# Introduction to CFEngine 3

Riccardo Murri <riccardo.murri@uzh.ch>

13/03/2015

# What is CFEngine?

*"CFEngine is a software solution that helps system administrators and other stakeholders in the IT organization become more agile and respond faster to business requirements while ensuring SLAs and regulatory compliance, through automation." —*
*http://cfengine.com/learn/what-is-cfengine/*

# *Huh?* What *the heck* is CFEngine?

*"Cfengine, or the configuration engine is an agent / software robot and a high level policy language [...] to administrate and configure large computer networks." — CFEngine v2,* https://www.gnu.org/software/cfengine/

*"For many users, CFEngine is simply a configuration tool – i.e. software for deploying and patching systems according to a policy." — CFEngine v3,* https://docs.cfengine.com/latest/guide-introduction.html

# Disclaimers

1. CFEngine works on GNU/Linux, *BSD, Solaris, and Windows. *My experience is limited to GNU/Linux only.*
2. CFEngine comes in a free/open-source version ("CFEngine community") and a commercial version ("CFEngine Enterprise"), with more features and professional support. *My experience is limited to the "CFEngine community" only.*

# Alternatives?

Puppet — a better CFEngine 2 ;-), written and extensible in Ruby.

Chef — a Ruby framework for managing systems.

SaltStack — A configuration management and remote execution framework. Conceptually very similar to Puppet, but written and extensible in Python.

Ansible — not quite an alternative: it's more geared to software deployment than configuration management. . .

# (Theory)

# So, how does it work?

> *"The idea of CFEngine is to create a single file or set of configuration files which will describe the setup of every host on your network. [...] the* ***configuration of the host is checked against this model and, if necessary, any deviations are fixed.***"[1]

The host configuration ("model") is built from atoms called **promises**.

