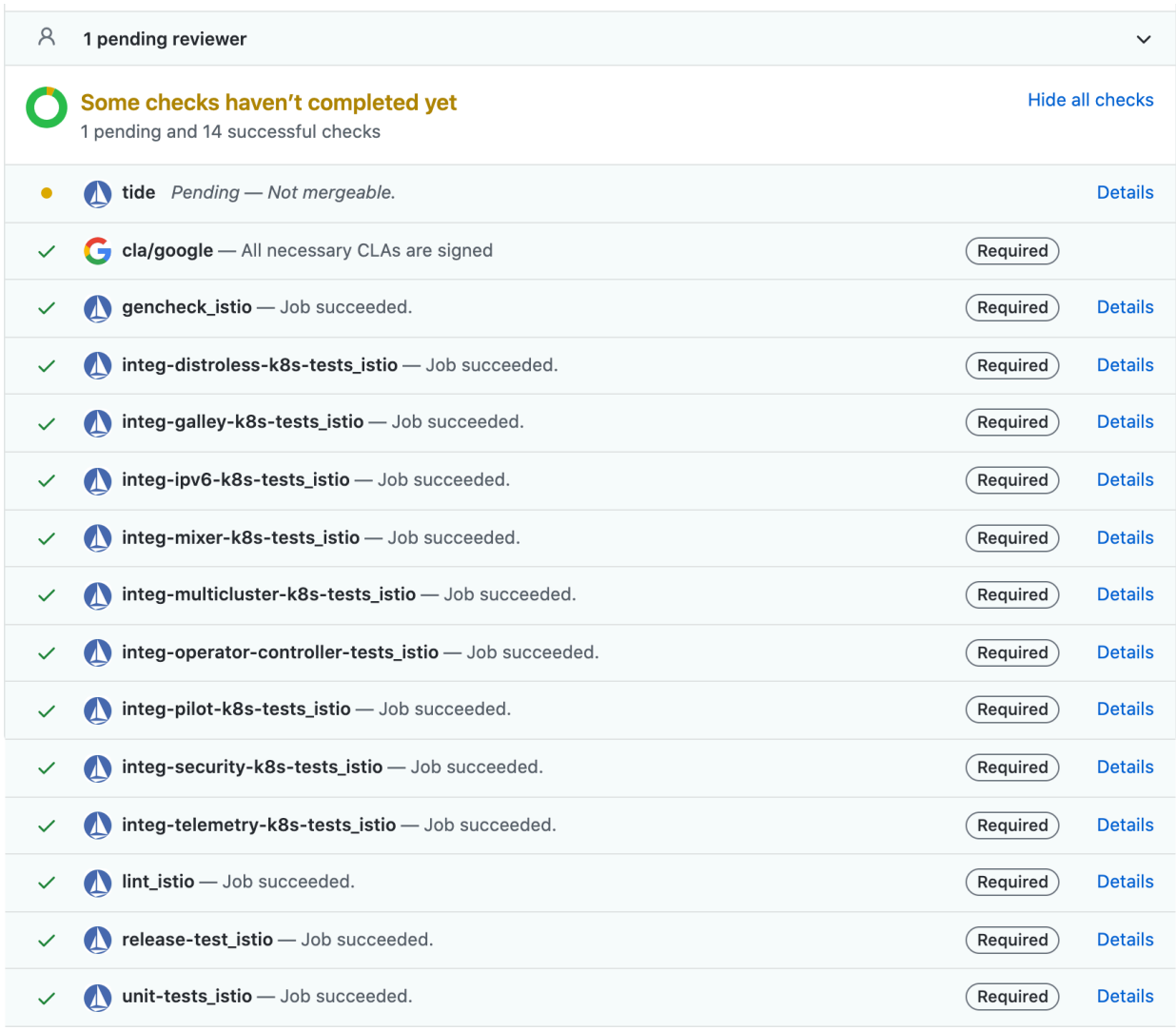
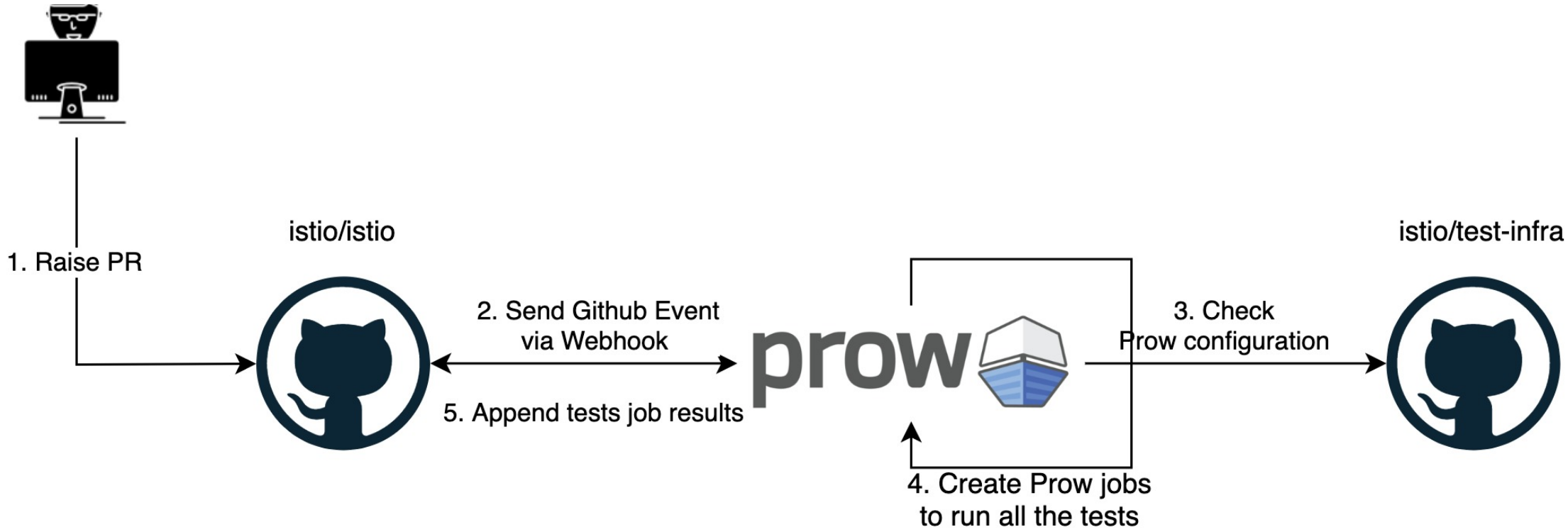
Istio Test Framework & Prow CICD Pipeline

