Red Hat

ci-operator multi-stage tests

Bruno Barcarol Guimarães

Red Hat

Introduction
- Motivation

Test definitions
- Phases
- Images
- Credentials
- Parameters
- Dependencies
- etc.

Step registry
- Discoverable
- Referenceable
- Verifiable
- Reusable

Red Hat

2022-10-20

`ci-operator` multi-stage tests

└Overview

This presentation can be viewed independentely, but is also a continuation of the previous two, which can be found in the `ci-docs` page:

- The initial `ci-operator` presentation has more details about the overall architecture and details and can help connect the topics presented here.
- The E2E test presentation has some extra historical context and shows in more detail how multi-stage tests are used in the OpenShift CI system.

- https://docs.google.com/document/d/1md-1BMf4_7mtKgGVoeZ3jOh4zSIBSjwl6vTTAYESwIM
  - *Multi-Stage Tests Design Document*
- https://docs.ci.openshift.org
  - docs/architecture/step-registry
  - docs/architecture/ci-operator

Red Hat

Documentation for the topics covered today is somewhat scattered among several pages. The main content is in the dedicated step registry page, but some descriptions and examples can also be found in more general pages such as `ci-operator` and others.

The original design document is also a good source of information about the basic architecture. It also describes very well the historical context in which multi-stage tests and the step registry were developed and added to `ci-operator` and the OpenShift CI.

ca. Aug 2019

- Two test types.
  - container
  - template
- Desire to create tests for increasingly varied scenarios.
- Existing tests already complex and barely maintained.

Red Hat

2022-10-20

`ci-operator` multi-stage tests
└Introduction
└Motivation
└Introduction / motivation

That context in summary is this: `ci-operator` started its life supporting only simple *container* tests. These are fairly self-contained tests which execute a single command using a container image.
Then (likely on a Sunday), *template* tests were added. These were amorphous tests which bypassed most of the configuration format and instead injected a new test definition at runtime. Creating and maintaining a template test was unnecessarily difficult, so practically only a few people had the knowledge and the stomach to do it.
At the same time, the OpenShift CI was growing, being used both by more components and for more varied types of tests. It was clear that requiring a new template test for each new test scenario would be impossible, so a new format for test definitions had to be created.

Ah, the templates...

complex, esoteric and fragile

difficult to extend and use

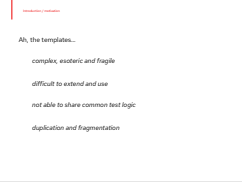
not able to share common test logic

duplication and fragmentation

Red Hat

2022-10-20

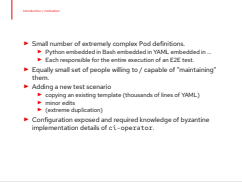
`ci-operator` multi-stage tests
└Introduction
└Motivation
└Introduction / motivation



This sentiment is visible in the design document, which constantly mentions the limitations of templates which impeded the maintenance of existing and creation of new tests.

- Small number of extremely complex Pod definitions.
  - Python embedded in Bash embedded in YAML embedded in ...
  - Each responsible for the entire execution of an E2E test.
- Equally small set of people willing to / capable of “maintaining” them.
- Adding a new test scenario
  - copying an existing template (thousands of lines of YAML)
  - minor edits
  - (extreme duplication)
- Configuration exposed and required knowledge of byzantine implementation details of `ci-operator`.

Red Hat



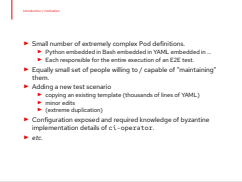
Here is a small selection (an entire presentation could be made about them):

- Template tests are defined in a single YAML file containing, among other items, a Pod definition. The definition consisted of several inline `bash` scripts, sometimes with multiple fragments of other languages inside them.
- The entirety of the test flow had to be contained in this single YAML file.
- Because definitions were completely self-contained, creating new types of tests required blindly copying colossal YAML files and editing usually just a few lines, creating massive duplication and divergence. Changing common code required editing all copies.
- The integration with other parts of `ci-operator` (images, releases, etc.) was very precarious, requiring test authors to know obscure aspects of the underlying implementation.

