GoodData

Test Driven Infrastructure

with Puppet, Docker, Test Kitchen and Serverspec

About Me

Russian engineer living in Prague

- Yury Tsarev
- Mostly QA & Systems Engineering background
- Previously
  - Sysadmin in Russian bank
  - QA/Sr. SWE in SUSE Linux
  - Sr. QA Engineer in Cloud Infra team of GoodData
- Currently QA Architect in GoodData focusing on
  - Private cloud
  - Internal PaaS
  - Continuous Delivery

- Contacts
  - yury.tsarev@gooddata.com
  - https://github.com/ytsarev
  - https://linkedin.com/in/yurytsarev
  - https://twitter.com/xnullz

About GoodData

Information relevant to the talk

- GoodData runs cloud-based business intelligence (BI) and big data analytics platform
- Operates on top of Openstack-based Private Cloud and Internal Platform-as-a-Service
- Several datacenters, hundreds of hardware servers, thousands virtual machines, internally complex distributed system to manage
- Relies on Puppet for Configuration Management

What to Expect From This Talk ?

- A real-life story of infrastructure development process evolution
- A practical guide with opinionated set of tools for testing infrastructure at scale
- A framework which components are ready to be adjustable or replaceable for your specific case
- No kittens, no unicorns, no Docker worship

Infrastructure as a Code

Puppet driven infra as a prerequisite

- Every infra change is tracked through puppet code, no exceptions
- Puppet code is stored in git
- Puppet code is a shared ground for whole DevOps organization
- Puppet code has quality problems

Puppet Architecture in GoodData

Server Type is an entrypoint

- Masterless Puppet
- Popular Roles&Profiles pattern is **not** implemented
- Instead there is notion of Type(can be understood as Server Type/Role) which is a main entrypoint for code execution
- Type is a combination of puppet modules describing a resulting server state
- Relatively huge codebase
  - Around 150 puppet modules
  - More than 100 types
- Applying a type on the instance is as easy as propagate $EC2DATA_TYPE environment variable, e.g. from openstack metadata endpoint

