A STRUCTURED APPROACH TO CONTROLLED INTRODUCTION OF CHANGES

AGENDA

- nomenclature
- phased rollout success condition
  - configuration validity & service health
- life cycle management / introducing change
  - problem definition
- a solution: feature-flags

AUTOMATION CHANGES OVER TIME

- automation is the embodiment of processes in programmatic fashion
- products and processes change, thus automation must change over time

NOMENCLATURE

- blast radius containment
- phased rollout

BLAST RADIUS CONTAINMENT

- strictly about limiting impact of a change
- for example, make sure we cannot commit to more than X devices in one transaction
  - this would prevent making one network wide transaction changing all instances of Y at the same time

PHASED ROLLOUT

- method of rolling out a change in controlled fashion by applying a subset of the change at a time
- for example, total change is to 1000 devices
  - start by changing one device
  - validate change went well
  - proceed with changing next device
  - validate and proceed to next
  - repeat for all devices…
    - or abort on failure

PHASED ROLLOUT SUCCESS CONDITION

- key to phased rollout is concept of **success condition**
- we roll out change in small parts - sliding window
- how do we know if a change went well?
- how do we know if we can proceed and configure next device?

CONFIGURATION VALIDITY

- configuration commit only includes syntax and semantic checks
  - an empty configuration is valid
    - but would lead to unhealthy device / service
- thus, need concept of **success condition** beyond basic config validity

SERVICE HEALTH

- need to understand if service is **healthy**
- monitor operational state of service
  - is BGP neighbor up?
  - is interface up?
  - can we ping?
- service specific! not generic…

INTRODUCING (NAIVE) CHANGE

- change service configuration template
- git commit
- deploy new NSO package
- re-deploy service instances

-> new config is now active in network

EXAMPLE CHANGE

- modify MTU of service from default (1500) to 9100

CHANGE SERVICE TEMPLATE

```xml
<config-template xmlns="http://tail-f.com/ns/config/1.0">
    <devices xmlns="http://tail-f.com/ns/ncs">

        <device tags="nocreate">
            <name>{/device}</name>
            <config tags="merge">
                <interface-configurations xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg">
                    <interface-configuration>
                        <active>act</active>
                        <interface-name>{/interface}</interface-name>
                        <description>Link to {/remote/device} [{/remote/interface}]</description>
                        <mtus>
                            <mtu>
                                <owner>{$INTERFACE_TYPE}</owner>
                                <!-- new hard-coded MTU -->
                                <mtu>9100</mtu>
                            </mtu>
                        </mtus>
                        <shutdown tags="delete" when="{/shutdown='false'}"/>
                        <!-- ... other config stuff ... -->
                    </interface-configuration>
                </interface-configurations>
            </config>
        </device>
    </devices>
</config-template>
```

