Red Hat

ci-operator

(not an operator)

Bruno Barcarol Guimarães

Red Hat

ci-operator

2022-07-01

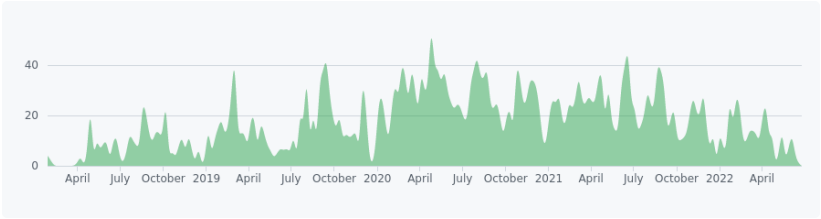
History
- `ci-tools`

Motivation
- Past
- Prow

Architecture
- Overview
- Initialization
- Example
- Etc.

Red Hat

https://github.com/openshift/ci-tools/graphs/contributors



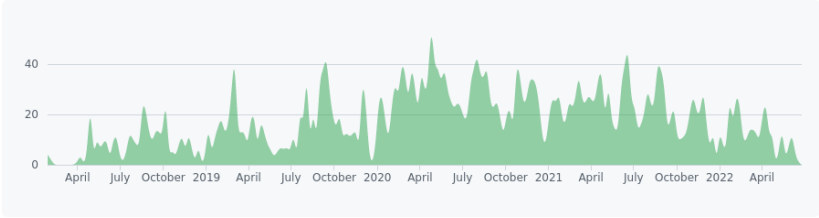

```
$ git log --format='%ad %an %s' --date=format-local:%Y-%m-%d | tail -3
2018-01-31 Steve Kuznetsov Initial implementation of the ci-operator code
2018-01-31 Steve Kuznetsov Initialize vendor directory with Glide
2018-01-31 Steve Kuznetsov Initialize .gitignore
```

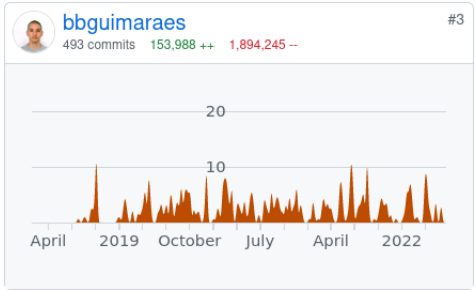

Red Hat

2022-07-01

ci-operator
└History
└ci-tools
└History / ci-tools

Here is the contributor graph for `ci-tools`, going back to 2018 – the small mount at the beginning is 31 Jan. You can see the "initial implementation of [...] `ci-operator`" in the `git` log.
None of the original authors remains in the team (in fact, only one remains in the company), but the code lives on.





2022-07-01

ci-operator
└History
└ci-tools
└History / ci-tools

I am the chief code deleter =) (these numbers make little sense since we include the `vendor` directory in the repository)
I was involved but not directly participating in the beginning, back then we were part of the "CI/CD" team and I worked on both sides of the slash. The first thing I remember working on was the implementation of the `--target` argument, although my version is not the one in the repository (development was chaotic at the time).

Red Hat

```
$ git log --format='%ad %an %s' --date=format-local:%Y-%m-%d --graph 78af6eb7c
* 2019-06-13 Bruno Barcarol Guimarães Merge Makefiles
*   2019-06-13 Bruno Barcarol Guimarães Merge ci-operator-prowgen
|\
…
| * 2018-08-23 Steve Kuznetsov Add an OWNERS file
| * 2018-08-23 Steve Kuznetsov Reorganize code, add Makefile
| * 2018-08-23 Steve Kuznetsov Copy content from openshift/release
*   2019-06-13 Bruno Barcarol Guimarães Merge ci-operator
|\
…
| * 2018-04-06 Clayton Coleman Add .gitignore for binary
| * 2018-02-09 Steve Kuznetsov Refactor StepLink to be functional
| * 2018-02-08 Steve Kuznetsov Add the release tagging step
| * 2018-02-02 Steve Kuznetsov Add a dry-run mode to the entrypoint
| * 2018-01-31 Steve Kuznetsov Initial implementation of the ci-operator code
| * 2018-01-31 Steve Kuznetsov Initialize vendor directory with Glide
| * 2018-01-31 Steve Kuznetsov Initialize .gitignore
* 2019-06-13 Bruno Barcarol Guimarães Initial commit
```


Red Hat

2022-07-01

```
ci-operator
└─History
  └─ci-tools
    └─History / ci-tools
```

If you've ever looked at the repository's history, you might have seen it is a bit strange. There are commits named "merge `Makefiles`", "merge `ci-operator-prowgen`", and "merge `ci-operator`", followed by forks in the history (n.b.: not branches) with their own histories and initial commits, and finally an "initial commit" a year and a half *later*.

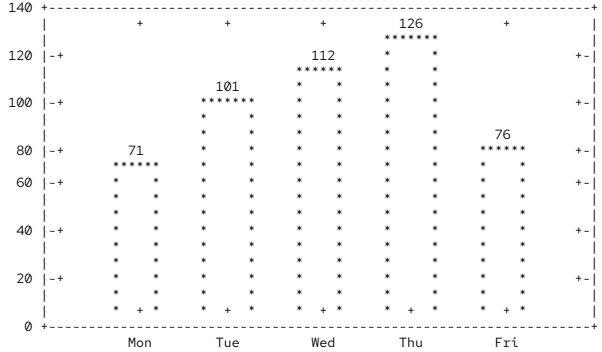
- https://github.com/openshift/ci-tools.git
  - https://github.com/openshift/ci-operator.git
  - https://github.com/openshift/ci-operator-prowgen.git
  - …

2022-07-01

ci-operator
└History
└ci-tools
└History / ci-tools

This is because it started its life as multiple separate repositories that were later joined into what is now `ci-tools`. Its precursors can still be found in Github and are still occasionally of historical significance (pull requests and issues are still there).

```
$ git log --format=%ad --date=format-local:'%a %u' --author 'Bruno Barcarol Guimarães' \
    | sort | uniq -c | sort -nk 3,3 \
    | gnuplot …
```



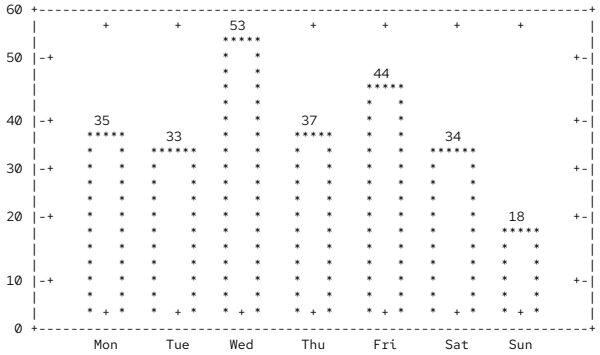
Red Hat

2022-07-01

ci-operator
└History
└ci-tools
└History / ci-tools

Other things you find when excavating through `git` logs: back then, development was *very* chaotic, and Saturday/Sunday pushes were not uncommon.

```
$ git log --format=%ad --date=format-local:'%a %u' --author 'Clayton Coleman' \
    | sort | uniq -c | sort -nk 3,3 \
    | gnuplot …
```



