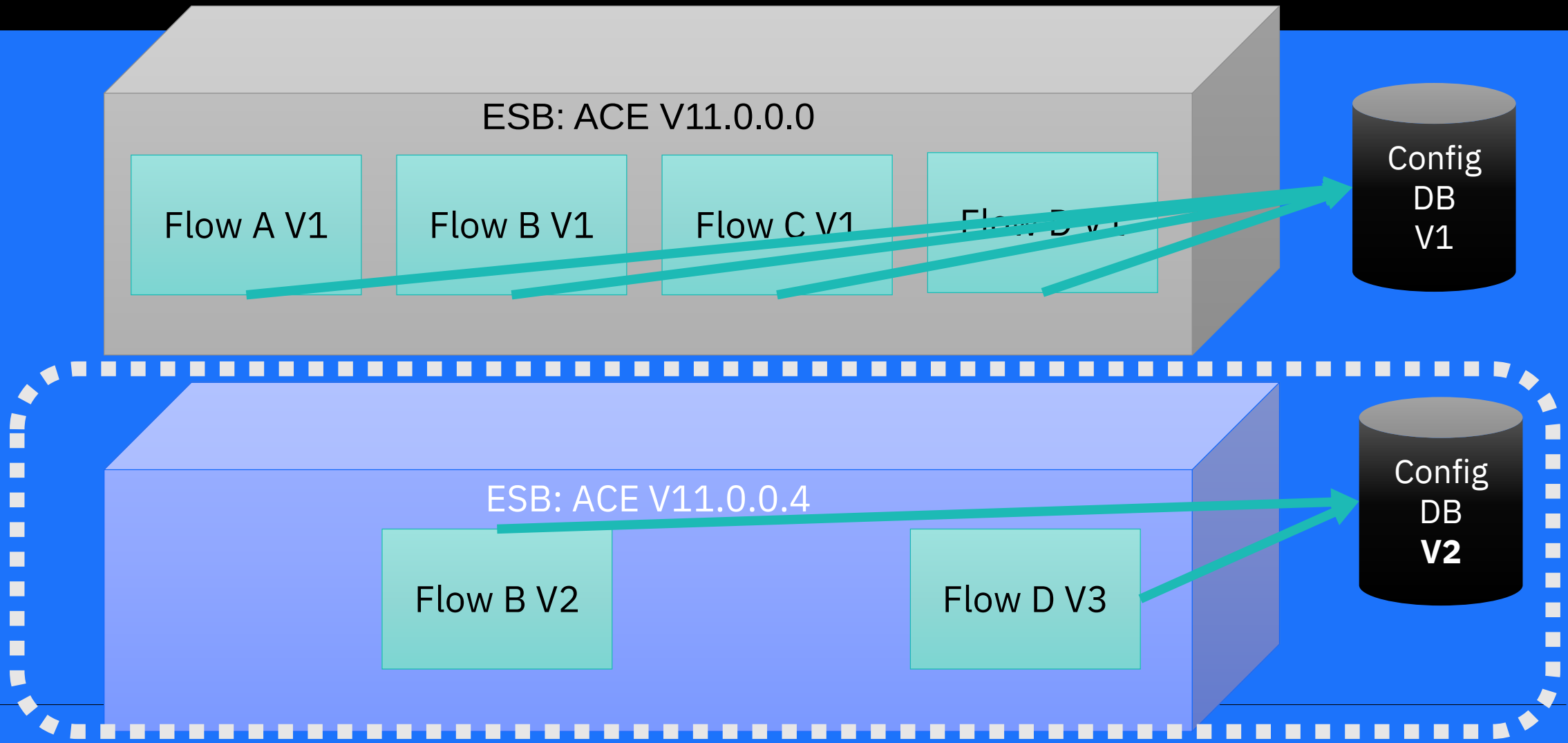
(Retro) Test Boundary

Importance of Independence



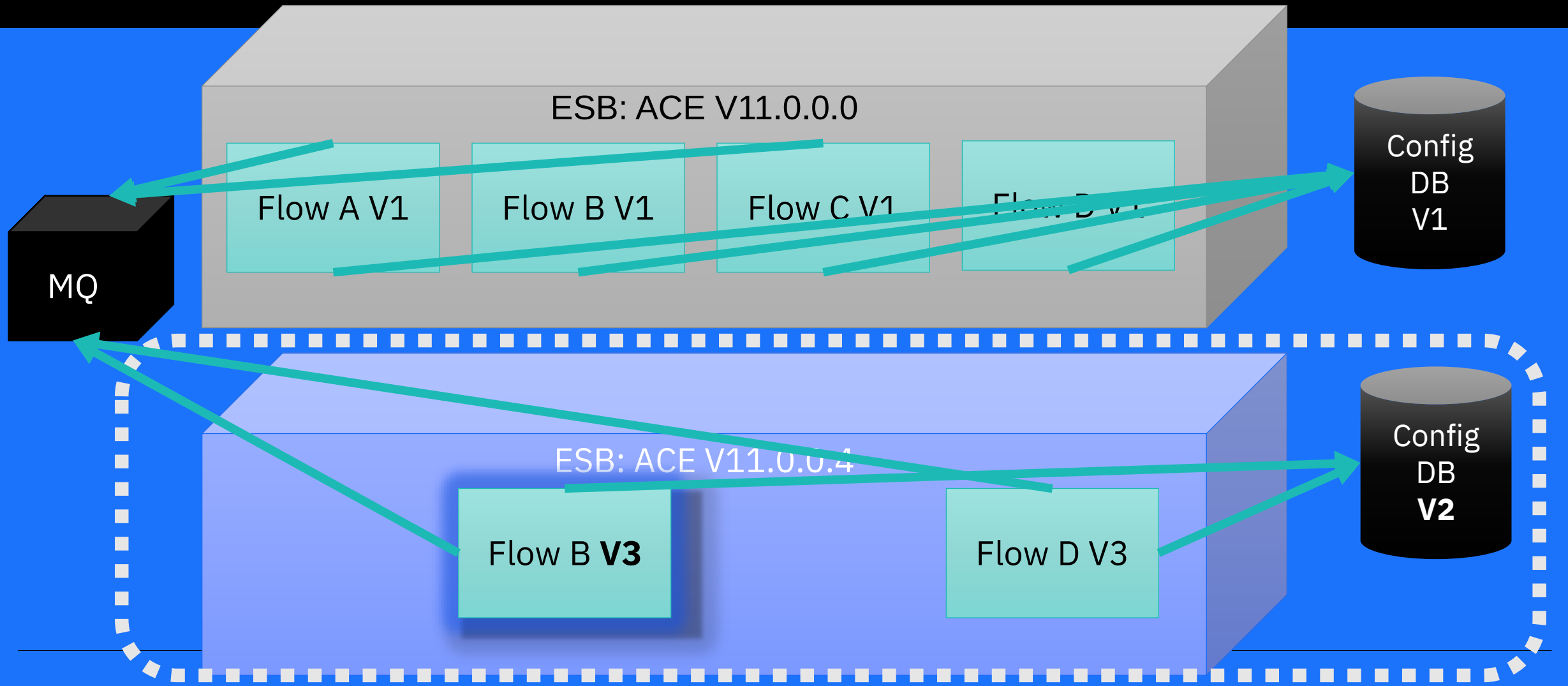
(Retro) Test Boundary

Importance of Independence

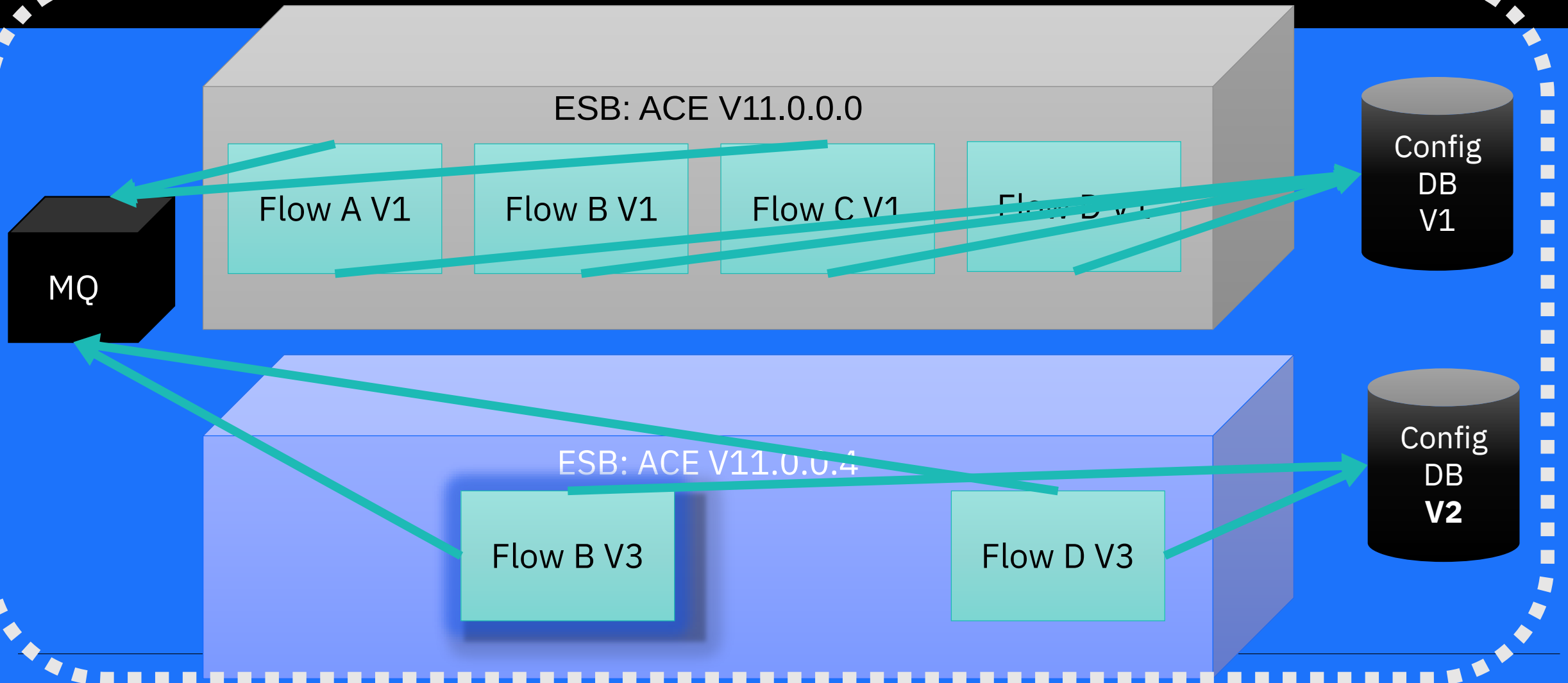


(Retro) Test Boundary

Importance of Independence

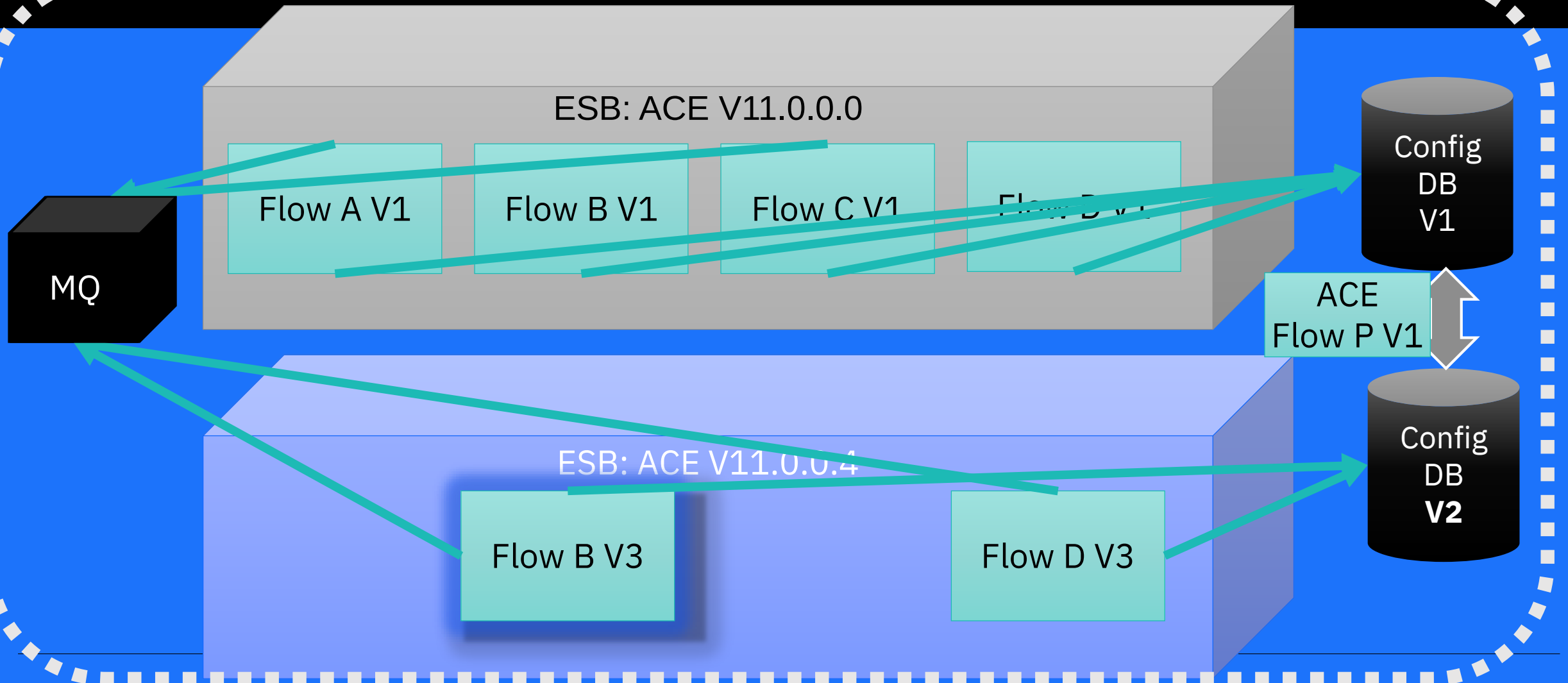


Importance of Independence



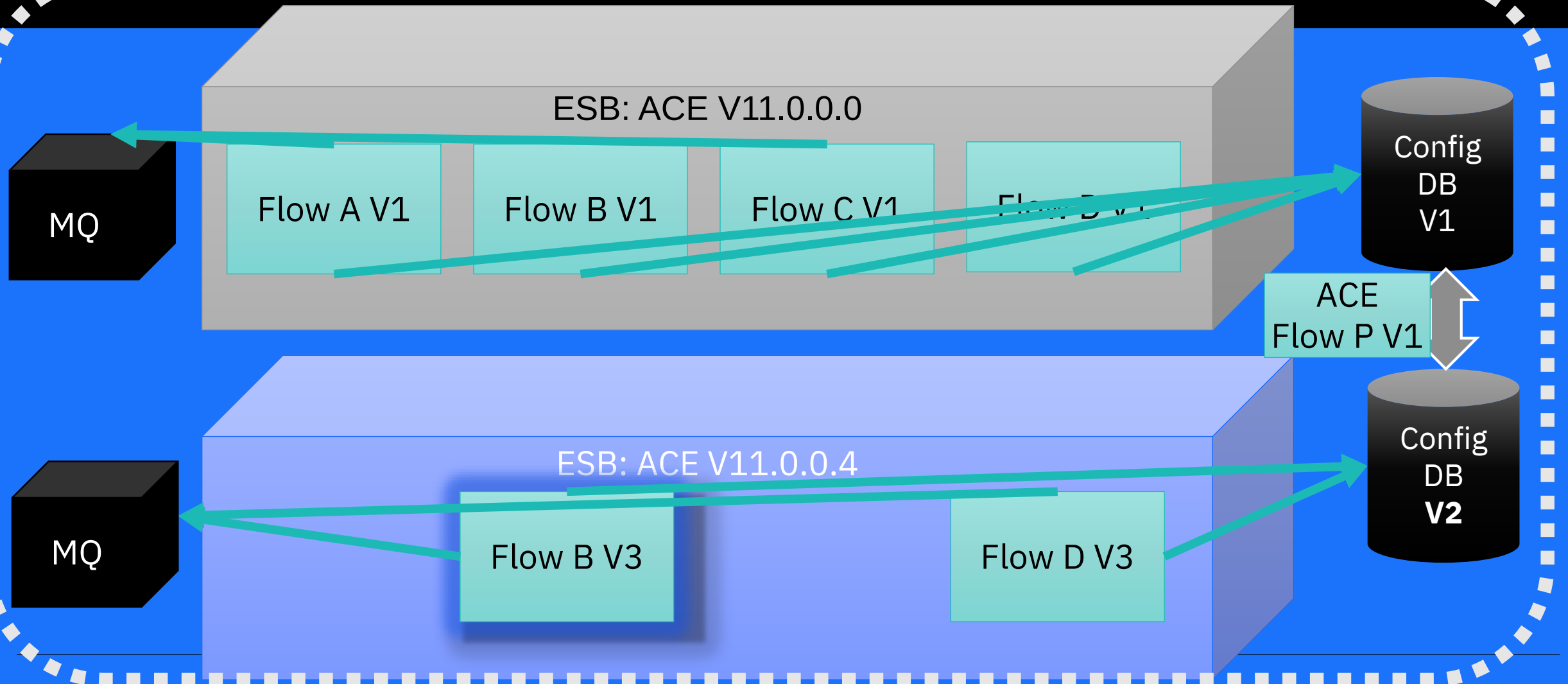
(Retro) Test Boundary

Importance of Independence



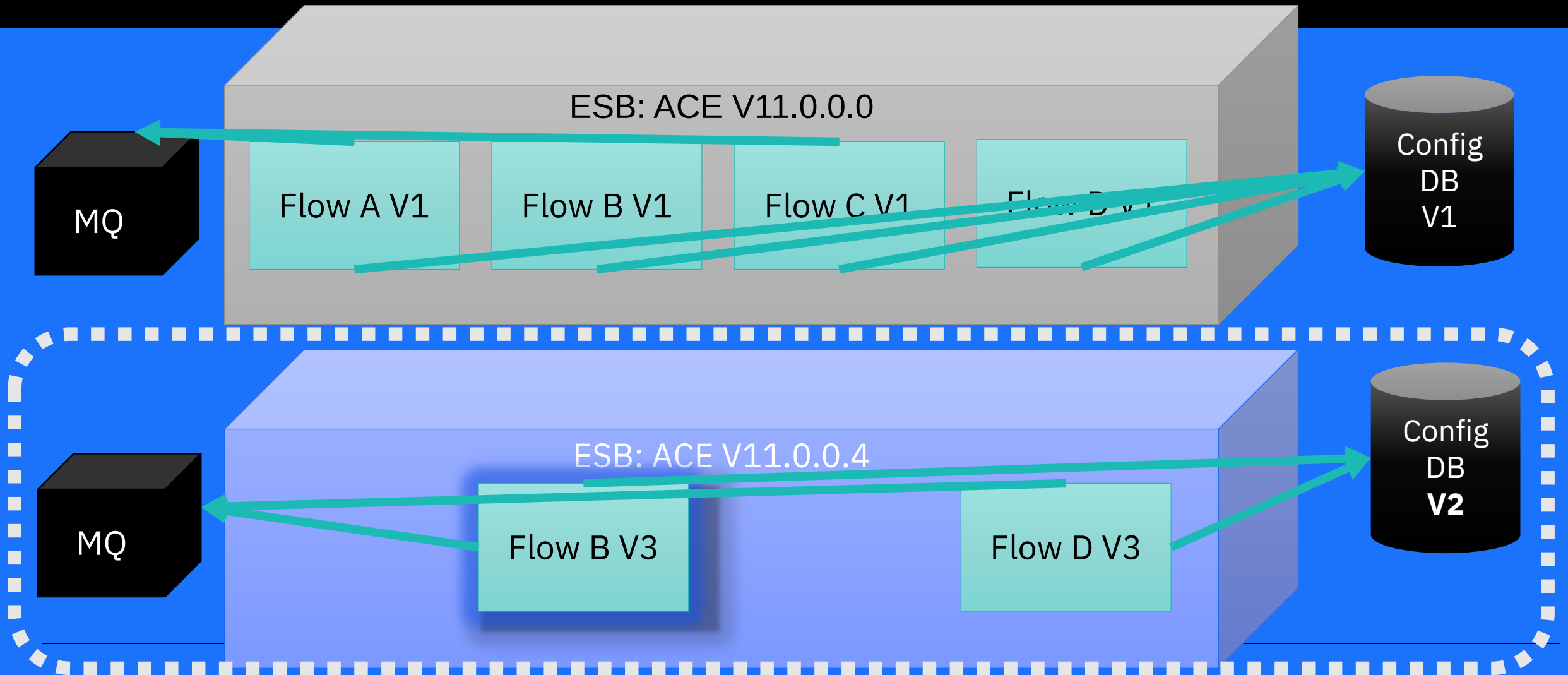
(Retro) Test Boundary

Importance of Independence



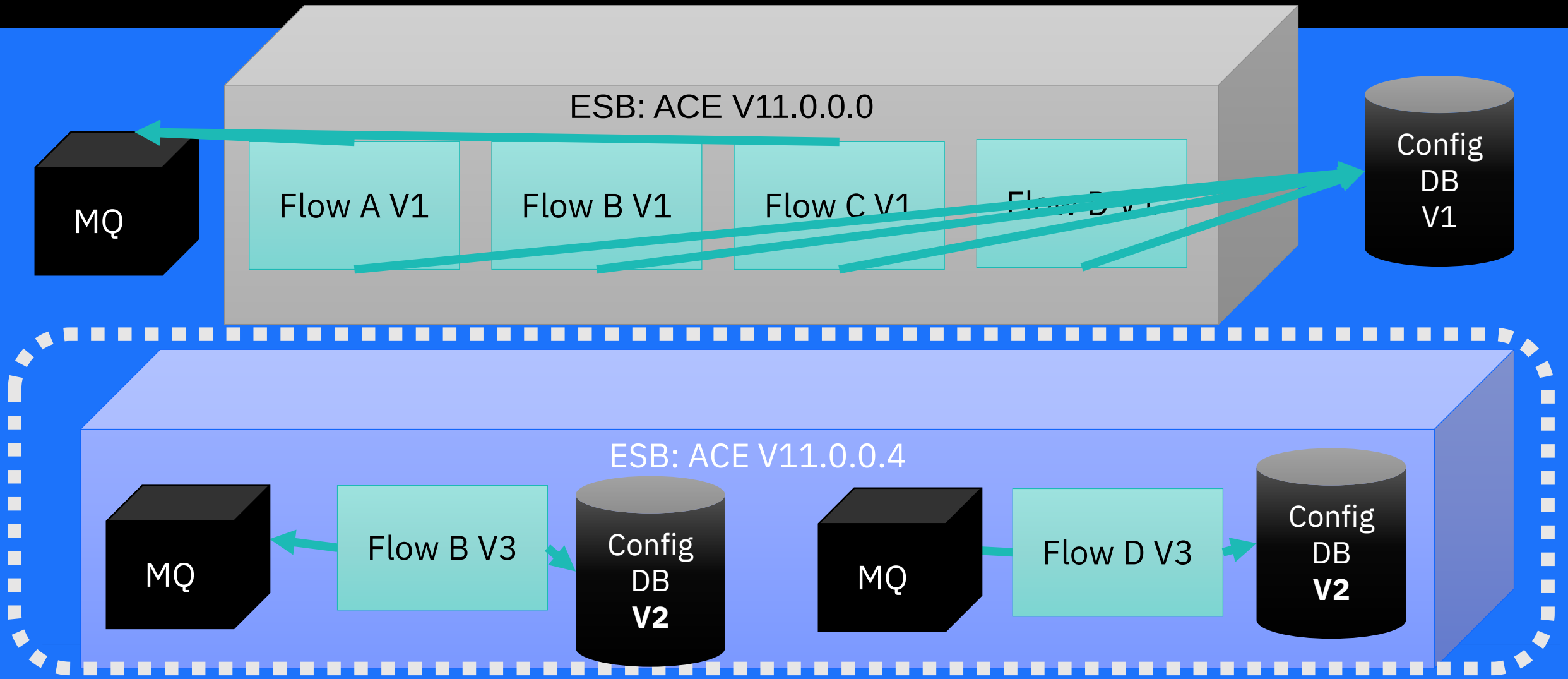
(Retro) Test Boundary

Importance of Independence



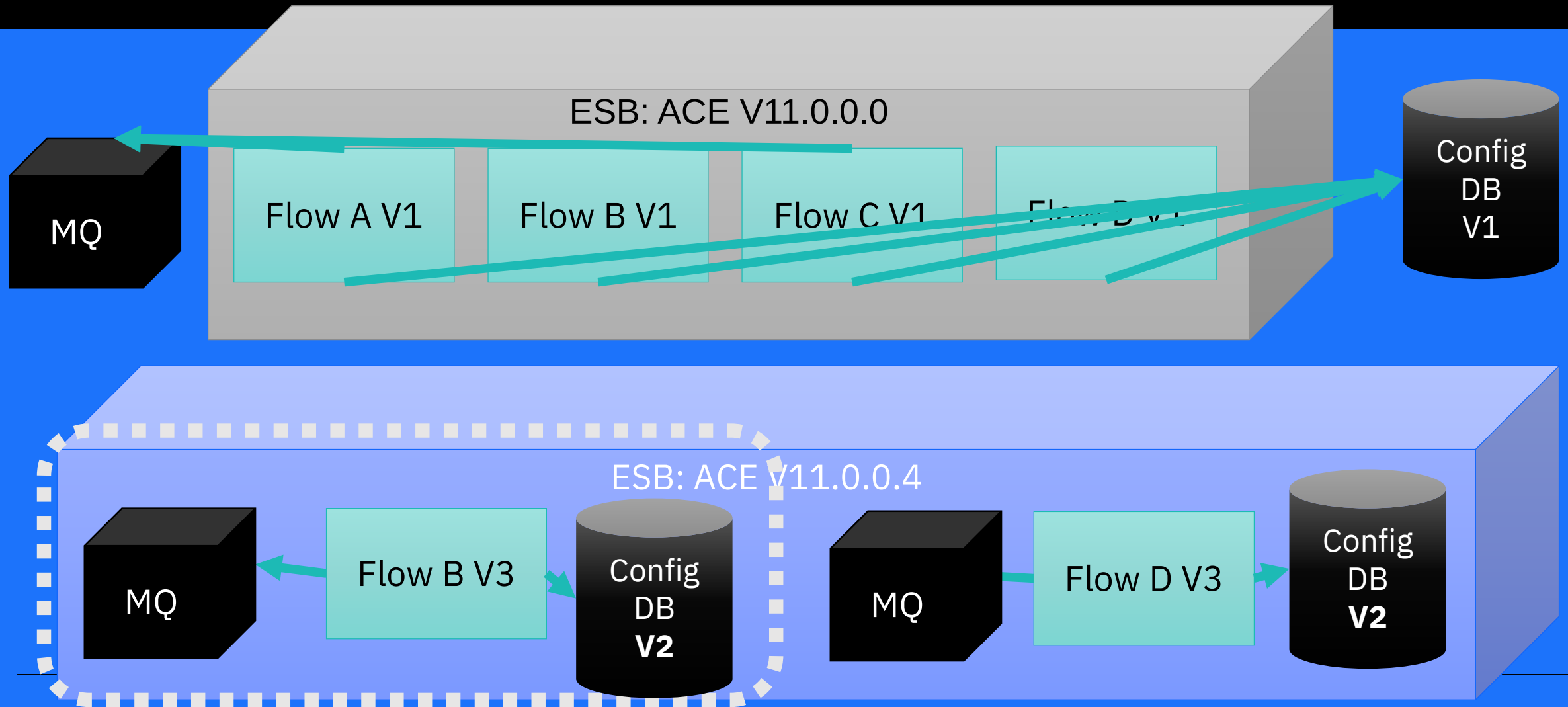
(Retro) Test Boundary

Importance of Independence



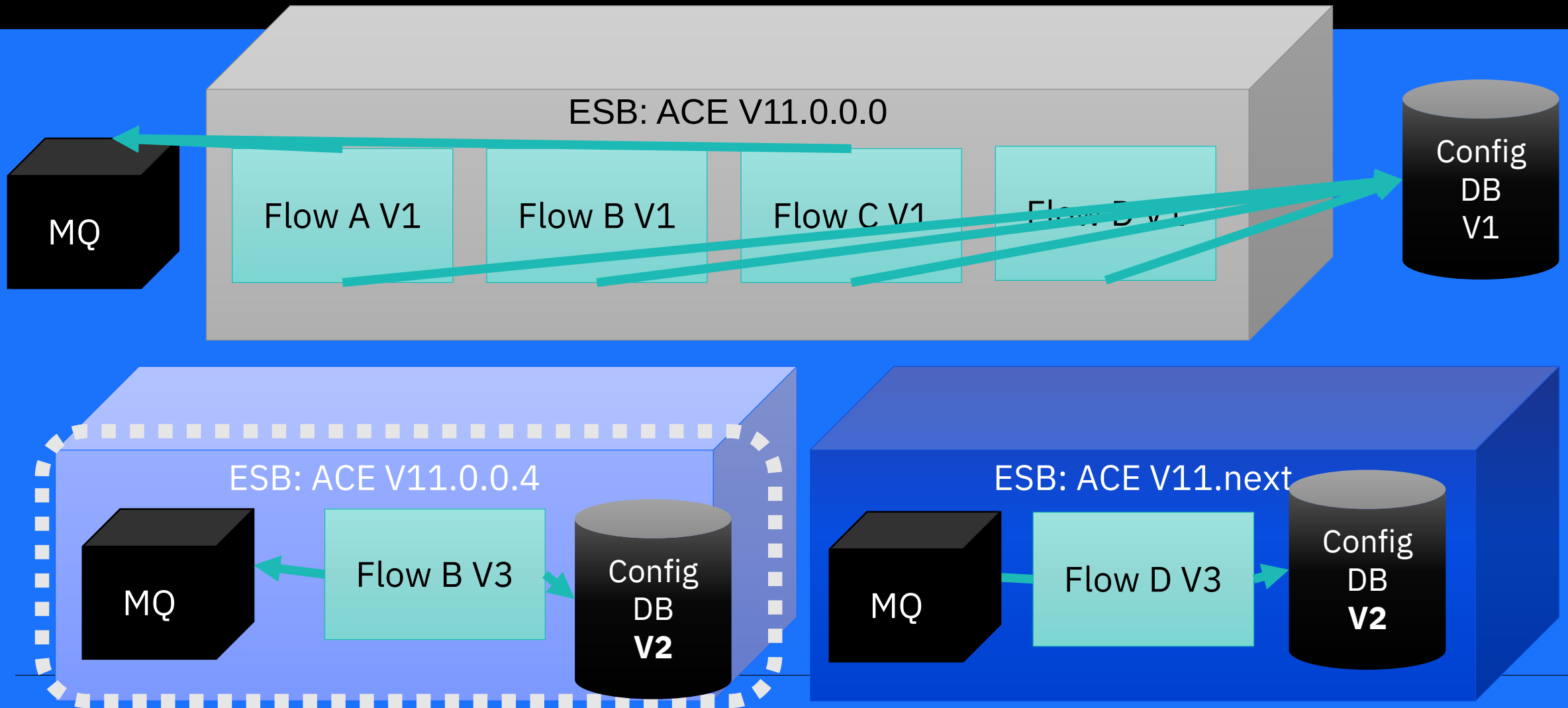