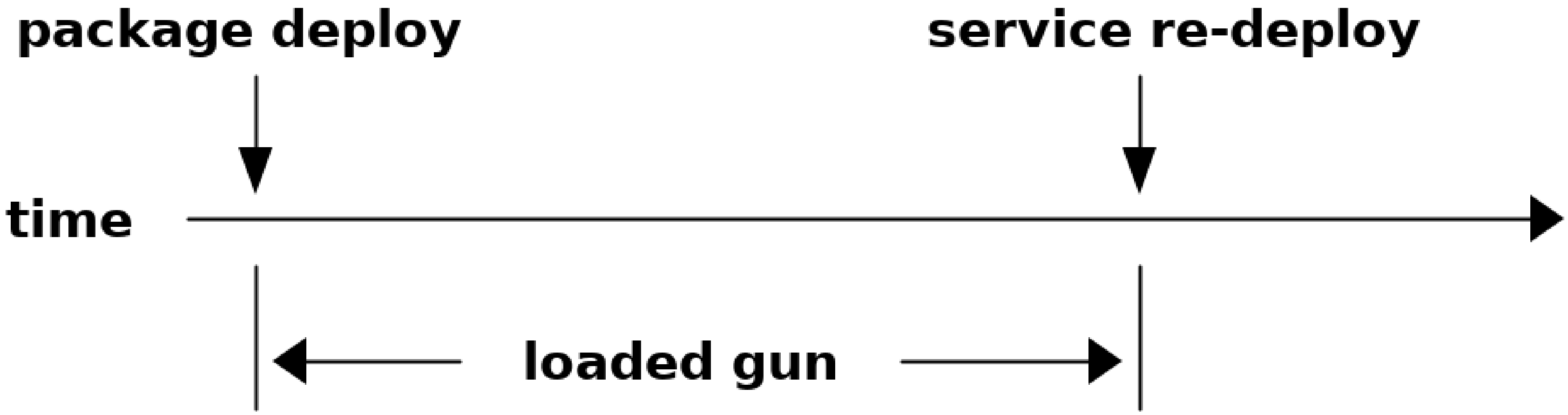
BENEFIT OF HARD-CODED VALUE

- **no** choice means **avoiding** test cases - good!
- configurable MTU means we need to test all or reasonable set of values
- combinatorial explosion with many config knobs

LOADED GUN SCENARIO

LATENT CHANGE = LOADED GUN

- latent change in service code = loaded gun
  - starts when service package is deployed
  - end when service is re-deployed

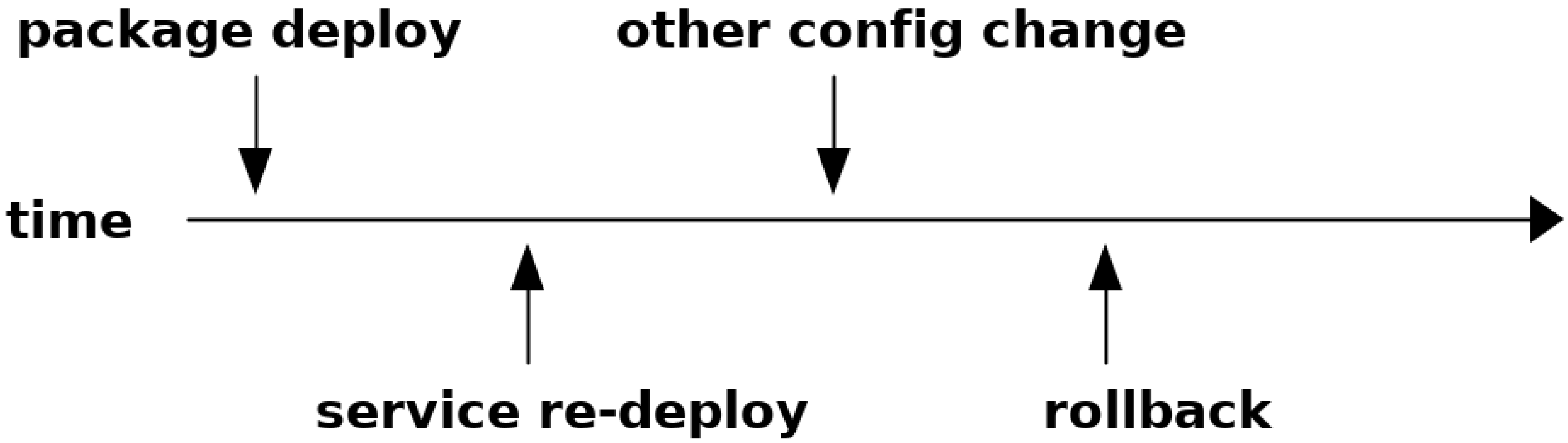


KALLE FIRED THE GUN

- anyone else coming in doing (trivial) change in loaded gun window will inadvertently push MTU change
  - Kalle wants to fix spelling mistake in description
    - considered trivial, didn't do `commit dry-run`
    - pushes MTU change causing service outage

NO REVERT

- template change & re-deploy moves forward
- no way back
  - except rollback, but only works for naive scenario
  - interleaved transactions make rollback useless



