Ubuntu CI Engine Documentation

Release

Canonical CI Engineering Team

November 01, 2014

Contents

General purpose and usage reference documentation:

CHAPTER 1

Introduction

1.1 What is the Ubuntu CI Engine?

The Ubuntu CI Engine is the implementation of the CI Airline designed to perform continuous integration (CI) of source code and binary packages under a converged workflow. The project is being implemented in multiple phases.

1.1.1 Phase 0

Phase 0 is the implementation of the 'CI Core'. In the simplest terms, the Phase 0 system accepts a set of source packages and produces an Ubuntu image on which tests are executed. The results are then provided to the user via the web interface.

This is the basis on which later phases will be built. While Phase 0 only implements a small portion of the total CI Airline system, it does provide a useful system capable of performing the basic CI operations of building and testing.

Features

The essiential set of Phase 0 features include:

- CI from Debian source packages.
- Building of binary packages and complete images.
- Tests are executed on the produced image.
- Tests are defined via autopackage testing.
- Results are archived and retrieved via a web interface.
- A micro-service oriented architecture deployed in an OpenStack cloud environment.

Constraints

The Phase 0 implementation is constrained so that development can focus on a robust core of functionality that will be the basis for all future work.

- Tickets are processed serially. Pending tickets are queued and executed in FIFO order.
- The interface is limited to a command line interface (CLI) for creating tickets and a web interface for monitoring and viewing results.
- Build and test results and logs are provided as raw artifacts.

- Source packages are used as input.
- Binary packages and images are built from a default series and image.
- OpenStack cloud images are produced and used for testing.
- Tests are limited to autopackage tests defined in the source packages.

1.1.2 Future Phases

The plan for future phases is yet to be determined.

CHAPTER 2

Using the Ubuntu CI Engine

As the Ubuntu CI Engine evolves through development, the usage will become more user friendly and more robust.

2.1 Prerequisites

The Ubuntu CI Engine current only processes source packages as input. These source packages will be uploaded unmodified into a Launchpad PPA. As a result, the source packages used as input must be signed by a user with permission to upload to the PPAs used by the Ubuntu CI Engine. These teams will be managed through Launchpad team membership granted via a CI team member.

Launchpad Setup

Prior to creating a source package to upload and a ticket, a user will need setup a Launchpad account and OpenPGP key:

- Create an https://launchpad.net/ account.
- Create and upload an OpenPGP key to Launchpad. Instructions are available at https://help.launchpad.net/YourAccount/ImportingYourPGPKey.

Creating a Source Package

Once the account setup is complete, users can create source packages. A full tutorial on creating and modifying source packages are beyond the scope of this document. There are many Ubuntu development and packaging guides available including:

- http://packaging.ubuntu.com/html/
- https://wiki.ubuntu.com/UbuntuDevelopment

Setting up the system for ticket creation

The following steps are needed to be able to use the command line interface:

```
sudo add-apt-repository ppa:canonical-ci-engineering/ci-airline-phase-0
sudo apt-get update
sudo apt-get install uci-cli
```

2.2 Creating a Ticket

Ubuntu CI Engine requests, known as tickets, are created through a command line interface.

```
ubuntu-ci [-S|--url] create_ticket -t "Ticket name" -d "Ticket description" -b 123 -o user@example.co
```

This returns a ticket ID which can later be used to monitor and check for results. Once the ticket is created, the Ubuntu CI Engine does the rest. The ticket will be queued by the Ticket System and executed in FIFO (First In First Out) order. As the system is limited to processing one ticket at a time, it may take multiple hours for a single ticket to be completed.

2.2.1 Parts of a ticket submission

The general input specification for a ticket is one or more source packages and a list of binary package names to add to or remove from the image that will be generated. These are specified with the appropriate create_ticket arguments. The full set of required and optional create_ticket arguments are:

```
usage: ubuntu-ci [-h] [-v {1,2,3}] [-u URL] [-S]
                 {create_ticket,update_ticket,status,get_image} ...

optional arguments:
  -h, --help            show this help message and exit
  -v {1,2,3}, --verbosity {1,2,3}
                        Verbosity level; 1=errors only, 2=verbose output,
                        3=very verbose output
  -u URL, --url URL     Development ticket system url
  -S, --staging
  -t TITLE, --title TITLE
                        Ticket title
  -d DESCRIPTION, --description DESCRIPTION
                        Ticket description
  -b BUG, --bug BUG     Related bug number
  -o OWNER, --owner OWNER
                        Email address of the ticket owner
  -s SOURCES, --sources SOURCES
                        Path to source.changes files. Source package files
                        (e.g. source.dsc, source.orig.tar.gz, etc.) are
                        required to be in the same directory as their
                        respective source.changes.
  -w, --wait            Do not queue the ticket for processing.
actions:
  {create_ticket,status,get_image}
                        commands
  create_ticket         Create a new ticket
  update_ticket         Update an existing ticket.
  status                Get ticket status. Use no flags for all tickets
  get_image             Retrieve the image produced by a ticket.
```

Note: The -t and -d arguments can be specified with a quoted text string.

2.2.2 Ticket Defaults

Images are produced from a default source image and series, which is currently based on Ubuntu 13.10 (saucy).

2.2.3 Specification of a source package

A source package is a standard Debian source package which may optionally contain dep8 autopackage tests. If autopackage tests are defined, they will be used to validate the image that is produced. All tests must pass for a ticket to complete CI successfully.

Every upload of a source package must include a version bump.

2.3 Monitoring a Ticket

As a ticket progresses through the Ubuntu CI Engine, the status of the ticket is updated to reflect it's current processing step. This status can be checked via the web interface.

2.4 Collecting Ticket Results

Once a ticket is completed, it's status will be updated appropriately and all build and test results and logs will be available via the web interface.

2.5 Exploiting Test Results

The test runner produces results in the subunit v1 format.

The subunit stream can be downloaded from the web interface and processed locally in different ways.

Note that the subunit stream contains the dep-8 test results.

The subunit v1 format is text only and as such can be read but it's not especially user-friendly:

```
test: dsc0t-build
successful: dsc0t-build [ multipart
Content-Type: text/plain;charset=utf8
stderr
0^M
Content-Type: text/plain;charset=utf8
stdout
12^M
build: OK
run: OK
0^M
]
```

subunit provide filters to convert a stream into more readable outputs.

2.5.1 Converting to the python unittest format

```
$ subunit-1to2  <subunit.stream | subunit2pyunit
```

This filter is appropriate to inspect failures in tests and will produce (for example):

```
$ subunit-1to2  <fail.stream | subunit2pyunit
tests.test_pass
tests.test_pass ... ok
tests.test_fail
tests.test_fail ... FAIL
======================================================================
FAIL: tests.test_fail
tests.test_fail
----------------------------------------------------------------------
testtools.testresult.real._StringException: Traceback (most recent call last):
  File "tests.py", line 31, in test_fail
    self.assertTrue(False)
  File "/usr/lib/python2.7/unittest/case.py", line 424, in assertTrue
    raise self.failureException(msg)
AssertionError: False is not true


----------------------------------------------------------------------
Ran 2 tests in 0.001s

FAILED (failures=1)
```

2.5.2 Converting to the junitxml format

```
$ subunit-1to2  <subunit.stream | subunit2junitxml >results.xml
```

This produces a 'results.xml' file in the junitxml format.

CHAPTER 3

Ticket and general user interaction

3.1 General principles

Ticket itself

Ticket collects metadata about what components and branches to treat as feature branches. It will contain a description of this feature and an ETA (from the registrant). Other informations like against which image number we want to have our test running (generated from ticket creation time) and a proposed list of integration tests will also be provided. Finally, we select which ones we want to run against (automated ticket will have this pre-selected with a default per component).

We want as well the user to know about the status of their current delivery. This one can be a direct MP or a first-class ticket entered into the system.

The ticket will include evolving metadata with time and reflect dynamically the states related to it.

- what is building right now as per the ticket definition, and the different build status in the isolated environment on all those components.
- driving the build ordering as well if we have dependencies (but not strict bump) to help upstream only taking care of ABI bump at delivery time.
- getting a way to do a non change rebuild upload for one or more components.
- attaching source package instead of branches for features involving more than just projects we handle in bzr branches.
- knowing where we stands on all those feature branches and sources compared to latest in distro (if the source package isn't the latest version in the development version anymore)
- what MP are pending against this ticket, meaning against all attached feature branches.
- what delta do we have between the various feature branches in this ticket and their corresponding trunks.
- knowing if we are able to merge or not against those trunks. If we can merge at a T time (and tests are all passing), propose a way to merge trunks easily into those branches.
- after getting the progress on their build, attaching latest available specific image (3 images per ticket): feature branch + fixed image number, feature branch + latest available image, trunk + feature branch merged + latest available image.
- knowing what latest image number is available, be able to change with it if test on latest image passed.
- getting tests progress while they are run dynamically. Represents them clearly against those previous 3 image tests

- ensuring that involved parties like core-devs and design team are involved if the ticket needs their review. A packaging change will require core-devs to ack their change. The design team will be in the process if there is a design change involved. Those should work through credentials.
- CI general health and global warnings if needed
- status on the corresponding components health
- gives an easy way once all those criteria are met (third-party acking, everything built and all tests passing) to give a "go" to the engine to deliver those different trunks.
- show the progress on the merging back to trunk, building packages there, tests passing, migration in UNAPPROVED/NEW, -proposed, release pocket and close the ticket completely once the next image is kicked in. Demonstrate explicitly when something is blocking there the whole pipeline for other delivery.

Tickets interactions

We also want to be able to show where their ticket or MP is in the component queue, and what time they can expect (in average) before seeing it delivered.

Global view

Finally, in this global view, we want to show the health of all projects:

- seeing all components (projects/branches) that we have in the CI system with global/general metadata (what test environment is going to be used, what tests are associated with that components, number of tickets opened against them and so on)
- giving a view for the managers to see what ticket their team are working on, and what's the progress on them as well as global status (build/tests/ability to merge to trunk).
- if a direct commit to trunk blocked the project and that's the only way to fix it back (another direct commit to trunk), surface that. All other tickets being blocked by that state (as touching that same component) should reflect that info as well.
- when tickets are expected to be delivered (based on the ETA), so that we can identify hot spots (times where a lot of landing will happen simultaneously and will clash) and try to shuffle them around to not having them in one landing (eventually by a global override on all tickets)
- a single point to see across all projects where different teams need to assess/review before the delivery takes place (pending packaging changes triggering a core-dev review, design review needed)
- CI general health

CHAPTER 4

Non Functional Stats Service

The Non Functional Stats Service (NFSS) is composed of a few different parts:

- A python3 / pyramid app that exposes a RESTful interface to a postgres data store.
- A client-side javascript-based UI that talks to the above.

While the system is generally very simple, there are a few things that sysadmins ought to be aware of:

4.1 Generating Client Keys

Test data is submitted to the data store via the RESTful api, and this API call is secured with oauth. Client access keys need to be generated for every external client that wants to be able to post data into the database. This is achieved by changing to the nf-stats-service directory and running:

```
$ python3 -m nfss keys-add
```

This is an interactive script that will ask for client details, and finally will write a python script that can be used by the external client to insert data into the data store.

A list of client keys can be generated in a similar fashion:

```
$ python3 -m nfss keys-list
```

A specific client id can be revoked by specifying it's client access key like so:

```
$ python3 -m nfss keys-del cj2DriLAGxmxinDyJzvDVQVltRSLNI
```

(obviously the client key will change, this is just an example).

4.2 Cleaning the database

Part of the oauth authentication scheme involves storing nonce values in the database. In order to prevent this table from filling up, we install a daily cron job that cleans the database. This can be achieved manually by running:

```
$ python3 -m nfss database-clean
```

Although this should never need to be done manually, since the restish charm installs a cron daily job to run this command.

4.3 Defining your Graph

4.3.1 Outline

A graph definition consists of two files; a .js and a .html.

The javascript file defines a controller that handlers receiving the 'newdata' event and prepares the raw data for display.

The html file presents the massaged data to the user (for instance creating a table using ng-repeat directives).

The front end also needs to know about the graph definition and needs to be added to the 'definitions' datastructure found in graphing.module.js.

4.3.2 Develop Locally With Ease

Under normal conditions, the Same Origin Policy prevents Javascript-enabled websites such as this one from making AJAX requests to sites on other domains. This is an important security measure on the web, but unfortunately prevents us from being able to develop the graph definition files locally and test them with live data from the production server.

Fortunately, Chromium offers a way to disable the Same Origin Policy for development purposes, allowing you to develop your graph definition files locally while still making AJAX requests to the live production server, so you have live production data to display in your experimental graph.

All you have to do is launch Chromium with this command:

```
chromium-browser --disable-web-security /path/to/web_static/index.html
```

Obviously it is extremely important not to surf the wider internet while Chromium is in this mode, but it sure makes it easy to iterate rapidly on the graph definition files. Don't forget to close Chromium when you're done!

4.3.3 Naming your Graph Definition File

Graph definition files are stored under /web_static/graphs and when looking to load them, the web page will first attempt to load `your-project-name_your-test-name.html`, and failing to find that, it will then search for `your-test-name.html` and finally `your-project-name.html` before giving up.

When deciding which name to use for your graph definition, you need to consider the data structure output by your tests.

- If you have one project that has a number of different tests which all output the same data structure, you'll want to define `your-project-name.html` to render that graph data.
- If you have many projects all running one test which all output the same data structure, you'll want to define `your-test-name.html`
- If for some reason you have a combination of projects and tests which do not output a consistent data structure, then you have the ability to define graphs that are unique to each project+test combination, however this is discouraged because it will likely result in a high degree of code duplication between graph definitions. In this case use `your-project-name_your-test-name.html`

4.3.4 Data Processing

Alright, so you've got your data in the db and now you want to display a pretty graph of it? Great! First, you should read the files in web_static/graphs/ to get some examples of what graph definition files look like. Graph definition files can contain any arbitrary HTML, and can also contain any arbitrary AngularJS directives, such as ng-repeat.

The only hard requirement is that a controller defined in the javascript listens out for the 'newdata' event and handles the raw data in some way (otherwise it will never receive any data).

For instance the simplest controller + event handler is shown below (taken from the default graph definition code).