Challenges at Scale

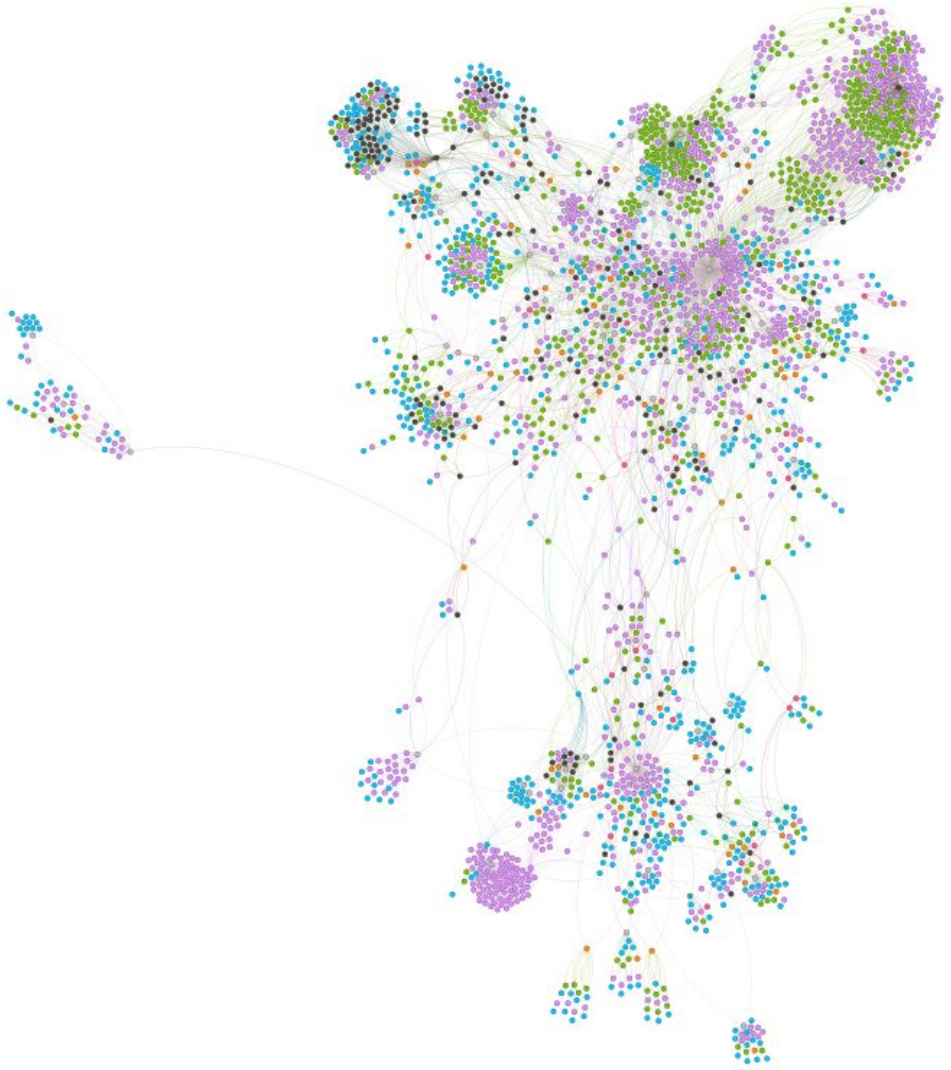
Complex distributed platform to manage

- Puppet code is a base for everything:
  from Openstack Private Cloud up to application frontend
- Several hundred of physical servers and thousands of VMs
- Multiple datacenters
- Multiplied by huge number of puppet types
- Tightly coupled modules with multiple interdependencies
- Complexity creates reliability and quality problems

Dependencies Depicted

Our puppet dependency graph

- Representation of puppet modules interdependencies
- 2650 Resources
- 7619 Dependencies between them



Manual way of puppet validation

Does not work at scale

- Checkout new puppet code
- Run `puppet apply --noop`
- Evaluate the output
- If --noop looks fine then make real apply
- Manual smoke testing

Obviously such process **does not scale**

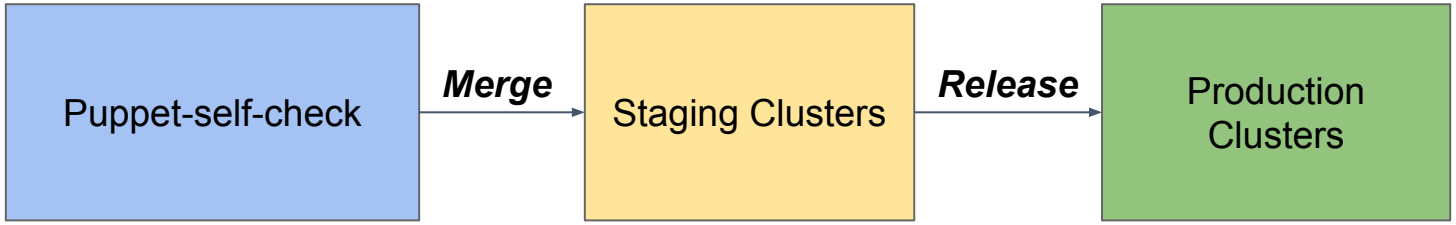
Introducing Puppet Self Check

As we need some test before the merge

- Linting (puppet-lint)
- Puppet catalog compilation
- Automated --noop run in fakeroot
- Integration with Jenkins
- Detailed feedback right in Pull Request

Minimalistic Deployment Pipeline

Consisting of only one **testing** job so far



Puppet Self Check is not enough

Crucial, but only initial coverage

- Covering
  - Style errors
  - Syntax errors
  - Catalog compilation errors like circular dependencies
- Missing
  - Configuration file errors
  - Ability to check if services/processes were able to start
  - No configuration testing
  - No service smoke testing
- We want to catch the issues way **before** the merge
  - Shifting testing left is great for quality and velocity
  - Staging should uncover minimal amount of complex integration issues

Introducing Test Kitchen

Something more for the next step of test pipeline

- http://kitchen.ci/
- Advanced test orchestrator
- Open Source project
- Originated in Chef community
- Very pluggable on all levels
- Implemented in Ruby
- Configurable through simple single yaml file
- "Your infrastructure deserves tests too."

Test Kitchen architecture

Main components and verbs

- Driver: what type of VM/containerization/cloud to use
  - *Amazon EC2*, *Blue Box*, *CloudStack*, *Digital Ocean*, *Rackspace*, *OpenStack*, *Vagrant*, *Docker*, *LXC containers*
- Provisioner: which configuration management tool to apply
  - *Chef*, *Puppet*, *Ansible*, *SaltStack*
- Verifier: test automation type to verify the configuration correctness with
  - *Bats*, *shUnit2*, *RSpec*, *Serverspec*

