KubeCon | CloudNativeCon

North America 2023

KubeCon | CloudNativeCon
North America 2023

Cluster API Deep Dive
Improving performance up to 2k Clusters

Fabrizio Pandini & Stefan Büringer, VMware

Who we are

Stefan Büringer
Staff Engineer
VMware

Fabrizio Pandini
Staff Engineer
VMware

Before we start…

All facts in this presentation are based on a true story

This is a talk about the performance optimization work done in Cluster API v1.5.0,
but the theory and the concepts behind this work **applies to all Kubernetes controllers**.

Same for the lessons learned and experiences we gathered.

Agenda:

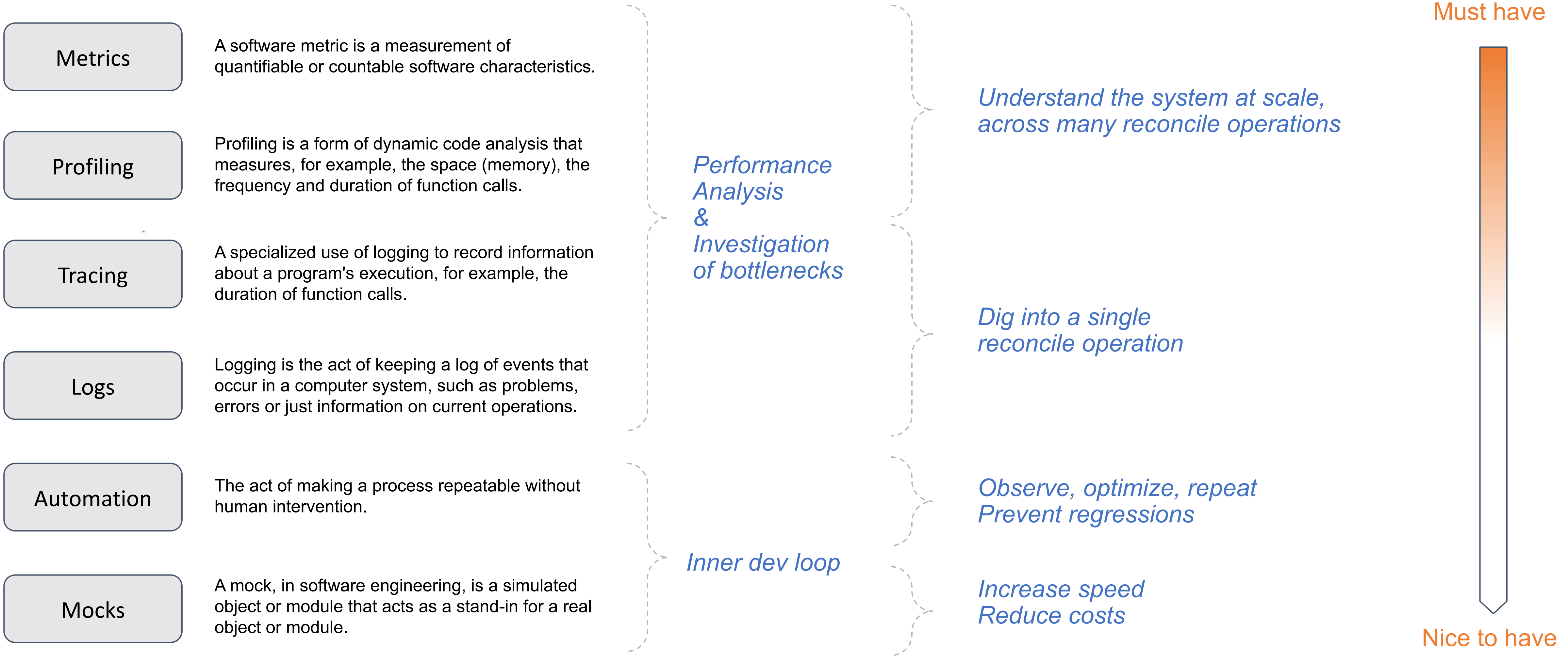
- How to get ready for performance optimization
- Scaling Cluster API up to 2K clusters
  - Controller optimization 101
  - Performance optimization in practice

Get the right tools for the job

What you need

Why do you need it

TL;DR



Metrics: A software metric is a measurement of quantifiable or countable software characteristics.

Profiling: Profiling is a form of dynamic code analysis that measures, for example, the space (memory), the frequency and duration of function calls.

Tracing: A specialized use of logging to record information about a program's execution, for example, the duration of function calls.

Logs: Logging is the act of keeping a log of events that occur in a computer system, such as problems, errors or just information on current operations.

Automation: The act of making a process repeatable without human intervention.

Mocks: A mock, in software engineering, is a simulated object or module that acts as a stand-in for a real object or module.

Performance Analysis & Investigation of bottlenecks

- *Understand the system at scale, across many reconcile operations*
- *Dig into a single reconcile operation*

Inner dev loop

- *Observe, optimize, repeat Prevent regressions*
- *Increase speed Reduce costs*

Must have

Nice to have

What metrics do you need ?

Metric categories

Why do you need it

“User-facing” Metrics

A user-facing metric is a measurement of quantifiable or countable software characteristics that **directly relates to the user perception of how the system works**.
E.g. Machine provisioning time

Define goals
Measure success

“Internal” Metrics

A “internal” metric is a measurement of quantifiable or countable software characteristics that **documents how the system works under the hood**.
Not performing “internal” metrics are usually perceived by the users as slowness, unresponsiveness etc.
E.g. Average reconcile time

Understand how the system performs at scale

Point to what needs to be optimized next

How can I get all of this?