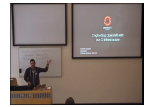
Red Hat

2022-07-01

ci-operator
└History
└ci-tools
└History / ci-tools

Dogfooding Openshift with our CI infrastructure,
Michalis Kargakis (2018-01-27)
`https://www.youtube.com/watch?v=rLLEjodflYw`

Motivation



Dogfooding Openshift with our CI infrastructure,
Michalis Kargakis (2018-01-27)
`https://www.youtube.com/watch?v=rLLEjodflYw`

Continuing with the historical theme, but moving on to motivation, there is a presentation by Michalis which was posted in our chat recently. It is contemporaneous – n.b.: four days before the beginning of the `ci-operator` repository/history – and goes through many of the problems the CI system at the time had.

- Jenkins

They can be summarized mostly with one word.
The CI "system" was a mixture of Ruby (as was OpenShift v2), Python, and Bash scripts, used a very inefficient model on top of that, and was mostly unmaintained and unmaintainable.
Meanwhile, Kubernetes (our upstream project) had `test-infra` and Prow: a CI system which used the platform itself and a language and concepts that were familiar to OpenShift developers.
OpenShift also had many unique features which could be used to extend the upstream system (at the time, many were later contributed upstream, although a unique set still remains).
So we decided to adopt Prow and integrate it into our product.

- Jenkins (R.I.P.)
  - Ruby (OpenShift v2), Python, Bash, ...
  - *slow*
  - unmaintained
  - *unmaintainable*
- Kubernetes
  - `https://github.com/kubernetes/test-infra.git`
  - Prow
- OpenShift
  - `ImageStreams`
  - `Builds`
  - multi-tenancy
  - ...

They can be summarized mostly with one word.
The CI "system" was a mixture of Ruby (as was OpenShift v2), Python, and Bash scripts, used a very inefficient model on top of that, and was mostly unmaintained and unmaintainable.
Meanwhile, Kubernetes (our upstream project) had `test-infra` and Prow: a CI system which used the platform itself and a language and concepts that were familiar to OpenShift developers.
OpenShift also had many unique features which could be used to extend the upstream system (at the time, many were later contributed upstream, although a unique set still remains).
So we decided to adopt Prow and integrate it into our product.

https://github.com/kubernetes/test-infra/blob/master/prow/life_of_a_prow_job.md

```
apiVersion: prow.k8s.io/v1
kind: ProwJob
metadata:
  name: 32456927-35d9-11e7-8d95-0a580a6c1504
spec:
  job: pull-test-infra-bazel
  decorate: true
  pod_spec:
    containers:
    - image: gcr.io/k8s-staging-test-infra/bazelbuild:latest-test-infra
  refs:
    base_ref: master
    base_sha: 064678510782db5b382df478bb374aaa32e577ea
    org: kubernetes
    pulls:
    - author: ixdy
      number: 2716
      sha: dc32ccc9ea3672ccc523b7cbaa8b00360b4183cd
    repo: test-infra
  type: presubmit
```


Red Hat

2022-07-01

ci-operator
└Motivation
└Prow
└Motivation / prow

In Prow, the unit of work is the ProwJob: a native Kubernetes object which links the information from the repository (repository, revision, pull request, etc.) to the work that must be done in the build cluster (in the form of a Pod spec).

https://github.com/openshift/release/blob/master/ci-operator/jobs/openshift/ci-tools/openshift-ci-tools-master-presubmits.yaml

```yaml
presubmits:
  openshift/ci-tools:
  - branches:
    - ^master$
    - ^master-
    cluster: build04
    labels:
      ci.openshift.io/generator: prowgen
      pj-rehearse.openshift.io/can-be-rehearsed: "true"
    name: pull-ci-openshift-ci-tools-master-unit
    spec:
      containers:
      - args:
        - --gcs-upload-secret=/secrets/gcs/service-account.json
        - --image-import-pull-secret=/etc/pull-secret/.dockerconfigjson
        - --report-credentials-file=/etc/report/credentials
        - --target=unit
        command:
        - ci-operator
        image: ci-operator:latest
```


Red Hat

A Prow job has to be registered before it can be executed. We do that using files under `ci-operator/jobs` in the `openshift/release` repository. Note that this is extremely anachronistic, almost nothing shown here existed at the time:

- there was only a single cluster
- there was no `prowgen`
- there was no `rehearse`
- artifact upload was done differently
- images were not imported from other clusters
- there was no results server

However, we do not want every user to have to build their CI pipeline from zero...

https://github.com/openshift/release/blob/master/ci-operator/config/openshift/ci-tools/openshift-ci-tools-master.yaml

```
base_images:
  os:
    name: centos
    namespace: origin
    tag: stream8
binary_build_commands: >
  make production-install
build_root:
  from_repository: true
  use_build_cache: true
images:
- context_dir: >
    images/ci-operator/
  from: os
  inputs:
    bin:
      paths:
      - destination_dir: .
        source_path: >
          /go/bin/ci-operator
  to: ci-operator
```

```
promotion:
  namespace: ci
  tag: latest
test_binary_build_commands: >
  make race-install
tests:
- as: unit
  commands: make test
  container:
    from: src
- as: e2e
  steps:
    test:
    - as: e2e
      commands: make e2e
      from: test-bin
```


Red Hat

2022-07-01

ci-operator
└Motivation
└Prow
└Motivation / prow

The `ci-operator` configuration file is a standard description of the CI pipeline for a given repository. This is the configuration used in `ci-tools`. It simpler because we are not an OpenShift component, but otherwise demonstrates the major concepts:

- input images
- image builds
- unit/E2E tests
- image promotion

- https://docs.ci.openshift.org
  - /docs/architecture/ci-operator
  - /docs/architecture/ci-operator-internals
  - /docs/architecture/ci-operator-internals/steps
- https://github.com/openshift/ci-docs/pulls
  - #233
  - #235
  - #266

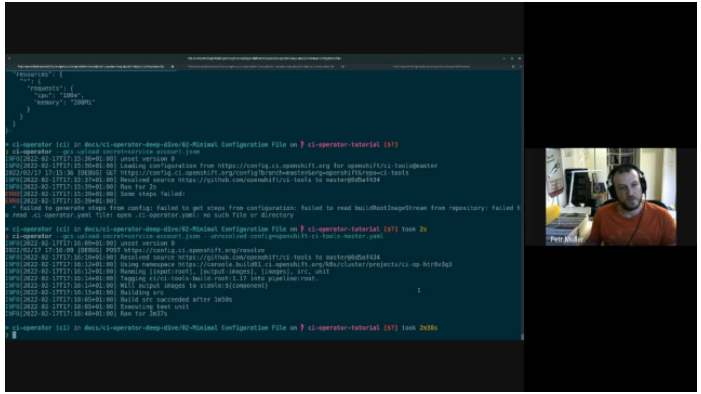
Red Hat

2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview

We have a few architectural descriptions of `ci-operator` and `ci-tools` in general. The `architecture/ci-operator` page is meant for users but, since they are also developers, goes into fairly technical detail. The `...-internals` page is meant for internal use and has abundant references to source code.