For these reasons and many others, the set of maintainers of these test definitions was virtually non-existent.

- Small number of extremely complex Pod definitions.
  - Python embedded in Bash embedded in YAML embedded in …
  - Each responsible for the entire execution of an E2E test.
- Equally small set of people willing to / capable of “maintaining” them.
- Adding a new test scenario
  - copying an existing template (thousands of lines of YAML)
  - minor edits
  - (extreme duplication)
- Configuration exposed and required knowledge of byzantine implementation details of `ci-operator`.
- *etc.*

Red Hat



Here is a small selection (an entire presentation could be made about them):

- Template tests are defined in a single YAML file containing, among other items, a Pod definition. The definition consisted of several inline `bash` scripts, sometimes with multiple fragments of other languages inside them.
- The entirety of the test flow had to be contained in this single YAML file.
- Because definitions were completely self-contained, creating new types of tests required blindly copying colossal YAML files and editing usually just a few lines, creating massive duplication and divergence. Changing common code required editing all copies.
- The integration with other parts of `ci-operator` (images, releases, etc.) was very precarious, requiring test authors to know obscure aspects of the underlying implementation.

For these reasons and many others, the set of maintainers of these test definitions was virtually non-existent.

Test definitions

Red Hat

- regular `ci-operator` test
  - images
  - release images
  - artifacts
  - cluster profiles
  - ...

Red Hat

`ci-operator` multi-stage tests

The first thing to note about multi-stage tests is that they are just another type of test, just as container tests are. This makes all other parts of `ci-operator` easily and naturally available to them.

On the implementation side, they are just another type of `ci-operator` step and are fully defined by the configuration file (and registry). Parsing and interpreting a multi-stage test does not require any of the arcane techniques used in the implementation of template tests.

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

```
tests:
- as: e2e
  steps:
    test:
    - as: e2e
      commands: … make e2e
      from: test-bin
      # …
```

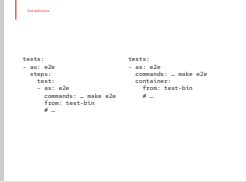

Red Hat

Multi-stage tests are defined in the configuration file just like any other type of test. A multi-stage test is distinguished by the `steps` entry. Definitions can be complex, but `e2e` in `ci-tools` is a good example of a minimal test.
(the actual test also has a `credentials` section, which is not revelant for this example and will be explained later)

```yaml
tests:
- as: e2e
  steps:
    test:
    - as: e2e
      commands: … make e2e
      from: test-bin
      # …
```

```yaml
tests:
- as: e2e
  commands: … make e2e
  container:
    from: test-bin
    # …
```


Red Hat



With this format, it is equivalent to the `container` test type (unifying their implementation is a long-term goal). These tests will have slightly different executions, but will have the same overall effect.

- pre/test/post
- serial execution
  - “short-circuit” execution for pre/test
  - post steps always executed
- each step corresponds to a Pod
  - shared data can be placed in a special directory

Red Hat

What differentiates multi-stage tests from simple container tests (or from templates, for that matter) are the *execution phases*: `pre`, `test`, and `post`. Superficially, they are simple sequences of steps (isolated test scripts) executed in serial, but each phase has distinct semantic characteristics.

`pre` is a sequence of preparatory steps. It should perform the setup necessary for the test to be executed (e.g. creating an ephemeral cluster). If any of the steps fail, it is assumed that the test cannot continue and the rest of the `pre` steps as well as all of the `test` steps are not executed.

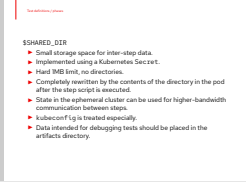
`test` is a sequence of one or more steps which execute the actual test code. If any of the steps fail, the rest of the steps are not executed.