[1] https://www.gnu.org/software/cfengine/cfdetails.html

# The agent workflow

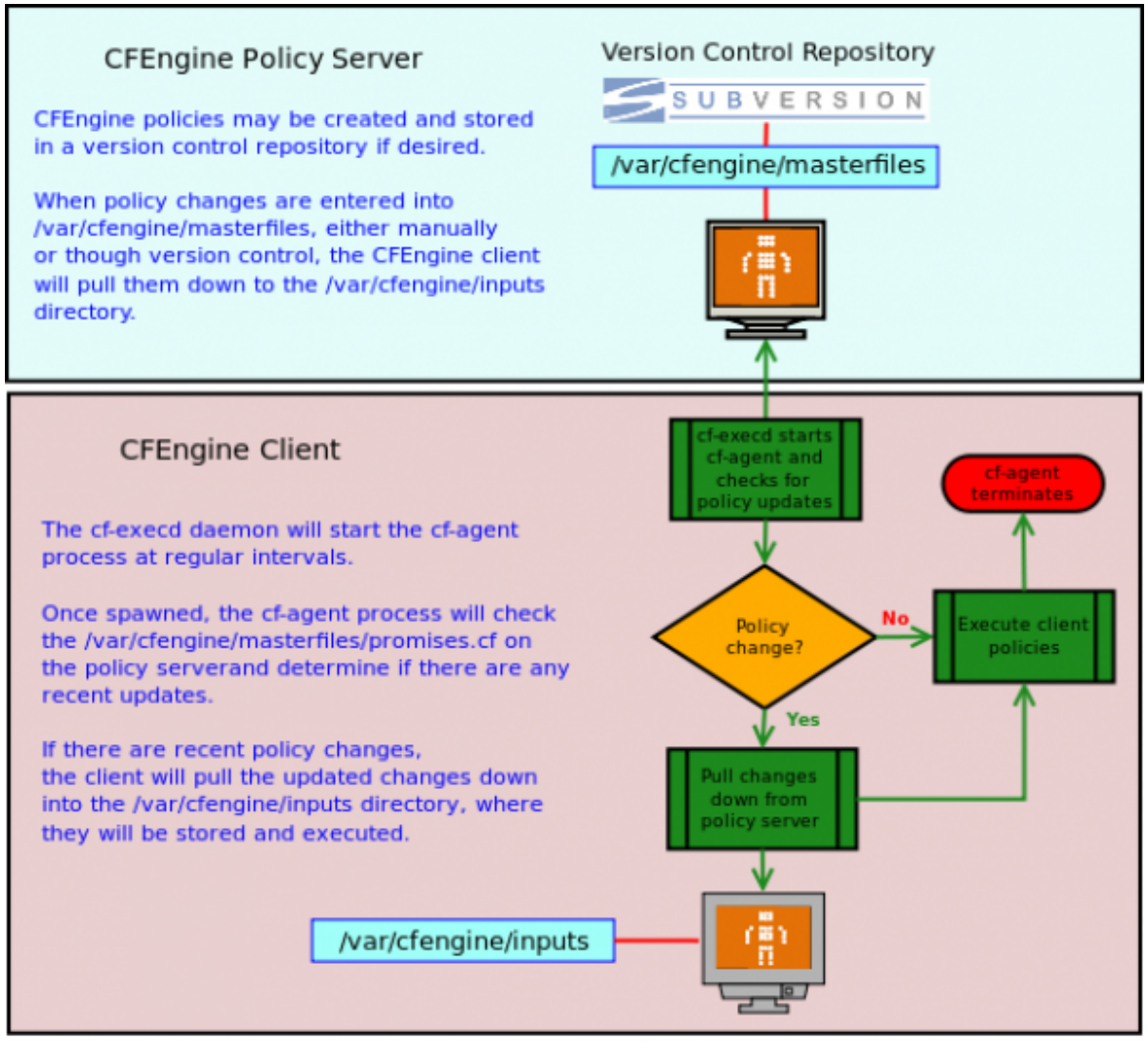



Image source: http://cfengine.com/

# Promises

"A promise is the documentation or definition of an intention to act or behave in some manner. They are the rules which CFEngine clients are responsible for implementing."[2]

```
files:
  "/tmp/email.[a-z0-9]+"
  file_select   => days_old("10"),
  delete        => tidy,
  comment       => "Delete temporary email files";
```

[2] https://docs.cfengine.com/latest/guide-writing-and-serving-policy.html

# Promises

"A promise is the documentation or definition of an intention to act or behave in some manner. They are the rules which CFEngine clients are responsible for implementing."[2]

```
files:
  "/tmp/email.[a-z0-9]+"
  file_select    => days_old("10"),
  delete         => tidy,
  comment        => "Delete temporary email files";
```

Target of the promise. A set of files in this case.

[2] https://docs.cfengine.com/latest/guide-writing-and-serving-policy.html

# Promises

"A promise is the documentation or definition of an intention to act or behave in some manner. They are the rules which CFEngine clients are responsible for implementing."[2]

```
files:
   "/tmp/email.[a-z0-9]+"
   file_select    => days_old("10"),
   delete         => tidy,
   comment        => "Delete temporary email files";
```

Additional criteria for building the promise target.

[2] https://docs.cfengine.com/latest/guide-writing-and-serving-policy.html

# Promises

"A promise is the documentation or definition of an intention to act or behave in some manner. They are the rules which CFEngine clients are responsible for implementing."[2]

```
files:
   "/tmp/email.[a-z0-9]+"
   file_select    => days_old("10"),
   delete         => tidy,
   comment        => "Delete temporary email files";
```

Action to take. In this case, delete all ("tidy").

[2] https://docs.cfengine.com/latest/guide-writing-and-serving-policy.html

# Promises

A promise has three possible outcomes:

| **state is. . .** | ▼state was. . .▼ | |
|---|---|---|
| ▼ | as promised | not as such |
| **as promised** | **kept** | **repaired** |
| **not as such** | *(impossible!)* | **not kept** |

# Bundles, I

Promises must be grouped into **bundles**.

```
bundle agent test {
  vars:
    "secrets" slist => { "/etc/shadow",
                         "/etc/sudoers" };
  files:
    "$(secrets)"
      create => "false",
      perms  => mog("0400", "root", "root");
}
```

What CFEngine component should execute this bundle. More on this later.