(Retro) Test Boundary

Importance of Independence



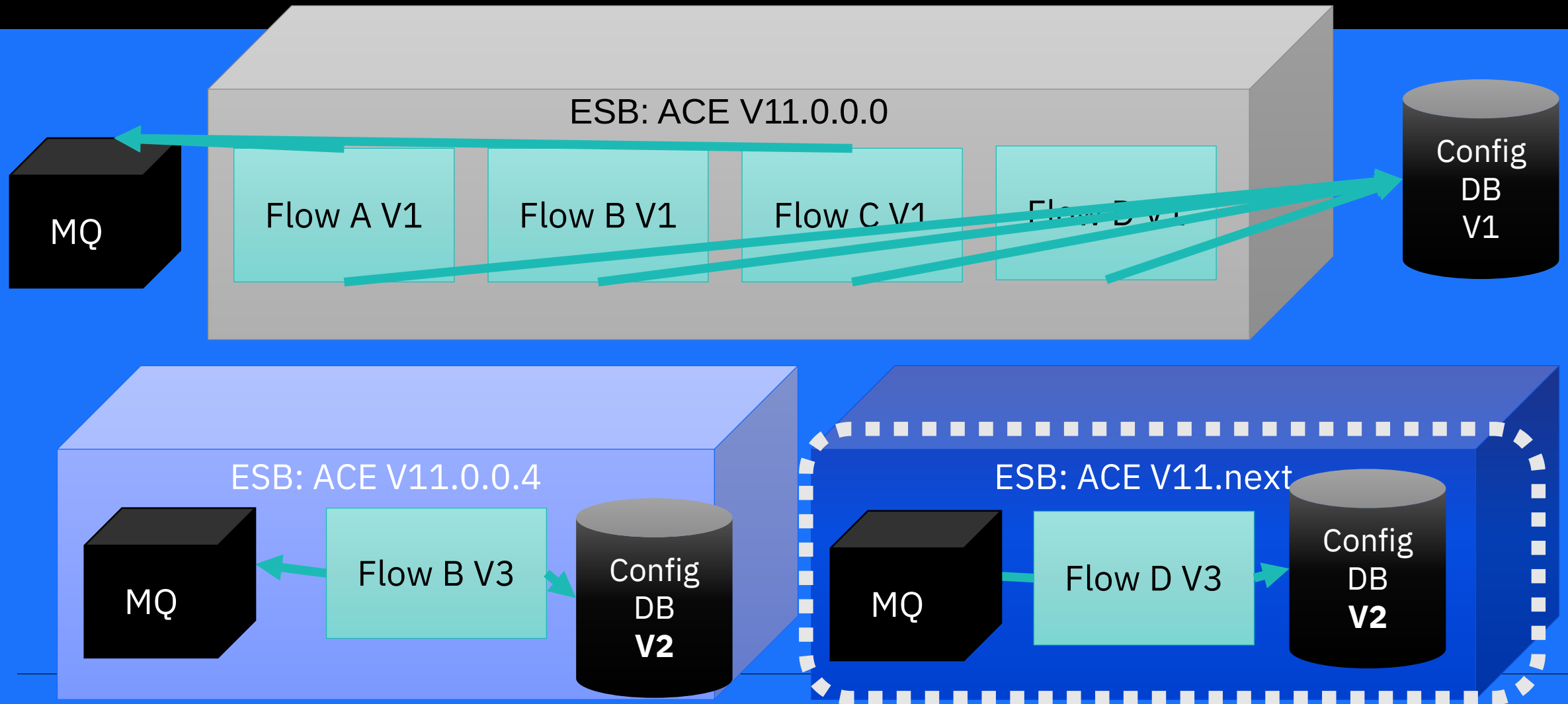
(Retro) Test Boundary

Importance of Independence



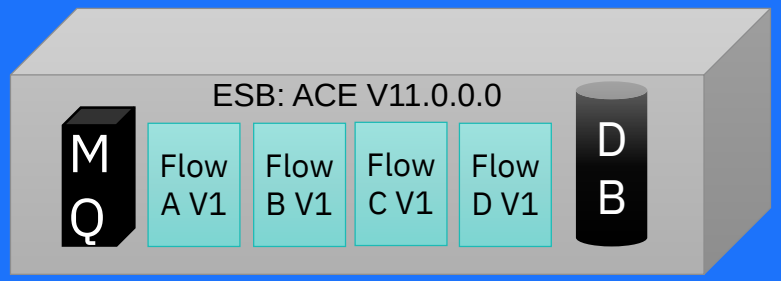
(Retro) Test Boundary

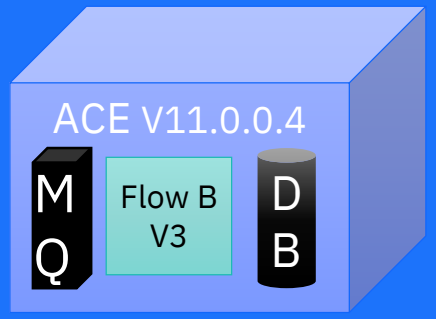
Importance of Independence

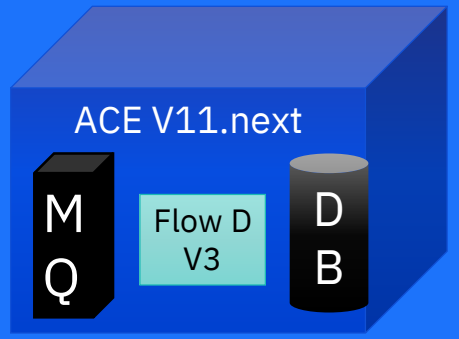


(Retro) Test Boundary

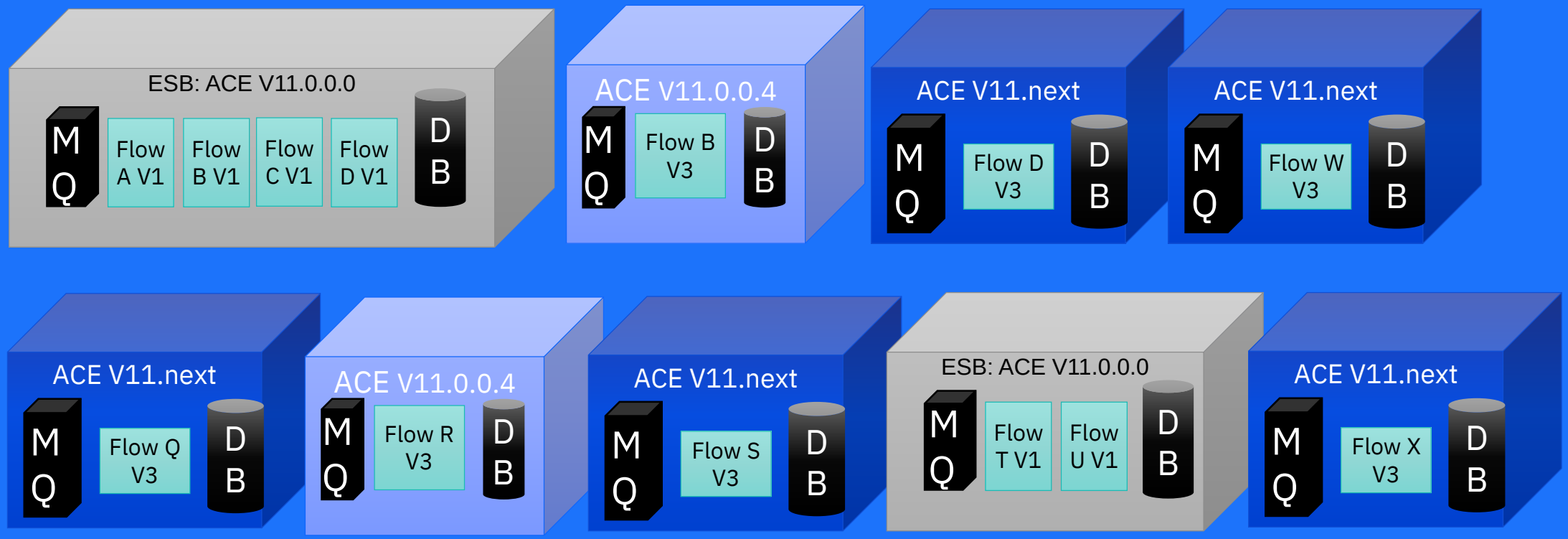
What about importance of size?



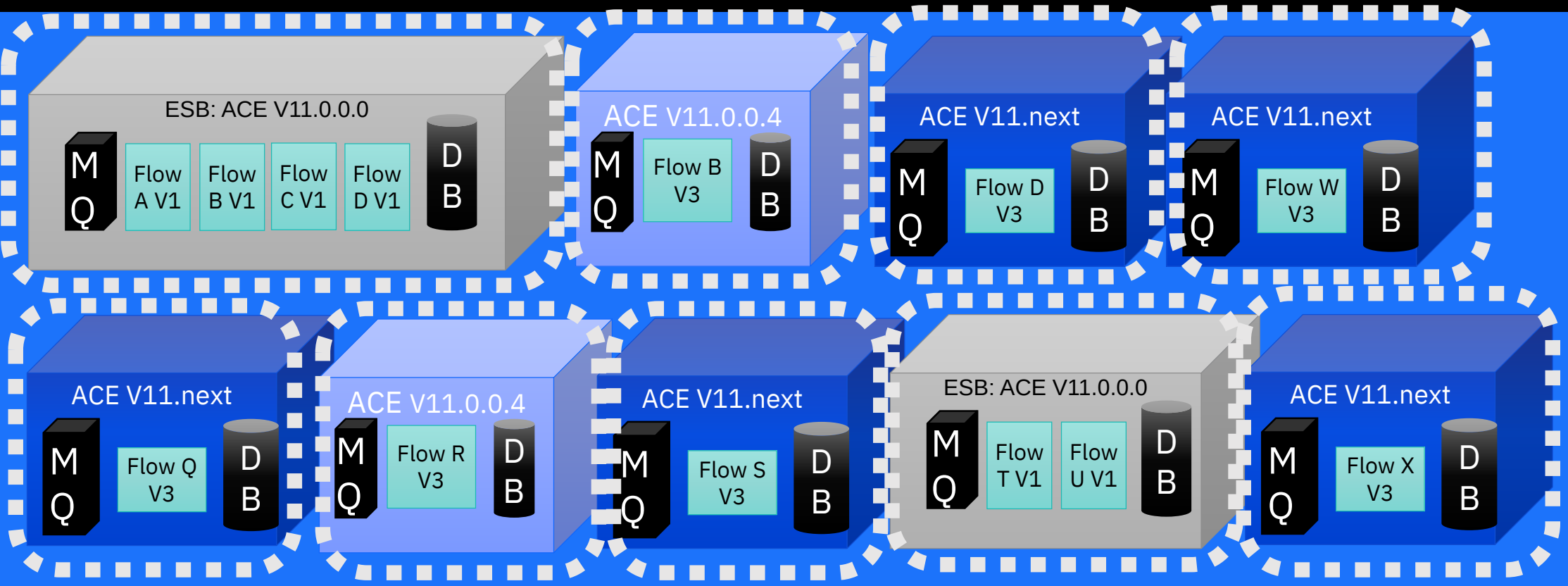




What about importance of size?



What about importance of size?



(Retro) Test Boundary

Granularity

What is your deployment unit?

One Operation?

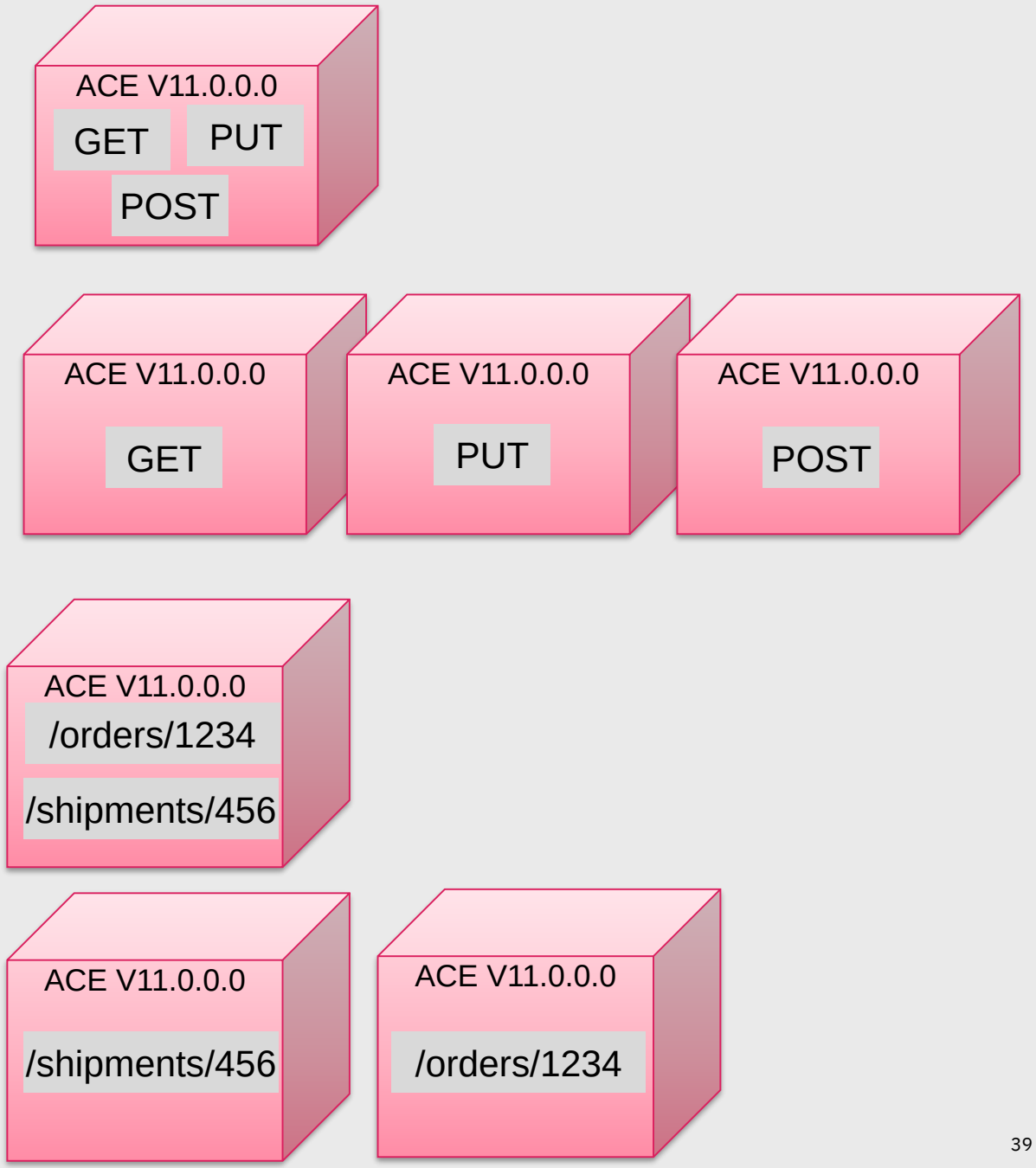
One service / API e.g. Swagger / WSDL?

A group of Integrations co-located on a server?

What will change (Deploy)?