Note. these two examples are the complete code for the default graph definitions (default.html & default.js).

```
angular.module('NonFunctional.graphing')
.controllerProvider.register("defaultGraphData", ['$scope', '$rootScope',
    function($scope, $rootScope) {
        $scope.graphData = [];
        // Expose a couple of helper methods to the template.
        $scope.dateFormatter = dates.dateFormatter;
        $scope.dateParser = dates.dateParser;

        $rootScope.$on("newdata", function(event, data) {
            $scope.graphData = [{ values: data.data }];
        });
    }
]);
```

With this data handling in place we can display a simple list of the data like so:

```
<div ng-controller="defaultGraphData">
  <ol>
    <li ng-repeat="item in graphData[0].values">
      <b>{{dateFormatter()(dateParser(item.date_entered))}}:</b>
      <br />
      {{item.data}}
    </li>
  </ol>
</div>
```

This will result in the following output in the browser:

For a more thorough real-world example for handling data I'll use the bootspeed data. The JSON blob that you submit to the db looks like this:

```
{
    "image_release": "utopic",
    "image_build_number": "1:20140428:20140411.3",
    "kernel": 4.42,
    "xorg": 4.77,
    "kernel_init": 0,
    "desktop": 15.15,
    "plumbing": 8.43,
    "image_arch": "mako",
    "image_md5": "n/a",
    "ran_at": "2014-04-28 14:18:17.76909-04",
    "image_variant": "touch",
    "machine_mac": "",
    "machine_name": "mako",
    "boot": 32.77
}
```

Then the db will return that data wrapped with a little bit of metadata like this:

```
{
    "data": {
        "image_release": "utopic",
        "image_build_number": "1:20140428:20140411.3",
```

```
        "kernel": 4.42,
        "xorg": 4.77,
        "kernel_init": 0,
        "desktop": 15.15,
        "plumbing": 8.43,
        "image_arch": "mako",
        "image_md5": "n/a",
        "ran_at": "2014-04-28 14:18:17.76909-04",
        "image_variant": "touch",
        "machine_mac": "",
        "machine_name": "mako",
        "boot": 32.77
    },
    "date_entered": "2014-04-28 18:18:17.76909+00",
    "id": 21272
}
```

But, and here's the big trick, the d3 graphing library we're using doesn't like this data structure at all. As you can see, all the numbers we want to graph (kernel, xorg, desktop, plumbing) are all keys in the same object. D3 doesn't have any provision for having multiple data points within the same object, and instead requires a data structure that looks like this:

```
[
    {
        key: 'Kernel',
        values: [ { date_entered: "...", value: 1 }, { date_entered: "...", value: 2 }, ... ]
    },
    {
        key: 'Plumbing',
        values: [ { date_entered: "...", value: 1 }, { date_entered: "...", value: 2 }, ... ]
    },
    {
        key: 'XOrg',
        values: [ { date_entered: "...", value: 1 }, { date_entered: "...", value: 2 }, ... ]
    },
    {
        key: 'Desktop',
        values: [ { date_entered: "...", value: 1 }, { date_entered: "...", value: 2 }, ... ]
    }
]
```

So, in order to convert from the original data structure to the expected data structure you must write a function to massage the data into a usable state. Obviously, the function that converts **your data** into d3's graphable data structure is going to depend heavily upon the structure of your data, so there aren't any hard and fast rules I can give here. The example from bootspeed.html is a good starting point, which I will attempt to explain:

```
// setup event handling any new data and refreshing the graph.
$rootScope.$on("newdata", function(event, data) {
    // Massage the raw data into something usable by the tables/charts
    // (declared in the html).
    massageGraphData(data);
    // Refresh the graph once the new data is ready.
    $scope.api.refresh();
});

// Helper function used in massaging the data.
function isolator(key) {
    return function(item) {
```

```
        return {
            // using the dates service here that provides date manipulation
            // functions.
            date_entered: dates.dateParser(item.date_entered),
            timespan: Math.max(item.data[key], 0) }
    }
}

$scope.massageGraphData = function(blob) {
    $scope.rawData = blob;
    $scope.graphData = [
    {
        key: 'Kernel',
        values: blob.data.map(isolator('kernel'))
    },
    {
        key: 'Plumbing',
        values: blob.data.map(isolator('plumbing'))
    },
    {
        key: 'XOrg',
        values: blob.data.map(isolator('xorg'))
    },
    {
        key: 'Desktop',
        values: blob.data.map(isolator('desktop'))
    }
    ];
}
```

In this example, there are two functions, `isolator` and `massageGraphData`. The isolator is a meta-function (that is, a function that returns another function). The purpose of this, is that you can call it with the json key you want to extract from the larger blob, and it returns a function that is programmed to take a blob, return just that one key, along with the date_entered key, and nothing else (it strips non-essential data from the json blob). So the inner function inside `isolator` will return an object that only contains keys `date_entered` (which we graph on the X axis) and `timespan` (which we graph on the Y axis).

Note that there's nothing special about the name `timespan`. You can use whatever name makes sense for your data. The only important thing is that later on when you define your data accessors, you need to use the same name so that d3 can find your data within the structure.

We setup an event handler that will trigger whenever any new data is provided. The data supplied in this events corresponds to the complete json data blob returned by the REST API endpoint `/api/v1/:project_id/:test_id`.

Note. By default this is all of your data points from within the last 30 days, but that can be controlled by the `start_date` and `end_date` URL query parameters.

Next, there's the `massageGraphData` function. This function will take the raw `blob` data from the event handler and produce a usable subset of the data for graphing.

In particular, `blob.data` will be a list of objects that look like the second JSON example listed earlier in this document. As you can see I'm calling `blob.data.map` several times in `massageGraphData`. The `map` function iterates over every item in the list (eg, every data point), calls the function returned by `isolator`, and returns a new list with the successive values returned by the function returned by `isolator`. The end result of this is that `scope.graphData[0].values` is a list of objects which contain only `date_entered` and `timespan` keys, such that `timespan` refers to the `kernel` value from the original data blob. `scope.graphData[1].values` will be similar, but with the `plumbing` key instead of the `kernel` key, and so on.

Note: Assigning your `massageGraphData` function to the $scope like this allows you to write unit tests against

your controller so that you can prove that it works as expected.

4.3.5 Choosing your Chart Type

The next part of the graph definition is technically free-form HTML, although most likely you'll want to define some sort of chart or graph. Technically speaking, you can do absolutely anything you want with AngularJS directives. If you wanted to go totally crazy, you could create a scatter plot by defining your own SVG tag, and then using AngularJS' `ng-repeat` in order to create an arbitrary number of circles with your x, y, and radius values dropped in, however I don't generally recommend fiddling with SVG data directly because then you don't get nice things like labelled axes.

In general, you're going to want to use one of the pre-defined nvd3 charts, which you can read more about here:

http://krispo.github.io/angular-nvd3

I'll continue using the bootspeed graph as my example, which uses a stacked-area-chart, but you can also refer to app_startup_benchmark.html which defines a (non-stacked) line chart.

The setup for the graph in the html is pretty simple. Note the surrounding div that has a 'ng-controller', attribute. This tells angular that the bootspeedCtrl controller (defined in the js file) is the contoller backend for this div and will supply the 'options' and 'data' datastructures.

```
<div ng-controller="bootspeedCtrl">
    <nvd3 options="options" data="graphData" api="api"></nvd3>
</div>
```

The actual options for the chart are defined in the controller and is a dictionary containing the key/values for the chart.

```
$scope.options = {
    chart: {
        type: "stackedAreaChart",
        height: 400,
        margin: {left:100, top:10, bottom:40, right:100},
        x: modifiers.accessor('date_entered'),
        y: modifiers.accessor('timespan'),
        useInteractiveGuideLine: true,
        xAxis: {
            tickFormat: dates.dateFormatter(),
            staggerLabels: true,
        },
        yAxis: {
            tickFormat: modifiers.numberFormatter(',.2f', 's'),
            tickPadding: 10,
        },
    },
};
```

- `type` This is the most important option, without it nothing will show. In this example we are using a "stackedAreaChart". Please see this link for more options: http://krispo.github.io/angular-nvd3/.
- `height` and `margin` can be adjusted to your liking. Don't define a width because it's defined to be 100% in the CSS, which makes the most efficient use of the screen space.
- `x` and `y` tell d3 how to find the x and y values in the data structure you created. There's nothing special about the values `timespan` or `date_entered`, they just need to match what you defined in `isolator` example.
- `xAxisTickFormat` and `yAxisTickFormat` can be adjusted to your liking. If you don't like the default date format, you can pass in a printf-style date format string to the `dateFormatter()` function, or you can change it to the `numberFormatter()` if your X axis isn't time for whatever reason.

- `modifiers.numberFormatter()` (available in support.module.js) takes two arguments, the format string, and the units. In this case the numbers we're graphing are seconds, so 's' is passed in. This can be any arbitrary string and is simply concatenated onto the end of the formatted numbers for display purposes only. You can read more about the format string mini-language here:
  - https://github.com/mbostock/d3/wiki/Formatting#d3_format
  - https://docs.python.org/release/3.1.3/library/string.html#formatspec
  - support.module.js for other date and data modifier methods

  So for example I'm using `,.2f` here, which means "round the number to two decimal places, and use a comma as the thousands-separator".

4.3.6 Raw Data Table

If you would like to display a raw data table beneath your graph, you can pretty well copy & paste this exact snippet into your code:

```
<table>
    <tr ng-repeat="series in graphData | reverse">
        <td><b>{{series.key}}:</b></td>
        <td ng-repeat="item in series.values">{{numberFormatter(',.2f', 's')(item.timespan)}}</td>
    </tr>
    <tr>
        <td><b>Date Entered:</b></td>
        <td ng-repeat="item in graphData[0].values">{{dateFormatter(ISO_ISH)(item.date_entered)}}</td>
    </tr>
</table>

// You need to expose the helper methods to the scope (within the
// controller):
$scope.ISO_ISH = dates.ISO_ISH;
$scope.dateFormatter = dates.dateFormatter;
$scope.numberFormatter = modifiers.numberFormatter;
```

This example is using Angular directives to fill out data values into a literal HTML table, and does so in a way that doesn't hard-code any knowledge about the number of data series (aka lines) on the graph or their names. Notice how similar this snippet is between bootspeed.html and app_startup_benchmark.html (basically the only difference is `item.timespan` vs `item.delta`).

And again I'd like to emphasize the true arbitrariness of this HTML here. If you don't want the data table, don't include it. If you don't want the graph, don't include it! If you want to add some paragraphs explaining how to interpret the data, by all means, throw some `p` tags in there. Honey badger don't care.

If the default.js data handler does what you need for the data but you want to display it differently then you can re-use the default.js file, define your own html with whatever custom tables etc. you need. The trick to do this is when you add the graph definition to the definitions dictionary:

```
this.definitions = {
    // Existing default definition.
    'default': {
        'templateUrl': 'graphs/default.html',
        'deps': 'graphs/default.js'
    },
    // Defining your graph def here.
    'my_test_name': {
        'templateUrl': 'graphs/my-custom-test-display.html',
        'deps': 'graphs/default.js'  // <-- Note the use of the default controller here.
```

```
    },
};
```

Internal documentation:

CHAPTER 5

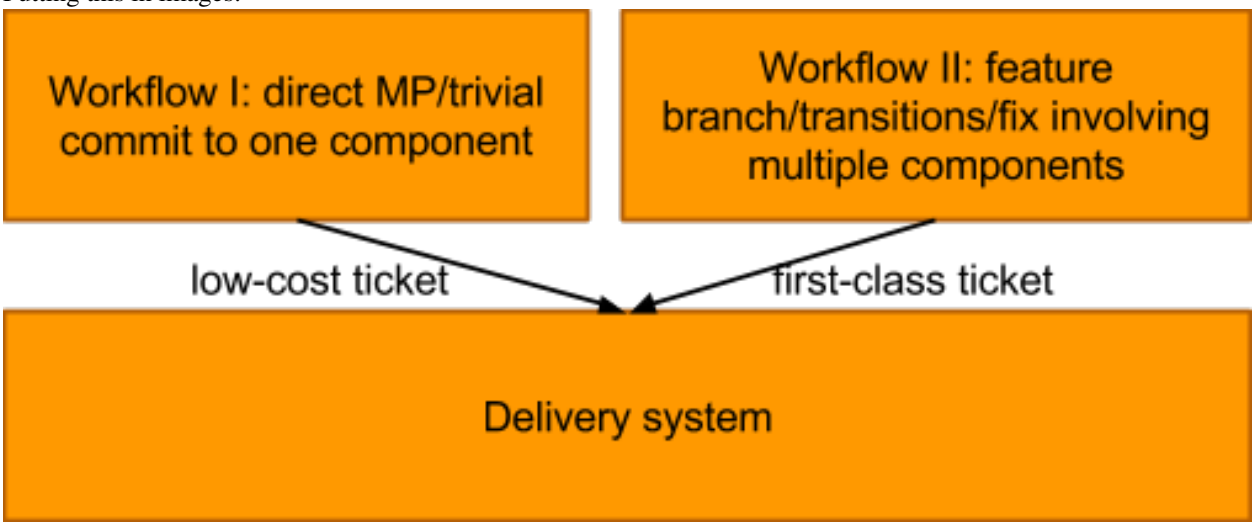
Workflow Diagrams

5.1 Introduction

The whole workflow is composed of 3 layers:

- One layer is responsible for delivering trunk to distro and handling the transaction to it. The delivery can be one or multiple components in one shot.
- The two other layers are the 2 different workflow to deliver a work. Either a MP to a project (we handle direct commit to trunk as well that way) or a feature branch that matured for a while. In any case, this is created as a ticket entry for the first base layer.

Putting this in images:



Components:

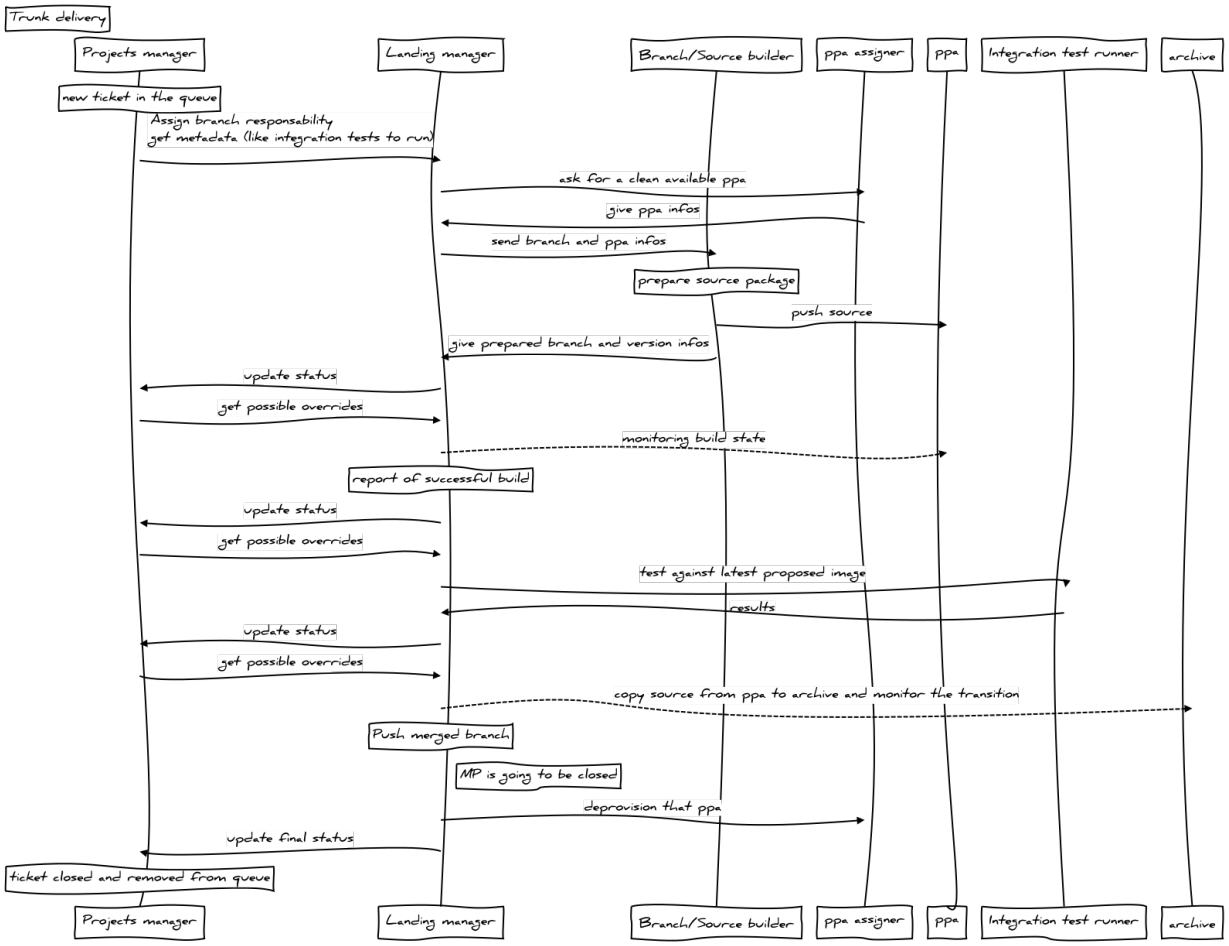
We can achieve all those functionalities with mainly 7 components:

