
usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION
26
Mark Burgess and Diego Zamboni
Modern Infrastructure Engineering with CFEngine 3
26
Short Topics in
System Administration
Modern Infrastructure Engineering
with CFEngine 3
Mark Burgess and Diego Zamboni
ISBN-13: 978-1-931971-98-0
90000
9 781931 971980
usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

26 *Short Topics in*
System Administration

Modern Infrastructure Engineering with CFEngine 3

Mark Burgess and Diego Zamboni

Published by the USENIX Association
2012

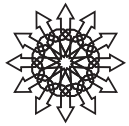
ISBN 978-1-931971-98-0

To purchase additional copies, see www.usenix.org/lisa/books.

The USENIX Association
2560 Ninth Street, Suite 215
Berkeley, CA USA 94710

www.usenix.org

Contents



1. Introducing CFEngine

As technology becomes more sophisticated,
the cost of introducing variations declines.

—Alvin Toffler, *Future Shock*, 1970

CFEngine 3 is a third-generation infrastructure automation framework, with self-healing capabilities and a desired-state, model-oriented approach. It is licensed under the GPL version 3, in an open source Community edition, and there is a commercially licensed Enterprise edition with extended verification, reporting and scalability features. CFEngine is suitable for managing systems composed of everything from a single host to hundreds of thousands of hosts, because it is designed to bring consistency and knowledge of implementation. That applies to the smallest of systems where the temptation is to make changes ad hoc, and to the largest, where it would be impossible to implement without machine assistance. CFEngine scales because it has a fundamentally decentralized and knowledge-oriented design. We say that CFEngine manages hosts "from within," because each host takes responsibility for its own state by running the CFEngine agent.

To scale systems, without losing control, you need not only efficiency but a strong knowledge of the system, which engages human understanding and participation. As of this writing, the smallest installations of CFEngine are on mobile phones, and the largest installations we know of regulate around 200,000 machines under a common administration. CFEngine can manage a great many aspects of system configuration and maintenance, including:

- Application management
- Storage management
- Service management
- Operating system management

It does this, from the bottom up, through the use of a powerful configuration engine (hence the name), steered by policy written in a Domain Specific Language for specifying self-healing change operations. Some capabilities include

- Installing and maintaining software
- Setting up and maintaining IT services
- Editing system configuration files and other files
- Creating symbolic links and aliases
- Checking and correcting file permissions, ownership and security attributes
- Deleting unwanted files and rotating logs (garbage collection)
- Compressing selected files

- Distributing files within a network
- Automatically mounting remote file systems
- Verifying the presence and integrity of important files and file systems
- Executing commands and scripts
- Applying security-related patches and similar system corrections
- Managing system server processes

By combining primitives like these into a self-maintaining model, we can build up greater predictability about our systems, and take the step towards mission-critical infrastructure.

CFEngine's purpose is to implement such a knowledge-based infrastructure through configuration management. In practical terms, this means that CFEngine greatly simplifies the tasks of system configuration and maintenance. For example, to customize a particular system, it is no longer necessary to write a program that performs each required action in a procedural language like Perl or your favorite shell. Instead, you write a much simpler policy description that documents *how* you want your hosts to be configured. The CFEngine software determines what needs to be done in terms of implementation and/or remediation from this specification. Such policy descriptions are also used to ensure that the system remains configured as the system administrator wishes over time.

Here is a brief example of such a policy description, which we have annotated:

Sample Policy Example 1: Introducing CFEngine configuration

```
bundle agent copy_and_cleanup
{
   vars:
      "tmpdirs" slist => { "tmp", "scratch1", "scratch2" };  Define a list variable.

   files:                                                    File specifications.
      "/usr/local/bin"
      comment => "Permission governance on locally compiled software",
      perms => mog("755", "root", "bin"),      File ownership and permission settings.
      depth_search => recurse("inf");              Fix this and all its subdirectories.

      "/$(tmpdirs)"                                    Clean up temporary directories.
                                        $(tmpdirs) will loop over all the values declared above.
      comment => "Policy for preventing crippling disk fill",
      delete => tidy,                                  Delete everything in the directory.
      file_select => days_old("7"),                Select things that are 7 days or older.
      depth_search => recurse("inf");

      solaris::                                  The following applies only to Solaris systems.
      "/etc/pam.d" => "security@example.com",
      comment => "PAM settings are set globally by security team",
      copy_from => remote_cp("/config/pam/solaris", "pammaster"),
                                    Copy files to the local system from the "pammaster" server.
      depth_search => recurse("inf");

      linux::                                    The following applies only to Linux systems.
      "/etc/pam.d/common-auth" => "security@example.com",
      comment => "PAM settings are set globally by security team",
      copy_from => remote_cp("/config/pam/common-auth", "pammaster");
}
```

The first **files** promise specifies that all of the files in the directory */usr/local/bin* should be owned by user root and group bin and have the file mode 755. When CFEngine runs with this configuration description it will correct any ownership and/or permissions which deviate from these specifications. Thus, this promise serves to express a policy about the proper ownerships and permissions for the executables in the local binaries directory.