You get most of it for free! (not everything)

CAPI observability stack

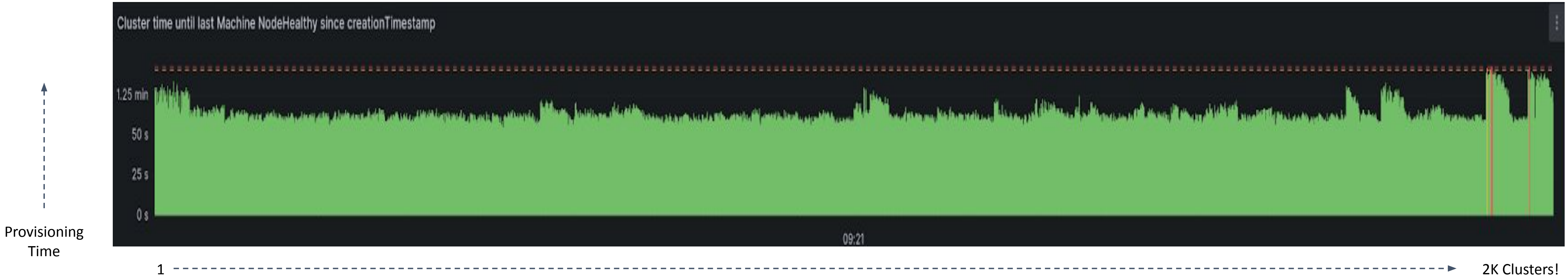
🟢	“User-facing” Metrics	kube-state-metrics provides the basic tooling to get metrics from CRDs **You need to provide a config defining how metrics should be derived from your CRDs**	Config for CAPI CRDs kube-state-metrics deployment Prometheus deployment Grafana deployment
	“Internal” Metrics	Go SDK, client-go, and controller-runtime have you covered Consider adding custom metrics only if strictly necessary	EXTRA BONUS **Dashboards included!**
	Profiling	Go SDK has you covered	Parca deployment
🟢	Tracing	**You need to instrument your code** Start small, add more when you need it	Tempo deployment
	Logs	You probably already have them Consider adding more logs only if necessary	Promtail deployment Loki deployment
🟡	Automation	**You need to write your own automation**, but you probably already have something	Scale E2E test
🟠	Mocks	**Mocking complex systems like a cloud machine can be time-consuming** The investment usually pays back	Cluster API In Memory provider

2k Clusters!

After getting the right tools for the job, we defined our performance goal:

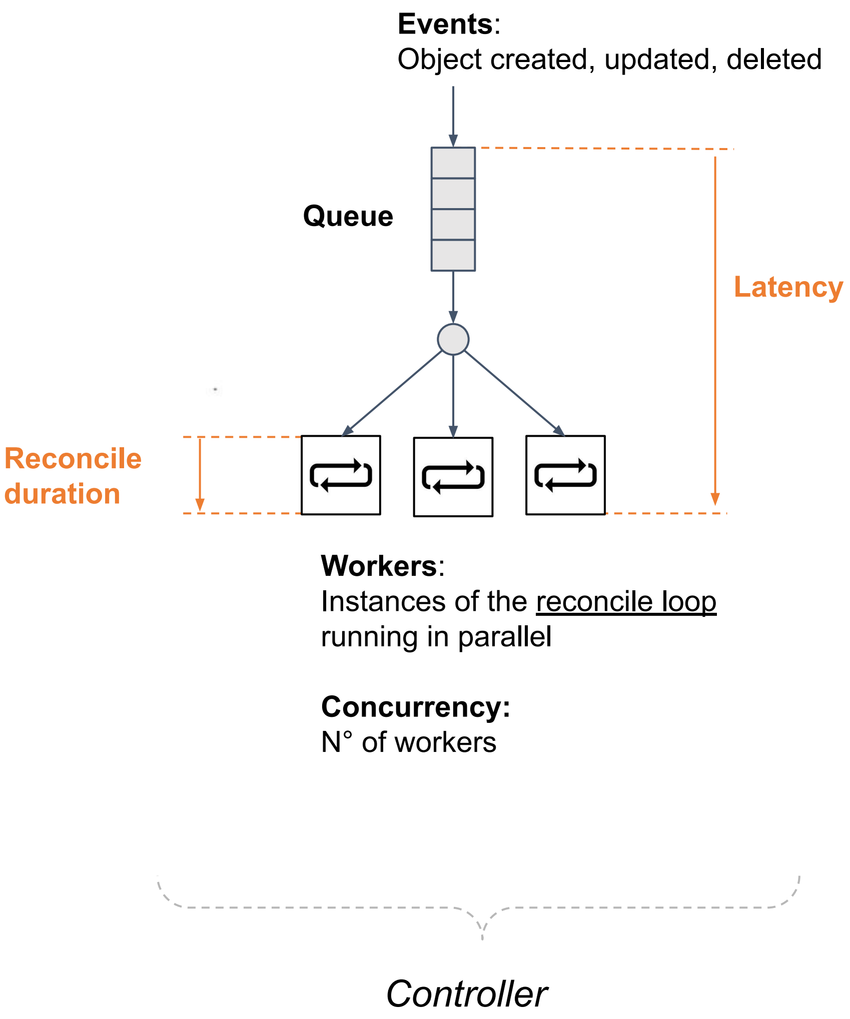
Cluster provisioning time (a business metric) must remain ~constant from Cluster n°1 to n° 2k

Spoiler, we did it! The delta between the average time to provision the first 100 clusters and the last 100 clusters show a neglectable increase of +2.3%.