Documenting (and understanding in the first place) is an ongoing effort, and there are several work-in-progress documents you can also consult.

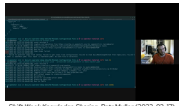


Shift Week Knowledge Sharing, Petr Muller (2022-02-17)
https://drive.google.com/file/d/1ye_Xim2oV4iJaQtBQrDUjre3vwDvZKDT/

Red Hat

2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview



Shift Week Knowledge Sharing, Petr Muller (2022-02-17)
https://drive.google.com/file/d/1ye_Xim2oV4iJaQtBQrDUjre3vwDvZKDT/

I am going to take a different route than I normally would because we've already had a very good presentation by Petr in a previous knowledge sharing session. It was remarkably similar to what I had in mind for a `ci-operator` introduction, and formed part of the basis for the development of `https://github.com/openshift/ci-docs/pull/235`.

Instead, here we're going the opposite way, with an abstract architecture overview.

- Inputs
  - repository / `git` revision(s)
  - command-line arguments
  - configuration
- Outputs
  - test results
  - images

Red Hat

2022-07-01

`ci-operator`
└Architecture
└Overview
└Architecture / overview

Conceptually, the work done by `ci-operator` is relatively simple: it receives a list of source code references, command-line arguments, and a configuration file as input and produces test results and container images as output. The devil is, as always, in the details.

- Inputs
  - repository / `git` revision(s)
  - command-line arguments
  - configuration
- Outputs
  - test results
  - images
- Implementation
  - build cluster/node
  - temporary namespace
  - image pipeline
  - test types
  - cloud providers
  - remote storage
  - image promotion

Red Hat

2022-07-01

`ci-operator`
└Architecture
└Overview
└Architecture / overview

Conceptually, the work done by `ci-operator` is relatively simple: it receives a list of source code references, command-line arguments, and a configuration file as input and produces test results and container images as output. The devil is, as always, in the details.

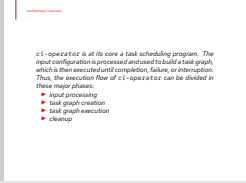
`ci-operator` is at its core a task scheduling program. The input configuration is processed and used to build a task graph, which is then executed until completion, failure, or interruption. Thus, the execution flow of `ci-operator` can be divided in these major phases:

- *input processing*
- *task graph creation*
- *task graph execution*
- *cleanup*

Red Hat

2022-07-01

`ci-operator`
└Architecture
└Overview
└Architecture / overview



This introduction from the "internals" documentation describes exactly what is needed to understand how `ci-operator` works. The main data structure is the *step graph*.

```
$ ci-operator \
    --unresolved-config ci-tools-master.yaml \
    --print-graph \
    --target unit --target e2e
…
src [input:root]
test-bin src
unit src
e2e test-bin
…

https://pkg.go.dev/golang.org/x/tools/cmd/digraph
```


Red Hat

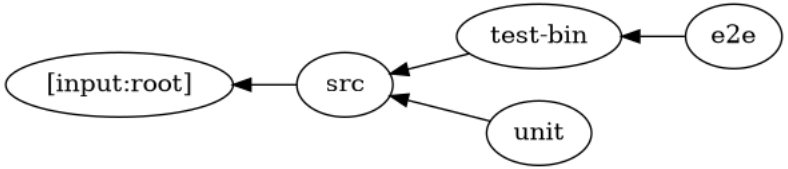
2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview

There is an obscure flag which can be used to display the graph. The output is meant to be used with the (equaly obscure) `digraph` Golang tool. Each line contains a step in the first column and its dependencies as subsequent columns.

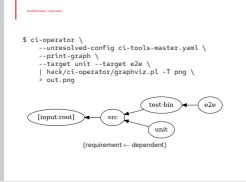
Note: some of the functionality described here (e.g. displaying a sub-graph with `--target`) is not yet in `master`.

```
$ ci-operator \
    --unresolved-config ci-tools-master.yaml \
    --print-graph \
    --target unit --target e2e \
    | hack/ci-operator/graphviz.pl -T png \
    > out.png
```

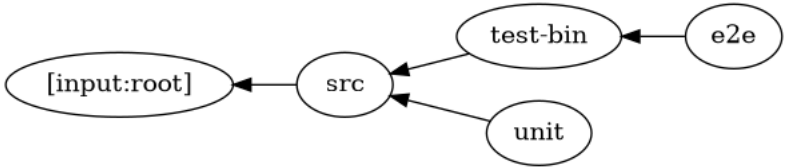


(requirement ← dependent)

Red Hat

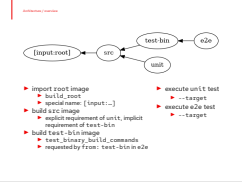


With a bit of magic (a.k.a. Perl), it can be translated to the DOT language and visualized.