`post` is a sequence of steps which releases any resources acquired by the previous phases. It is always executed and, unlike the others, always executed in its entirety, even if some of its steps fail.

$SHARED_DIR

- Small storage space for inter-step data.
- Implemented using a Kubernetes `Secret`.
- Hard 1MB limit, no directories.
- Completely rewritten by the contents of the directory in the pod after the step script is executed.
- State in the ephemeral cluster can be used for higher-bandwidth communication between steps.
- `kubeconfig` is treated especially.
- Data intended for debugging tests should be placed in the artifacts directory.

Red Hat



Unlike template tests:

- Each step is executed in its own isolated pod.
- Execution is serial: a step is executed only after the previous step ends.
- `ci-operator` manages the execution of individual steps.

For data which has to be passed between steps, a small storage space is provided, which is mounted in every pod. It is implemented using Kubernetes `Secrets`, so it has limitations. However, most tests can use external means (such as the ephemeral OpenShift cluster) for larger data. The artifacts directory is also available, just like in other types of tests.

Because `ci-operator` is optimized for OpenShift E2E tests, some files in the shared directory (such as the `kubeconfig` for ephemeral clusters) are treated specially.

Images

- `from`
  - pipeline images
    - `root`, `src`, `bin`, …
    - `base_images`
    - `images`
  - “stable” images
    - `releases`
    - `tag_specification`
- `from_image`
  - ≈ `base_images`
  - `from_image:`

```
namespace: ocp
name: upi-installer
tag: 4.12
```


Red Hat

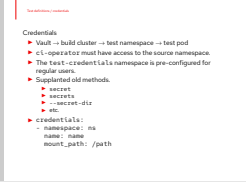
Unlike container tests, steps can be executed using any `ci-operator` image. Pipeline images, which are either imported or built, can be used, as well as images from release streams or payloads.

A mechanism (`from_image`) exists for using imported images in shared test definitions. Here, we will just note its mode of operation is equivalent to `base_images`; shared definitions will be explained in the step registry section.

Credentials

- Vault → build cluster → test namespace → test pod
- `ci-operator` must have access to the source namespace.
- The `test-credentials` namespace is pre-configured for regular users.
- Supplanted old methods.
  - `secret`
  - `secrets`
  - `--secret-dir`
  - etc.
-
```
credentials:
- namespace: ns
  name: name
  mount_path: /path
```


Red Hat



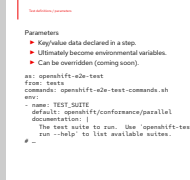
Another major improvement over other test types is the `credentials` section available in steps. The `Secret` objects listed therein will be imported into the test namespace and mounted in the corresponding pods.
The `Secret` must already exist and be accessible, but no other setup is necessary. For regular users, the current flow is to create a secret collection in Vault and synchronize it to the build clusters using special values in the credentials. A `test-credentials` namespace is preconfigured in each cluster for this purpose.

Parameters

- Key/value data declared in a step.
- Ultimately become environmental variables.
- Can be overridden (coming soon).

```
as: openshift-e2e-test
from: tests
commands: openshift-e2e-test-commands.sh
env:
- name: TEST_SUITE
  default: openshift/conformance/parallel
  documentation: |
    The test suite to run.  Use 'openshift-test
    run --help' to list available suites.
# …
```


Red Hat



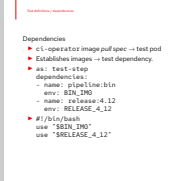
Parameters are a way of generating slight test variations without needing a completely new step definition. Ultimately, they are key/value data which become simple environmental variables set in the corresponding pods. One or more steps can declare a parameter, optionally with a default value – all parameters must be declared and be given a value in a test definition.

The hierarchical relation between registry components (explained in a later section) allows great freedom in how parameters can be given values and how tests can be composed.