This is great!
But the most interesting part is to look back at how we managed to **keep the system responsive** while scaling up (and while working at scale).

Controller optimization 101



Events:
Object created, updated, deleted

Queue

Latency

Reconcile duration

Workers:
Instances of the reconcile loop running in parallel

Concurrency:
N° of workers

Controller

The main performance characteristic of a controller is the **latency** between an **object create, update or delete** and its **successful reconciliation**

The "*worst case latency*" is the time required until the last object in the queue can be processed, and it can be determined via:

$$\frac{\textit{\# objects in the queue}}{\textit{concurrency}} \textit{ * reconcile duration}$$

An example:

$$\frac{\textit{2k clusters}}{\textit{10 workers}} \textit{ * 3s = 10m}$$ 😟

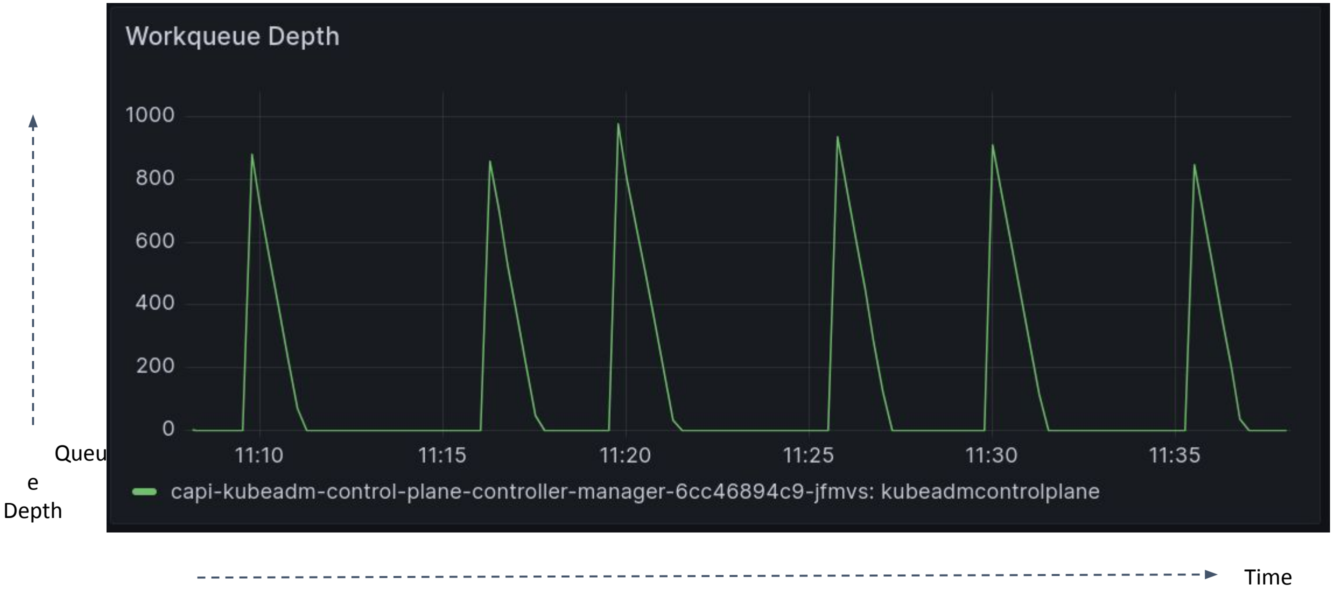
Controller optimization 101

Is it possible that all 2k Clusters are in the reconcile queue at the same time?

Yes, because on top of events generated when an object is created, updated or deleted, controller-runtime does a **periodic resync**, i.e. it periodically adds all objects to the queue.