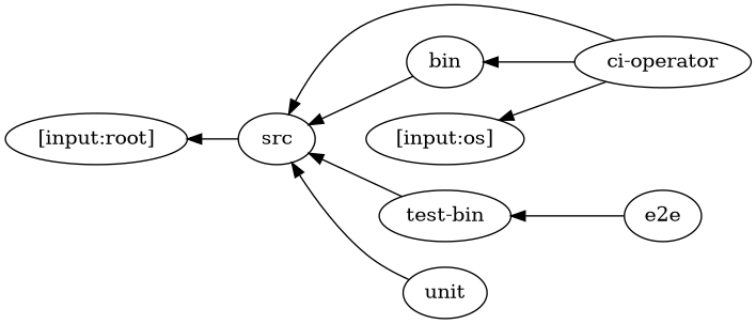


- import `root` image
  - `build_root`
  - special name: `[input:…]`
- build `src` image
  - explicit requirement of `unit`, implicit requirement of `test-bin`
- build `test-bin` image
  - `test_binary_build_commands`
  - requested by `from: test-bin` in `e2e`
- execute `unit` test
  - `--target`
- execute e2e test
  - `--target`

Red Hat



… --target unit --target e2e --target ci-operator …

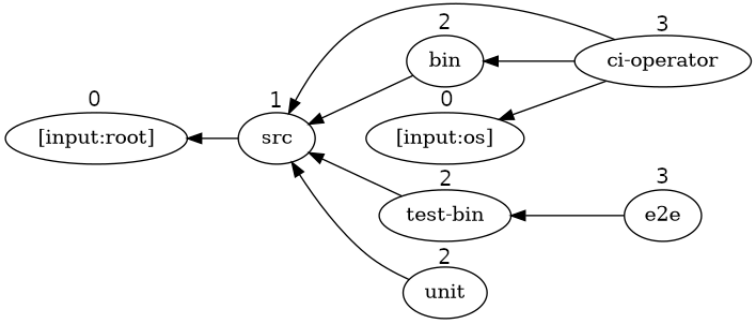


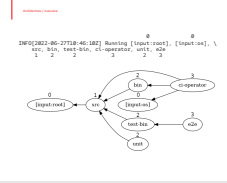
Red Hat

If we add the `ci-operator` target to the graph (an image build), we can see the effect in the resulting graph:

- the previous graph is unchanged at the bottom
- the `ci-operator` node is added, unsurprisingly
- it uses an input image (`base_images`) as a base
- it also uses the `bin` image as a base, which in turn is built from the `src` image

```
                                        0           0
INFO[2022-06-27T10:46:10Z] Running [input:root], [input:os], \
    src, bin, test-bin, ci-operator, unit, e2e
     1    2      2           3         2    3
```





This graph is what the line in the `ci-operator` output which contains "Running" followed by a sequence of names represents: it is a topological order of the graph, i.e. a one-dimensional projection of the sequence(s) of steps to be performed.

As described previously, these are all executed in parallel as much as possible, with dependent steps starting as soon as their requirements are satisfied. This means this linearized version is only one of the possible total orders, as the dependency between steps is only a partial order.

This visualization assigns a number to each node according to its "maximum depth" starting from the roots, giving an idea of which steps can be executed in parallel.

`https://docs.ci.openshift.org/docs/architecture/ci-operator-internals/steps/#step-types`

- build steps
- release steps
- (optional) operator steps
- auxiliary steps
- test steps
- output steps

Red Hat

2022-07-01

`ci-operator`
└Architecture
└Overview
└Architecture / overview

Mastering `ci-operator` is in some sense learning the purpose and implementation of each type of step. The "internals" page has sections for each category, with individual sub-sections for each and every step type.

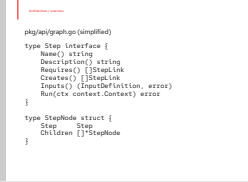
pkg/api/graph.go (simplified)

```
type Step interface {
    Name() string
    Description() string
    Requires() []StepLink
    Creates() []StepLink
    Inputs() (InputDefinition, error)
    Run(ctx context.Context) error
}

type StepNode struct {
    Step     Step
    Children []*StepNode
}
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview



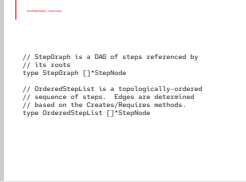
You can imagine what the execution code look like from that description, but here is a simplified version of it. This is the step interface and the graph structure.

```
// StepGraph is a DAG of steps referenced by
// its roots
type StepGraph []*StepNode

// OrderedStepList is a topologically-ordered
// sequence of steps.  Edges are determined
// based on the Creates/Requires methods.
type OrderedStepList []*StepNode
```


Red Hat

2022-07-01

ci-operator
└─Architecture
└─Overview
└─Architecture / overview



We have many types in `pkg/api/graph` which are all just slices of `StepNode` (`StepGraph`, `OrderedStepList`, etc.). Each is a separate type for semantic reasons.

```
// TopologicalSort validates nodes form a DAG and orders them
// topologically.
func (g StepGraph) TopologicalSort() (OrderedStepList, []error) {
    …
    var waiting []*StepNode
    …
}
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview

Here is a nice example with multiple types which are all the same underlying structure.

pkg/steps/run.go (simplified)

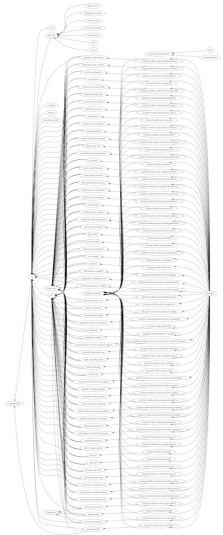
```
func Run(graph api.StepGraph) {
    var seen []api.StepLink
    results := make(chan *api.StepNode)
    for _, root := range graph {
        go runStep(root, results)
    }
    for out := range results {
        seen = append(seen, out.Step.Creates()...)
        for _, child := range out.node.Children {
            if api.HasAllLinks(child.Step.Requires(), seen) {
                go runStep(child, results)
            }
        }
    }
}
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Overview
└Architecture / overview

And finally, the code which executes all steps in the graph.



Red Hat

2022-07-01

`ci-operator`
└Architecture
└Overview
└Architecture / overview

Of course, reality is *much* more complicated… (this is the actual graph for `ci-tools`, which is still a long way from being as complicated as it can be)

cmd/ci-operator/main.go

```
steps, postSteps, err := defaults.FromConfig(
    ctx, o.configSpec, &o.graphConfig, o.jobSpec,
    o.templates, o.writeParams, o.promote, o.clusterConfig,
    leaseClient, o.targets.values, o.cloneAuthConfig,
    o.pullSecret, o.pushSecret, o.censor, o.hiveKubeconfig,
    o.consoleHost, o.nodeName)
```


Red Hat

2022-07-01

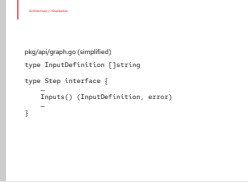
ci-operator
└Architecture
└Initialization
└Architecture / initialization

Moving on to what actually happens when `ci-operator` is executed, we start with this gigantic function call to turn all input values into a list of steps, still unordered.

pkg/api/graph.go (simplified)

```
type InputDefinition []string

type Step interface {
    …
    Inputs() (InputDefinition, error)
    …
}
```


Red Hat



Recall that each step has an `Inputs` method, which returns a list of strings based on the values it depends on – input image tags, source code revision, etc.

cmd/ci-operator/main.go (simplified)