- Projects manager
- Landing manager - Ticket manager (they do share most of their code)
- Branch listener
- Branch/Source builder
- ppa assigner
- Image builder

- Test runner

5.2 Delivery system

Case 1: delivering a ticket to trunk, (no pre-build package), success

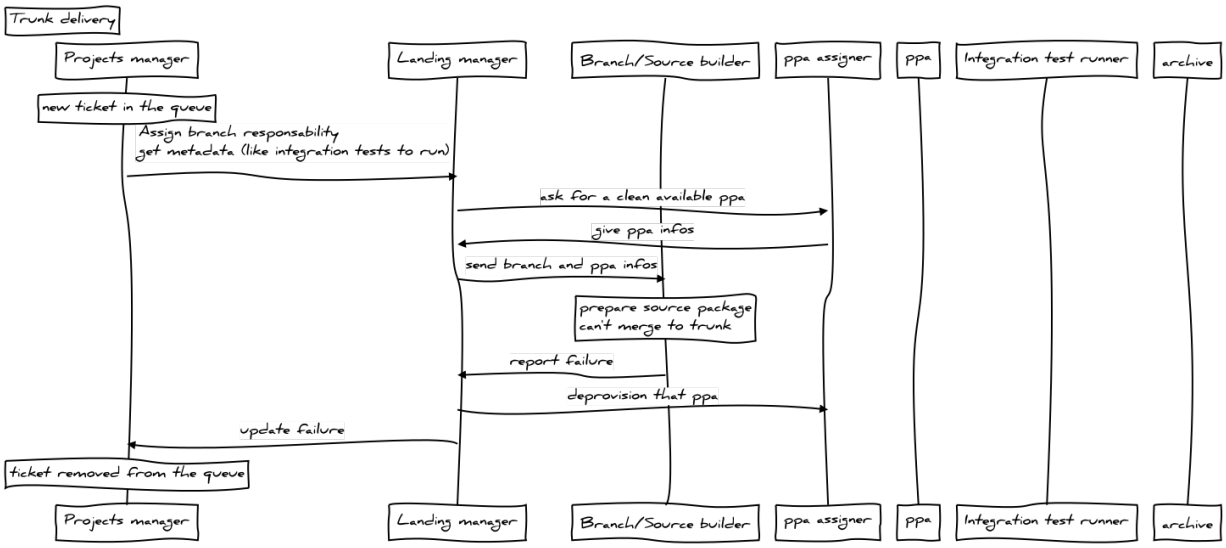


Note: in case of failure in any step, the Projects manager will comment on the MP and reject it.

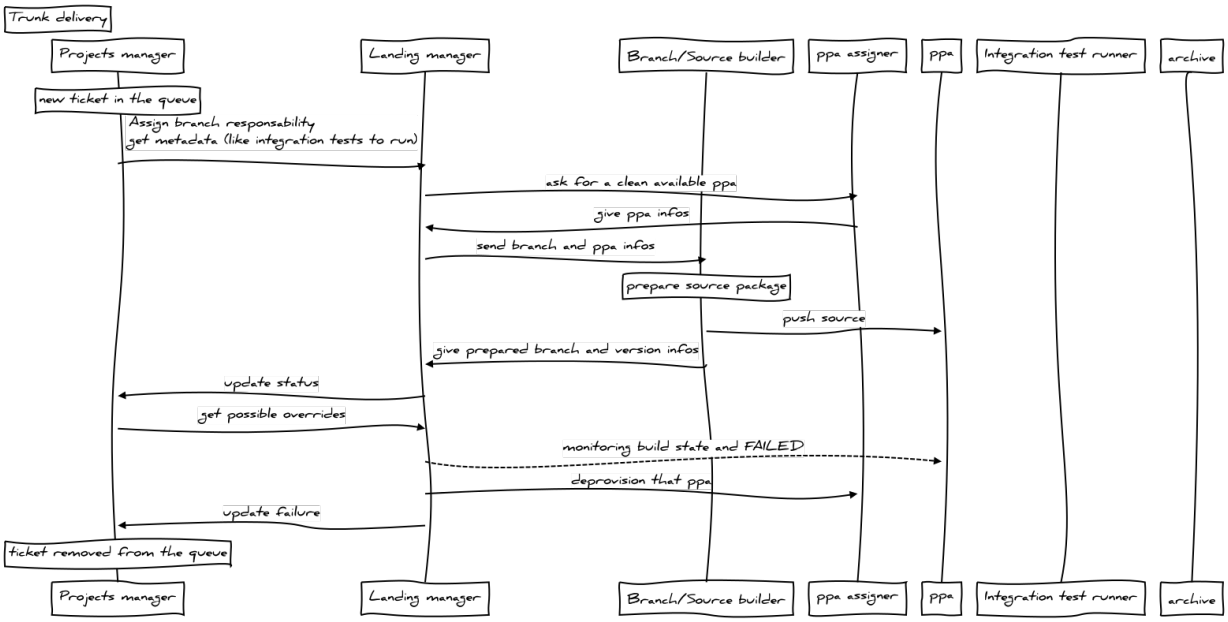
If direct push to trunk -> only set an error on the component and don't enable any other landing to it.

Note 2: while monitoring the ppa or tests, we still can get signal from Projects manager to Landing manager telling "ignore this arch" or "ignore this step". This impacts the landing manager on its view and can unblock/bypass some steps

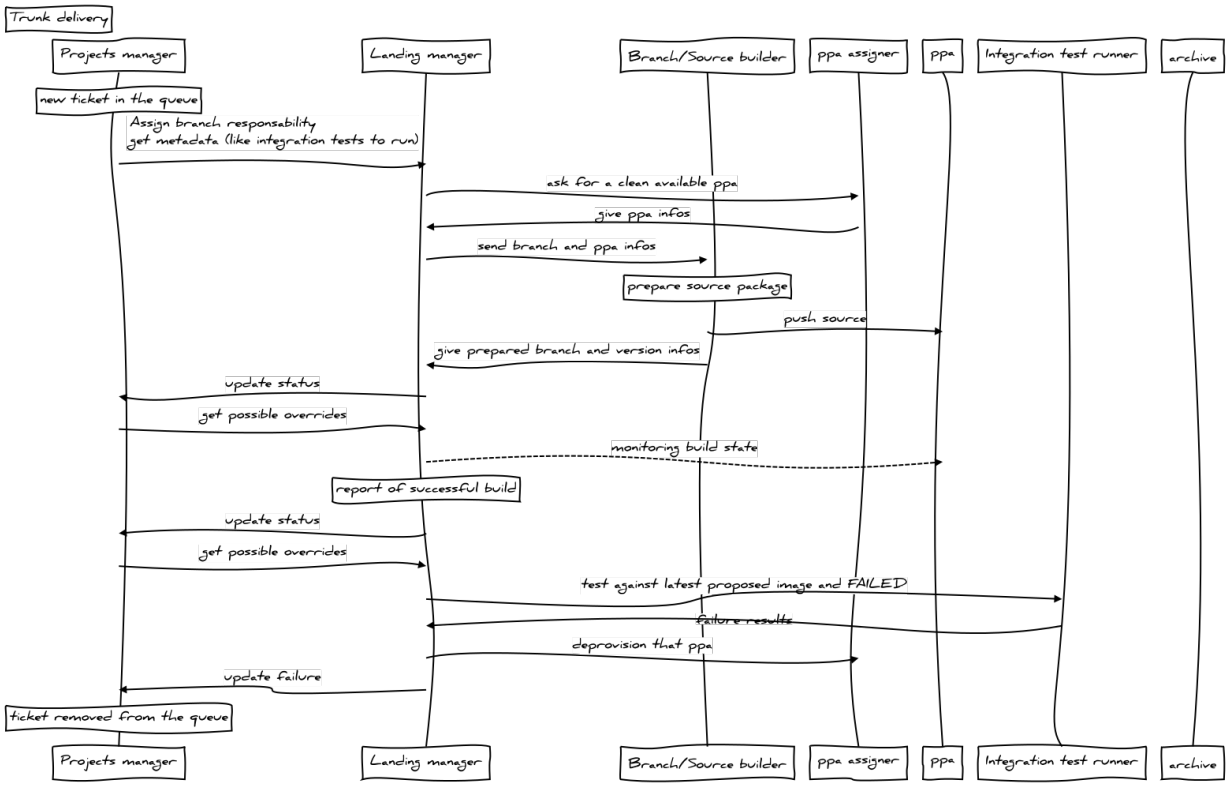
Case 2: delivering a ticket to trunk, merge fail



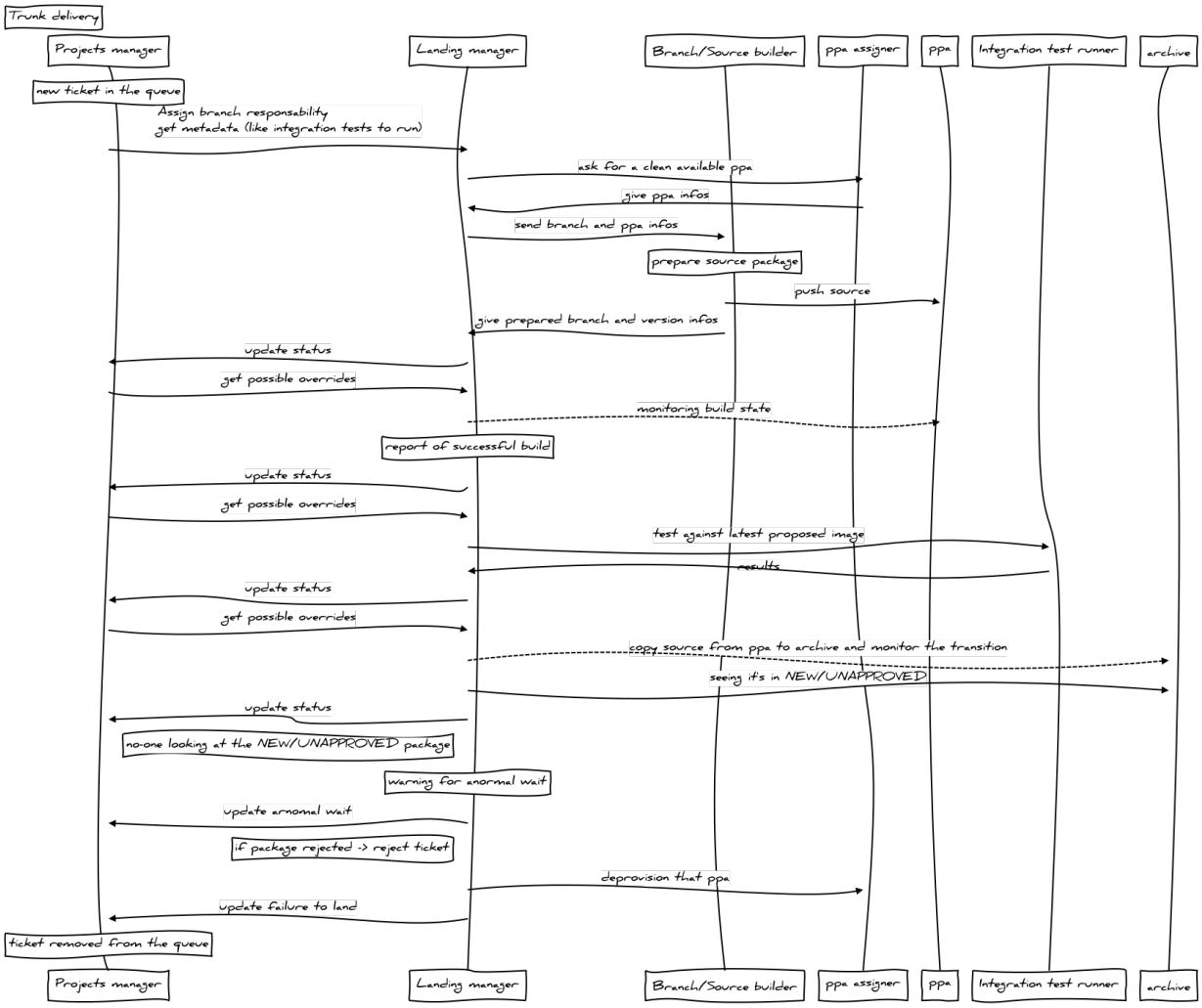
Case 3: delivering a ticket to trunk, build fail



Case 4: delivering a ticket to trunk, tests fail



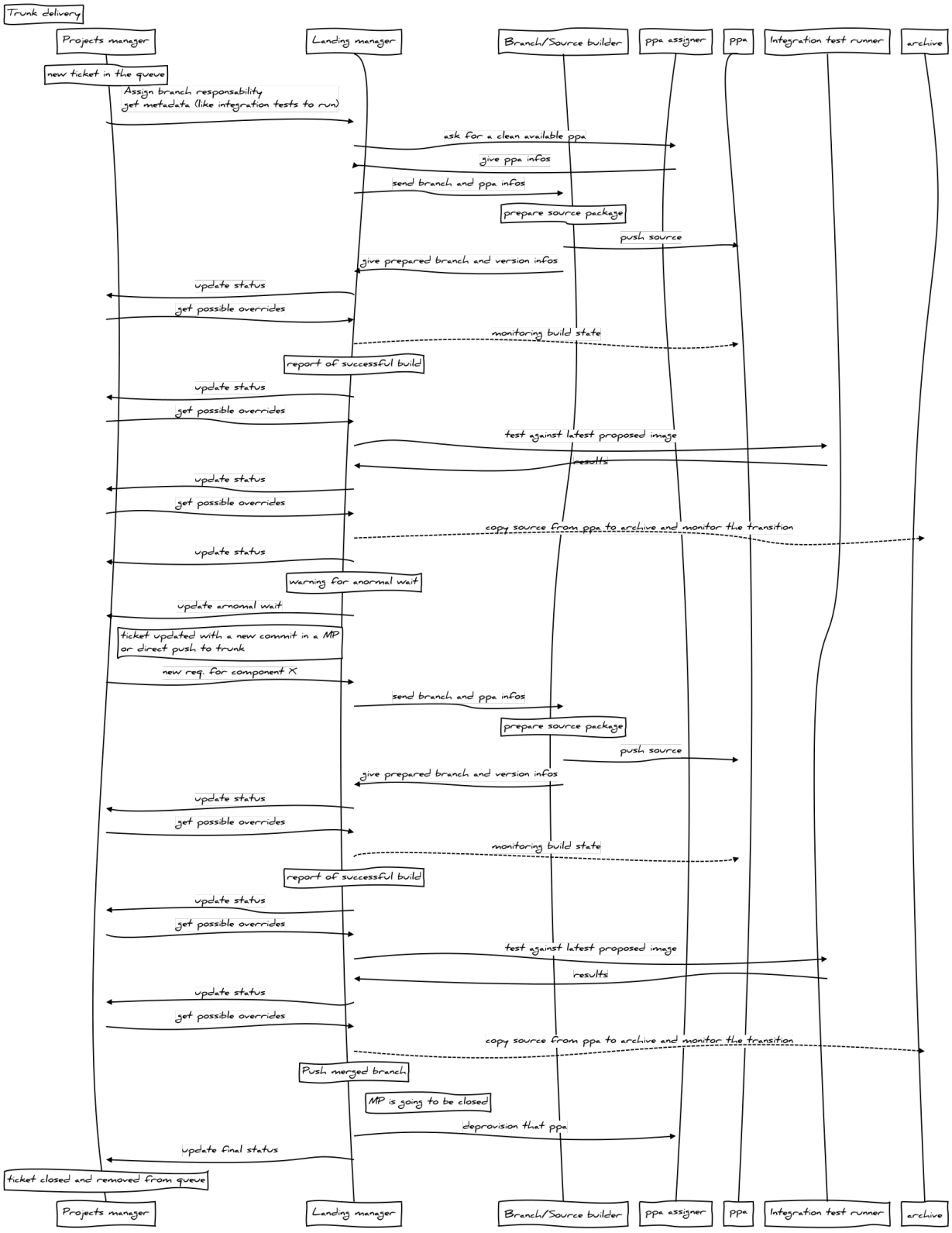
Case 5: delivering a ticket to trunk, blocked in UNAPPROVED/NEW