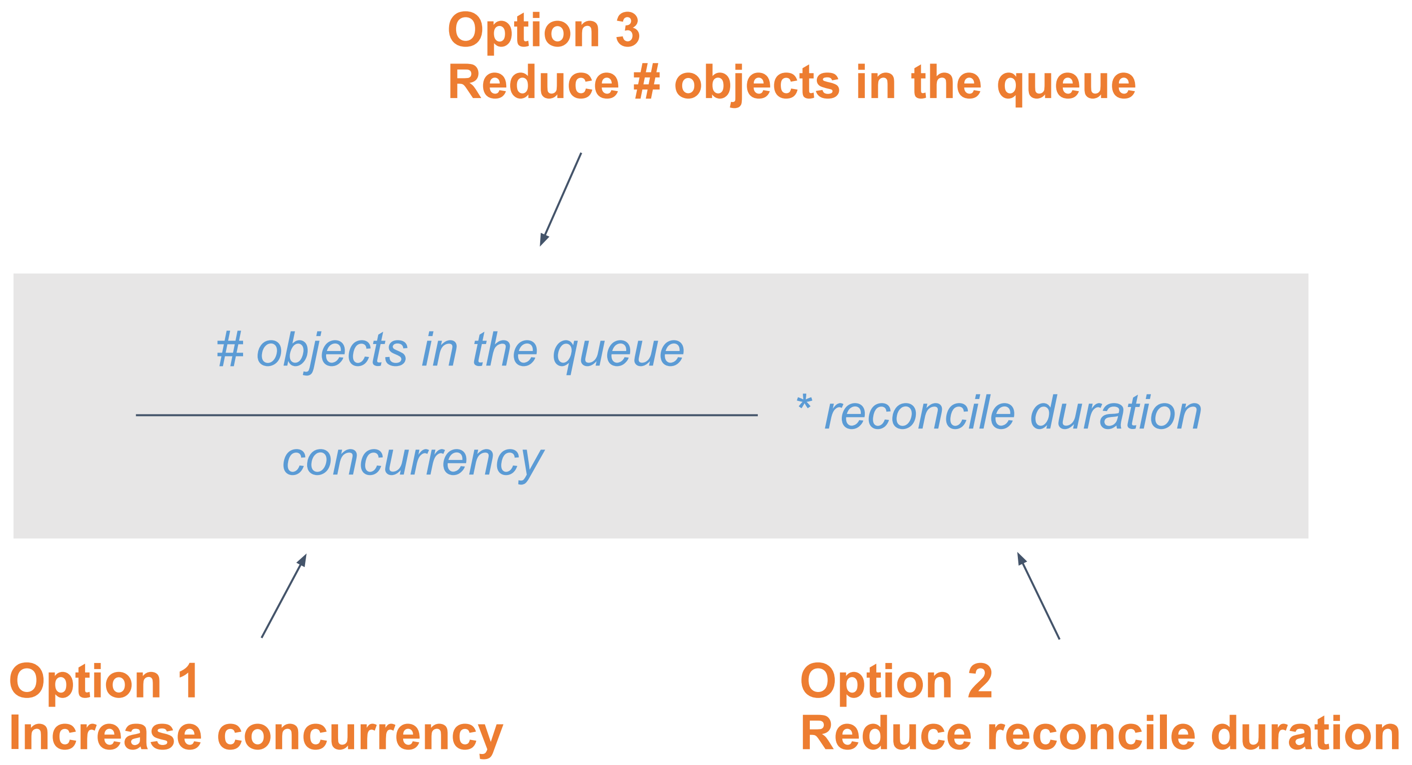
Additionally controllers can **"requeue" an object back into the queue**.

To stay responsive after the resync/requeue **controllers have to process events as fast a possible.**



Controller optimization 101

How to improve: $\dfrac{\textit{2k clusters}}{\textit{10 workers}} * 3s = 10m\ ?$



Option 3
Reduce # objects in the queue

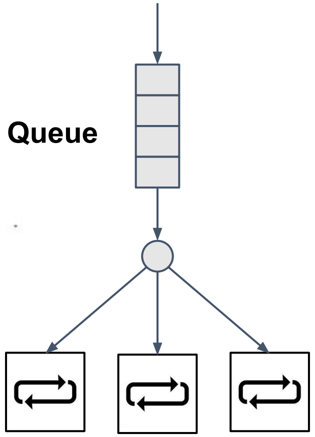
$$\frac{\textit{\# objects in the queue}}{\textit{concurrency}} * \textit{reconcile duration}$$

Option 1
Increase concurrency

Option 2
Reduce reconcile duration

Option 1: Increase concurrency

Events:
Object created, updated, deleted



Queue

Workers:
Instances of the reconcile loop running in parallel

Concurrency:
N° of workers

If we use only one worker (i.e. concurrency 1) the system will reconcile one object at a time.
This not performant enough for high scale.

On the other side, **if we use too many workers, this might not help the system to scale**, because it will put too much pressure on the API server.

In Cluster API the default concurrency is 10 workers:

○ Based on our observation 10 is good trade off when controllers have an average reconcile duration lower than 200-250ms

○ We only run the controller with the highest reconcile duration (KCP) with more workers, but to make it sustainable, **we invested a lot in improving KCP**.

TL;DR increasing concurrency is not a silver bullet

Option 2: Reduce reconcile duration

Reducing reconcile durations is more complex than tuning the number of workers.

However, **if you follow the lead of metrics, profiles and traces** you can **improve performance by doing small, surgical changes**.

In Cluster API we **iteratively improved** the performance by:

