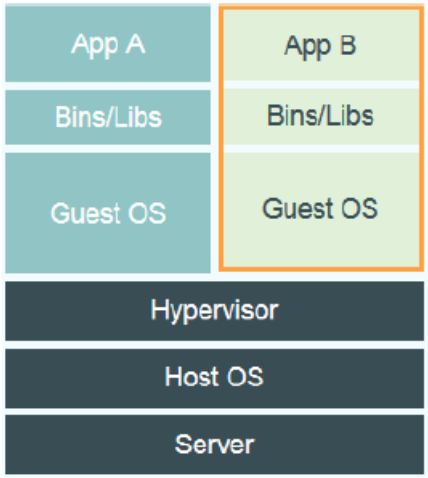
CHALLENGES AND OPPORTUNITIES IN DEVELOPING CONTAINER CLOUDS:
LESSONS LEARNED FROM IBM CONTAINER SERVICE

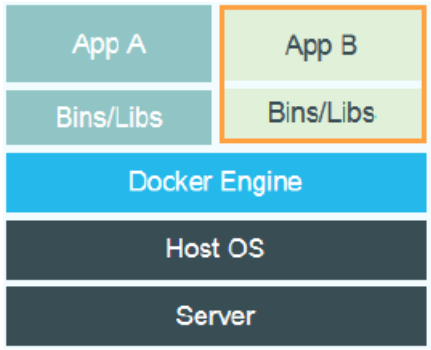
Gosia Steinder, IBM Research

What is Docker?

Does anyone in the room need to see this slide?



Virtual Machine



Docker Container

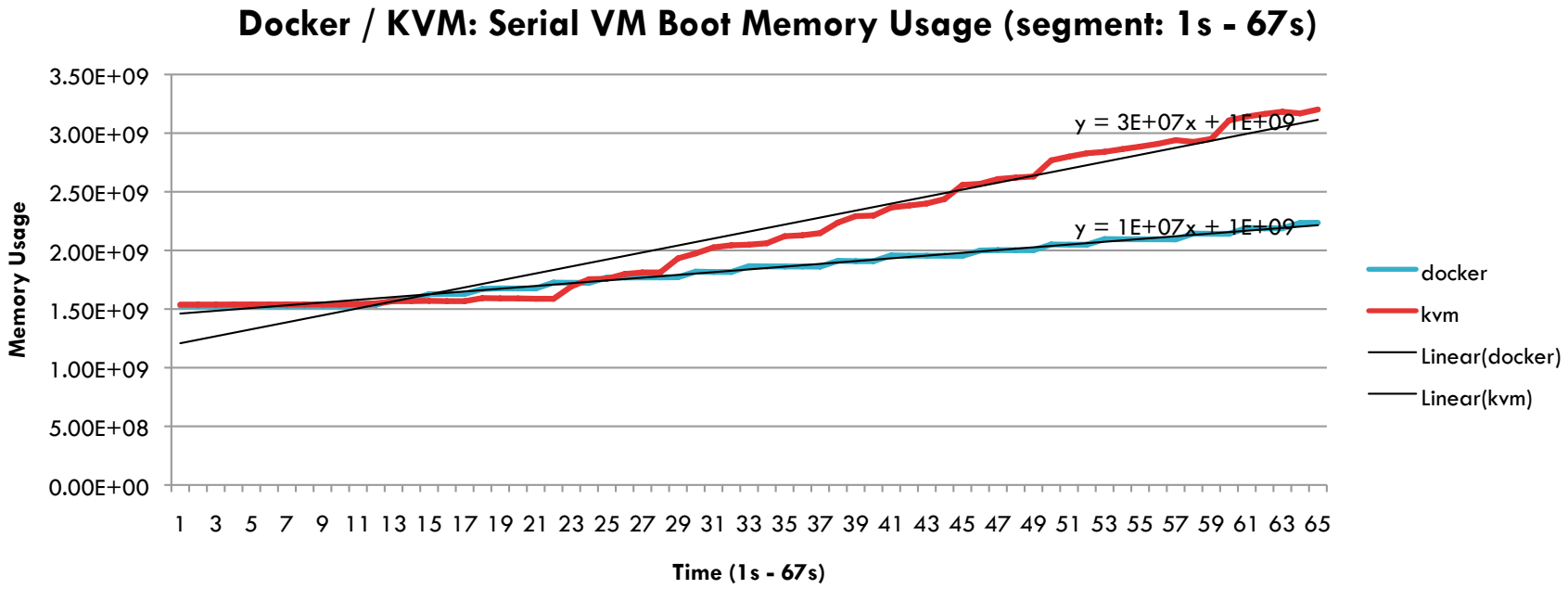
Docker = Linux namespaces + cgroups + overlay file system + image format

Docker = Linux namespaces + cgroups + overlay file system + image format
Docker = Linux namespaces + cgroups + overlay file system + image format

Why Docker?

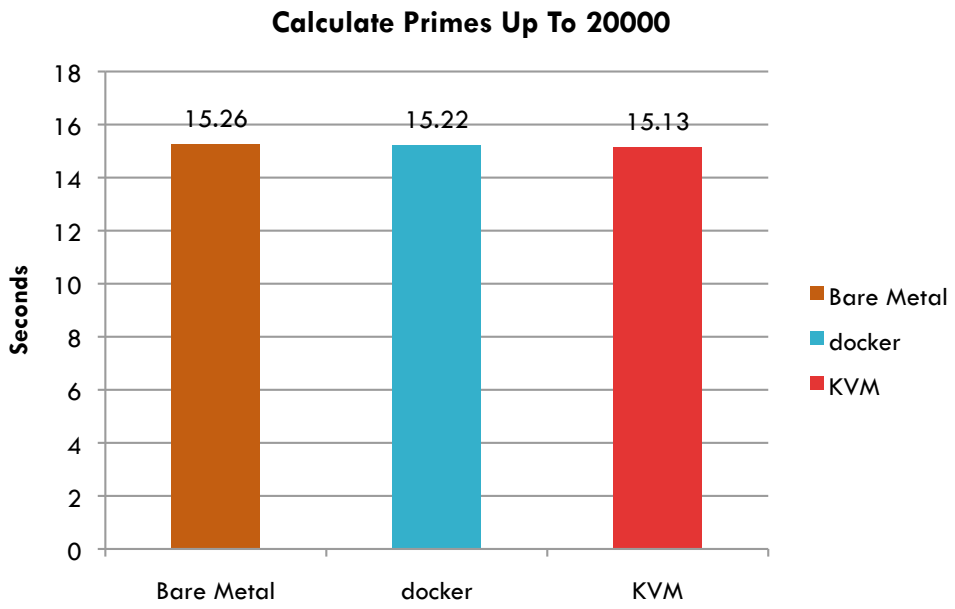
- **High Density**: Because containers share the same kernel and libraries we can run more applications on a server.
- **Fast Start up**: Because containers may have several layers in common only the new layers need to be copied, reducing build/transfer/boot/load times dramatically.
- **Portability** across environments
  - Deploying a consistent production environment is hard. Even if you use tools like chef and puppet, there are always OS and library updates that change between hosts and environments.
  - Docker gives us the ability to snapshot the OS into a common image, and, when combined with IBM's patterns technology will make it easy to deploy a collection of images comprising a given workload in another collection of Docker hosts.
- **Ecosystem**: Large and rapidly growing ecosystem of devops tools radically changing the way applications are developed, architected, packaged, and managed