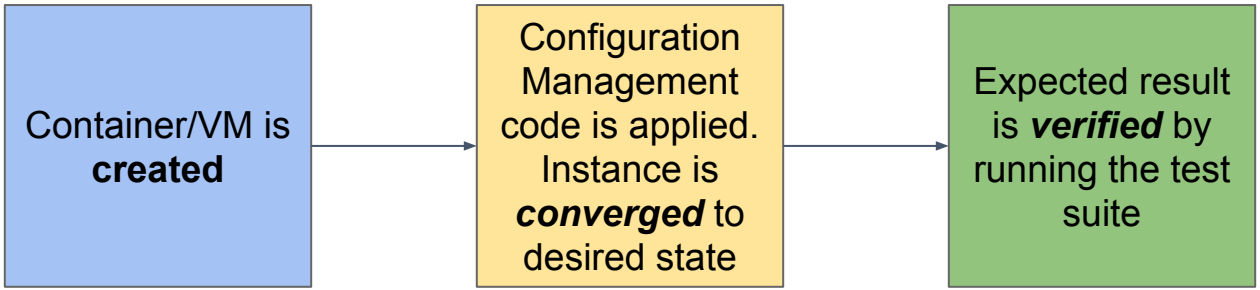
- Driver ***creates*** the instance
  - $ kitchen create ...
- Provisioner ***converges*** the puppet code
  - $ kitchen converge ...
- Verifier ***verifies*** the expected result
  - $ kitchen verify ....

Test Kitchen Sequential Testing Process

Create -> Converge -> Verify

Test Kitchen Verbs Meaning

What is actually happening



Which Driver to use ?

Or why we stick to Docker

- The openstack driver could be an obvious choice for our openstack-based private cloud
- But remember we have more than 100 puppet types to test?
- That would mean at least one full-blown VM for each type-under-test
- Even with minimum instance flavour it is too much
- **Thus, we stick to Docker**

And it wasn’t smooth ride

Docker Driver Specifics

What does it bring and take away

- Resource utilization is a game changer
  - Instead of spawning more than 100 VMs we are managing the testing load within small 3-nodes jenkins slave cluster
- Shared testing environment
  - Same containers spawned on jenkins slaves and on developer laptops
- Allows to create ***system*** containers that are mimicking VMs/servers
- It does **not** come for free
  - Docker specific limitations and constraints
  - Deviation from real VM scenario
  - Hard to debug issues with process concurrent behavior. Most of them relate to the fact that users are not namespaced in Linux kernel

Docker Driver Project & Configuration

Driver section of .kitchen.yml

- Separate project https://github.com/portertech/kitchen-docker
- ```
  driver:
    name: docker
    image: docker-registry.example.com/img:tag
    platform: rhel
    use_sudo: false
    provision_command:
      yum clean all && yum makecache
  ```

- Basic docker driver definition in .kitchen.yml
- Uses the image from private docker registry
- Specifies provision command for test container runtime preparation


  ```

Docker Driver Additional Features

That are useful for testing

- `volume:`
  - `- /ftp`
  - `- /srv`
- `cap_add:`
  - `- SYS_PTRACE`
  - `- SYS_RESOURCE`
- `dockerfile:`
  - `custom/Dockerfile`

- Volume management
- Kernel capabilities
- Custom Dockerfile for testing container

Now we are fully prepared for Create stage.
`kitchen create` will spawn fresh testing container.
Next one is Converge.
And Converge means Provisioner.

Puppet Provisioner

An obvious choice to run puppet code

- Also distinct upstream project https://github.com/neillturner/kitchen-puppet
- Uploads puppet code into instance under test
- Runs puppet there a.k.a. getting instance to ***converged*** state
- Provides extremely useful functionality for creating puppet related testing constraints
    - Puppet facts customization facility
    - Hiera
    - Facterlib
    - Custom installation, pre-apply, post-apply commands
    - And much more documented in provisioner_options.md

Puppet Provisioner Configuration

Provisioner section of .kitchen.yml

