
Challenges to Writing
Cloud Native Applications
Vallery Lancey,
Lead DevOps Engineer, Checkfront
Checkfront

Checkfront www.checkfront.com

Checkfront is a booking management platform trusted by over 4,000 tour and activity operators worldwide.

Our platform empowers businesses to sell tours and activities on any website by providing live inventory management, dynamic pricing, customer notifications, and channel distribution. Checkfront makes it easy to grow your business by automating your administration and housing all your business tools in one place.

About Vallery (that's me!)

My background: software engineering & systems administration.

What I do at Checkfront as the lead "DevOps engineer":

- Long term infrastructure & platform planning.
- Training and mentorship.
- Day to day ops.
- Contribute to product & tooling development.
- Lead the cloud push.

/vllry

This talk will use Kubernetes for concrete examples, but will focus on general cloud principals.

We're going to cover 3 areas:

- Properties of cloud native apps
- What cloud platforms cause/impose
- How to develop for those cloud platform constraints

Raise your hand if you have a question about the material at hand. For other questions, we'll have a full Q&A at the end.

What Makes Software Cloud Native?

The CNCF says cloud native software uses open source tech to be:

1. ***Containerized.*** Each part (applications, processes, etc) is packaged in its own container. This facilitates reproducibility, transparency, and resource isolation.

2. ***Dynamically orchestrated***. Containers are actively scheduled and managed to optimize resource utilization.

3. ***Microservices oriented.*** Applications are segmented into microservices. This significantly increases the overall agility and maintainability of applications.

