KubeCon | CloudNativeCon

Europe 2022

WELCOME TO VALENCIA

KubeCon CloudNativeCon Europe 2022

Kubernetes is your platform: Design Patterns For Extensible Controllers

Rafael Fernández López, Senior Software Engineer, SUSE
Fabrizio Pandini, Staff Engineer, VMWare

Kubernetes is your platform: Design Patterns For Extensible Controllers

"You don't learn to walk
by following rules.

You learn by doing,
and by falling over."

Richard Branson

Rafael Fernández López
Senior Software Engineer
SUSE

Fabrizio Pandini
Staff Engineer
VMWare

The problem statement

The crossing line

Developing Kubernetes extensions is quickly becoming a mainstream practice to solve problems in a cloud native way.

But when the business problem is complex, using a single controller quickly becomes a limit that developers are required to cross.

Design Patterns For Extensible Controllers

This talk will provide you with reusable design patterns derived from the concrete experience and the hard lessons learned on the field.

How to extend capabilities of a single controller

How to orchestrate many controllers

Let’s clarify some terms…

Controller: A single reconcile loop, eg. the ReplicaSet controller.

Manager/Operator: A component that runs one or more controllers, like the kube-controller-manager.

Custom resource: An instance of a Kubernetes type defined by using Custom Resource Definitions. A custom resource is formed by metadata, spec and status.

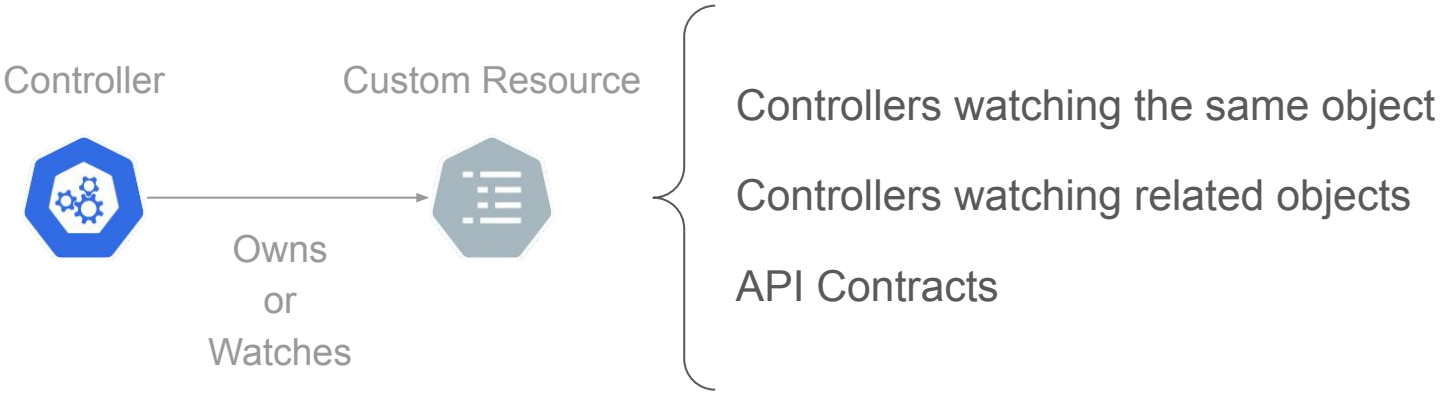
Owner: Controller responsible for driving a custom resource from the observed state to the desired state.

Watch: Action of a controller that looks for resource under your control and other resources as well.

