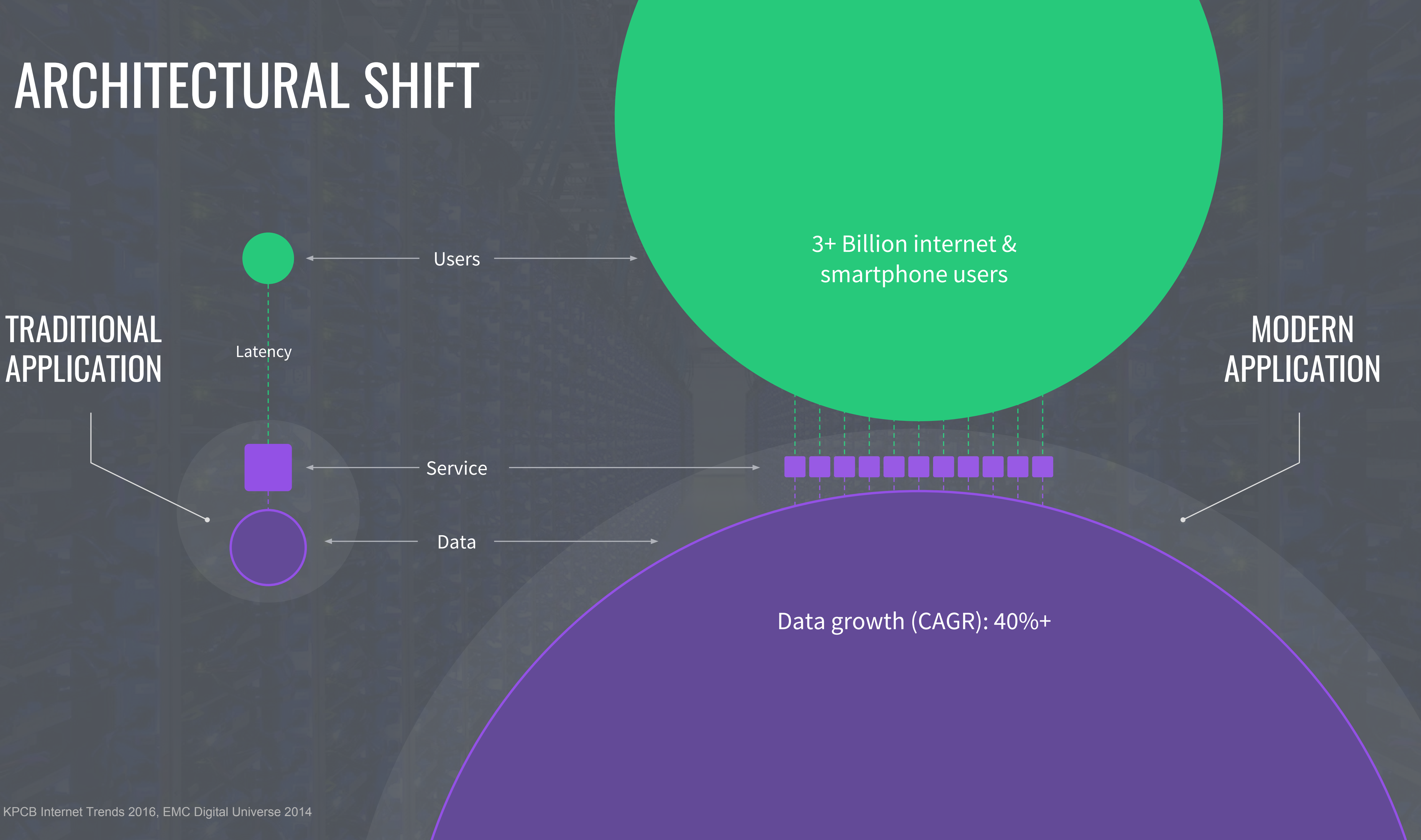
June 2017

DC/OS AND FAST DATA (THE SMACK STACK)

Benjamin Hindman - @benh

Elizabeth K. Joseph - @pleia2

ARCHITECTURAL SHIFT

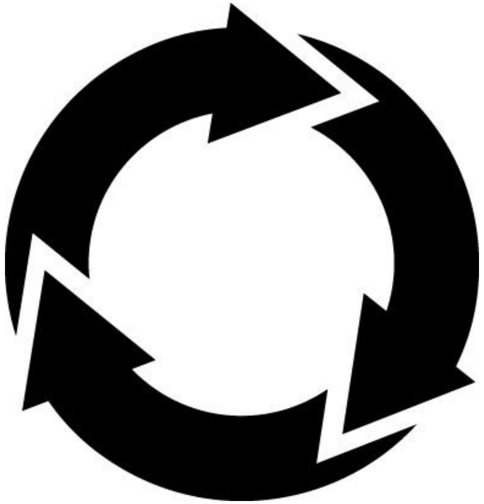


Source: KPCB Internet Trends 2016, EMC Digital Universe 2014

TODAY'S REINFORCING TRENDS

CONTAINERIZATION

MICROSERVICES



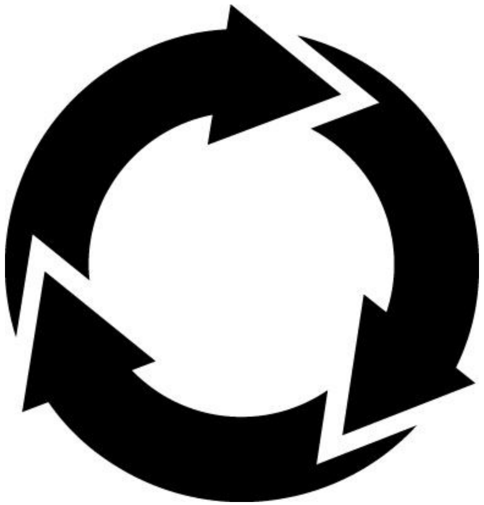
CONTAINER ORCHESTRATION

BIG DATA & ANALYTICS

TODAY'S REINFORCING TRENDS

CONTAINERIZATION

MICROSERVICES



CONTAINER ORCHESTRATION

FAST **~~BIG~~ DATA & ANALYTICS**

FROM BIG DATA TO FAST DATA

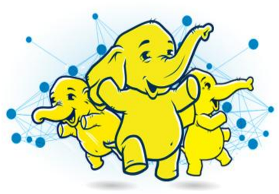
Days *Hours* *Minutes* *Seconds* *Microseconds*

Batch → **Micro-Batch** → **Event Processing**

Reports what has happened using descriptive analytics

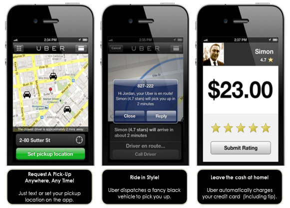
Solves problems using predictive and prescriptive analytics

Billing, Chargeback



Product recommendations

Real-time Pricing and Routing



Real-time Advertising

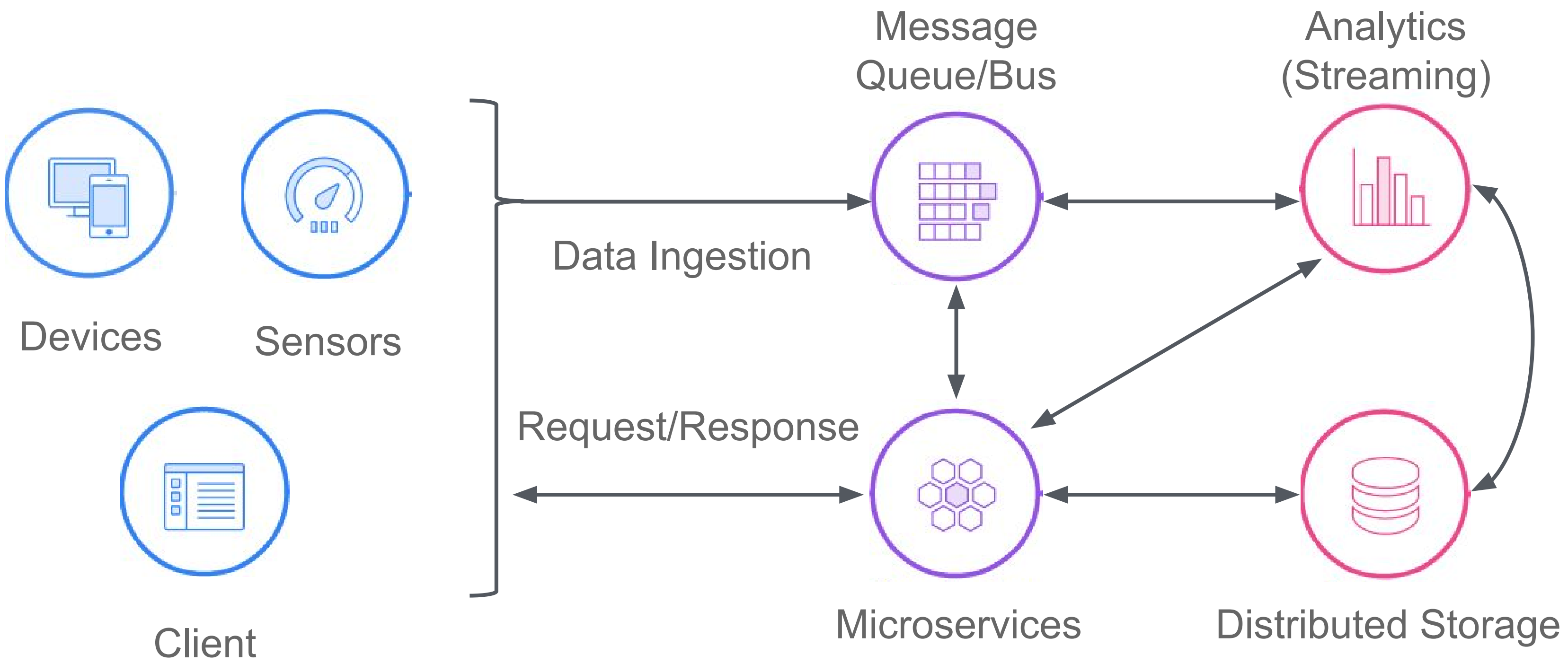
Predictive User Interface

ON THE EDGE, AND STILL REALLY BIG!

A380-1000: 10,000 sensors in each wing; produces more than 7Tb of IoT data per day

[1] https://goo.gl/2S4q5N

MODERN APPLICATION -> FAST DATA BUILT-IN

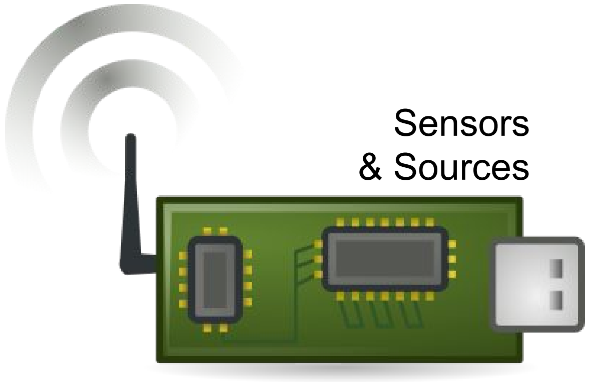


Use Cases:

- Anomaly detection
- Personalization
- IoT Applications
- Predictive Analytics
- Machine Learning

THE FOUNDATIONS OF FAST DATA

Data Processing



Sensors
& Sources

Message
Queue

Modern
Apps

MESSAGE QUEUES

Message Brokers

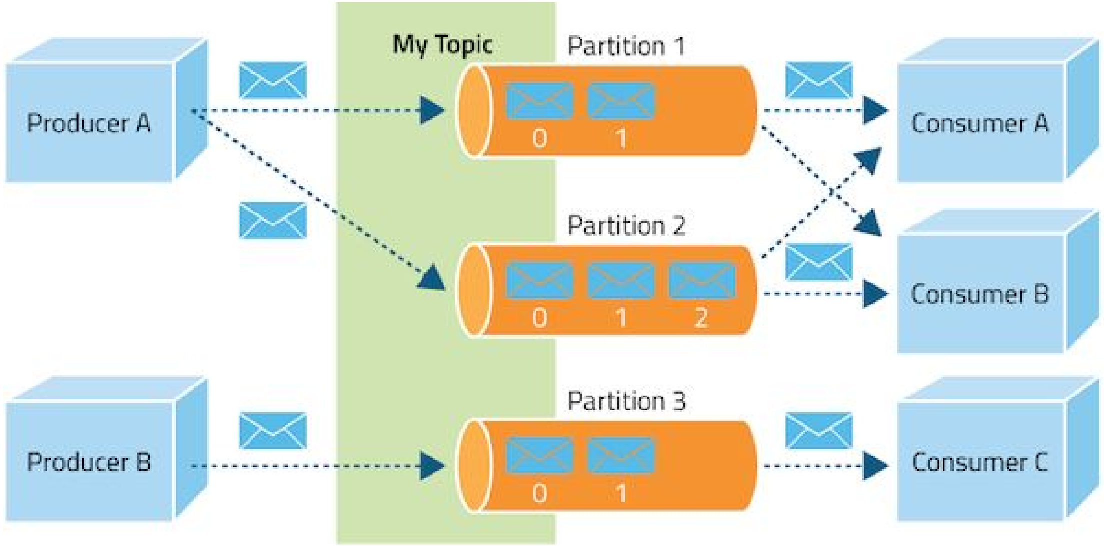
- Apache Kafka
- ØMQ, RabbitMQ, Disque

Log-based Queues

- fluentd, Logstash, Flume

see also queues.io

APACHE KAFKA



Typical Use: A reliable buffer for stream processing

Why Kafka?