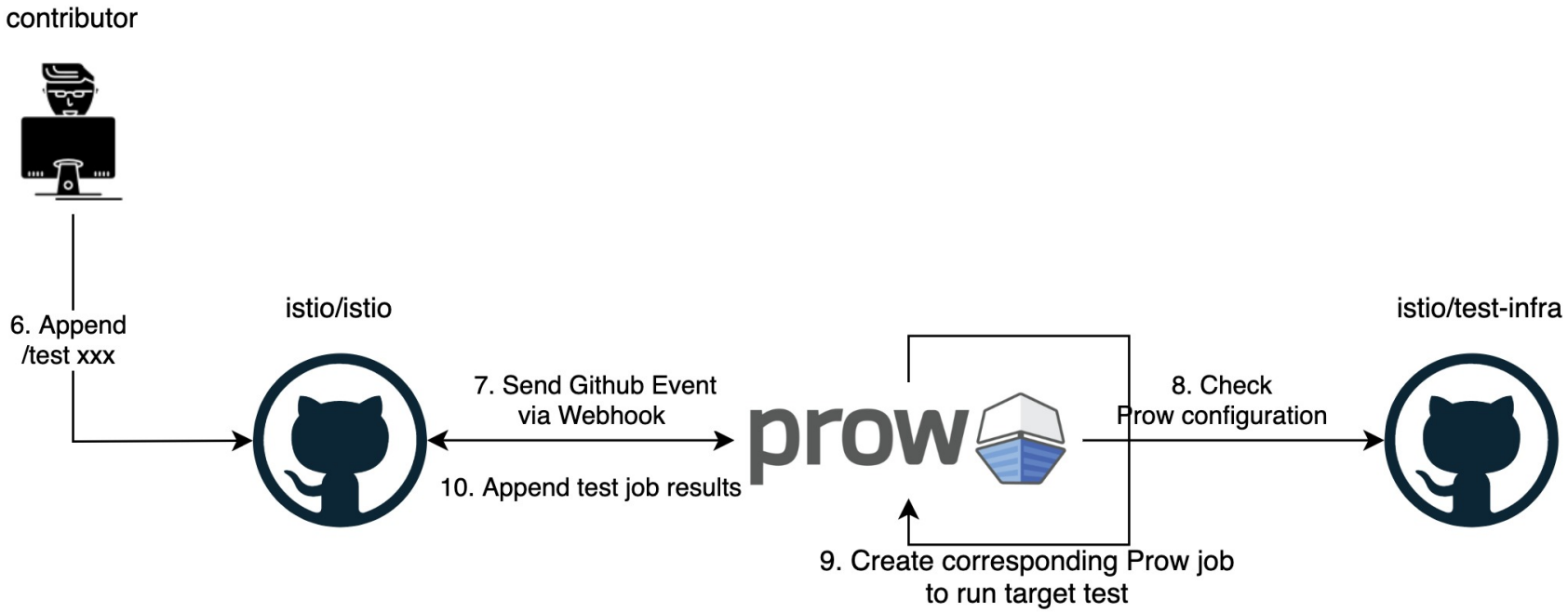
Email: llcao@cn.ibm.com
Github ID: morvencao

When you raise a PR to istio repo, there are quite a few tests running, how are they triggered under the hood?