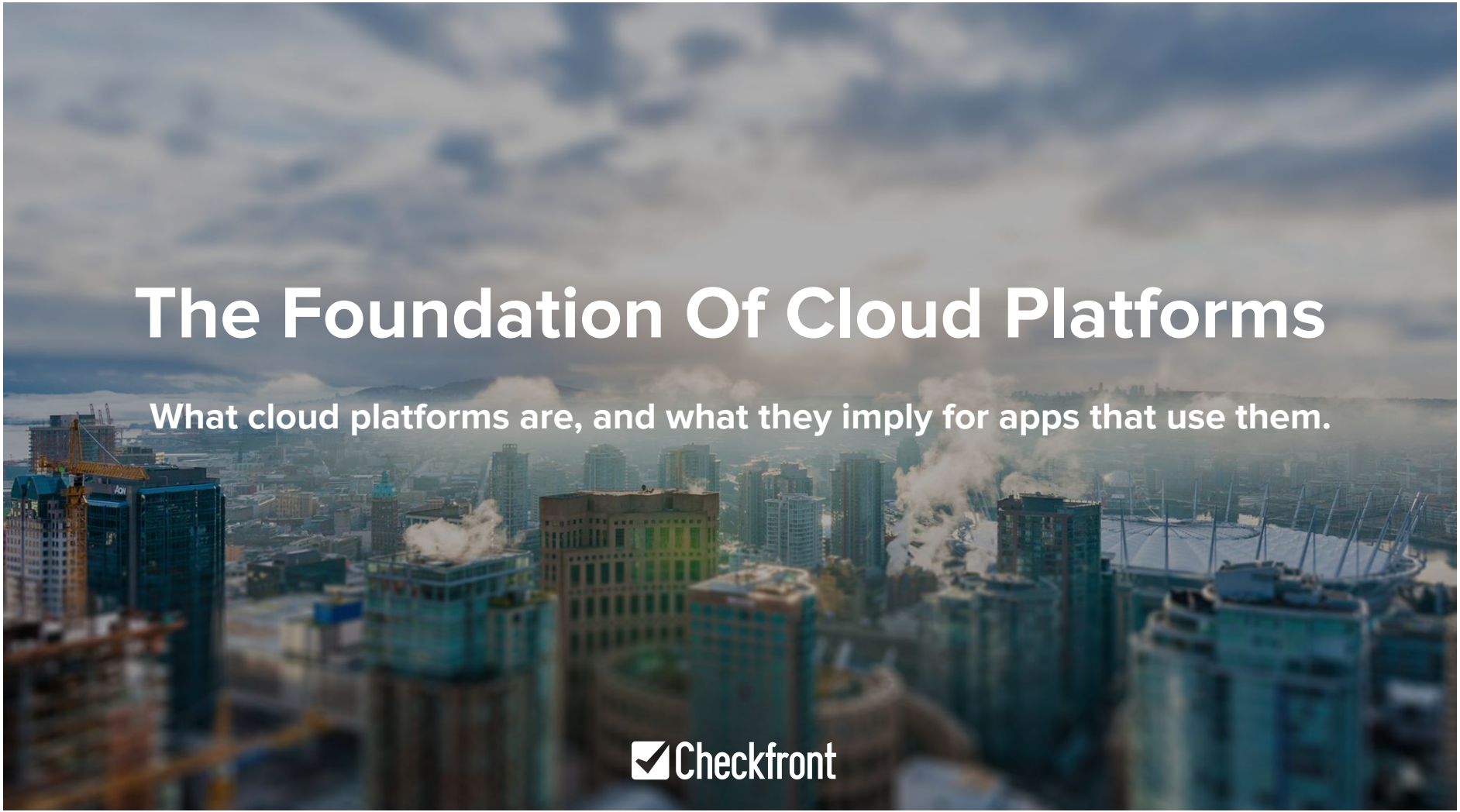
cncf.io/about/faq

Cloud Native

1. Automatic scaling, load balancing, and replication.
2. Low-touch deployment.
3. Declarative state (config + apps).
4. Distinct separation between the platform, and the services running on it.

... Or Not

1. Changing the number of instances requires intervention.
2. Deploying is largely manual.
3. Applications need manual, imperative administration.
4. Little or no separation of concern between the host systems and the service(s) on them.


The Foundation Of Cloud Platforms
What cloud platforms are, and what they imply for apps that use them.
Checkfront

What is a Cloud Platform?

The platform is a fundamental requirement of a cloud system.

It handles...

- Infrastructure provisioning and teardown.
- Application deployment.
- Network discovery and routing.

Roles: The Runtime Environment

The platform takes on the role of running the app.

The runtime environment includes:

- (Virtual) hardware
- Container runtime
- Config/secret injection
- Scaling
- Service connectivity

The app becomes dependent on this behavior, and its limitations.

Roles: Ephemeral Replication

Service replication is a core tenet of cloud software.

Replication relies on treating containers as ephemeral. Therefore, data must be explicitly handled to be preserved.

This makes persistent data the biggest snag in designing cloud native software.

Roles: Network Provider

Deployed applications will have changing # of replicas, and changing IP addresses. This requires service discovery, routing, and load balancing.

The platform is responsible for translating requests for “service X” to a suitable instance of service X.

Cloud platforms typically use an edge proxy (EG Kubernetes ingress) to route outside systems from a single endpoint, to services in the cluster. This edge proxy is often responsible for TLS termination and port/protocol support.

- Persistent data storage
- Service coupling
- Internal API calls
- Testing & local development

Persistent Data Storage

4 fundamental patterns for replication:

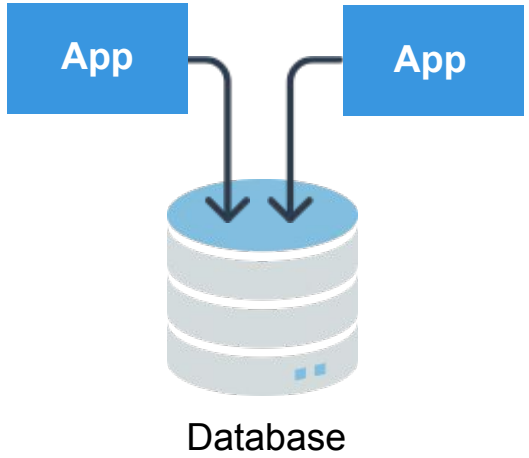
1. Only 1 replica (no replication).
2. Only 1 replica of any given shard, multiple shards.
3. Multiple replicas, data is volatile and session specific.
4. Multiple replicas, data is replicated between instances (static or dynamic replicas).