```
func (o *options) resolveInputs(steps []api.Step) {
    var inputs api.InputDefinition
    for _, step := range steps {
        inputs = append(inputs, step.Inputs()...)
    }
    inputs = append(inputs, string(o.configSpec))
    inputs = append(inputs, o.extraInputHash.values...)
    stat := os.Stat(exec.LookPath(os.Args[0]))
    inputs = append(inputs, fmt.Sprintf(
        "%d-%d", stat.ModTime().UTC().Unix(), stat.Size()))
    sort.Strings(inputs)
    o.inputHash = inputHash(inputs)
    if len(o.namespace) == 0 {
        o.namespace = "ci-op-{id}"
    }
    o.namespace = strings.Replace(
        o.namespace, "{id}", o.inputHash, -1)
}
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Initialization
└Architecture / initialization

These values, along with those derived from the inputs such as the version and configuration file of `ci-operator`, are combined and hashed, generating a final string which is unique for a given set of inputs.
This string is then used as a namespace, guaranteeing that:

- unrelated jobs are isolated from each other
- related jobs share resources as much as possible

cmd/ci-operator/main.go (simplified)

```
var encoding = base32
    .NewEncoding("bcdfghijklmnpqrstvwxyz0123456789")
    .WithPadding(base32.NoPadding)

func inputHash(inputs api.InputDefinition) string {
    hash := sha256.New()
    for _, s := range inputs {
        hash.Write([]byte(s))
    }
    return encoding.EncodeToString(hash.Sum(nil)[:5])
}
```

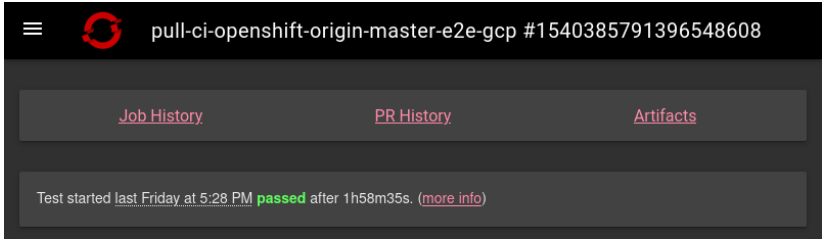

Here is the (very simple) hash calculation.

cmd/ci-operator/main.go (*very* simplified)

```
steps, postSteps := defaults.FromConfig(…)
o.resolveInputs(steps)
nodes := api.BuildPartialGraph(steps, o.targets.values)
stepList := nodes.TopologicalSort()
logrus.Infof(
    "Running %s",
    strings.Join(nodeNames(stepList), ", "))
o.initializeNamespace()
steps.Run(ctx, nodes)
for _, step := range postSteps {
    runStep(ctx, step)
}
```


ci-operator
└Architecture
└Initialization
└Architecture / initialization

Next, the graph is built by creating edges based on the dependency relations between steps (e.g. a test requires its container image to be imported or built). Then, the test namespace is initialized and, finally, the steps are executed. So here is our final approximation of a full `ci-operator` execution.



https://prow.ci.openshift.org/view/gs/origin-ci-test/pr-logs/pull/27275/pull-ci-openshift-origin-master-e2e-gcp/1540385791396548608

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

From here, it is instructive to examine a typical E2E test, which differs quite a bit from the simple container tests we have considered so far. We will look at this test in `openshift/origin` which creates an ephemeral GCP cluster.

https://github.com/openshift/release/blob/master/ci-operator/config/openshift/origin/openshift-origin-master.yaml

```
tests:
- as: e2e-gcp
  steps:
    cluster_profile: gcp-openshift-gce-devel-ci-2
    workflow: openshift-e2e-gcp-loki
```


Red Hat

2022-07-01

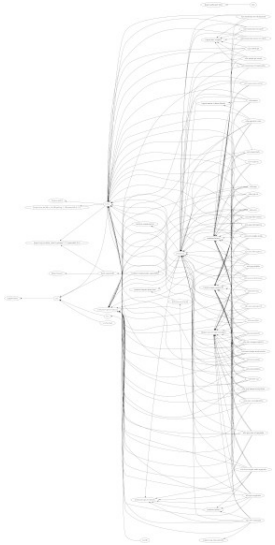
ci-operator
└Architecture
└Example
└Architecture / example

This job is generated from this innocent-looking entry in the `ci-operator` configuration file.

N.b.: take it from someone who has seen all incarnations of Prow job definitions we have had, the fact that this test is generated from these four lines is a *miracle*.

```
INFO[2022-06-24T17:28:45Z] ci-operator version v20220621-3b245f722
INFO[2022-06-24T17:28:45Z] Loading configuration from https://config.ci.openshift.org for openshift/…
INFO[2022-06-24T17:28:46Z] Resolved source https://github.com/openshift/origin to master@a946e2b9, m…
INFO[2022-06-24T17:28:46Z] Building release previous from a snapshot of ocp/4.10
INFO[2022-06-24T17:28:46Z] Building release initial from a snapshot of ocp/4.11
INFO[2022-06-24T17:28:46Z] Building release latest from a snapshot of ocp/4.11
INFO[2022-06-24T17:28:47Z] Using namespace https://console.build02.ci.openshift.org/k8s/cluster/proj…
INFO[2022-06-24T17:28:47Z] Running [input:root], [input:ocp_builder_rhel-8-golang-1.15-openshift-4.8…
INFO[2022-06-24T17:28:47Z] Tagging ocp/builder:rhel-8-golang-1.15-openshift-4.8 into pipeline:ocp_bu…
INFO[2022-06-24T17:28:47Z] Tagging ocp/builder:rhel-8-golang-1.18-openshift-4.11 into pipeline:ocp_b…
…
INFO[2022-06-24T17:28:48Z] Building src
INFO[2022-06-24T17:32:03Z] Build src succeeded after 3m57s
INFO[2022-06-24T17:32:03Z] Building hello-openshift
INFO[2022-06-24T17:32:03Z] Building tests
INFO[2022-06-24T17:35:23Z] Build hello-openshift succeeded after 3m20s
INFO[2022-06-24T17:35:23Z] Tagging hello-openshift into stable
INFO[2022-06-24T17:42:13Z] Build tests succeeded after 6m48s
INFO[2022-06-24T17:42:13Z] Tagging tests into stable
INFO[2022-06-24T17:42:14Z] Creating release image registry.build02.ci.openshift.org/ci-op-8yq06grj/r…
INFO[2022-06-24T17:43:34Z] Snapshot integration stream into release 4.11.0-0.ci.test-2022-06-24-1742…
INFO[2022-06-24T17:43:34Z] Acquiring leases for test e2e-gcp: [gcp-openshift-gce-devel-ci-2-quota-sl…
INFO[2022-06-24T17:43:34Z] Acquired 1 lease(s) for gcp-openshift-gce-devel-ci-2-quota-slice: [us-cen…
INFO[2022-06-24T17:43:34Z] Running multi-stage test e2e-gcp
INFO[2022-06-24T17:43:34Z] Running multi-stage phase pre
INFO[2022-06-24T17:43:34Z] Running step e2e-gcp-ipi-install-hosted-loki.
…
INFO[2022-06-24T19:27:16Z] Releasing leases for test e2e-gcp
INFO[2022-06-24T19:27:17Z] Ran for 1h58m30s
INFO[2022-06-24T19:27:17Z] Reporting job state 'succeeded'
```


Here is what the output looks like.

