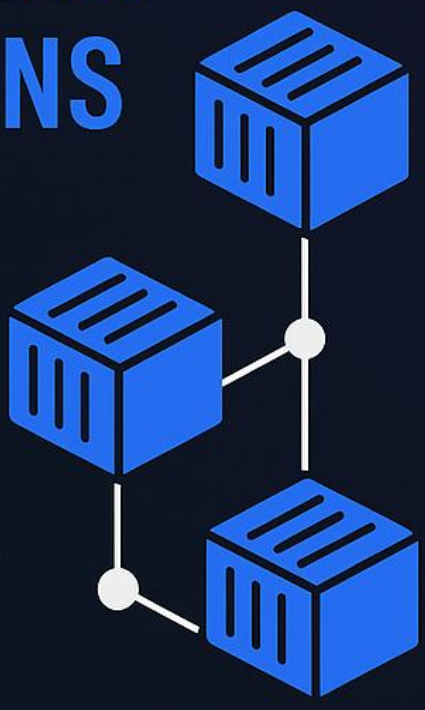
MASTERING KUBERNETES

MULTI-CONTAINER DESIGN PATTERNS

A Complete Guide
for Platform Engineers

Build Scalable,
Maintainable Container
Architectures

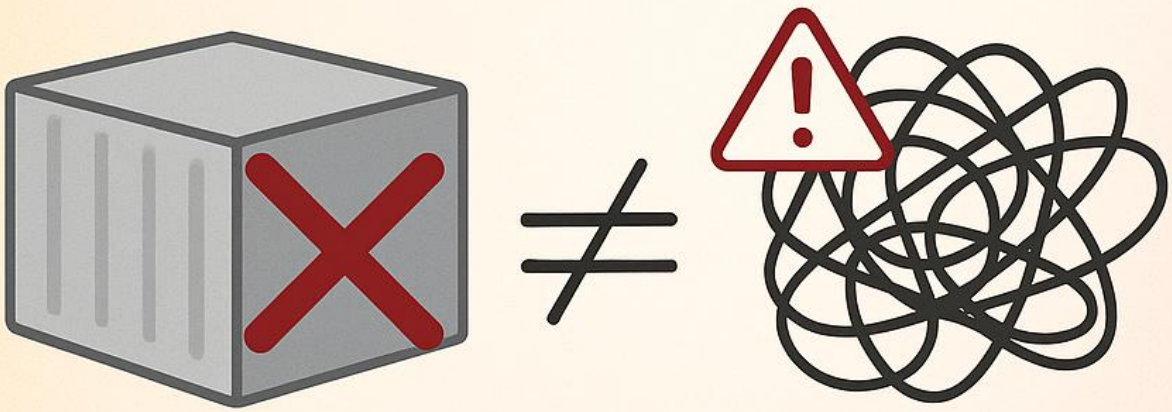


THE PROBLEM

🤔 MOST DEVELOPERS THINK:

Pod = One Container

But this limits Kubernetes' true power!



- ❌ Single container doing everything
- ❌ Tight coupling
- ❌ Hard to maintain
- ❌ Limited reusability

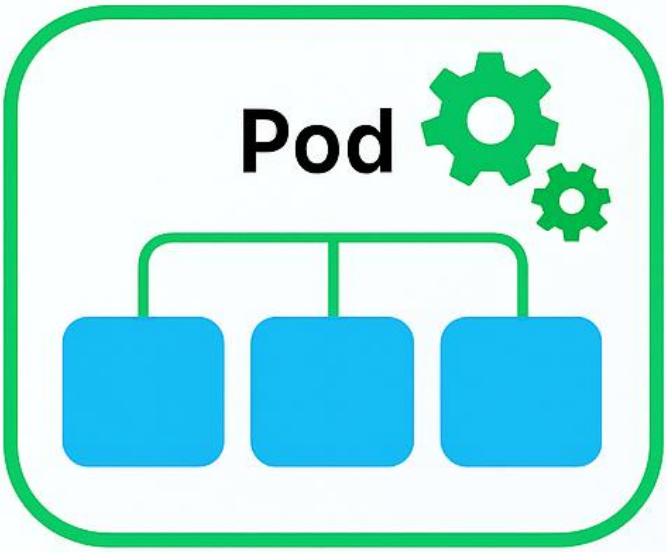
THE SOLUTION

✅ THE REALITY:

Pods are Shared Execution Environments

Multiple containers working together as a cohesive unit

- ◎ Separation of concerns
- ◎ Enhanced modularity
- ◎ Reusable components
- ◎ Unix philosophy: “Do one thing well”



WHAT IS A POD?