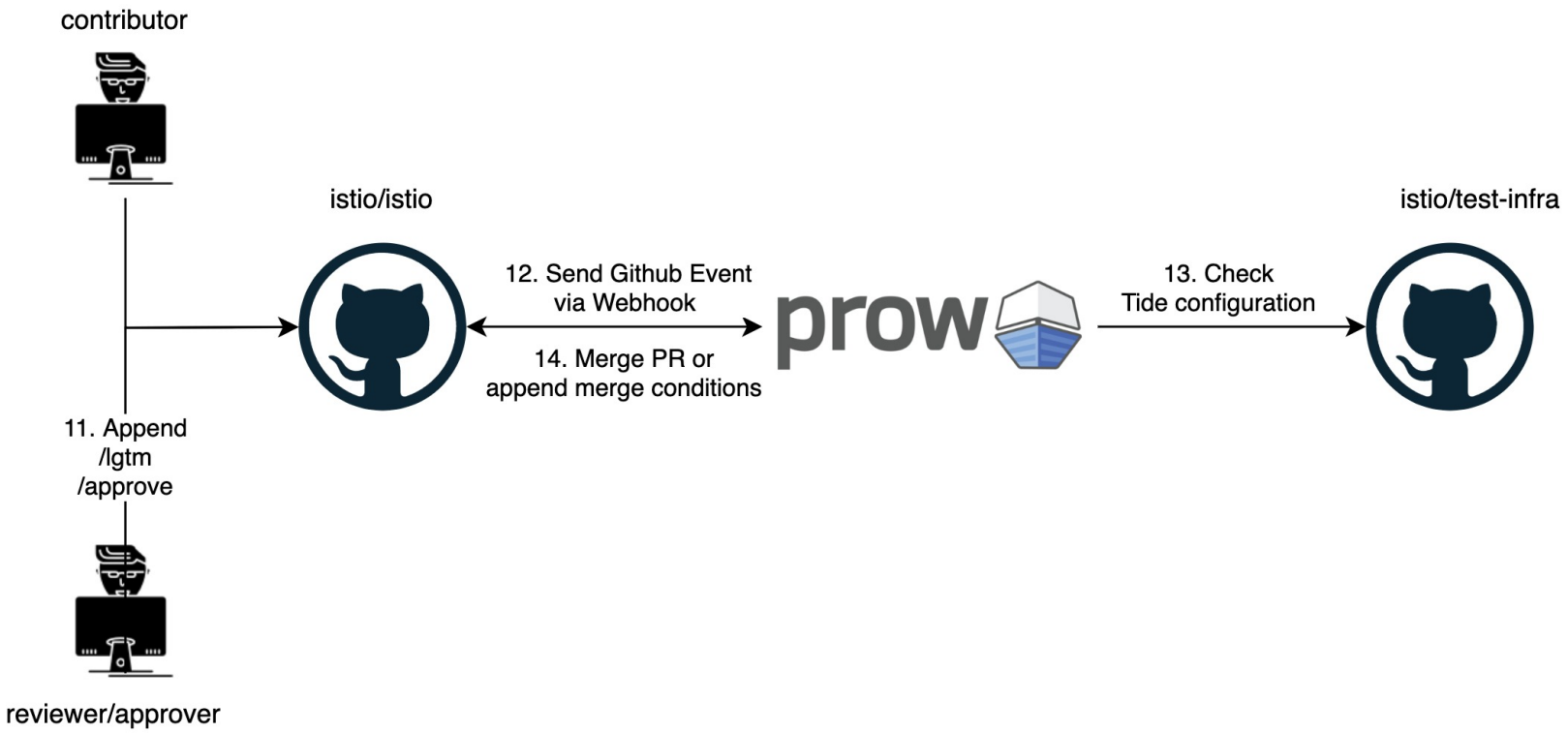




Overall process for triggering prow jobs and getting test results



Overall process for triggering specific test and getting test result

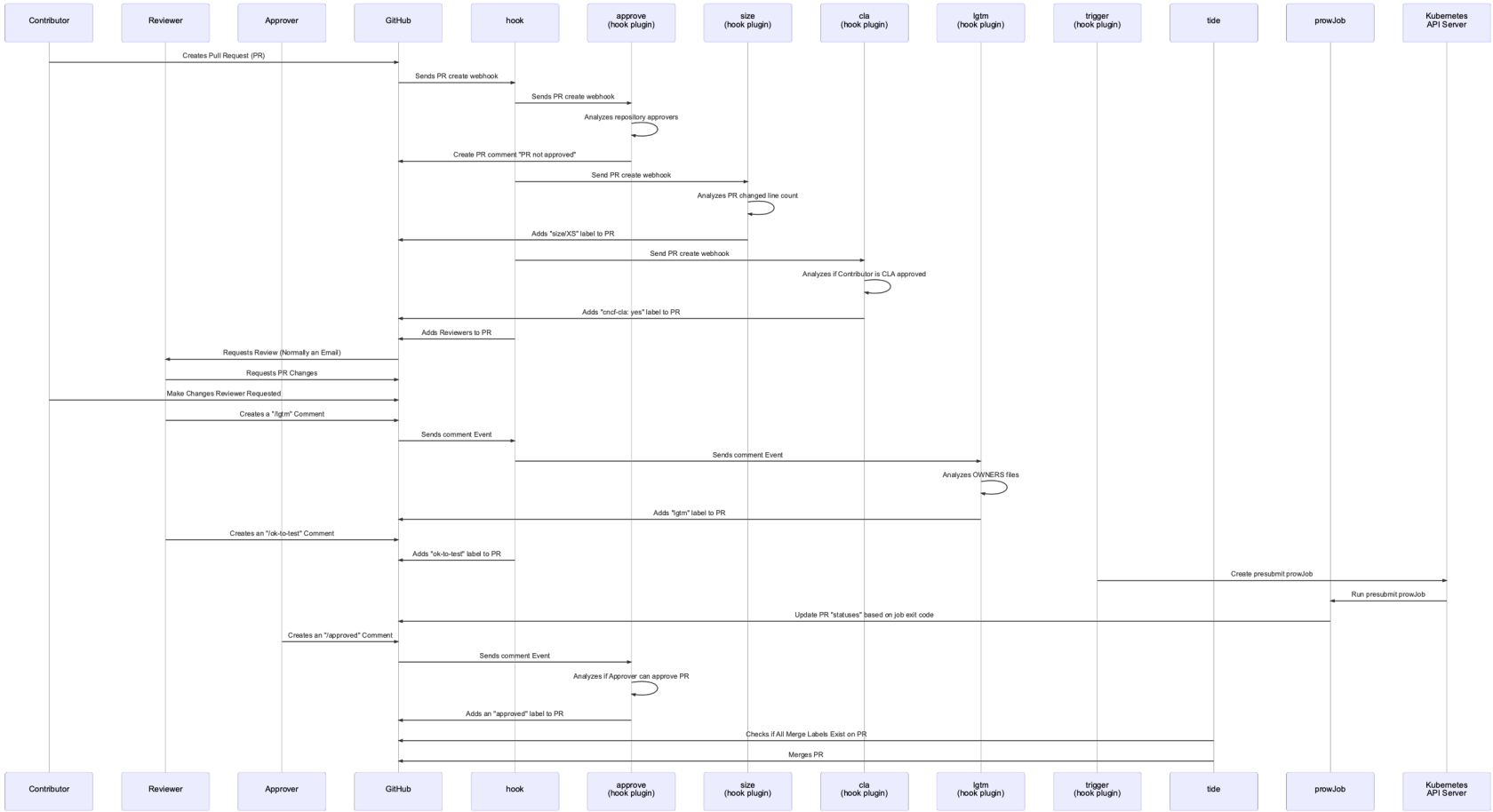


Overall process for review & approve & merge PR

So what's Prow?