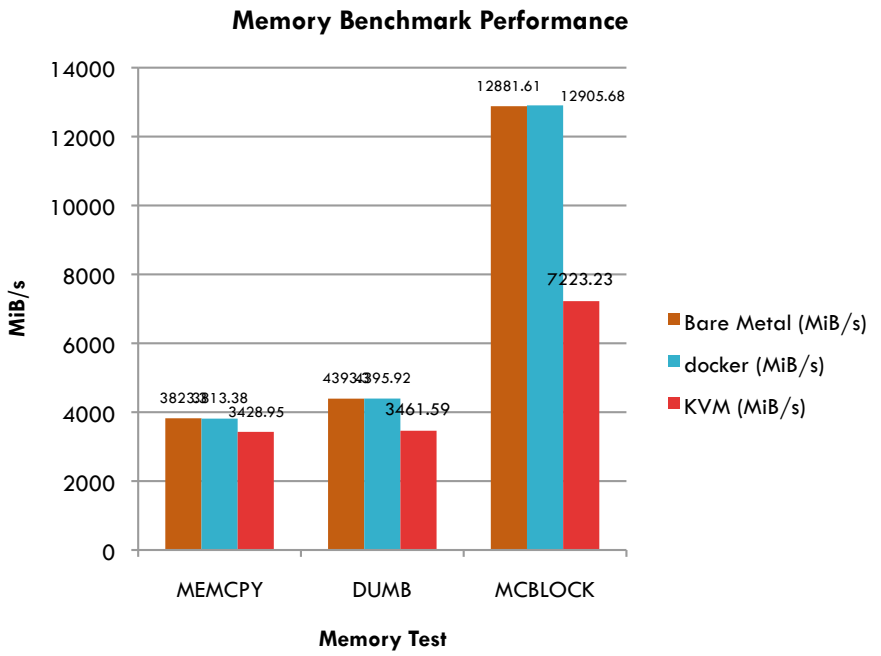
Containers: High-density advantage



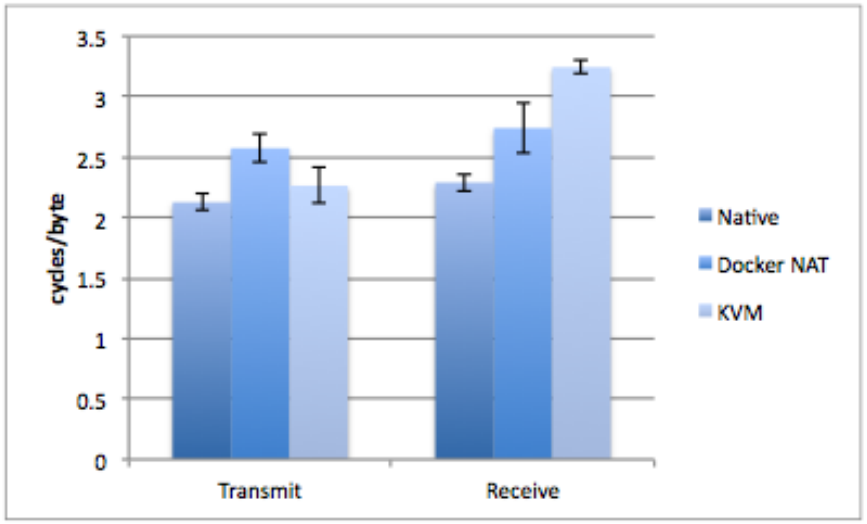
(Russell Boden study)

Containers: Performance advantage

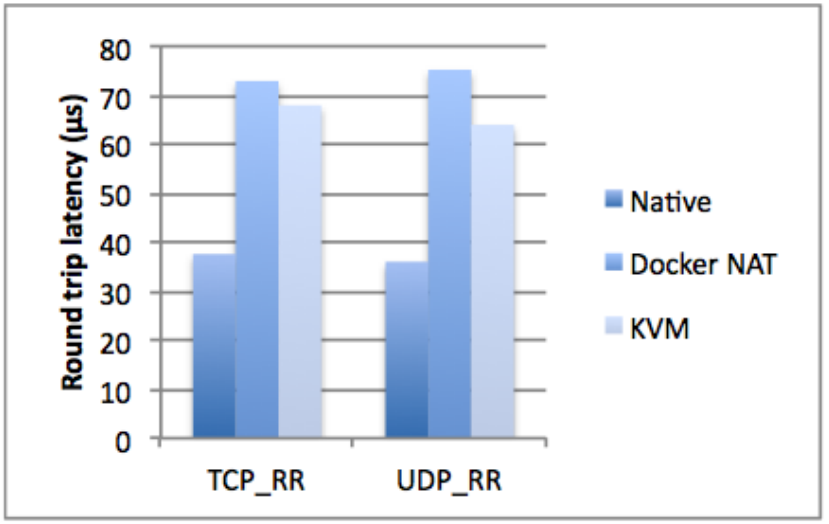




Docker: Networking challenge



Bulk transfer rate measured using perf stat -a



Network transfer latency measured using netperf

Docker default networking (docker bridge) introduces considerable network overheads and exhibits performance inferior to KVM. Docker host networking matches native performance.