Shared Network

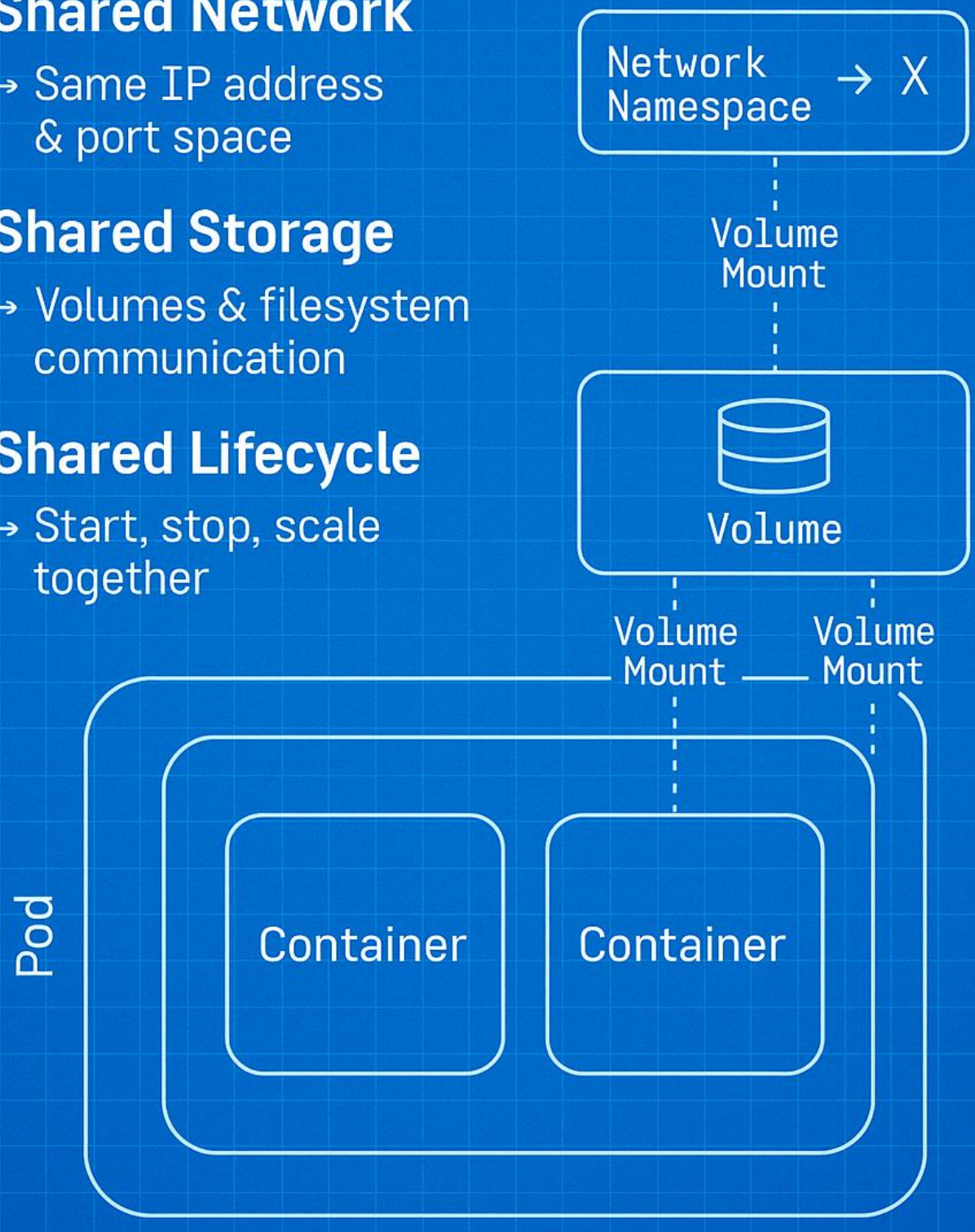
→ Same IP address & port space

Shared Storage

→ Volumes & filesystem communication

Shared Lifecycle

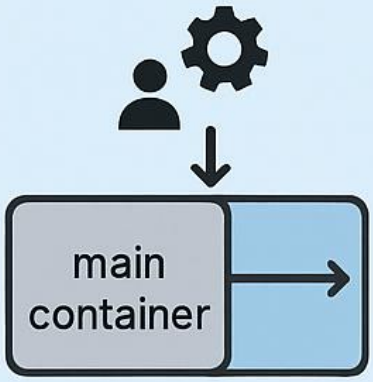
→ Start, stop, scale together



🎯 Pattern Overview

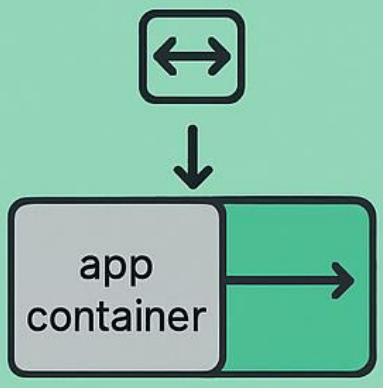
1️⃣ 3 ESSENTIAL MULTI-CONTAINER PATTERNS