```
provisioner:
  name: puppet_apply
  modules_path: puppet/modules
  manifests_path: puppet/manifests
  hiera_data_path: puppet/hieradata
  facterlib: /etc/puppet/facter
  install_custom_facts: true
  custom_facts:
    ec2data_type: web_frontend
    docker: 1
    ec2data_freeipa_otp: test
    ec2data_nopuppet_cron: 1
    ec2_public_ipv4: 127.0.0.1
    ...
  custom_install_command: |
     # custom setup script
```

- Specifies *local* paths to manifests, modules, hiera under test
- Overrides the puppet facts to create testing constraints
- Describe custom script to be executed before puppet run

Looks like we are good with Provisioner and can proceed to Verifier?

NO.
The puppet run will miserably fail.

External Dependencies

Or how to test in isolation

- Quite frequently puppet types under test will require some external service to communicate with
- We want to avoid external communication in most cases
- Because we want fast and deterministic results
- And we do not want to spoil production services with test data
- Same goes for additional load
- Exceptions (the things that we still do want to communicate)can be core services like ntp, rpm repository server, etc.

Introducing Shellmock

Extremely simple stub/mocking tool

- https://github.com/gooddata/shellmock
- Solution to external dependency problem a.k.a dependency injection
- A simple ruby script that placed instead of package manager strictly within testing instance
- Intercepts calls to package manager
- Installs a mock instead of real package if this mock is defined in shellmock configuration
- If mock is not defined passes the request to real package manager for real package installation
- Same trick is used to bypass number of docker limitations like sysctl calls within container

Shellmock Example Configuration

Separate shellmock.yaml that lives in puppet repo

Mock configuration	Resulting mock
ipa-client: /usr/sbin/ipa-client-install: | echo 'server = freeipa.example.com' > /etc/ipa/default.conf echo 'fake' > /etc/krb5.keytab	$ cat /usr/sbin/ipa-client-install echo 'server = freeipa.example.com' > /etc/ipa/default.conf echo 'fake' > /etc/krb5.keytab
sssd: /usr/sbin/sssd:	$ cat /usr/sbin/sssd #!/bin/bash echo I am a fake /usr/sbin/sssd

- Format is simple

```
package:
  /path/to/executable:|
      contents
```

- If no content specified the mock defaults to simple message that returns
  exit code 0

If all external dependencies are satisfied the Converge will be passing.
`**kitchen converge**` will successfully apply puppet.

Now it is time to deal with Verifier.

Verifier

A component that executes test framework

- Serves as tiny configuration for test execution
- Tests are written separately with the test framework of choice
- Out of the box support for *Bats*, *shUnit2*, *RSpec*, *Serverspec*, *Cucumber*

Why Serverspec?

Because it rocks

- Standalone open source project
  http://serverspec.org/
- Rich resource library
  http://serverspec.org/resource_types.html
- DSL to describe expected configuration state
- Based on famous RSpec framework
- Multi OS support
- Readable, flexible, extensible
- Low entry barrier

Serverspec DSL Examples

```
describe file('/var/log/httpd') do
  it { should be_directory }
end
```

```
describe command('apachectl -M') do
  its(:stdout) { should
contain('proxy_module')}
end
```

```
describe default_gateway do
  its(:ipaddress) { should eq
'192.168.10.1' }
  its(:interface) { should eq 'br0'
}
end
```

```
describe cgroup('group1') do
  its('cpuset.cpus') { should eq 1 }
end
```

```
describe
docker_container('focused_curie') do
  its(:HostConfig_NetworkMode) {
should eq 'bridge' }
  its(:Path) { should eq '/bin/sh' }
end
```

```
describe port(53) do
  it { should
be_listening.with('udp') }
end
```

- Command resource deserves special attention
- It allows to check anything you can do in shell
- More advanced scenarios will require creation of custom resources

What and How to Test?

A frequent question

- The most effective way to create a serverspec test is to think: “What would I anyway manually check after server deployment?”
- Then express it in serverspec DSL
- Most powerful serverspec tests are covering not only puppet produced configuration but **testing the outcome**

Testing the Outcome

DSL plus a bit of basic Ruby

```
under_kerberos_auth = ['', 'status', 'status.json',
'status/payload', 'images/grey.png']
under_kerberos_auth.each do |path|
  describe command("curl -k -X POST http://127.0.0.1/#{path}") do
    its(:stdout) { should match(/301/) }
  end
  describe command("curl -k -X POST https://127.0.0.1/#{path}")
do
    its(:stdout) { should match(/401/) }
    its(:stdout) { should match(/Authorization Required/) }
  end
end
```

- The test that checks behaviour of actual API endpoints after instance converge
- Notice that the test is still isolated within one instance
- Serverspec/RSpec is internal DSL so you can easily use Ruby right in the test definition

Test Suite Composition

There is no official reference

- Serverspec provides only DSL and abstraction over configuration resources
- The test suite organization and composition is completely up to you
- Great example of advanced, yet simple setup https://github.com/vincentbernat/serverspec-example
- GoodData implementations that were built on top of that example
  - https://github.com/gooddata/serverspec-core
  - https://github.com/gooddata/serverspec-ui
- These projects are only test executors, actual tests should be placed together with the code, e.g. same git repo with puppet

What Was Added on Top of Serverspec

Tiny bits of functionality

- YAML based configuration to define
  - Environments, e.g. dev/prod or geographical data centers
  - Host to role/type assignment
- Tests are placed in the directory structure according to defined hierarchy
- Parallel execution
- Reporting in multiple formats
- Coding style for tests (rubocop)

Shell Verifier and Serverspec

Flexibility of invocation