Alternative container networking options

Measuring network throughput using file transfer in iperf

	Host network (native)	Container with Linux Bridge	Container with OVS	VM
Mean Gbps	33.9	17.9	14.3	14.3

Building a container cloud: Key challenges

- Elasticity – need to aggregate multiple hosts into system that can appear as a single docker host and be grown and shrank elastically
- Multi-tenancy – need for cost-effective and secure resource sharing and isolation
- Ecosystem – need to develop and support a large system of tools to enable workload development, deployment, and life-cycle management
- Hybrid – "there is no cloud but hybrid-cloud"; 70% of surveyed companies are or planning to be hybrid cloud users by 2017
- Visibility and control – give users what they need, let them know what they get, don't constrain usage for the sake of simplicity (this can always be done using abstractions)

API choices

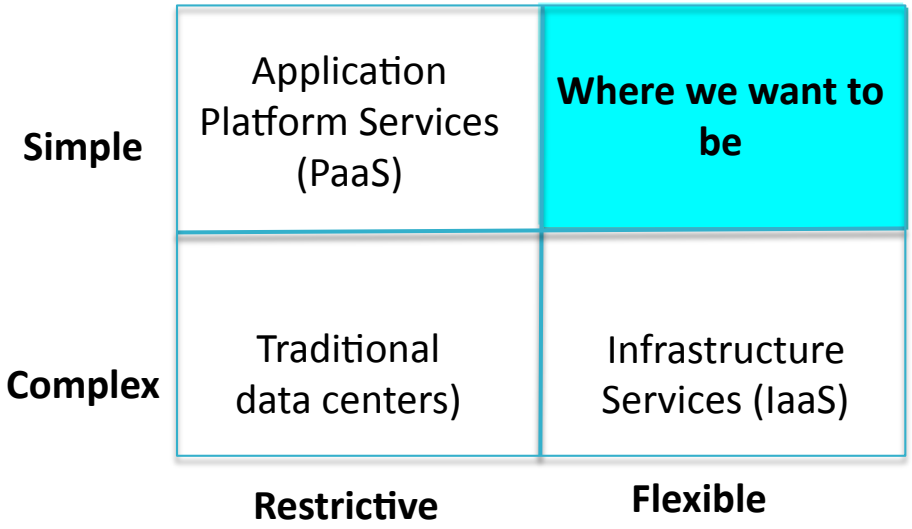
- Docker API
  - just as you see it on the local host
  - or as close as possible to that
- PaaS-like interfaces
  - Marathon
  - Kubernetes
- Workload-specific interfaces – e.g., Hadoop/Spark

What kinds of API?

Cloud computing conundrum:

Rigid abstraction layering constrain simultaneous flexibility & simplicity

- IaaS offer flexible programmable infrastructure, while workload management complexity is left to LoB users (e.g., virtual system patterns)
- SaaS & PaaS hide complexity but restricts choices and cannot address the heterogeneity of existing workloads (e.g., virtual application patterns, Cloud Foundry)



Docker API as the base abstraction helps us offer the flexibility and simplicity at the same time.

Technology choices

	Kubernetes	Marathon	Openstack
API	PaaS	PaaS	IaaS
Network	✗	✗	✓
Multi-tenancy	✗	✗	✓
Advanced scheduling	✗	✓	✓ (using private extensions)
Community*	?	?	?

* All these technologies have strong communities but none of them is targeting Docker users specifically.

Delivery models

- Shared container service
  - Cheap to build and operate
  - Security issues
- Dedicated in VMs
  - Expensive to build and operate
  - Improved security
- Shared with isolation policies
  - Cheap to build and operate (with extra cost and effort for isolation)
  - Improved security (eliminates Docker-added vulnerability risk)

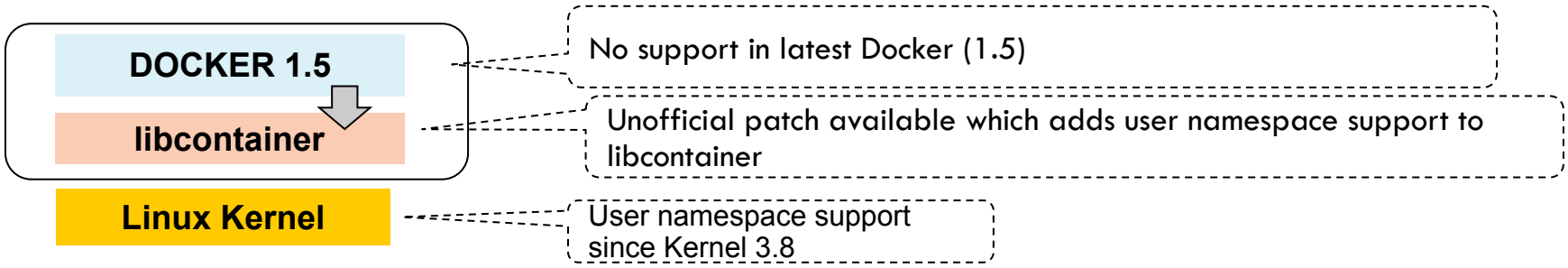
Comparing Docker Security to Virtualization Technologies

***Assuming that Docker adds User Namespace support**, how would it compare to VM-based systems?*

- **To escape a secure Docker, an attacker in a container would need to find a privilege escalation attack on the shared kernel. Such kernel vulnerabilities occur roughly once a year (the last was discovered in June 2014)**
- **To escape a type 2 hypervisor such as KVM:**
  - an attacker in a guest VM would need root in the VM, would need to find a vulnerability in QEMU, and then also find a privilege escalation attack on the native kernel. QEMU vulnerabilities also are discovered roughly once a year (the last was found in May 2014)
  - Statistically, over the past few years, **it is roughly half as likely to find both a QEMU and Kernel vulnerability at the same time, as just finding the kernel one, and this combination occurs roughly every 2 years**
- **To escape a type 1 hypervisor, such as VMware ESX:**
  - An attacker in a guest VM would need root in the VM, and would need to find a vulnerability in the VMware hypervisor.
  - The last such vulnerability in ESX was in February 2013, and this occurs roughly once every **2 years**
- **In terms of side and covert channels, there is no significant difference**
- **The bottom line is that Docker has a greater risk of vulnerability, since a single kernel vulnerability is sufficient to completely break the system**
  - *Since patch management will be done with delay it may not be sufficient to protect the system*

Docker: User namespace isolation

- **Docker has a high risk of containment failure unless user namespaces are used to separate root in the container from root outside**
  - **Docker 1.5 still does not have user namespace separation and thus is NOT secure**
  - **Docker is working on adding user namespace support; it is anticipated "soon", but no specific timeline exists. It will likely not appear until 1.7 (Q3?) at the earliest**
- All other reported vulnerabilities, to date, have been fixed and we have not discovered any new significant ones