Warming up…

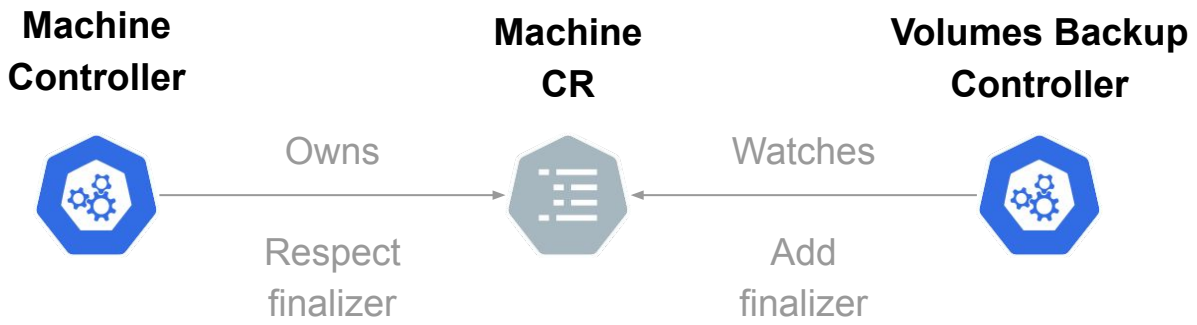
Orchestrate many controllers

This set of patterns allows developers to break down complex problems in a set of tasks assigned to a composable, smaller controllers.



Controllers watching the same object

A controller owns the CR, the second controller dynamically “attach” additional behaviours; eventually controllers can synchronize using finalizers or annotations.



Lesson learned…

Use when

- You want to make progress after a precondition is met:
  e.g. Machine is being deleted.

Pros

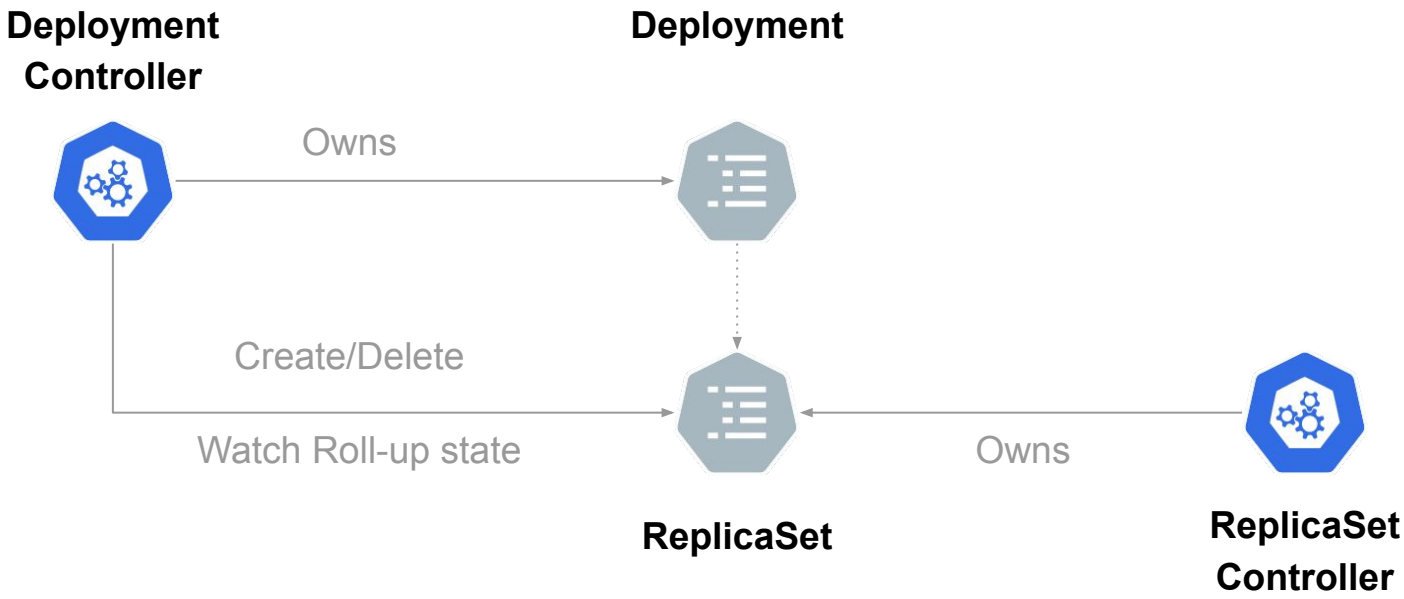
- Responsibility is well defined: Each controller does one task.
- It is a battle tested pattern (e.g garbage collector).