- High-throughput, distributed, persistent publish-subscribe messaging system
- Created by LinkedIn; used in production by 100+ web-scale companies [1]

[1] https://cwiki.apache.org/confluence/display/KAFKA/Powered+By

DELIVERY GUARANTEES

- **At most once**—Messages may be lost but are never redelivered
- **At least once**—Messages are never lost but may be redelivered (Kafka)
- **Exactly once**—Messages are delivered once and only once (this is what everyone actually wants, but no one can deliver!)

Murphy's Law of Distributed Systems:

Anything that can go wrong, will go wrong … partially!

STREAMING ANALYTICS

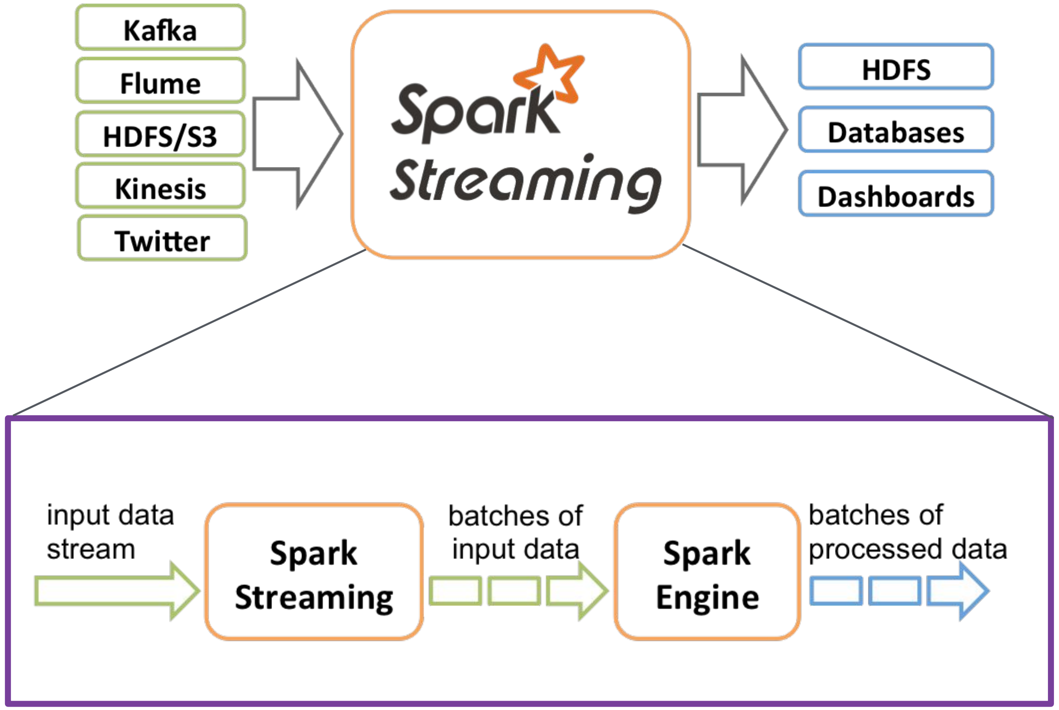
Microbatching

- Apache Spark (Streaming)

Native Streaming

- Apache Flink
- Apache Storm/Heron
- Apache Apex
- Apache Samza

APACHE SPARK (STREAMING)



Typical Use: distributed, large-scale data processing; micro-batching

Why Spark Streaming?

- Micro-batching creates very low latency, which can be faster
- Well defined role means it fits in well with other pieces of the pipeline

DISTRIBUTED STORAGE

NoSQL

- ArangoDB
- mongoDB
- Apache Cassandra
- Apache HBase

SQL

- MemSQL

Filesystems

- Quobyte
- HDFS

Time-Series Datastores

- InfluxDB
- OpenTSDB
- KairosDB
- Prometheus

see also iot-a.info

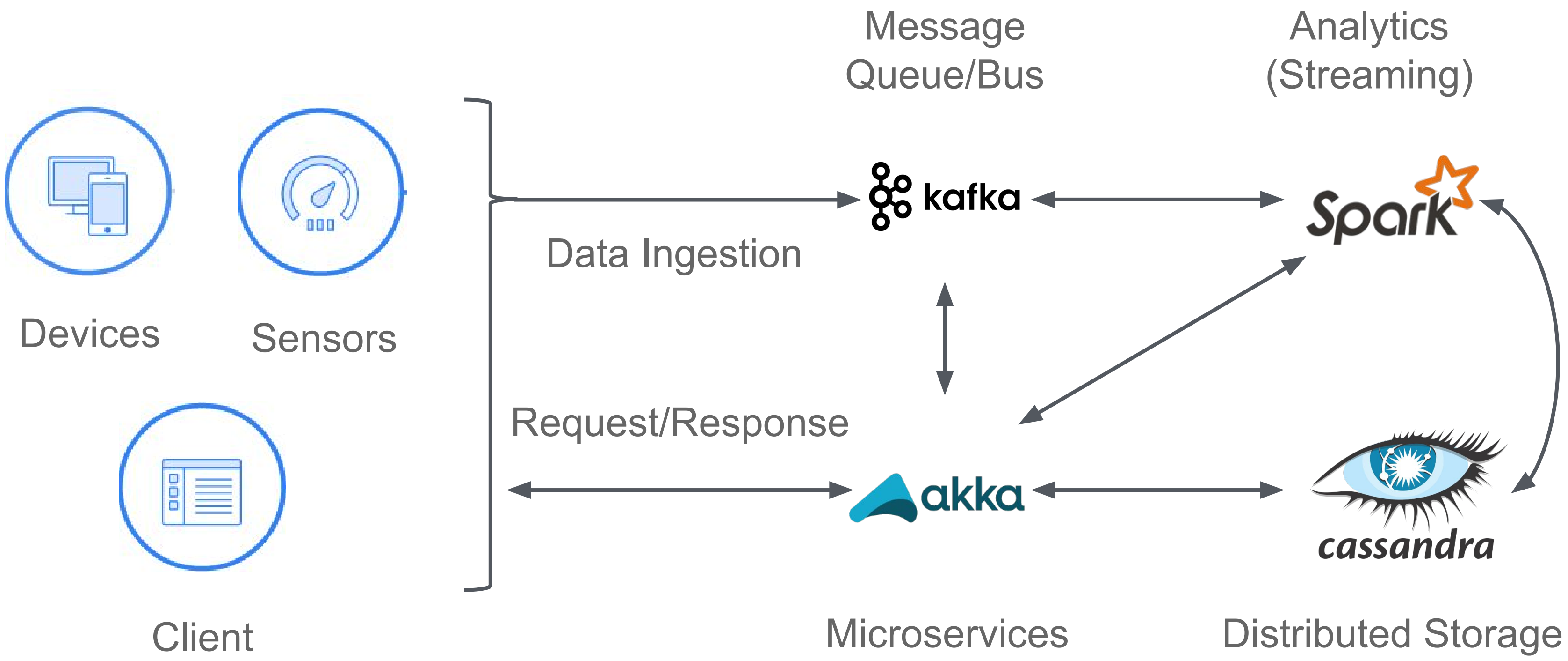
APACHE CASSANDRA

Typical Use: No-dependency, time series database

Why Cassandra?

- A top level Apache project born at Facebook and built on Amazon's Dynamo and Google's BigTable
- Offers continuous availability, linear scale performance, operational simplicity and easy data distribution

A GOOD STACK ...



Use Cases:

- Anomaly detection
- Personalization
- IoT Applications
- Predictive Analytics
- Machine Learning

how do we operate these distributed systems?

most organizations have many stateless independent (micro)services, the *distributed systems* I’m talking about here are …


state
latency
consensus
synchronous
high
tolerance
tor
idempotence
consistency
dynamic
rate
fault
unreliable
cap
clients
network
backpressure
limiting
replication
eventual
switch
topology
partial
nondeterministic
asynchronous
partition
failure
theorem
paxos
performance
scalability
administrators
availability
insecure
bandwidth
power
stateful
multiple

how do we *scale the operations* of distributed systems?

SMACK STACK

Apache Spark: distributed, large-scale data processing

Apache Mesos: cluster resource manager

Akka: toolkit for message driven applications

Apache Cassandra: distributed, highly-available database

Apache Kafka: distributed, highly-available messaging system

distributed systems
are *hard* to operate

DATA & ANALYTICS DAY 2 OPS CHALLENGES

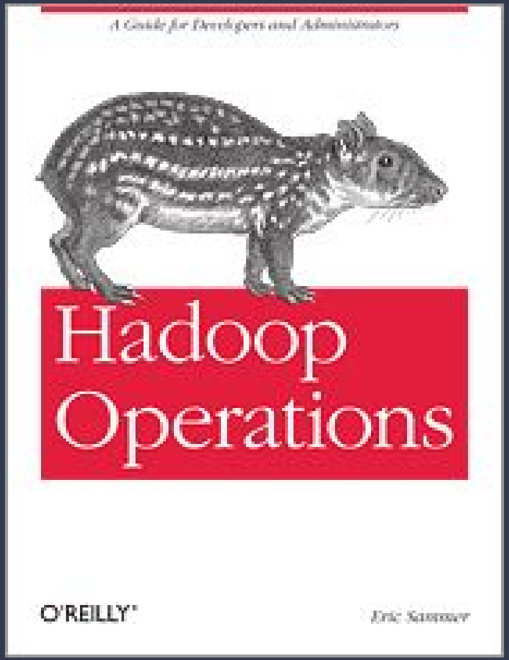
- Bare metal storage (or someone else’s problem)
- Drive down job latency and drive up utilization
- Run multiple versions simultaneously
- Upgrade complicated data systems

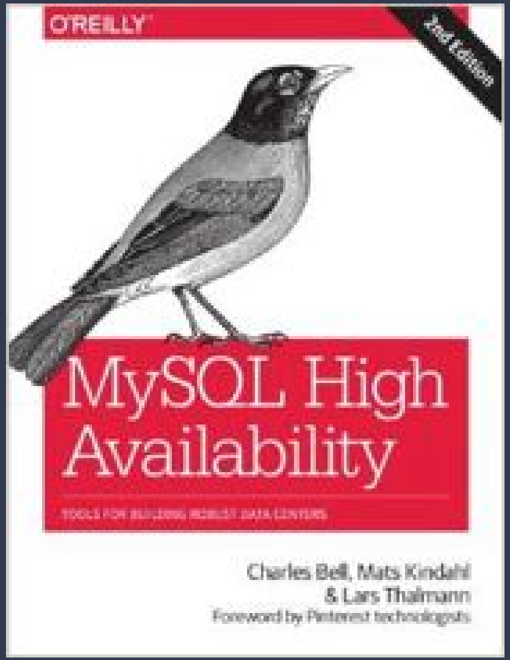
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

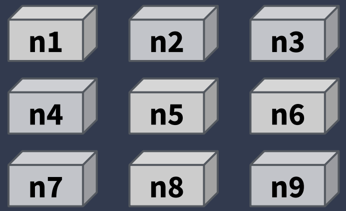
	fault tolerance
+	high availability
+	latency
+	bandwidth
+	CPU/mem resources
+	...
=	configuration