Removing noise

Identifying slow reconcilers

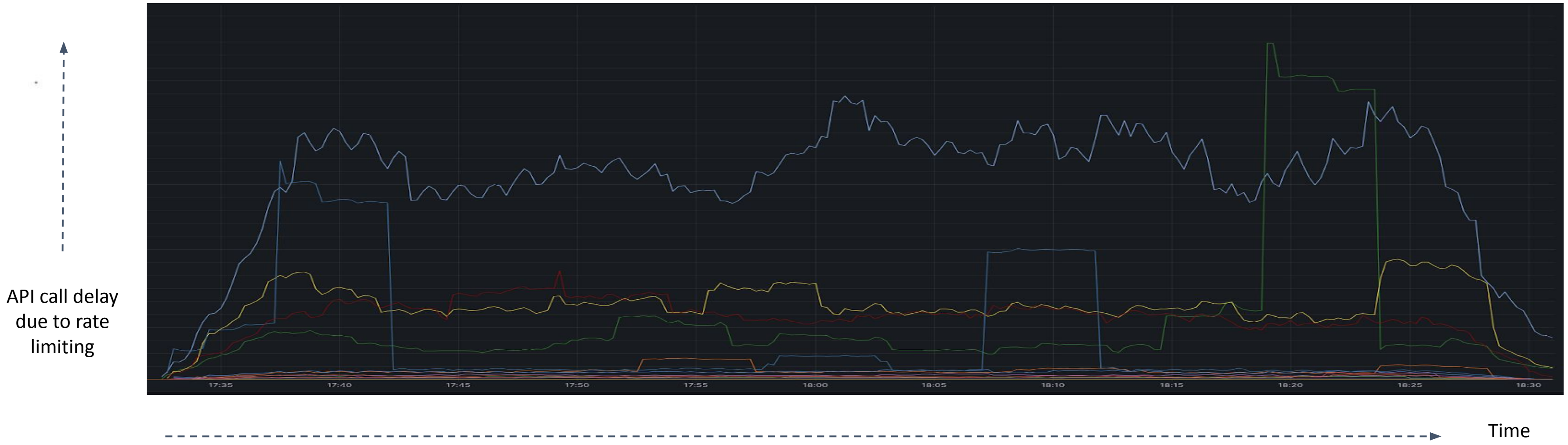
Removing bottlenecks

Repeat

Remove noise: client-side rate limiting

Kubernetes has client- and server-side rate-limiting[1] of requests to the API server, which are mechanisms that protect the system from clients doing excessive API calls.

Cluster API's default rate-limiting is 20 QPS and 30 burst, **which is enough for up to 100 Clusters.** When you cross this limit, rate-limiting "randomly" delays your API calls, and this makes reconcile durations non-deterministic[2].



TL;DR; when running at scale increase QPS and burst (we used 100 QPS, 200 Burst)

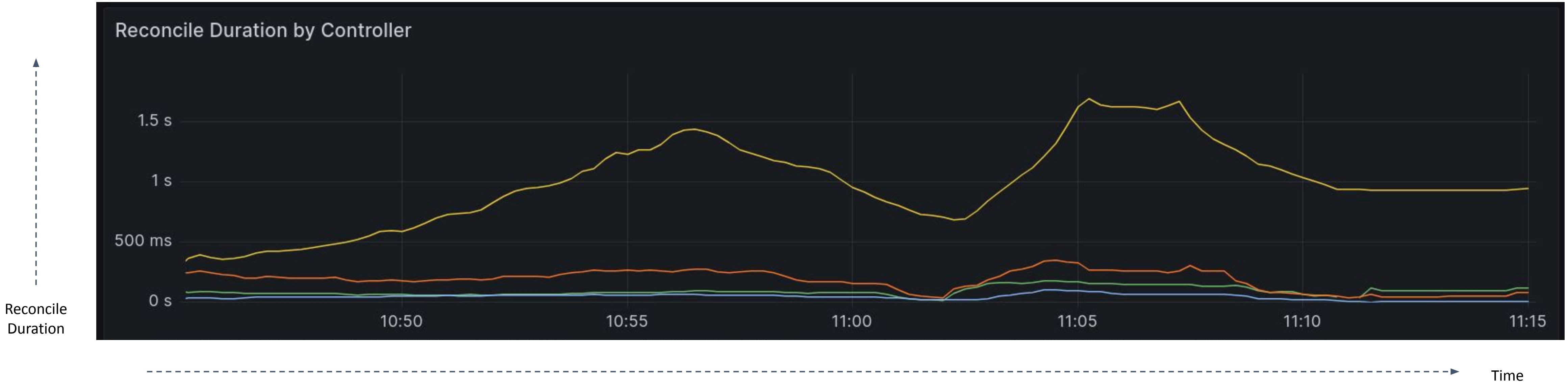
[1] API Priority and Fairness

[2] Increasing concurrency doesn't help as all reconcilers share the same client and thus also the same rate-limiting.

How to identify slow reconcilers

KubeCon CloudNativeCon North America 2023

This step is easy! Just run the scale test and look at the “reconcile duration” controller-runtime metric



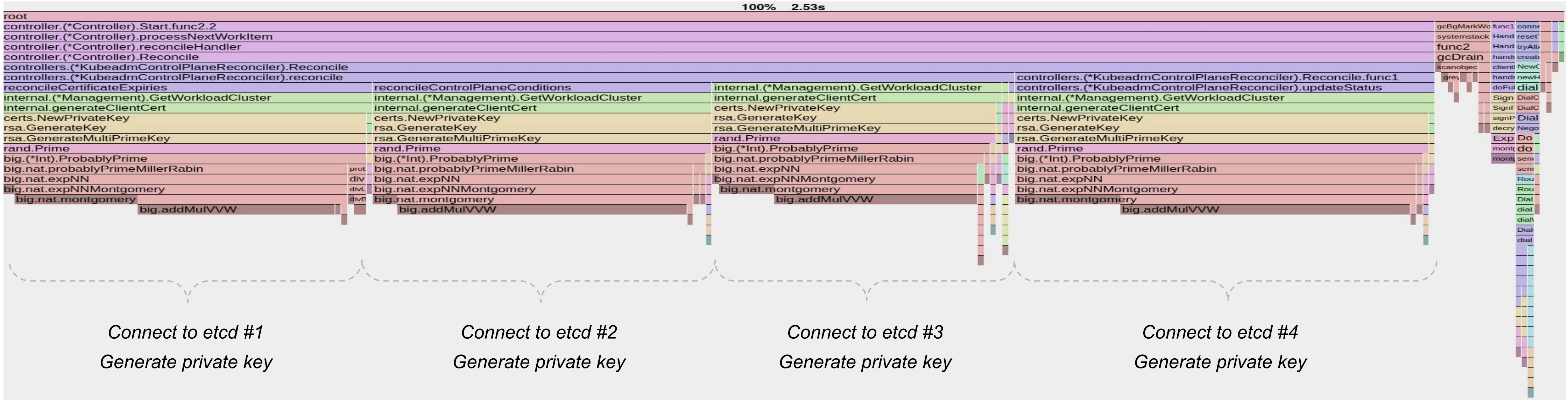
Next: focus on slow reconcilers one-by-one and dig deeper via metrics, profiling and tracing.