Cons

- Limited orchestration options, e.g. finalizers.
- Behavioral dependencies between controllers are not well documented, they do not surface in the API.

Controllers watching related objects

A controller owns a “parent” CR and creates/deletes “child” CRs; another controller owns the “child” CR.



Lesson learned…

Use when

- There are parent-child semantic relations between the objects you control.

Pros

- Proved effectiveness, it is in Kubernetes core.
- Support sophisticated variants, e.g. history of changes, rollbacks.

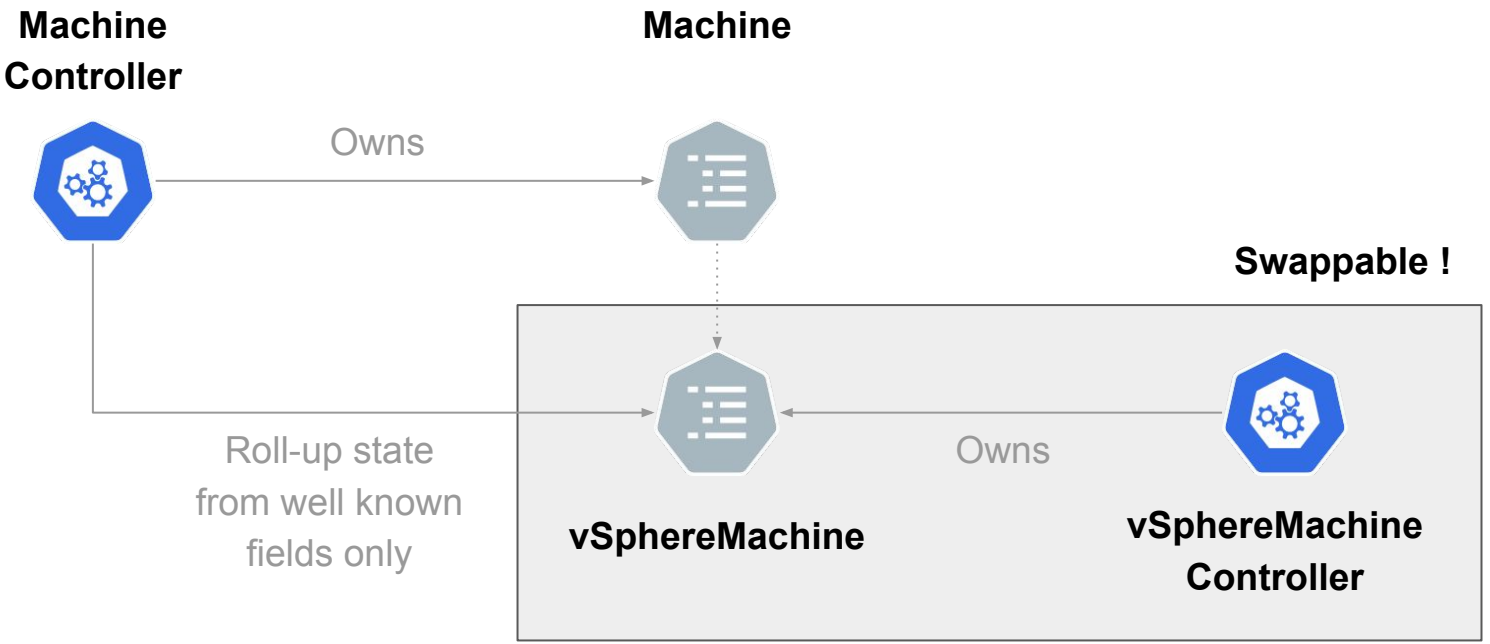
Cons

- Not trivial to debug, why is the resource not progressing?
- Not trivial to evolve, e.g change in a resource might impact more that one controller.