FEATURE GROUPING

GOALS

- no loaded gun
- going backwards rollback
- success condition / service health
- avoid combinatorial explosion
  - allows testing

FEATURE-FLAGS

- well known concept in software development
- move introduction of change from commit/deploy time to run time
  - temporal decoupling of development and operations!!!
- focus on transition / change
- limited life time

FEATURE-FLAG

- emphasize old -> new transition

```
list backbone-interface {
  key "device interface";
  // other things

  container feature-flags {
    leaf high-mtu {
      type boolean;
      description "Enable new high MTU (9100). Disable for old MTU (1500)";
      default "false";
    }
  }
}
```

FEATURE-FLAG IN TEMPLATE

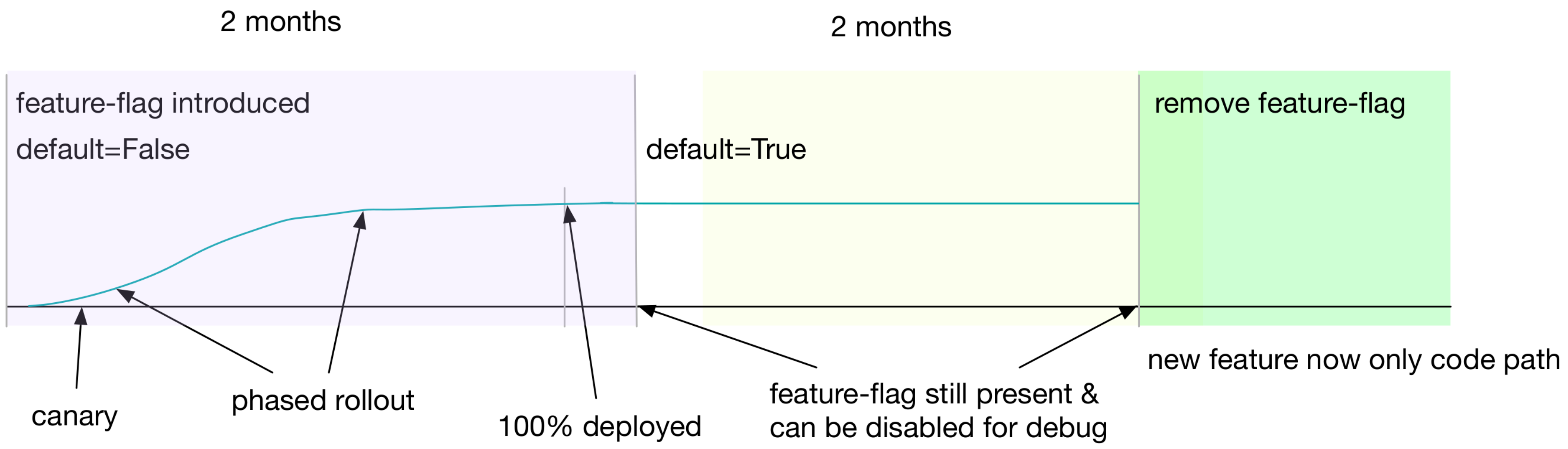
```xml
<config-template xmlns="http://tail-f.com/ns/config/1.0">
    <devices xmlns="http://tail-f.com/ns/ncs">

        <device tags="nocreate">
            <name>{/device}</name>
            <config tags="merge">
                <interface-configurations xmlns="http://cisco.com/ns/yang/Cisco-IOS-XR-ifmgr-cfg">
                    <interface-configuration>
                        <active>act</active>
                        <interface-name>{/interface}</interface-name>
                        <description>Link to {/remote/device} [{/remote/interface}]</description>
                        <mtus>
                            <mtu>
                                <owner>{$INTERFACE_TYPE}</owner>
                                <!-- new high MTU conditioned on feature-flags -->
                                <mtu when="/feature-flags/high-mtu='true'">9100</mtu>
                            </mtu>
                        </mtus>
                        <shutdown tags="delete" when="{/shutdown='false'}"/>
                        <!-- ... other config stuff ... -->
                    </interface-configuration>
                </interface-configurations>
            </config>
        </device>
    </devices>
</config-template>
```