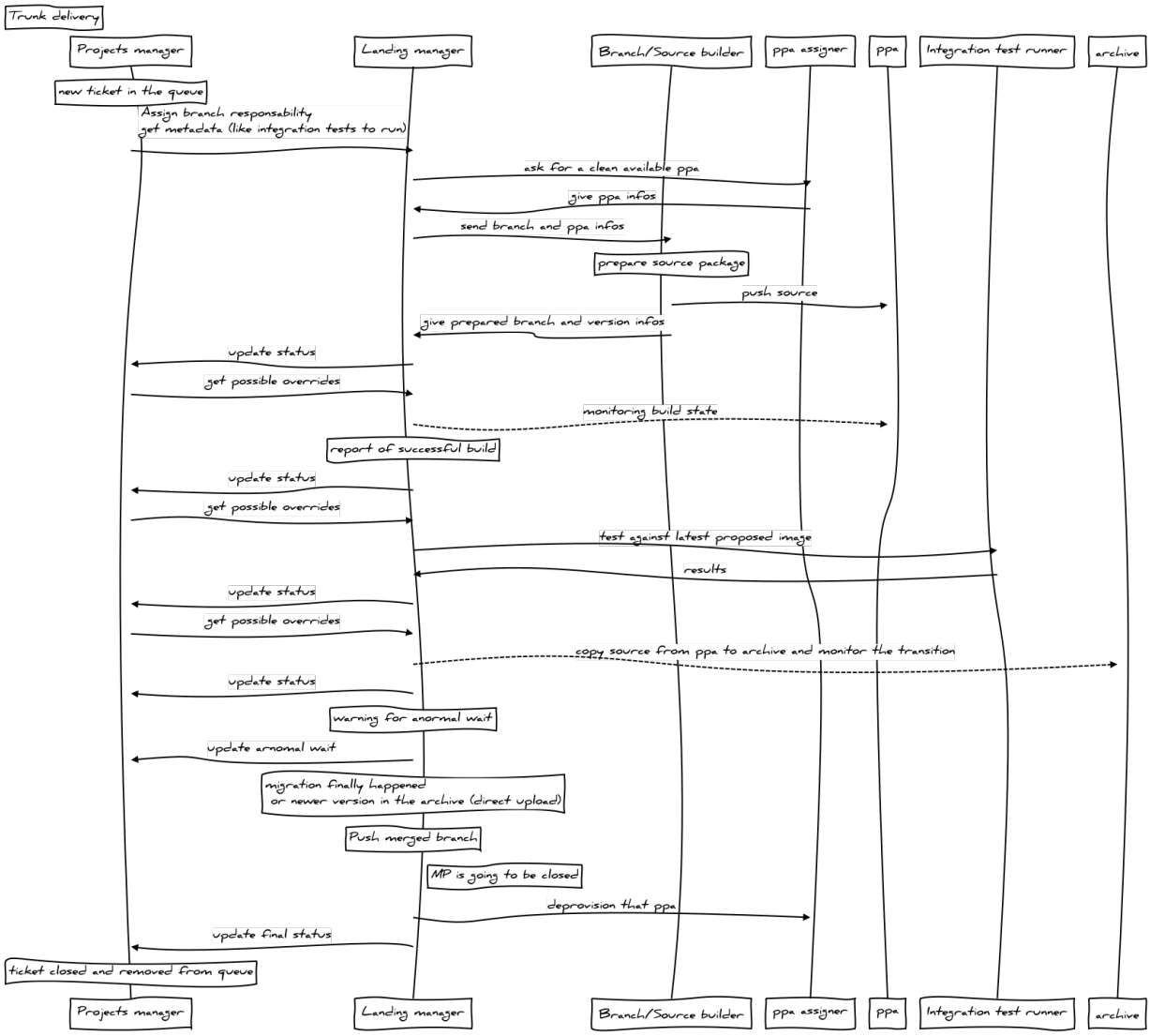
Case 6: delivering a ticket to trunk, blocked in proposed

Note: the package is in the archive at this moment. There is no way to backout the change, so the merge needs to go in one way or another. As the component is blocked in proposed, there is no advantage of unblocking the queue as blocked in proposed means that further unrelated landings will still be blocked in proposed. It will continue blocking potentially other unrelated packages (if a transition happens). The only way then is to unblock the package.

variant a: fix in the same component itself



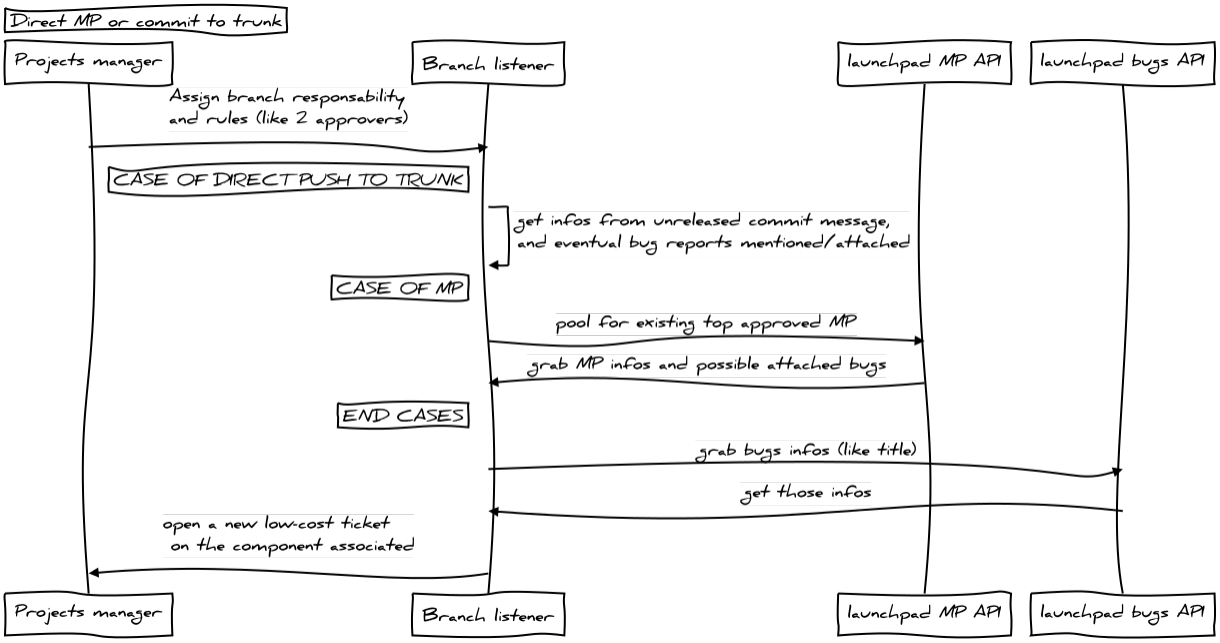
variant b: fix outside the CI system: in another component or by a direct upload to that component



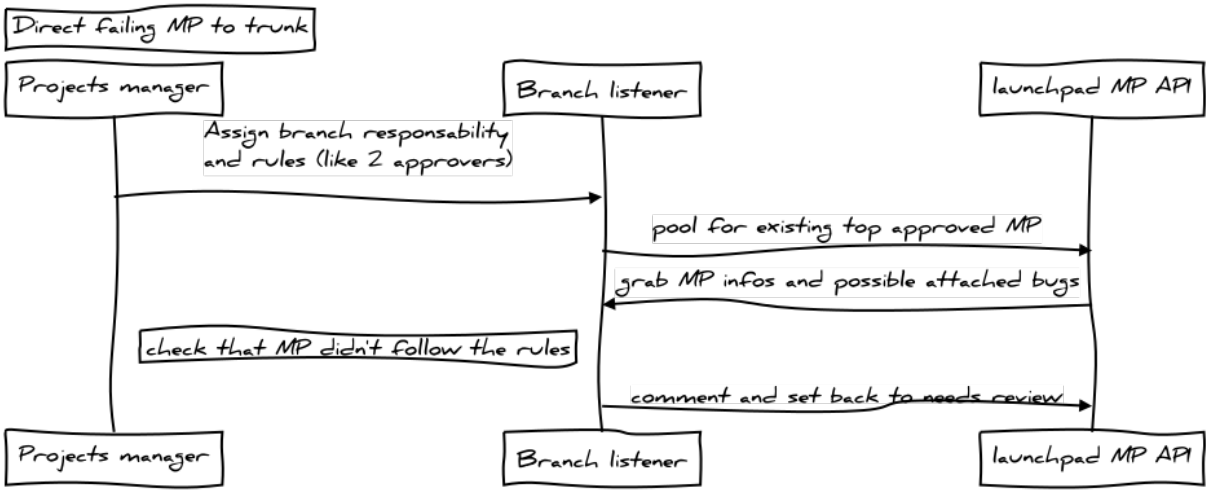
5.2.1 Workflow I: direct MP/trivial commit to one component (low-cost ticket)

Reminder: this is the case of a quick bug fix/feature (< 2 commits). The main case is manual ticket landing.

Workflow I A: direct commit to trunk or MP, all infos set



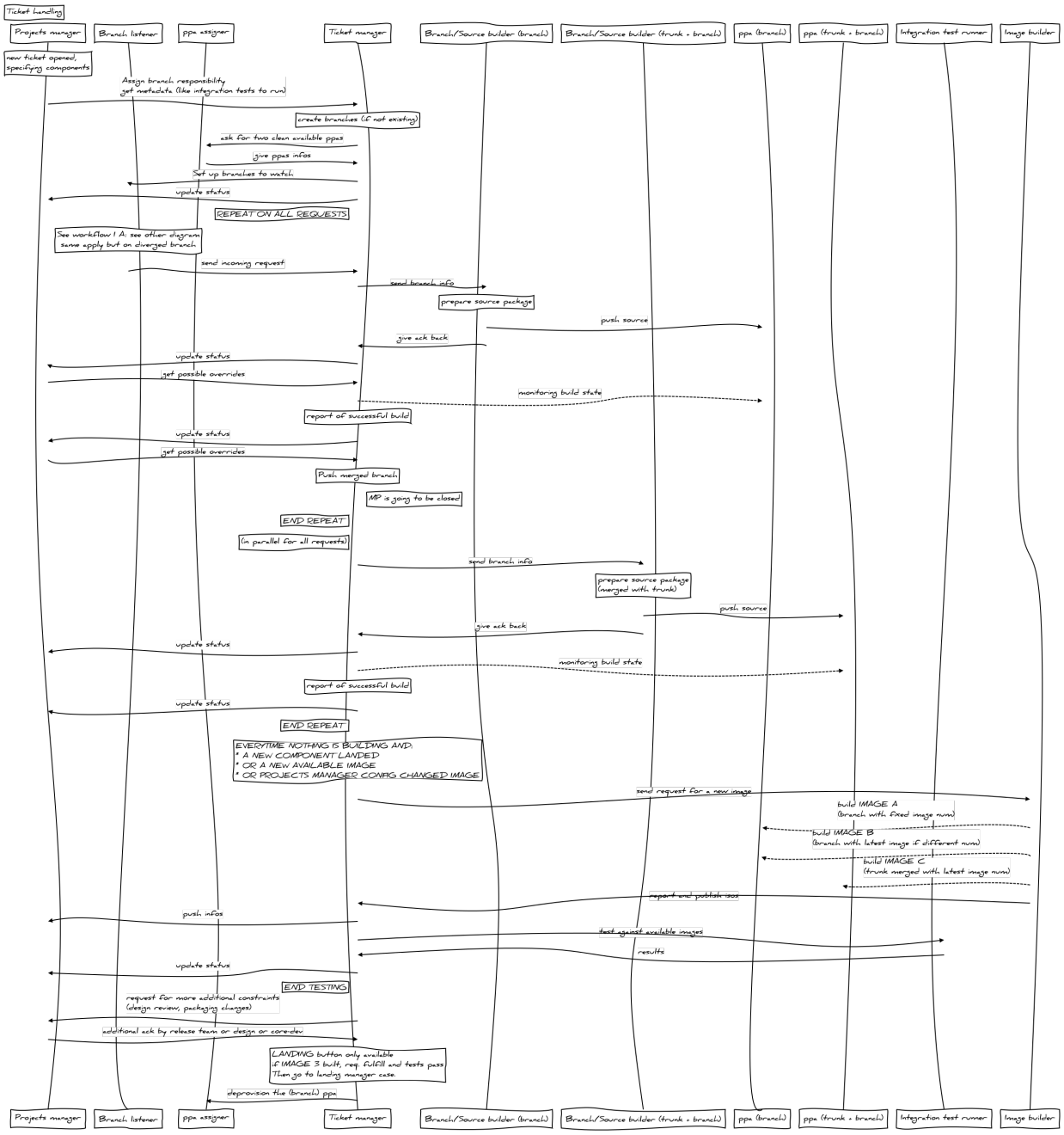
Workflow I B: direct MP, not following project rules (no commit message/enough approver)



5.2.2 Workflow II: opening a ticket

Notes: ticket manager and landing manager shares most of their code. Only the order and some interface changes.

Workflow II A: feature branch/transitions/fix involving multiple components (opening a first class ticket)



5.3 Components versus number of instances:

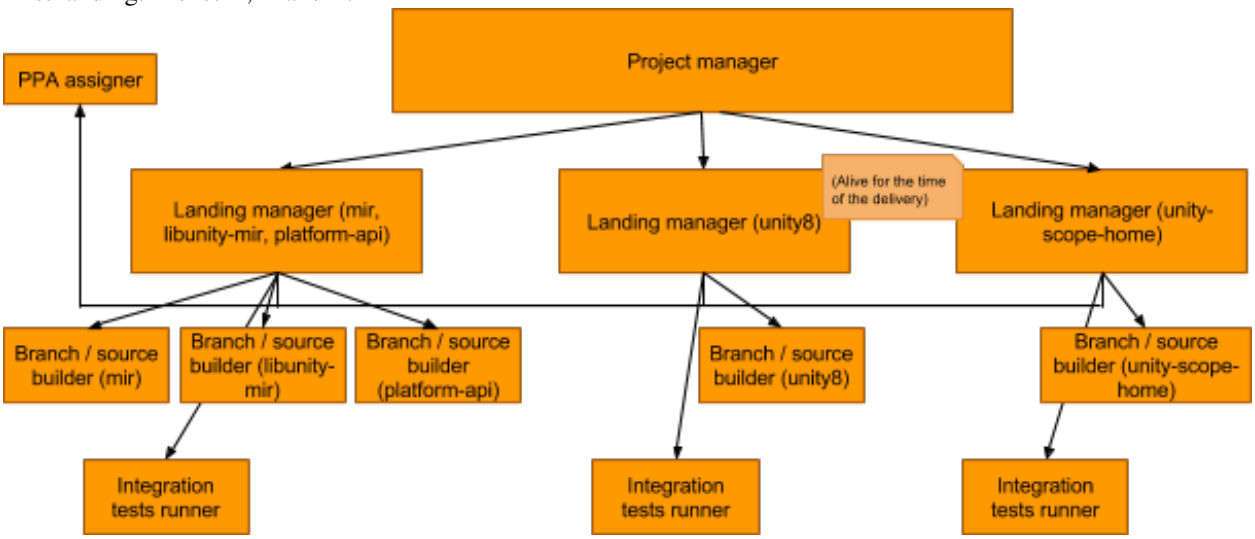
Please note that all services are *NOT* running at the same time. This diagram is just to show what is spawn by what and what takes care of one or multiple components.