# CFEngine 3 components

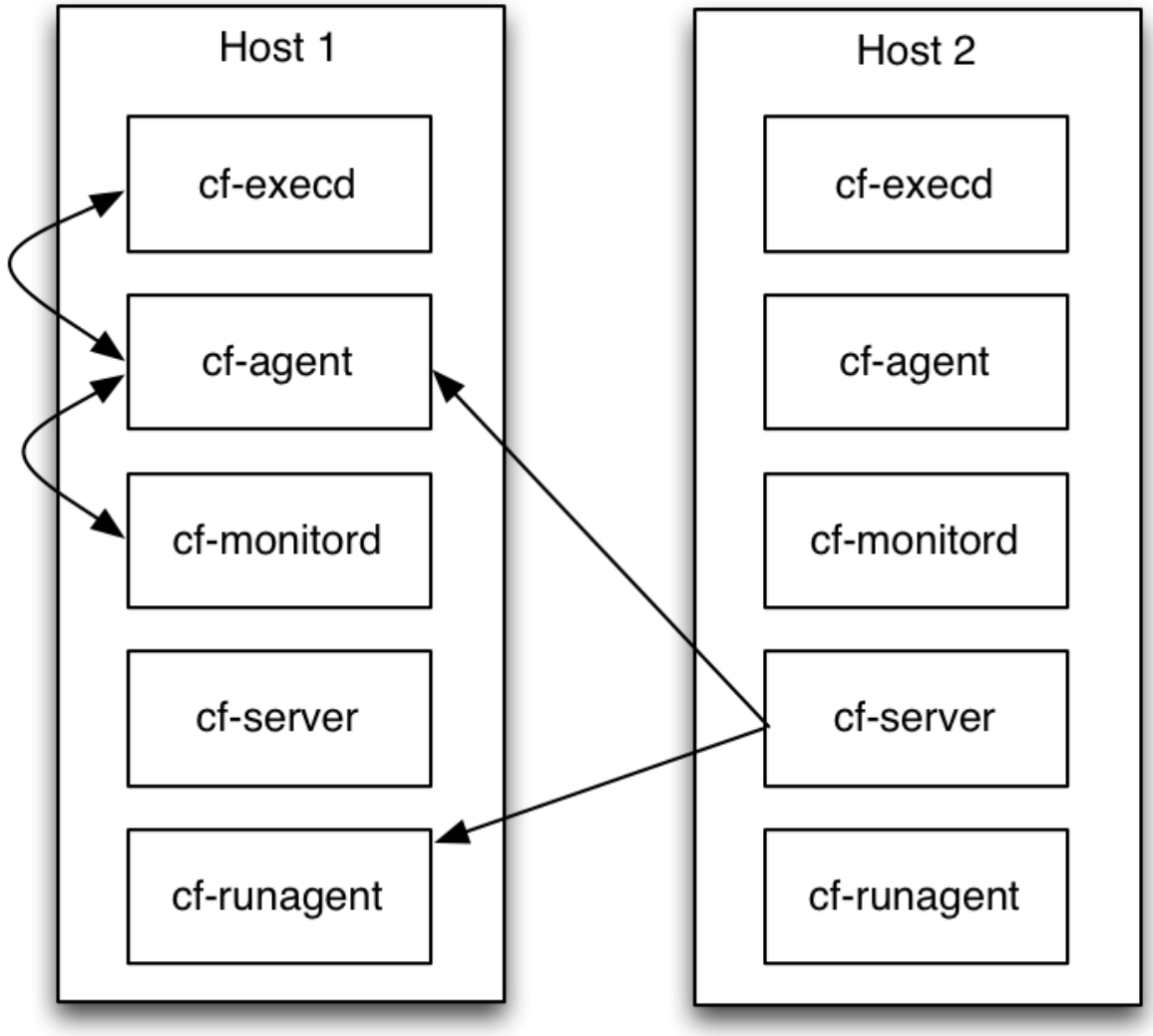



Image source: http://cfengine.com/

# (Practice)

# Example: keep secret files only readable by root

```
bundle agent keep_secrets {
  vars:
    "secrets" slist => { "/etc/shadow",
                         "/etc/sudoers" };
  files:
    "$(secrets)"
      create => "false",
      perms  => mog("0400", "root", "root");
}
```

Define `secrets` as a *list of strings*. Also available: simple strings, and (fake) integers and floats.