SOCIOTECHNICAL

- technically, FF is *just another input*
- NSO won't treat it differently
- difference is in concept
  - clear life cycle for FF
  - introduce FF for change transition
  - when done, **remove** FF
    - keeps down input / permutations over time

FEATURE-FLAG LIFE CYCLE



ANTI-PATTERN

- we could introduce new MTU leaf
- allows arbitrary pick of MTU
- BAD - we want choice of 1500 or 9100
- reduce choice / permutations

```
list backbone-interface {
  key "device interface";
  // other things

  leaf mtu {
    type uint16 {
      range "1500..9100";
    }
    description "MTU of service";
    default "1500";
  }
}
```

ANTI-PATTERN

- better, reduction of choice to 2 values
- still, over time, new config knobs leads to combinatorial explosion
- focus on transitional nature

```
list backbone-interface {
  key "device interface";
  // other things

  leaf mtu {
    type uint16 {
      range "1500 | 9100";
    }
    description "MTU of service, either 1500 (old) or 9100 (new)";
    default "1500";
  }
}
```

PHASED ROLLOUT

- feature-flags are per service instance
- many flags to flip for 10000 service instances
- automatic?

SLIDING WINDOW

- sliding window relies on **success condition**
- in practice: service self-test

PROCEDURE

- find list of feature-flags
- for each;
  - go to service owning feature-flag
  - run service self-test, early exit on fail
  - flip feature-flag
  - run service self-test
    - rollback on error (flip back flag)
    - continue to next FF instance on success

INTROSPECTION

- feature-flag can be specific type
  - makes it a generic procedure to find through introspection
  - flag type can indicate *direction*

```
typedef ff-boolean-false-to-true {
  type boolean;
  description "A boolean feature flag that transitions from false to true";
  default false;
}
```

FEATURE-FLAG NAVIGATOR

- this is a mock-up

```
show feature-flags feature-flags
| feature-flag                                            | type          | progress |
|---------------------------------------------------------+---------------+----------|
| /infrastructure/base-config/feature-flags/foobar        | false-to-true |      73% |
| /infrastructure/backbone-interface/feature-flags/bar    | false-to-true |      14% |

show feature-flags instances
| instance                                                          | type          | value | complete |
|-------------------------------------------------------------------+---------------+-------+----------|
| /infrastructure/base-config{901-R1-2053}/feature-flags/foobar     | false-to-true | false | false    |
| /infrastructure/base-config{901-R1-2054}/feature-flags/foobar     | false-to-true | true  | true     |
| /infrastructure/bb-intf{901-R1-2053 et-9/0/0}/feature-flags/bar   | true-to-false | false | true     |
| /infrastructure/bb-intf{901-R1-2053 et-10/0/0}/feature-flags/bar  | true-to-false | true  | false    |
```

SERVICE SELF-TEST

- what is it in practice?
- a YANG action!
- specific to service type
- by returning simple common structure it can be used in generic fashion

RETURN GENERIC

```
action self-test {
  tailf:info "Perform self-test of the service";
  tailf:actionpoint "backbone-interface-self-test";
  output {
    leaf success {
      type boolean;
    }

    container interface {
      // service specific health / state about the interface
    }
    container is-is {
      // service specific health / state about IS-IS
    }
    container pim {
      // service specific health / state about PIM
    }
  }
}
```

```
action self-test {
  tailf:info "Perform self-test of the service";
  tailf:actionpoint "ibgp-neighbor-self-test";
  output {
    leaf success {
      type boolean;
    }

    container bgp {
      // BGP specific health / state
    }
  }
}
```

EXAMPLE

```
def get_state(kp_unused, log, root=None, service=None, action_output=None):
    log.info("get_state for {} {}".format(service.device, service.interface))
    dev = root.devices.device[service.device]
    os = utils.get_dev_os(dev)
    state = service.state
```