SIDECAR PATTERN



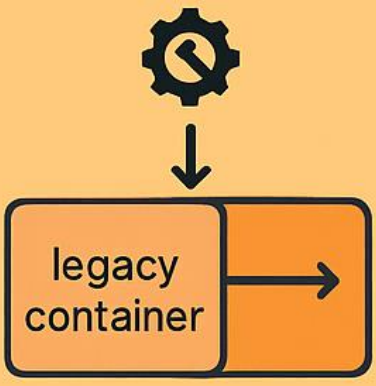
Helper container extends functionality

AMBASSADOR PATTERN



Proxy container handles connections

3 ADAPTER PATTERN



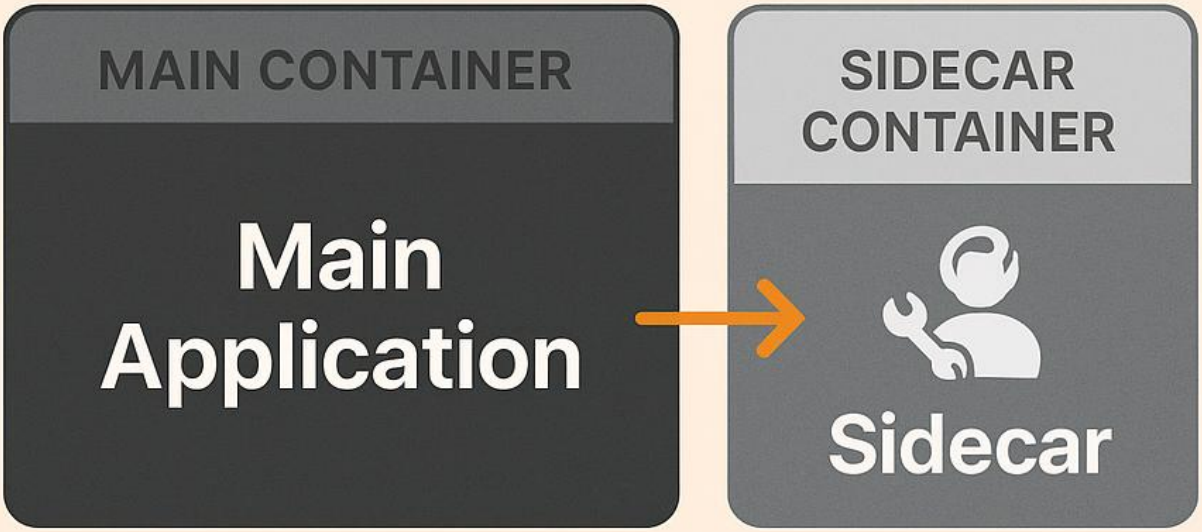
Transform container converts output

Each solves specific architectural challenges

SIDECAR PATTERN
Deep Dive

SIDECAR PATTERN: The Swiss Army Knife

Helper container alongside main application



COMMON USE CASES:

- Logging agents (Fluentd, Filebeat)