# Example: keep secret files only readable by root

```
bundle agent keep_secrets {
  vars:
    "secrets" slist => { "/etc/shadow",
                         "/etc/sudoers" };
  files:
    "$(secrets)"
      create => "false",
      perms  => mog("0400", "root", "root");
}
```

Since `secrets` is a *list*, this implcitly *loops* over all its values.

# Example: keep secret files only readable by root

```
bundle agent keep_secrets {
  vars:
    "secrets" slist => { "/etc/shadow",
                         "/etc/sudoers" };
  files:
    "$(secrets)"
      create => "false",
      perms  => mog("0400", "root", "root");
}
```

`mog()` = mode-owner-group. It's a standard library *body*.

# Bodies

*"the definition of a promise can grow complicated. Complex promises are best understood by breaking them down into independent, re-usable components. The CFEngine reserved word* `body` *is used to encapsulate the details of complex promise attribute values. Bodies can optionally have parameters."*

```
body perms secret {
 mode   => "0600";
 owners => { "root" };
 groups => { "root" };
}
```

```
body perms
 mog(mode, user,group)
{
 owners=>{"$(user)"};
 groups=>{"$(group)"};
 mode  =>"$(mode)";
}
```

*Reference:*

https://docs.cfengine.com/latest/guide-language-concepts-bodies.html

# Example: edit a file

```
bundle agent nss_myhostname {
  files:
      "/etc/nsswitch.conf"
      edit_line => enable_nss_myhostname;
}

bundle edit_line enable_nss_myhostname {
  replace_patterns:
      "(?x) ^hosts:\s* (((?!myhostname)\w+\s+)+)$"
      replace_with => value(
        "hosts: $(match.1) myhostname");
}
```

Editing is a complex operation, that requires a sequence of action. Therefore, it has a special kind of bundle.

# Example: edit a file

```
bundle agent nss_myhostname {
  files:
      "/etc/nsswitch.conf"
      edit_line => enable_nss_myhostname;
}

bundle edit_line enable_nss_myhostname {
  replace_patterns:
      "(?x) ^hosts:\s* (((?!myhostname)\w+\s+)+)$"
      replace_with => value(
        "hosts: $(match.1) myhostname");
}
```

Full PCRE syntax is supported everywhere in CFEngine.

# Example: edit a file

```
bundle agent nss_myhostname {
  files:
      "/etc/nsswitch.conf"
      edit_line => enable_nss_myhostname;
}

bundle edit_line enable_nss_myhostname {
  replace_patterns:
      "(?x) ^hosts:\s* (((?!myhostname)\w+\s+)+)$"
      replace_with => value(
        "hosts: $(match.1) myhostname");
}
```

CFEngine will not let you substitute a regexp with a string that matches the same regexp! Need to learn what "negative lookahead assertions" are...

# Example: edit a file

```
bundle agent nss_myhostname {
  files:
      "/etc/nsswitch.conf"
      edit_line => enable_nss_myhostname;
}

bundle edit_line enable_nss_myhostname {
  replace_patterns:
      "(?x) ^hosts:\s* (((?!myhostname)\w+\s+)+)$"
      replace_with => value(
        "hosts: $(match.1) myhostname");
}
```

Also available: `insert_lines`, `delete_lines`, `field_edits`. Which can all be combined together.

# Contexts / Classes

Contexts[3] are just boolean **constants**. The context "`any`" is CFEngine's alias for "true".