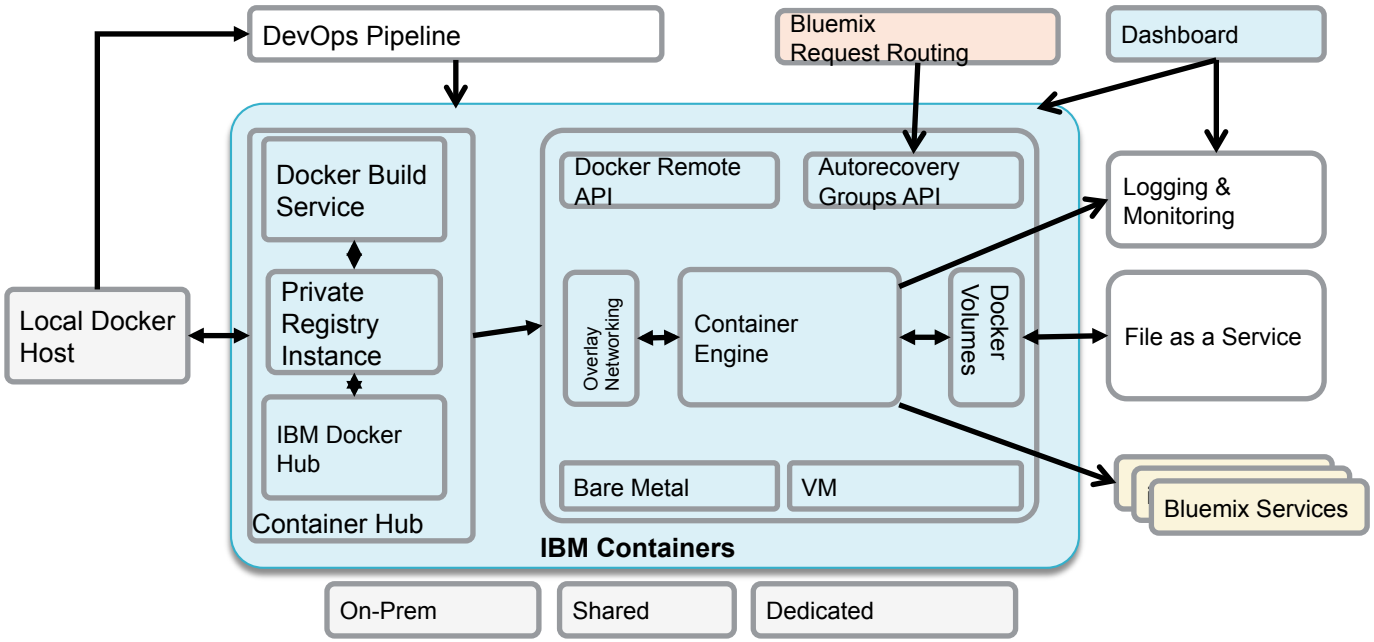


Our Point of View

"Containers are the foundation of our cloud…
and our cloud is tailored for Enterprise"

- Cloud workloads are built and run as containers
- Focus on Hybrid Cloud Application Portability
- Bare metal deployment, enterprise security, scale and resiliency
- Leap frog (and marginalize) hypervisors