Dependencies

- `ci-operator` image *pull spec* → test pod
- Establishes images → test dependency.
- ```
  as: test-step
  dependencies:
  - name: pipeline:bin
    env: BIN_IMG
  - name: release:4.12
    env: RELEASE_4_12
  ```
- ```
  #!/bin/bash
  use "$BIN_IMG"
  use "$RELEASE_4_12"
  ```


Red Hat



Tests sometimes need to refer to imported or built images, not as their execution image, but as a general *pull spec*. For this, the `dependencies` section can be used. `name` references either a pipeline or release image: the *pull spec* referring to the image in the temporary namespace will be available to the corresponding pod as an environmental variable.

This also establishes the dependencies between the required (`ci-operator`) steps and the test so that the latter is only executed when the images are available.

```
# openshift-e2e-tests-ref.yaml
dependencies:
- name: "release:latest"
  env: OPENSHIFT_UPGRADE_RELEASE_IMAGE_OVERRIDE


# openshift-e2e-tests-commands.sh
openshift-tests run-upgrade \
    "${TEST_UPGRADE_SUITE}" \
    --to-image \
        "${OPENSHIFT_UPGRADE_RELEASE_IMAGE_OVERRIDE}" \
    --options "${TEST_UPGRADE_OPTIONS-}" \
    --provider "${TEST_PROVIDER}" \
    -o "${ARTIFACT_DIR}/e2e.log" \
    --junit-dir "${ARTIFACT_DIR}/junit"
```


Red Hat

2022-10-20

`ci-operator` multi-stage tests
└Test definitions
└Dependencies
└Test definitions / dependencies

One (simplified) example is shown here, where the upgrade test references whatever the `latest` release payload is in a particular test namespace.

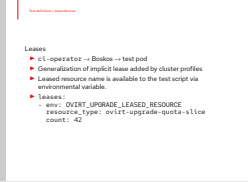
Leases

- `ci-operator` → Boskos → test pod
- Generalization of implicit lease added by cluster profiles
- Leased resource name is available to the test script via environmental variable.
- `leases:`
  `- env: OVIRT_UPGRADE_LEASED_RESOURCE`
  `resource_type: ovirt-upgrade-quota-slice`
  `count: 42`

Red Hat

2022-10-20

`ci-operator` multi-stage tests
└Test definitions
└Dependencies
└Test definitions / dependencies



`ci-operator` has an integration with the Boskos leasing server, which allows *resources* of limited capacity to be declared. `ci-operator` will request these resources before starting the execution of the test, periodically renew them, and release them when the test ends.
Initially (even in the template days, but also currently), this was triggered by cluster profiles. Each profile is declared in `ci-tools` and has a resource type associated with it.
`leases` are a generalization of this concept. Instead of a cluster profile, or in addition to it, a test or registry component can declare additional resources which it requires before it is allowed to start. The name of the resource is available as an environmental variable, which is often used to pass some information to the test step.

- best-effort steps
- catalogues / optional operators
- `KUBECONFIG`
- cluster profiles
- oc CLI injection
- no `ServiceAccount` credentials
- cluster claims
- VPN connection
- ...

Red Hat

This is an abbreviated list of aspects of multi-stage tests which cannot be covered today due to time constraints. Contrary to template tests, most of the implementation is properly documented, so consult the pages linked at the beginning of the presentation.

Step registry

Goals

- discoverable
- referenceable
- verifiable
- reusable

Red Hat

Beyond improving the implementation of tests, the design of multi-stage tests also had the major goal of improving the *process* of creating tests. It identified several attributes which the new format should have, discussed in the next sections.