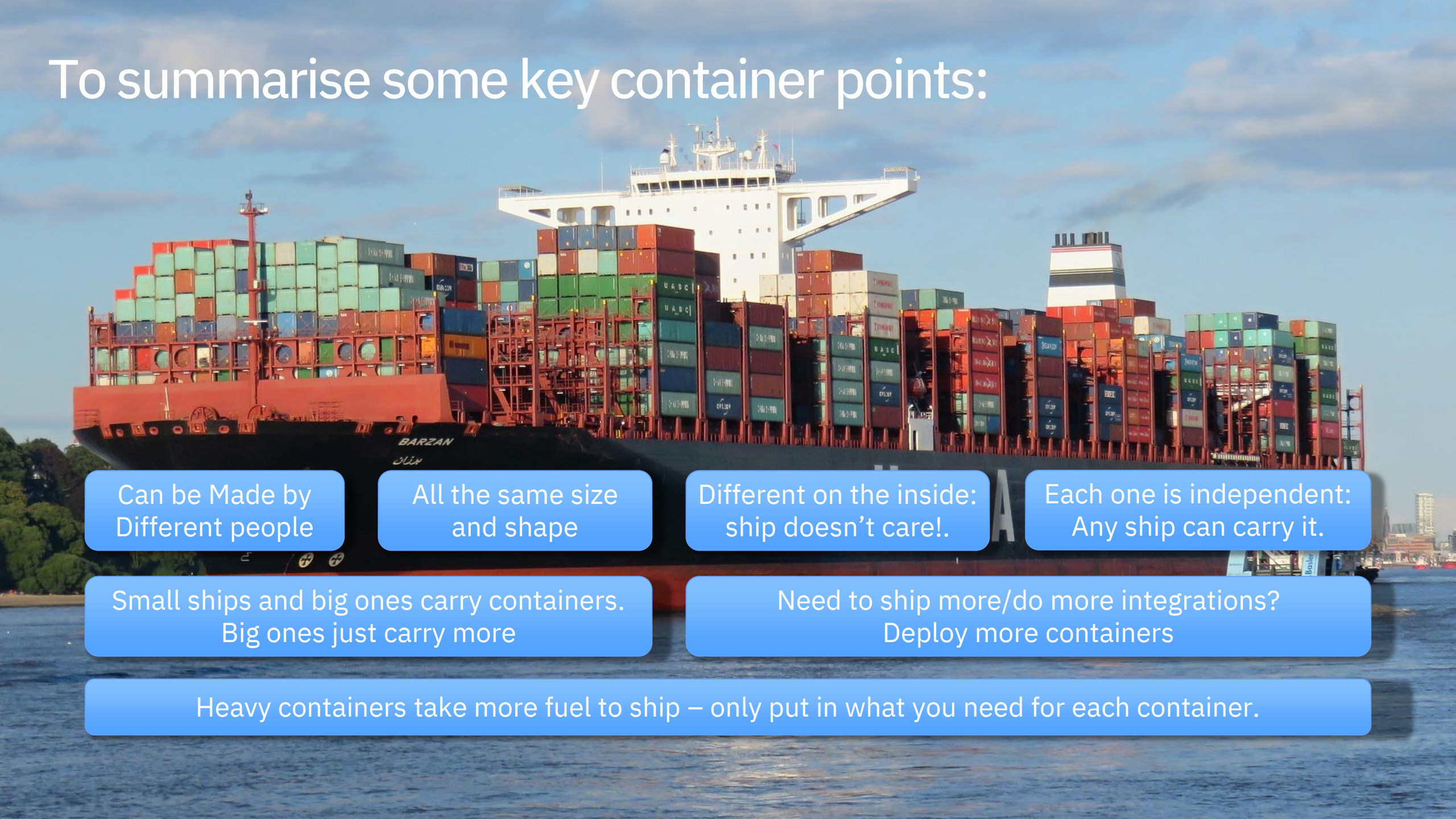
Where is your testing boundary?

What takes resources (Scale)?



Containers: History and Philosophy

To summarise some key container points:



Can be Made by Different people

All the same size and shape

Different on the inside: ship doesn't care!.

Each one is independent: Any ship can carry it.

Small ships and big ones carry containers. Big ones just carry more

Need to ship more/do more integrations? Deploy more containers

Heavy containers take more fuel to ship – only put in what you need for each container.

Agile Integration Architecture

Modernizing integration to enable ***business*** *agility*

Fine grained deployment

Architecture & design

→ Improve build independence and production velocity (***deployment*** *agility*)

Decentralized Ownership

People & Process

→ Accelerate agility and innovation (***development*** *agility*)

Cloud native infrastructure

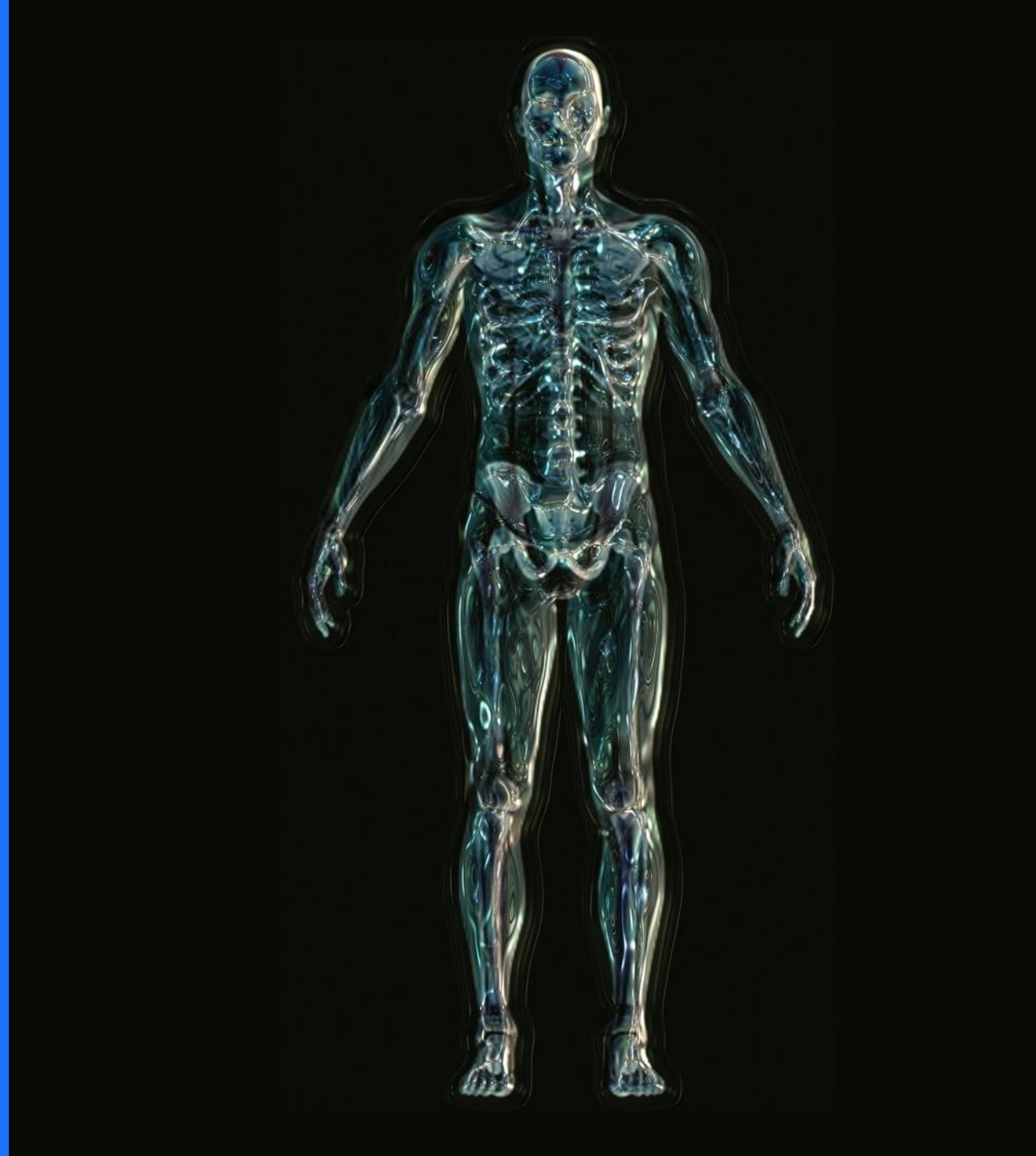
Infrastructure & Technology

→ Dynamic scalability and inherent resilience (***operational*** *agility*)

Anatomy of a container:

Container Image

- Anything else
- Helpers and Utilities
- Static Configuration
- Integrations (BARs)
- App Connect
- Operating System



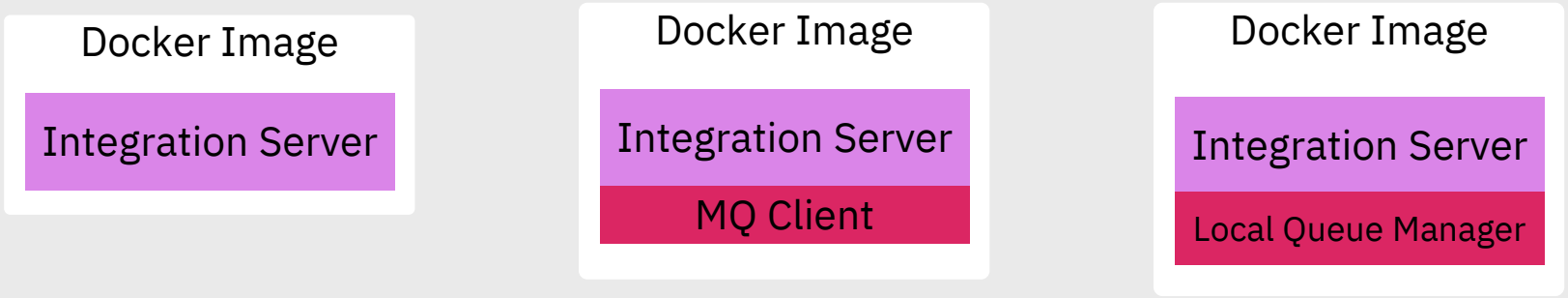
Anatomy of a container:

Container Image

- Anything else
- Helpers and Utilities
- Static Configuration
- Integrations (BARs)
- MQ client or QM?
- App Connect
- Operating System

Image content tailored to each integration.

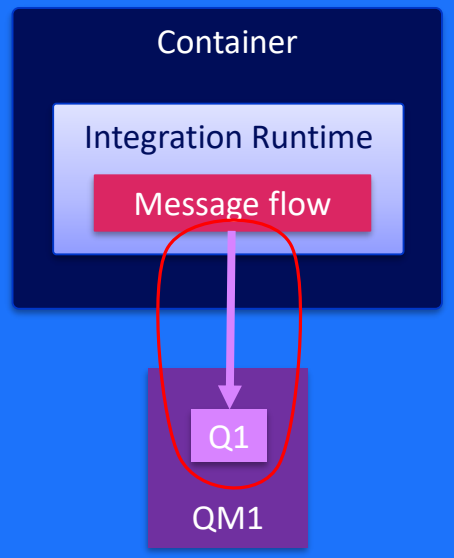
Different on the inside: ship doesn't care!.



MQ connectivity	Yes (HTTP API)	Yes (client binding)	Yes (server binding)
1PC for MQ	No	Yes	Yes
EDA nodes	No	No	Yes
2PC	No	No	Yes
Horizontally scalable	Yes	Yes	Yes (with loss of sequencing)
Persistent volume	Not required	Not required	Required (if durability desired)
Start up	Fast	Fast	Slower
Disk space	Smallest	Medium	Largest

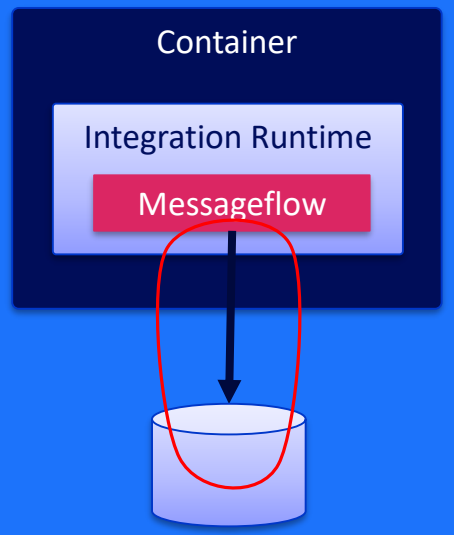
Why? Example: Single vs two phase commit

a) Single transaction with an MQ queue



- One-phase commit sufficient
- No local queue manager required

b) Single transaction with a database

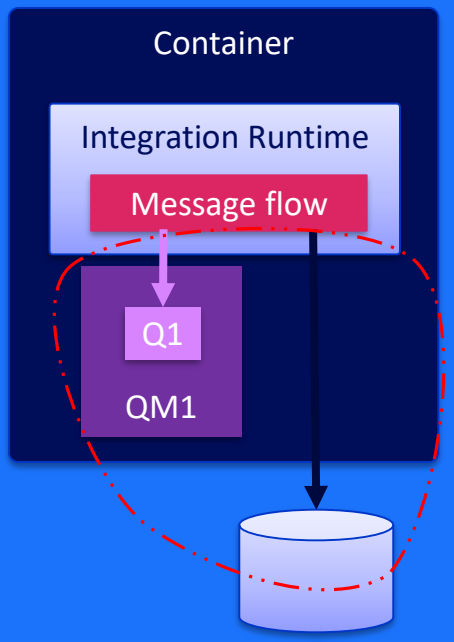


- One-phase commit sufficient
- No local queue manager required

c) Separate transactions to a database and a queue

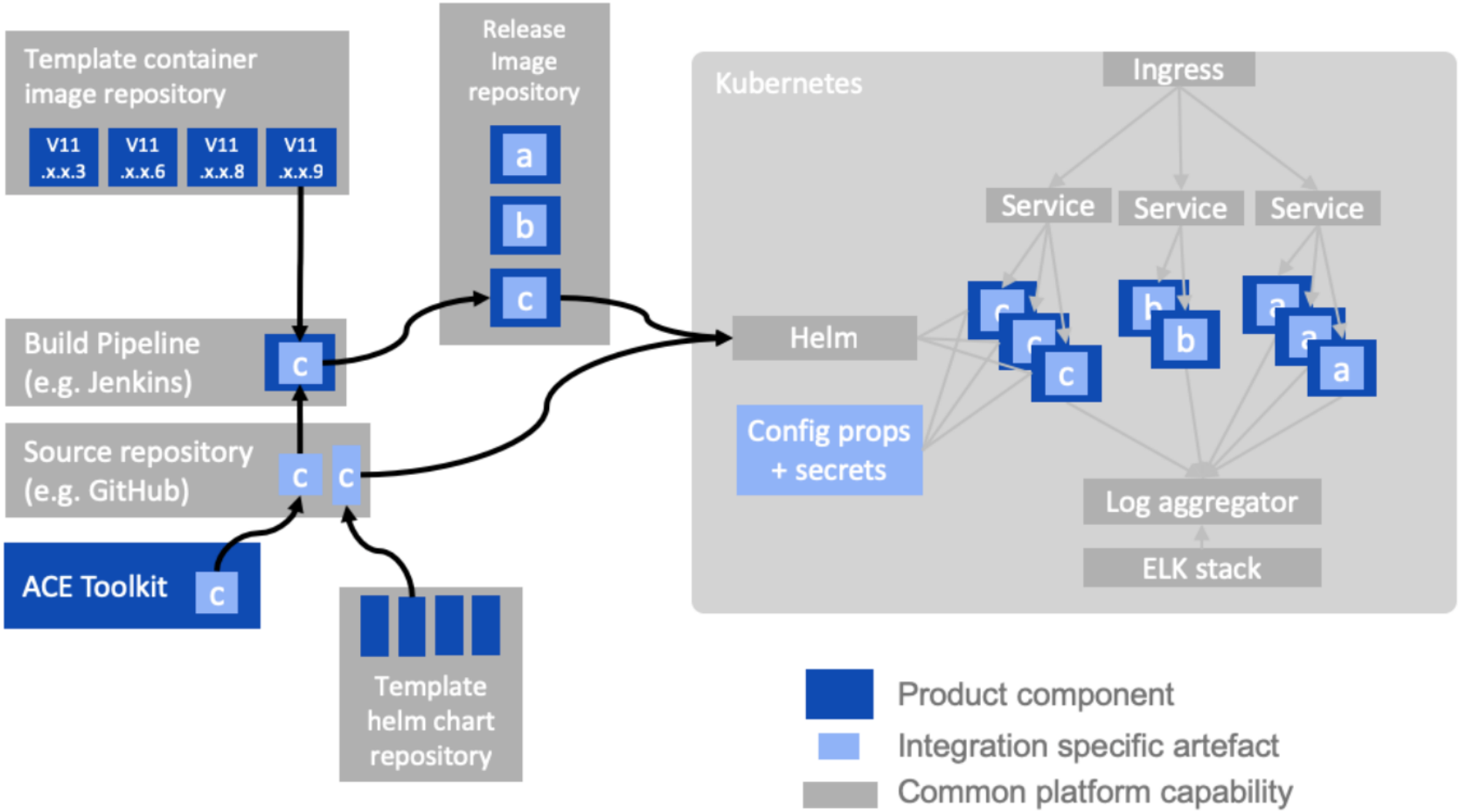
- One-phase commit sufficient
- No local queue manager required

d) Combined transaction to a database and a queue



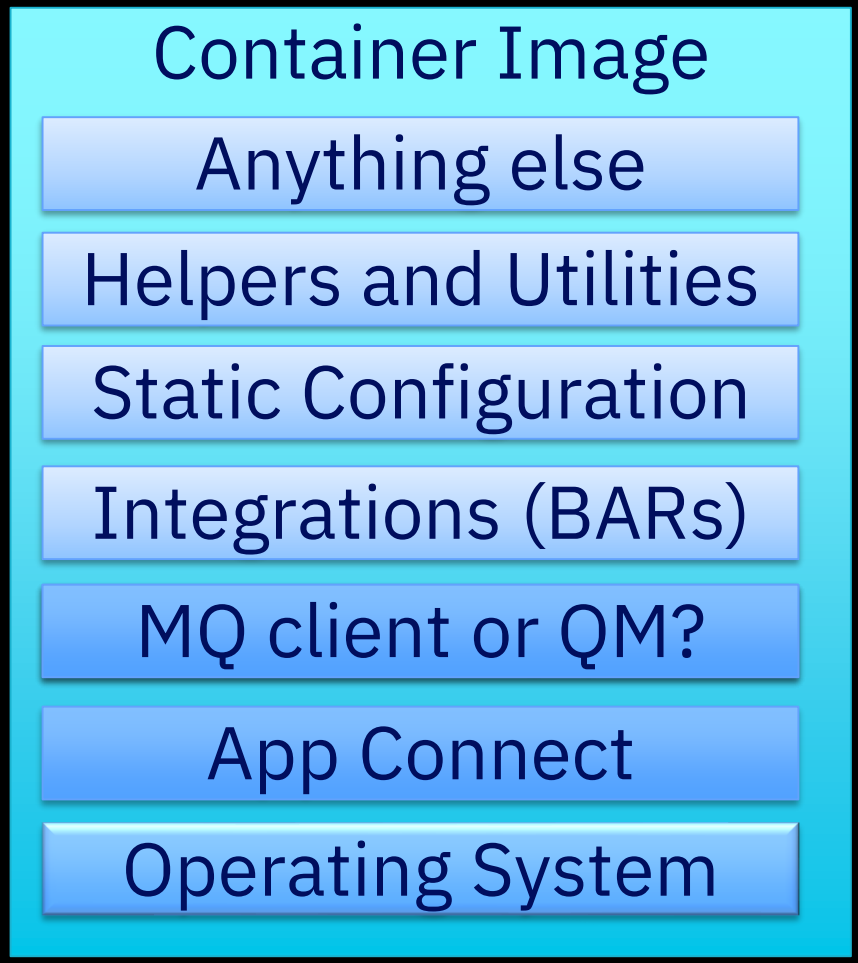
- Two-phase commit required
- Local queue manager required to co-ordinate.

Image + Config + Orchestration = Runtime

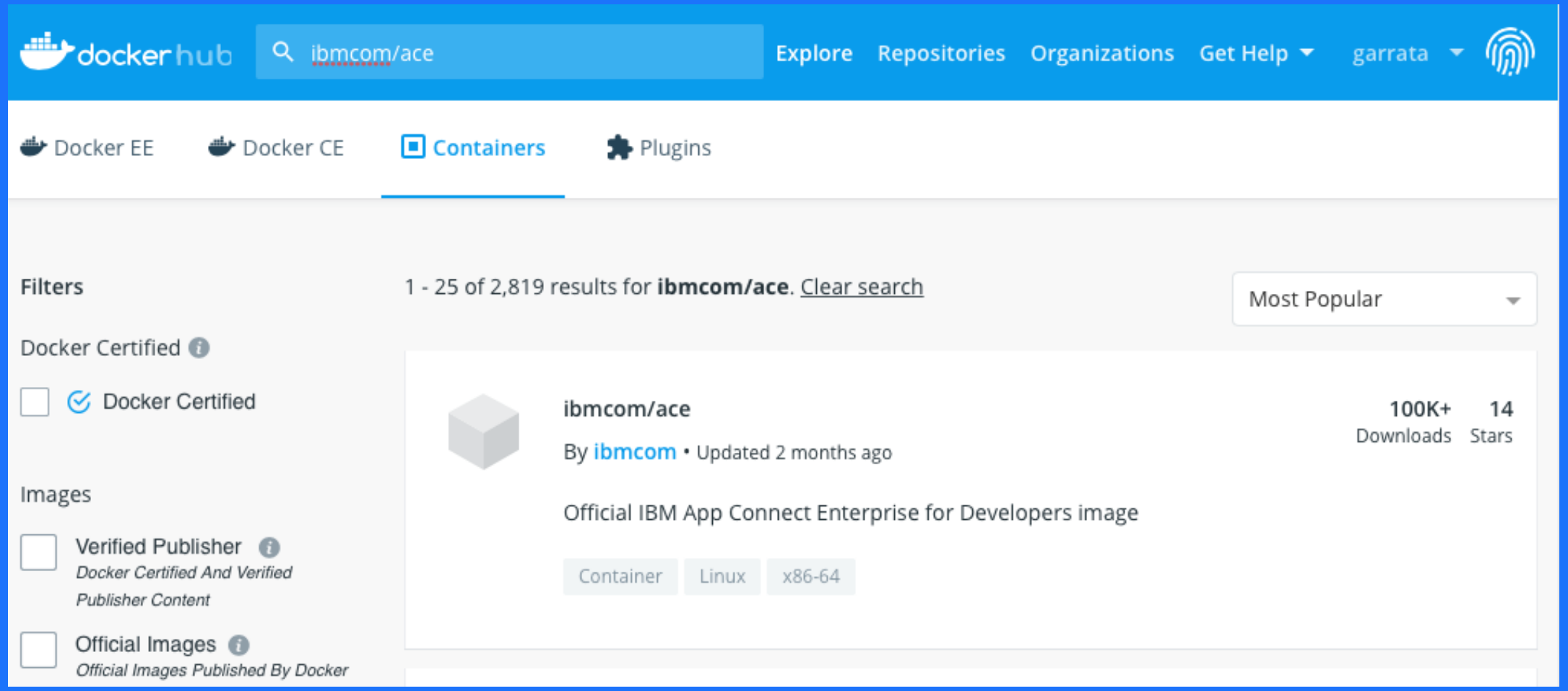


Step 1: Create your image(s)

- Docker images are ‘layered’ – each component is built on top of the previous one
- Use pre-built layers e.g. o/s, App Connect pre-built images or build your own.
- The dockerfile defines what is in your image: We have samples that we use to build our images.
- Create repositories of images at various stages of build for re-use
- Define ‘static’ config: Non/slow mover config can go in the build.



IBM pre-built examples:
Search for ibmcom/ace on hub.docker.com



Instructions, dockerfile and code/scripts that we used for ours – so you can use/copy/change them to build your own

- `11.0.0.3` , `latest` (Dockerfile)

Building a container image

Download a copy of App Connect Enterprise (ie. `ace-11.0.0.2.tar.gz`) and place it in the `deps` folder. When building the image use `build-arg` to specify the name of the file: `--build-arg ACE_INSTALL=ace-11.0.0.2.tar.gz`

- **Important:** Only ACE version **11.0.0.2 or greater** is supported.

Choose if you want to have an image with just App Connect Enterprise or an image with both App Connect Enterprise and IBM MQ Advanced.

Using App Connect Enterprise for Developers

Get ACE for Developers edition. Then place it in the `deps` folder as mentioned above.