Let’s have some fun now!

API Contracts

It is a variant of the controller watching related objects pattern where the second CR is pluggable and it is expected to have a set of well known fields (the API contract).



Lesson learned…

Use when

- There is a semantic relation between an object and a set of related objects (machine - infrastructure machines).
- It is possible to identify a set of common API fields across all the related object implementations (status.infrastructureReady).

Pros

- It allows swappable implementations!
- Extensively used in Cluster API.

Cons

- It requires all the stakeholders to agree on an API contract.
- Evolving the contract requires coordination across all stakeholders.
- It requires to use unstructured / generic client.

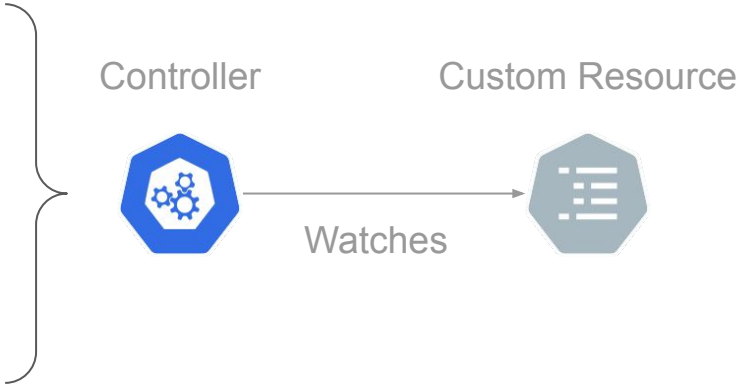
Shifting perspective

Extend Capabilities of a single controller

This set of patterns explore how developers can implement plug-in systems that can bring to a next level the number of use cases a single controller can support.

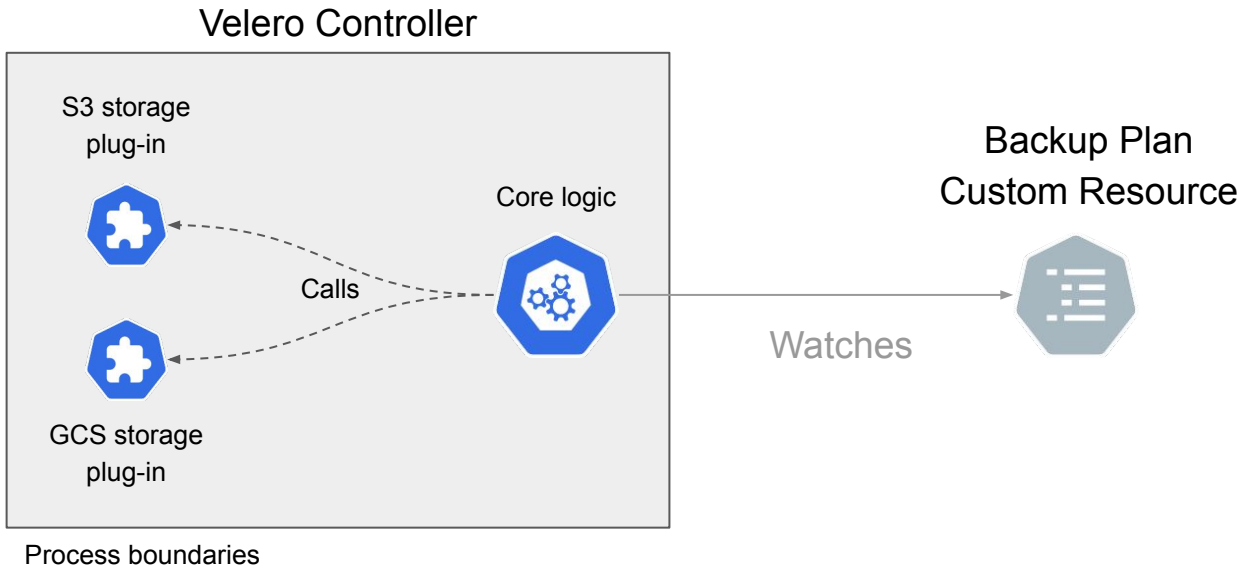
In process extensions

Out of process Extensions



In process extensions

The controller runs plugins into its own process; each plugin extends a controller behaviour.