- https://steps.ci.openshift.org
- https://steps.ci.openshift.org/workflow/ipi-aws
- https://steps.ci.openshift.org/chain/ipi-aws-pre
- https://steps.ci.openshift.org/reference/ipi-install-install

Red Hat

2022-10-20

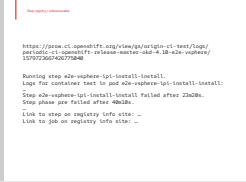
ci-operator multi-stage tests
└Step registry
└Discoverable
└Step registry / discoverable

Each component of a test definition is easily discoverable. There is no longer the need to excavate giant bash scripts embedded in YAML definitions. steps.ci.openshift.org is a web interface for test definitions which is heavily cross-linked. All jobs, tests, and registry components are listed, in a manner which makes it easy to discover exactly how a test is or can be defined.

```
https://prow.ci.openshift.org/view/gs/origin-ci-test/logs/
periodic-ci-openshift-release-master-okd-4.10-e2e-vsphere/
1579723667426775040


Running step e2e-vsphere-ipi-install-install.
Logs for container test in pod e2e-vsphere-ipi-install-install:
…
Step e2e-vsphere-ipi-install-install failed after 23m20s.
Step phase pre failed after 40m10s.
…
Link to step on registry info site: …
Link to job on registry info site: …
```


`ci-operator` multi-stage tests
- Step registry
  - Referenceable
    - Step registry / referenceable



When a job fails, the `ci-operator` output includes links to the definitions of the steps which failed.

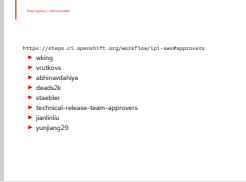
`https://steps.ci.openshift.org/workflow/ipi-aws#approvers`

- wking
- vrutkovs
- abhinavdahiya
- deads2k
- staebler
- technical-release-team-approvers
- jianlinliu
- yunjiang29

Red Hat

2022-10-20

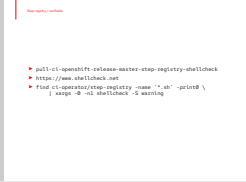
`ci-operator` multi-stage tests
└Step registry
└Referenceable
└Step registry / referenceable



Each of those pages also has links to the definition in GitHub, as well as a list of its `OWNERS` (reviewers and approvers). This way, the error output has (in principle) the information required to go from a failure to the possible cause and to those who may be able to assist.

- `pull-ci-openshift-release-master-step-registry-shellcheck`
- `https://www.shellcheck.net`
-
```
find ci-operator/step-registry -name '*.sh' -print0 \
    | xargs -0 -n1 shellcheck -S warning
```


Red Hat



The quality of scripts in the step registry is verified using `shellcheck`, a (Haskell) program which identifies problems in `bash` source code (syntax errors, unquoted variables, etc.). It is executed as a blocking pre-submit job which verifies every shell script in the registry directory.

- reference
- chain
- workflow

Red Hat

2022-10-20

`ci-operator` multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

The step registry is a central place where parts of test definitions are stored. Several types of registry *components* can be combined and used by a large number of tests.

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

```
ref:
  as: ipi-install-install
  from: installer
  grace_period: 10m
  commands: ipi-install-install-commands.sh
  cli: latest
  resources:
    requests:
      cpu: 1000m
      memory: 2Gi
```

(cont.)

Red Hat

2022-10-20

`ci-operator` multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

A *reference* is the lowest level of step definition. It corresponds directly to the step definition inline in a test shown in previous examples. In this way, sharing code between tests can be as simple as moving a step definition virtually unchanged to the registry and referencing it.

In this example, the installation step which creates ephemeral clusters is defined once in the registry and used everywhere it is needed with a simple `ref: ipi-install-install`.