Persistent Data Storage: Single Replica

Replication strategy #1: don't.
Rarely a good idea.

- May be the temporary result of a lift-and-shift.
  - *EG: 1 replica deployment of "the europe-west1 server".*

- May be used for services that can accept downtime.
  - *Internal, not productivity-vital services.*
  - *Features that can be impaired with little overall UX impact.*



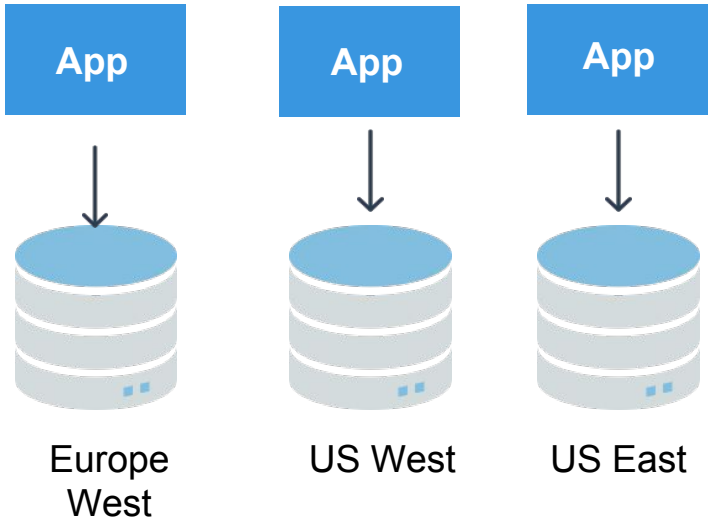
Persistent Data Storage: *Unreplicated Shards*

Replication strategy #2: don't replicate data, but split data into shards.

Extremely common pattern. A natural quickfix for outgrowing strategy #1.

This allows the data store to be split into multiple servers.

- Usually done with client units (users, companies, websites, etc).
- Usually done with regional shards.

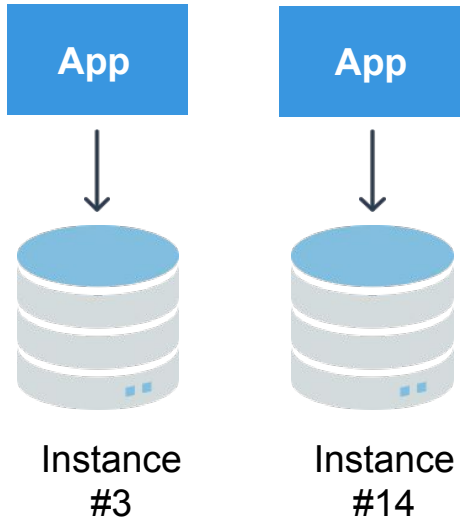


Persistent Data Storage: Multiple volatile copies

Replication strategy #3: have multiple replicas of the data store, but don't replicate data between them.

Primary use of this pattern: accidentally.

Can be used for caches, to avoid the write overhead of updating a clustered cache.