CFEngine has a concise syntax for "if" clauses using contexts:

```
vars:
  os_centos|os_rhel::
    "rules"     string => "/etc/sysconfig/iptables";
  os_debian|os_ubuntu::
    "rules"     string => "/etc/iptables/rules.v4";
```

[3] Previously called "Classes"

# Contexts / Classes

Contexts[3] are just boolean **constants**. The context "`any`" is CFEngine's alias for "true".

CFEngine has a concise syntax for "if" clauses using contexts:

```
vars:
  os_centos|os_rhel::
    "rules"     string => "/etc/sysconfig/iptables";
  os_debian|os_ubuntu::
    "rules"     string => "/etc/iptables/rules.v4";
```

This part will only be executed when the context `os_centos` or the context `os_rhel` have the "true" value.

[3] Previously called "Classes"

# Example: deploy template file, I

```
bundle agent iptables {
vars:
  os_centos|os_rhel::
    "rules"     string => "/etc/sysconfig/iptables";
  os_debian|os_ubuntu::
    "rules"     string => "/etc/iptables/rules.v4";
  any::
    "www_ports" ilist => { "80", "443" };
files:
  "$(rules)"
  edit_template  => "iptables.tmpl",
  edit_defaults  => empty,
  perms          => secret,
  create         => "true";
}
```

What template file to use.

# Example: deploy template file, I

```
bundle agent iptables {
vars:
  os_centos|os_rhel::
    "rules"     string => "/etc/sysconfig/iptables";
  os_debian|os_ubuntu::
    "rules"     string => "/etc/iptables/rules.v4";
  any::
    "www_ports" ilist => { "80", "443" };
files:
  "$(rules)"
  edit_template  => "iptables.tmpl",
  edit_defaults  => empty,
  perms          => secret,
  create         => "true";
}
```

Initial content of the editing buffer. In other words: what to do if the destination file already exists, scratch or append?

# Example: deploy template file, I

```
bundle agent iptables {
vars:
  os_centos|os_rhel::
    "rules"     string => "/etc/sysconfig/iptables";
  os_debian|os_ubuntu::
    "rules"     string => "/etc/iptables/rules.v4";
  any::
    "www_ports" ilist => { "80", "443" };
files:
  "$(rules)"
  edit_template  => "iptables.tmpl",
  edit_defaults  => empty,
  perms          => secret,
  create         => "true";
}
```

Needed to un-do the limiting effect of context `os_debian` up above.

# Example: deploy template file, II

The actual contents of the template file:

```
# iptables.tmpl
[% CFEngine www:: %]
-A INPUT -p tcp -dport $(www_ports) -j ACCEPT
[% CFEngine am_policy_hub:: %]
-A INPUT -p tcp -dport 5308 -j ACCEPT
```

This is the way contexts are applied to templates. All subsequent lines are inserted if and only if context `www` is defined.

# Example: deploy template file, II

The actual contents of the template file:

```
# iptables.tmpl
[% CFEngine www:: %]
-A INPUT -p tcp -dport $(www_ports) -j ACCEPT
[% CFEngine am_policy_hub:: %]
-A INPUT -p tcp -dport 5308 -j ACCEPT
```

Since `www_ports` is a list, this implicitly loops over all values.

# Example: deploy template file, II

The actual contents of the template file:

```
# iptables.tmpl
[% CFEngine www:: %]
-A INPUT -p tcp -dport $(www_ports) -j ACCEPT
[% CFEngine am_policy_hub:: %]
-A INPUT -p tcp -dport 5308 -j ACCEPT
```

CFEngine defines this context on the policy servers.

# Contexts / Classes, II

Additionally, contexts can be defined with specific promises, which evaluate a boolean expression:

```
classes:
  "os_debian"
  expression => "debian.!ubuntu";

  "os_debian6"
  expression => "os_debian&(debian_6|debian_squeeze)";
```

Class `os_debian` will be defined if class `debian` is defined and class `ubuntu` is not.