1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

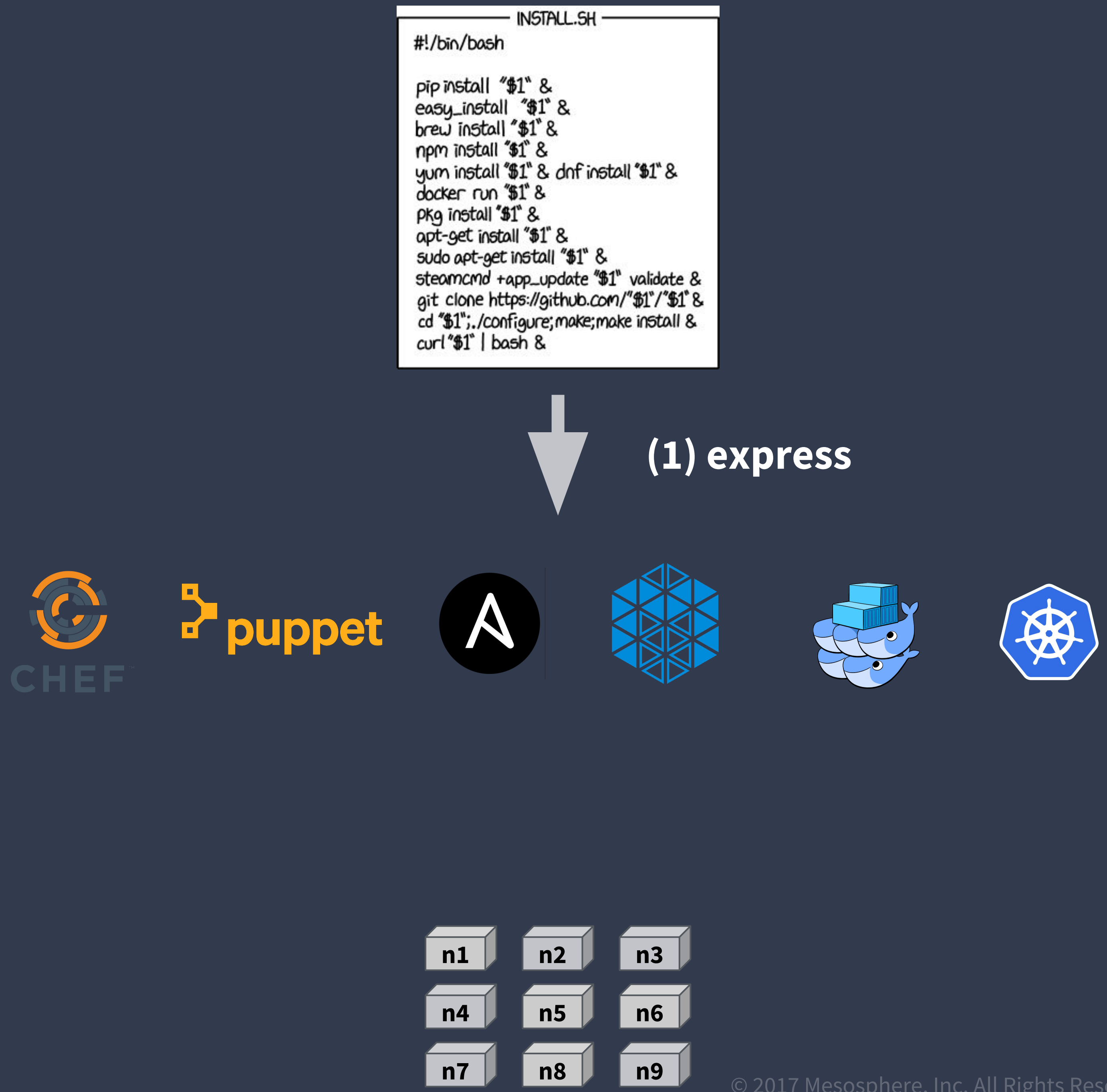




1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1



1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1



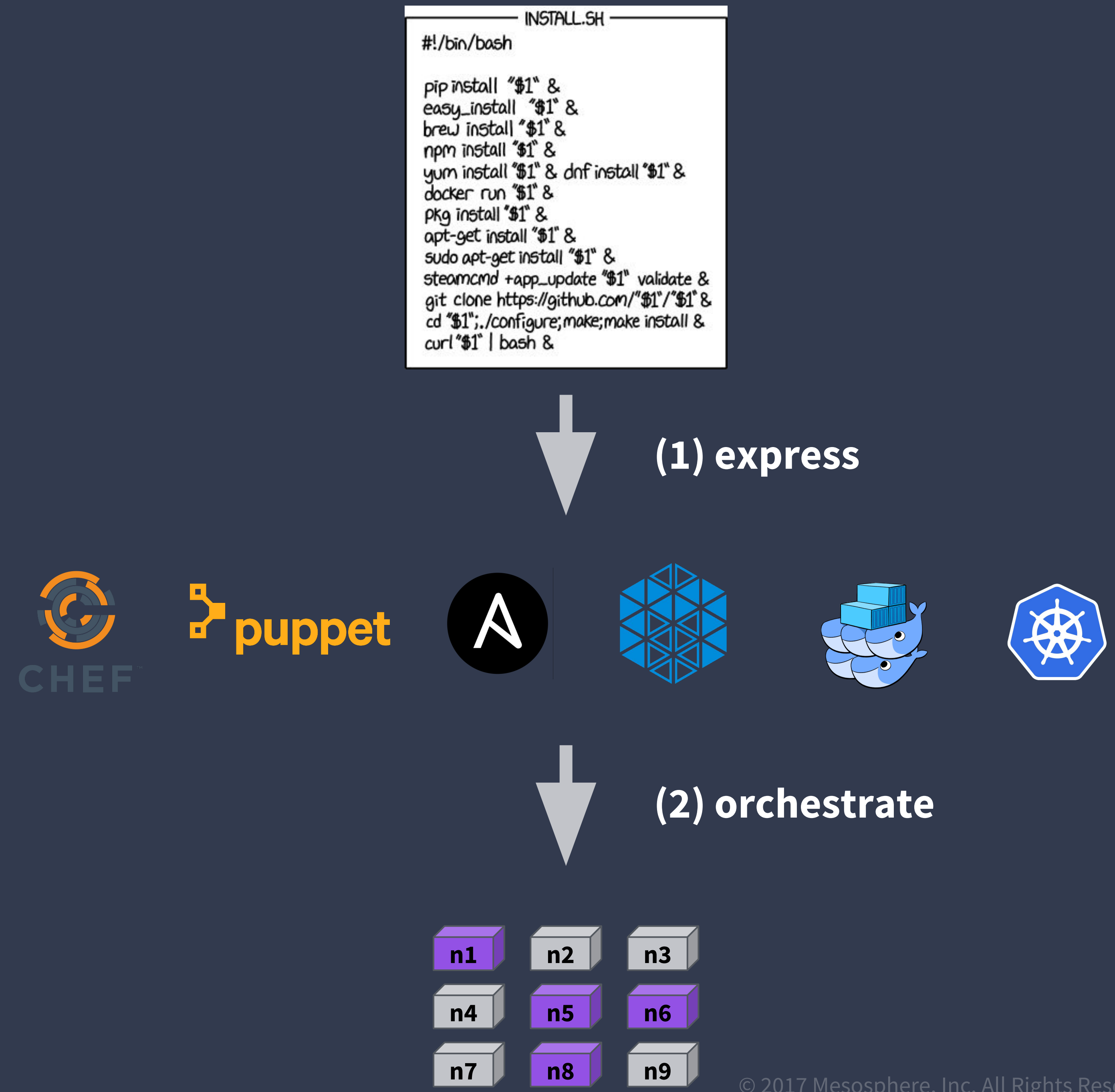
1: download

2: deploy

3: monitor

4: maintain

5: upgrade → goto 1



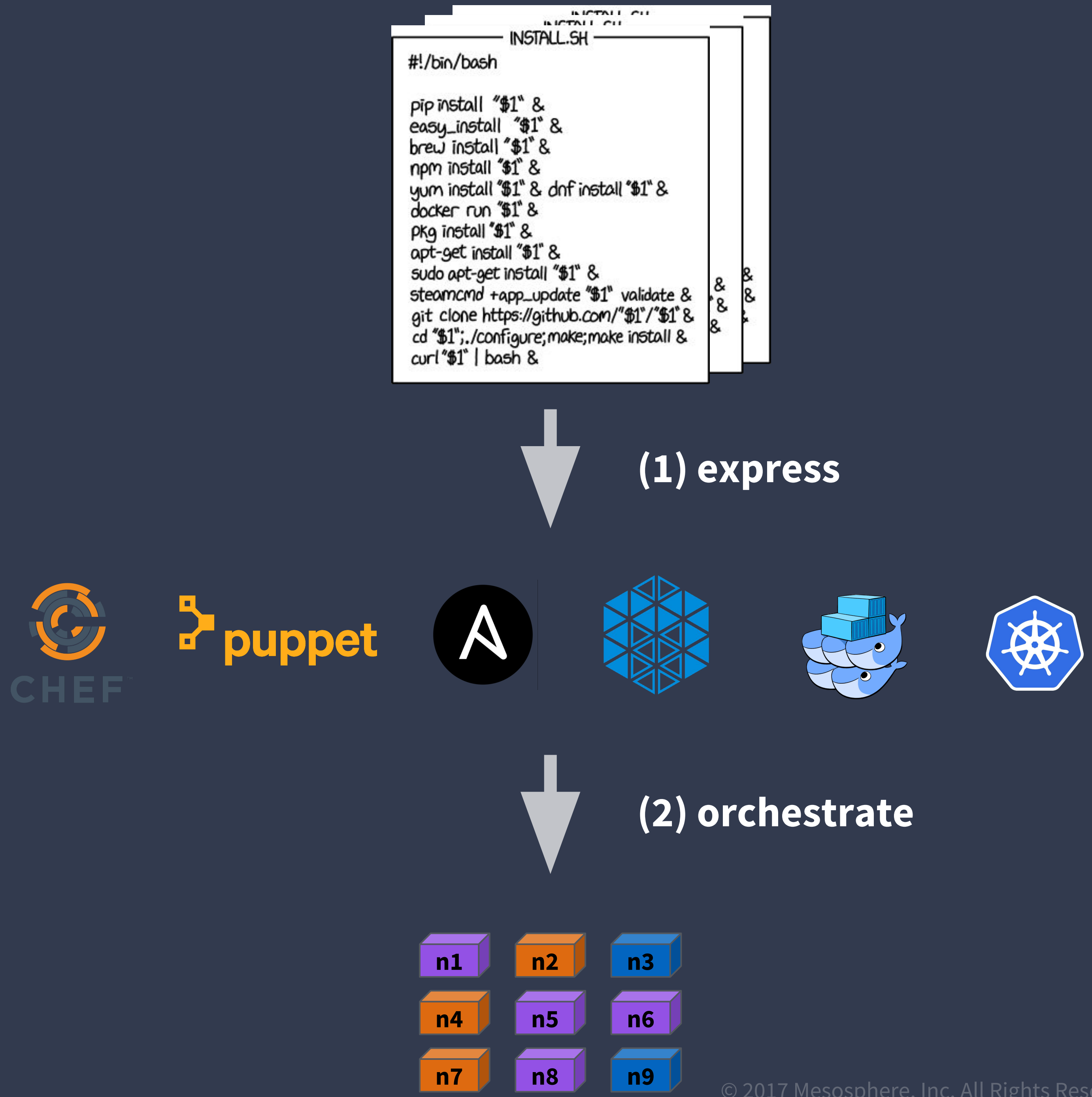
1: download

2: deploy

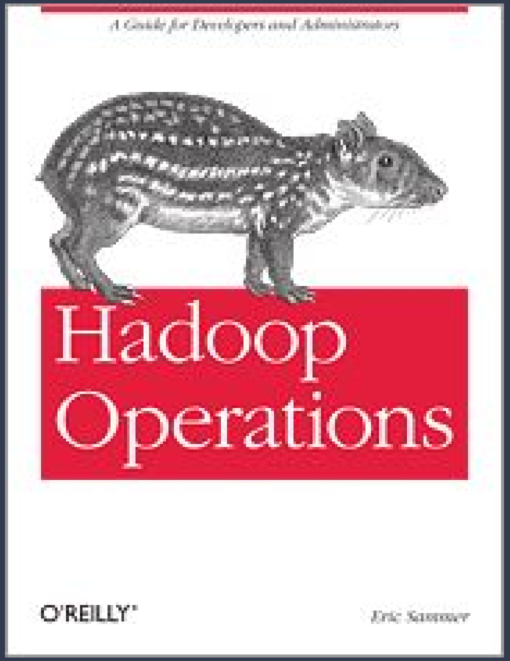
3: monitor

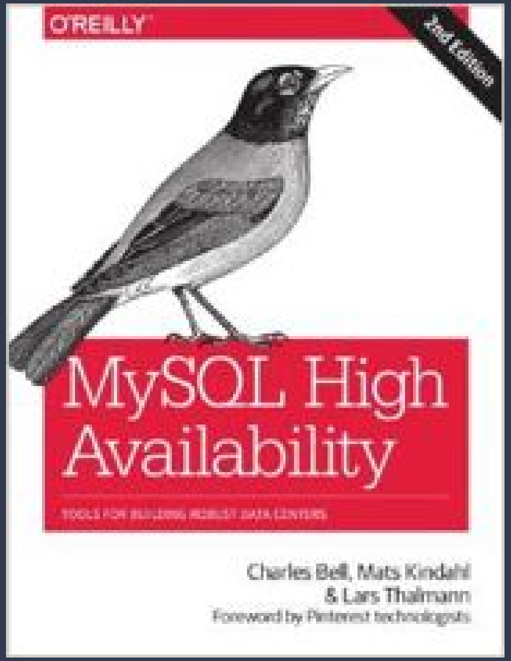
4: maintain

5: upgrade → goto 1

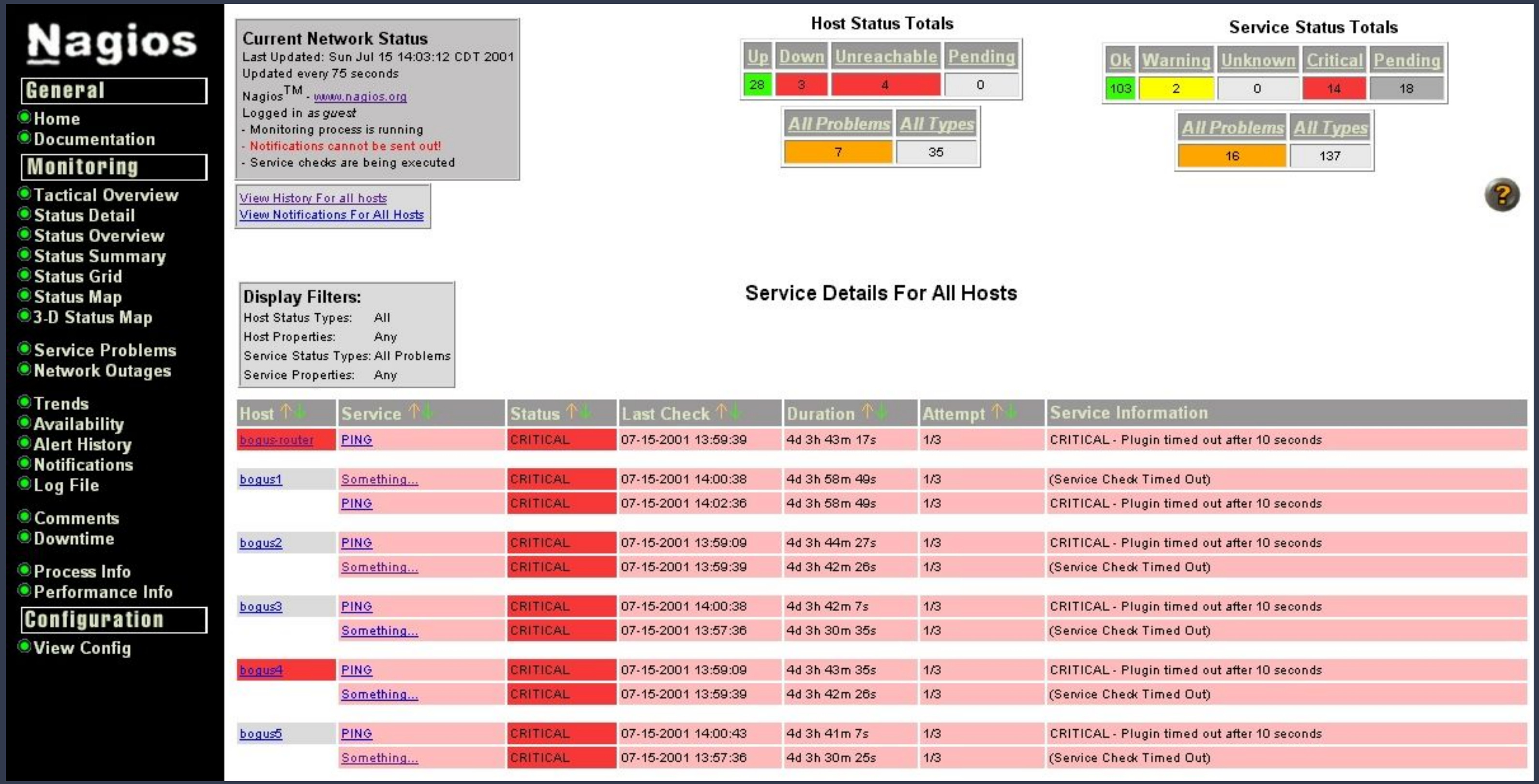


1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1



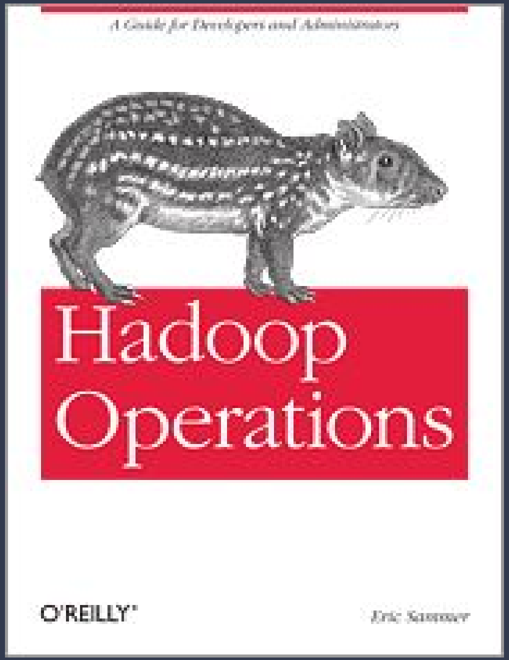