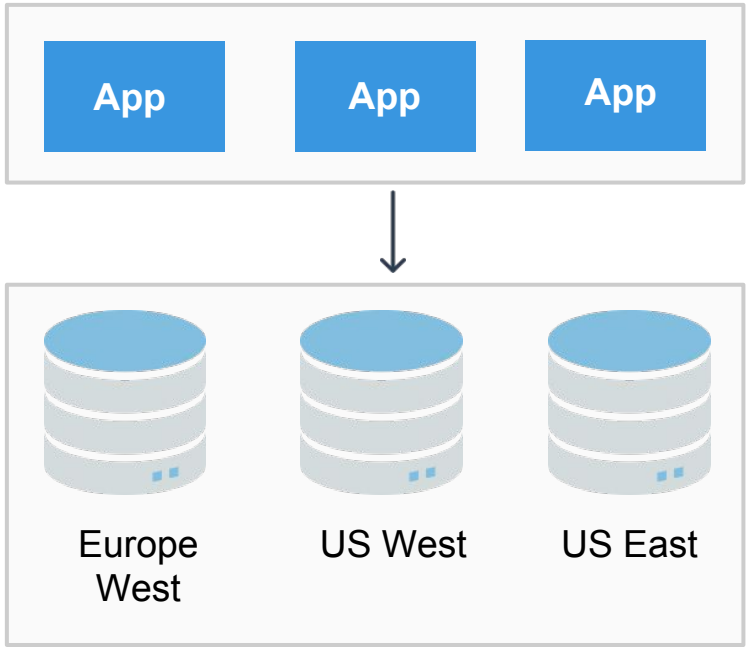


Persistent Data Storage: Runtime data replication

Replication strategy #4: have multiple replicas of the data store, replicate data between them at runtime.

This is usually the ideal pattern, but requires the most ops work.

Ease/feasibility is determined by the underlying data store. Some (EG Cassandra) are designed for this. Others (EG SQLite) are not at all.



Persistent Data Storage: Runtime data replication

In total, service spinnup entails:

- Finding or creating a suitable volume.
- Binding to that volume.
- Finding the network address of a seed node.
- Connecting to the seed node and syncing data.
- Election of new seed nodes.

Third party services (for example, GCS or S3 for images) are extremely helpful for small teams. Runtime replication has a large setup and ongoing maintenance cost.

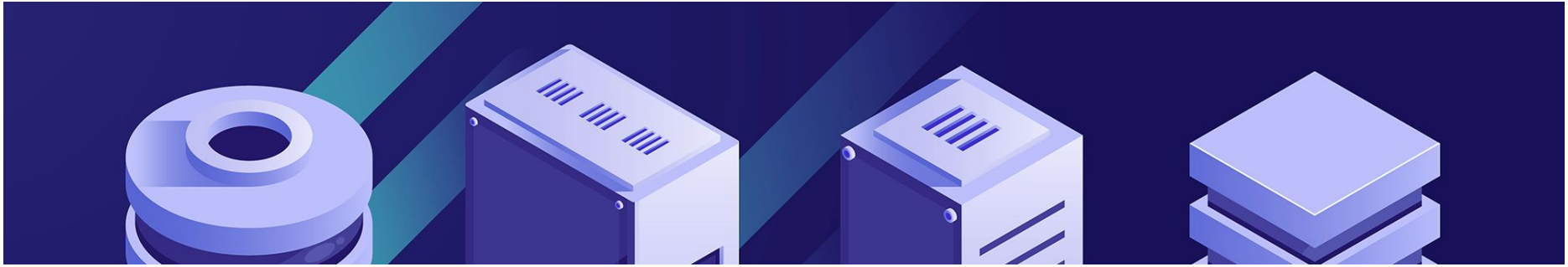
Service Coupling

We want to be able to tightly couple service instances if they frequently interact, to reduce network delays.

Many cloud platforms have a concept of closely-linked services. In Kubernetes, a ***deployment*** spec can define multiple containers. Deployments create instances called ***pods***.

- **Containers in a pod share a network and optional volumes.**
  - *Creates 1:1 container state within a pod.*

- **Containers in a pod are run on the same machine.**
  - *Very fast network connection.*

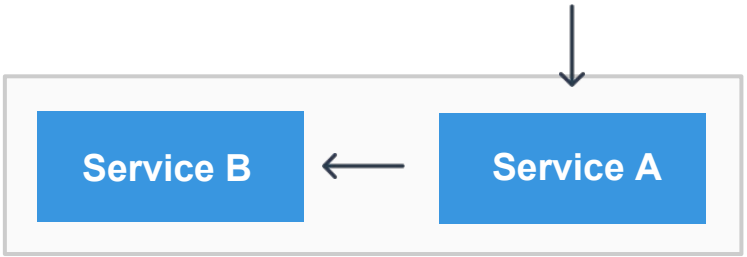


Service Coupling: What To Couple

- **Databases and services that interact directly with them.**
  - *Provided database replication doesn't spread undue constraints to the service.*

- **Services that call one another.**
  - **Especially high with high frequency or throughput.**