Case delivering to trunk:

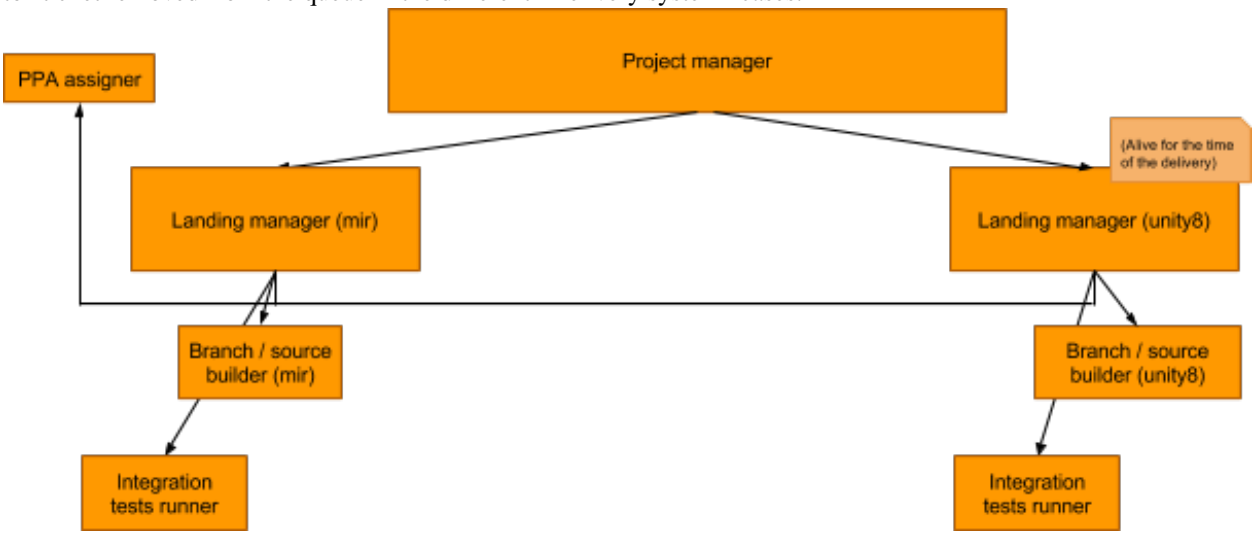
- one ticket A with mir, libunity-mir, platform-api
- one ticket B with libunity-mir only
- one ticket C with mir

- one ticket D with unity8
- one ticket E with unity-scope-home

First landing: Ticket A, D and E:

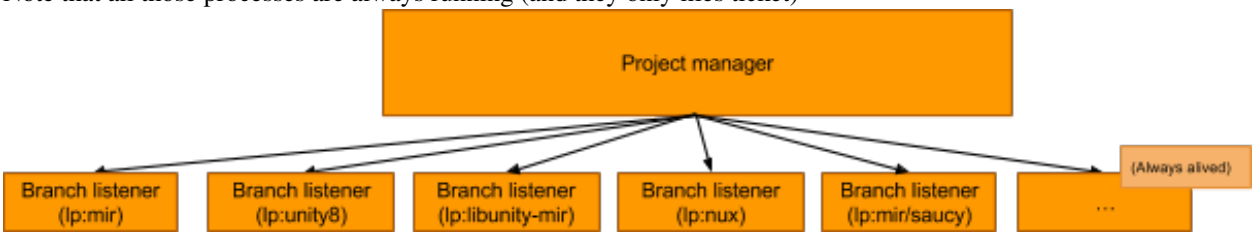


Second landing: Ticket B (starts as soon as A is treated) and D (starts as soon as D is treated) "Treated" corresponds to "ticket removed from the queue in the different "Delivery system" cases.



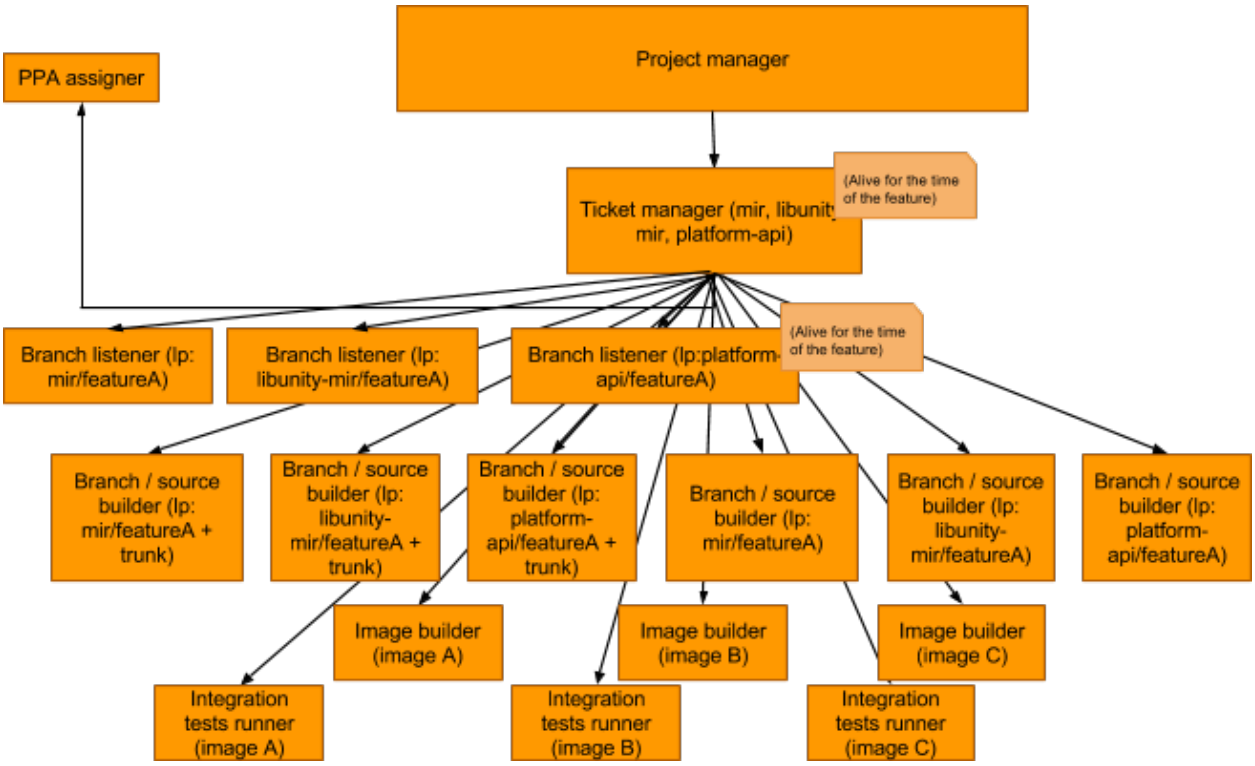
Workflow I

Note that all those processes are always running (and they only files ticket)



Workflow II

Feature involving mir, libunity-mir, and platform-api



5.3.1 Sources

Below are the sources processed at http://bramp.github.io/js-sequence-diagrams/ to produce the sequence diagrams above.

Case 1

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos

Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image
Test runner->Landing manager: results
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager–>archive: copy source from ppa to archive and monitor the transition
Note over Landing manager: Push merged branch
Note right of Landing manager: MP is going to be closed
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager: update final status
Note over Projects manager: ticket closed and removed from queue

Case 2

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package can't merge to trunk
Branch/Source builder->Landing manager: report failure
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager:update failure
Note over Projects manager: ticket removed from the queue

Case 3

Title: Trunk delivery
participant Projects manager
participant Landing manager

participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state and FAILED
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager: update failure
Note over Projects manager: ticket removed from the queue

Case 4

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build

Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image and FAILED
Test runner->Landing manager: failure results
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager: update failure
Note over Projects manager: ticket removed from the queue

Case 5

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image
Test runner->Landing manager: results
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager–>archive: copy source from ppa to archive and monitor the transition
Landing manager->archive: seeing it's in NEW/UNAPPROVED
Landing manager->Projects manager: update status
Note left of Landing manager: no-one looking at the NEW/UNAPPROVED package
Note over Landing manager: warning for anormal wait
Landing manager->Projects manager: update arnomal wait
Note left of Landing manager: if package rejected -> reject ticket
Landing manager->ppa assigner: deprovision that ppa

Landing manager->Projects manager: update failure to land
Note over Projects manager: ticket removed from the queue

Case 6

variant a

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image
Test runner->Landing manager: results
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager–>archive: copy source from ppa to archive and monitor the transition
Landing manager->Projects manager: update status
Note over Landing manager: warning for anormal wait
Landing manager->Projects manager: update arnomal wait
Note right of Projects manager: ticket updated with a new commit in a MP or direct push to trunk
Projects manager->Landing manager: new req. for component X
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides

Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image
Test runner->Landing manager: results
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager–>archive: copy source from ppa to archive and monitor the transition
Note over Landing manager: Push merged branch
Note right of Landing manager: MP is going to be closed
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager: update final status
Note over Projects manager: ticket closed and removed from queue

variant b

Title: Trunk delivery
participant Projects manager
participant Landing manager
participant Branch/Source builder
participant ppa assigner
participant ppa
participant Test runner
participant archive

Note over Projects manager: new ticket in the queue
Projects manager->Landing manager: Assign branch responsability get metadata (like integration tests to run)
Landing manager->ppa assigner:ask for a clean available ppa
ppa assigner->Landing manager: give ppa infos
Landing manager->Branch/Source builder: send branch and ppa infos
Note over Branch/Source builder: prepare source package
Branch/Source builder->ppa:push source
Branch/Source builder->Landing manager: give prepared branch and version infos
Landing manager->Projects manager:update status
Projects manager->Landing manager:get possible overrides
Landing manager–>ppa: monitoring build state
Note over Landing manager:report of successful build
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager->Test runner: test against latest proposed image
Test runner->Landing manager: results
Landing manager->Projects manager: update status
Projects manager->Landing manager:get possible overrides
Landing manager–>archive: copy source from ppa to archive and monitor the transition

Landing manager->Projects manager: update status
Note over Landing manager: warning for anormal wait
Landing manager->Projects manager: update arnomal wait
Note over Landing manager: migration finally happened or newer | version in the archive (direct upload)
Note over Landing manager: Push merged branch
Note right of Landing manager: MP is going to be closed
Landing manager->ppa assigner: deprovision that ppa
Landing manager->Projects manager: update final status
Note over Projects manager: ticket closed and removed from queue

Workflow I A

Title: Direct MP or commit to trunk
participant Projects manager
participant Branch listener
participant launchpad MP API
participant launchpad bugs API

Projects manager->Branch listener: Assign branch responsability and rules (like 2 approvers)

Note left of Branch listener:CASE OF DIRECT PUSH TO TRUNK
Branch listener->Branch listener: get infos from unreleased commit message, and eventual bug reports mentioned/attached
Note left of Branch listener:CASE OF MP
Branch listener->launchpad MP API: pool for existing top approved MP
launchpad MP API->Branch listener: grab MP infos and possible attached bugs
Note left of Branch listener:END CASES
Branch listener->launchpad bugs API: grab bugs infos (like title)
launchpad bugs API->Branch listener: get those infos
Branch listener->Projects manager: open a new low-cost ticket on the component associated

Workflow I B

Title: Direct failing MP to trunk
participant Projects manager
participant Branch listener
participant launchpad MP API

Projects manager->Branch listener: Assign branch responsability and rules (like 2 approvers)
Branch listener->launchpad MP API: pool for existing top approved MP
launchpad MP API->Branch listener: grab MP infos and possible attached bugs
Note left of Branch listener: check that MP didn't follow the rules
Branch listener->launchpad MP API:comment and set back to needs review

Workflow II A

Title: Ticket handling
participant Projects manager
participant Branch listener
participant ppa assigner
participant Ticket manager
participant Branch/Source builder (branch)
participant Branch/Source builder (trunk + branch)
participant ppa (branch)
participant ppa (trunk + branch)
participant Test runner

Note over Projects manager: new ticket opened, specifying components
Projects manager->Ticket manager: Assign branch responsibility get metadata (like integration tests to run)
Note over Ticket manager: create branches (if not existing)
Ticket manager->ppa assigner:ask for two clean available ppas
ppa assigner->Ticket manager: give ppas infos
Ticket manager->Branch listener: Set up branches to watch
Ticket manager->Projects manager:update status
Note left of Ticket manager: REPEAT ON ALL REQUESTS
Note over Branch listener: See workflow I A: see other diagram same apply but on diverged branch
Branch listener->Ticket manager: send incoming request
Ticket manager->Branch/Source builder (branch): send branch info
Note over Branch/Source builder (branch): prepare source package
Branch/Source builder (branch)->ppa (branch):push source
Branch/Source builder (branch)->Ticket manager: give ack back
Ticket manager->Projects manager:update status
Projects manager->Ticket manager:get possible overrides
Ticket manager–>ppa (branch): monitoring build state
Note over Ticket manager:report of successful build
Ticket manager->Projects manager: update status
Projects manager->Ticket manager:get possible overrides
Note over Ticket manager: Push merged branch
Note right of Ticket manager: MP is going to be closed
Note left of Ticket manager: END REPEAT

Note left of Ticket manager: (in parallel for all requests)
Ticket manager->Branch/Source builder (trunk + branch): send branch info
Note over Branch/Source builder (trunk + branch): prepare source package (merged with trunk)
Branch/Source builder (trunk + branch)->ppa (trunk + branch):push source
Branch/Source builder (trunk + branch)->Ticket manager: give ack back
Ticket manager->Projects manager:update status
Ticket manager–>ppa (trunk + branch): monitoring build state
Note over Ticket manager:report of successful build

Ticket manager->Projects manager: update status
Note left of Ticket manager: END REPEAT

Note over Ticket manager: EVERYTIME NOTHING IS BUILDING AND: * A NEW COMPONENT LANDED * OR A NEW AVAILABLE IMAGE * OR PROJECTS MANAGER CONFIG CHANGED IMAGE#
Ticket manager->Image builder: send request for a new image
Image builder–>ppa (branch): build IMAGE A (branch with fixed image num)
Image builder–>ppa (branch): build IMAGE B (branch with latest image if different num)
Image builder–>ppa (trunk + branch): build IMAGE C (trunk merged with latest image num)
Image builder->Ticket manager: report and publish isos
Ticket manager->Projects manager: push infos
Ticket manager->Test runner: test against available images
Test runner->Ticket manager: results
Ticket manager->Projects manager: update status
Note left of Ticket manager: END TESTING

Ticket manager->Projects manager: request for more additional constraints (design review, packaging changes)
Projects manager->Ticket manager: additional ack by release team or design or core-dev
Note over Ticket manager: LANDING button only available if IMAGE 3 built, req. fulfill and tests pass Then go to landing manager case.
Ticket manager->ppa assigner: deprovision the (branch) ppa

CHAPTER 6

Component Specification

6.1 Existing Component Pieces

Here's a list of what we have available today that might be of use in the CI Airline system.

Branch/Source Builder

- Build to PPA and monitor - lp:cupstream2distro-config already does this for the daily release process.
- Code coverage - lp:pbuilderjenkins has hooks to enable code coverage
- Static code checkers - lp:pbuilderjenkins has hooks to perform license, pep8 and pyflakes checks

Lander

- Copy from PPA to archive - lp:cupstream2distro-config already does this for the daily release process. (? not 100% sure about this, need to ask vila/didrocks)

Project Manager

- Project specific configuration - We are doing this now with lp:cupstream2distro-config but with a very Jenkins focused implementation.

Test Runner

- Test full touch images - lp:ubuntu-test-cases/touch?
- Test custom touch images based on PPAs and packages - lp:ubuntu-test-cases/touch (? need to ask doanac/plars)

Image Building

- Didier has done this manually for ISO images, using a squash FS process. Need to automate this and modify to work with touch images (which have an Ubuntu and platform specific component). (takes about 15 minutes)

Ubuntu CD Image & Germinate

- Believe using this process will be too time consuming to be used to generate the test images. Plan is to use the Image Building step above.