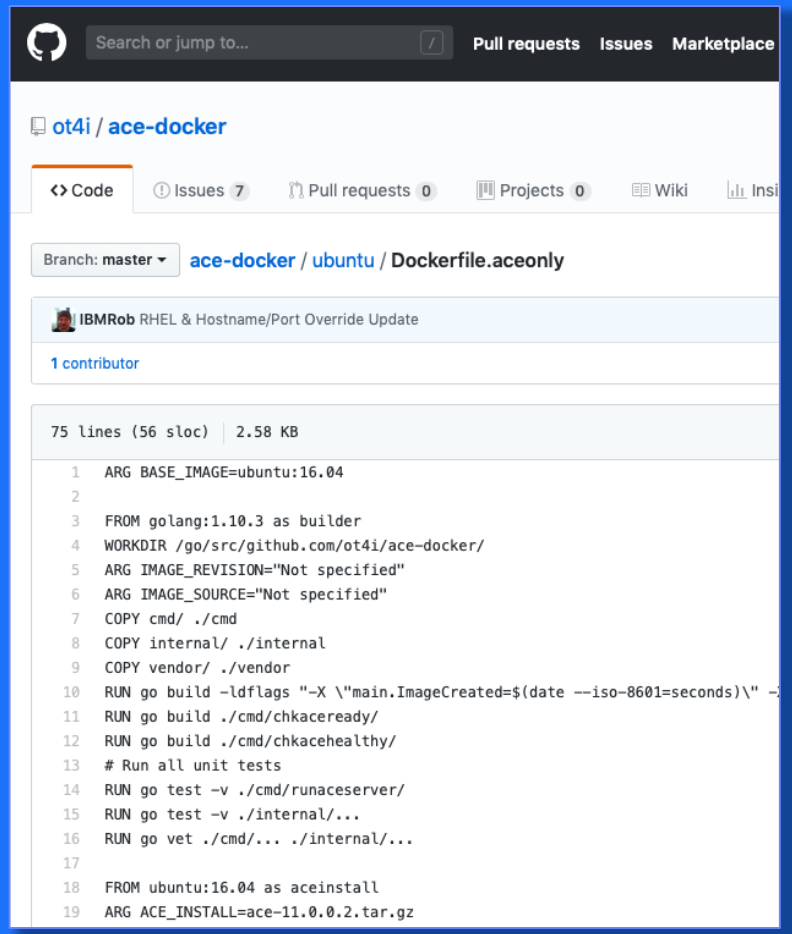
Using MQ production image

When building an image with both ACE and MQ, the docker file uses the MQ Advanced for Developers image in docker registry as base image by default on Ubuntu. On RedHat Enterprise Linux the image can be built using the MQ instructions or downloaded from [IBM Passport Advantage] (https://www.ibm.com/software/passportadvantage/).

When building a production image with MQ, follow the MQ instructions to build your own production MQ image. Then, when building the ACE with MQ image use `build-arg` to set the `BASE_IMAGE` to your production MQ image. More details below.

Build an image with App Connect Enterprise and MQ

Info on how to get the Developers or production image for MQ



Search or jump to... / Pull requests Issues Marketplace

ot4i / ace-docker

Code | Issues 7 | Pull requests 0 | Projects 0 | Wiki | Insi

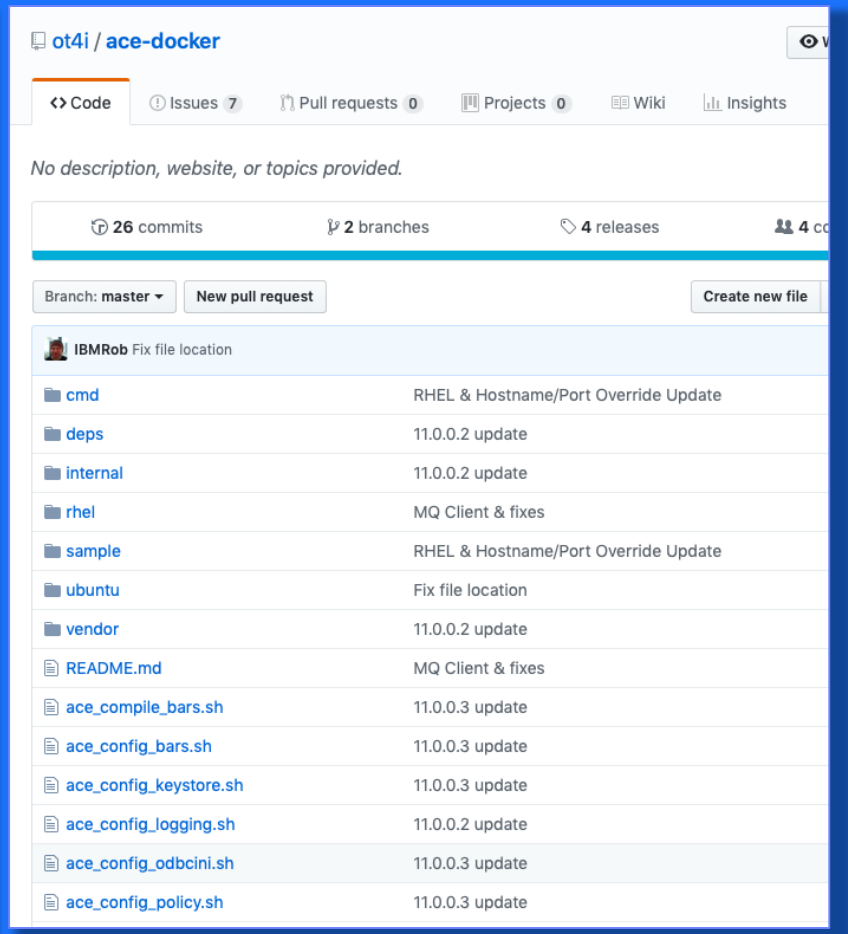
Branch: master ▾ ace-docker / ubuntu / Dockerfile.aceonly

IBMRob RHEL & Hostname/Port Override Update

1 contributor

75 lines (56 sloc) | 2.58 KB

```
1  ARG BASE_IMAGE=ubuntu:16.04
2
3  FROM golang:1.10.3 as builder
4  WORKDIR /go/src/github.com/ot4i/ace-docker/
5  ARG IMAGE_REVISION="Not specified"
6  ARG IMAGE_SOURCE="Not specified"
7  COPY cmd/ ./cmd
8  COPY internal/ ./internal
9  COPY vendor/ ./vendor
10 RUN go build -ldflags "-X \"main.ImageCreated=$(date --iso-8601=seconds)\" -
11 RUN go build ./cmd/chkaceready/
12 RUN go build ./cmd/chkacehealthy/
13 # Run all unit tests
14 RUN go test -v ./cmd/runaceserver/
15 RUN go test -v ./internal/...
16 RUN go vet ./cmd/... ./internal/...
17
18 FROM ubuntu:16.04 as aceinstall
19 ARG ACE_INSTALL=ace-11.0.0.2.tar.gz
```



ot4i / ace-docker

Code | Issues 7 | Pull requests 0 | Projects 0 | Wiki | Insights

No description, website, or topics provided.

26 commits | 2 branches | 4 releases | 4 co

Branch: master ▾ | New pull request | Create new file

IBMRob Fix file location

cmd	RHEL & Hostname/Port Override Update
deps	11.0.0.2 update
internal	11.0.0.2 update
rhel	MQ Client & fixes
sample	RHEL & Hostname/Port Override Update
ubuntu	Fix file location
vendor	11.0.0.2 update
README.md	MQ Client & fixes
ace_compile_bars.sh	11.0.0.3 update
ace_config_bars.sh	11.0.0.3 update
ace_config_keystore.sh	11.0.0.3 update
ace_config_logging.sh	11.0.0.2 update
ace_config_odbcini.sh	11.0.0.3 update
ace_config_policy.sh	11.0.0.3 update

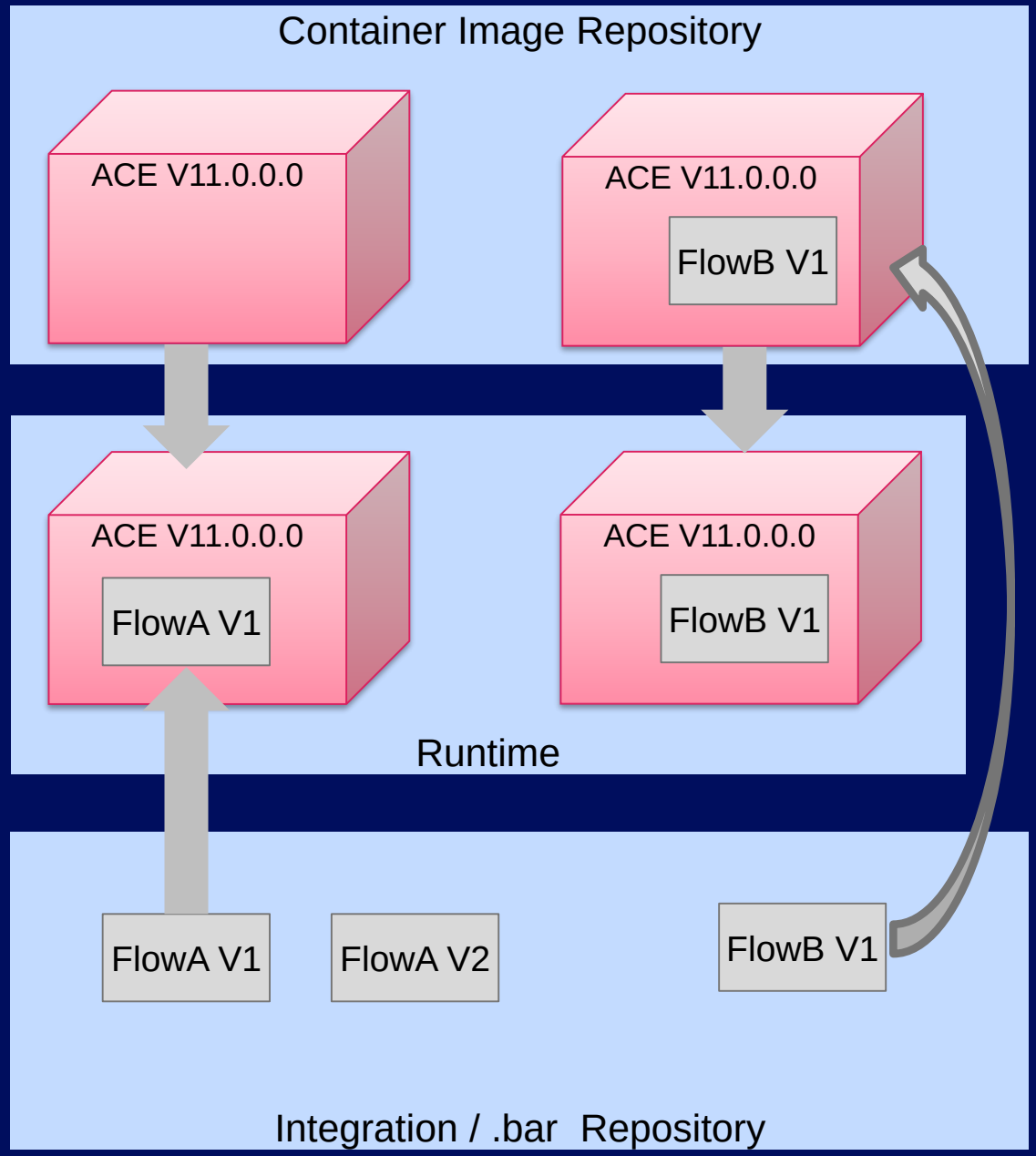
Immutability==Consistency

Do you deploy an App Connect container and then deploy flows at runtime?

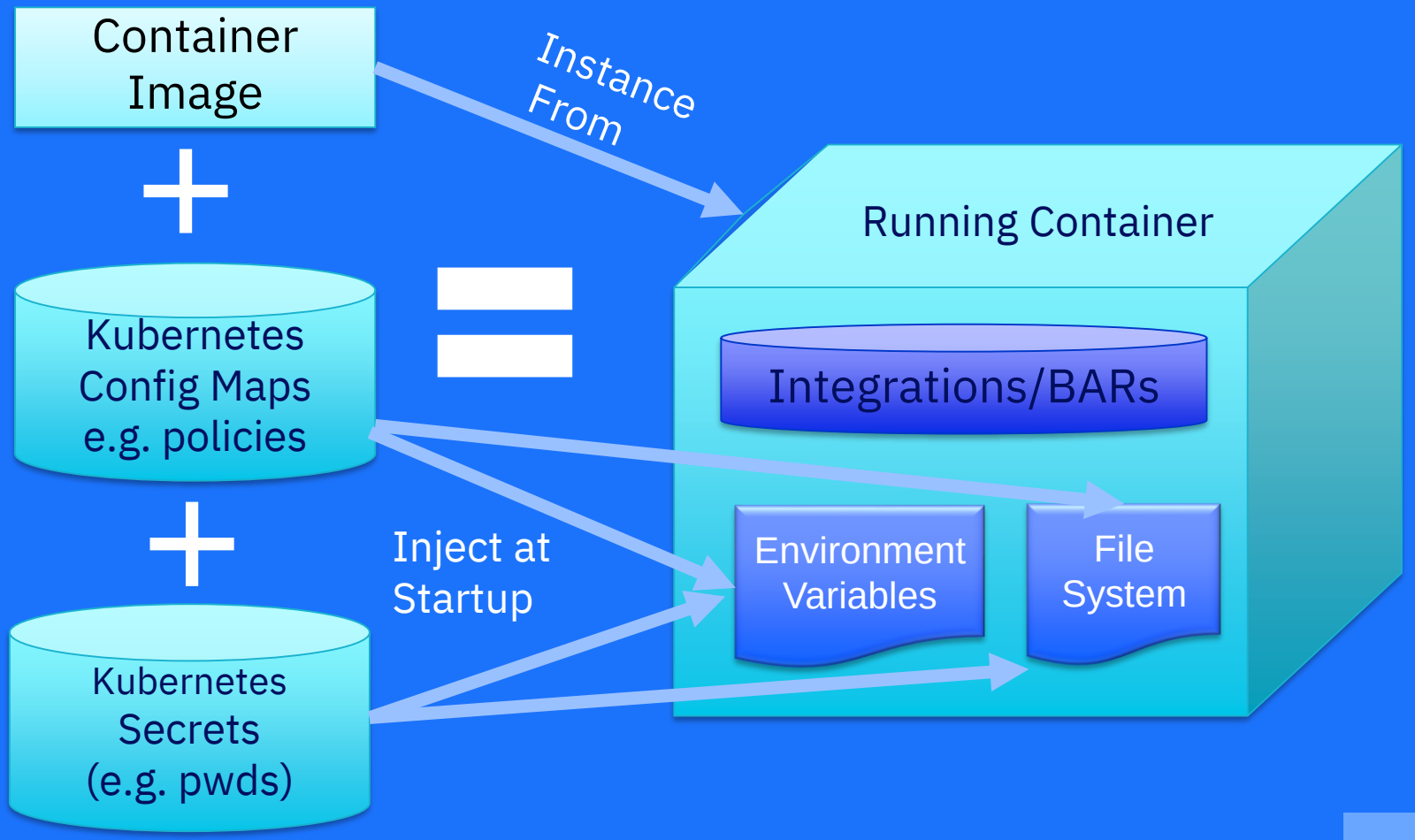
Or do you build a container with all of the flows ready-built into it?

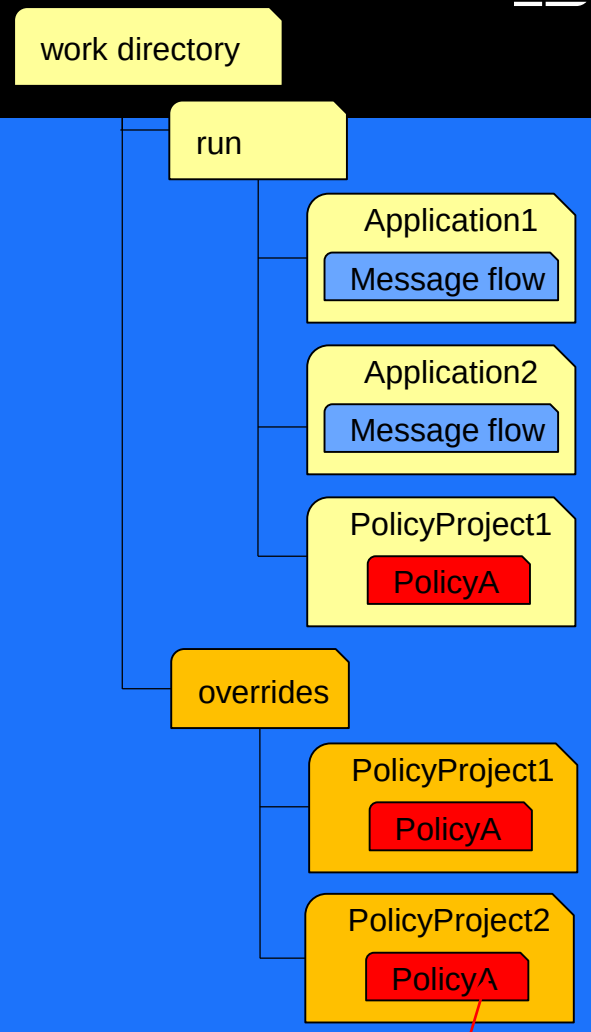
Which makes DEV look more like PROD?

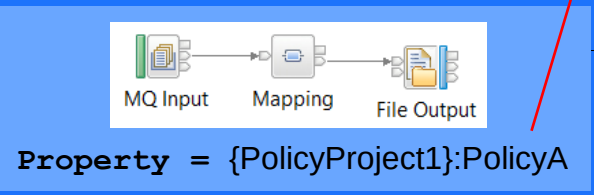
Which is simpler to deploy?



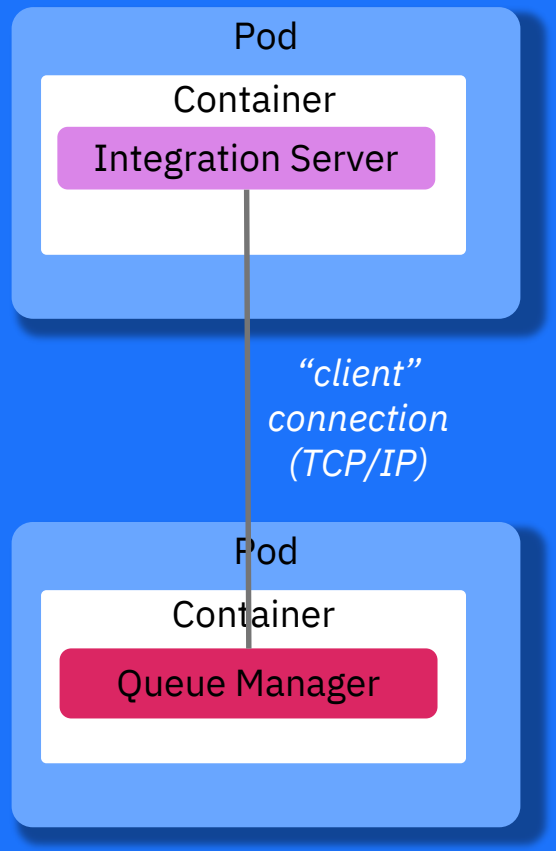
Step 2: Define your configuration points

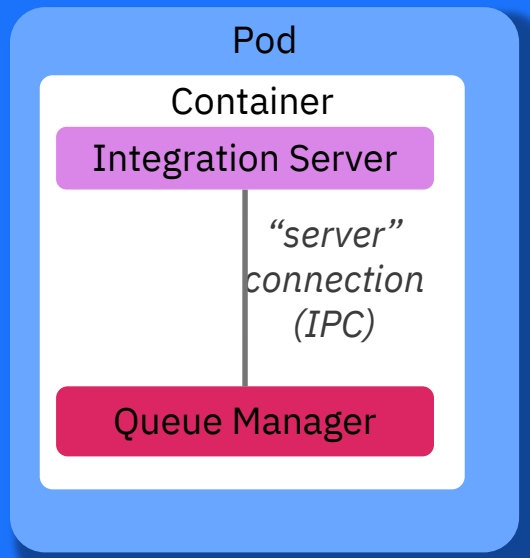


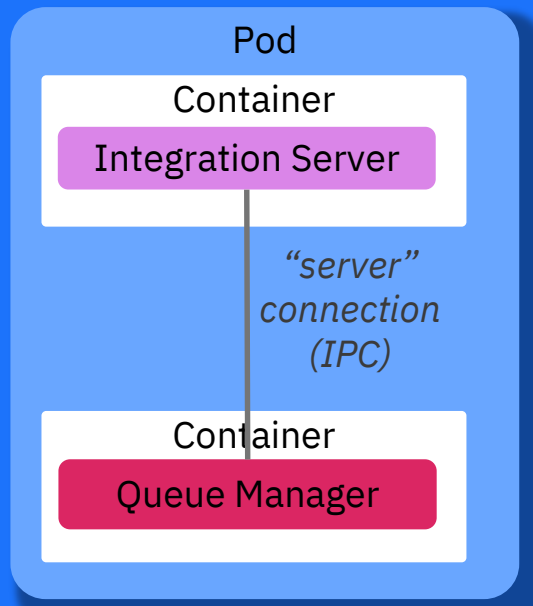




Step 3: Create your pod definitions







A pod is more than just a group of containers

A pod shares namespace, filespace and ip address (localhost)

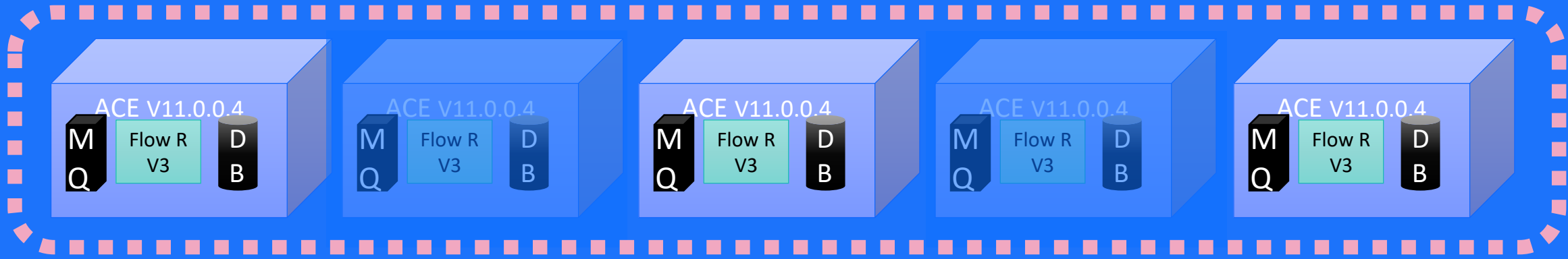
Multiple containers in a pod allows independence of container builds / versions / updates etc. with a consistent deployable unit