(cont.)

```
  credentials:
  - namespace: test-credentials
    name: loki-stage-collector-test-secret
    mount_path: /var/run/loki-secret
  # …
  env:
  - name: OPENSHIFT_INSTALL_EXPERIMENTAL_DUAL_STACK
    default: "false"
    documentation: Using experimental Azure dual-stack support
  # …
  dependencies:
  - name: "release:latest"
    env: OPENSHIFT_INSTALL_RELEASE_IMAGE_OVERRIDE
  # …
  documentation: |-
    The IPI install step runs the OpenShift Installer …
```


Red Hat

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

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


Red Hat

Sequences of steps can be combined into *chains*, which are analogous to the phases of a multi-stage test. Chains are the main method of code reuse in the registry. Beyond simply grouping steps, they can also contain definitions for step parameters and other options, as explained later.

```
https://steps.ci.openshift.org/workflow/ipi-aws


workflow:
  as: ipi-aws
  steps:
    pre:
    - chain: ipi-aws-pre
    post:
    - chain: ipi-aws-post
  documentation: |-
    The IPI workflow provides pre- and post- steps that
    provision and deprovision an OpenShift cluster with a
    default configuration on AWS, allowing job authors to
    inject their own end-to-end test logic.

    All modifications to this workflow should be done by
    modifying the ipi-aws-{pre,post}`chains to allow other
    workflows to mimic and extend this base workflow without
    a need to backport changes.
```


Finally, a complete test definition can be grouped into a *workflow*. A workflow definition is exactly the same as a test definition, and usually encapsulates a test flow from beginning to end.

With a workflow, a shared test definition can be as simple as a few lines. If necessary, each phase (i.e. `pre`, `test`, `post`) can still be redefined in the test which includes the workflow. This will replace the entire sequence of steps for that particular phase.

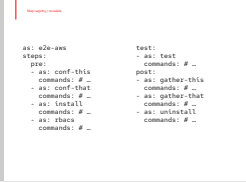
```yaml
as: e2e-aws
steps:
  pre:
  - as: conf-this
    commands: # …
  - as: conf-that
    commands: # …
  - as: install
    commands: # …
  - as: rbacs
    commands: # …
```

```yaml
test:
- as: test
  commands: # …
post:
- as: gather-this
  commands: # …
- as: gather-that
  commands: # …
- as: uninstall
  commands: # …
```


Red Hat

2022-10-20

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable



The next few examples will demonstrate the process of going from a completely idiosyncratic test to one that can be shared by multiple definitions in a few lines of YAML.

This definition shows a typical OpenShift E2E test: a cluster is created based on some configuration steps, the tests are executed, and the cluster is destroyed. Here, `commands…` in each definition stands for the particular entries which would be declared in each step, which can anywhere from several lines to several pages long.

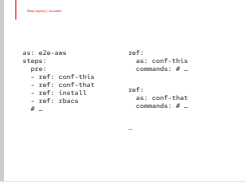
```
as: e2e-aws
steps:
  pre:
  - ref: conf-this
  - ref: conf-that
  - ref: install
  - ref: rbacs
  # …
```

```
ref:
  as: conf-this
  commands: # …

ref:
  as: conf-that
  commands: # …

…
```


Red Hat



The first… step in the process would be to move individual step definitions to the registry and replace the original ones with references. This allows any number of tests to use the shared definition and is already an enormous improvement over template tests.

```
as: e2e-aws
steps:
  pre:
  - chain: aws-pre
  test: # …
  post:
  - chain: aws-post
```

```
chain:
as: aws-pre
  steps:
  - ref: conf-this
  - ref: conf-that
  - ref: install
  - ref: rbacs
```

```
chain:
as: aws-post
steps:
# …
```


Red Hat

Next, a sequence of steps from the registry can be placed in one or more chains. This allows the sequence (and/or its configuration) to be changed without the need to modify every test definition.

```
as: e2e-aws
steps:
  workflow: aws-ipi
  test: # …
```

```
workflow:
  as: aws-ipi
  pre:
  - chain: aws-pre
  post:
  - chain: aws-post
```