- Monitoring agents (Prometheus)
- Security proxies (Istio)
- Config management
- Secret rotation

💻 SIDECAR IN ACTION

```
volumes:
  name: shared-logs
  emptyDir: {}
```

Shared: Volume for communiciation

Both containers mount the same volume!

AMBASSADOR PATTERN

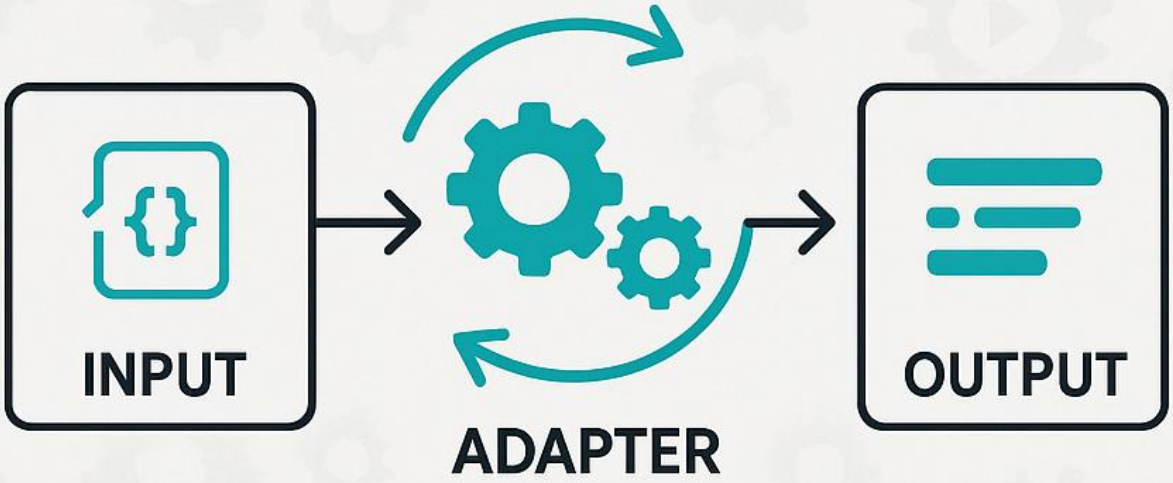
The Proxy Layer

PERFECT FOR:

- Service discovery abstraction
- Connection pooling
- Load balancing
- Protocol translation
- Circuit breaking & retry logic