Step 4: Create your deployment charts**

‘For this integration, Make it like this’ – **Kubernetes is declarative**

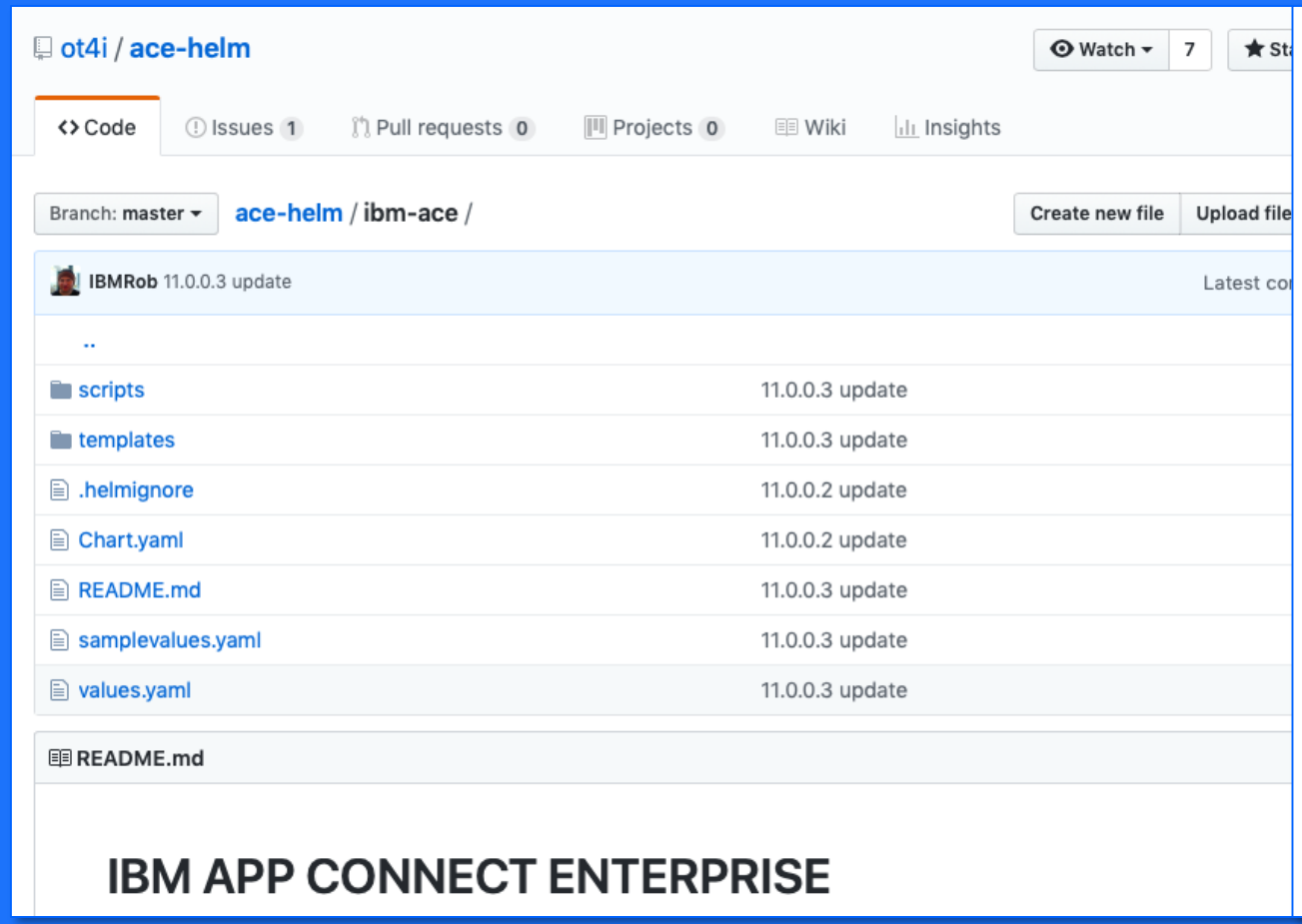
- E.g. Use a Pod with:
  - Flow R, Version 3 on ACE 11.0.0.4
  - MQ version AAA
  - Database version BBB

- Make it a set of replicas of the same image
- Minimum of 3 running at all times
- Maximum of 5 running – K8s: you scale it
- Expose ports x, y and z



**Container Philosophy:
Everything MUST have a nautical name..

Did we mention we have sample helm charts?



https://github.com/ot4i/ace-helm

Container Philosophy Part 2:

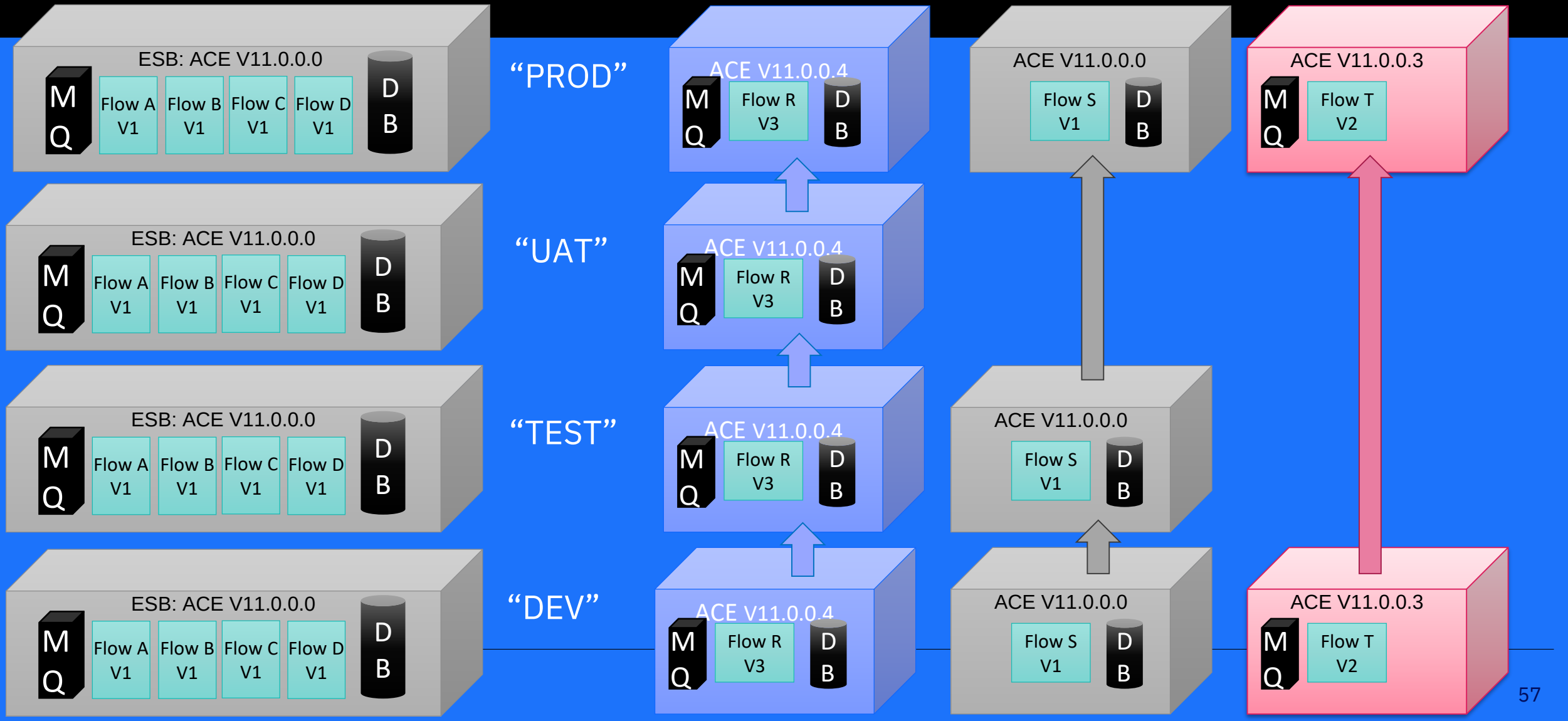
Containers are plentiful and cheap

If we lose one (crashes), just spin up another**

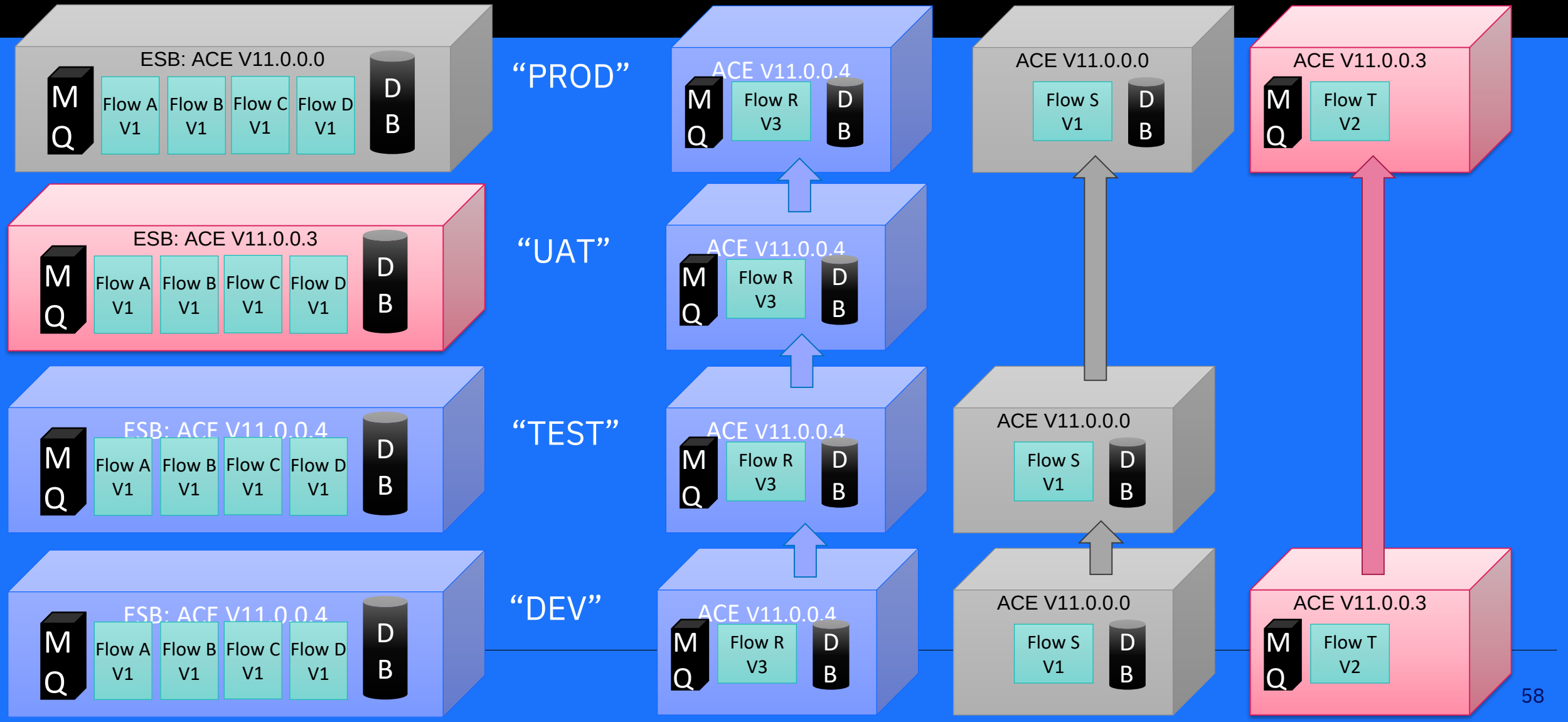
“Just Another Container” – not “an ACE container”?

** If you have state, make sure you know where it is

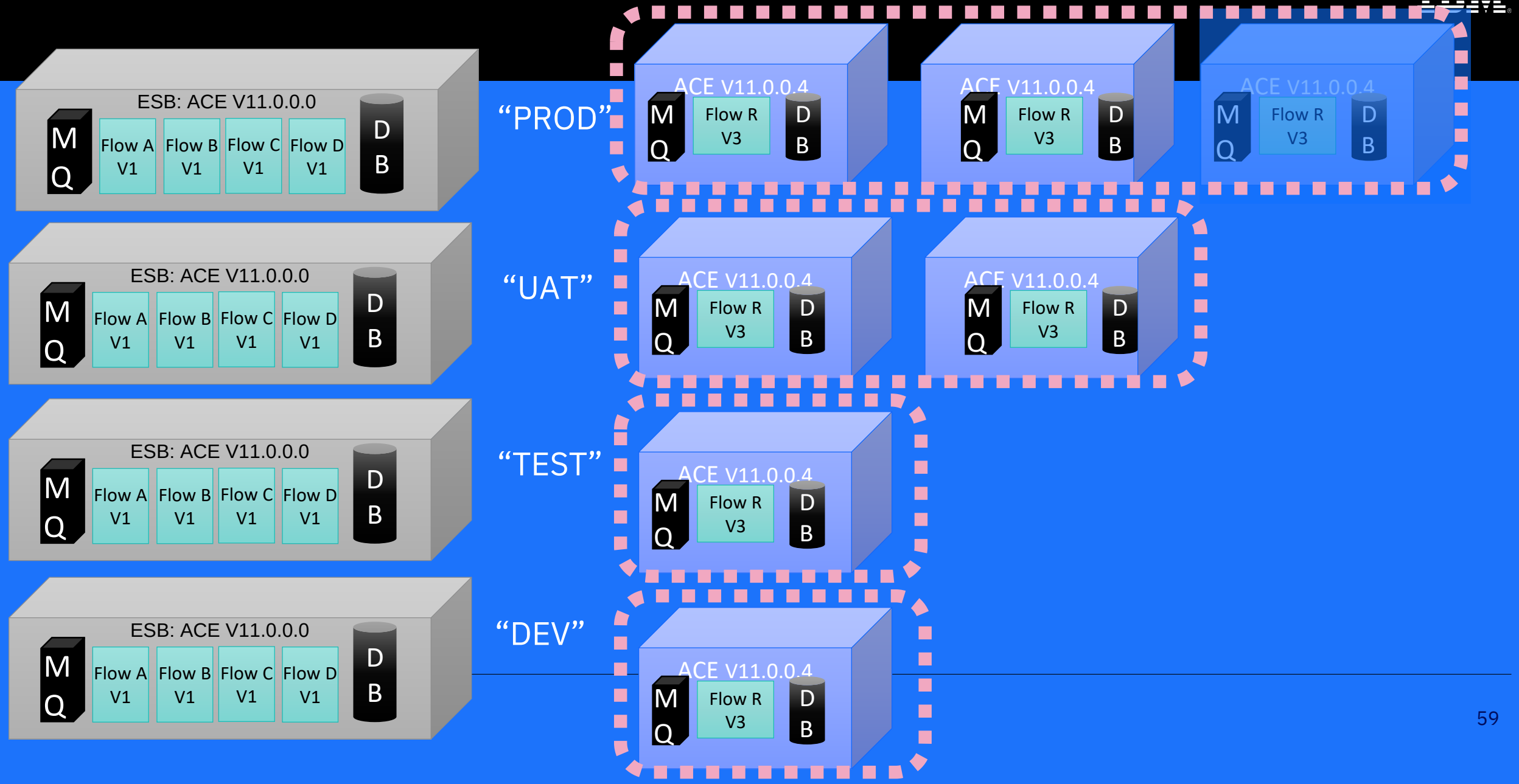
Infrastructure as code – no more “environments”?



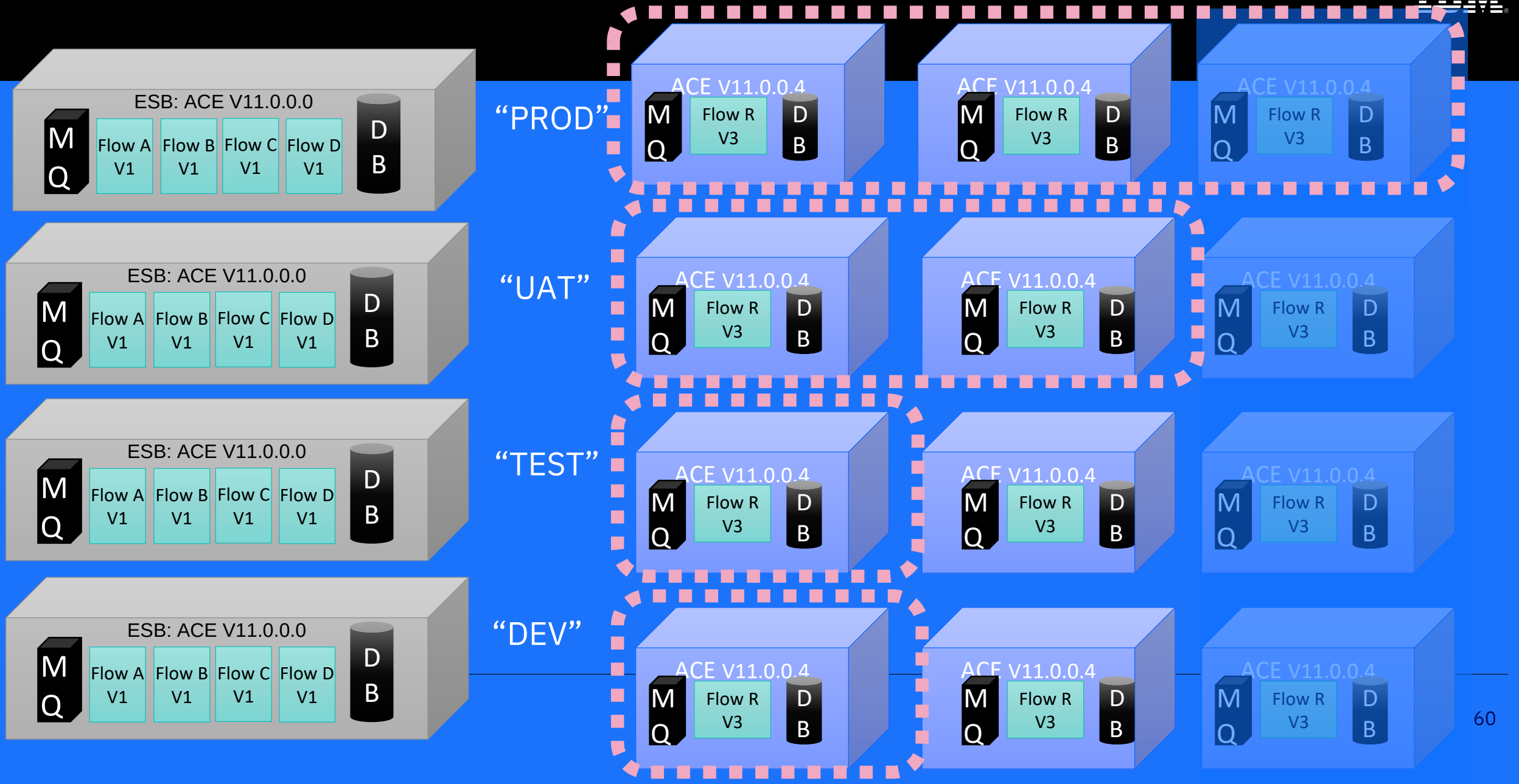
No more “Environment Drift?”



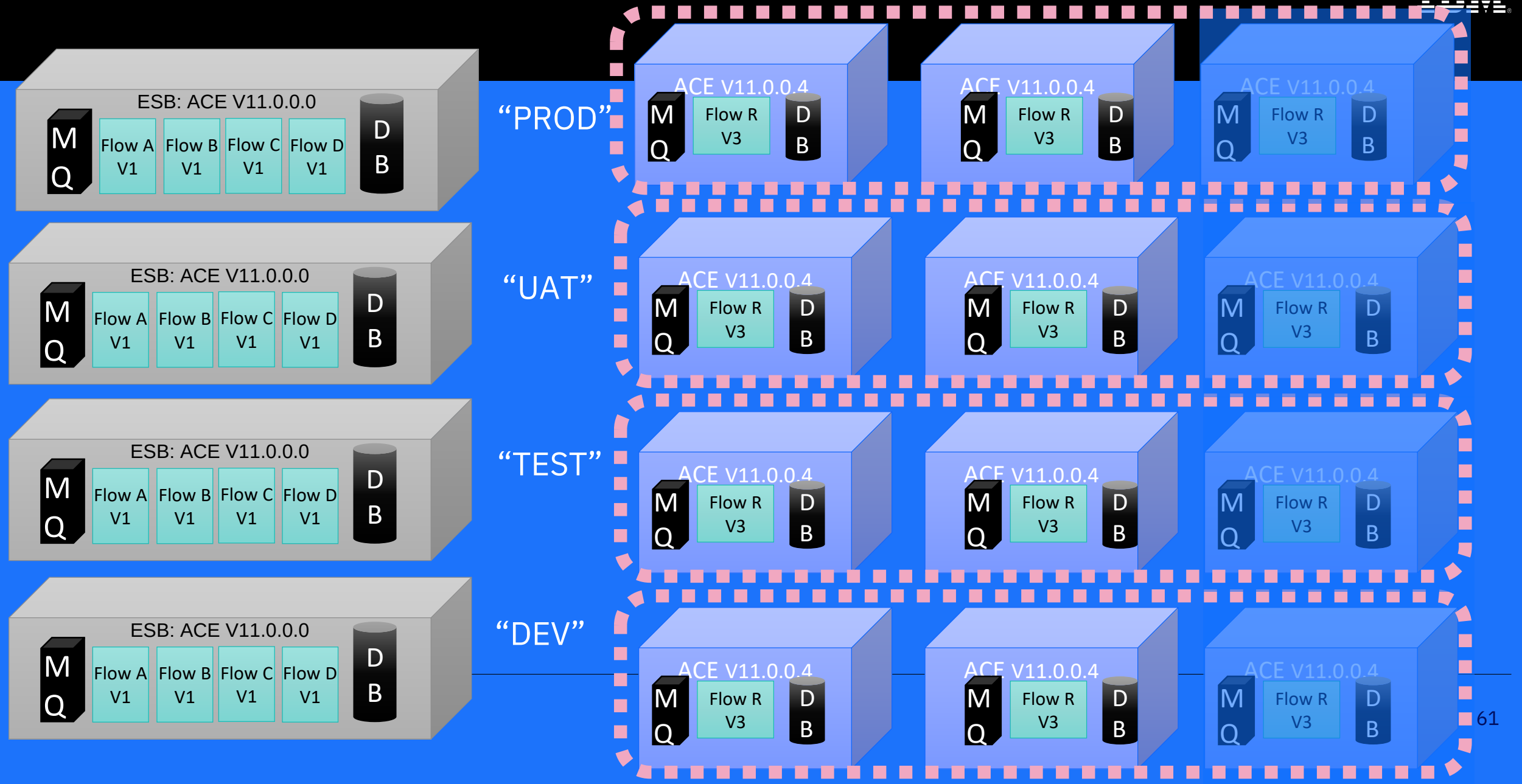
Shift DevOps “Left”: Performance and HA/DR in DEV