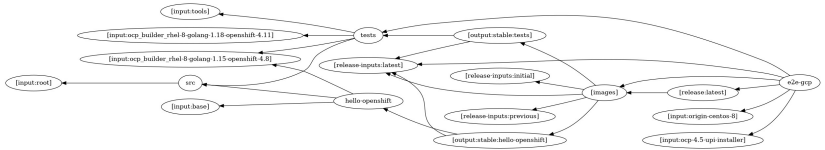


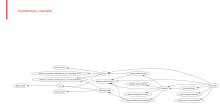
Red Hat

ci-operator
└Architecture
└Example
└Architecture / example

2022-07-01

(a reminder that this is what we are dealing with)





Thankfully, for this particular test, we ”only” have to deal with this sub-graph. Let’s examine each part.

```
ci-operator version v20220621-3b245f722
Loading configuration \
    from https://config.ci.openshift.org \
    for openshift/origin@master
Resolved source https://github.com/openshift/origin \
    to master@a946e2b9, merging: \
    #27275 457391d6 @DennisPeriquet
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

ci-operator version v20220621-3b245f722
Loading configuration \
from https://config.ci.openshift.org \
for openshift/origin@master
Resolved source https://github.com/openshift/origin \
to master@a946e2b9, merging: \
#27275 457391d6 @DennisPeriquet

We start with the part you are already familiar with.

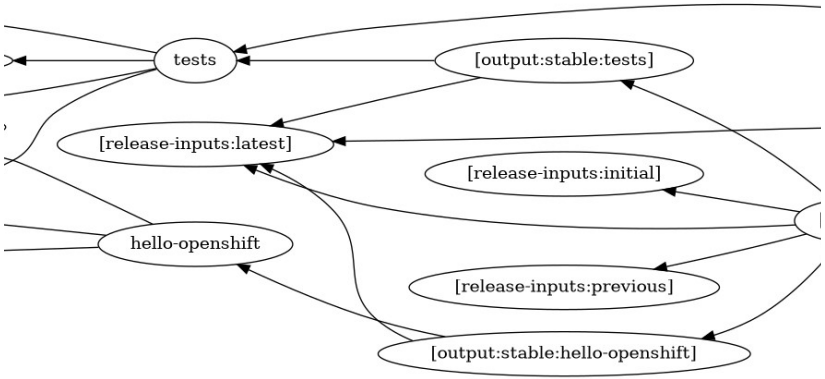
```
Building release previous from a snapshot of ocp/4.10
Building release initial from a snapshot of ocp/4.11
Building release latest from a snapshot of ocp/4.11


releases:
  initial:
    integration:
      name: "4.11"
      namespace: ocp
  latest:
    integration:
      include_built_images: true
      name: "4.11"
      namespace: ocp
  previous:
    integration:
      name: "4.10"
      namespace: ocp
```


Red Hat

2022-07-01

```
ci-operator
└─Architecture
  └─Example
    └─Architecture / example
```

Release images are also root nodes, so they are imported immediately. These are all integration streams, which means they come from an `ImageStream` and can simply be copied into the test namespace (this is what ”snapshot” refers to).



Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

These correspond to the `[release-inputs:...]` steps here.

```
Using namespace https://console.build02.ci.openshift.org/\
    k8s/cluster/projects/ci-op-8yq06grj
Running [input:root], \
    [input:ocp_builder_rhel-8-golang-1.15-openshift-4.8], \
    [input:ocp_builder_rhel-8-golang-1.18-openshift-4.11], \
    [input:tools], [input:base], [input:origin-centos-8], \
    [input:ocp-4.5-upi-installer], [release-inputs:previous], \
    [release-inputs:initial], [release-inputs:latest], \
    src, tests, hello-openshift, \
    [output:stable:tests], [output:stable:hello-openshift], \
    [images], [release:latest], e2e-gcp
```


Red Hat

2022-07-01

ci-operator
└─Architecture
└─Example
└─Architecture / example

Here we get assigned a temporary namespace and print the execution graph. Note the same left-to-right order of dependencies, with the actual test at the far right of the list.

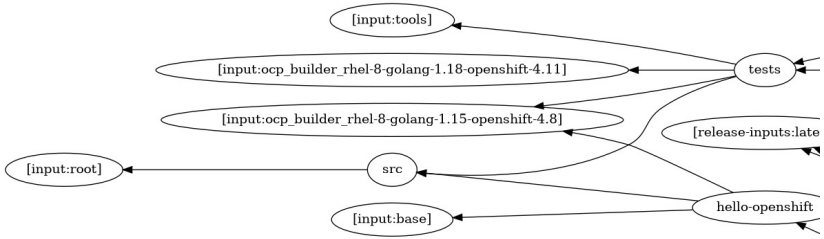
```
Tagging ocp/builder:rhel-8-golang-1.15-openshift-4.8 into \
    pipeline:ocp_builder_rhel-8-golang-1.15-openshift-4.8.
Tagging ocp/builder:rhel-8-golang-1.18-openshift-4.11 into \
    pipeline:ocp_builder_rhel-8-golang-1.18-openshift-4.11.
Tagging openshift/release:rhel-8-release-golang-1.18-openshift-4.11 \
    into pipeline:root.
Tagging origin/centos:8 into pipeline:origin-centos-8.
Tagging ocp/4.11:base into pipeline:base.
Tagging ocp/4.5:upi-installer into pipeline:ocp-4.5-upi-installer.
Tagging ocp/4.11:tools into pipeline:tools.
```

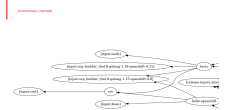

Red Hat

2022-07-01

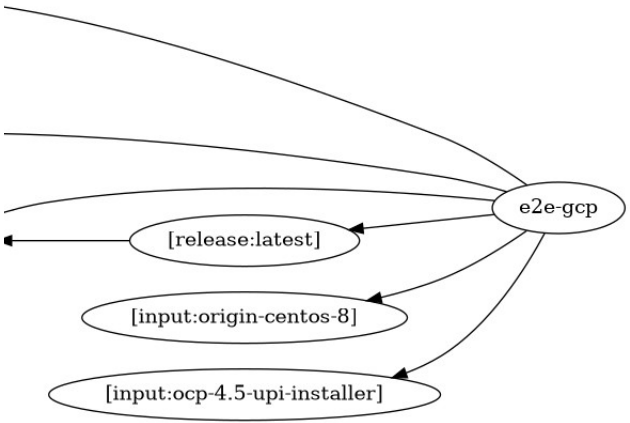
ci-operator
└Architecture
└Example
└Architecture / example

The roots of the graph are usually input images, since both the build root and base images are depended on for image builds and tests.





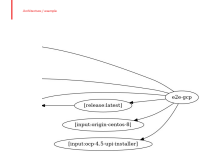
We see them at the far left side of the graph: they are all in the format `[input:…]`. Note the dependent image builds.



Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example



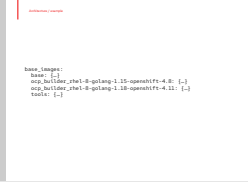
And again at the far right for input images depended on by the test (more on that later). These are all independent, so they are imported in parallel.