The following slides show what we found and how we improved performance.

Avoid duplicate expensive operations

We started looking at profiling to check where controllers were spending most of the time.
The first thing we noticed was that some controllers were **doing the same expensive operation multiple times**. This is obviously not good for performance.

For instance, KCP, when connecting to etcd, was generating a private key and a certificate multiple times; this is an expensive operation. Another expensive operation is e.g. creating Kubernetes clients.



Connect to etcd #1
Generate private key

Connect to etcd #2
Generate private key

Connect to etcd #3
Generate private key

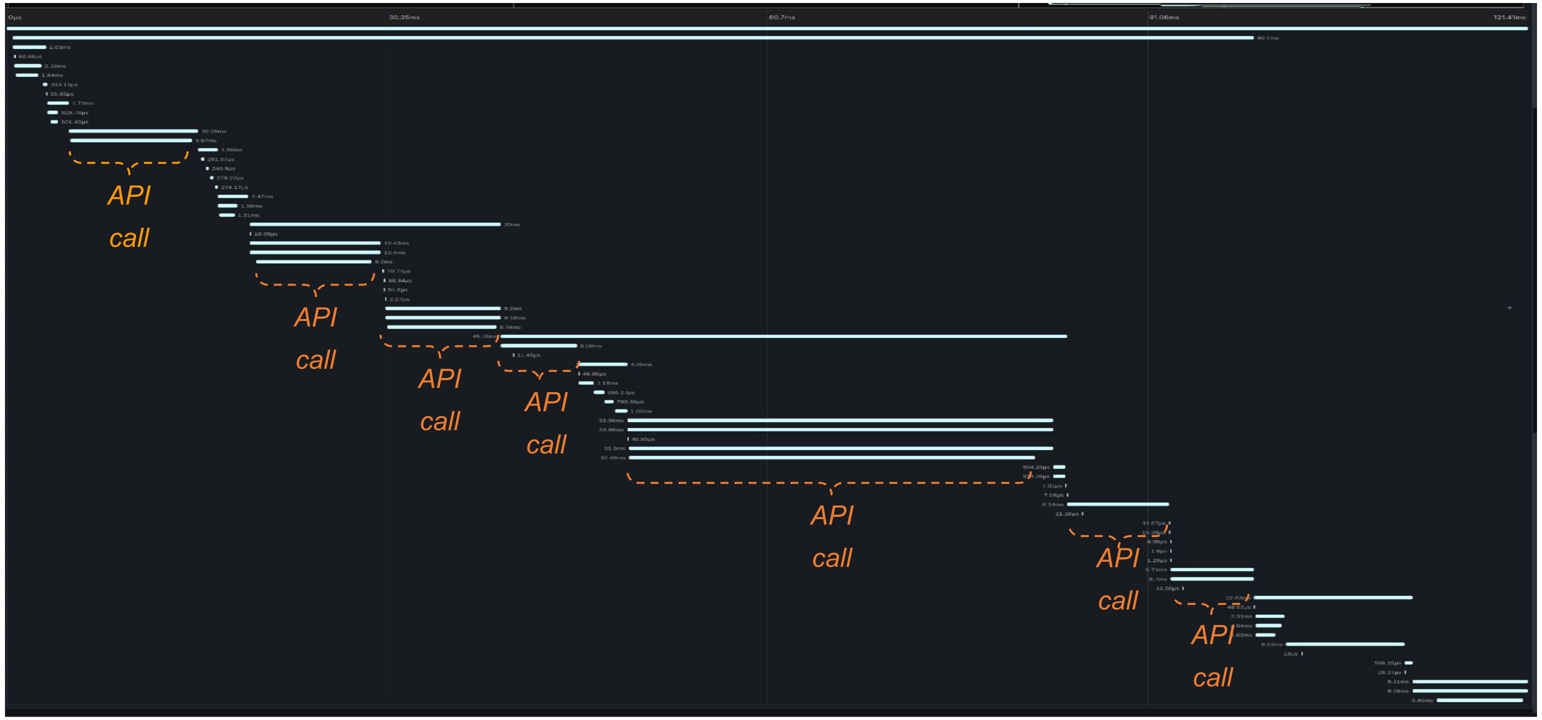
Connect to etcd #4
Generate private key

This issue was addressed by **caching** private keys/clients for each cluster, and **reusing them across many reconciles**. As a result reconcile duration went down by ~75%

API calls

Controllers use clients to read/write data from/to the Kubernetes API server.

As the complexity of your controller grows, **the number of API calls quickly increases, and this can impact performance because of network latency**. This becomes evident as soon as you start looking at traces.