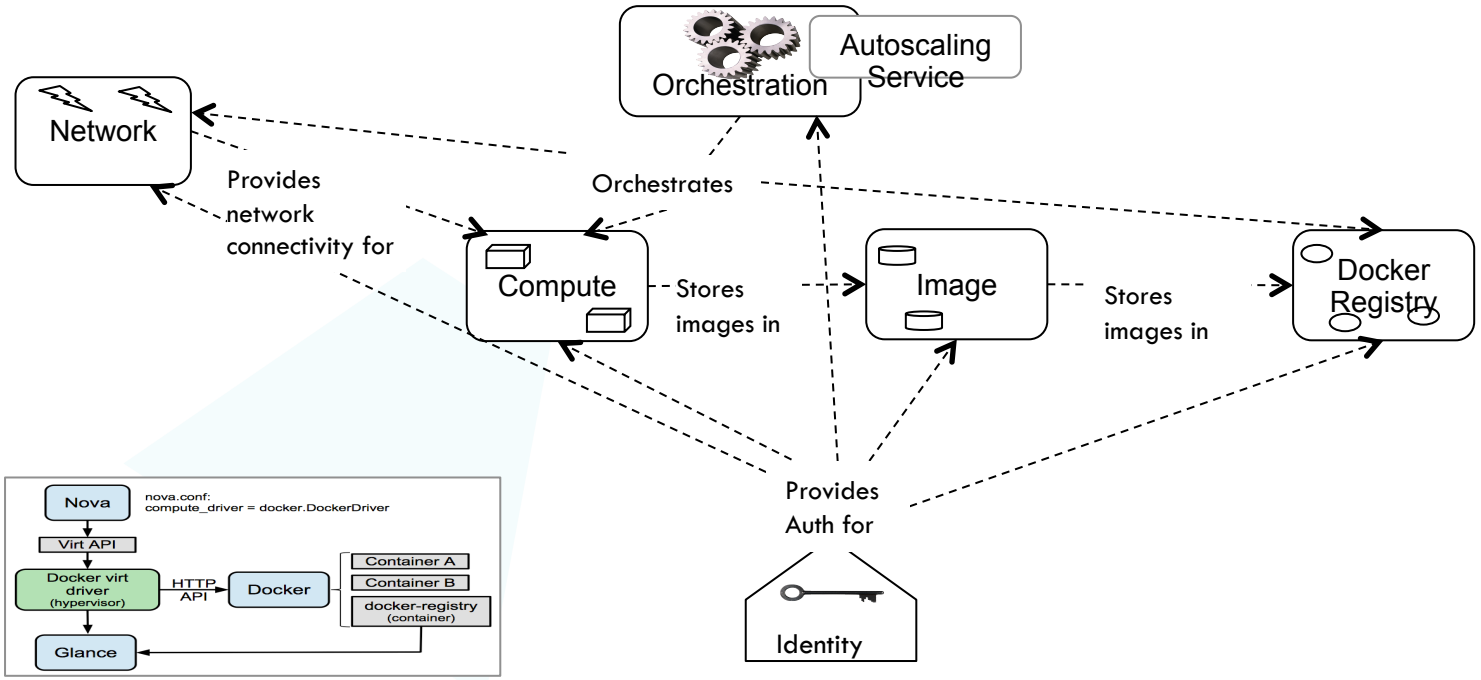
Reference architecture



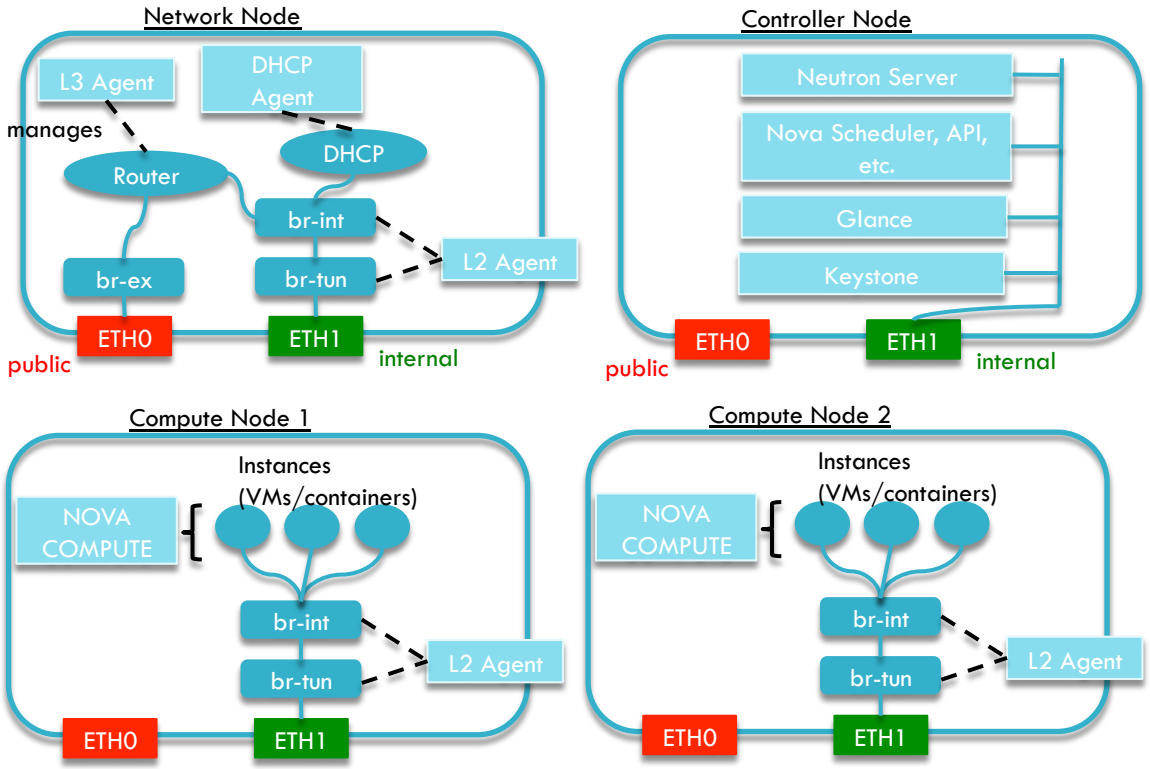
Key goals

- Container Hub - Private and public registries with enterprise-ready content
- Full lifecycle management for both a container, a composite application comprised on multiple containers and the container runtime environment itself
- DevOps Build Pipeline for container images and multi-container templates
- Container Service
  - Multi-tenant and single-tenant deployment plans
  - Exploitation of bare metal
  - Support for template model that includes multi-container single-host (MCSH) and multi-container multi-host (MCMH) models
  - Policy based resource management (placement, cleanup, movement)
- Networking
  - Support for private overlay network between a group of containers
  - Fine grained connectivity control within tenant network via subnets and security groups
- Storage
  - Support for container movement between hosts without loss of core container filesystem
  - Support for persistent non-brittle volume attach
- Enterprise Enablement – security, performance, availability, visibility, control, content

Our initial implementation of the Container Engine



Networking in Openstack



- ✓ Multi-host networking
- ✓ Private networks
- ✓ Dynamic IP assignment
- ✓ Public floating Ips
- ✓ Network quotas
- ✓ Security groups

Operational visibility

Scenario 1: Ephemeral Instances

- ***Containers for App A fail shortly after provisioning. Reprovisioning automation results in the same systemic failure.***

  - How to root cause the issue when containers keep dying before we can access them?
  - How to avoid cascading failures?

Scenario 2: Unresponsive Systems

- ***My app stopped responding. Access to the Docker instance fails, and all my in-app monitors went completely silent.***

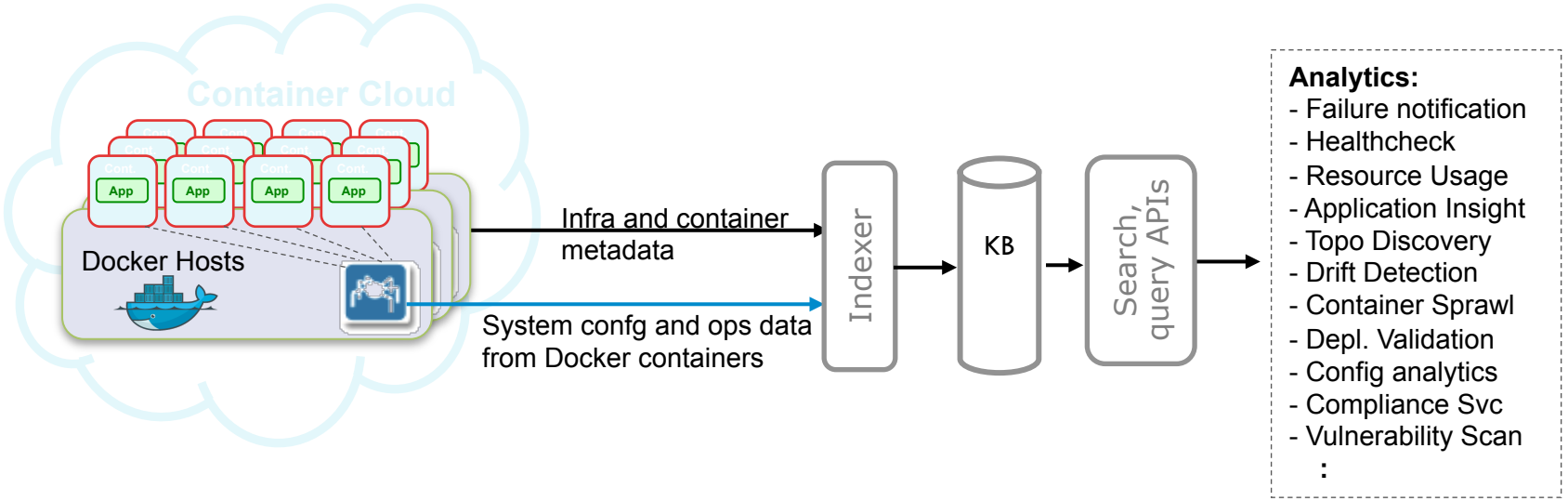
  - In-band monitoring solutions fail at the exact moment we need them the most.
  - How can we provide a better, always-on solution for health, monitoring, compliance, etc.?