```
verifier:
  name: shell
  remote_exec: true
  command: |
    sudo -s <<SERVERSPEC
    export SERVERSPEC_ENV=$EC2DATA_ENVIRONMENT
    export SERVERSPEC_BACKEND=exec
    serverspec junit=true tag=~skip_in_kitchen \
      check:role:$EC2DATA_TYPE
    SERVERSPEC
```

- From kitchen configuration point of view it is just shell verifier
- remote_exec makes command to be executed within testing instance (execution from host is default)
- command invokes serverspec test suite with control variables and params

Create -> Converge -> Verify harness is ready.

`**kitchen verify**` will test the infra code.

But only for one puppet type.

How to describe multiple types?

Platforms and Suites

Additional structures in .kitchen.yml

- The way to create multiple test configurations
- Platforms are *usually* used to specify distribution/image to test on top of
- Suites are used to reflect semantics of test run
  In our case it is puppet type
- The configuration options from main components of Driver/Provisioner/Verifier
  can be overridden on Platform and Suite levels

Multi Distro Testing With Platforms

Covering EL6 to EL7 migration

```
platforms:
  - name: el6
    driver:
      image: docker-registry.example.com/el6:6.8
      cap_add: SYS_PTRACE
  - name: el7
    driver:
      image: docker-registry.example.com/el7:7.2
      dockerfile: t/kitchen/centos7-dockerfile
      volume:
        - /sys/fs/cgroup:/sys/fs/cgroup:ro
      run_command: /usr/lib/systemd/systemd
```

- Driver configuration is overridden per platform
- Additional tricks to run systemd within container

Multiple Types Description With Suites

```
suites:
  - name: frontend
    provisioner:
      custom_facts:
        ec2data_type: frontend
  - name: backend
    provisioner:
      custom_facts:
        ec2data_type: backend
  - name: database
    provisioner:
      custom_facts:
        ec2data_type: database
    driver:
      cap_add:
        - SYS_RESOURCE
```

- Driver and Provisioner are overridable on suite level as well and has a priority over Platform level

Resulting Test Composition

How kitchen composes suite and platform