```
base_images:
  base: {…}
  ocp_builder_rhel-8-golang-1.15-openshift-4.8: {…}
  ocp_builder_rhel-8-golang-1.18-openshift-4.11: {…}
  tools: {…}
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example



These images are imported directly using `base_images`.

```
build_root:
  from_repository: true
```

https://github.com/openshift/origin/blob/master/.ci-operator.yaml

```
build_root_image:
  name: release
  namespace: openshift
  tag: rhel-8-release-golang-1.18-openshift-4.11
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

The `build_root` image comes from the repository (you can see how these things start to get difficult to track).

As an aside, when `from_repository` is used, we have to make sure the Prow job (which executes `ci-operator`) is configured with `decorate` and does not have `skip_cloning`. `ci-operator` will then have access to the the repository code, where the `.ci-operator.yaml` file resides.

```
https://steps.ci.openshift.org/workflow/openshift-e2e-gcp-loki

workflow:
  as: openshift-e2e-gcp-loki
  steps:
    allow_best_effort_post_steps: true
    pre:
    - ref: ipi-install-hosted-loki
    - chain: ipi-gcp-pre
    test:
    - ref: openshift-e2e-test
    post:
    - chain: ipi-gcp-post
  documentation: |-
    The Openshift E2E GCP workflow executes the common
    end-to-end test suite on GCP with a default cluster
    configuration with loki as log collector.
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

Finding the source of the other images requires us to start looking into multi-stage tests. The `openshift-e2e-gcp-loki` workflow declared in the test can be found in the step registry. It includes a chain called `ipi-gcp-pre`...

```
https://steps.ci.openshift.org/chain/ipi-gcp-pre

chain:
  as: ipi-gcp-pre
  steps:
  - chain: ipi-conf-gcp
  - chain: ipi-install
  documentation: |-
    The IPI setup step contains all steps that
    provision an OpenShift cluster with a default
    configuration on GCP.
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

which includes a chain called `ipi-conf-gcp`...

https://steps.ci.openshift.org/chain/ipi-conf-gcp

```
chain:
  as: ipi-conf-gcp
  steps:
  - ref: ipi-conf
  - ref: ipi-conf-gcp
  - ref: ipi-install-monitoringpvc
  documentation: >-
    This chain generates an install-config.yaml
    file configured to run clusters in the GCP CI
    project.
    The GCP specific configs are added to
    the file generated by the ipi-conf steps.
    This resulting file is stored in the shared
    directory for future consumption.
```


2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

which includes steps called `ipi-conf` and `ipi-conf-gcp`...

```
https://steps.ci.openshift.org/reference/ipi-conf
https://steps.ci.openshift.org/reference/ipi-conf-gcp
```

```
ref:
  as: ipi-conf
  from_image:
   namespace: origin
   name: centos
   tag: '8'
```

(from_image ≈ base_images + from)

Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

and here we finally find our `centos:8` image. `from_image` entries for all steps are collected and added to the input images in `base_images`.

https://steps.ci.openshift.org/reference/gather-gcp-console

```
ref:
  as: gather-gcp-console
  optional_on_success: true
  from_image:
    namespace: ocp
    name: "4.5"
    tag: upi-installer
    …
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

Similarly, `upi-installer` can be found in the `gather-gcp-console` step.

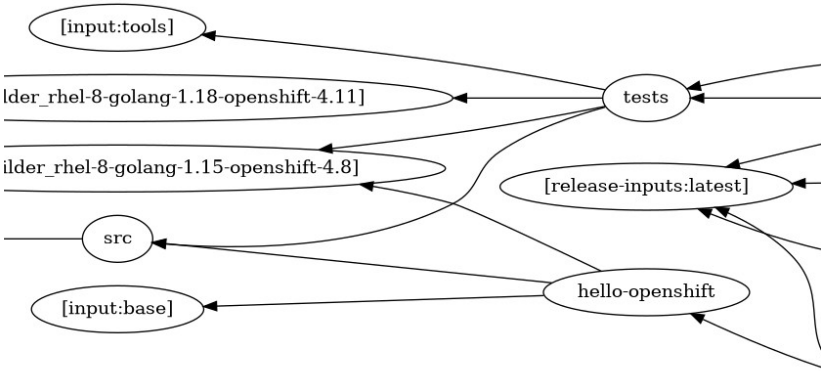
```
Building src
Build src succeeded after 3m57s
Building hello-openshift
Building tests
Build hello-openshift succeeded after 3m20s
Tagging hello-openshift into stable
Build tests succeeded after 6m48s
Tagging tests into stable
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

After the required images are imported, image builds can be started.



Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

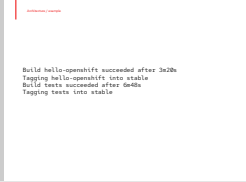
`src` is built first, since it is a requirement for the other two, which are then built in parallel.