Scenario 3: Agent Updates across Entire Inventory

- ***Transitioning from shiny tool S to shinier tool E for operational monitoring. Need to reprovision each of our 1000 instances with the new runtime component.***

  - DevOps and CD surely helps; but still, how fun:)
    - Is the risk worth the effort? How often can we do these (we have baggage)

Seamless monitoring with Docker crawlers

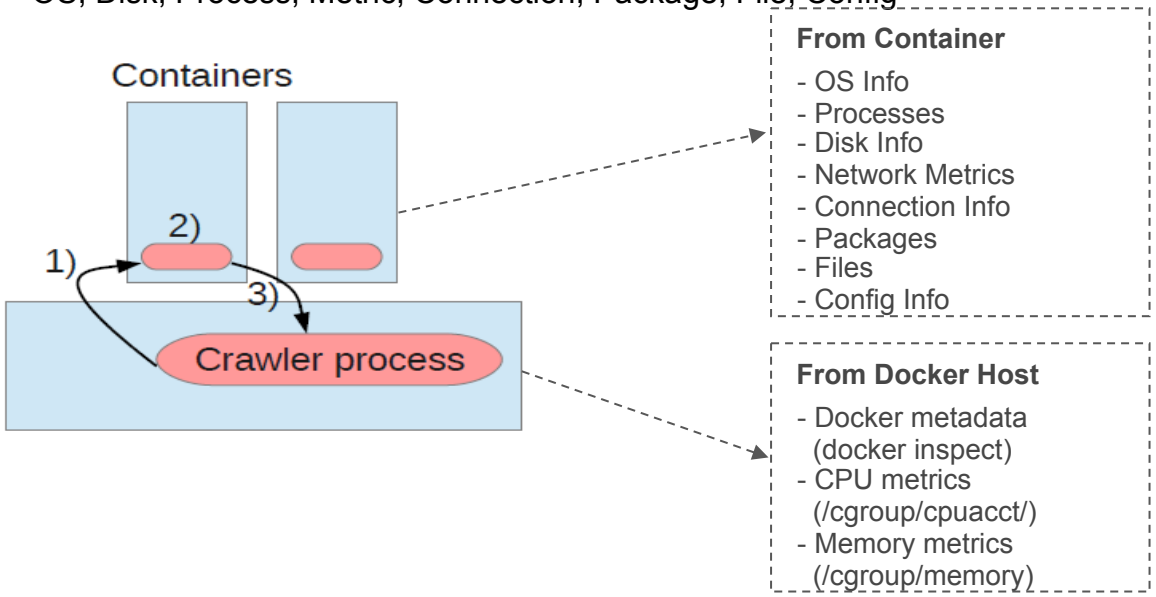


Our Approach:

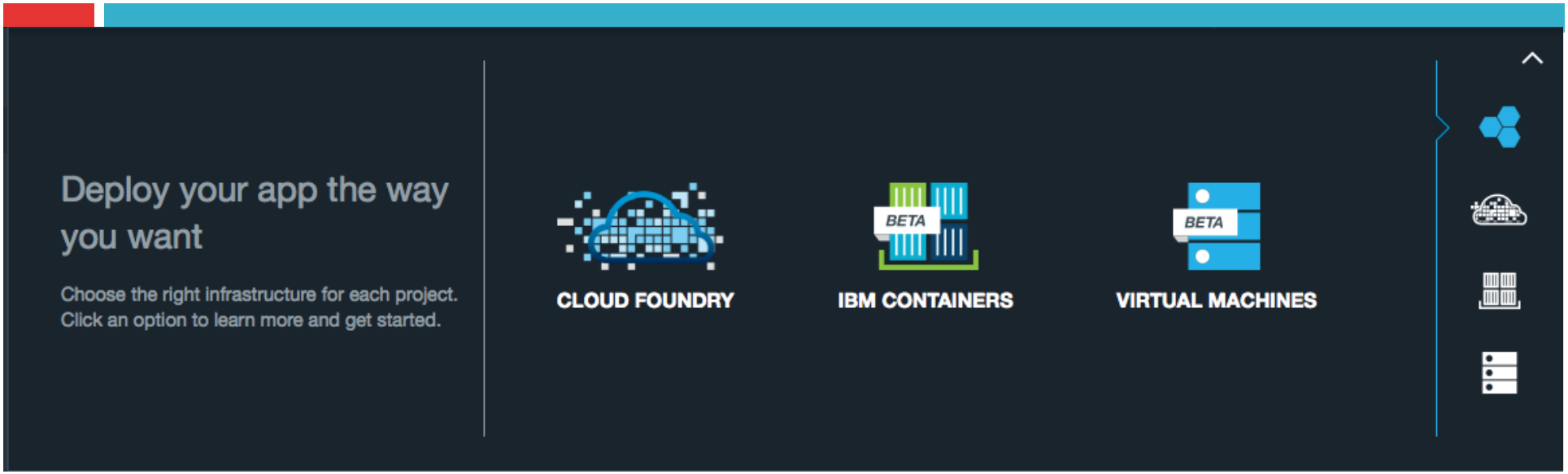
- Seamlessly "crawl" the cloud like we crawl the web
- Query/mine the cloud like we query/mine the web

Docker crawler data

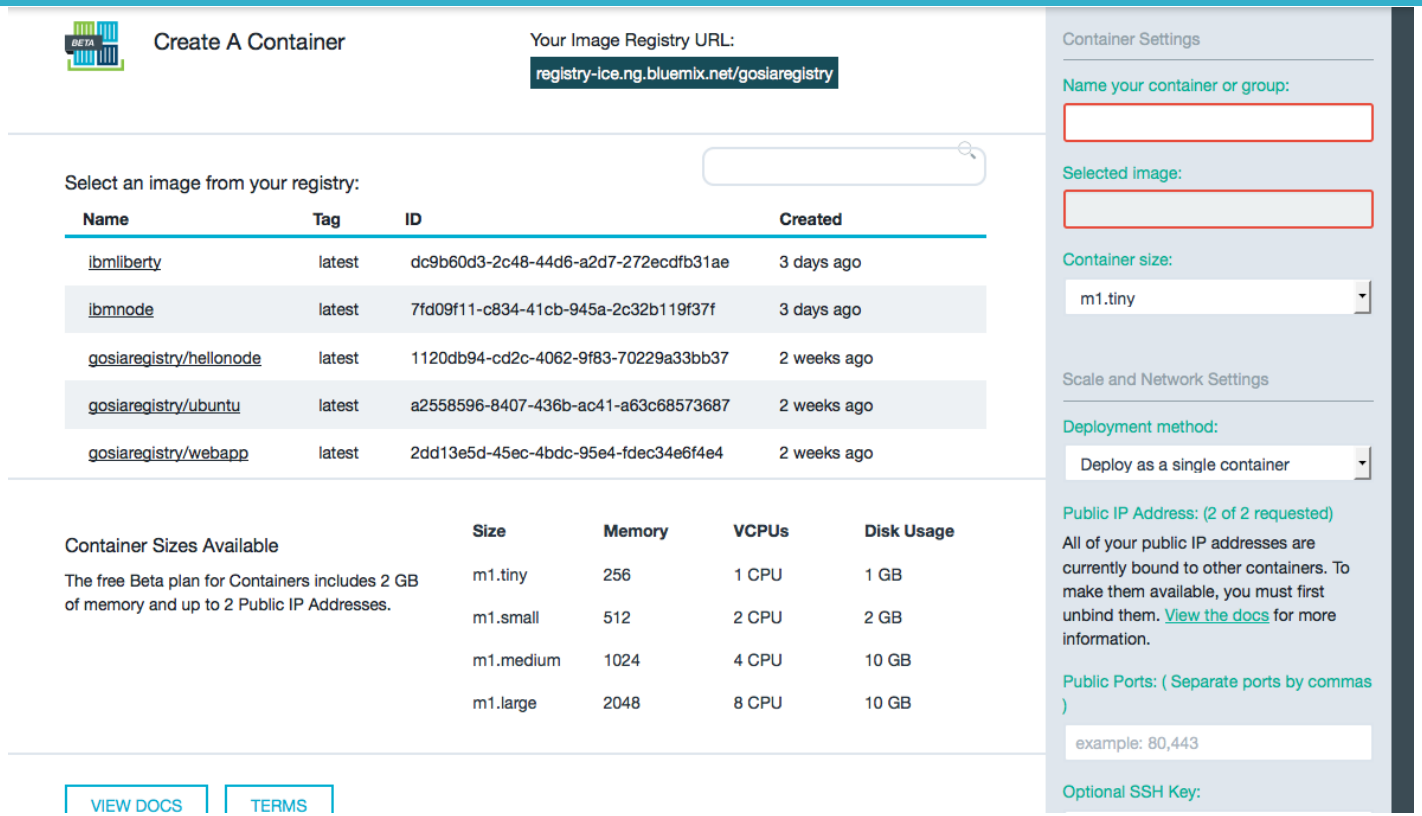
- System state: Persistent (file system) + Volatile (OS memory context)
- Features:
  - OS, Disk, Process, Metric, Connection, Package, File, Config