# Contexts / Classes, II

Additionally, contexts can be defined with specific promises, which evaluate a boolean expression:

```
classes:
  "os_debian"
  expression => "debian.!ubuntu";

  "os_debian6"
  expression => "os_debian&(debian_6|debian_squeeze)";
```

The boolean *AND* operator can be spelled as dot "`.`" or with the more familiar "`&`".

# Contexts / Classes, II

Additionally, contexts can be defined with specific promises, which evaluate a boolean expression:

```
classes:
  "os_debian"
  expression => "debian.!ubuntu";

  "os_debian6"
  expression => "os_debian&(debian_6|debian_squeeze)";
```

One of the quirks of CFEngine: context `debian` is defined on Ubuntu systems as well.

# Example: install an updated package

```
bundle agent cve_2014_0160 {
  vars:
      "to_update" slist => {"openssl", "libssl1.0.0"};
    ubuntu_12_04::
      "ok_version" string => "1.0.1-4ubuntu5.12";
    debian_wheezy::
      "ok_version" string => "1.0.1e-2+deb7u5";

  packages:
      "$(to_update)"
      package_policy  => "update",
      package_version => "$(ok_version)",
      package_select  => "<=",
      package_method  => generic;
}
```

Action to perform. Also available: `install`, `remove`, etc.

# Example: install an updated package

```
bundle agent cve_2014_0160 {
  vars:
      "to_update" slist => {"openssl", "libssl1.0.0"};
    ubuntu_12_04::
      "ok_version" string => "1.0.1-4ubuntu5.12";
    debian_wheezy::
      "ok_version" string => "1.0.1e-2+deb7u5";

  packages:
      "$(to_update)"
      package_policy  => "update",
      package_version => "$(ok_version)",
      package_select  => "<=",
      package_method  => generic;
}
```

When to perform the action. Read: if the installed version is <= of the promised version.

# Example: install an updated package

```
bundle agent cve_2014_0160 {
  vars:
      "to_update" slist => {"openssl", "libssl1.0.0"};
    ubuntu_12_04::
      "ok_version" string => "1.0.1-4ubuntu5.12";
    debian_wheezy::
      "ok_version" string => "1.0.1e-2+deb7u5";

  packages:
      "$(to_update)"
      package_policy  => "update",
      package_version => "$(ok_version)",
      package_select  => "<=",
      package_method  => generic;
}
```

Let CFEngine figure out what package manager to use.

Or you can be explicit and say: `apt`, `yum`, `rpm`, etc.

# Contexts / Classes, III

Contexts can be set depending on the outcome of single promises:

```
files:
  "/etc/locale.gen"
  create     => "false",
  edit_line => uncomment_lines_matching(
                    "(de|en|fr|it)_.+", "#"),
  classes    => if_repaired("run_localegen");
```

Context `run_locale_gen` will be set if this promise is "repaired". One should use it to conditionally run the `locale-gen` command:

```
commands:
  os_debian.run_localegen::
    "/usr/sbin/locale-gen";
```

# Example: extend CFEngine via modules

CFEngine can read back the output of any command and set/unset contexts or variables depending on it.

```
bundle local_ctx {
  commands:
    "cat locals.txt"
    module => "true";
}
```

```
# /var/cfengine/locals.txt
+pizza_box
=location[rack]=10
=location[height]=2
```

Instructs CFEngine to read and parse the command output.

# Example: extend CFEngine via modules

CFEngine can read back the output of any command and set/unset contexts or variables depending on it.

```
bundle local_ctx {
  commands:
    "cat locals.txt"
    module => "true";
}
```

```
# /var/cfengine/locals.txt
+pizza_box
=location[rack]=10
=location[height]=2
```

Sets context `pizza_box`

# Example: extend CFEngine via modules

CFEngine can read back the output of any command and set/unset contexts or variables depending on it.

```
bundle local_ctx {
  commands:
    "cat locals.txt"
    module => "true";
}
```

```
# /var/cfengine/locals.txt
+pizza_box
=location[rack]=10
=location[height]=2
```

Sets variables `location[rack]` and `location[height]`

# Example: extend CFEngine via modules

CFEngine can read back the output of any command and set/unset contexts or variables depending on it.

```
bundle local_ctx {
  commands:
    "cat locals.txt"
    module => "true";
}
```

```
# /var/cfengine/locals.txt
+pizza_box
=location[rack]=10
=location[height]=2
```

Any line not starting with +, −, or = is ignored.