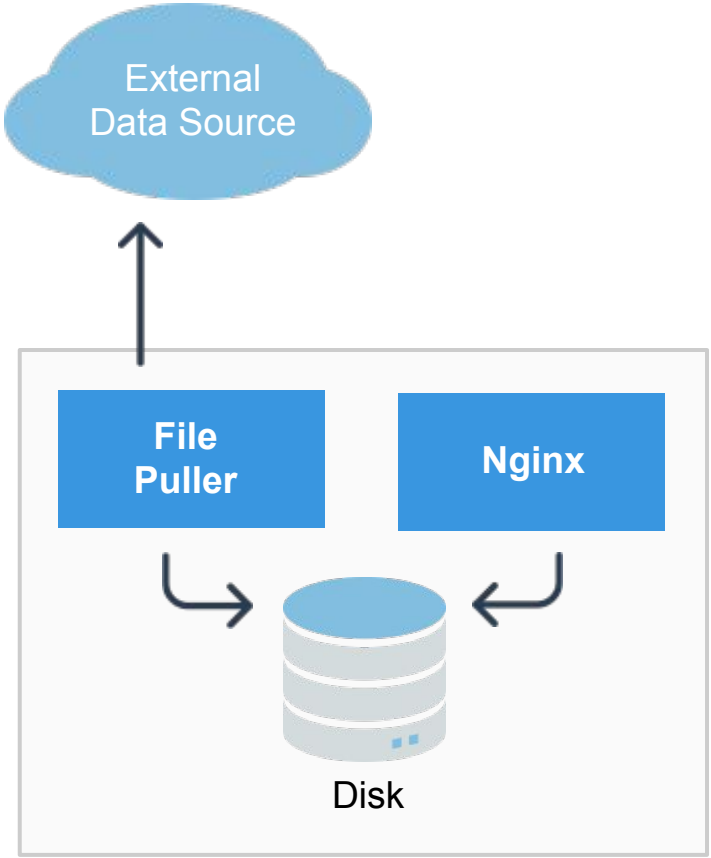
- **Services that share 1:1 instance state.**



In this example, we pull files at runtime, rather than building them into the Nginx image.

This pattern can be useful for:

- Allowing content changes independently of deployment rollouts.
- Directly passing on-disk resources between services.



Pod Example: File Puller

Service Coupling: When Not To

If we couple too many services together, we see some consequences:

- Pods with persistent data stores are limited by data replication concerns
- Pods become harder to schedule, requiring more contiguous resources.
- Services can’t scale selectively. This allocates resources needlessly.
- Access controls typically apply pod-pod, not for within a pod.*

Sounds like a monolith...

* This is more of a platform-specific issue.

Internal API Calls

We can try to co-locate services that interact a lot, but sometimes it's not feasible.

Cloud platforms & distributed systems tend to exaggerate intra-app latency:

- Running as independent pods increases latency
- Splitting one service into two co-located service increases latency.

Therefore, we want to be smart about how we build our services and APIs, to reduce that back-and-forth.

Internal API Calls: Sources of Delay

Substantial network delays normally stem from excessive synchronous calls.

Use threads or async as much as you can!

There are other problems to improve/avoid. Most are in upstream service designs, which force downstream services to make more calls than necessary.

Internal API Calls: Simplifying API Actions

APIs are often designed around database CRUD, rather than the high-level actions taken by downstream systems.

EG: fetch normalized data, then use results to make another fetch.

This can be avoided by adding endpoints that encompass downstream actions.

Internal API Calls: Batch Endpoints

Use batch API endpoints for calls made repetitively.

Batch endpoints decrease network + auth overhead.

Batch endpoints can be used to leverage work common to individual queries.

- Avoid repeating JOIN/WHERE/subquery work on database calls.
- Re-use data fetched in each call.
- Avoid re-transferring large payloads in the API call.
- Re-use initialized data structures.