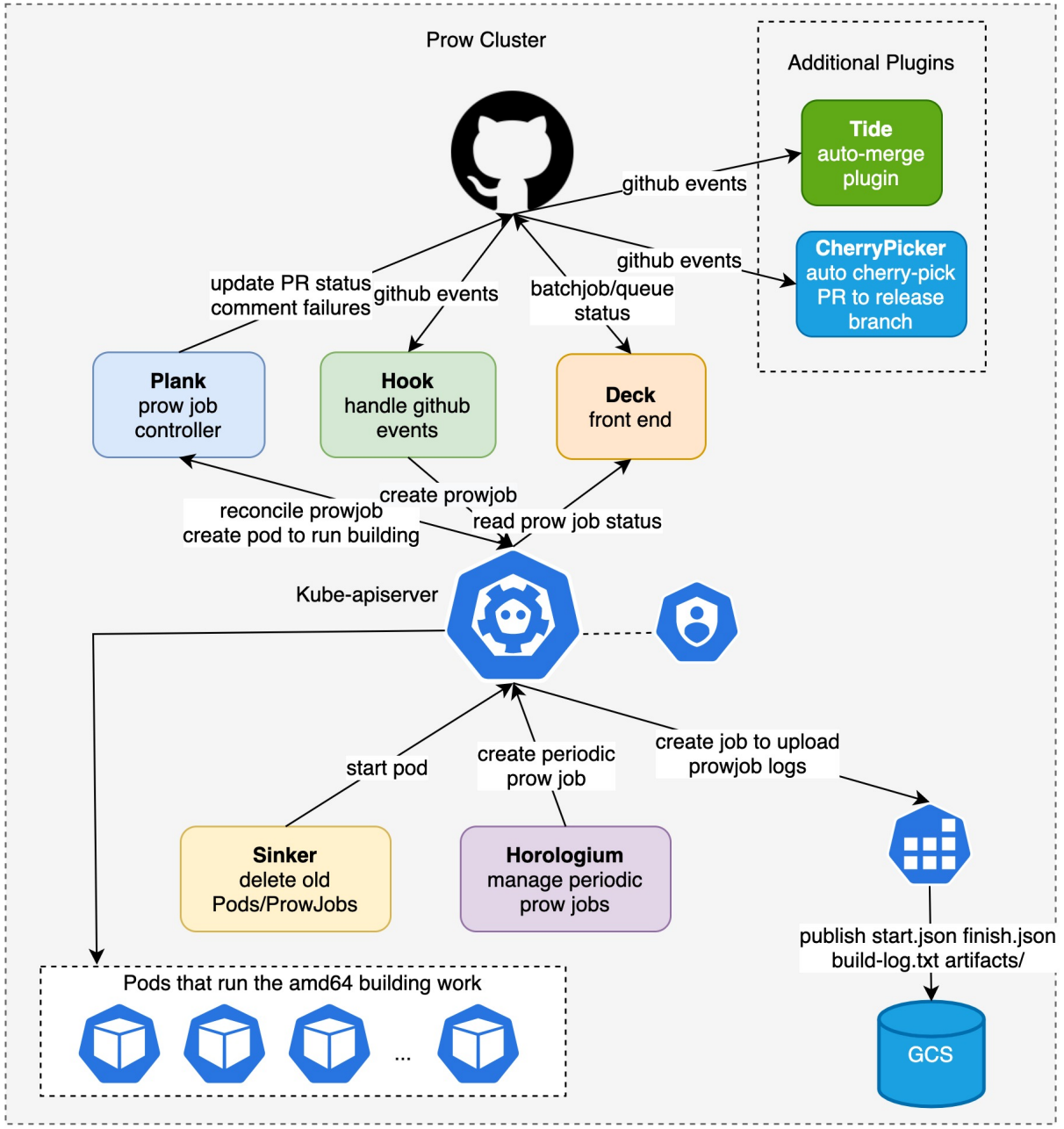
Prow is CICD Kubernetes-based CI/CD system, it provides:

- **Job execution for testing, batch processing, artifact publishing**
  - GitHub events are used to trigger post-PR-merge (postsubmit) jobs and on-PR-update (presubmit) jobs
  - Support for multiple job execution platforms
- **Pluggable GitHub bot automation that implements */foo* style commands**
- **GitHub merge automation(Tide) with batch testing logic**
- **Front end for viewing jobs, merge queue status, and more**
- **Automatic GitHub org/repo administration configured in source control**
- **…**



Prow's Interactions Sequence with PR

Prow Deployment Architecture



Prow Job Type

Three types of prow jobs:

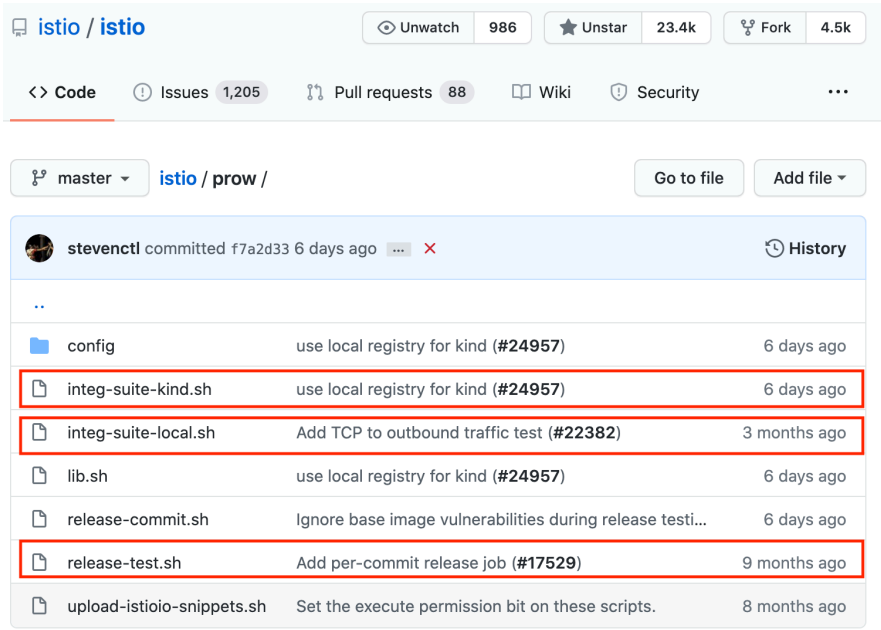
- **Presubmits** run against code in PRs
- **Postsubmits** run after merging code
- **Periodics** run on a periodic basis