6.2 Launchpad Components

6.2.1 Merge Proposal

Purpose:

- A merge proposal in launchpad, contains the changes to be built, tested and eventually merged into trunk.

Deployment:

- This is already provided by Launchpad

Interactions:

- Trunk Delivering System - serializes the build and testing of MPs destined for trunk and performs the actual merge. Sends feedback to the MP on results of build and testing.
- Ticketing Tracker - the tracker monitors MPs that are part of a ticket request.

6.2.2 BZR Repos

Purpose:

- Project branches created for isolation of new features. Created by the ticket system when a new ticket is created for each project involved. Will eventually be merged to trunk

Deployment:

- This is already provided by Launchpad

Interactions:

- Ticket Environment Setup - Creates and removes the bzr repos as they are needed.

6.2.3 PPAs

Purpose:

- A PPA in launchpad, encapsulates the build of a bzr branch. Stores the packages once they are built. Will contain a single source for 'direct to trunk' MPs or multiple sources for tickets.

Deployment:

- This is already provided by Launchpad

Interactions:

- Branch/Source Builder - supplies source packages to build
- Trunk Delivering System - copies successful packages from PPAs into distro archive
- Ticket environment Setup - creates a PPA to manage projects and builds under a ticket
- Image Builder and Store - pulls packages from the PPA for image generation

6.2.4 Distro Archive

Purpose:

- This is the package archive. Packages are copied to here from the branch/ticket specific PPAs once they are committed.

Deployment:

- This is already provided by Launchpad

Interactions:

- Trunk Delivering System - Copies packages from the PPAs to archive. Also monitors for successful delivery.

6.3 Planned Component Specification

6.3.1 Ticket System

Purpose

The Ticket System contains project-centric knowledge such as location of trunk names of integration tests, point of contact, etc. as well as the ticketing management for the CI Engine which includes the status of current MPs being built an

It maintains the following sets of data:

- The history and status of tickets (the ticket tracker).
- The definition and configuration of projects, their branches, source packages and produced binary packages (the project manager).
- The test database. (This will not be part of phase 0)

Deployment

- Can run as a juju service.
- Runs as a persistent service, when it goes down, the engine halts.
- State needs to serialized whenever it is updated, so that status can be regenerated on a restart.

Development Plan

Phase 0

- Define API.
- Define backend ticket storage.
- Define backend project specification storage.
- Handle status updates from the Landing Manager.
- Provide read API for the Web Server.

TODO:

- Bundle the ticket creation API
  - For speed, I have left the different APIs to fully create a ticket as separate. If/when there is time, it would be much improved to bundle this together into one single 'Bundle API'

Future

- Integrate with the Landing Manager, supply metadata to build and test a single project and generate an automatic low-cost ticket (phase 0 will accept only source package uploads directly, the branch polling for new merges and creation of
- Define backend test database storage, and if it should be a separated system component.

Interactions

Ticket System

- Provide API for Web UI
- Provide API for Landing Manager to update tickets and subtickets. Also provide API for any system component that needs information about a ticket or subticket.

Project Manager

- Provide API for any system component that needs information about a project/branch/merge proposal/source package/binary package.

Design

Project Manager

In phase 0 the project manager will store and manage information about:

- Source package upload: component that will be tracked by a subticket. Is a direct source package upload.
- Source package: a source package as found in the archive.

We'll be provided a list of binary packages that should be monitored and tested, and a script outside project manager will generate the respective sources list.

This data will be updated by submitting a file to project manager, in JSON format, that should contain the binary name, source package name and version per entry.

For next phases, we'll extend it to store info about:

- Binary package: a binary package as found in the archive, produced by a known source package.
- Upstream project: a launchpad project, that might have branches.
- Bazaar branches: launchpad bazaar branches, might be associated with a source package. A branch can produce only one source package at a time.
- Merge proposal: a launchpad merge proposal, that will be tracked by a subticket.

Ticket System

Tickets and subtickets A ticket is a way of grouping parts of a bug fix or feature. In the ticket you'll describe the objective, add relevant information like blueprints or other references and the ticket status will reflect the status of its subtickets. Other system components will rely on the ticket status to take actions.

A subticket refers to a source package upload or a merge proposal.

When a developer submits the package upload request, the ticket system creates a subticket for each uploaded package and/or merge proposal, and a ticket that contains all of them.

In phase 0, it won't be possible to add more subtickets to a ticket after ticket is created (no editing). To create a ticket with several parts, the developer needs to submit all of them at once. If a subticket fails, the developer needs to submit another request that will generate another ticket with a subticket.

The ticket and subtickets will keep only the current step of the workflow they are and their status, as defined in ticket and subticket models.

In phase 1, if the subticket processing fails, the subticket is closed as failed, and the developer can submit another source package upload to the same ticket. It'll also be possible to add merge proposal subtickets.

Ticket System features Create, monitor and manage ticket related activities

- Define and create a ticket including:
- describe the targeted feature
- Links to any bugs/BPs/other design docs that are related
- People/teams working/owning on the ticket
- Components involved (packages/PPAs/feature branches/infrastructure required to code/test/build/deploy)
- PPA(s) dedicated to the ticket
- Describe the test suites that need to be run (smarts can be added in the long run to assist users in choosing the Right test to run). In phase 0 it consists on the dep8 tests in all source packages that are relevant.
- Knowing what image is supposed to be 'frozen' for developing against
- Provides test and status of a ticket:
- Built images in the image store
- Status of the ticket as it progresses through the CI engine
- Clear indication of merge/build/test failures and where the failure occurred, make it obvious to developers what a corrective course of action should be
- Provide a means to execute a landing when all criteria have passed
- Provides ticket management:
- list and status of all tickets

REST APIs

Tickets

List tickets List all open tickets.

```
curl --dump-header - http://localhost:8000/api/v1/ticket/
```

Create ticket Create a ticket from one or more source package uploads.

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"owner": "owner@example.cor
```

Get ticket *Full ticket*

Returns ALL information about the given ticket.

```
curl --dump-header - http://localhost:8000/api/v1/fullticket/1/
```

Model info

Return the ticket model info only for the given ticket.

```
curl --dump-header - http://localhost:8000/api/v1/ticket/1/
```

Get open tickets Show a list of all open tickets.

```
curl --dump-header - http://localhost:8000/api/v1/opentickets/
```

Get ticket status Get the status of all tickets or tickets for a specific status. (Mostly for the WebUI)

All

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/
```

Queued

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=100
```

Package Building

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=200
```

Image Building

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=300
```

Image Testing

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=400
```

Package Publishing

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=500
```

Failed

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=999
```

Complete

```
curl --dump-header - http://localhost:8000/api/v1/ticketstatus/?current_workflow_step=1000
```

Update ticket status To be used by the CLI and the lander

```
curl --dump-header - -H "Content-Type: application/json" -X PATCH --data '{"current_workflow_step": "
```

Mark ticket complete To be used by the lander.

```
curl --dump-header - -H "Content-Type: application/json" -X PATCH --data '{"current_workflow_step": "
```

Update subticket status To be used by the CLI and the lander

```
curl --dump-header - -H "Content-Type: application/json" -X PATCH --data '{"current_workflow_step": "
```

Create source package upload To be used by the CLI

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"sourcepackage": "/api/v1/s
```

Create artifact *Ticket*

Valid types are: "RESULTS", "LOGS", "IMAGE"

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"name": "my_artifact", "tic
```

Subticket

Valid types are: "SPU", "RESULTS", "LOGS"

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"name": "my_artifact", "sub
```

Create subticket To be used by the CLI

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"source_package_upload": "/
```

Project

Get source package return all source packages

```
curl --dump-header - http://localhost:8000/api/v1/sourcepackage/
```

Add source package This action would be completed by the CLI when it encounters a new source package that the ticket system hasn't had before.

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"name": "my-package"}' http
```

Get binary package return all binary packages

```
curl --dump-header - http://localhost:8000/api/v1/binarypackage/
```

Models

Ticket System will be django + REST + Postgres. The phase 0 models are defined below with the proposed future models coded out below that.

Phase 0

sourcepackage A source package is an existing source package in the Ubuntu archives. A source package record is created when a source package has changes submitted to the ticket system for the first time.

sourcepackageupload A 'Source Package Upload' is a file uploaded by the user that is related to the changes being made. It will have a related source package and artifacts.

ticket A ticket is created to get a change (or set of changes) into the Ubuntu archive. Creating a ticket allows the changes to be tracked through the processes of package building, image building, image testing and publishing as well as the results to be seen by the user.

subticket Each source package upload that is added to a ticket will have its own subticket. A ticket can contain one of more subtickets. The subticket allows the user to track the progress and results of the package building for each source package upload.

artifact Artifacts can be multiple things. When a ticket is initially created, it will have artifacts attached to it which are the source package upload files. Artifacts can also be test results and log files. They can be assigned to a relevant ticket or subticket based on the step of the process.

Future Apps/Models

person

```
class Person(models.Model):
    name = models.CharField(max_length=4096)
    email = models.EmailField(max_length=200)
    is_team = models.BooleanField(default=False)
```

testsuites

```
class TestSuites(models.Model):
    pass
```

project

```
class Project(models.Model):
    # Class that defines an upstream project.
    name = models.CharField(max_length=4096)
    display_name = models.CharField(max_length=4096)
    maintainer = models.ForeignKey("Person")
    contact = models.ForeignKey("Person")
    description = models.TextField()
```

branch

```
class Branch(models.Model):
    unique_name = models.CharField(max_length=4096)
    owner = models.ForeignKey("Person")
    type = models.CharField(choice=["trunk", "development", "maintenance", "regular"])
    project = models.ForeignKey("Project", null=True, blank=True)
```

binarypackage

```
class BinaryPackage(models.Model):
    name = models.CharField(max_length=4096)
    sourcepackage = models.ForeignKey("SourcePackage")
    seeded = models.BooleanField(default=False)
```

mergeproposal

```
class MergeProposal(models.Model):
     branch = models.ForeignKey("Branch")
     submitter = models.ForeignKey("Person")
     status = models.CharField(choices=["work_in_progress", "needs_review", "approved", "rejected", "r
     lp_weblink = models.CharField(max_length=4096)
```

Person API's - TODO

add_person Add a person (or a team) to the database.

person

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"name": "Chris Johnston", '
```

team

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"name": "Canonical CI Engi
```

get_person *Return all persons*

```
curl http://localhost:8000/api/v1/person/
```

search by name

```
curl --dump-header - http://localhost:8000/api/v1/person/?name__exact=My%20Name
curl --dump-header - http://localhost:8000/api/v1/person/?name__iexact=my%20name
curl --dump-header - http://localhost:8000/api/v1/person/?name__startswith=My
curl --dump-header - http://localhost:8000/api/v1/person/?name__istartswith=my
```

search by email

```
curl --dump-header - http://localhost:8000/api/v1/person/?email__exact=user@example.com
curl --dump-header - http://localhost:8000/api/v1/person/?email__iexact=User@example.com
curl --dump-header - http://localhost:8000/api/v1/person/?email__startswith=user
curl --dump-header - http://localhost:8000/api/v1/person/?email__istartswith=User
```

show/don't show teams

```
curl --dump-header - http://localhost:8000/api/v1/person/?is_team=True
curl --dump-header - http://localhost:8000/api/v1/person/?is_team=False
```

6.3.2 Web Server

Purpose:

- Provides human interface into the CI system.
- Create/manage/delete tickets
- Status of tickets
- Artifact access?

Deployment:

- Can run as a juju service.
- Needs a relationship to the Project Manager.
- Has no internal state, just provides a view to the data in the Project Manager.
- Provides public access with authorization to view private data.

Interactions:

- Project Manager - provides an interface for working with tickets and providing latest status.

6.3.3 Branch listener

Purpose:

- Track MPs and the trunks for the components involved in a ticket. tarmac-ish.

Requirements

https://bugs.launchpad.net/tarmac/+bug/1253770

- Determine when an MP has been updated and run tests accordingly.
- Given a trunk branch, return a list of all MPs available.
- Given an MP and a revision ID, check and merge.
- Apply tests to non-Approved branches and after any update/commit.

Tarmac meets most of our needs.

Service Requirements (Juju)

- Read access to LP.
- MP comment write access to LP.
- Tarmac installed (currently the stable release is being investigated but a fork might be in order)

Proposed API

Inbound

- check_branch/?branch=<branch>[&revision=<revision>]
- merge_branch/?branch=<branch>[&revision=<revision>]

Outbound

Currently all outbound APIs will be those tarmac already uses.

6.3.4 Lander

Purpose

Phase 0

Coordinates building of source packages into a full image, testing and publication of those packages and image upon successful completion of the tests.

Future

Expands upon Phase 0 by adding MPs as a build source and adds the ability to test packages without building a full image. It then coordinates the building and testing of MPs or source packages and publication into the archive. The Lander merges the MP after a successful publication while ensuring that MP's target branch has remained unchanged.

Deployment

Phase 0

- Can run as a Juju service.
- Needs relationship to Ticket System, PPA Assigner, Branch/Source Builder, Image Builder and Test Runner.
- No public access needed.

Future

- Incoming requests are queued and played back in the event of a restart. On restart, checks are made to determine the state of each request.

Interactions

Phase 0

- Ticket System - Provides build requests.
- Branch/Source Builder - Source packages are dispatched to the builder.
- PPA - Location where the package build will take place and published.
- PPA Assigner - Provides a single PPA to perform package builds.
- Image Builder - Builds a complete image from the PPAs and packages.
- Test Runner - Runs the specified test (if any) on the produced image.
- Data Store - The logs and artifacts are archived to the data store.

Future

- Ticket System - Provides meta information regarding the project owning a branch (i.e. what trunks to process, which tests to run, etc.). Status updates are sent to the project manager.
- LP Merge Proposal - MPs are the unit of work managed by the trunk delivering system, if the necessary criteria is in place, the MP will be merged to trunk.
- Branch/Source Builder - MP branches are dispatched to the builder.
- PPA - Location where the package build will take place.
- PPA Assigner - Provides a single PPA to perform package builds.
- Test Runner - Runs the specified test (if any) on the packages from the PPA.
- Image Builder - Builds a complete image from the PPAs and packages.

- Data Store - The packages are copied to the archive from the PPA on successful completion of testing.

Development Plan

Phase 0

- Define APIs.
- Deploy an instance.
- Allocate a PPA (interact with PPA Assigner).
- Build a source package in the PPA.
- Monitor a PPA for build status.
- Report status (interact with Ticket System).
- Build an image (interact with Image Builder).
- Initiate integration tests (interact with Test Runner).
- Publish packages from one PPA to another PPA.
- Handle binary build failures.
- Handle image build failures.
- Handle integration test failures.

Future

- Spawn an MP build (interact with Branch/Source Builder).
- Ensure that trunk has not changed during execution.
- Merge an MP to trunk.
- Binary copy a package from PPA to the archive.
- Monitor archive for status of package.
- Full end-to-end “Direct merge proposal to trunk” success case.
- Handle source build failures.
- Handle trunk changing during execution failures.
- Handle failure to merge.
- Handle push to archive failures.
- Handle component failure recovery/restart.

Design

Phase 0

A Lander service handles the workflow by using Jenkins to schedule individual tasks. When a request is received to the Lander from the Ticket System, it triggers the master Jenkins job with the request parameters. The master job then triggers a series of child jobs to execute the workflow.

The child jobs themselves execute a service handler. The service handler is responsible for setting up the progress queue and triggering the service via its REST API. As the service handler runs, it outputs the progress updates it receives to the console (to be viewed via Jenkins) and pushes the progress update to the Ticket System.

When the service completes, the service handler closes with a return code matching the status of the service itself. The result data from the service is archived as a Jenkins artifact. That data is combined with the existing set of job parameters to be used as input to the next Jenkins child job.

Only one master job may execute at a time. If additional build requests are received, they will be queued by Jenkins.

The Lander supplies regular notification events to the Ticket System while a job is executing. There are no notifications sent when the service is idle.

The Lander archives the Jenkins console logs and the archived results of each job to the data store.

Future

The Lander will support concurrent build requests as long as they don't have conflicting sources. Execution priority is given to first class ticket requests.

API

Phase 0

execute_request Schedules a new request for building source packages and creation and test of an image.

URL Pattern

http://lander-url:8080/api/v1/execute_request (HTTP POST)

Parameters

- ticket: The identification handle for a ticket request.
- source_packages: an array of data-store URLs to the source package files.
- binary_packages: an array of binary package names to use when constructing the image.
- series: the Ubuntu series to base the image (i.e. 'trusty')
- base_image: the image to use as source for the new image
- progress_trigger: the amqp exchange to use for sending progress events.

Example

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data  '{"ticket": "1234", "source_
```

status Returns the current status of the lander service and state of the currently running jobs for debugging purposes.