```
$ kitchen list
```

Instance	Driver	Provisioner	Verifier	Transport	Last Action
frontend-el6	Docker	PuppetApply	Shell	Rsync	<Not Created>
frontend-el7	Docker	PuppetApply	Shell	Rsync	<Not Created>
backend-el6	Docker	PuppetApply	Shell	Rsync	<Not Created>
backend-el7	Docker	PuppetApply	Shell	Rsync	<Not Created>
database-el6	Docker	PuppetApply	Shell	Rsync	<Not Created>
database-el7	Docker	PuppetApply	Shell	Rsync	<Not Created>

- Kitchen composes test run (Instance column) out of suite + platform definitions
- Each puppet type (specified as suite) is going to be tested on each specified platform

Now We Can Make It Test Driven!

Having it all combined we can write infra code with test-first approach

- Write serverspec expectation for new code
- $ kitchen verify <type>
- Observe related test is red
- Write the puppet code
- $ kitchen converge <type> # reapply the modified puppet code
- $ kitchen verify <type>
- Observe related test is green
- Commit the changes and create PR to puppet repo

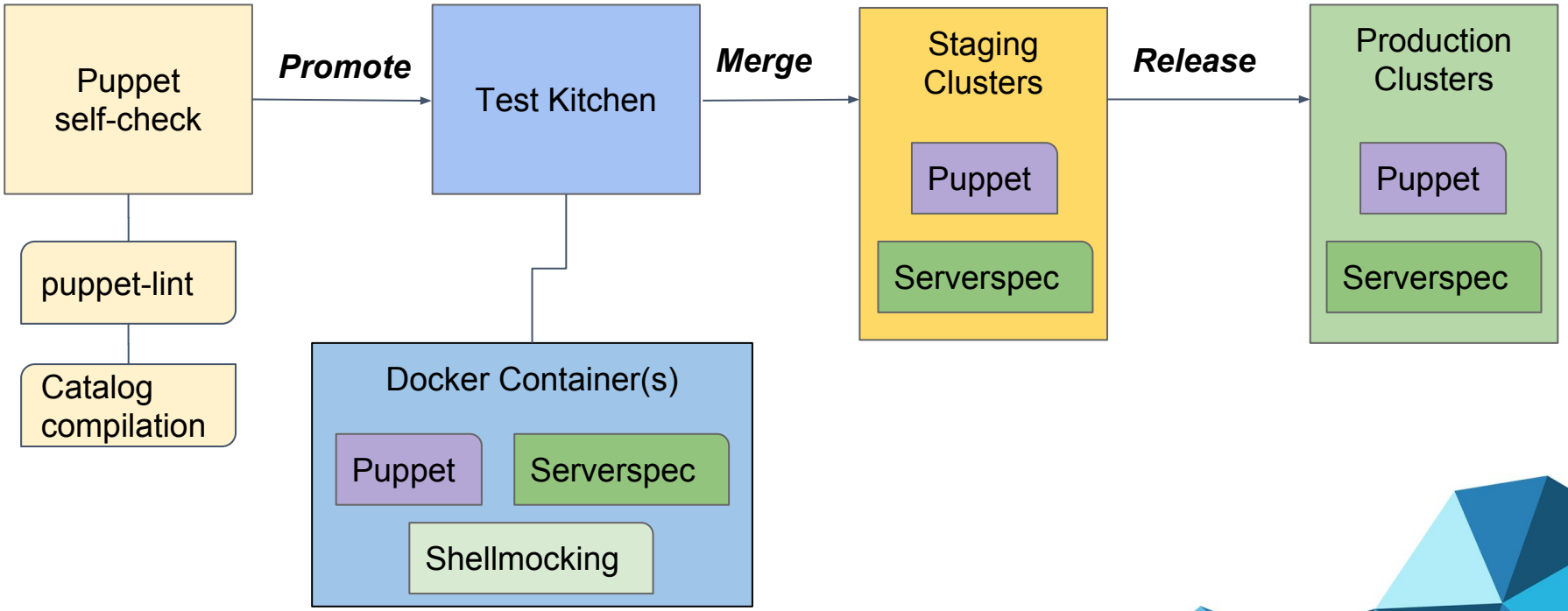
TDD Has to Scale

Technically and culturally

- Infrastructure testing should be integral part of development process and supported by strong testing culture within organization
- We test on a developer desktop even **before** git commit
- After commit/PR but **before** the merge we need to trigger test pipeline
- Serverspec test suite should be reused later in the pipeline **after** the merge on actual staging servers
- And ultimately test suite should travel to production servers **after** release for continuous regression self-testing integrated with monitoring

Infrastructure Deployment Pipeline

Puppet-self-check + Test Kitchen



Next Challenge: Test Only Affected Types

And spare resources and time for Pull Request testing

- We have more than 100 puppet types
- It is acceptable number for puppet-self-check because it is fast and does not consumes lot of resources
- It is different for Test Kitchen: we **cannot** spawn more than 100 containers for each Pull Request! It is suboptimal and resource demanding

- We need to figure out affected puppet types in automated way

Getting Diff From Puppet Code Change

How to actually deduct affected types?

- Puppet build complex graph out of
  - Interdependent modules
  - Hiera variables
  - Autogenerated custom facts
- So a mere file based diff will not help
- Puppet generated graph is too complex to reliably analyze

- Solution
  - Compare compiled catalogs
  - Create affected type list
  - Feed the list to Test Kitchen

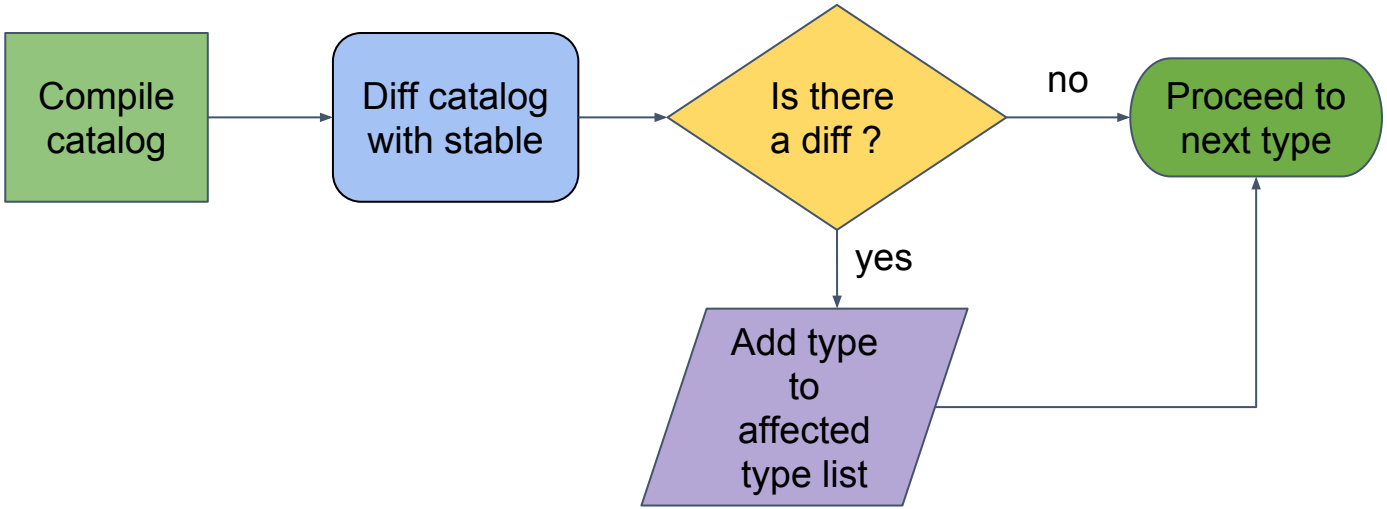
Introducing Puppet-Catalog-Diff

Yet another open source tool

- https://github.com/acidprime/puppet-catalog-diff
- Two puppet catalogs as an input
- Compares catalog json
- Diff as an output
- Helps to catch
  - Unexpected modifications in seemingly unrelated types
  - Idempotency issues

Extending Puppet-Self-Check

With the puppet-catalog-diff



Additional Goodness: Feedback right in PR

Code self reflection for developer

Affected puppet types

Click on the type for a detailed catalog diff

- type-jenkins-docker-slave.pp
- type-jenkins_ng_slave.pp
- type-kitchen.pp