Lesson learned…

Use when

- You want to make a policy/strategy extensible.
- Executing external binaries does not have security concerns.

Pros

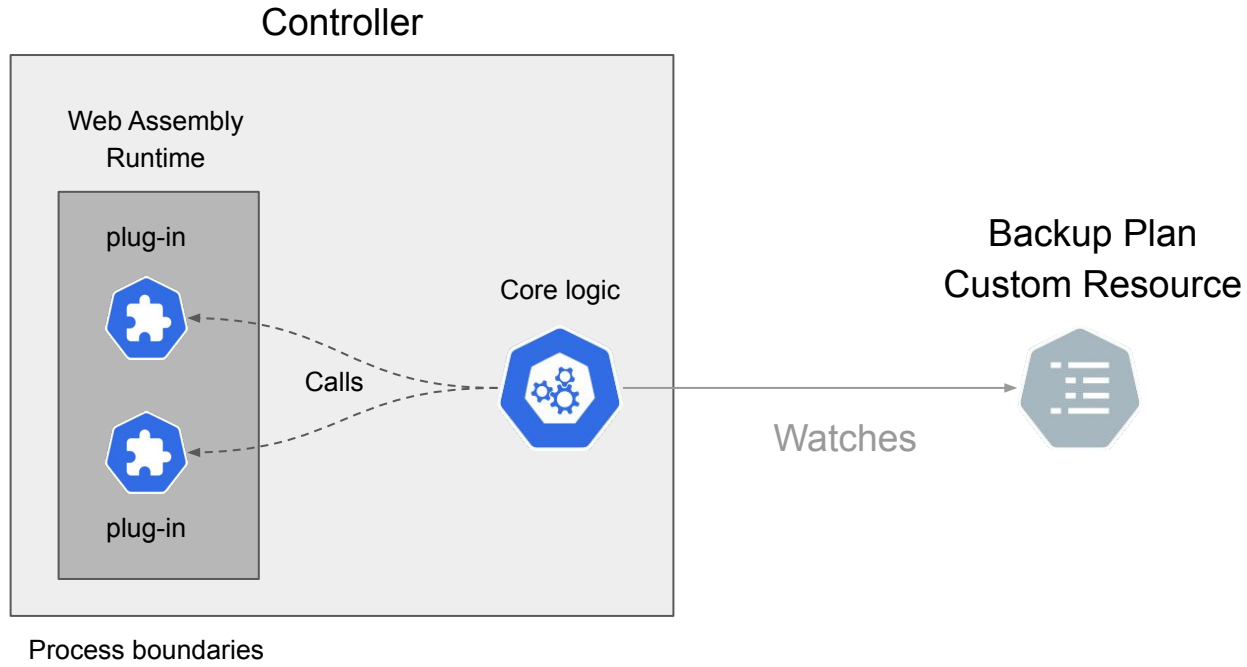
- Similar to the CNI, Velero model.
- Narrow unexpected error space, everything runs in one Pod.

Cons

- Several ways to do binary/discovery registration.
- There should be a contract between the controller and the plugin.
- Hard to prevent side-effects produced by the binary.
- It is required to have binary compatibility between controller and plugin, same architecture, OS compatibility.

In process extensions (WebAssembly variant)

Plugins run in a “WebAssembly runtime” inside the controller process.



Lesson learned…

Use when

- You want to limit plug-in side effects.
- Binary compatibility is an issue.