ADAPTER PATTERN

The Translation Layer



Container transforms output
for external systems

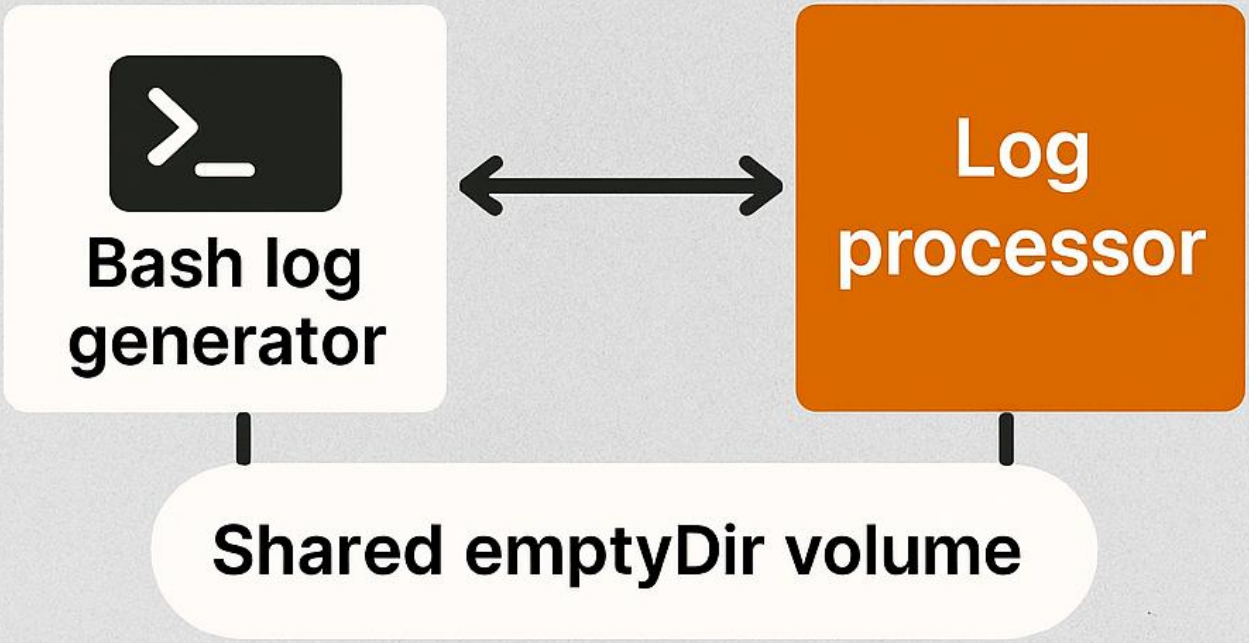
USE CASES:

- Log format transformation
- Metrics format conversion
- API response adaptation
- Protocol conversion
- Data normalization

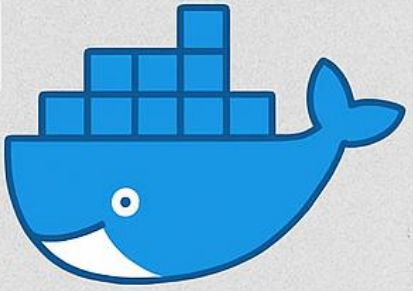
🔨 LET'S BUILD A LOGGING SIDECAR

ARCHITECTURE:

1. Bash log generator (main)
2. Log processor (sidecar)
3. Shared emptyDir volume



Real implementation with Docker & K8s!



docker

MAIN APPLICATION: Log Generator

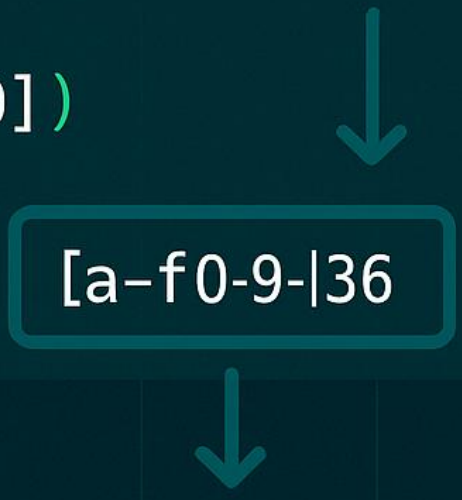
BASH

```
#/!bin/bash
while true; do
  MESSAGE= "$((date -Isecods) -
   GUID: $(uuidgen)"
  echo $(MESSAGE) >> /app/logs/
   my-test-log.log
  sleep $(shuf -i 1-3 -n 1)
  done
done
```

Generates timestamped logs with UUIDs

SIDECAR: Log Processor

```
#!/bin/bash
tail -f /var/log/myapp/*.log
i while read line; do
if [[ $line == [a-f0-9-]{36}]:
  then
  $(BASH_REMATCH[0])
fi
done
```