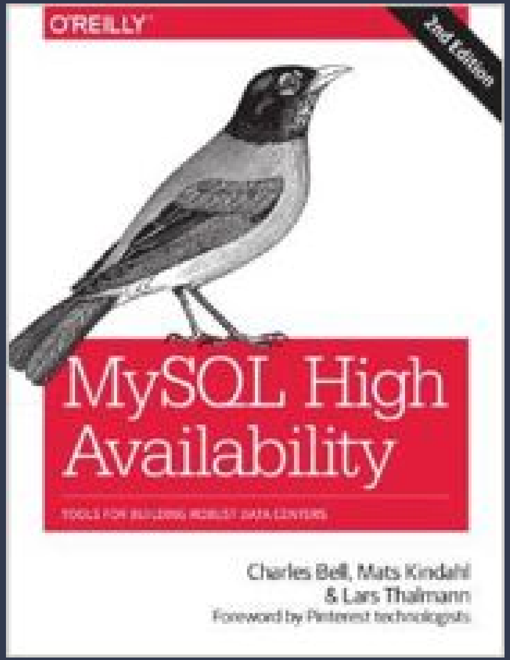
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1



1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

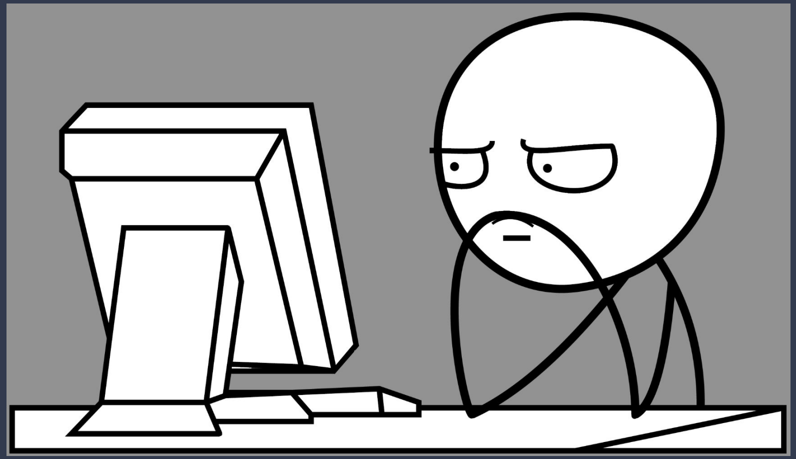
first, debug ...





1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

first, debug ...



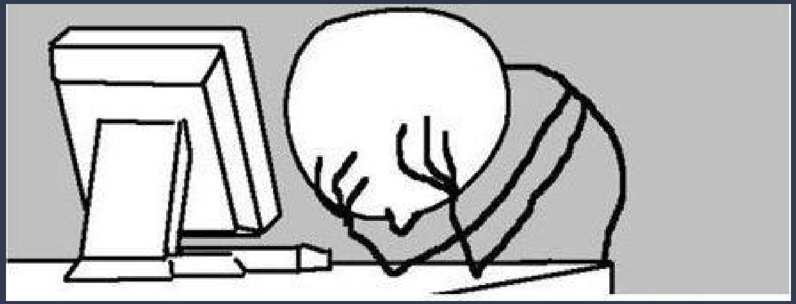
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

second, fix (scale, patch, etc)

...

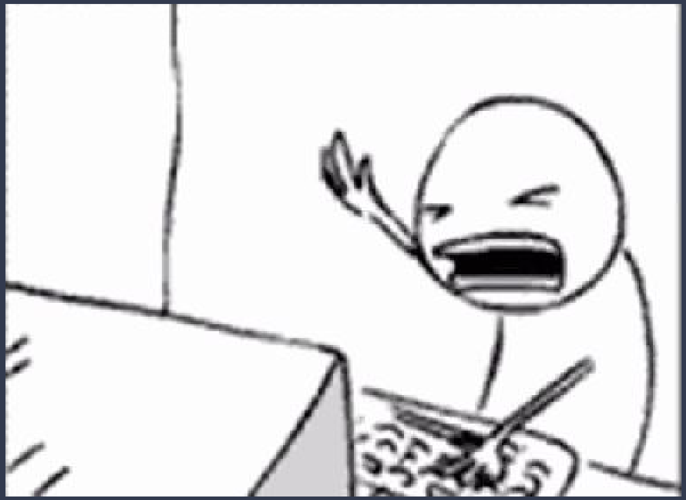
1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

then, debug again ...

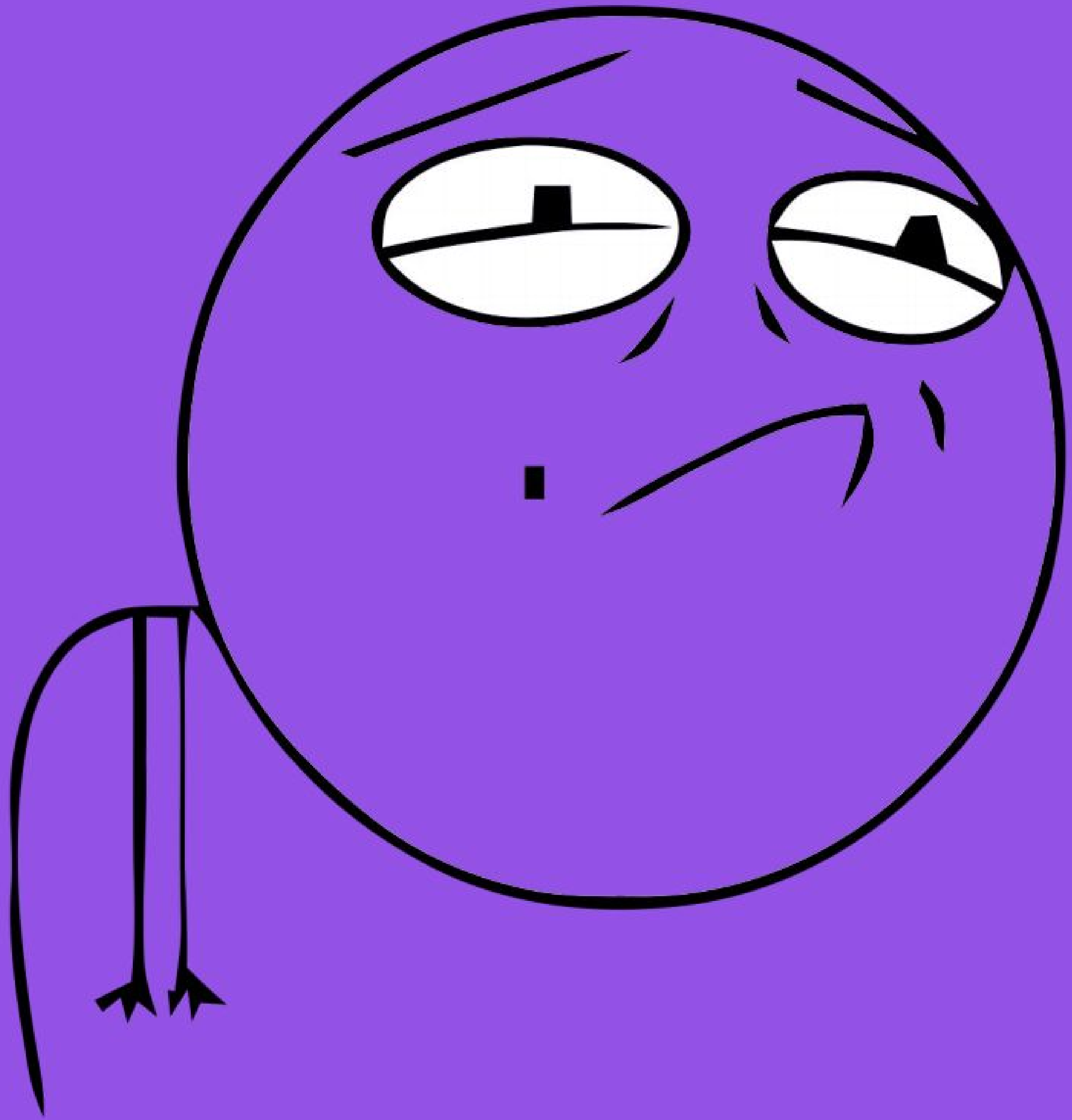


1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1

finally, write **code** so it never happens again ...