The **copy_from** promises prescribe different configurations for Linux and Solaris systems. On Solaris systems, files in */etc/pam.d* will be updated with those in the directory */config/pam/solaris* on a master server when the latter are newer. On Linux systems, only the file */etc/pam.d/common-auth* is updated from the PAM master configuration. Note, however, that both of these specifications implement the same underlying system configuration maintenance policy: update the relevant PAM configuration files from the master server if necessary.

The delete promise illustrates the use of implicit looping in CFEngine. The single directive in the example applies to each of the directories in the **tmpdirs** list. For each directory, CFEngine will delete all items in the directory or any of its subdirectories which have not been accessed in seven days (including ones where the filename begins with a period). Like the other directives in this sample configuration file, this stanza implements a policy: items in temporary directories which have not been used within a week will be deleted.

All CFEngine configuration descriptions are variations on these and similar themes, albeit more elaborate ones. Before turning to more details about the technical aspects of using CFEngine, a brief consideration of the most important underlying and guiding theoretical concepts is in order.

1.1 Fundamental Concepts

As we've stated, CFEngine operates on hosts in order to bring their configurations in line with their specified promises. Here are formal definitions of what we mean by these key terms:

> ***Definition 1: Host.*** *Generally, a host is a single computer that runs an operating system like Unix, Linux or Windows. We will sometimes talk about machines too, and a host can also be a virtual machine supported by an environment such as VMware or Xen/Linux.*
>
> ***Definition 2: Policy and promises.*** *Policy is a specification of what we want a host to be like, i.e., its* ***desired state****. Rather than being any sort of computer program, a policy is essentially a piece of documentation that describes technical details and characteristics. Each statement in CFEngine is called a* ***promise*** *because, once documented, the agent will try to keep it as a promise for as long as it is defined, not just once during a build process. A CFEngine policy is a collection of promises.*
>
> ***Definition 3: Configuration.*** *The configuration of a host is the actual state of its resources, e.g., the permissions and contents of files, the inventory of software installed, and the like. It is the state of affairs on a particular host at a given time.*

What are we aiming for with CFEngine? The answer is *policy-conformant configuration*. If we can promise the desired state, we can claim a host will behave predictably. We

want to formulate a specification for one or more hosts describing their characteristics and how they all interact (perhaps to solve a business problem); then we want to leave the details, implementation and maintenance to a robot agent: **cf-agent**.

Humans are good at understanding input and thinking up solutions but not very reliable at implementation: *doing*. Machines and software agents are good at carrying out tasks reliably, but are not good at understanding or finding actual solutions. With CFEngine, you let the distinct parts of your human-computer organization concentrate on what they are each good at doing. This is a manifesto for re-humanizing IT management, so that machines work for humans, not the other way around.

1.1.1 Promises and Repairs

A CFEngine policy can be thought of as a list of promises which the system itself makes to you, or an imaginary auditor, about its configuration state. Don't think of CFEngine's language as a programming language, but rather as a documentation language. Most promises involve the possibility of *change to the system,* if the desired state is not initially met. The ability to change allows the agent to fulfill its promises continuously over time. We call such changes *actions* or *operations*. As you probably already guessed, the auditor in this scenario is part of CFEngine itself. **Cf-agent** is also the mechanic or surgeon that performs the operations on the system, if it does not meet its promises.

By describing its operation in this manner, we can think of configuration management as a service that is provided, a service that is intimately connected with monitoring and maintenance, and which can be "bought" on demand without necessarily subordinating a system to a central authority.

> ***Definition 4: Operation.*** *A unit of change is called an operation. CFEngine deals with changes to a system implicitly: operations are embedded into the basic sentences only by the implication of keeping a promise about system state.*

For example, here is a promise about the attributes of a file:

```
files:
 "/etc/passwd"
 perms => mog("a+r,go-w","root","root";
```

There are implicit operations (actions) in this declaration: specifically, the operations that will change the attributes if/when they do not conform to this specification.