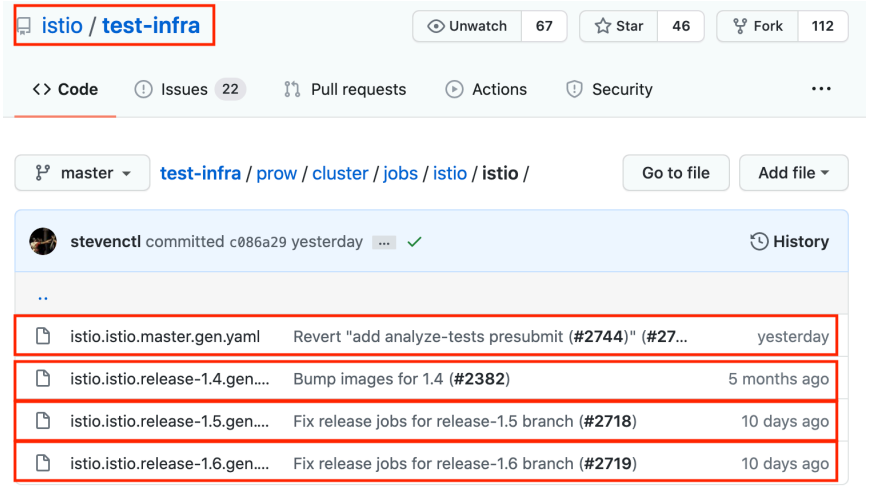
```yaml
org: istio
repo: istio
support_release_branching: true
image: gcr.io/istio-testing/build-tools:master-2020-06-25T05-18-39

jobs:
  - name: unit-tests
    command: [entrypoint, make, -e, "T=-v", build, racetest, binaries-test]

  - name: release-test
    type: presubmit
    command: [entrypoint, prow/release-test.sh]
    requirements: [gcp, docker]

  - name: release
    type: postsubmit
    command: [entrypoint, prow/release-commit.sh]
    requirements: [gcp, docker]

  - name: integ-galley-k8s-tests
    type: presubmit
    command: [entrypoint, prow/integ-suite-kind.sh, test.integration.galley.kube.presubmit]
    requirements: [kind]
    env:
      - name: TEST_SELECT
        value: "-postsubmit,-flaky,-multicluster"
```

Add Prow Job - 0

Add new prow job for your test suite

- Add script that run your test suite in **istio/istio** repo under ***prow*** folder
- Add configuration for new prow job in **istio/test-infra** repo in ***prow/config/jobs*** *folder*
- Configure the **entrypoint** in the new prow job repo in *prow/config/jobs*





Add Prow Job – 1

Prow Job Spec Example

```
presubmits:
  istio/istio:
  - always_run: true…
  - always_run: true…
  - always_run: true…
  - always_run: true…
  - always_run: true
    annotations:
      testgrid-dashboards: istio_istio
    branches:
    - ^master$
    decorate: true
    name: integ-pilot-k8s-tests_istio
    path_alias: istio.io/istio
    spec:
      containers:
      - command:
        - entrypoint
        - prow/integ-suite-kind.sh
        - test.integration.pilot.kube.presubmit
        env:
        - name: TEST_SELECT
          value: -postsubmit,-flaky,-multicluster
        image: gcr.io/istio-testing/build-tools:master-2020-06-30T00-03-39
        name: ""
```

test name shown in the PR

Need to start docker service before execute the entrypoint

Set up Kubernetes Cluster in docker with KinD

Make target in istio/istio repo

Add Prow Job – 2

Prow Job entrypoint & make target for kube env

prow > integ-suite-kind.sh

```
116  make init
117
118  if [[ -z "${SKIP_SETUP:-}" ]]; then
119    if [[ "${TOPOLOGY}" == "SINGLE_CLUSTER" ]]; then
120      time setup_kind_cluster "${IP_FAMILY}" "${NODE_IMAGE:-}"
121    else
122      # TODO: Support IPv6 multicluster
123      time setup_kind_clusters "${TOPOLOGY}" "${NODE_IMAGE:-}"
124
125      # Set the kube configs to point to the clusters.
126      export INTEGRATION_TEST_KUBECONFIG="${CLUSTER1_KUBECONFIG},${CLUSTER2_KUBECONFIG},${CLUSTER3_KUBECONFIG}"
127      export INTEGRATION_TEST_NETWORKS="0:test-network-0,1:test-network-0,2:test-network-1"
128    fi
129  fi
130
131  if [[ -z "${SKIP_BUILD:-}" ]]; then
132    time build_images "${PARAMS[*]}"
133    time setup_kind_registry
134    time kind_push_images
135  fi
136
137  # If a variant is defined, update the tag accordingly
138  if [[ -n "${VARIANT:-}" ]]; then
139    export TAG="${TAG}-${VARIANT}"
140  fi
141
142  # Run the test target if provided.
143  if [[ -n "${PARAMS:-}" ]]; then
144    make "${PARAMS[*]}"
145  fi
```

Setup single KinD cluster for single topology Or Multiple KinD clusters for multiCluster topology