Pros

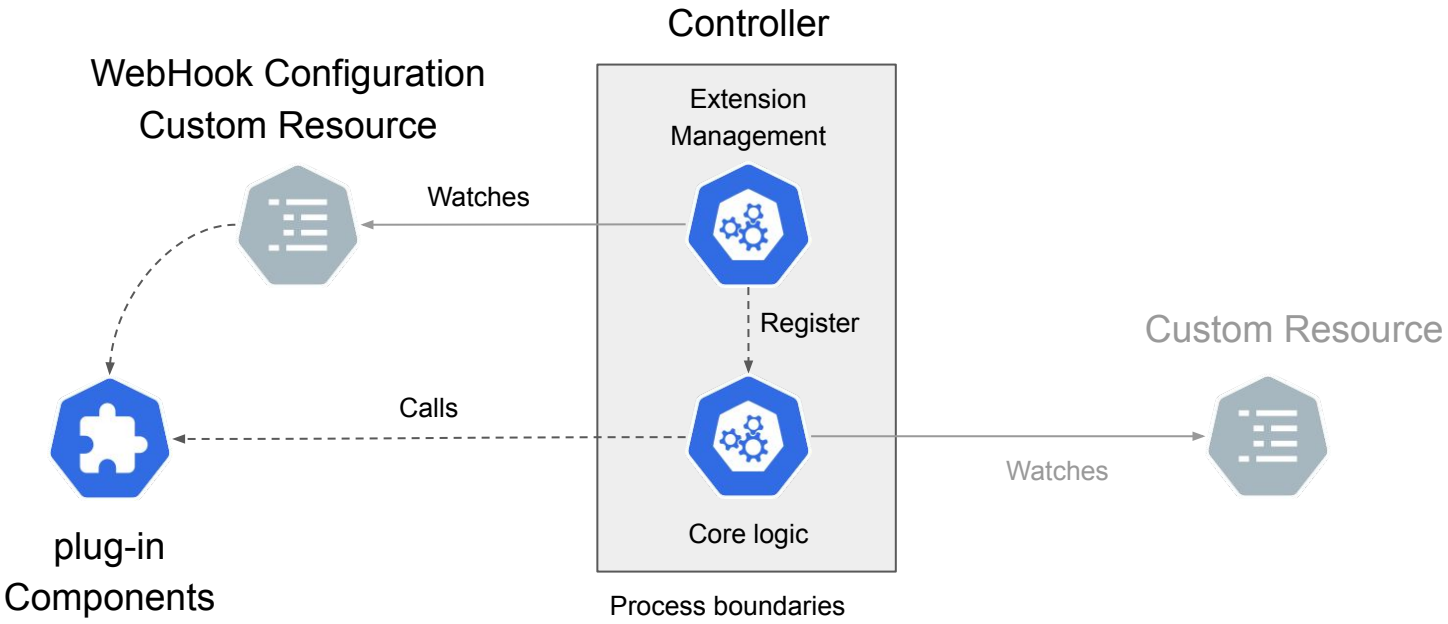
- WebAssembly is a cool technology with growing adoption.

Cons

- We are still at the early WASM (specially WASI) stages.
- Support for WebAssembly isn’t at the same level of maturity in all languages.
- It doesn’t solve the plugin distribution problem.

Out of process extensions

The controller watches for webhook configurations pointing to plugin instances running across the network. The controller calls registered plugins.



Lesson learned…

Use when

- Plug-ins require a dedicated security context.
- It is required to add/remove plugins at runtime.