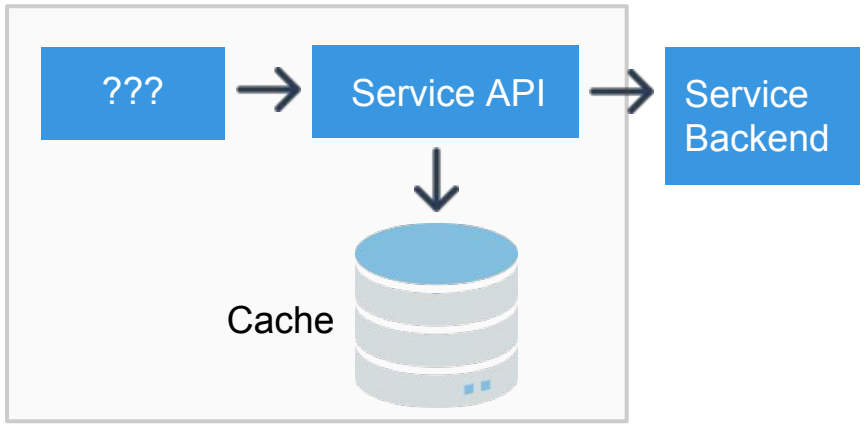
SQL batch example: https://github.com/vllry/kubecon18-batch

Internal API Calls: Caching

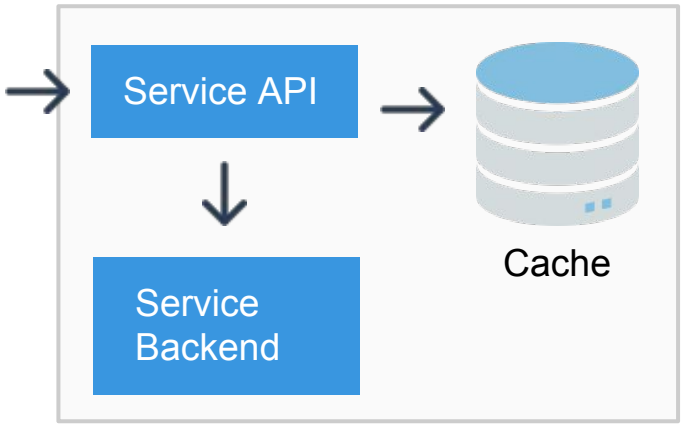
Use caches for stateless services. This will save processing time, and potentially network time.

Caches can be done server-side or client-side.

Client-side cache



Server-side cache



Testing

Integration and end-to-end testing is crucial in multi-component systems (which makes it a huge concern with microservices).

Cloud platforms and apps are still linked, and not separate concerns.

We will need to bring in some dependencies:

- Runtime engine (EG Docker)
- Config/secret injection
- Service routing/discovery
- Some coupled services (EG upstream database)

Testing:
Discrete components

Testability (and development) should be a design decision.

We want to be able to isolate segments of the app (EG a specific API endpoint), and run those sections in isolation. This is similar to unit testing scope concerns, on a much bigger scale.

To isolate segments, we should try to design dependencies in a “tree” fashion. To test a given service, we need to test/run (at most) that services and all downstream services.

Unit Testing Composition

 # Awkward to test.

```
def update_median():
        values = db.fetch_all_thingy()
        median =
statistics.median(values)
        db.write_median(median)
```