1: download
2: deploy
3: monitor
4: maintain
5: upgrade → goto 1



thesis:

distributed systems should (be able to) operate themselves; deploy, monitor, upgrade …

why:

(1) operators have *inadequate knowledge* of distributed system needs/semantics to make optimal decisions

why:

(1) operators have *inadequate knowledge* of distributed system needs/semantics
to make optimal decisions
(even after reading the book)

why:

(2) execution needs/semantics *can't easily or efficiently be expressed* to underlying system, and vice versa

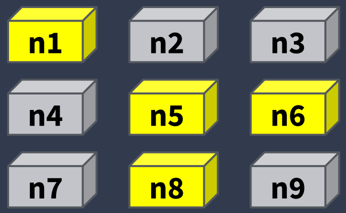
hadoop Map Reduce

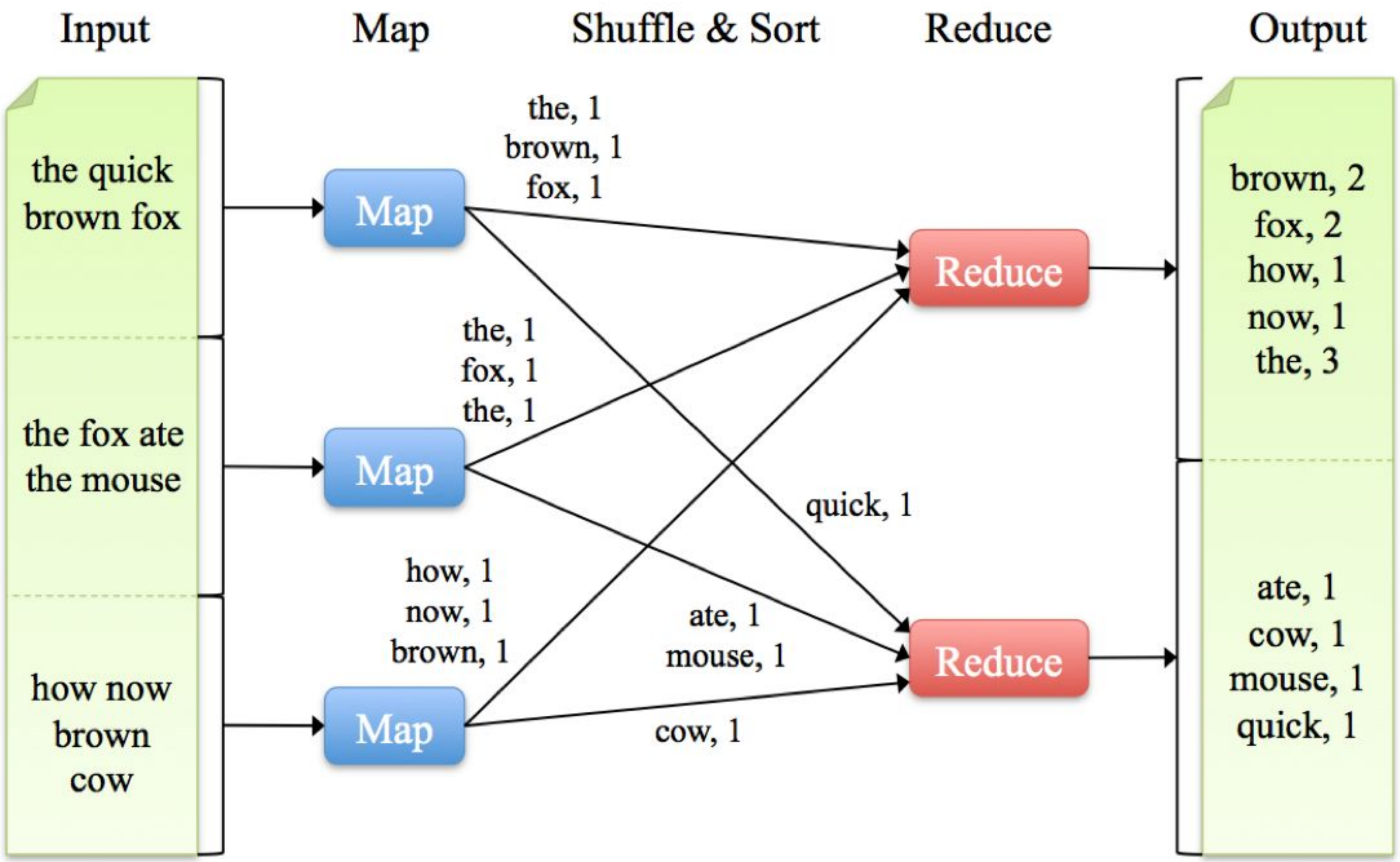
(1) express


INSTALL.SH

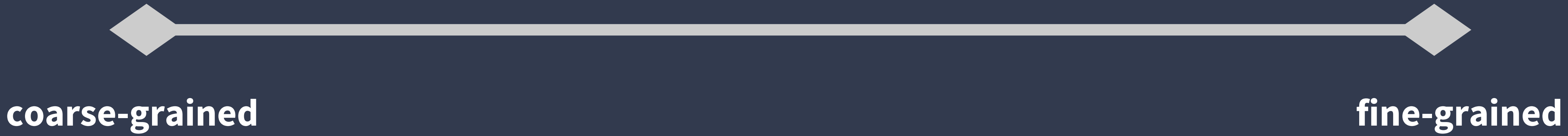
```
#!/bin/bash

pip install "$1" &
easy_install "$1" &
brew install "$1" &
npm install "$1" &
yum install "$1" & dnf install "$1" &
docker run "$1" &
pkg install "$1" &
apt-get install "$1" &
sudo apt-get install "$1" &
steamcmd +app_update "$1" validate &
git clone https://github.com/"$1"/"$1" &
cd "$1";./configure;make;make install &
curl "$1" | bash &
```

(2) orchestrate




Input
Map
Shuffle & Sort
Reduce
Output
the quick brown fox
the fox ate the mouse
how now brown cow
Map
Map
Map
the, 1
brown, 1
fox, 1
the, 1
fox, 1
the, 1
how, 1
now, 1
brown, 1
quick, 1
ate, 1
mouse, 1
cow, 1
Reduce
Reduce
brown, 2
fox, 2
how, 1
now, 1
the, 3
ate, 1
cow, 1
mouse, 1
quick, 1

configuration spectrum:

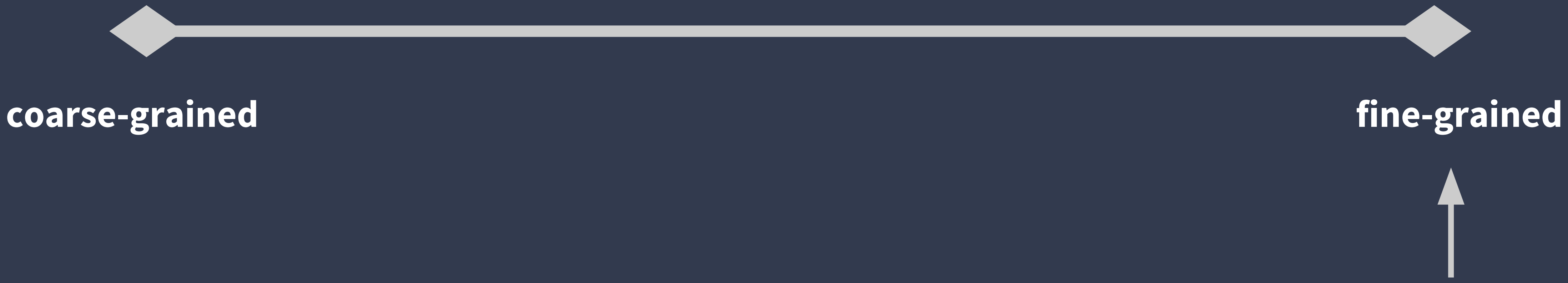


configuration spectrum:

coarse-grained **fine-grained**

easiest to express (how most of us would do it),
but worst resource utilization

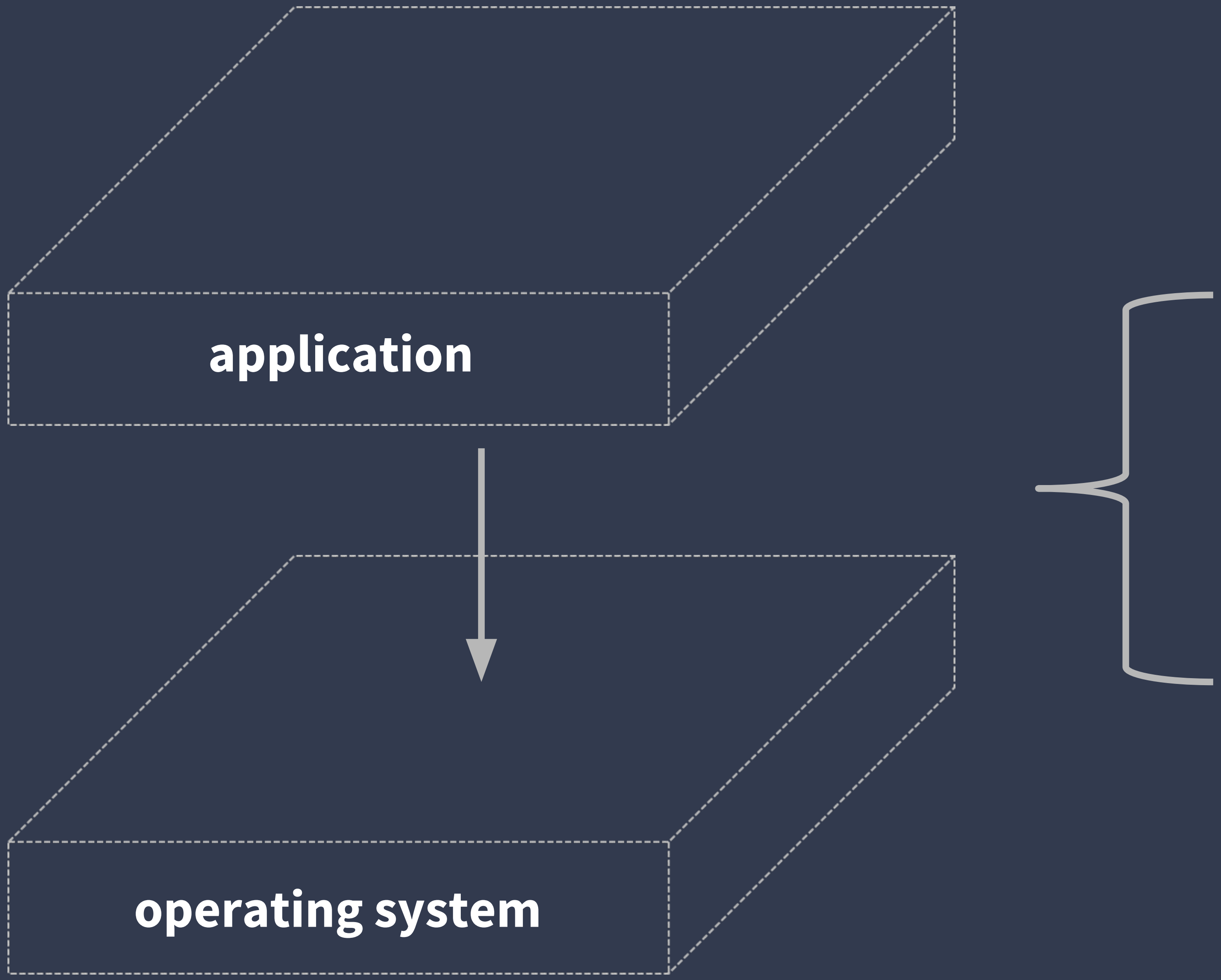
configuration spectrum:



**hardest to express (if even possible),
but best resource utilization**

why can’t Hadoop decide this for me?

applications "operate" themselves on Linux; when an application needs to "scale up" it asks the operating system to allocate more memory or create another thread ...