# Normal ordering

> *"Within a bundle, the promise types are executed in a round-robin fashion according to so-called normal ordering [...]. The actual sequence continues for up to three iterations of the following, converging towards a final state:* `meta`, `vars`, `defaults`, `classes`, `users`, `files`, `packages`, `guest_environments`, `methods`, `processes`, `services`, `commands`, `storage`, `databases`, `reports`"

Promises in a bundle are **not** executed in the order they are written; rather CFEngine groups them by type and executes types in a fixed order.

*Reference:* https://docs.cfengine.com/latest/guide-language-concepts-normal-ordering.html

# Normal ordering

> *"Within a bundle, the promise types are executed in a round-robin fashion according to so-called normal ordering [...]. The actual sequence continues for up to three iterations of the following, converging towards a final state:* `meta`, `vars`, `defaults`, `classes`, `users`, `files`, `packages`, `guest_environments`, `methods`, `processes`, `services`, `commands`, `storage`, `databases`, `reports`"

Note this! In other words, each promise may be re-evaluated up to three times during each bundle run.

*Reference:* https://docs.cfengine.com/latest/guide-language-concepts-normal-ordering.html

# Example: send email

```
bundle agent send_email(to, subj, body) {
  vars:
      "tmpfile"
      string => execresult("/bin/mktemp", "noshell");
  methods:
      "Write body contents into temp file"
      usebundle  => append_to_file("$(tmpfile)",
                                   "$(body)");
  commands:
      "/usr/bin/mail $(to) -s $(subj) <$(tmpfile)";
      "/bin/rm -f '$(tmpfile)'";
}
```

# Example: send email

```
bundle agent send_email(to, subj, body) {
  vars:
      "tmpfile"
      string => execresult("/bin/mktemp", "noshell");
  methods:
      "Write body contents into temp file"
      usebundle  => append_to_file("$(tmpfile)",
                                   "$(body)");
  commands:
      "/usr/bin/mail $(to) -s $(subj) <$(tmpfile)";
      "/bin/rm -f '$(tmpfile)'";
}
```

This results in a new file name at each pass. Hence, the bundle never "converges" and three emails are sent!

# What other promise types are there?

| | |
|---|---|
| meta | Information about the bundle, mainly useful for documentation purposes. |
| vars | *Definition of variables* |
| defaults | "Default" values for bundle parameters |
| classes | Definition of additional contexts |
| users | Create or delete users in `/etc/passwd` |
| files | *Create, edit, or audit files* |
| packages | *Install or remove software packages* |
| guest_environments | Control VMs using `libvirt` |
| methods | Invoke other bundles |
| processes | Stop or signal running processes. |
| services | Start or stop system services. |
| commands | *Run arbitrary commands.* |
| storage | Mount NFS filesystems. |
| databases | Create, alter, or manage DBs and tables on SQL, LDAP, or MS Registry |
| reports | Print lines to the log file. |

## (The good, the bad, and the ugly) *.reverse()*

# The ugly

It's (slowly) becoming a programming language, but the syntax is verbose and awkward, and is not getting better with time and releases:

```
bundle agent sysctl_data {
vars:
  "parms_vars[net.ipv4.tcp_tw_reuse]" string => "1";
  "parms_test_file" string => "/etc/sysctl";
  "parms_debug" string => "on";
  "parms_mgmt_policy" string => "ensure_present";
methods:
  "test" usebundle => sysctl("sysctl_data.parms_");
}
```

# The bad

After more than 6 years, there are still serious bugs in the core functionality.

```
$ cat test.cf
bundle agent test {
  vars:
      "variables" slist => variablesmatching(".+");
  reports:
    linux::
      "$(variables)=$($(variables))";
}

$ time cf-agent -f ./test.cf -b test
 [...]
real 43m5.345s
```

That, and “normal ordering” which is the root of all evil.