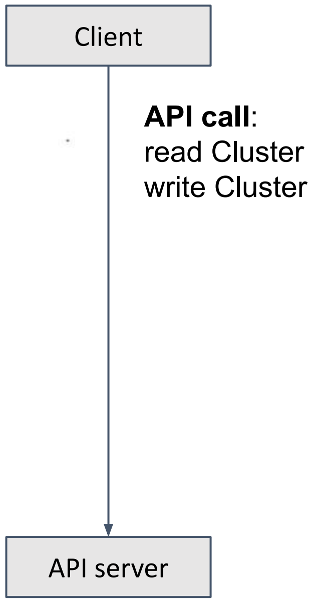


How can we **improve performance and avoid pressure on the API server with many API calls?**

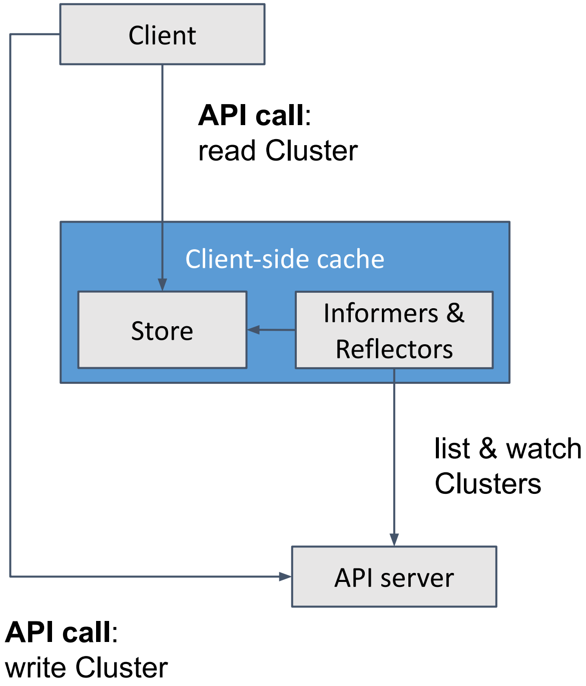
Caching API calls

Kubernetes clients in their simplest form, forward all API calls to the API server.

Controller-runtime's default client is an enhanced client, which includes a cache implemented using client-go primitives.

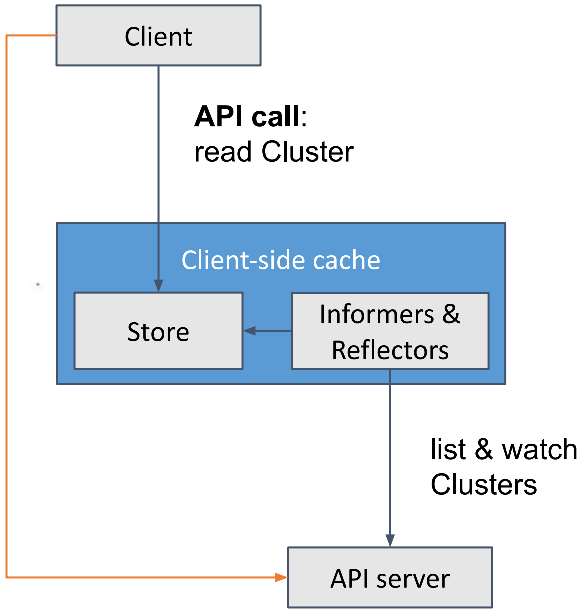


Kubernetes Client (without Cache)



Kubernetes Client with Cache

Caching caveats



API call:
write Cluster
Unstructured read and write (default)

In Cluster API we are using the controller-runtime's default client almost everywhere, so **most of the API calls are already cached** (µs vs ms).

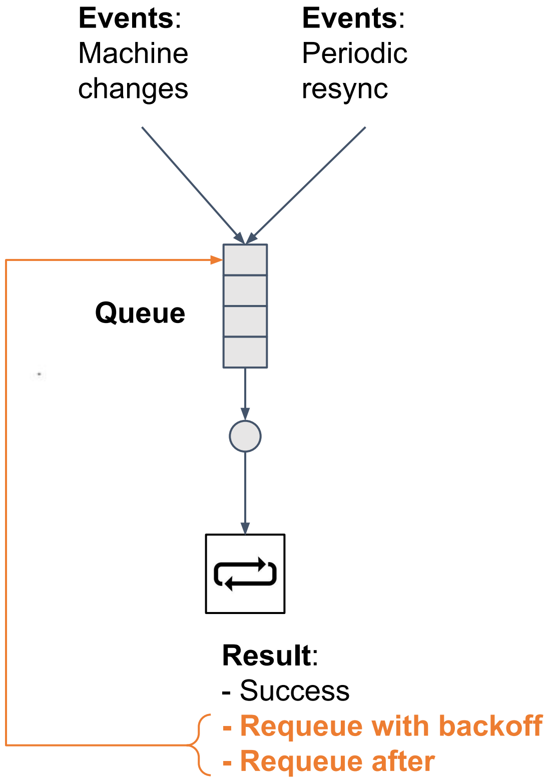
However, not all API calls are cached by default:

- **Write calls are not cached**, so we made sure to have as few write calls as possible.
- **Read calls are not cached by default for Unstructured types**. But it is possible to cache them with a custom cache configuration, which we did.