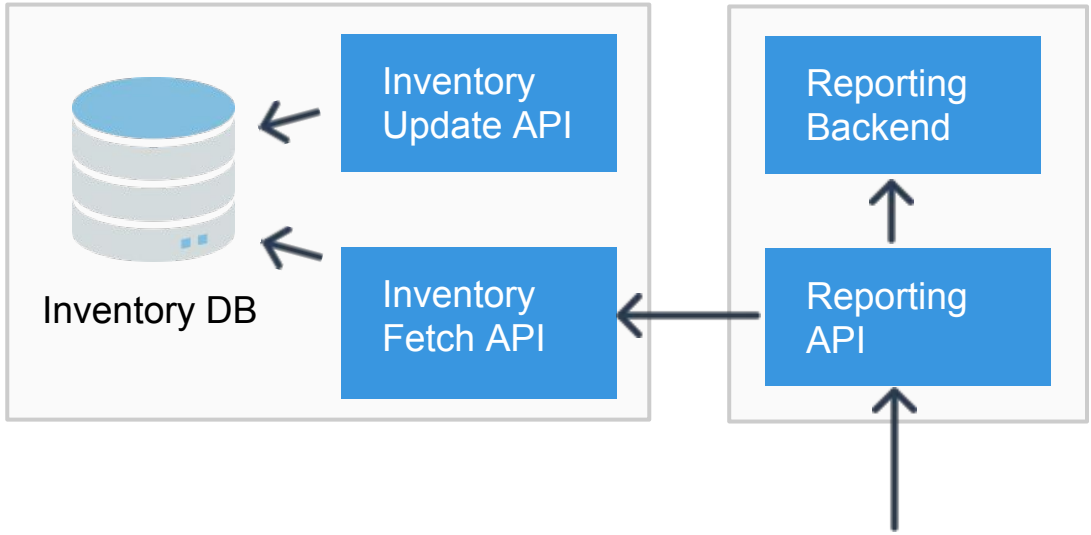
 # Still awkward to test, but the main logic is seperate.

```
def update_median():
        values = db.fetch_all_thingy()
        median = calculate_median(values)
        db.write_median(median)
```