123e4567-e89b-12d3-a456-4266544000

Extracts UUIDs from log stream in real-time

POD CONFIGURATION

```
apiVersion: v1
kind: Pod
metadata:
  name: test-logger
spec:
  volumes:
    name: shared-logs
  emptyDir: {}
  volumeMounts
    name: shared-logs
  mountPath: /app/logs
```

RESOURCE CONFIGURATION

RESOURCE ALLOCATION

```
containers:
  my-test-logger
  resources:
    memory: 200Mi
    cpu:  600m
  - log-reader
    memory: 200Mi
    cpu:  600m
```

Proper resource boundaries prevent conflicts

🚀 DEPLOYMENT & VERIFICATION

```
# Deploy the pod
kubectl apply -f / pod.yaml

# Check status
kubectl describe pod test-logger

# View sidecar logs
kubectl logs pod/test-logger
-c log-reader --follow

✅ Live log streaming from
the sidecar!
```

INIT CONTAINERS

The Setup Crew

Run BEFORE main containers start

PERFECT FOR:

- Database migrations
- Configuration setup
- Dependency verification
- Security scanning
- Environment preparation

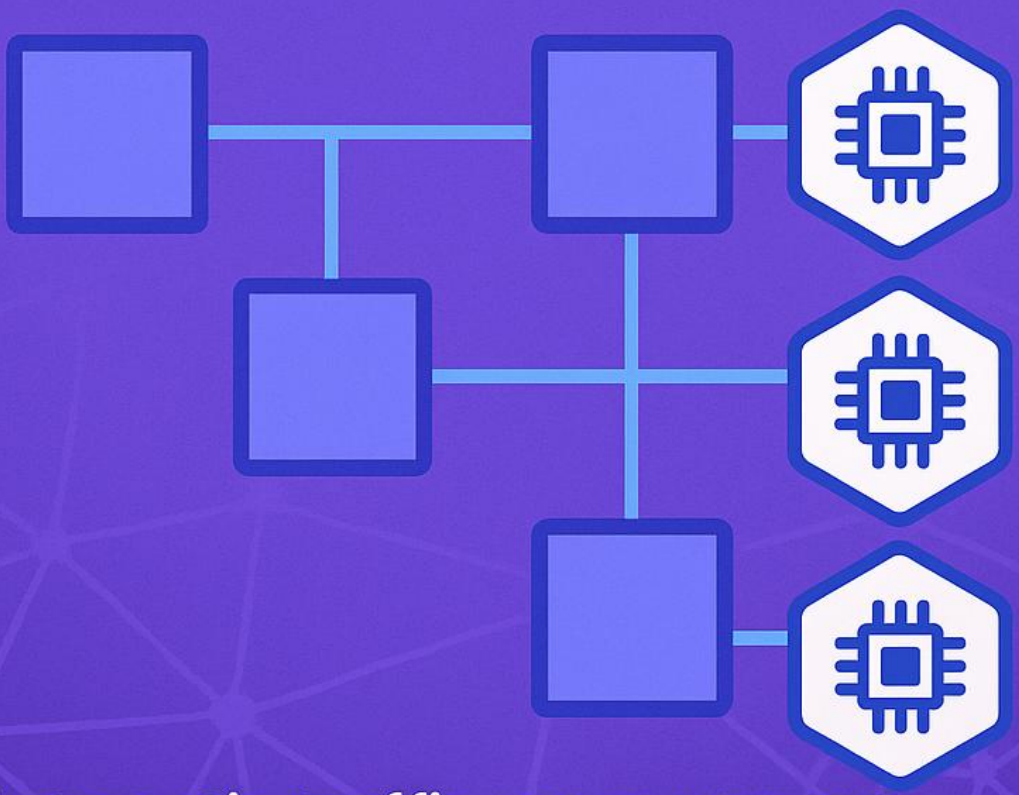
INIT CONTAINER ⋯⋯→ **MAIN CONTAINER**

initContainers run sequentially, then main containers start

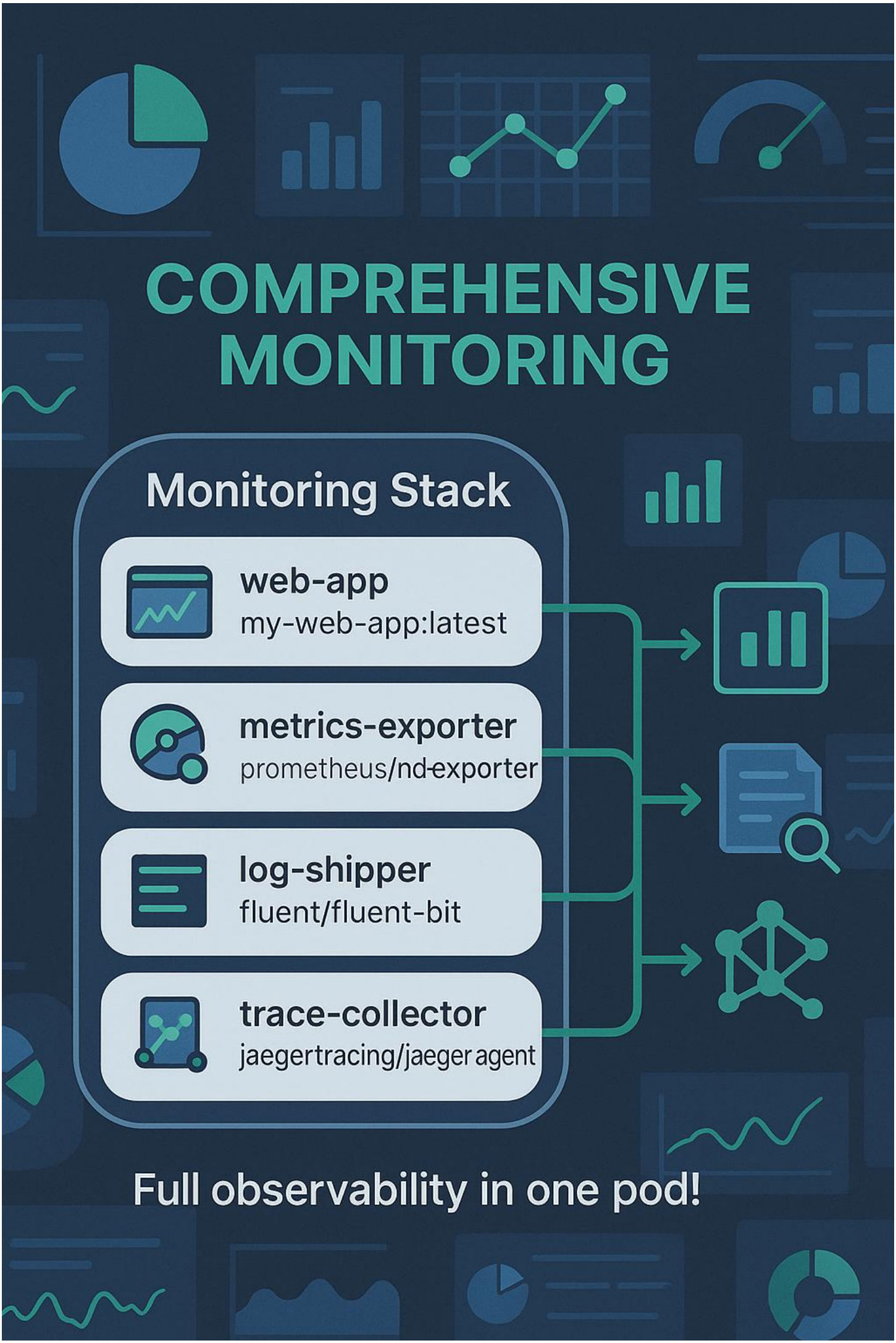
SERVICE MESH WITH SIDECARS

Istio automatically injects proxy sidecars

```
containers:
 - name: my-app
   image: my-apps:latest
 - name: istio-proxy
   image: istio/proxy2:1.20.0
```



Automatic traffic management, security, observability


COMPREHENSIVE MONITORING
Monitoring Stack
web-app
my-web-app:latest
metrics-exporter
prometheus/nd-exporter
log-shipper
fluent/fluent-bit
trace-collector
jaegertracing/jaeger agent
Full observability in one pod!