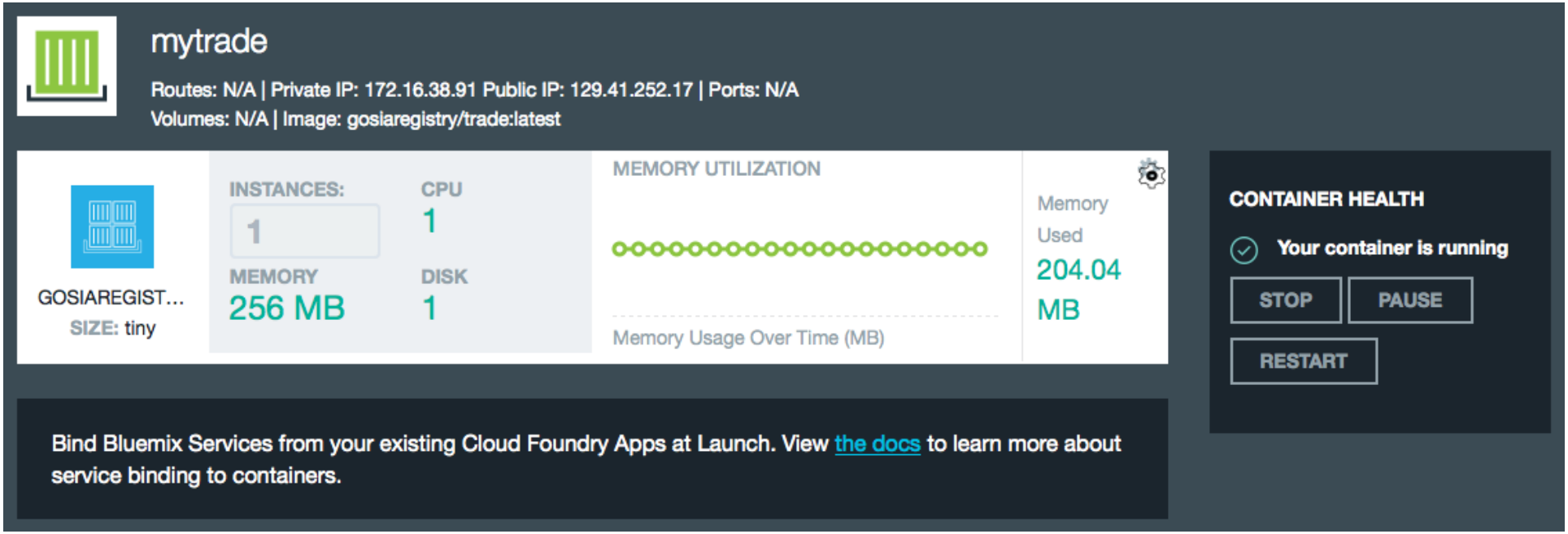
What it looks like?



What it looks like?



What it looks like?