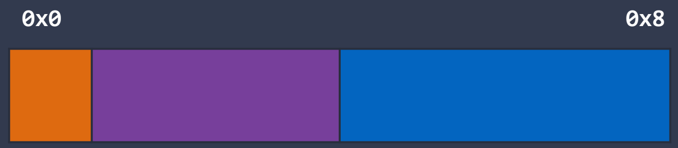


<u>syscall interface:</u>
memory allocate
clone/fork
create file
read, write
…

once upon a time ... before virtual memory

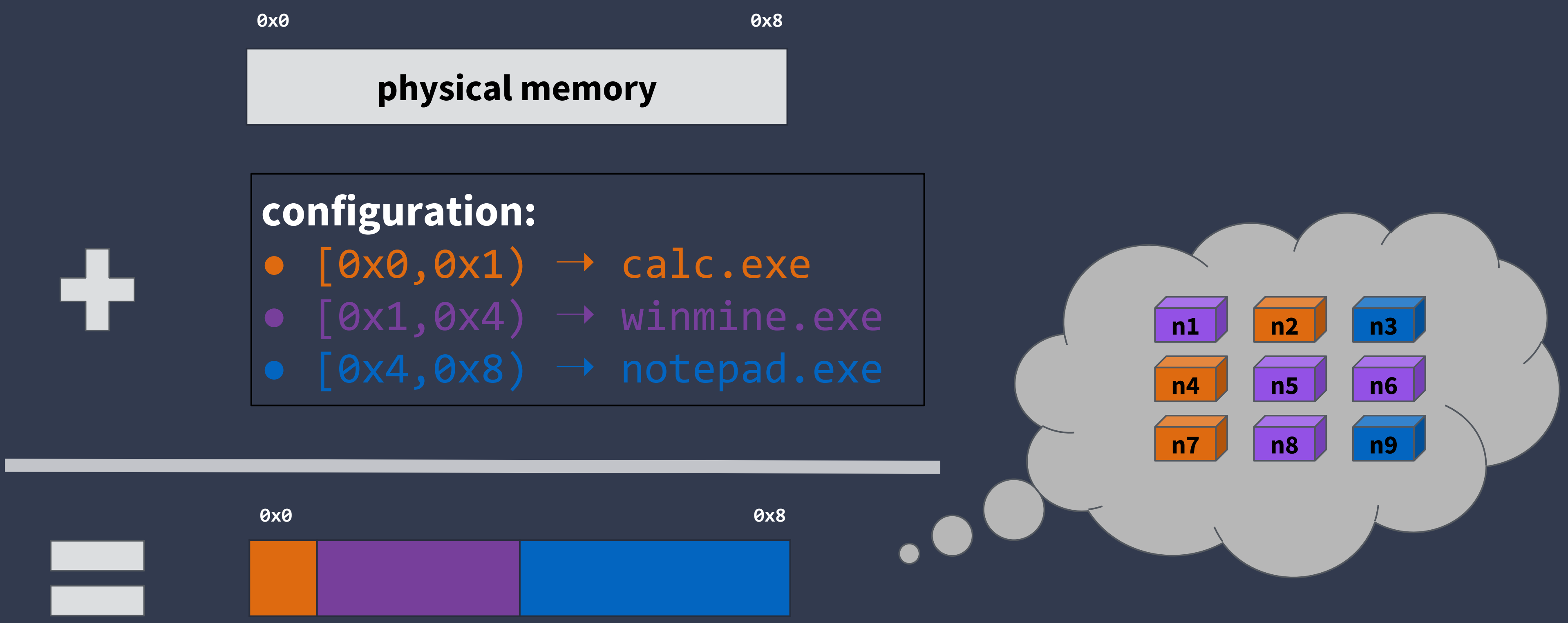
configuration:

- [0x0,0x1) → calc.exe
- [0x1,0x4) → winmine.exe
- [0x4,0x8) → notepad.exe



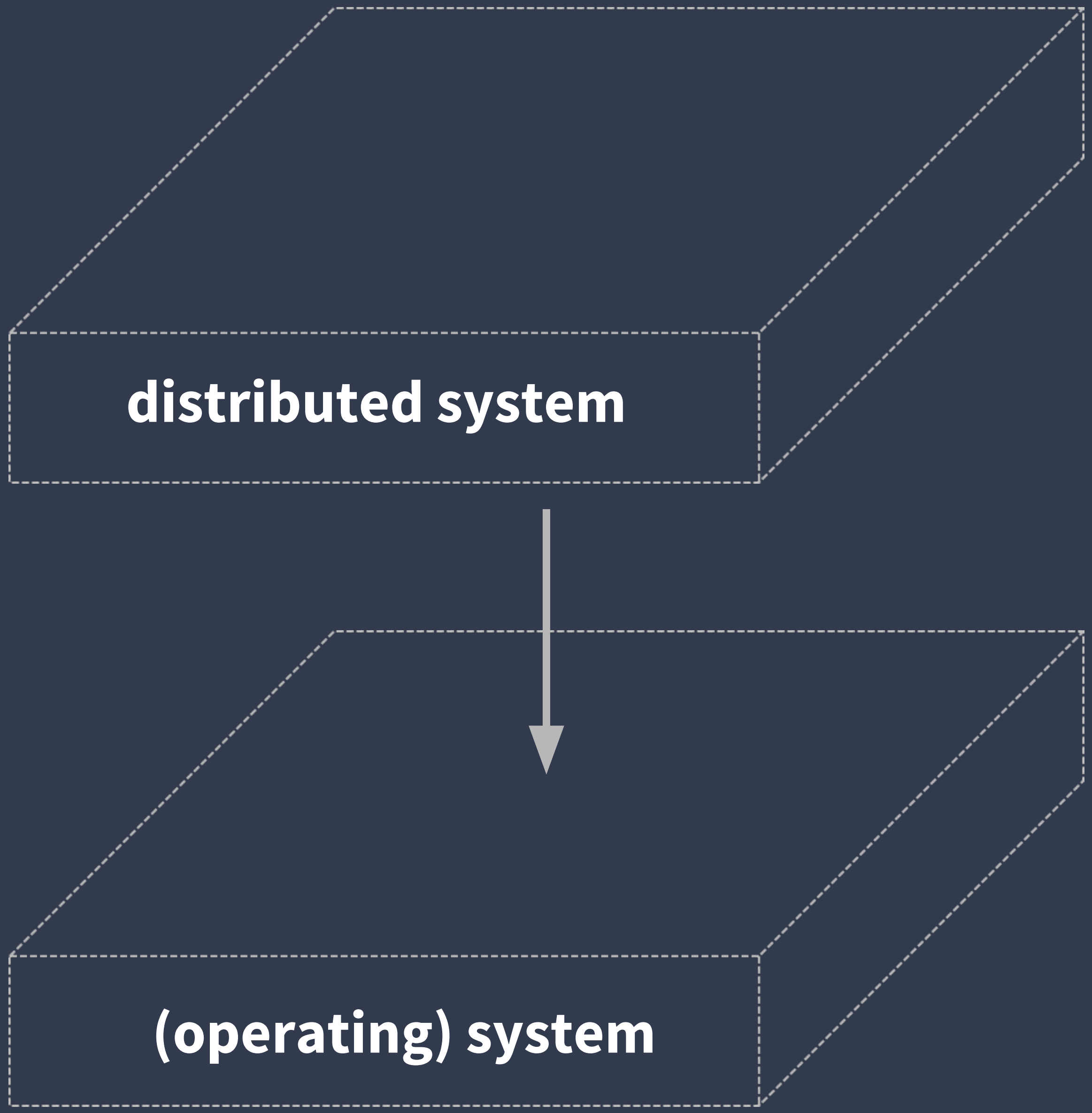
applications take physical memory address range as an **input**

once upon a time … before virtual memory



how:

distributed systems need *interface* to *communicate* with underlying system, *and vice versa*

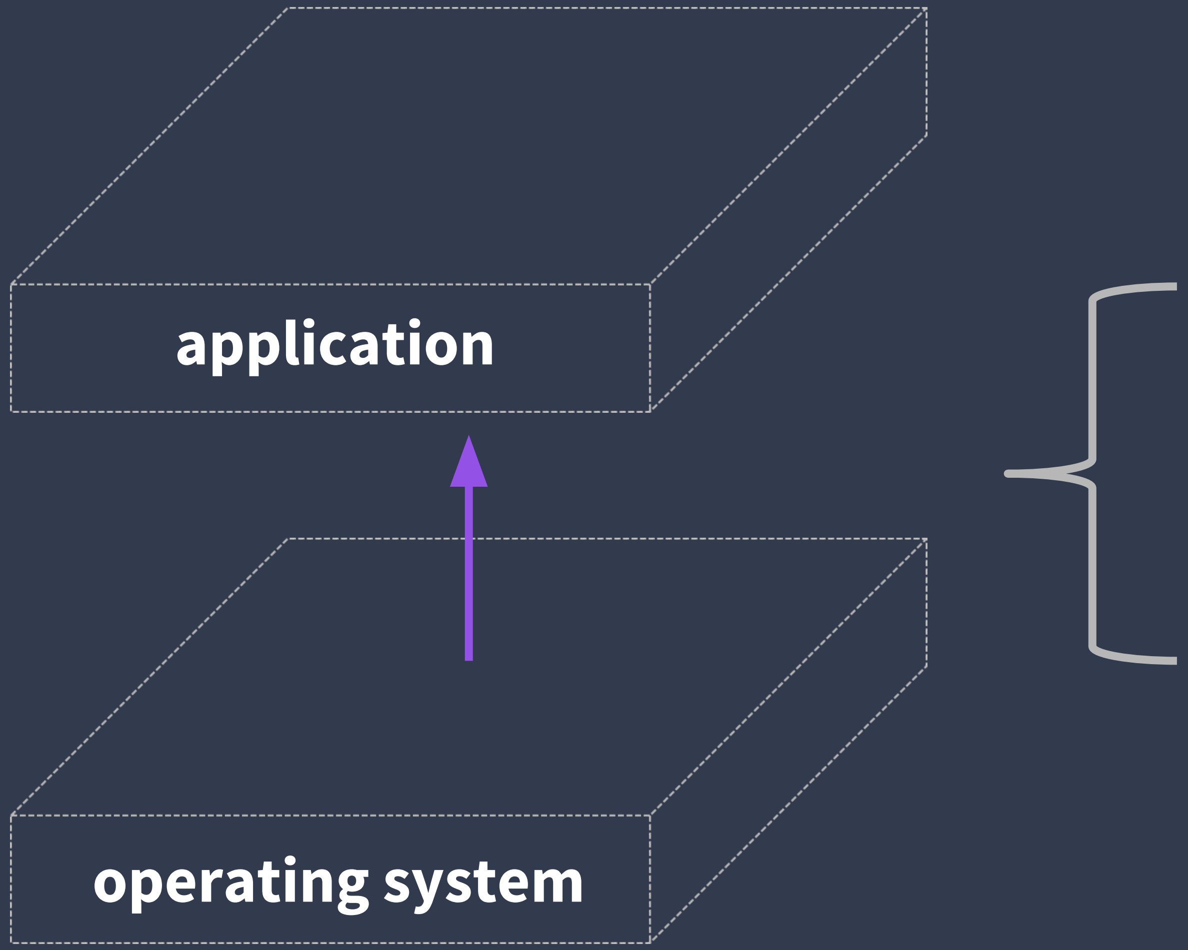


interface:
resource allocation
launch container/VM
create storage
attach/detach storage
...

vice versa:

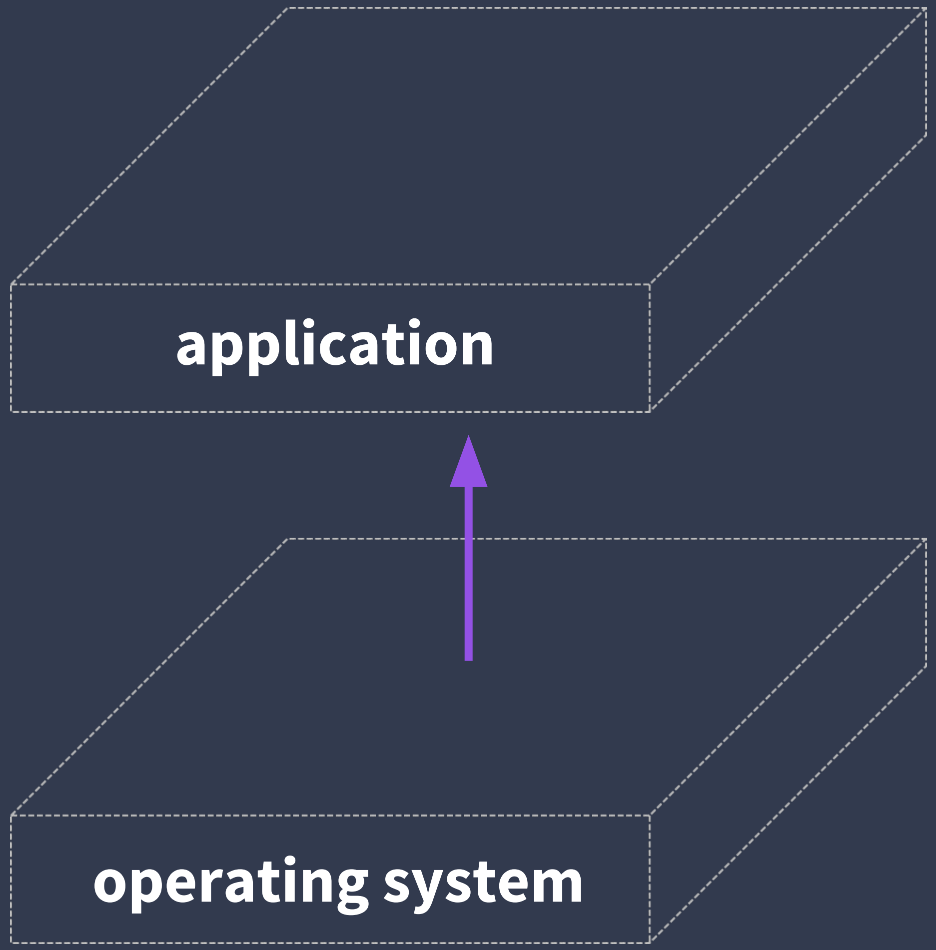
operating system should be able to *callback* into application