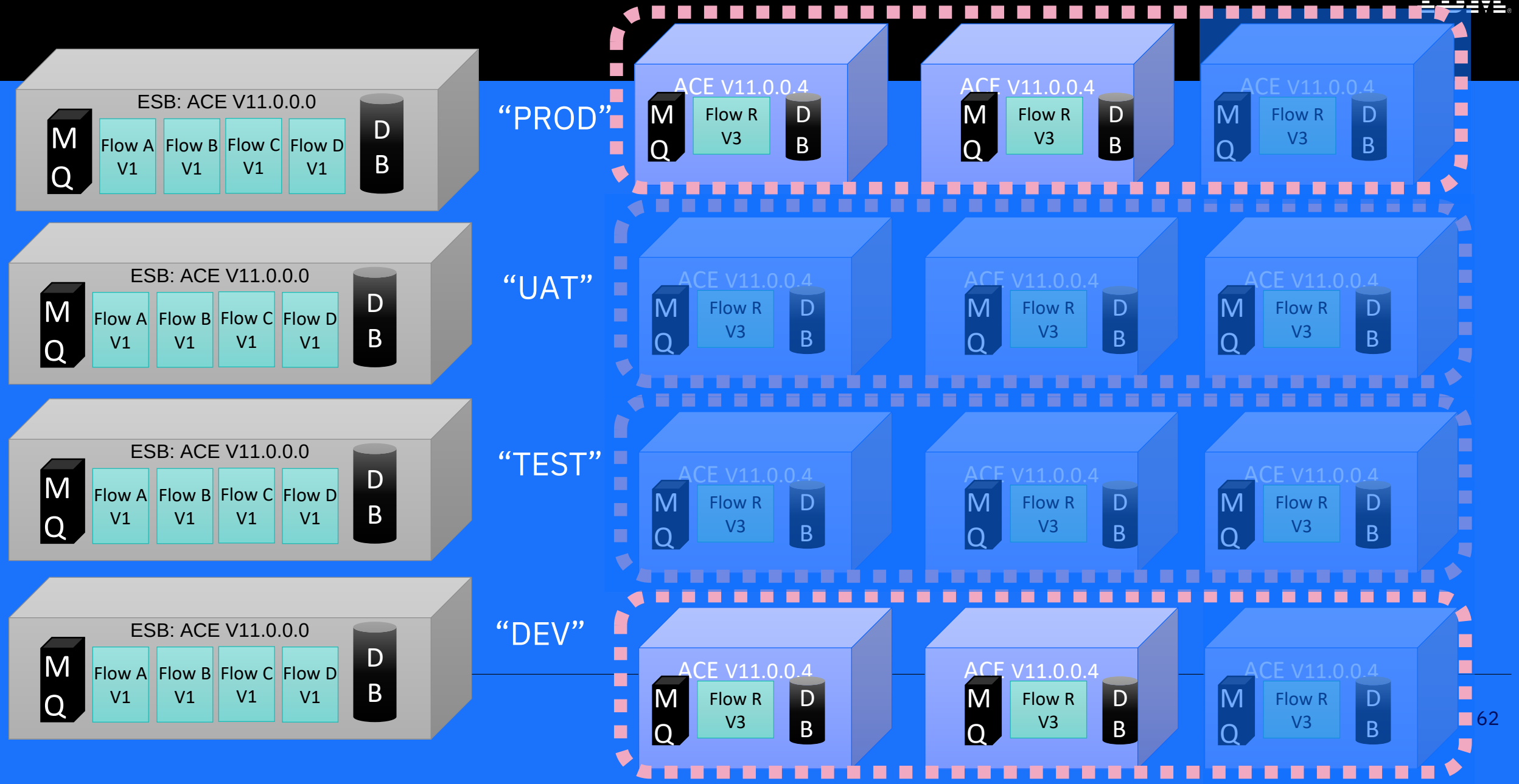
Shift DevOps “Left”: Performance and HA/DR in DEV



Shift DevOps “Left”: Performance and HA/DR in DEV



“Think Cloud” – Only spin up when you need it



Agile Integration Architecture

Modernizing integration to enable ***business*** *agility*

Fine grained deployment

Architecture & design

→ Improve build independence and production velocity (***deployment*** *agility*)

Decentralized Ownership

People & Process

→ Accelerate agility and innovation (***development*** *agility*)

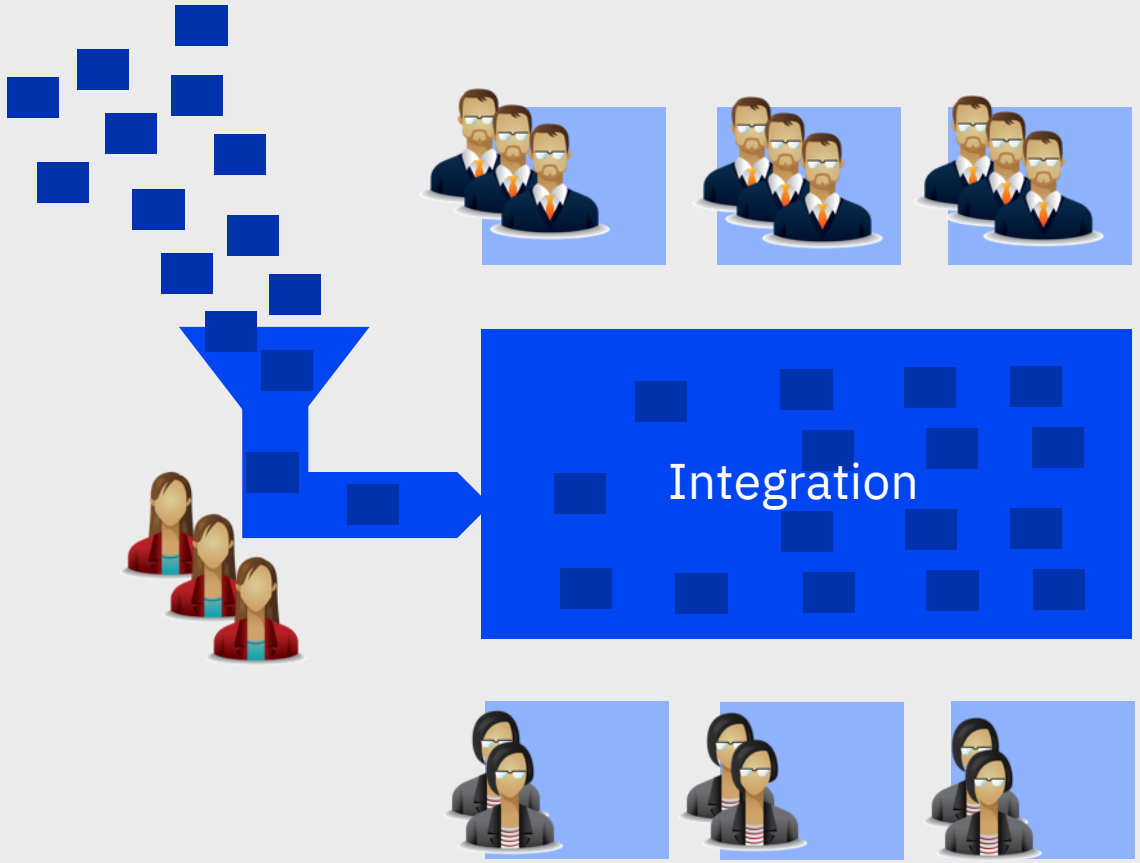
Cloud native infrastructure

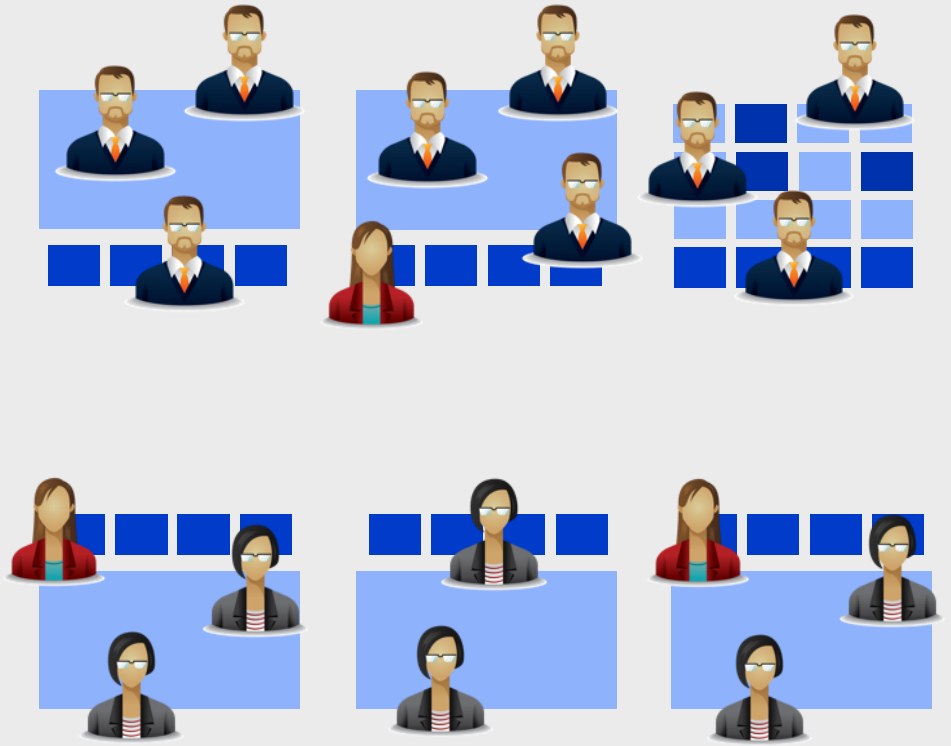
Infrastructure & Technology

→ Dynamic scalability and inherent resilience (***operational*** *agility*)

Decentralized Ownership

People & Process

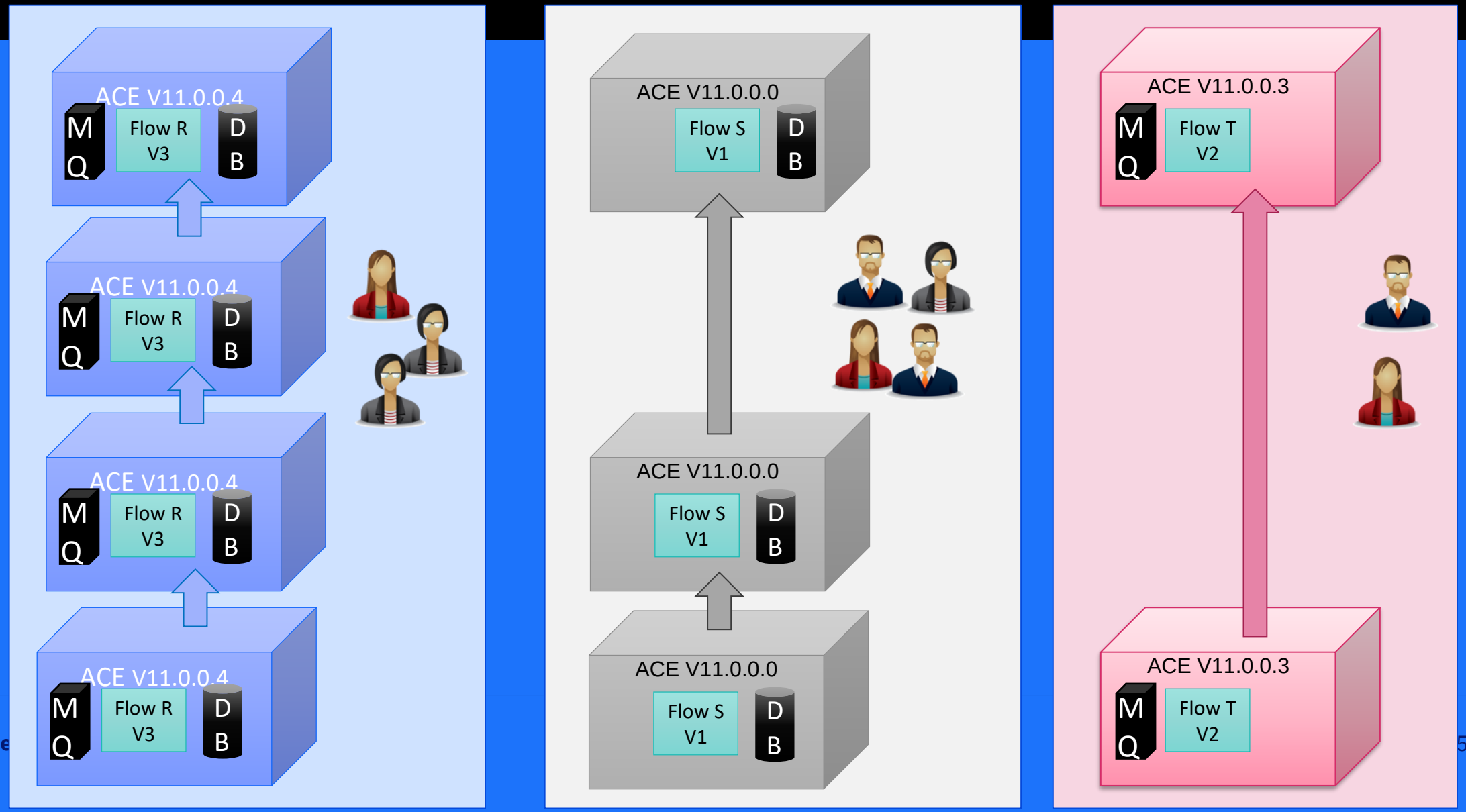




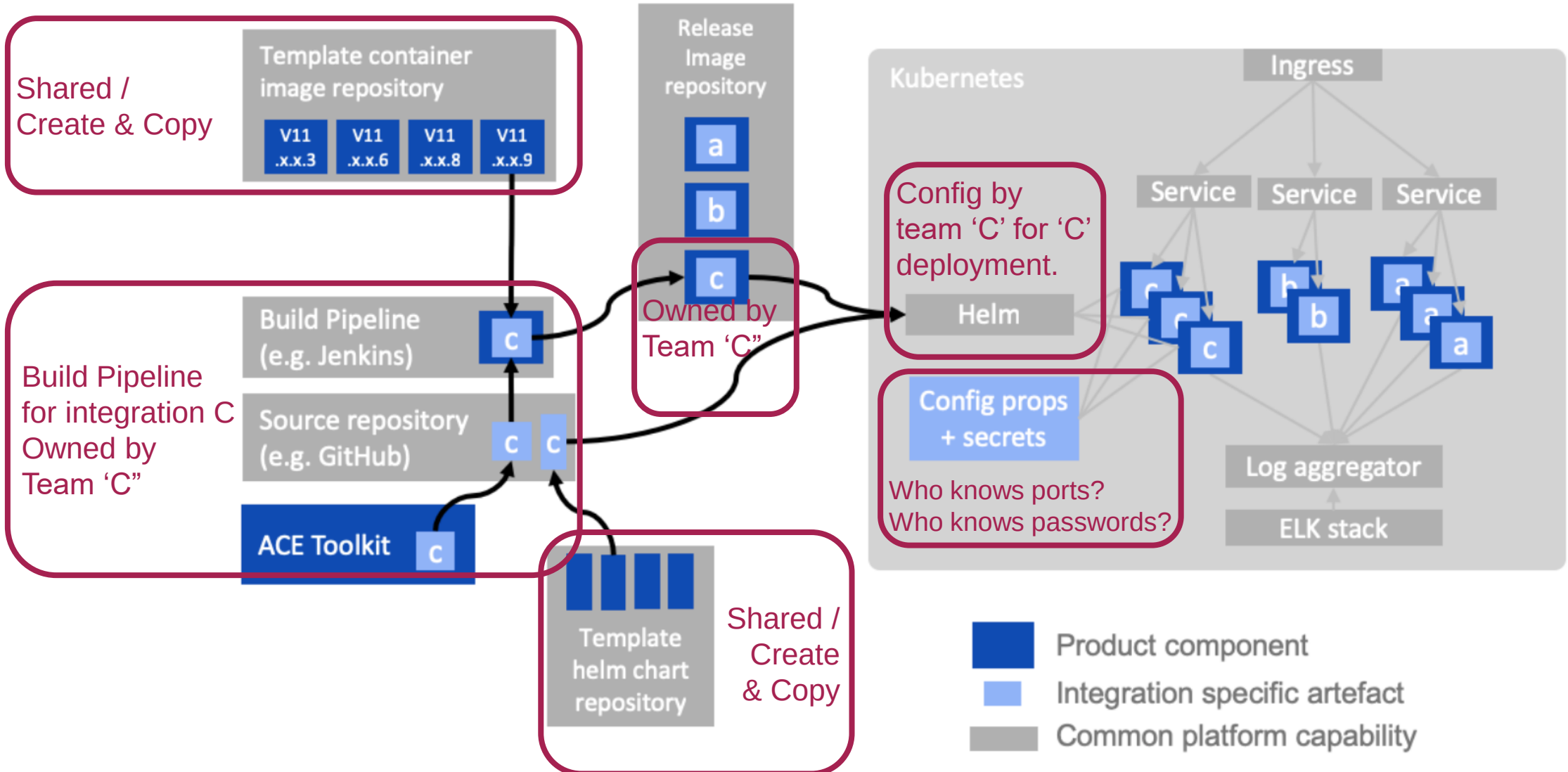
Centralized integration facility with plateauing productivity

Increased team agility as a result of greater autonomy and clearer ownership

Independent Deployment & Pipeline
Enables independent Lifecycle Ownership



Team owns their pipeline: Code+build+config+deploy



Logging

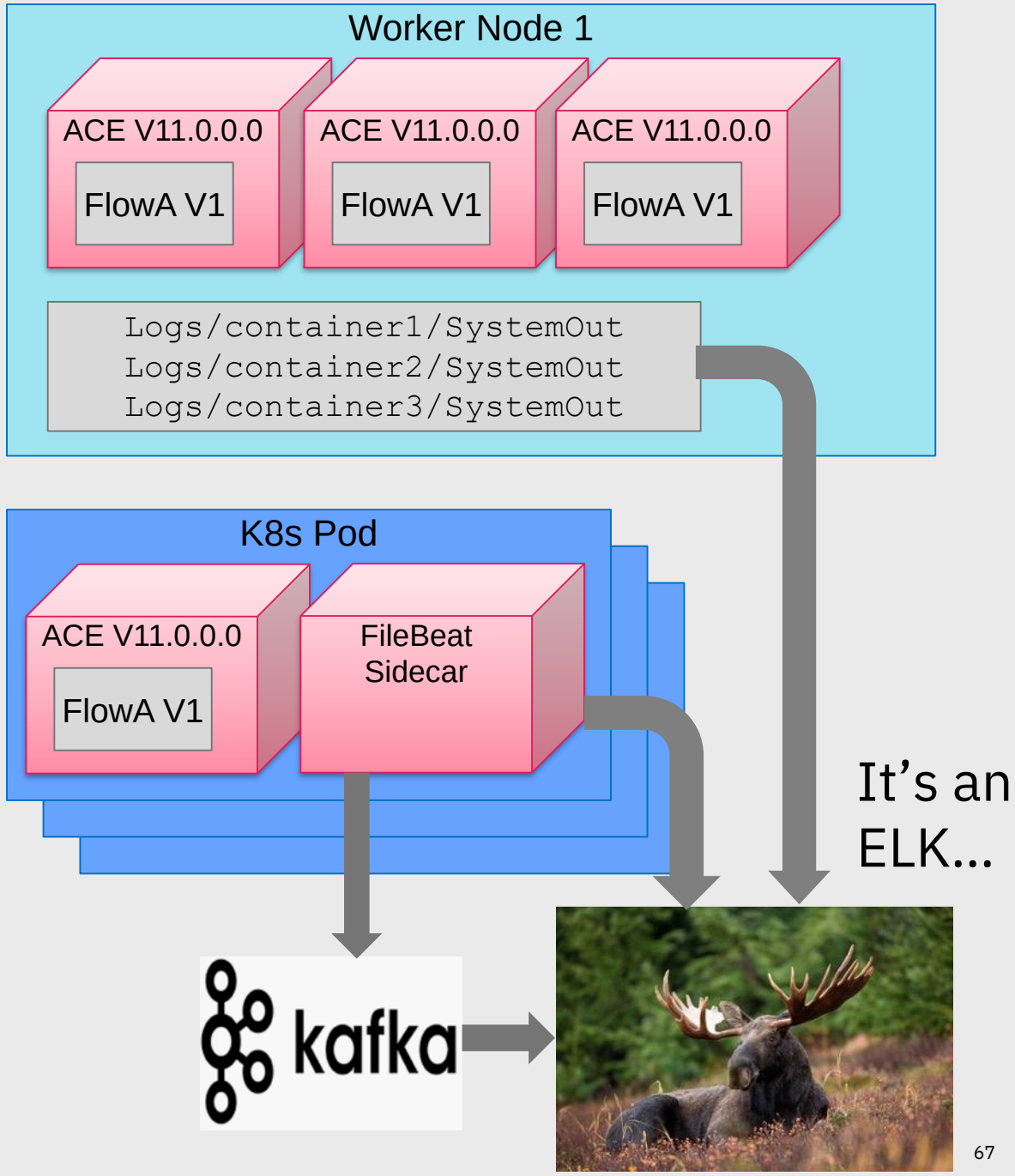
One place to log them all...

One place to find them!

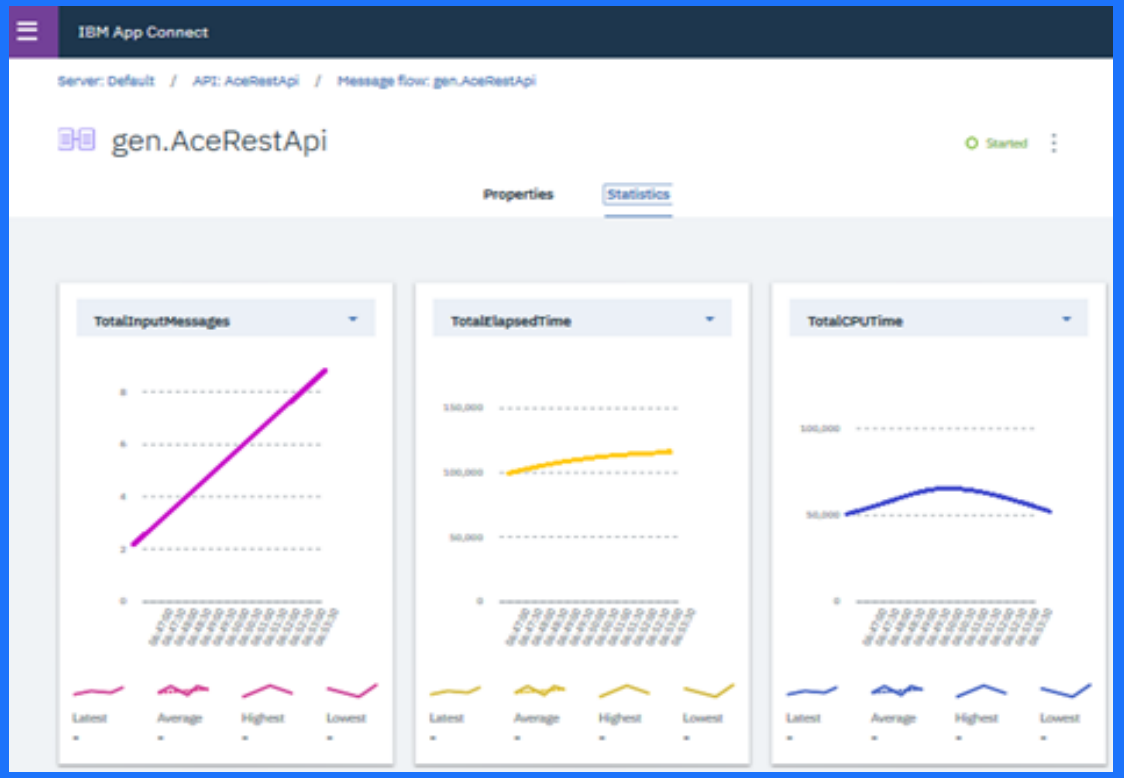
You will create a LOT of logs - the number will change dynamically.