# The good

Strives to provide *idempotent primitives*, upon which to build your own systems administration DSL.

Can react to changing conditions: facts are not gathered at the beginning of the run.

Can *really* be extended using any language.

Runs 3'200 (and counting) promises on the top of every hour on $O(100)$ hosts in the $S^3IT$ server room, and *keeps all of them.* (Really, it can deploy and configure the entire OpenStack software suite at a whim.)

# Appendix

# Further reading

Learning CFEngine 3 The one and only beginners' book.

https://docs.cfengine.com/docs/3.6/reference-promise-types.html
*The* Reference Manual, you need this every time. . .

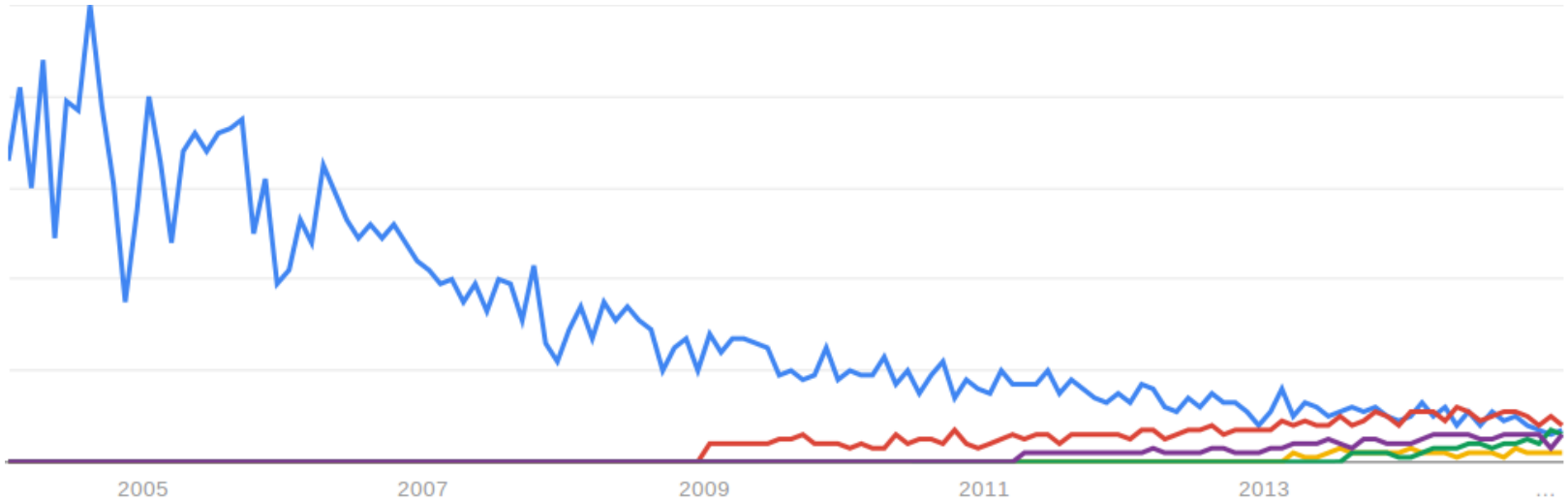
https://docs.cfengine.com/latest/guide-introduction.html What the various components are, and how they interact.

https://docs.cfengine.com/latest/guide-language-concepts.html
Overview of the CFEngine language syntax.

https://docs.cfengine.com/latest/guide-writing-and-serving-policy-promise
Overview of the promise types and best practices for structing a CFEngine code base.

https://docs.cfengine.com/latest/guide-design-center-configure-sketches-
Desgin Center / Sketches: how to re-use CFEngine code packaged and shared by others.

# CFEngine Puppet Salt Stack Ansible Chef




*Google trends: 2004–today (▲), 2009–today (▼)*