```
Build hello-openshift succeeded after 3m20s
Tagging hello-openshift into stable
Build tests succeeded after 6m48s
Tagging tests into stable
```

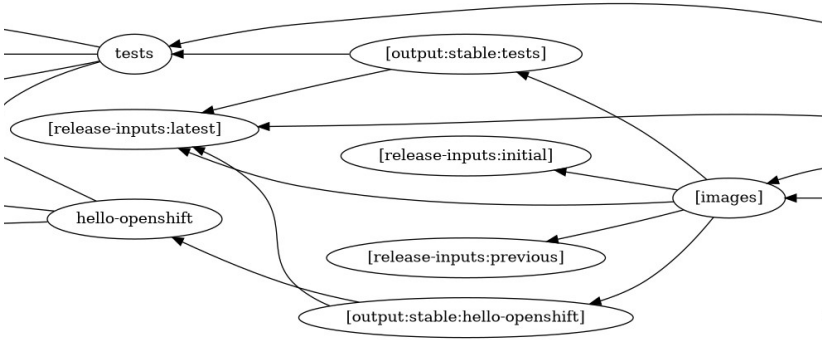

Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

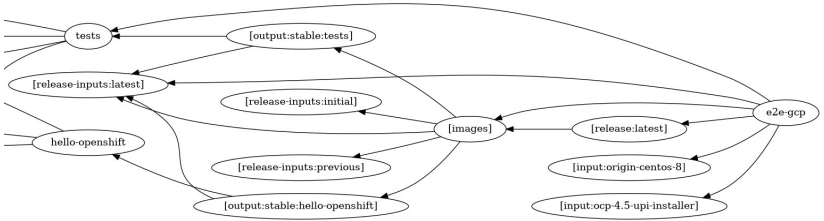


Each image that is built is also tagged into the `stable` `ImageStream`.



Red Hat

These are done by the steps in the format `[output:…]`.



Red Hat

This is difficult to capture since it is right in the middle of the graph, but notice the `[images]` step acting as a synchronization point between image build / release image imports and the release payload generation / test.
That is all it does: it is a synthetic step created purely to guarantee these operations are done in the proper order.

https://github.com/openshift/release/blob/master/ci-operator/jobs/openshift/origin/openshift-origin-master-postsubmits.yaml (simplified)

```
name: branch-ci-openshift-origin-master-images
spec:
  containers:
  - args:
    - --target=[images]
    - --promote
    command:
    - ci-operator
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

It can also be used as a target, such as we do in the `*-images` post-submit jobs.

```
Creating release image registry.build02.ci.openshift.org/\
    ci-op-8yq06grj/release:latest.
Snapshot integration stream into release \
    4.11.0-0.ci.test-2022-06-24-174214-ci-op-8yq06grj-latest \
    to tag release:latest
```


We now create the release payload from the `stable` `ImageStream`. The latter contains the initial images imported from the integration stream, along with the images just built from the input source code (recall they were tagged into `stable` after they were built).

By overlaying the new tags on top of the existing streams, we get a release payload where the repository images are overwritten by the ones which were just built from the input source code.

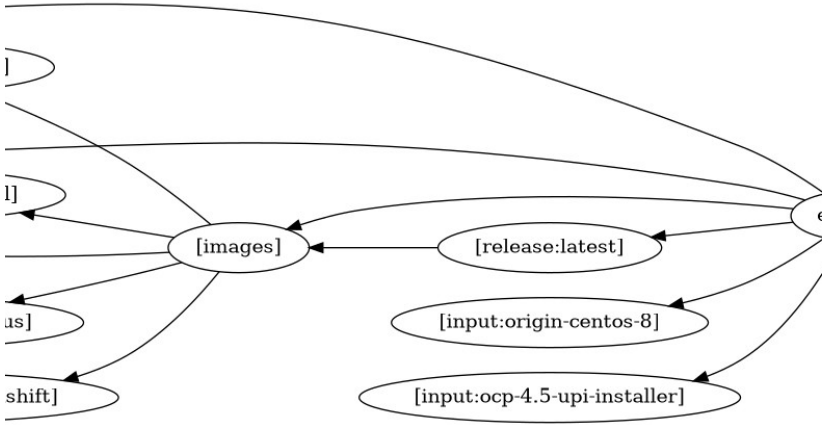
```
releases:
  latest:
    integration:
      include_built_images: true
      name: "4.11"
      namespace: ocp
```

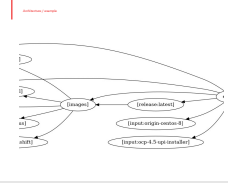

Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example


releases:
latest:
integration:
include_built_images: true
name: "4.11"
namespace: ocp





This is what the `[release:latest]` job does. The dependency on that step by the E2E test makes the resulting image available to the latter to be used by the installer to create the ephemeral cluster.

```
Acquiring leases for test e2e-gcp: \
    [gcp-openshift-gce-devel-ci-2-quota-slice]
Acquired 1 lease(s) for \
    gcp-openshift-gce-devel-ci-2-quota-slice: \
    [us-central1--gcp-openshift-gce-devel-ci-2-quota-slice-08]
```


Red Hat

2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example

We are finally at the beginning of the test execution now. Because it is an E2E test (approximated by "it declares a cluster profile"), `ci-operator` will contact the leasing server (i.e. Boskos) to acquire a lease for the ephemeral cluster.

https://github.com/openshift/ci-tools/blob/master/pkg/api/types.go (heavily abbreviated)

```
type ClusterProfile string

const ClusterProfileGCP2 ClusterProfile =
    "gcp-openshift-gce-devel-ci-2"

func (p ClusterProfile) LeaseType() string {
    switch p {
    case ClusterProfileGCP2:
        return "gcp-openshift-gce-devel-ci-2-quota-slice"
    }
}
```


Red Hat

2022-07-01

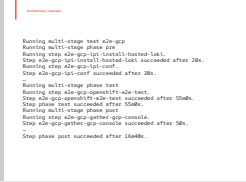
ci-operator
└Architecture
└Example
└Architecture / example

Each cluster profile (a string) is registered in `ci-tools` and associated with a resource type.

```
Running multi-stage test e2e-gcp
Running multi-stage phase pre
Running step e2e-gcp-ipi-install-hosted-loki.
Step e2e-gcp-ipi-install-hosted-loki succeeded after 20s.
Running step e2e-gcp-ipi-conf.
Step e2e-gcp-ipi-conf succeeded after 20s.
…
Running multi-stage phase test
Running step e2e-gcp-openshift-e2e-test.
Step e2e-gcp-openshift-e2e-test succeeded after 55m0s.
Step phase test succeeded after 55m0s.
Running multi-stage phase post
Running step e2e-gcp-gather-gcp-console.
Step e2e-gcp-gather-gcp-console succeeded after 50s.
…
Step phase post succeeded after 16m40s.
```


2022-07-01

ci-operator
└Architecture
└Example
└Architecture / example



```
Releasing leases for test e2e-gcp
Ran for 1h58m30s
Reporting job state 'succeeded'
```


Red Hat

2022-07-01

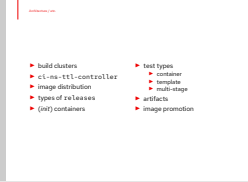
ci-operator
└─Architecture
└─Example
└─Architecture / example

- build clusters
- `ci-ns-ttl-controller`
- image distribution
- types of `releases`
- (*init*) containers
- test types
  - container
  - template
  - multi-stage
- artifacts
- image promotion

Red Hat

2022-07-01

`ci-operator`
└Architecture
└Etc.
└Architecture / etc.



Thank you

Red Hat