 # Easy to test.

```
def calculate_median(values):
        return statistics.median(values)
```

Service Testing Composition



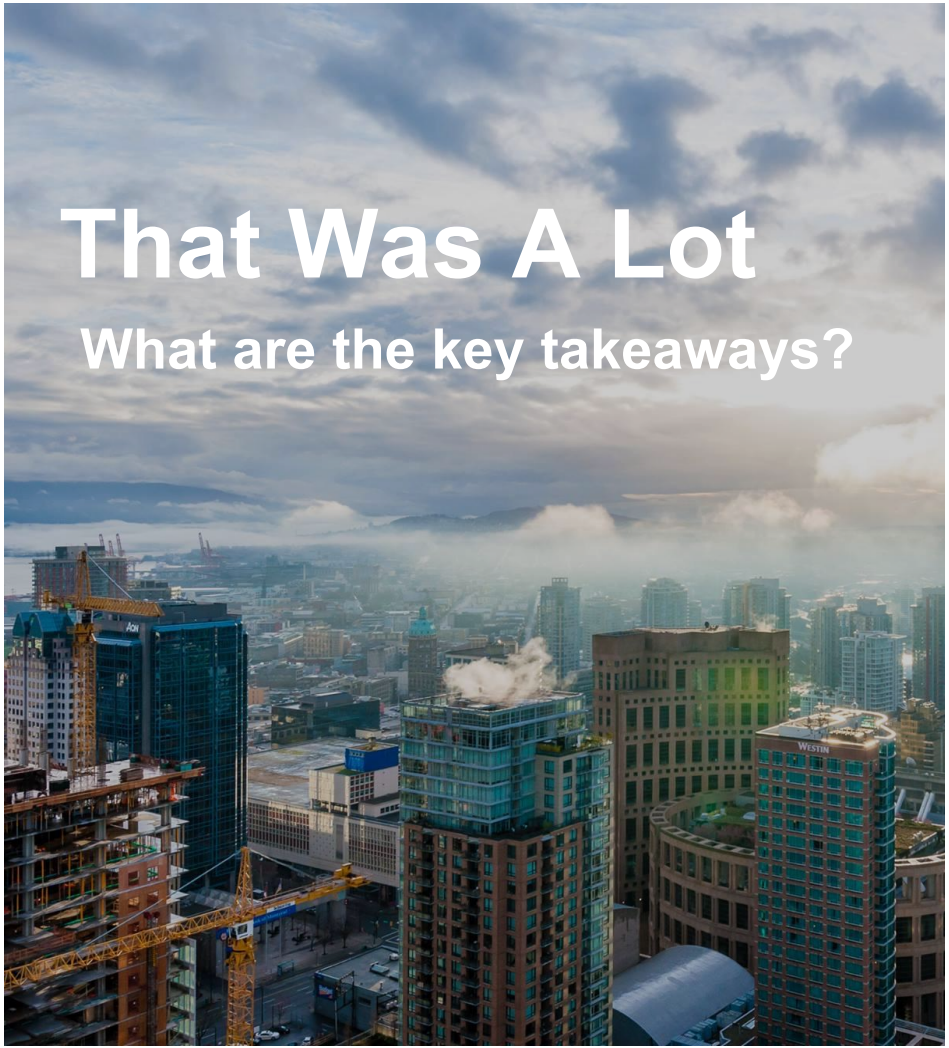
Local Development

Testing concerns relate heavily to local development - devs probably don't want to build and run the entire app stack in Minikube.

Try to make individual services friendly to run natively, without necessarily needing Docker. It can be unfriendly to development, and developers may need to work around Docker behavior (ruining "the same anywhere" anyway).

- Limit dependencies (especially circumstantial ones).
- Limit OS sprawl.

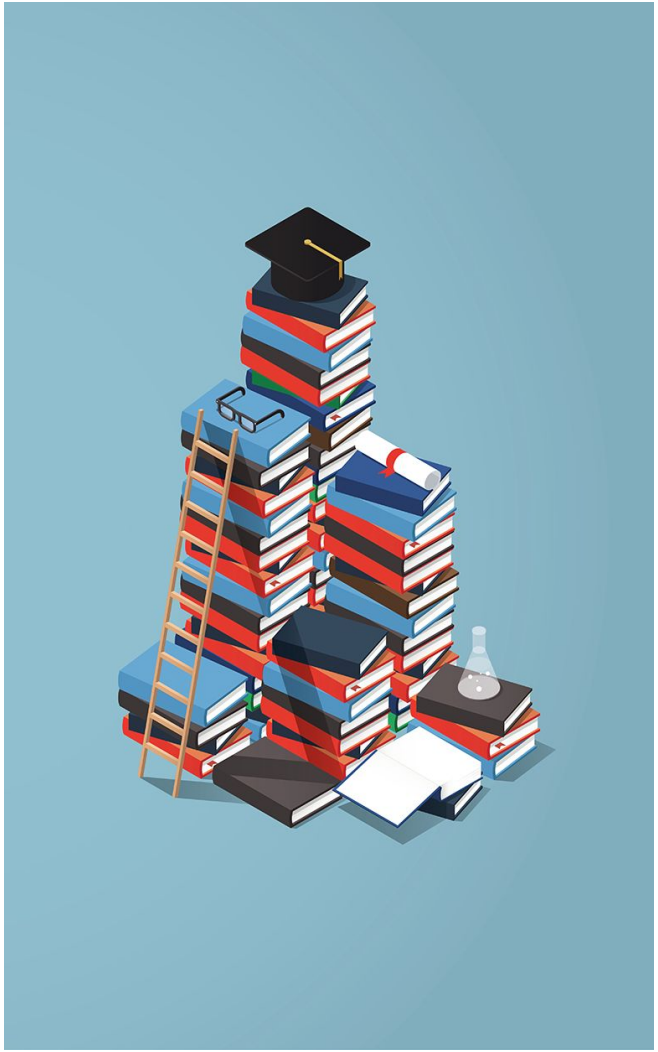
If these are difficult, the service may be too big and unspecialized.



- Many small but key details are platform-specific.
- Platform/Ops specialists should be active members of all architecture decisions.
- Service composition has a *huge* impact on performance, runtime concerns, and testability. Couple closely when performance requires it, and avoid tight coupling otherwise.

Some Suggested Reading

- **“Architecting For Scale”**, Lee Atchison
- **“Designing Distributed Systems”**, Brendan Burns
- **“The 12 Factor App”**, Adam Wiggins, https://12factor.net/
- **“Cloud Native Infrastructure”**, Justin Garrison & Kris Nova
- Guides to the platform of your choice (EG **Kubernetes Up And Running**)
- General material around building microservices, and microservice madness.



Time For Questions!

You can find me @vllry on Twitter.
(Or look for the hair this week.)