Red Hat

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

A complete test pattern (e.g. “E2E test on AWS”) can then be put into a workflow. This abstracts the setup and cleanup phases so that the test definition contains only a reference to the workflow and the actual test steps which are to be executed.

```
as: e2e-aws
steps:
  workflow: aws-ipi
```

```
workflow:
  as: openshift-e2e-aws
  pre:
  - chain: aws-pre
  test:
  - ref: openshift-e2e-test
  post:
  - chain: aws-post
```


Red Hat

2022-10-20

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

Going even further, an entire test suite can be shared among several repositories. In this case, the “OpenShift on AWS E2E” test suite is put into its own workflow (n.b.: which shares the pre/post chains with other workflows). Repositories can include this entire suite by simply declaring a test which references the workflow.

```
$ find ci-operator/step-registry/ -name 'ipi-conf-*-ref.yaml' \
    | wc -l
75
$ find ci-operator/step-registry/ -name 'ipi-conf-*-ref.yaml' \
    | sed 's,.*/,,; s/-ref\.yaml//' | shuf | head -15 | sort
ipi-conf-additional-enabled-capabilities
ipi-conf-alibabacloud
ipi-conf-azure-provisioned-des
ipi-conf-azurestack-creds
ipi-conf-azure-vmgenv1
ipi-conf-azure-workers-marketimage
ipi-conf-etcd-on-ramfs
ipi-conf-libvirt
ipi-conf-openstack-enable-octavia
ipi-conf-ovirt-generate-csi-test-manifest
ipi-conf-ovirt-generate-csi-test-manifest-release-4.6-4.8
ipi-conf-ovirt-generate-install-config
ipi-conf-ovirt-generate-install-config-params
ipi-conf-ovirt-generate-ovirt-config
ipi-conf-vsphere-zones
```


Since the configuration and installation steps have their own chains in the registry, it is very easy to combine one or more configuration steps to create a specific test scenario, then include all of the other registry components which implement the machinery behind the test (cluster installation, artifact gathering, etc.).

A quick look at the registry shows there is a large number of configuration steps, many of which can be combined (e.g. to create a test on a cluster with "additional capabilities" and "`etcd` on `ramfs` using `libvirt`).

```
https://docs.ci.openshift.org/docs/architecture/step-registry/
#hierarchical-propagation

as: openshift-e2e-test
env:
- name: TEST_SUITE
# …
```


Red Hat

2022-10-20

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

One last, more advanced method of reuse is what is called *hierarchical propagation*. Registry components used by a test are arranged as a tree: the test can include a workflow, which can include any number of chains, which can themselves include chains recursively, which ultimately include steps.
When all of these definitions are assembled to generate the final step list executed by the test, several configuration options are propagated from the root of the tree to the leaves (namely: parameters, dependencies, leases). This means definitions in tests override those in workflows, which in turn override those in chains, which in turn override those in steps. These definitions are strictly checked such that every option declared in a parent component has to be declared in a subcomponent and has to have a resulting value after this process completes.

```
tests:
- as: e2e
  steps:
    test:
    - ref: openshift-e2e-test
    env:
      TEST_SUITE: openshift/conformance/parallel
```


Red Hat

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

In this example, a test includes the step shown in the previous slide and specifies a value for its parameter. It must do so, since the parameter does not have a default. Similarly, giving the parameter a value without including a step which declares it would be an error.

```
workflow:
  as: openshift-e2e-serial
  steps:
    test:
    - ref: openshift-e2e-test
    env:
      TEST_SUITE: openshift/conformance/serial

tests:
- as: e2e
  steps:
    workflow: openshift-e2e-serial
```


Red Hat

2022-10-20

ci-operator multi-stage tests
└Step registry
└Reusable
└Step registry / reusable

Hierarchical propagation allows the parameter value to be defined in a workflow (or a chain). This way, any test which includes the workflow will use that value for the parameter, since it would propagate down from the workflow to the step.

While nonsensical in this case, the test could also give the parameter a value, which would override the value in the workflow, since tests are the root of the resolution tree.

Thank you

Red Hat