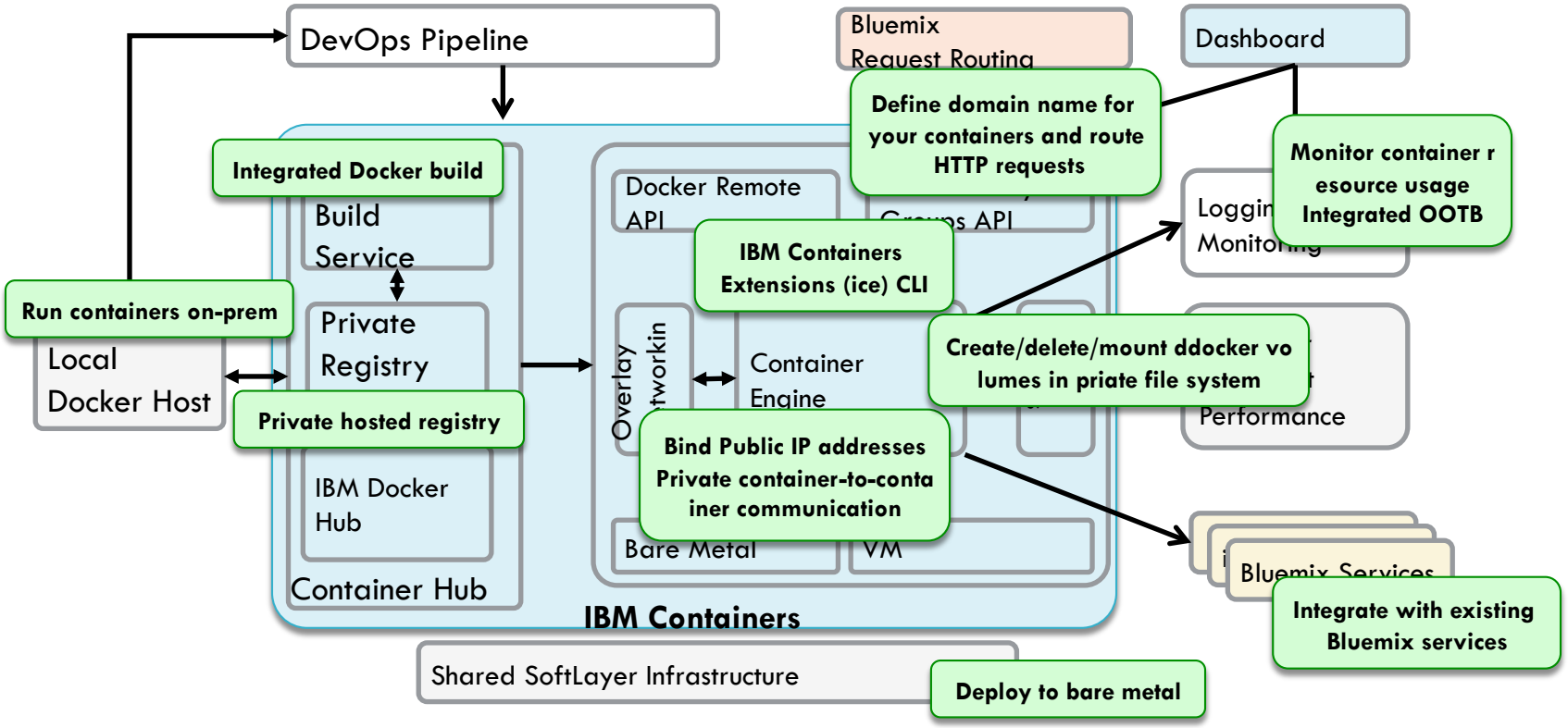


What it looks like?

```
> ice run --help
usage: ice run [-h] [--name NAME] [--memory MEMORY] [--env ENV]
               [--publish PORT] [--volume VOL] [--bind APP] [--ssh SSHKEY]
               IMAGE [CMD [CMD ...]]

positional arguments:
  IMAGE                 image to run
  CMD                   command & args passed to container to execute

optional arguments:
  -h, --help            show this help message and exit
  --name NAME, -n NAME  assign a name to the container
  --memory MEMORY, -m MEMORY
                        memory limit in MB, default is 256
  --env ENV, -e ENV     set environment variable, ENV is key=value pair
  --publish PORT, -p PORT
                        expose PORT
  --volume VOL, -v VOL  mount volume, VOL is VolumeId:ContainerPath[:ro],
                        specifying ro makes the volume read-only instead of
                        the default read-write
  --bind APP, -b APP    bind to Bluemix app
  --ssh SSHKEY, -k SSHKEY
                        ssh key to be injected in container
```

```
> ice --help
usage: ice [-h] [--verbose] [--cloud | --local]

        {logs, ip, images, rmi, login, help, ps, pause, group, namespace, start, version,
build, rm, unpause, run, inspect, stop, volume, restart, info, search, route, login}
        ...
```

What was accomplished?



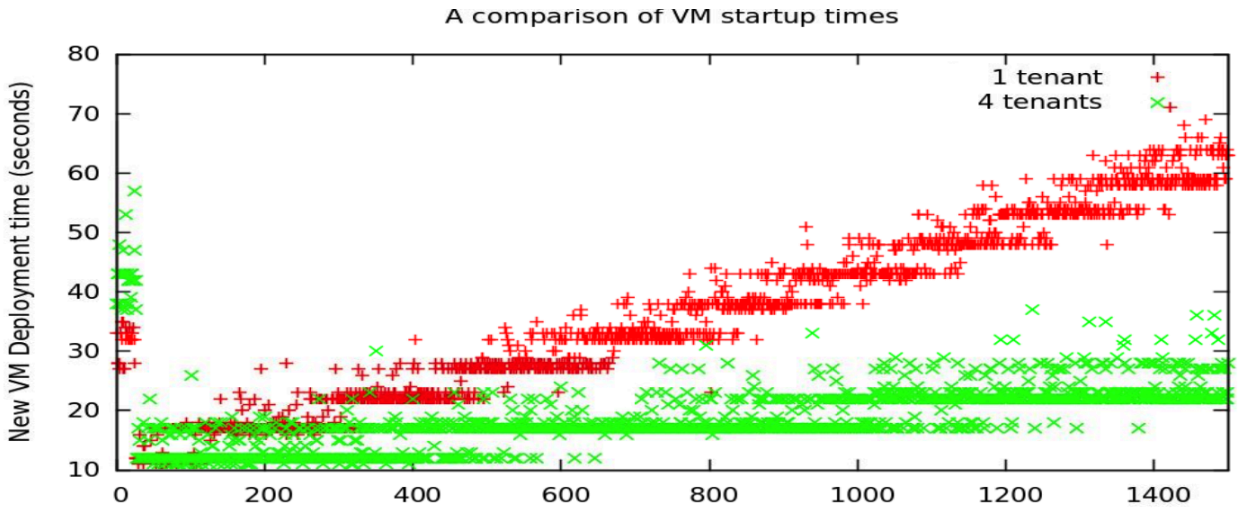
Lessons learned: Container-first approach

- We opted to Docker native APIs and stay as close to them as we can rather than a PaaS platform
  - Supports any workload (that fits in our container sizes!) – we see many types running
  - Easy to port workloads between local machine and our cloud
  - Easy to understand for anyone who knows Docker
- Conclusion: focus on container as a service and Docker compatibility is the right approach
  - Composite abstractions (such as POD, groups) can be built next or on top of this model
  - Extend community Docker CLI to work directly with our API (allow tenant tokens to be passed

Lessons learned: networking model

- We opted for every container-as-an-IP-host model
- Nice:
  - Gives appearance of a single host in a multi-host system
  - Full network as a service control: security groups, private networks, public IPs
  - More secure than host communication, more performant than Docker bridge
- Bad:
  - Costly – it takes time to allocate and configure a routable IP address
  - Limited scalability – each port has to be secured; IP table rules are costly

The cost of security groups



Measured with VMs, each VM with one port, all VMs share a default security group.
Startup time grows linearly with the number of endpoints on the same network.

Lessons learned: networking model

- Needed networking models:
  - IP host – for all workloads that require IP presence (not currently supported by docker)
  - Docker bridge – for all workloads that require only outbound connectivity
  - Network namespace shared with another container – for sidecars, replicated processes
  - No network – e.g., data containers
- Approach:
  - Experiment with alternatives to Nova+Docker driver, e.g., Docker swarm
  - Continue using Openstack Neutron
  - Prototype network extension for IP host capability leveraging leveraging Powerstrip
  - Work with community to re-architect Docker to allow pluggable Network Drivers (ongoing effort)

Next steps?

- Continuous delivery, continuous delivery, …
  - No challenge or difficulty outlined in this talk is as important and challenging as continuous delivery of the stack
- Continuous operations – we cannot be a beta service forever
- Lots of ideas for new features
  - Observe usage, learn from user feedback, prioritize …
- Hybrid
  - Hybrid-cloud scenarios will be our top priority

Thank you!

Contact: steinder@us.ibm.com