```
--------------------------------------------------------------------------------
catalog_type-jenkins-docker-slave.pp                          0.416596902766623%
--------------------------------------------------------------------------------
Catalog percentage removed:    0.59
Old version:   1474558901
Catalog percentage changed:    0.00
Only in old:
       package[ruby193-ruby-devel]
       package[gcc-c++]
       package[ruby193-rubygem-bundler]
       package[gcc]
       package[libxml2-devel]
New version:   1474561369
Node differences:      7
Added and removed resources:   +2 / -5
Total resources in old:        841
Only in new:
       exec[pull-kitchen-image]
       file_line[kitchen-alias]
Total resources in new:        838
Node percentage:       0.416596902766623
Catalag percentage added:      0.24

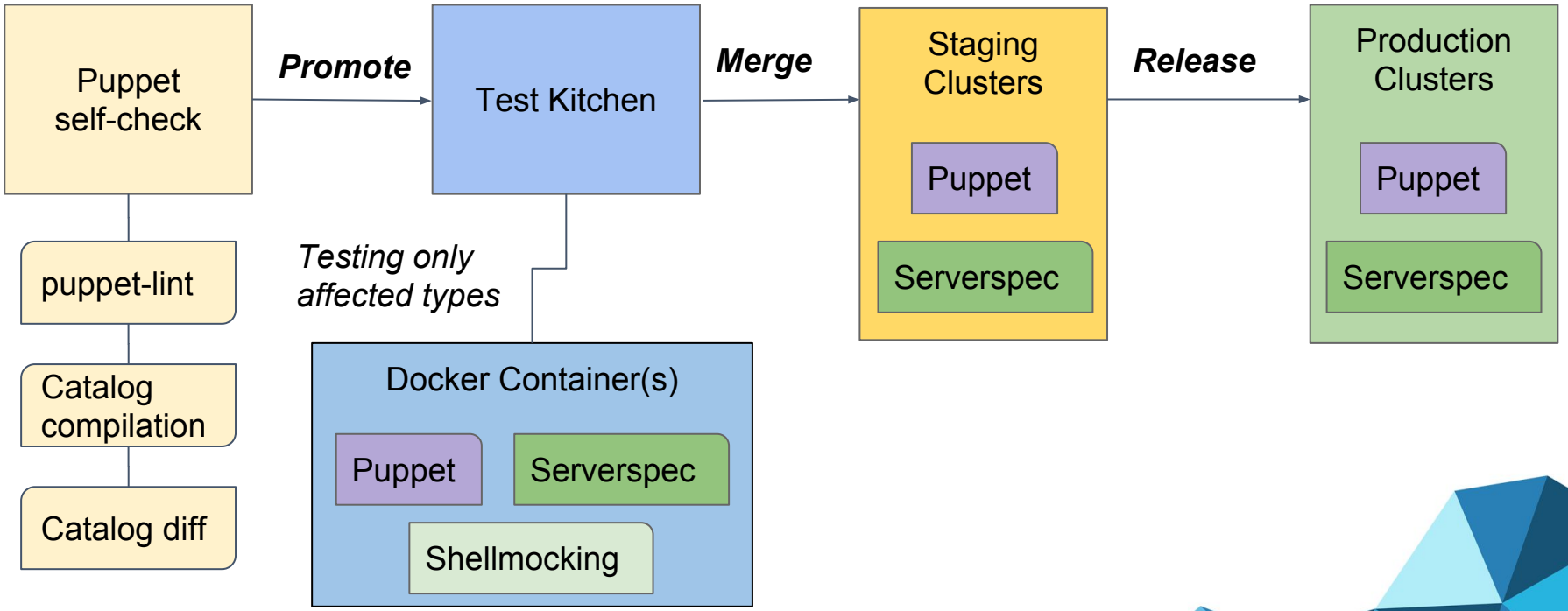
--------------------------------------------------------------------------------
1 out of 1 nodes changed.                                     0.416596902766623%
--------------------------------------------------------------------------------