learning from history … bidirectional interface



callback interface:
paging/swapping
CPU deallocation
…

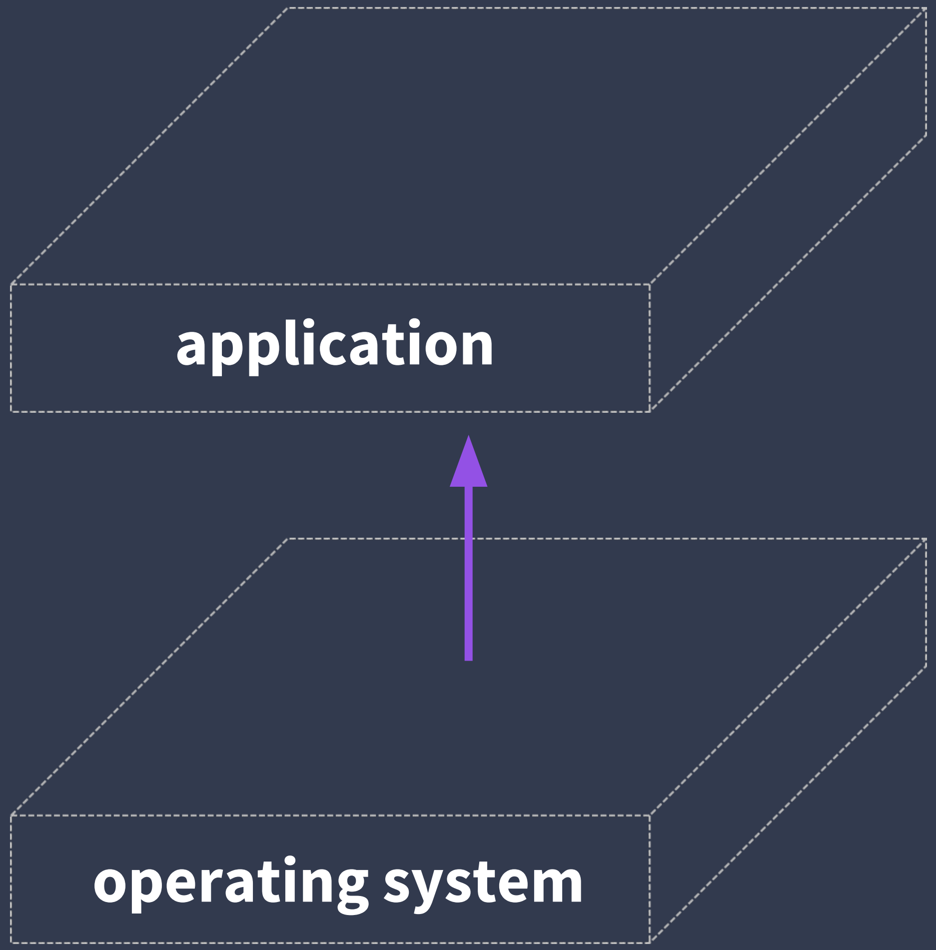
learning from history … bidirectional interface



callback interface:
paging/swapping
CPU deallocation
…

better than LRU, ask the application what pages to swap!

learning from history … bidirectional interface



callback interface:

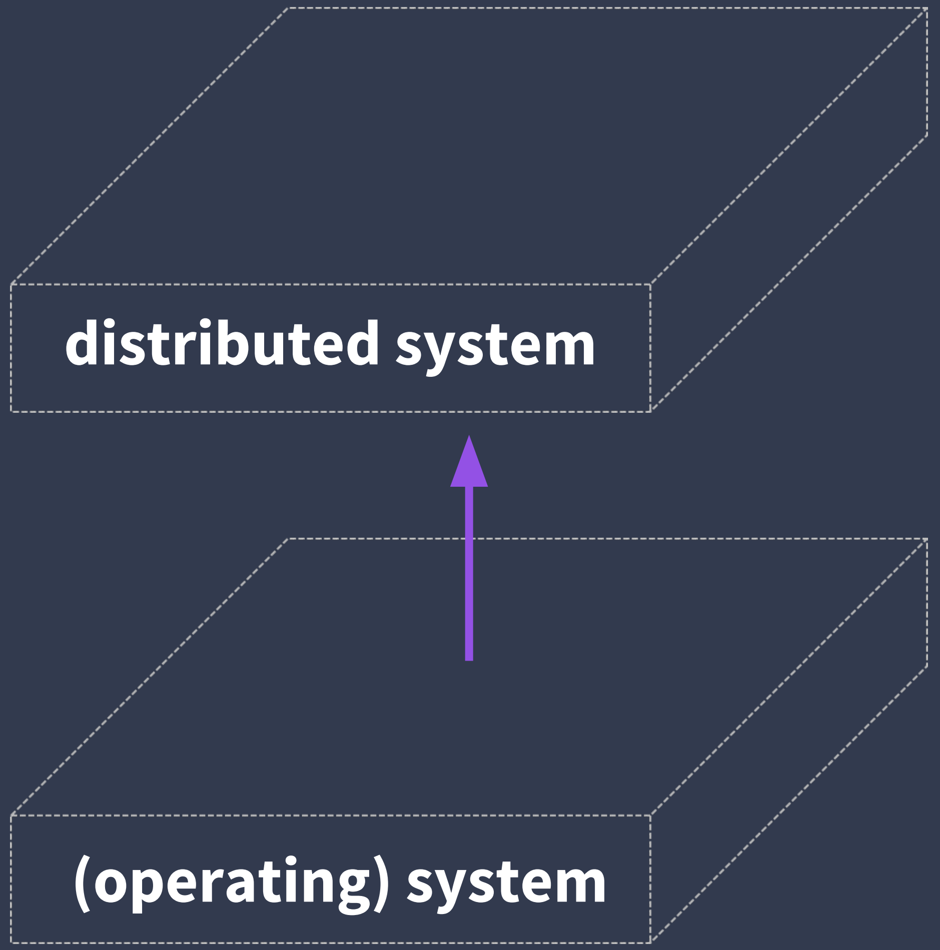
paging/swapping

CPU deallocation

…

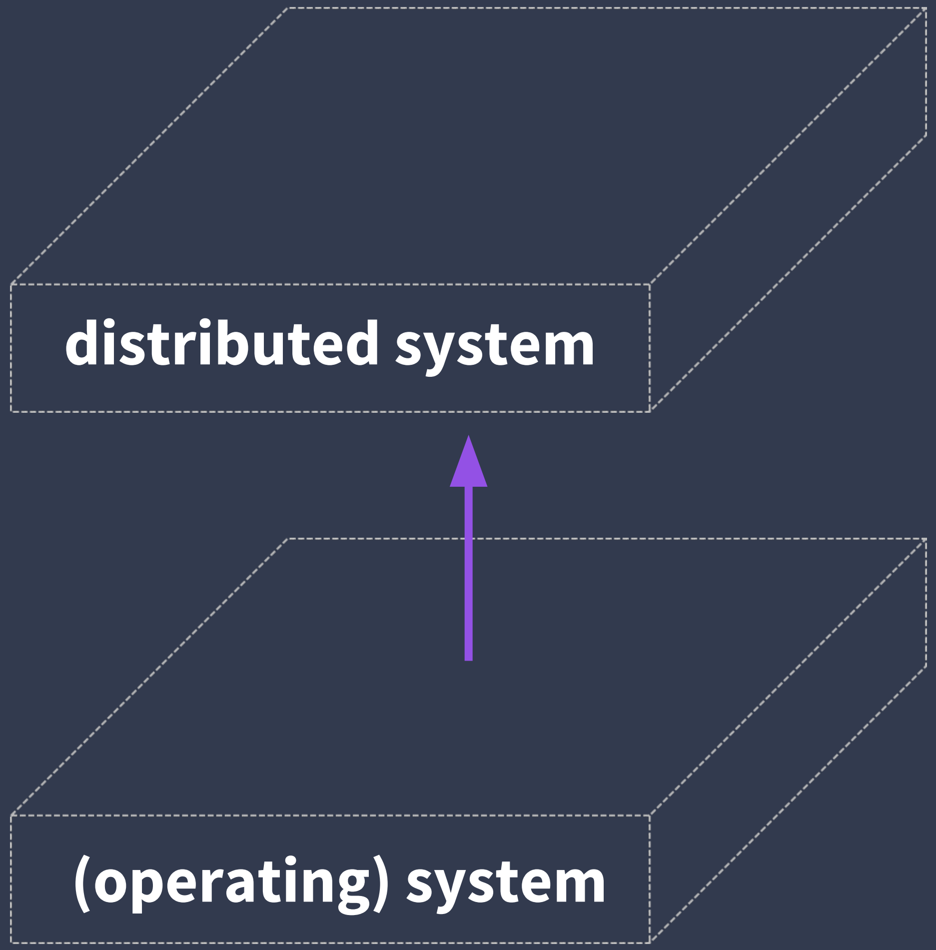
search for ‘scheduler activations’ and ‘Lithe composition’

distributed systems need bidirectional interface too



callback interface:
container/VM failed
resource deallocation
…

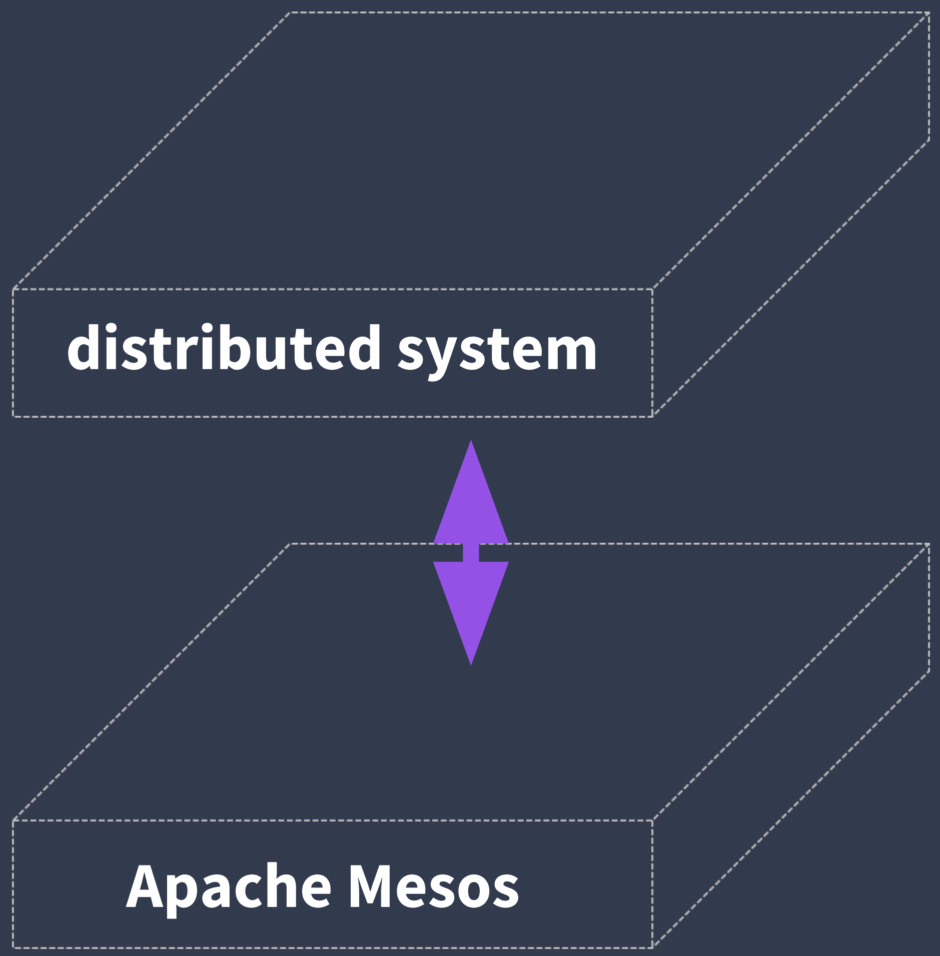
distributed systems need bidirectional interface too



<u>callback interface:</u>
container/VM failed
resource deallocation
…

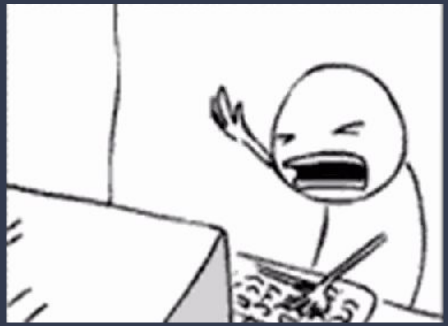
tell the distributed system about "planned failures" (i.e., maintenance)

Apache Mesos



Dogfooding: Apache Spark

reality is people are (already) building software that *operates* distributed systems ...