Build Istio images and push to local docker registry

Trigger the make target

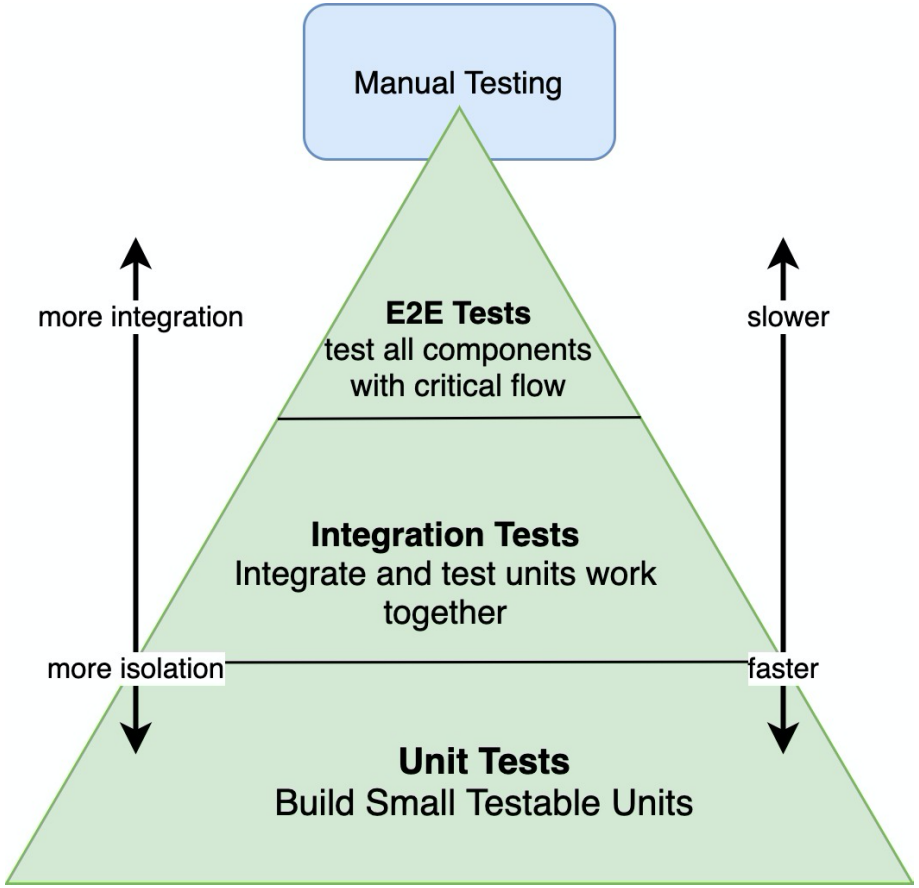
tests > integration > M tests.mk

```
75  # Generate presubmit integration test targets for each component in kubernetes environment
76  test.integration.%.kube.presubmit: | $(JUNIT_REPORT)
77      PATH=${PATH}:${ISTIO_OUT} $(GO) test -p 1 ${T} ./tests/integration/$(subst .,/,$*)/... -timeout 30m \
78      ${_INTEGRATION_TEST_FLAGS} ${_INTEGRATION_TEST_SELECT_FLAGS} \
79      2>&1 | tee >($(JUNIT_REPORT) > $(JUNIT_OUT))
80
81
82  # Presubmit integration tests targeting Kubernetes environment.
83  .PHONY: test.integration.kube.presubmit
84  test.integration.kube.presubmit: | $(JUNIT_REPORT)
85      PATH=${PATH}:${ISTIO_OUT} $(GO) test -p 1 ${T} $(shell go list ./tests/integration/... | grep -v /qualification | grep -v /examples) -timeout 30m \
86      ${_INTEGRATION_TEST_FLAGS} ${_INTEGRATION_TEST_SELECT_FLAGS} \
87      2>&1 | tee >($(JUNIT_REPORT) > $(JUNIT_OUT))
```

We have known how tests are triggered, but how to write tests?

Istio Tests Pyramid

- **Unit Tests***
- **Integration Tests***
- ~~**E2E Tests**~~
- **Release Tests**



Unit Tests

- **Unit tests should be fully hermetic**
- **All packages and any significant files require unit tests**
- **Unit tests are written using the standard Go testing package**
- **The preferred method of testing multiple scenarios or input is table driven testing**
- **Concurrent unit test runs must pass**
- **Unit tests coverage requirements**
- **Run unit tests:**

  make [|pilot|mixer|operator|…|-]test

  or for a single test:

  go test ./pilot/pkg/networking/core/v1alpha3/ -v –race

Unit Test Example

Example from istio/istio/pilot/pkg/kube/inject

Original Function:

```
// FindSidecar returns the pointer to the first container whose name matches the "istio-proxy".
func FindSidecar(containers []corev1.Container) *corev1.Container {
        for i := range containers {
                if containers[i].Name == ProxyContainerName {
                        return &containers[i]
                }
        }
        return nil
}
```

Unit Test Function:

```
func TestFindSidecar(t *testing.T) {
       proxy := corev1.Container{Name: "istio-proxy"}
       app := corev1.Container{Name: "app"}
       for _, tc := range []struct {
               name       string
               containers []corev1.Container
               index      int
       }{
               {"only-sidecar", []corev1.Container{proxy}, 0},
               {"app-and-sidecar", []corev1.Container{app, proxy}, 1},
               {"no-sidecar", []corev1.Container{app}, -1},
       } {
               got := FindSidecar(tc.containers)
               var want *corev1.Container
               if tc.index == -1 {
                       want = nil
               } else {
                       want = &tc.containers[tc.index]
               }
               if got != want {
                       t.Errorf("[%v] failed, want %v, got %v", tc.name, want, got)
               }
       }
}
```

Istio Integration Test Framework

Background:

- Hard to write tests case for cloud-based micro-services
- Running tests quickly and reliably is another challenge
- Supporting multiple cloud platform makes thing harder

Istio Integration Test Framework

Objects for the Istio Integration Test Framework:

❖ Writing Tests

- **Platform Agnostic:** The API abstracts away the details of the underlying platform
- **Reusable Tests:** Suites of tests can be written which will run against any platform that supports Istio

❖ Running Tests

- **Standard Tools:** Built on Go's testing infrastructure and run with standard commands (e.g. go test)
- **Easy:** Few or no flags are required to run tests out of the box
- **Fast:** With the ability to run processes natively on the host machine, running tests are orders of magnitude faster
- **Reliable:** Running tests natively are inherently more reliable than in-cluster

Writing Tests- 00

Getting Started

1. Create a new go package in **istio/test/integrations** for your test suites

```
$ cd ${ISTIO}/tests/integration
$ mkdir mysuite
```

2. Within that package, create go file and call ***framework.NewSuite()*** in ***TestMain***

```
func TestMain(m *testing.M) {
    framework.
        NewSuite("my_test", m).
        Run()
}
```

The call to ***framework.NewSuite()*** does the following:

- Starts the platform-specific environment. By default, the native environment is used. To run on Kubernetes, set the flag: --istio.test.env=kube
- Run all tests in the current package. This is the standard Go behavior for ***TestMain***

Writing Tests- 01

Add Tests

3. Define test the same package

```
func TestMyLogic(t *testing.T) {
    framework.
        NewTest(t).
        Run(func(ctx framework.TestContext) {
            // Create a component
            p := pilot.NewOrFail(ctx, ctx, cfg)

            // Use the component.
            // Apply Kubernetes Config
            ctx.ApplyConfigOrFail(ctx, nil, mycfg)

            // Do more stuff here.
        }
}
```

Every test will follow the pattern in the example above:

- Get the test context. The ***framework.TestContext*** is a wrapper around the underlying ***testing.T*** and implements the same interface. Test code should generally not interact with the ***testing.T*** directly.