Nodes with the most changes by percent changed:
1. catalog_type-jenkins-docker-slave.pp                                   0.42%

Nodes with the most changes by differeces:
1. catalog_type-jenkins-docker-slave.pp                                      7false
```

Final State of Pipeline

With the catalog diff extension



Side note on CI/CD tooling set

What do we use to orchestrate pipelines

- Jenkins - popular open source automation server
- Jenkins Job Builder - takes simple descriptions of Jenkins jobs in YAML or JSON format and uses them to configure Jenkins
    - Everything under GIT version control
- Zuul is used to gate the source code repository of a project so that changes are only merged if they pass tests.
    - Check pipeline for fast (most frequently unit) tests
    - Gate pipeline for longer resource demanding (integration) tests before actual merge
    - Optimistic merge strategy enables testing multiple Pull Requests in parallel

Replaceability

A note on agnostic approach

- Kitchen driver is agnostic to virtualization solution/cloud provider
- Kitchen provisioner is agnostic to configuration management solution
- Serverspec is agnostic to configuration type at all
- Capability to make brave movements in future!
  - Major upgrades like Puppet 3.x to 4.x
  - Change of configuration management solution
  - Using drivers other than Docker for different pipelines (e.g. openstack driver to test VM image bootability and configuration)
- Same framework can be used for completely different projects and solutions
  - Chef, ansible, salt, different cloud providers and even different test frameworks - the Test Kitchen approach will be still relevant

OSS, Upstream

Open source and contributions

- Multiple open source projects combined
  - Different upstreams diversifies the overall ecosystem
  - Different maintainers - different level of upstream response
  - Can be both good and bad
- GoodData has contributed multiple patches to related kitchen projects and serverspec
- GoodData has open-sourced internal implementations
  - https://github.com/gooddata/serverspec-core
  - https://github.com/gooddata/serverspec-ui
  - https://github.com/gooddata/shellmock

What Could be Done Better

If we would start from scratch

- More modular Puppet, less interdependencies
- More investment into unit tests, see rspec-puppet
- Reduction of number of puppet types with Roles & Profiles pattern

Benefits Recap

of Puppet, Test Kitchen, Docker, Serverspec tandem

- Scratch testing environment
- Testing infra in isolation
- Easy to test permutations
- Resource Efficiency
- Replaceability

- TDD for Infra
- Short feedback loop
- Naturally growing regression test suite
- Easily pluggable into CI/CD pipeline
- Open Source

Now you know how we build test driven infra
with open source tool set.
You can reuse it fully or partially with
adaptation to your specific case.
Hopefully see you upstream!

We are hiring!

A clear conference time abuse

- If you like challenges described above visit http://www.gooddata.com/company/careers
- In case you find something attractive don't hesitate to drop me a line directly to yury.tsarev@gooddata.com

Recommended Reading and Questions

To not to create useless ‘questions?’ slide

- Books that are full of practical wisdom
  - Puppet Best Practices
  - Infrastructure as Code
  - Test-Driven Infrastructure with Chef
- Slidedeck: https://goo.gl/QGBlSj
- **Thanks for not running away till the end of the talk! Questions?**