Url Pattern

http://lander-url:8080/api/v1/status (HTTP GET)

Example

```
curl --dump-header - http://lander-url:8080/api/v1/status
```

6.3.5 Branch/Source Builder

Purpose

Accepts one or more source package(s) and dispatches it to the provided PPA for building It then monitors the PPA for completion of the build and collection of results.

Deployment

- Standard Juju service with a juju-deployer config
- A service providing the REST interface
- A rabbitmq system where the service will send messages to a work queue. A configurable set of workers will then be able to process the messages. The queue should provide fault-tolerance so that requests will always be handled.
- No public access needed.

Interactions

- Lander sends a "build_source" request to the service (this includes swift URLs from the data-store on what to build)
- A request is placed in the queue (at each step below, the progress_trigger will be called to notify of status changes). * The source packages are dput to the PPA * The PPA is monitored for the completion of the build
- The Branch Source Builder sends progress updates back to the Lander. When the build is complete, the progress message will indicate completion and provide links to the build logs.

Service Design

The main web-service will be a stripped down REST-ful server providing:

REST APIs

build_source

Request a PPA build of the provided sources.

URL Pattern

http://bsbuilder-url:8080/api/v1/build_source (HTTP POST)

Parameters

- ticket_id
- cancel_url(can be null): a link to json url return {'building': true|false}
- source_packages: An array of data-store URLs containing all of the source package files.
- series
- ppa: The PPA allocated by the ppa-creator for this operation.
- archive

- progress_trigger: A string used to create a dedicated message queue between the Branch/Source builder and the build_source caller.

Example

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"ticket_id": 1, "cancel_url
```

Progress is communicated at regular intervals using the following messages:

Waiting

The build has not yet started.

```
message = {
    "state": "WAITING",
    "source_packages": source_packages,
    "ppa": ppa,
    "progress_trigger": progress_trigger
}
```

In progress

The source package(s) has been accepted by the PPA and are building.

```
message = {
    "state": "STATUS",
    "source_packages": source_packages,
    "ppa": ppa,
    "progress_trigger": progress_trigger
}
```

Completed

The build completed successfully.

```
message = {
    "state": "COMPLETED",
    "exit": True
    "source_packages": source_packages,
    "ppa": ppa,
    "progress_trigger": progress_trigger,
    "logs": ["https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1
}
```

Failed

The build failed.

```
message = {
    "state": "FAILED",
    "exit": True
    "source_packages": source_packages,
    "ppa": ppa,
    "progress_trigger": progress_trigger,
    "logs": ["https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1
}
```

status

Useful for debug and monitoring. This method will return information like the number of worker queues and if they are busy or not.

URL Pattern

http://bsbuilder-url:8080/api/v1/status (HTTP GET)

Example

```
curl --dump-header - http://bsbuilder-url:8080/api/v1/status
```

Worker Design

Steal logic from the daisy charm to set up a worker node. Then create a small python service using py-amqplib to pull messages off the queue. The service needs to respond to the caller's progress_trigger so it can track the state. The messages we should send are:

- STARTED - the message was pulled off the queue
- INPROGRESS - the source packages were dput to the PPA and its now building
- COMPLETE - the package has been built
- FAILURE - an error occurred at any step in the process

6.3.6 Image Builder

Purpose:

- Create a new image by:
- Downloading and mounting the base image
- Adding the requested PPAs and packages
- Repacking the image

Deployment

- Can run as a Juju service.
- Needs relationship to the Lander and the Data Store.
- Operations are transient, no need to save state.
- Will need to have some configuration data though, in particular, for Glance credentials
- No public access needed.

Interactions

- Lander - Get the PPA to use, list of packages, and URL to the base image. Send back the location of the rebuilt image.
- PPA - Pull previously built packages to add to the image.
- Data Store - Push rebuilt image. For cloud images, this will go to Glance rather than Swift.

REST API

status

Return the current status of the image builder and the state of currently running jobs.

Example

```
curl --dump-header - http://localhost:8000/api/v1/imagebuilder/status
```

build_image

Build an image including the requested PPAs and packages.

parameters:

- ticket_id : STRING
  - Ticket ID from the ticket system
- cancel_url(can be null): a link to json url return {'building': true|false}
- base_image : {"image_type": IMAGE_TYPE, "url_list": [...], "series": SERIES}
  - A json object containing the image_type (cloud for now), list of URLs pointing to the image artifact(s), and the Ubuntu series name.
- ppa_list : [...]
  - A list of PPAs to add to the image.
- package_list : [...]
  - A list of additional packages to install in the image.
- progress_trigger : STRING
  - The amqp exchange to use for sending progress events.

Example

```
curl --dump-header - -H "Content-Type: application/json" -X POST --data '{"base_image": {"image_type"
```

Development Plan

- Define APIs.
- Prototype the process.
- Rebuild cloud images.
- Push to glance.
- Support monitoring and progress requests.
- Juju deployment support.
- Rebuild touch images.
- Rebuild cd images.

6.3.7 Test Runner

Purpose

- Performs package testing on a given image.

Future

Expands upon phase 0 by allowing a set of PPAs and additional packages to be added before running the tests, support using autopilot on a touch device, perform a custom test (TBD), support executing tests on bare metal.

API for submitting test run requests

The 'Test Runner' exchanges requests/responses with the 'Lander'.

test_package

This run the DEP8 tests for each package in 'package_list' on 'image_id' and is sent by the 'Lander'.

```
test_image(ticket_id, image_id, package_list, progress_trigger)
```

The caller is responsible for providing a unique 'progress_trigger' identifier. This unique identifier is used to define the rabbit queue and the data store container.

This is composed of:

- boots the image from 'image_id',
- install each package from 'package_list' monitoring its execution and emitting progress messages (see below),
- collect the test results,
- send a 'done' message.

The 'progress' and 'done' messages are sent to a rabbit queue to communicate with the 'Lander'.

done

This ends the processing request and is sent to the 'Lander'.

```
done(progress_trigger, status=[FAIL, SUCCESS], test_result_urls)
```

'test_result_urls' is a list of subunit streams uploaded to the data store.

progress

This is sent to the 'Lander' to provide feedback during the test run.

```
progress(progress_trigger, message='% or ETA', current, total)
```

Based on test execution feedback, a progress message is sent at regular intervals until completion. 'total' is the number of tests to execute, 'current' is the rank of the test currently running. Both can be empty if this info is not available.

Deployment

- This can be deployed as a juju service, but the actual test runners may be on bare metal (vms only for phase 0).
- Needs relationship to the Lander and the Data Store.
- Operations are transient, no need to save state.

Interactions

- Lander - Supplies test requests and waits for results.

Development Plan

Phase 0

- Setup an instance from the image builder id
- Run the dep8 tests for a list of packages
- Collect the test results
- Send test results to the lander (via the data store).

Future

- Collect a set of artifacts (defined by the 'Lander')
- Execute test on bare metal (including touch device)
- Execute autopilot (requires an emulator or a graphic card)
- Execute arbitrary test (may require additional packages)

6.3.8 Queue Service

Purpose

Provides a generic queue service for the entire CI Engine. Common uses are:

- Task queues to manage work between a service accepting requests and a pool of workers to actually execute the request.
- Progress queues to provide progress updates between services.

Usage

Task Queues

The service and its pool of workers share a named queue defined in the service charm configuration files.

Progress Queues

The queue is created by the calling service and the details are passed to the callee. For example, if the Lander needs to receive progress from the Image Builder, the Lander creates the queue and passes it to the Image Builder as part of the Image Builder API.

Updates are sent every 60 seconds until the task being tracked is complete, at which point it can stop sending updates.

If no progress is received by the caller with the span of 10 updates, the caller can declare the callee unresponsive and can take action to retry or fail the original request.

Design

The RabbitMQ service is used.

A library API is provided to facilitate access.

API

TBD

6.4 Planned Library Utilities

6.4.1 Data Store

Purpose:

- Store artifacts and other file-based data.

Requirements

- Accept both private and public files.
- Give access via HTTP to public files.
- Provide a library based API for other components to use.

Proposed API

DataStore(component, identifier=None) Creates a data store object which is used to communicate with the swift service. The component and identifier are used internally to select the correct container in which to store the data.

list_files() List the files in the container

put_file(filename, contents) Add a file to the data store.

get_file(filename) Get a file from the data store.

change_visibility(filename, private=False) Used to make public files private and private files public.

delete_file(filename) Remove a file from the data store.

delete(recursive=False) Remove the container.

file_path(filename) Get the public url for a file.

clear() Remove all files from the container.

change_visibility(public=False) Make the container public/private.

Internal Methods

_create_container(container_name)

_get_container(component, identifier=None)

_get_container_url()

_get_file_name(filename)

_has_file(filename)

_setup_auth()

CHAPTER 7

CI Engine Service APIs

7.1 Branch listener

To Be Determined.

7.2 Examples

The section provides some examples of how the APIs would be used in various use cases.

7.2.1 Initial Successful Request

The developer is making the first request. The starting point is the base image and two empty PPAs.

User -> CLI

The user creates a request, providing a source package and binary packages to add.

```
python ubuntu-ci create_ticket -t "Ticket name" -d "Ticket description" -b 123 -o user@example.com -s
```

CLI -> Ticket System

The CLI creates a ticket through the ticket system, then pushes the source package files into the data store.

Create the ticket:

```
create_ticket {
    "owner": "default-user@example.com",
    "title": "Default title",
    "description": "Default description",
    "bug_id": "https://bugs.launchpad.net/bugs/1234567",
}
```

Returns:

```
"http://ticket-system-url:8000/api/v1/ticket/1/"
```

Create the source package upload:

```
create_source_package_upload {
    "sourcepackage": "/api/v1/sourcepackage/X/",
    "version": "1.5~dev.1"
}
```

Returns:

```
"http://ticket-system-url:8000/api/v1/spu/1/"
```

Create the sub-ticket for the source package:

```
create_subticket {
    "source_package_upload": "/api/v1/spu/1/",
    "ticket": "/api/v1/ticket/1",
    "assignee": "default-user@example.com"
}
```

Returns:

```
"http://ticket-system-url:8000/api/v1/subticket/1/"
```

For each source package file, upload to the data store:

```
data_store = DataStore("ticket.1", "sources", public=True)
while open("autopilot_1.5~dev.1_source.changes") as f:
    url = data_store.put_file("autopilot_1.5~dev.1_source.changes", f.read())
```

Returns (as 'url'):

```
https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1.sources/autopilot_1.5~dev.1_source
```

then create the artifact:

```
create_artifact {
    "name": "autopilot_1.5~dev.1_source.changes",
    "subticket": "/api/v1/subticket/1/",
    "reference": "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1.sources/autopilo
    "type": "SPU"
}
```

Returns:

```
"http://ticket-system-url:8000/api/v1/artifact/1/"
```

Ticket System -> Lander

The ticket system requests a build of the source packages and image through the lander:

```
execute_request {
    "ticket_id": "1",
    "source_packages": [
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1.diff
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1.dsc",
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1_sourc
    ],
    "binary_packages": ["python-autopilot", "python3-autopilot"],
    "series": saucy,
    "base_image": "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/images/saucy-desktop-amd
    "progress_queue": "ticket-1-exchange"
}
```

Returns TBD result.

Lander -> PPA Assigner

The lander requests a PPA to perform the build:

```
get_ppa {
   "ticket_id": 1
}
```

Returns:

```
"ppa:ci-team/ppa_build_1"
```

Lander -> Branch/Source Builder

The lander sends the source files to the ppa:

```
build_source {
    "source_packages": [
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1.diff
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1.dsc",
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/autopilot_1.5~dev.1_sourc
    ],
    "ppa": "ppa:ci-team/ppa_build_1",
    "progress_queue": "bsbuilder-1-exchange"
}
```

Returns:

Nothing (just a successful status code).

Branch/Source Builder -> Lander

Build progress messages are passed back via the progress queue:

```
TBD
```

Completion is signaled on the progress queue:

```
message {
    "state": "COMPLETED",
}
```

Lander -> Image Builder

The lander requests a new image from the image builder:

```
build_image {
    "base_image": {
        "image_type": "CLOUD",
        "url_list": [
            "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/images/saucy-desktop-amd64.iso
        ],
        "series": "saucy",
    "ppa_list": ["ppa:ci-team/ppa_build_1", "ppa:ci-team/ppa_archive"],
```

```
    "package_list": ["python-autopilot", "python3-autopilot"],
    "progress_queue": "image-builder-1-exchange"
}
```

Image Builder -> Lander

Build progress messages are passed back via the progress queue:

```
TBD
```

Completion and location of image is sent via a message on the progress queue:

```
message {
    "state": "COMPLETED",
    "url_list": [
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1/ticket.1.iso"
    ]
}
```

Lander -> Test Runner

The image and list of binary packages is sent to the test runner:

```
test_image {
    "test_request_id": "1",
    "image_url": "http://glance_url/image-1.iso",
    "package_name": "unity8"
}
```

Test Runner -> Lander

Returns status through progress messages:

```
message {
    "test_request_id": "1",
    "message": "10%",
    "current": "1",
    "total": "10",
}
```

Test completion is signaled by a final progress message:

```
message {
    "state": "COMPLETED",
    "test_request_id": "1",
    "status": "SUCCESS",
    "test_result": "PASSED",
    "artifacts": [
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1.test-runner/autopilot-ru
        "https://swift.canonistack.canonical.com/v1/AUTH_bucket_id/ticket.1.test-runner/autopilot.xml
}
```

Lander -> Ticket System

The lander provides progress to the ticket system through the ticket system's progress API:

```
TBD
```

Completion of a build is provided to the ticket system through the ticket system's progress API:

```
TBD
```

CHAPTER 8

Style and Technology Guidelines

This is intended to keep us on the same page about choices we make. Rules we have agreed on:

- Python is the language of choice when possible. We should try and use Python3 where possible, but in cases where we need Python2 libraries we'll try and keep the code Python3 friendly.
  - Code should pass pep8 rules and pyflakes tests. All editors have plugins that flag violations.
- Django 1.5 is the web framework of choice.
- New services need a REST interface and "tastypie" is the recommended tool for implementing the interface.
  - **TODO:** what's the recommendation for authentication/authorization?
- New services should include unit testing.
  - **TODO:** should we add guidelines about how much to test? ie just the model or also test REST interface?
- **TODO**: recommended REST client? tastypie-queryset-client looks promising and will feel transparent to people familiar with Django.

CHAPTER 9

Manually setting up launchpad OAuth token

There is a great how-to for doing this:

> https://help.launchpad.net/API/SigningRequests

This document just shows the actual *curl* commands required to do everything.

step 1: Get a request token

Run the following command command to obtain a request token:

```
curl --dump-header - -H "Content-Type: application/x-www-form-urlencoded" -X POST --data 'oauth_consu
```

This will return something like:

```
oauth_token=token_value&oauth_token_secret=secret_value
```

step 2: Authenticate

Authentication of the above token is required via your browser by visiting:

```
https://launchpad.net/+authorize-token?oauth_token={oauth_token}
eg:
https://launchpad.net/+authorize-token?oauth_token=token_value
```

step 3: Get the access token

Run the following command to exchange the request token for an access token:

```
curl --dump-header - -H "Content-Type: application/x-www-form-urlencoded" -X POST --data 'oauth_signa
```

which returns something like:

```
oauth_token=token&oauth_token_secret=token_secret&lp.context=None
```

The oauth_token and oath_token_secret returned here plus the oauth_consumer_key will then be required by our API's to use Launchpad.

CHAPTER 10

Automatic creation of launchpad OAuth tokens

The following steps will enable the automatic creation of launchpad OAuth tokens. Running bin/create_lp_creds.py present in lp:uci-engine and following the on-screen instructions will produce the credentials file, lp_creds.txt. Sourcing this file will allow the engine to interact with launchpad on your behalf.