BEST PRACTICE

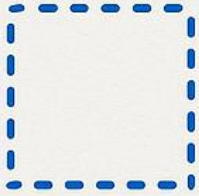
DESIGN PRINCIPLES

1 Single Responsibility

Each container =
one clear purpose

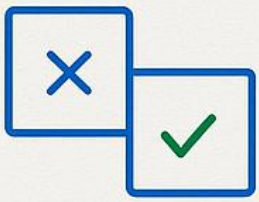
2 Resource Boundaries

Define appropriate limits



3 Failure Isolation

Containers fail
independently



4 Security First

Non-root users,
minimal images
network policies

COMMON PITFALLS

AVOID THESE MISTAKES

- Tight coupling between containers
- Resource competition
- Complex log management
- Poor debugging strategies

SOLUTIONS:

- Well-defined interfaces
- Proper resource allocation
- Standardized logging
- Comprehensive monitoring

PERFORMANCE OPTIMIZATION

PERFORMANCE CONSIDERATIONS

NETWORK

- Localhost connmunication (very fast)
- No encryption overhead
- Shared network namespace

STORAGE

- emptyDir for fast ephemeral storage
- tmpfs for high-performance temp storage

RESOURCES

- CPU sharing strategies
- Memory allocation patterns

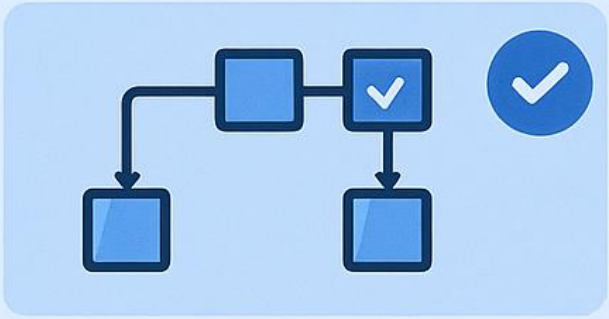
TESTING STRATEGIES

TESTING MULTI-CONTAINER PODS

UNIT TESTING

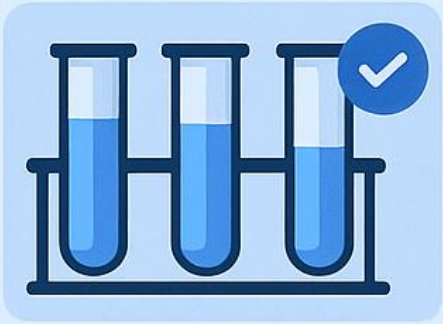
```
>_
docker run -rm
my-app:test
npm test
```

INTEGRATION TESTING



kubectl apply -f test-pod.yaml
kubectl wait –for=condition=ready
pod/test-pod

LOAD TESTING



Test combined resource usage under load

Each combiner **AND** their inter-actions!

Each container **AND** their interactions!

MIGRATION ROADMAP

FROM SINGLE TO MULTI-CONTAINER:

FROM TO

1. Identify separable concerns
2. Extract functionality
3. Define communication interfaces
4. Gradual rollout with feature flags
5. Monitor and validate

Start small, think big!

WHAT'S NEXT?

WEBASSEMBLY (WASM):

Lightweight alternatives to sidecars

SERVERLESS MULTI-CONTAINE:

Knative exploring pod-level scaling

EDGE COMPUTING:

Multi-container patterns at the edge

The future is multi-container native!

KEY TAKEAWAYS

- Think in terms of PODS, not just containers
- Embrace SEPARATION OF CONCERNS
- Design clear COMMUNICATION interfaces
- Monitor at CONTAINER & POD levels
- Start small, scale thoughtfully

Ready to transform your architecture?

Like if this helped!

Share with your team!

Comment your experiences!

#Kubernetes #DevOps #PlatformEngineering