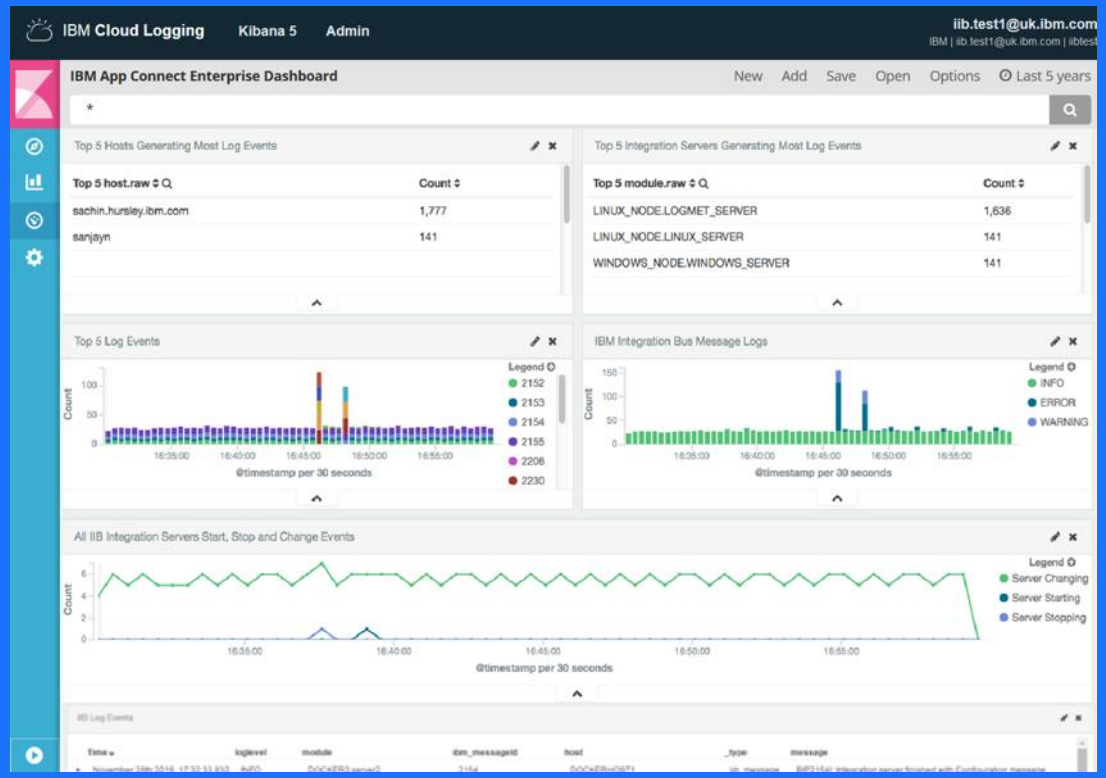
Forward and Manage your logs centrally – then filter as you need.

Standardise format as far as possible.



Monitoring: “Just another container/Pod”?



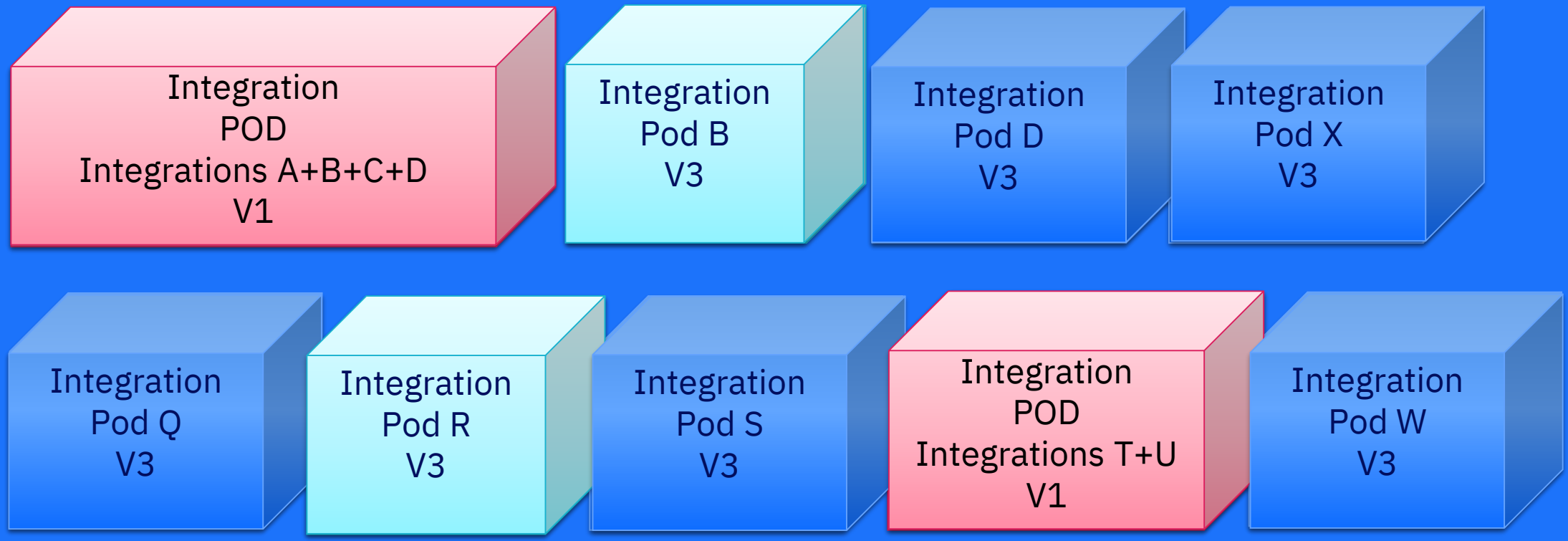


Ops want consistency of infrastructure and operation

Traditional infrastructure
(Pets)

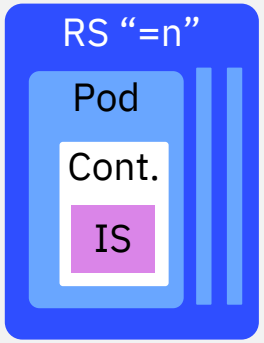
Cloud native infrastructure
(Livestock)

Why does OPS care what's in a pod?
"Just another Container" or JAC surely?

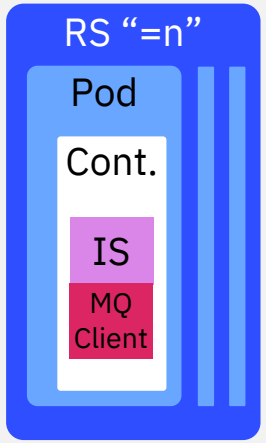


OPS still need to know the PATTERN:
State can complicate things a lot – know were it is!

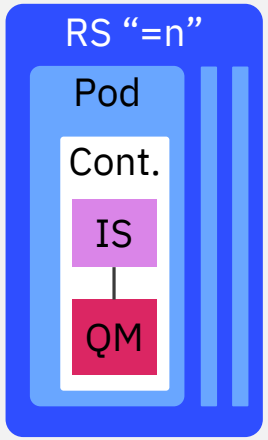
Scalable no MQ Client



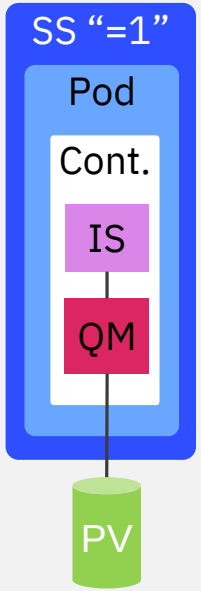
Scalable with MQ Client



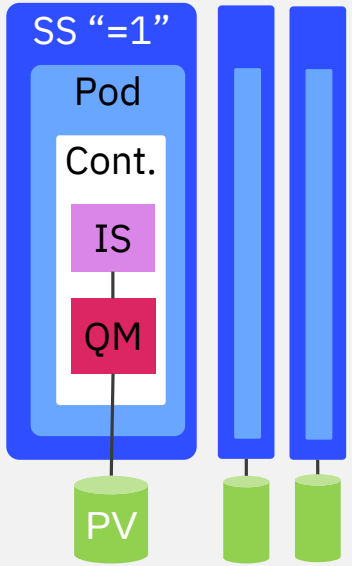
Scalable, with non-persistent local MQ



Single instance with persistent local MQ



Active/active with persistent independent local MQ



Traditional (with node)

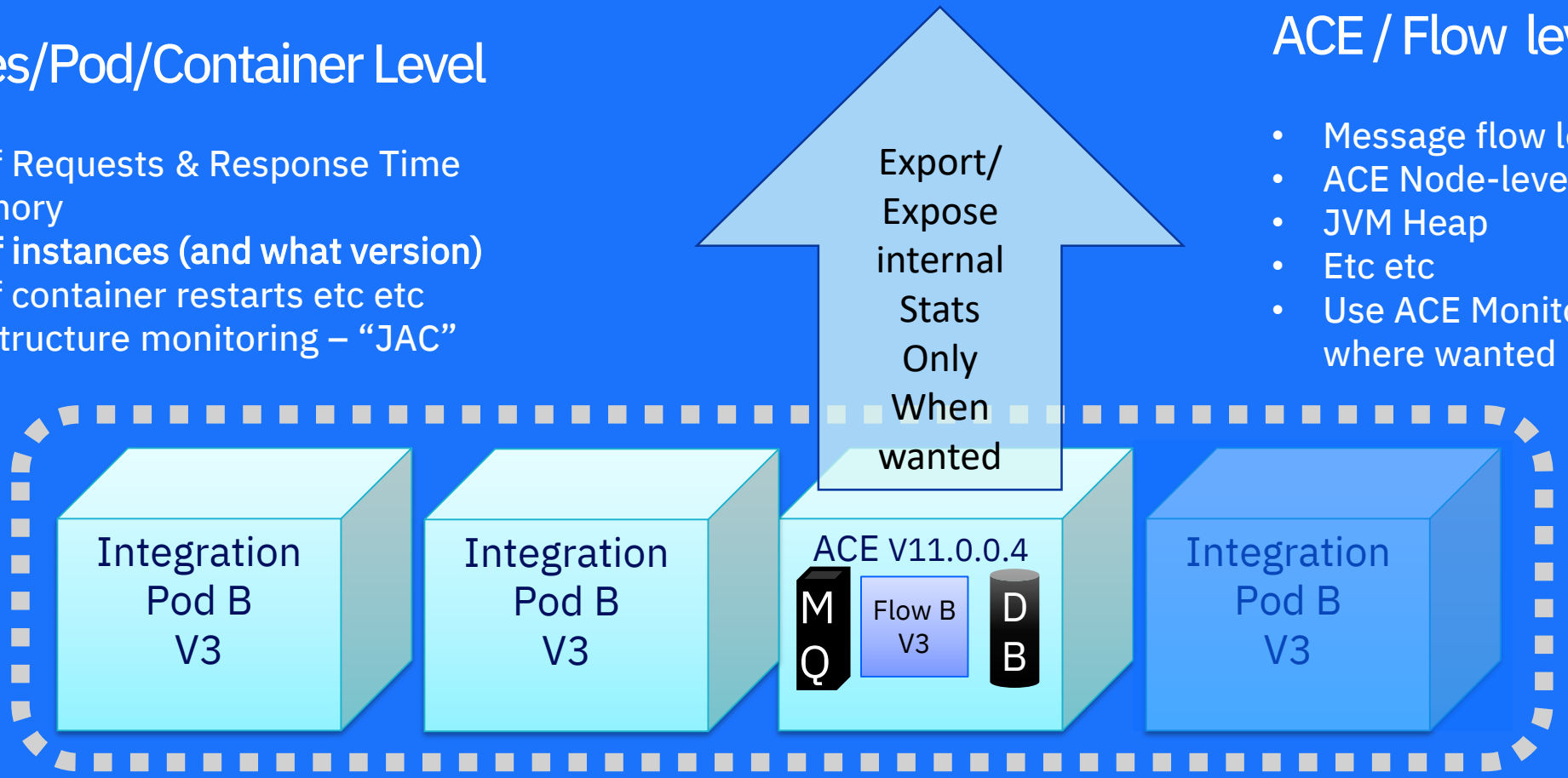
Shown separately

Stateless	Stateless	Stateless	Stateful	Stateful	Stateful
Continuous availability	Continuous availability	Continuous availability	High Availability	Continuous Service Availability High Message Availability	Continuous Availability
Dynamic horizontal scaling	Dynamic horizontal scaling	Dynamic horizontal scaling	Vertical scaling only	Non-dynamic horizontal scaling	Non-dynamic horizontal scaling
No EDA	No EDA	Enables EDA, but note that must be replica=1 for sequencing	Enables EDA and 2PC	Enables EDA and 2PC	Enables EDA and 2PC

Monitoring: Infrastructure vs Application

Kubernetes/Pod/Container Level

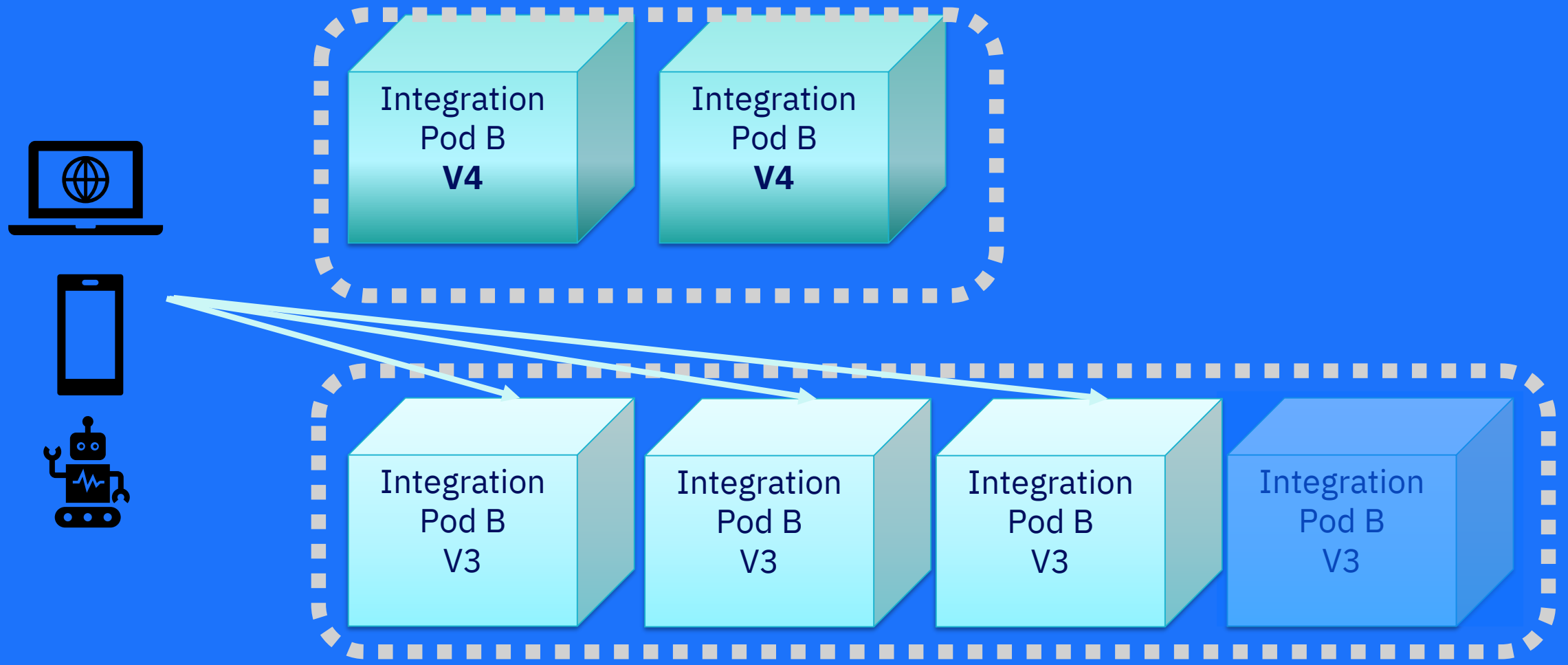
- Number of Requests & Response Time
- CPU / Memory
- **Number of instances (and what version)**
- Number of container restarts etc etc
- Use Infrastructure monitoring – "JAC"



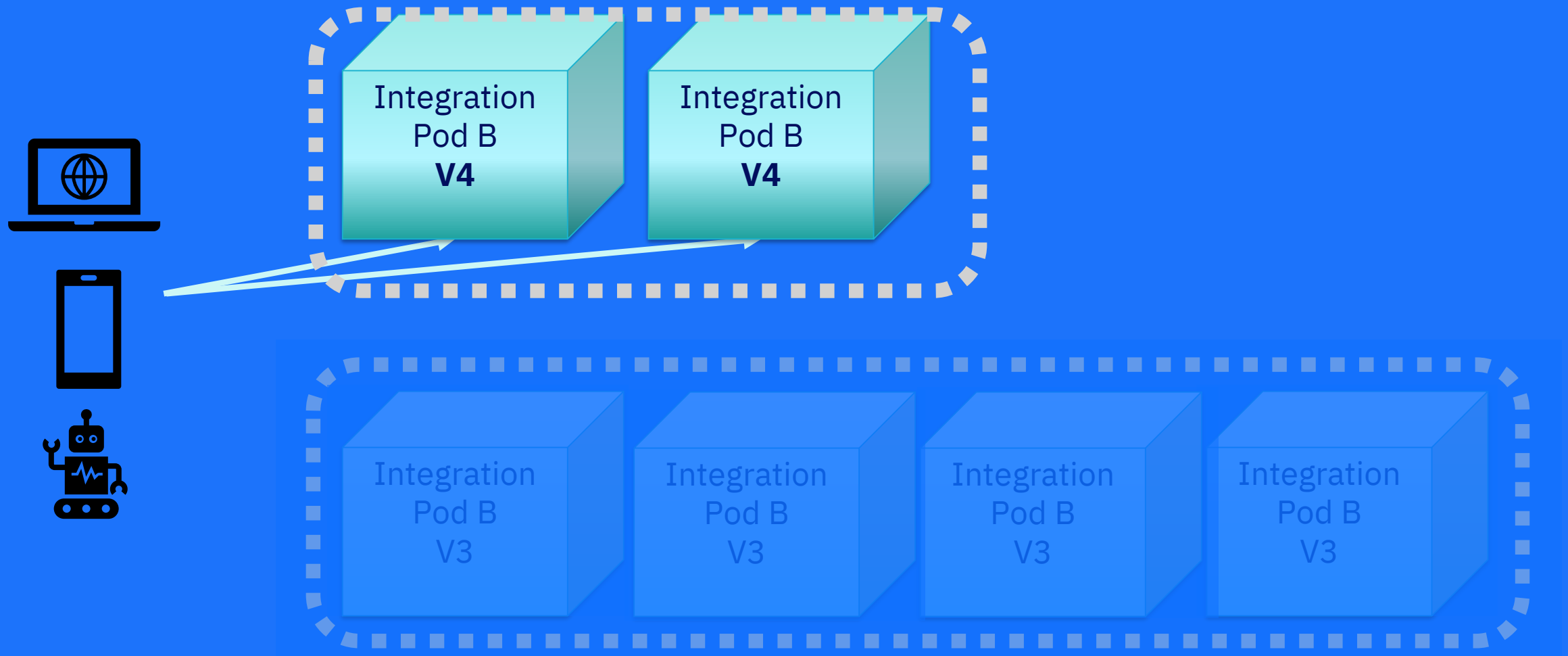
ACE / Flow level

- Message flow level data
- ACE Node-level stats
- JVM Heap
- Etc etc
- Use ACE Monitoring and export where wanted

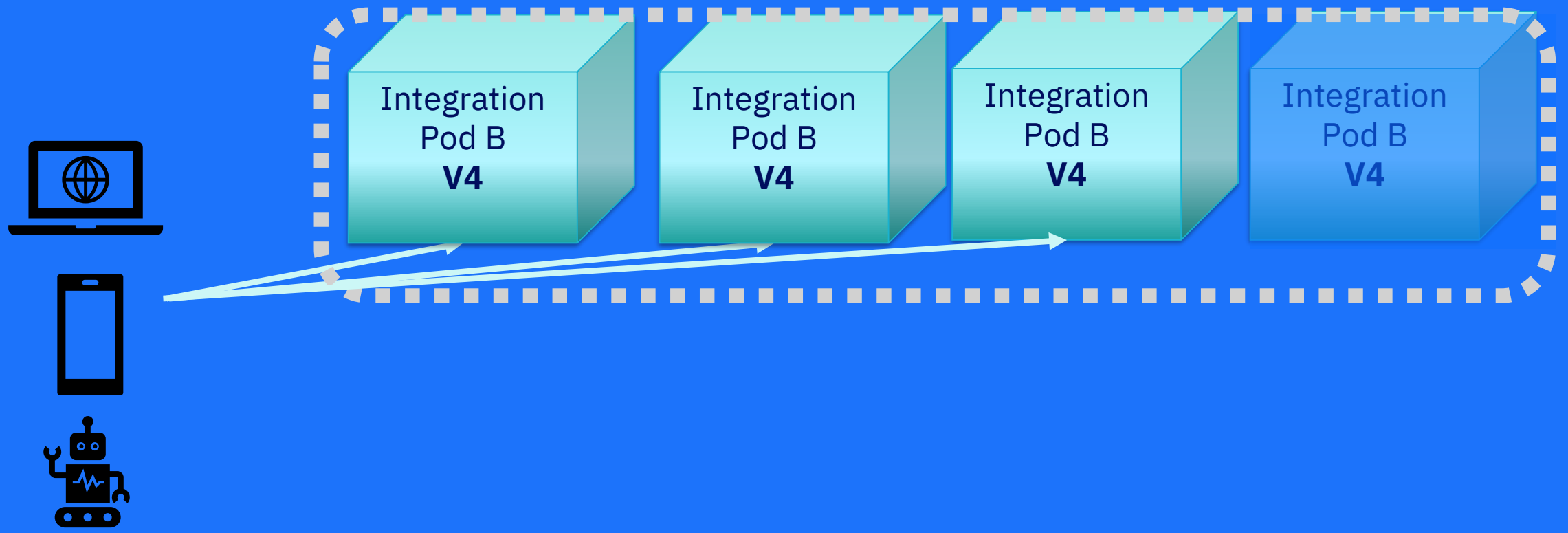
Versioning and upgrading…



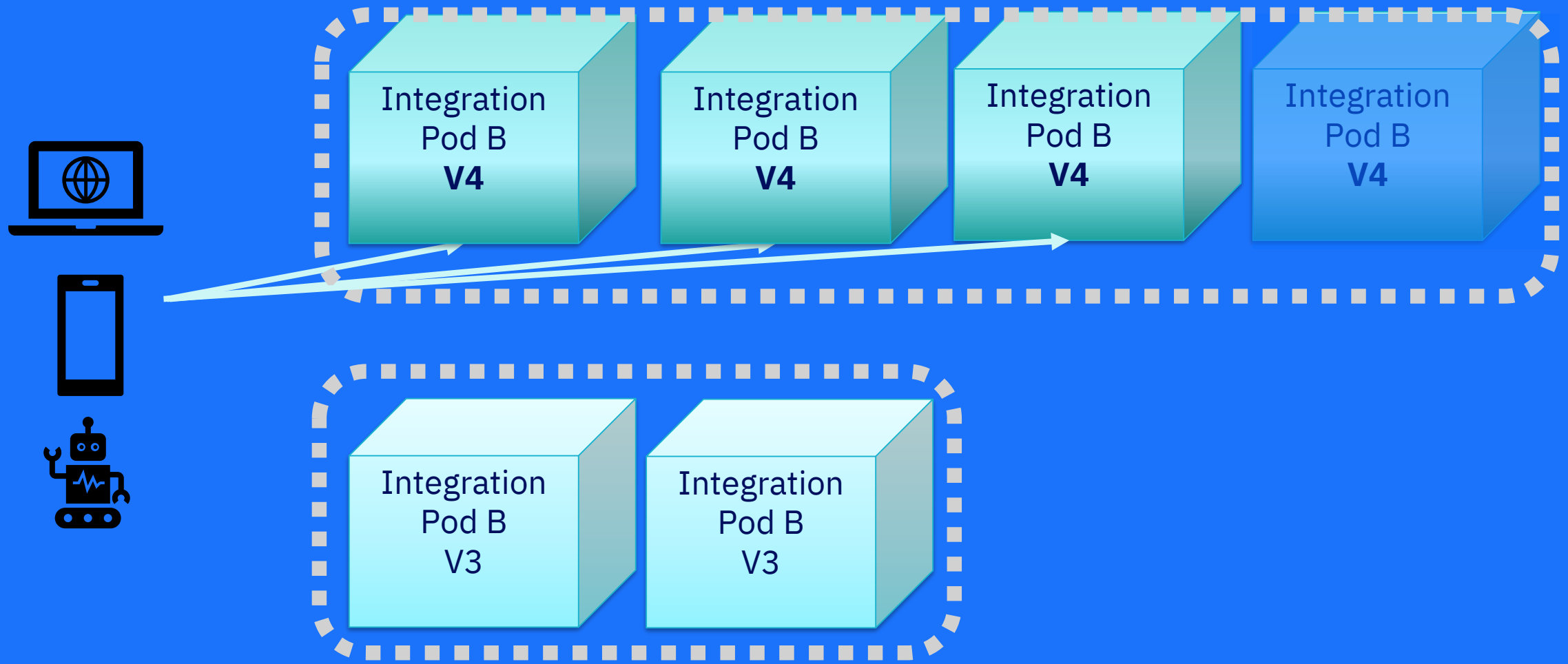
Enough pods? Reroute requests & drop old version