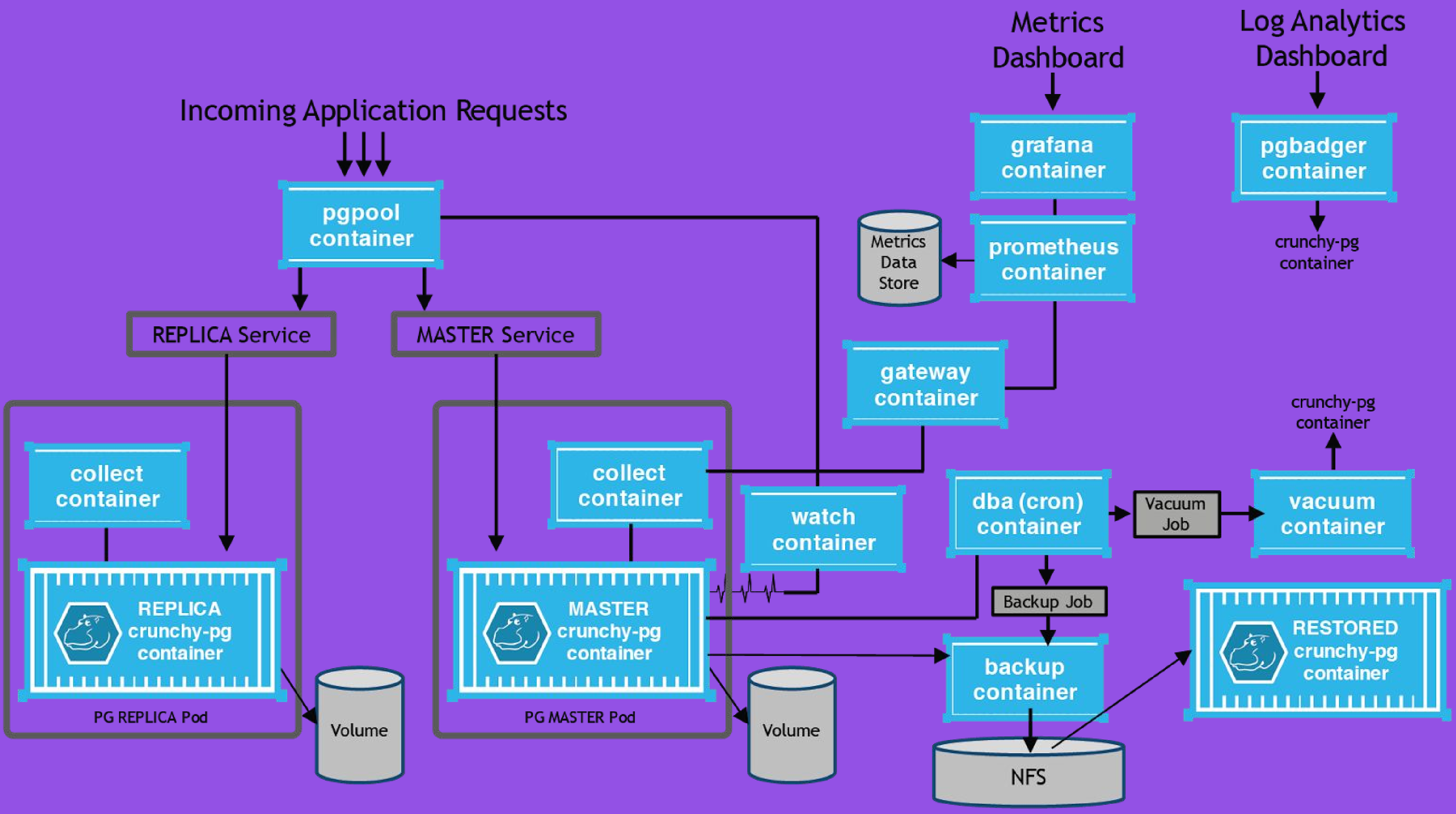


common pattern: ad hoc control planes

goal: provide *distributed system** as software as a service (SaaS) to the rest of your internal organization or to sell to external organizations

solution: a *control plane* built out of ad hoc scripts, ancillary services, etc, that deploy, maintain, and upgrade said SaaS

* e.g., analytics via Spark, message queue via Kafka, key/value store via Cassandra



https://github.com/crunchydata/crunchy-containers

```
$ kubectl create -f $LOC/kitchensink-master-service.json
$ kubectl create -f $LOC/kitchensink-slave-service.json
$ kubectl create -f $LOC/kitchensink-pgpool-service.json
$ envsubst < $LOC/kitchensink-sync-slave-pv.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-master-pv.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-sync-slave-pvc.json
$ kubectl create -f $LOC/kitchensink-master-pvc.json
$ envsubst < $LOC/kitchensink-master-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-slave-dc.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-sync-slave-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-pgpool-rc.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-watch-sa.json
$ envsubst < $LOC/kitchensink-watch-pod.json | kubectl create -f -
```

https://schd.ws/hosted_files/cnd2016/8d/Containerizing%20PostgreSQL%20and%20Making%20it%20Cloud%20Native%20Ready%20-%20Jeff%20McCormick.pdf

```
$ kubectl create -f $LOC/kitchensink-master-service.json
$ kubectl create -f $LOC/kitchensink-slave-service.json
$ kubectl create -f $LOC/kitchensink-pgpool-service.json
$ envsubst < $LOC/kitchensink-sync-slave-pv.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-master-pv.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-sync-slave-pvc.json
$ kubectl create -f $LOC/kitchensink-master-pvc.json
$ envsubst < $LOC/kitchensink-master-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-slave-dc.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-sync-slave-pod.json | kubectl create -f -
$ envsubst < $LOC/kitchensink-pgpool-rc.json | kubectl create -f -
$ kubectl create -f $LOC/kitchensink-watch-sa.json
$ envsubst < $LOC/kitchensink-watch-pod.json | kubectl create -f -
```

https://schd.ws/hosted_files/cnd2016/8d/Containerizing%20PostgreSQL%20and%20Making%20it%20Cloud%20Native%20Ready%20-%20Jeff%20McCormick.pdf

what happens if there's a bug in the control plane?

what if my control plane has diverged from yours?

what happens when a new release of the distributed system invalidates an assumption the control plane previously made?

a better world …

control planes should be built into the distributed systems itself by the experts who built the distributed system in the first place!

as an industry we should strive to build a standard interface that distributed systems can leverage

vice versa:

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes …

a better world ...

control planes should be built into distributed systems themselves by the experts who built the distributed system in the first place!

as an industry we should strive to build a standard interface distributed systems can leverage

our standard interface should be bidirectional to avoid sub-optimal outcomes

how do we scale the operations of distributed systems?

let them *scale* themselves!

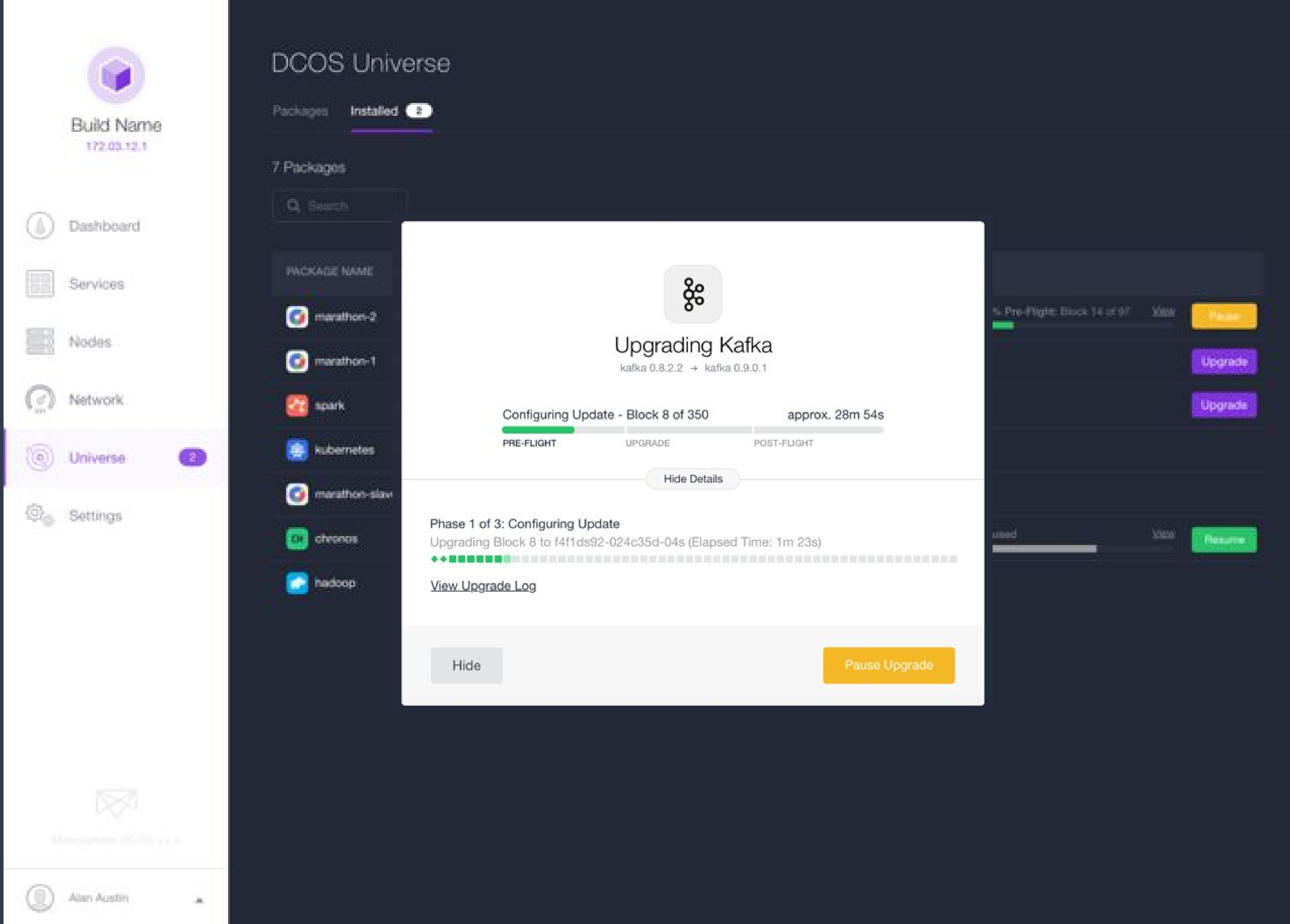
OPERATING SYSTEMS ARE FOR APPLICATIONS

“SaaS” Experience using DC/OS

Spark	DataStax Enterprise	GitLab	Elasticsearch	HDFS
Jenkins	Confluent Kafka	Cassandra	MariaDB	Zeppelin
Riak	ArangoDB	JFrog Artifactory	Storm	MemSQL

DC/OS

DC/OS SERVICE MANAGES IT'S OWN UPGRADES



DC/OS: AVOIDING CLOUD LOCK-IN #2

	CAPABILITY	AWS	AZURE	GCP	DC/OS
Storage	**Object Storage**	S3	Blob Storage	Cloud Storage	Quobyte
	Block Storage	Elastic Block Storage (EBS)	Page Blobs, Premium Storage	GCE Persistent Disks	EMC² ScaleIO
	File Storage	Elastic File System	File Storage	ZFS / Avere	EMC² ScaleIO
DB	**Relational**	RDS	SQL Database	Cloud SQL (MySQL)	MariaDB, CRATE.IO, MEMSQL, MySQL
	NoSQL	DynamoDB	DocumentDB	Datastore, Bigtable	cassandra, ArangoDB, riak
Data & Analytics	**Full Text Search**	CloudSearch	Log Analytics, Search	N/A	elastic
	Hadoop / Analytics	Elastic Map Reduce (EMR)	HDInsight	Dataproc, Dataflow	hadoop HDFS, Spark
	Stream Processing / Ingest	Kinesis	Stream Analytics, Data Lake	Kinesis	kafka, Spark Streaming
	Data Warehouse	Redshift	SQL Data Warehouse	BigQuery	citusdata, Impala, APACHE DRILL
Other	**Monitoring**	CloudWatch	Application Insights, Portal	Stackdriver Monitoring	DATADOG, netsil, ruxit, sysdig
	Serverless	Lambda	Azure Functions	Google Cloud Functions	GALACTIC FOG

@dcos

chat.dcos.io

users@dcos.io

/groups/8295652

/dcos
/dcos/examples
/dcos/demos

THANK YOU!

DEMO!

QUESTIONS?

bigger picture:

abstractions exist for good reasons, but without sufficient communication they force sub-optimal outcomes …