```
bzr branch lp:uci-engine
cd uci-engine
./bin/create_lp_creds.py  # Follow the instructions to choose 'Change anything'
                            on the web page that has been just created.
. lp_creds.txt            # Source the credentials file
```

CHAPTER 11

ppa-hooks

[12:03] <ev> infinity, cjwatson, pitti, others: do we have a means of extracting coverity data, code coverage, and test results inside packages built in a PPA (from dep8 tests)? Could we (ab)use binarypkgmangler for such a task?

[12:04] <ev> we're trying to be good citizens of Launchpad in the new CI Airline architecture, but this is one area where we've seemingly needed pbuilder hooks

[12:59] <pitti> ev: not pkgbinarymangler, as that runs too late; AFAIK you need to change the ./configure/CFLAG arguments, don't you?

[13:00] <pitti> ev: so it might be a modified dpkg which supplies these extra CFLAGS by default, or something like that

[13:01] <pitti> ev: I don't know whether it's just CFLAGS or whether the build system needs to support gcov/lcov in other ways; so far I've just used the gnome-common macros

[13:49] <ev> pitti: sorry, I'm not sure I follow. How can we provide a modified dpkg to a PPA? If you upload dpkg to a PPA does it automatically pick up and use that version (if so, very clever). Does pkgbinarymangler really run too late for extracting out the coverage, coverity, and test case artifacts?

[13:50] <cjwatson> It does

[13:50] <cjwatson> Since the PPA itself is in sources.list for any builds to that PPA, and the builder upgrades its chroot at the start of each build

[13:51] <cjwatson> pkgbinarymangler could certainly be hacked to extract artifacts, provided that something has arranged to generate them in the first place ...

[13:51] <cjwatson> (dpkg seems a bit low-level for this though, and that would be an utter pain to maintain)

[13:52] <cjwatson> You could divert dpkg-buildflags, as long as you only care about packages that use it (we could reasonably mandate that for things we own)

[13:52] <cjwatson> Most of our stuff probably uses it already by way of dh9

[13:57] <pitti> ev: right, I actually meant dpkg-buildflags (I thought that was in dpkg)

[14:00] <cjwatson> pitti: It's in dpkg-dev, yes, but it would be quite a bit of ongoing cost to do that by uploading a modified dpkg - would have to keep merging

[14:00] <cjwatson> Should be easy enough to divert if that's what's needed - call the underlying one and tweak

[14:00] <pitti> right

[14:12] <ev> well the problem then becomes how do you make the package doing the diverting of dpkg-buildflags a requirement for all the packages in the PPA without explicitly asking for it in their control files

[14:12] <ev> at least as I see it

[14:15] <cjwatson> ev: That's not a problem if it's one of the things that's already preinstalled in the chroot; pkgbinarymangler would qualify

[14:19] <ev> cjwatson: I thought while pkgbinarymangler was determined to be a good target for extracting the artifacts, it ran too late to divert dpkg-buildflags? Or did I misunderstand what you said above?

[14:25] <pitti> ev: it currently only diverts dpkg-deb, but it could additionally divert dpkg-buildflags

[14:29] <ev> pitti: ohh. So if I understand correctly: given a PPA that we want to extract gcov data from, we upload a fork of pkgbinarymangler that diverts dpkg-buildflags to include the gcov flags and also splits out the coverage data into a "-coverage" package?

[14:33] <pitti> ev: or upload that mangler to ubuntu, and make it check something in the PPA to see whether you want cov enabled

[14:36] <pitti> ev: of course the first tests should actually happen with a forked pacakge in a PPA, yes

CHAPTER 12

Using Juju LXC For Local Development

Development efforts can be sped up using local juju deployments with lxc.

12.1 Setting up Juju LXC

You'll need a .juju/environments.yaml file with a "local" entry like:

```
default: local
environments:
  local:
    type: local
    default-series: precise
    lxc-clone: true
    lxc-clone-aufs: true
    admin-secret: secret
    apt-http-proxy: http://10.0.3.1:8000
```

Install squid-deb-proxy. Then create /etc/squid-deb-proxy/mirror-dstdomain.acl.d/20-juju-local with the following contents:

```
ppa.launchpad.net
private-ppa.launchpad.net
```

Restart the squid-deb-proxy service. All apt downloads will now be cached by the proxy server running on your local machine.

Next, you will need swift credentials set up. This ensures the services still have access to object storage (Swift), virtual machine image storage (Glance), and virtual machine instantiation (Nova). An example .hpcloud-rc file should look like:

```
# FOR SWIFT
export JUJU_ENV=local
export OS_USERNAME="<your email address>"
export OS_TENANT_NAME="<your tenent name from horizon>"
export OS_PASSWORD=<something special>
export OS_AUTH_URL="https://region-a.geo-1.identity.hpcloudsvc.com:35357/v2.0"
export OS_REGION_NAME=region-a.geo-1

# FOR GLANCE (same credentials as above by default)
export GLANCE_OS_USERNAME="$OS_USERNAME"
export GLANCE_OS_AUTH_URL="$OS_AUTH_URL"
export GLANCE_OS_REGION_NAME="$OS_REGION_NAME"
```

```
export GLANCE_OS_TENANT_NAME="$OS_TENANT_NAME"
export GLANCE_OS_PASSWORD="$OS_PASSWORD"

# FOR OAUTH TOKENS
export CI_LAUNCHPAD_PPA_OWNER=<lp login-id>
export CI_LAUNCHPAD_USER=<lp login-id>
export CI_OAUTH_CONSUMER_KEY="ci-airline"
export CI_OAUTH_TOKEN=<Please see Note below+++>
export CI_OAUTH_TOKEN_SECRET=<Please see Note below+++>
```

Note: +++ Use the OAuth values generated in OAuth setup to fill these

If your swift server only supports the 1.0 auth protocol (TempAuth does not support 2.0), you additionally need to set some `$ST_*` variables for `python-swiftclient`, and include the tenant/project name into the user name:

```
export OS_USERNAME="<project:username>"
export OS_TENANT_NAME="<project>"
export OS_PASSWORD="<password"
export OS_AUTH_URL="http://your.swift.server:8080/auth/v1.0"
export OS_REGION_NAME=

# env for python-swiftclient
export ST_AUTH=$OS_AUTH_URL
export ST_USER=$OS_USERNAME
export ST_KEY=$OS_PASSWORD

# GLANCE_*, CI_* as above
```

If you use the Python virtualenv produced by `testing/venv.py`, you must additionally set `export EPHEMERAL_CLOUD_NET_ID=..` in that rc file, as neutron is broken in the venv, and thus it cannot determine the net ID by itself.

You'll now have the settings in-place. In order to iterate rapidly, It would be advisable to ensure juju's template image has all the dependencies pre-installed so you don't wait for that every time you re-deploy. You can do that with the following simple manual hack:

```
juju bootstrap
juju deploy cs:ubuntu # Wait for juju status to show it deployed
juju destroy-environment --force -y local

# You'll now have an lxc container named juju-trusty-template.
# Modify it with these commands:
sudo lxc-start -d --name juju-trusty-template
sudo lxc-attach --name juju-trusty-template -- add-apt-repository -y ppa:canonical-ci-engineering/ci-
sudo lxc-attach --name juju-trusty-template -- apt-get update
sudo lxc-attach --name juju-trusty-template -- apt-get install -y rabbitmq-server python-amqplib pyth

sudo lxc-stop --name juju-trusty-template
```

Alternatively for a precise deployment you want:

```
# You'll now have an lxc container named juju-precise-template.
# Modify it with these commands:
sudo lxc-start -d --name juju-precise-template
sudo lxc-attach --name juju-precise-template -- add-apt-repository -y ppa:canonical-ci-engineering/ci
sudo lxc-attach --name juju-precise-template -- add-apt-repository -y cloud-archive:icehouse
sudo lxc-attach --name juju-precise-template -- apt-get update
sudo lxc-attach --name juju-precise-template -- apt-get install -y rabbitmq-server python-amqplib pyt
```

```
sudo lxc-stop --name juju-precise-template
```

12.2 Host Configuration

Some additional changes will be necessary on the LXC host system for the imagebuilder to work properly. With some of the changes we have planned, many of these should not be needed soon. Be aware that making these changes may have an effect on other LXC containers you run on your host system.

First, ensure that nbd is loaded on the host. Module loading will not work in LXC, but the module will be available under lxc if it is loaded on the host.

Add the following lines to /var/lib/lxc/juju-precise-template/config:

```
# Allow mounting filesystems under LXC
aa_profile = lxc-container-default-with-mounting
# Allow full access to the block device with major number 43, which
# should be nbd (see /proc/devices)
lxc.cgroup.devices.allow = b 43:* rwm
```

Modify the LXC default apparmor rules to allow bind mounting filesystems under LXC. In /etc/apparmor.d/lxc/lxc-default, add the following line before the "}":

```
mount options=(rw, bind, ro),
```

Then run:

```
sudo /etc/init.d/apparmor reload
```

12.3 Working with the code

Code modifications can be done using the following iterations:

```
1) bzr branch lp:uci-engine
2) cd uci-engine
3) <do changes>
4) juju destroy-environment --force -y local; juju bootstrap
5) rm -rf tmp/
6) ./juju-deployer/deploy.py # Or './juju-deployer/deploy.py branch-source-builder' to only deploy Br
7) <check the services>
8) Repeat from step 3-7 for iterative development
```

12.4 Upgrade

The development effort can be further sped up using the `--upgrade` option in `deploy.py`. The generic steps of upgrading are given in the Upgrade section. The following steps are specific to the local development:

```
1) bzr branch lp:uci-engine
2) cd uci-engine
3) <do changes>
4) juju destroy-environment --force -y local; juju bootstrap
5) rm -rf tmp/
6) ./juju-deployer/deploy.py --build-only --working-dir ./tmp
```

```
7) ./juju-deployer/deploy.py   # use ./juju-deployer/deploy.py branch-source-builder for deploying or
8) <do changes in charms for the deployed services locally>
9) ./juju-deployer/deploy.py --upgrade all # this will upgrade the modified charms and config
10) Repeat the steps 8-9 for fixing and testing the charms that were already deployed
```

CHAPTER 13

Upgrading a Deployment

Upgrading charms can be sped up using the `--upgrade` option given in `deploy.py`. The upgrade option is deployment aware and only attempts to upgrade services that are deployed. Likewise, it will not modify any services that are not described in the deployer configuration files.

The `--upgrade` option is described as:

```
--upgrade {charms,code,config,all}
                Upgrade an existing deployment. This argument allows
                you to choose the order and number of upgrade
                operations. ie - you could just update the charms,
                update the configs, or update the code.
```

The upgrade choices allow for fine grain control over specific aspects of a deployment.

- **charms** - This upgrades the charms themselves, performing `juju upgrade-charm --force` for each deployed service.
- **config** - Updates the options for the deployed services via `juju set`. This includes an update of the application code (as that is part of the deployment configuration).
- **code** - This is a special case of the config option tht only updates the deployed application code. This is to provide an efficient path for the common development case of iterating on application code.
- **all** - This is a synonym for `--upgrade charms --upgrade config`.

These options are particularly useful when combined with the lxc deployment method described under "Upgrade" under *Using Juju LXC For Local Development*

13.1 Examples

Assuming the initial setup of *Using Juju LXC For Local Development* or *Setting Up a Cloud Deployment* has been performed, the generic steps where upgrade becomes useful are:

Note: In all of the below examples, if only a few services were deployed, the `--upgrade` option will only upgrade the services that were already deployed. No new services will be deployed as part of the upgrade.

13.1.1 –upgrade all

```
1) bzr branch lp:uci-engine
2) juju bootstrap
3) cd uci-engine
4) ./juju-deployer/deploy.py
   or
4) ./juju-deployer/deploy.py test-runner  # for deploying only test-runner service
5) <do changes in charms and code for the deployed services>
6) ./juju-deployer/deploy.py --upgrade all
```

13.1.2 –upgrade charms

```
1-4) <same as --upgrade all example above>
5) <do changes to charms for the deployed services>
6) ./juju-deployer/deploy.py --upgrade charms
```

13.1.3 –upgrade config

```
1-4) <same as --upgrade all example above>
5) <do changes to charm config and/or code for the deployed services>
6) ./juju-deployer/deploy.py --upgrade config
```

13.1.4 –upgrade code

```
1-4) <same as --upgrade all example above>
5) <do changes to code for the deployed services>
6) ./juju-deployer/deploy.py --upgrade code
```

13.2 Upgrading adt-run for the test runner

The test runner depends on **adt-run** to run the tests.

When needed (new upstream version, bug fix) the CI gated branch needs to be updated.

Gating these updates guarantees that they can be tested before being deployed and that hot fixes can also be deployed without requiring upstream intervention (those fixes should be rare and upstreamed in any case).

Then, there is a recipe to build autopkgtest into the phase-0 PPA.

Trigerring that recipe will build **autopkgtest** for the series that are used in the CI engine.

The last step is to update the deployed test runner workers themselves:

```
juju run --service ci-airline-tr-rabbit-worker 'apt-get update; apt-get install autopkgtest'
```

CHAPTER 14

Setting Up a Cloud Deployment

In order to allow the engine to access launchpad on your behalf, the following lines need to be added to your respective novarc files.

```
export CI_OAUTH_CONSUMER_KEY=ci-airline
export CI_OAUTH_TOKEN=<Please see Note below+++>
export CI_OAUTH_TOKEN_SECRET=<Please see Note below+++>
export CI_LAUNCHPAD_USER=<lp login-id>
export CI_LAUNCHPAD_PPA_OWNER=<lp login-id>
```

Note: +++ Use the OAuth values generated here to fill these

In addition, for HPcloud deployments, the following line should also be added to the novarc file.

```
export CI_TEMPURL_SIGNING_KEY=<the same as HP_SECRET_KEY>
```

CHAPTER 15

Deploying with Nagios

The nagios and nrpe charms from the charm store can be added to a deployment to test the actual nagios checks. After deploying the two charms, add the relationships between nrpe and the desired component(s) to monitor and then between nrpe and nagios. First add the extra charms:

```
juju deploy nagios
juju deploy nrpe
```

The nagios charm is known to fail the install hook due to a missing 'apt-get update'. To resolve this:

```
juju run --service nagios "apt-get update"
juju resolved --retry nagios/0
```

Use 'juju status' to monitor for a successful nagios deployment. Then add the relationships:

```
juju add-relation nrpe ci-airline-bsb-worker
juju add-relation nrpe nagios
```

Now find the nagios password:

```
juju ssh nagios/0 sudo cat /var/lib/juju/nagios.passwd
```

Use 'juju status' to find the public-address of the nagios instance, then navigate to 'http://<public-address>/nagios3' with a username of 'nagiosadmin' and the password from above.

CHAPTER 16

Indices and tables

- *genindex*
- *modindex*
- *search*