But caching comes with other caveats too:

- The cache **increases the memory** used by controllers (with 2k clusters, CAPI core controller required 2-4GB depending on the topology).
- You can have **stale reads** from the cache.
  This *usually* is not a problem, because as soon as the new data arrives a new reconcile will be triggered.

Option 3: Reduce # objects in the queue



We can’t control events for objects being created, updated or deleted.

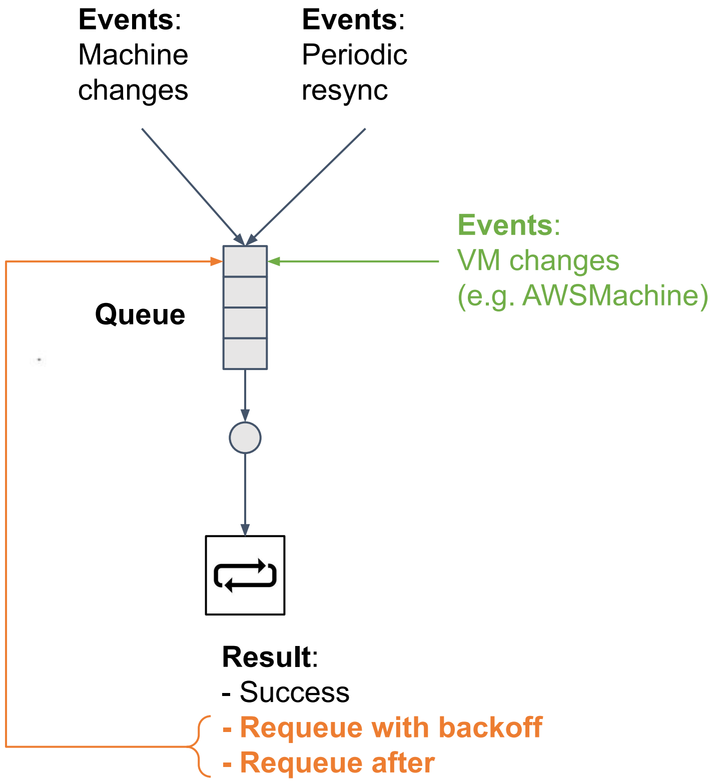
We can’t control periodic resync*.

But we can **control how our controller requeues objects.**

At the end of reconcile we can tell controller-runtime to requeue the current object, if we want to reconcile the object later again.

* Theoretically we could reduce the load by increasing the resync period, but this would be just a workaround as eventually the resync will happen

Requeing from controllers



We use *requeue with backoff* for errors. That's ok.

On the other hand sometimes we use *requeue after* to wait for something to happen in the system, e.g. while reconciling the Machine we requeue to wait for the corresponding VM to be ready.

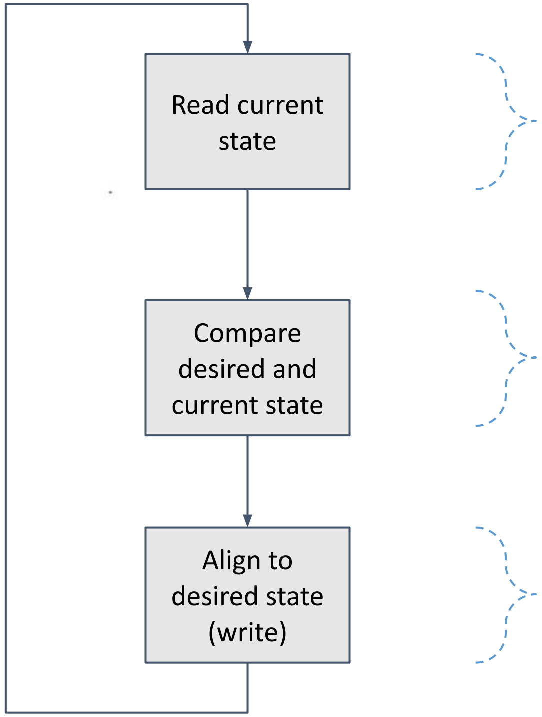
But **reconciling an object every few seconds puts unnecessary load on the system**, and this doesn't help at scale.

Instead we can **use an additional watch** for VM objects, so a reconcile will be only triggered when necessary.

* Theoretically we could reduce the load by increasing the resync period, but this would be just a workaround as eventually the resync will happen

Reconciler best practices

“Ideal”
Reconcile implementation


Read current state
Compare desired and current state
Align to desired state (write)

Read everything at the beginning of the reconcile to avoid duplicate read API calls later

Pay attention to client caching caveats

Avoid duplicate expensive operations
e.g. generating private keys, creating Kubernetes clients, etc.

Write only once per reconcile (in Cluster API we use defer patch)

Avoid reconcile after, use watches instead if possible

Optimization Do’s and Don’ts

Do’s

- Get the right tools for the job
- Define measurable goal(s)
- Invest time in learning how controllers and client-go works
- Iterate fast and with small improvements

Don’ts

- Don’t optimize without data that proves there is a problem and where (don’t drive blind!)

Thank you!

Kudos to:

- Christian Schlotter for driving the work for generating metrics from CRDs
- Lennart Jern for triggering initial discussions about reconciler optimizations
- Killian Muldoon and Yuvaraj Kakaraparthi for helping to develop Mocks & Automation in Cluster API

Please scan the QR Code above to leave feedback on this session

Session QR Codes will be sent via email before the event