- Get and use **components**. Each component (e.g. Pilot, Mixer, Apps) defines its own API.

Writing Tests- 02

Suite-level Checks

4. Support suite-level checks

```
func TestMain(m *testing.M) {
    framework.
        NewSuite("mysuite", m).
        // Deploy Istio on the cluster
        Setup(istio.Setup(nil, nil)).
        // Run your own custom setup
        Setup(mySetup).
        Run()
}

func mySetup(ctx resource.Context) error {
    // Your own setup code
    return nil
}
```

In the ***TestMain***, you can also restrict the test to particular environment, apply labels, or do test-wide setup, such as deploying Istio.

Writing Tests- 03

Sub-Tests

5. Istio test framework supports nested tests with ***ctx.NewSubTest(),*** similar to golang ***t.Run()***

```
func TestMyLogic(t *testing.T) {
    framework.
        NewTest(t).
        Run(func(ctx framework.TestContext) {

            // Create a component
            g := galley.NewOrFail(ctx, ctx, cfg)

            configs := []struct{
                name: string
                yaml: string
            } {
                // Some array of YAML
            }

            for _, cfg := range configs {
                ctx.NewSubTest(cfg.name).
                    Run(func(ctx framework.TestContext) {
                        ctx.ApplyConfigOrFail(ctx, nil, mycfg)
                        // Do more stuff here.
                    })
            }
        })
}
```

Note: calling ***subtest.Run()*** delegates *to* **t.Run()** in order to create a child ***testing.T***

Writing Tests- 04

Parallel Tests

5. Run test in parallel where possible:

```
func TestMyLogic(t *testing.T) {
    framework.
        NewTest(t).
        RunParallel(func(ctx framework.TestContext) {
            // ...
        }
}
```

Many tests can take a while to start up for a variety of reasons, such as waiting for pods to start or waiting for a particular piece of configuration to propagate throughout the system. It may be desirable to run these sorts of tests in **parallel** in some cases.

Note: Parallel tests rely on Go's ***t.Parallel()*** and will, therefore, have the same behavior.

Writing Tests- 05

Sub-tests and Parallel Tests

6. Sub-tests and parallel tests

```
func TestMyLogic(t *testing.T) {
    framework.NewTest(t).
        Run(func(ctx framework.TestContext) {
            ctx.NewSubTest("T1").
                Run(func(ctx framework.TestContext) {
                    ctx.NewSubTest("T1a").
                        RunParallel(func(ctx framework.TestContext) {
                            // Run in parallel with T1b
                        })
                    ctx.NewSubTest("T1b").
                        RunParallel(func(ctx framework.TestContext) {
                            // Run in parallel with T1a
                        })
                    // Exits before T1a and T1b are run.
                })

            ctx.NewSubTest("T2").
                Run(func(ctx framework.TestContext) {
                    ctx.NewSubTest("T2a").
                        RunParallel(func(ctx framework.TestContext) {
                            // Run in parallel with T2b
                        })
                    ctx.NewSubTest("T2b").
                        RunParallel(func(ctx framework.TestContext) {
                            // Run in parallel with T2a
                        })
                    // Exits before T2a and T2b are run.
                })
        })
}
```

A parallel test will run in parallel with siblings that share the same parent test. The parent test function will exit before the parallel children are executed.

Writing Tests- 06

Using Components

7. **Components** are utilities that provide abstractions for Istio resources

```
func TestMyLogic(t *testing.T) {
    framework.
        NewTest(t).
        Run(func(ctx framework.TestContext) {
            // Create the components.
            g := galley.NewOrFail(ctx, ctx, galley.Config{})
            p := pilot.NewOrFail(ctx, ctx, pilot.Config {})

            // Apply configuration via Galley.
            ctx.ApplyConfigOrFail(ctx, nil, mycfg)

            // Wait until Pilot has received the configuration update.
            p.StartDiscoveryOrFail(t, discoveryRequest)
            p.WatchDiscoveryOrFail(t, timeout,
                func(response *xdsapi.DiscoveryResponse) (b bool, e error) {
                    // Validate that the discovery response has the configuration applied.
                })
            // Do more stuff...
        }
}
```

Components are maintained in components package, which defines various Istio components such as galley, pilot, and namespaces.

Each component defines their own API which simplifies their use from test code, abstracting away the environment-specific details

Writing Components - 00

Getting Started

1. Create a new go package in **pkg/test/framework/components**

```
$ cd ${ISTIO}/pkg/test/framework/components
$ mkdir mycomponent
```

2. Within that package, define your component's API

```
package mycomponent

type Instance interface {
    resource.Resource

    DoStuff() error
    DoStuffOrFail(t test.Failer)
}
```

NOTE: A common pattern is to provide two versions of many methods: one that returns an error as well as an **OrFail** version that fails the test upon encountering an error. This provides options to the calling test and helps to simplify the calling logic.

Writing Components - 01

Implement Component - 00

3. Implement your component, both a native and Kubernetes version

```
package mycomponent

type nativeComponent struct {
    id resource.ID
    // ...
}

func newNative(ctx resource.Context) (Instance, error) {
    if config.Galley == nil {
        return nil, errors.New("galley must be provided")
    }

    instance := &nativeComponent{}
    instance.id = ctx.TrackResource(instance)

    //...
    return instance, nil
}

func (c *nativeComponent) ID() resource.ID {
    return c.id
}
```

Each implementation of the component must implement ***resource.Resource***, which just exposes a unique identifier for your component instances used for resource tracking by the framework. To get the ID, the component must call ***ctx.TrackResource*** during construction.