Pros

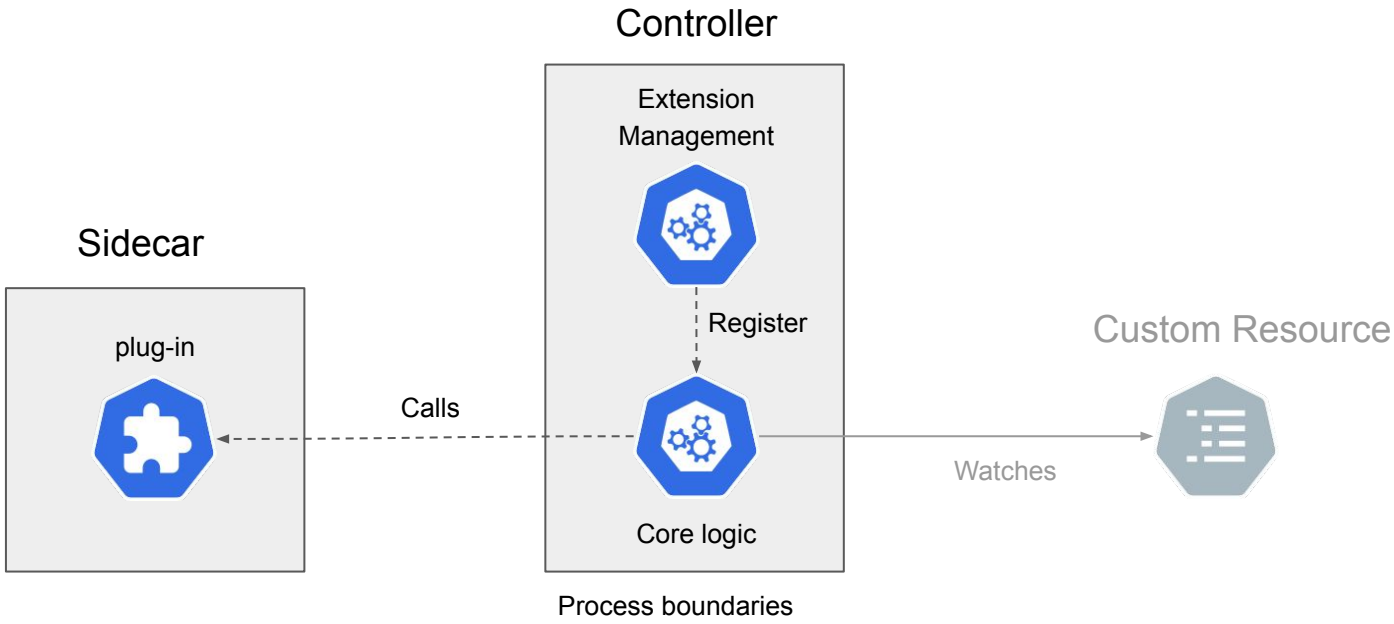
- Plug in side-effects cannot impact the main controller.
- Same maintenance approach for plug in as the rest of the system.

Cons

- The controller has a dependency on external services
  - It is required to manage authentication/authorization, network policies, failure policies, back pressure etc.
- Organizational implications.

Out of process extensions (sidecar variant)

Plugins are deployed as a sidecar of the main controller. This simplifies discovery and networking while keeping a separation between processes.



Lesson learned…

Use when

- You know in advance which plugins you want to use.
- Sharing the same security context is not a concern.

Pros

- Network problems are limited, shared network namespace.

Cons

- Plugin and controller share the same lifecycle.

What's next?

We all have a common goal…

Everyone today is trying to use Kubernetes controllers to solve more complex and challenging use cases.

This is a hard problem and we must solve it as a community.

The problems yet to solve

- How to document behavioral dependencies?
- How to make it easier debug when a resource is not progressing? (e.g. SIG instrumentation work for better distributed tracing)
- How to define contracts for extensibility points (API contracts or Plug in contracts)?
  - How to manage contracts evolving over time ?
- Do we need new tooling and frameworks (e.g Runtime SDK)?
- Binaries vs WebAssembly? Do we need both or can we converge to a “preferred” approach?
- Binary/WebAssembly packaging and distribution: do we need standalone deployments or should we converge on something simpler, like leveraging on OCI standards and image repositories?

Conclusion

"You don't learn to walk by following rules.
You learn by doing, and by falling over."

Richard Branson

Let’s work together to move the art of developing extensible controllers to the next level!

Thank you!

Find us on Twitter or join Kubernetes on Slack

@ereslibre ereslibre

@fabriziopandini fabrizio.pandini