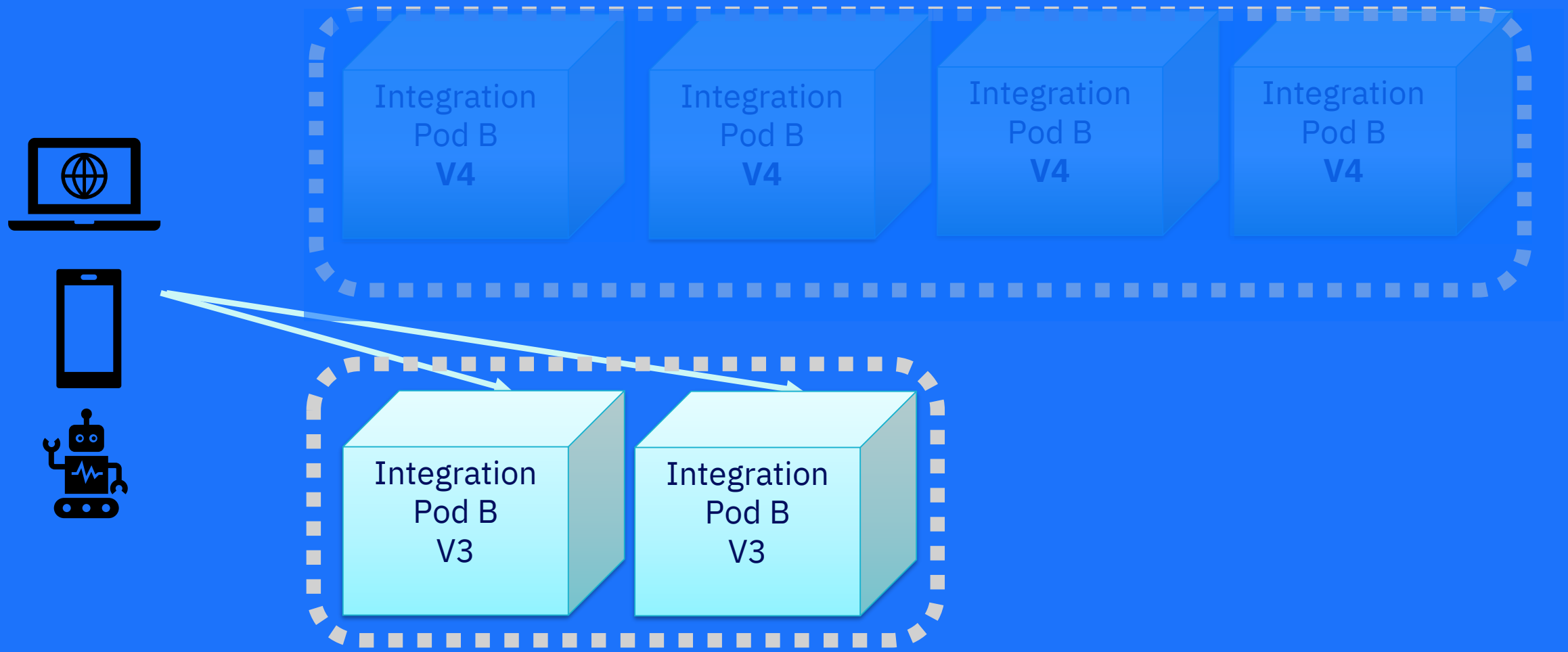
And finally fully roll out new version



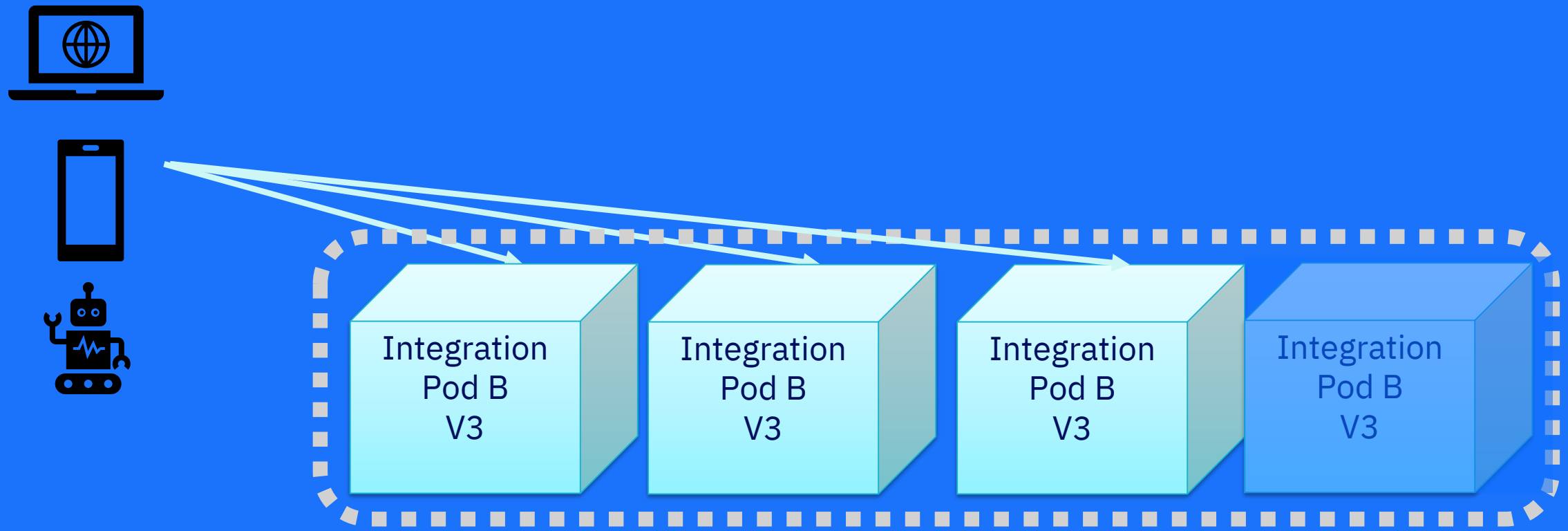
Rollback? Consider same ‘roll forward’ pattern



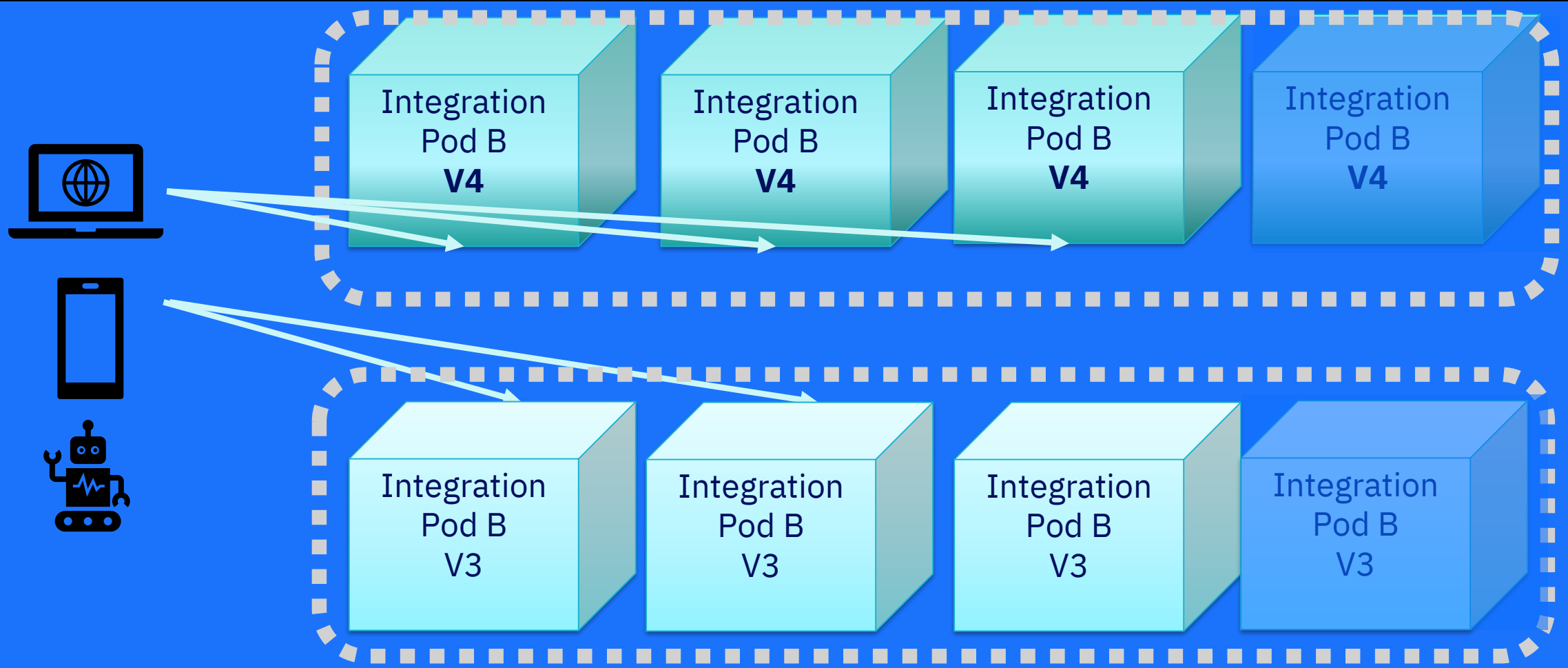
Redeploy immutable V3 containers/pods/chart And remove ‘bad’ version



And finally we are back where we were before



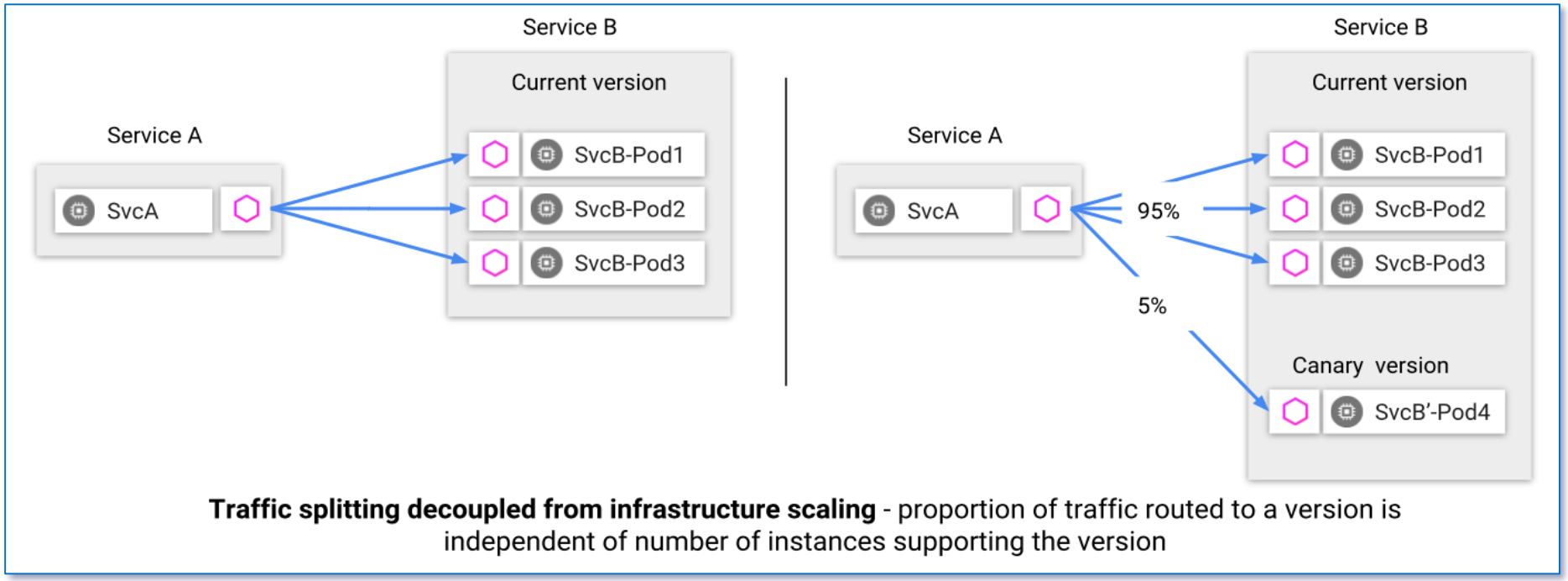
Can we run both in parallel? Yes. How to route requests?



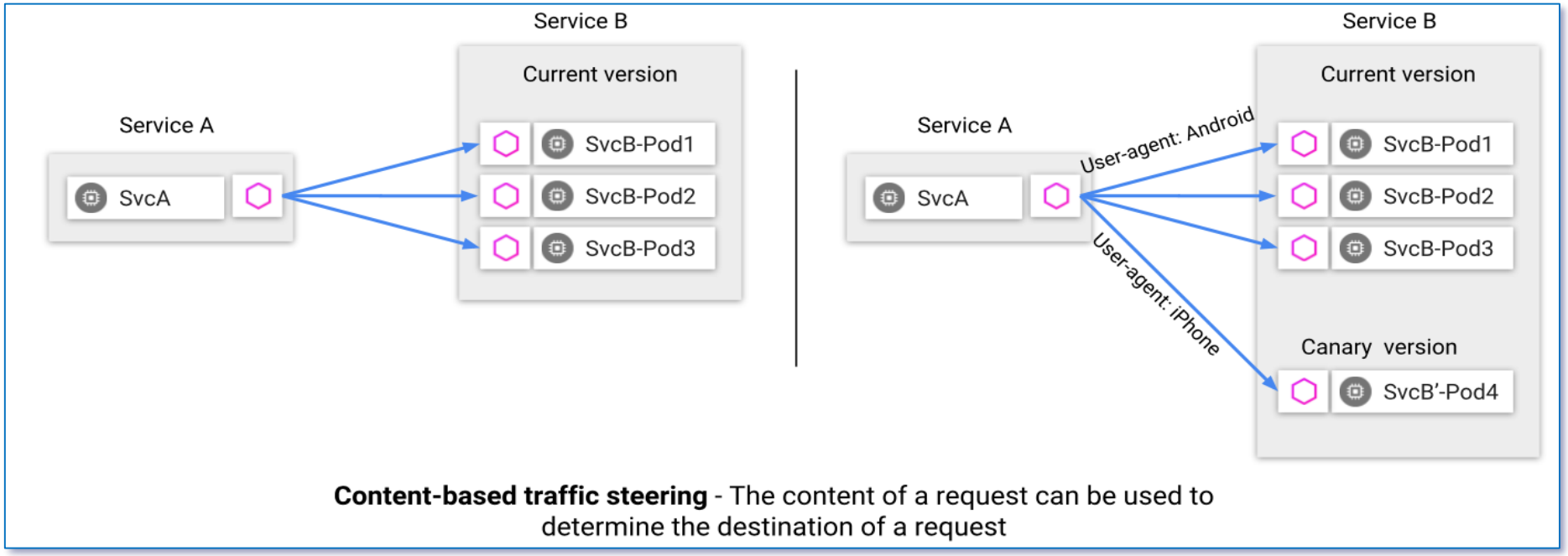
Use ISTIO Service Meshes and Envoy proxies to route traffic with rules.

Use your infrastructure where you can.

Don’t ‘Code’ it in App Connect if you can delegate to the infrastructure.



Traffic splitting decoupled from infrastructure scaling - proportion of traffic routed to a version is independent of number of instances supporting the version



Content-based traffic steering - The content of a request can be used to determine the destination of a request

Service Mesh Concepts

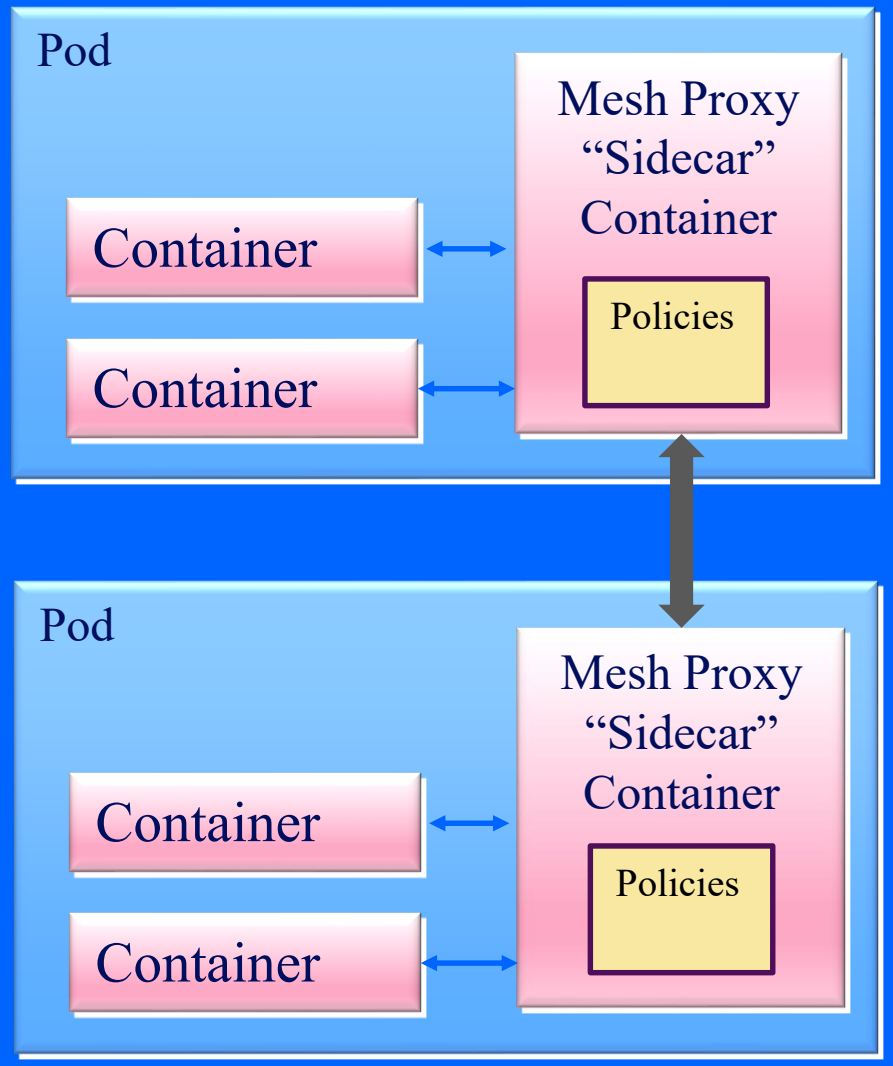
Service Mesh Proxy is deployed as a ‘sidecar’ container in the pod and implements a network proxy.

All traffic in and out MUST go through the proxy (enforced at ‘network’ level – the sidecar updates iptables on init)

One proxy per pod, so scalable

No code change to application /integration containers

You can put logging, MTLS, load balancing, routing etc etc in the mesh Proxy, not App Connect – keeps integration .bars simpler



Agile Integration Architecture

Modernizing integration to enable ***business*** *agility*

Fine grained deployment

Architecture & design

→ Improve build independence and production velocity (***deployment*** *agility*)

Decentralized Ownership

People & Process

→ Accelerate agility and innovation (***development*** *agility*)

Cloud native infrastructure

Infrastructure & Technology

→ Dynamic scalability and inherent resilience (***operational*** *agility*)

Key Takeaways

Know WHY you want to 'Be Agile'

Know what outcomes and value you want Agile to give you.

Remove existing pain points

Consider your granularity: it will affect your agility.

Define your deployment and test boundaries

Find out what changes – and how often!

Embrace the container concept: Remove dependencies. Bind in your flows and App Connect versions.

Same image in Dev, Test, Prod: Inject configuration

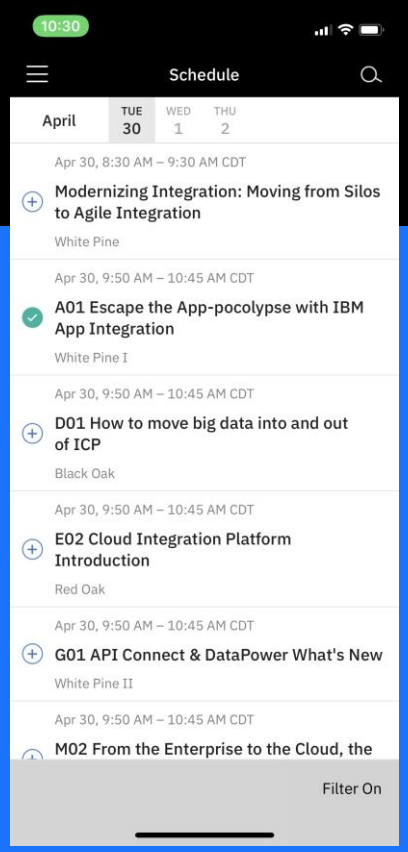
Let your containers run anywhere

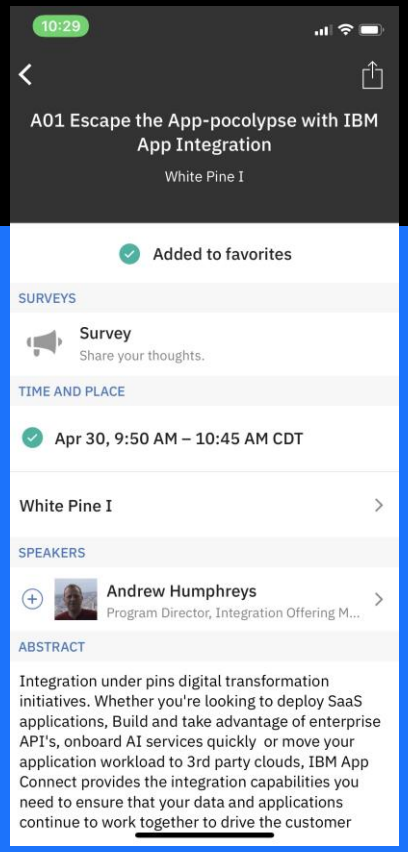
Delegate NFRs to your Agile Architecture

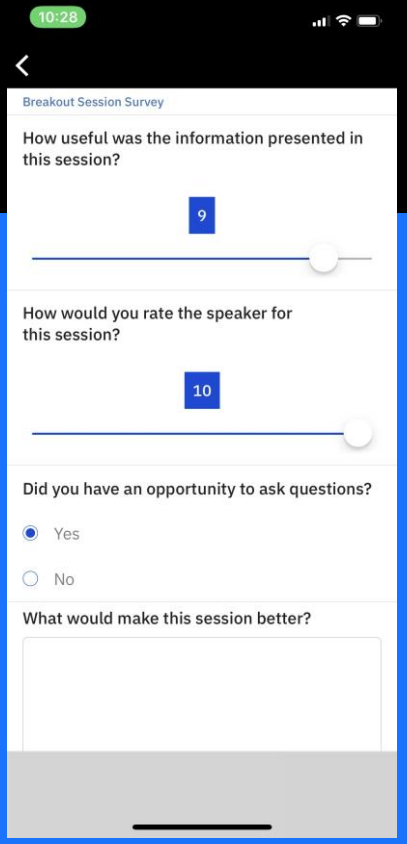
Use K8s and Service Meshes for endpoint routing & config

Use K8s for continuous availability

Use centralized logging: The number of containers will change dynamically.







Don’t forget to fill out the survey!

Select your session, select survey, rate the session (A06) and submit!

Thank You