Writing Components - 02

Implement Component - 01

4. Provide an environment-agnostic constructor for your component:

```
package mycomponent

func New(ctx resource.Context) (i Instance, err error){
    err = resource.UnsupportedEnvironment(ctx.Environment())
    ctx.Environment().Case(environment.Native, func() {
        i, err = newNative(ctx)
    })
    ctx.Environment().Case(environment.Kube, func() {
        i, err = newKube(ctx)
    })
    return
}

func NewOrFail(t test.Failer, ctx resource.Context) Instance {
    i, err := New(ctx)
    if err != nil {
        t.Fatal(err)
    }
    return i
}
```

Writing Components - 03

Implement Component - 02

5. Using your component in test case

```
func TestMyLogic(t *testing.T) {
    framework.
        NewTest(t).
        Run(func(ctx framework.TestContext) {
            // Create the components.
            g := myComponent.NewOrFail(ctx, ctx)

            // Do more stuff...
        }
}
```

NOTE: When a component is created, the framework tracks its lifecycle. When the test exits, any components that were created during the test are automatically closed.

Running Tests - 00

- **Running Istio tests**

  Istio Test Framework is built on top of Golang's testing infrastructure, therefore, to run tests under /tests/integration/mysuite can be simply done by

  go run ./tests/integration/mysuite/...

- **Test Parellelism and Kubernetes**
  - Istio only supports one instance in each cluster
  - Multiple Istio instance in one K8s cluster may conflicts
    - Run one suite per command (e.g. go test ./tests/integration/mysuite/...)
    - Disable parallelism with -p 1 (e.g. go test -p 1 ./...). A major disadvantage to doing this is that it will also disable parallelism within the suite, even when explicitly specified via *RunParallel*

Running Tests - 01

❖Test Selection

When no flags are specified, the test framework will run all applicable tests. It is possible to filter in/out specific tests using 2 mechanisms:

- The standard *-run <regexp>* flag, as exposed by Go's own test framework
- *--istio.test.select <filter-expr>* flag to select/skip framework-aware tests that use labels

```
func TestMain(m *testing.M) {
    framework.
        NewSuite("galley_conversion", m).
        // Test is tagged with "Presubmit" label
        Label(label.CustomSetup).
        Run()
```

Then we can explicitly select execution of such tests using label based selection:

go test ./... --istio.test.select +customsetup
go test ./... --istio.test.select –customsetup
go test ./... --istio.test.select +customsetup,-postsubmit

Runnng Tests with Flags

Istio Test support platform Flags

❖ **Native** *--istio.test.env=native*

❖ **Kubernetes** *--istio.test.env=kube*

Flag	Default	Description
istio.test.env	`native`	Specify the environment to run the tests against. Allowed values are: `kube`, `native`. Defaults to `native`.
istio.test.work_dir	''	Local working directory for creating logs/temp files. If left empty, os.TempDir() is used.
istio.test.hub	''	Container registry hub to use. If not specified, `HUB` environment value will be used.
istio.test.tag	''	Common container tag to use when deploying container images. If not specified `TAG` environment value will be used.
istio.test.pullpolicy	`Always`	Common image pull policy to use when deploying container images. If not specified `PULL_POLICY` environment value will be used. Defaults to `Always`
istio.test.nocleanup	`false`	Do not cleanup resources after test completion.
istio.test.ci	`false`	Enable CI Mode. Additional logging and state dumping will be enabled.
istio.test.kube.config	`~/.kube/config`	Location of the kube config file to be used.
istio.test.kube.minikube	`false`	If `true` access to the ingress will be via nodeport. Should be set to `true` if running on Minikube.
istio.test.kube.systemNamespace	`istio-system`	Deprecated, namespace for Istio deployment. If '', the namespace is generated with the prefix "istio-system-".
istio.test.kube.istioNamespace	`istio-system`	Namespace in which Istio ca and cert provisioning components are deployed.
istio.test.kube.configNamespace	`istio-system`	Namespace in which config, discovery and auto-injector are deployed.
istio.test.kube.telemetryNamespace	`istio-system`	Namespace in which mixer, kiali, tracing providers, graphana, prometheus are deployed.
istio.test.kube.policyNamespace	`istio-system`	Namespace in which istio policy checker is deployed.
istio.test.kube.ingressNamespace	`istio-system`	Namespace in which istio ingressgateway is deployed.
istio.test.kube.egressNamespace	`istio-system`	Namespace in which istio ingressgateway is deployed.
istio.test.kube.deploy	`true`	If `true`, the components should be deployed to the cluster. Otherwise, it is assumed that the components have already deployed.
istio.test.kube.helm.chartDir	`$(ISTIO)/install/kubernetes/helm/istio`	
istio.test.kube.helm.valuesFile	`values-e2e.yaml`	The name of a file (relative to `istio.test.kube.helm.chartDir`) to provide Helm values.
istio.test.kube.helm.values	`''`	A comma-separated list of helm values that will override those provided by `istio.test.kube.helm.valuesFile`. These are overlaid on top of a map containing the following: `global.hub=${HUB}`, `global.tag=${TAG}`, `global.proxy.enableCoreDump=true`, `global.mtls.enabled=true`, `galley.enabled=true`.

Diagnosing Failures

❖Working Directory

```
$ go test galley/... --istio.test.work_dir /foo
  ...

$ ls /foo
  galley-test-4ef25d910d2746f9b38/

$ ls /foo/galley-test-4ef25d910d2746f9b38/
  istio-system-1537332205890088657.yaml
  ...
```

❖Enabling CI Mode

go test pilot/... --istio.test.ci

❖Preserving State (No Cleanup)

go test pilot/... --istio.test.nocleanup

❖Additional Logging

go test ./... --log_output_level=mcp:debug

Thanks!