> ***Definition 5: Outcome.*** *The outcome of a promise is how we can assess its state after CFEngine has attempted to verify it. The outcome of any promise can be one of three possibilities:*
>
> • *Promise kept (was and is ok)*
>
> • *Promise not kept (not ok)*
>
> • *Promise repaired (was not but now ok)*

CFEngine 3 uses these categories very consistently when reporting on the state of the system. Clearly, having a promise kept is closely related to the concept of system *compliance,* measured in relation to a specification. Thus it is very easy to create compliance frameworks written as CFEngine promises.

1.1.2 Convergence

A key property of CFEngine is convergence. This is an important characteristic that distinguishes it from general computer languages. It is a property that helps to prevent systems from diverging: running away in an uncontrollable fashion.

> ***Definition 6: Convergence.*** *An operation is convergent if it always brings the configuration of a host closer to its promised state, and has no effect if the host is already in that state. We can summarize this in functional terms by the following meta-rules:*
>
> *CFEngine(any state) -> desired state*
>
> *CFEngine(desired state) -> desired state*
>
> *We shall sometimes call a "desired state" a "healthy state," using the metaphor that a badly configured host is suffering from a kind of sickness.*

Here is an example used during the editing of an ASCII file:

```
"/path/myfile".
edit_line => append_if_no_line("Important configuration line");
```

This operation tells CFEngine to append the given text to the end of a file, only if it is not already there. The policy-conformant configuration is therefore that the line is present, and once that is achieved nothing more will be done. We say that the operation **append_if_no_line** is convergent.

Don't underestimate the value of convergence. It provides you with stability and thus *predictable knowledge* about your system. Because CFEngine's language interface strongly discourages you from doing anything non-convergent, it also helps to prevent mistakes. The price is that you will have to learn to think in a convergent way—and that is new for most people who come to CFEngine for the first time.

1.1.3 Classes, Contexts and Declarations: From One to Many Hosts

One of the features that makes CFEngine policies readable is the ability to hide away all of the complex decision-making that needs to be performed by the agent. To realize this ambition, CFEngine uses a *declarative* language to express policy.

A declarative language is not like a flow-chart; it is more like an inventory of intent. In an imperative language, one focuses on the procedure. In a declarative language, one focuses on the intention, or the presumed result.

One example of this is the use of classes, or context expressions, in CFEngine. Classes and contexts are a way of making decisions, without writing many "if-then-else" clauses. A class is an identifier which has the value "true" when a particular test is true. It is a kind of Boolean variable; if you like, it caches the result of an "if" test whose value was discovered by probing the system. A class is used to limit the scope of CFEngine actions to the appropriate system(s) and/or under the appropriate conditions, i.e., to say when and where promises should be kept.

The benefit of classes is that all of the testing can be hidden away in the bowels of CFEngine, and only the results need be visible if or when they are needed.

> ***Definition 7: Classes.*** *A class is a way of slicing up and mapping out the complex environment of one or more hosts into regions that can then be referred to by a symbol or name. They describe scope: where something is to be constrained.*

For example, the class **debian** is true if and only if **cf-agent** is running on a host that has Debian GNU/Linux as its operating system.

1.1.4 Voluntary Cooperation

Another fundamental property of CFEngine components is that every host retains its individual autonomy. A host can always opt out of CFEngine-based governance if its administrator wants to. This principle leads to a fundamental design and implementation decision:

> ***Definition 8: Autonomy.*** *No CFEngine component is capable of receiving information that it has not explicitly asked for itself.*

It is important to understand what this means. It does not mean that centralized control of hosts cannot be achieved. Centralized control is the way that most users choose to use CFEngine. Indeed, all you have to do to achieve centralized control is to make a policy decision for all your hosts to fetch policy specifications from a central authority.

Autonomy does mean that if your environment has some small groups or sub-cultures with special needs, it is possible for them to retain their special identity. No one claiming to be their own self-appointed authority can ride roughshod over their local decisions.

Where does policy come from then? Each host works from a policy specification that CFEngine expects to find in a local directory (usually */var/cfengine/inputs* on a Unix-like host). If you want your host to be controlled from some central manager or authority, then your policy must contain bootstrapping specifications that say: "It is my decision that I should download and follow the policy specification located at the central manager."

Each host can turn this policy decision off at any time. This is a key part of the CFEngine security model.

1.1.5 Scalability

CFEngine is designed to be scalable at a low cost. Its scalability is at least as good as any other system, because it allows for maximal distribution of workload. Moreover, because it is very lightweight and has few dependencies, very little hardware or software is required to grow a system to thousands of hosts.

> ***Definition 9: Scalable distributed action.*** *Each host is responsible for carrying out checks and maintenance on/for itself, based on its local copy of policy.*

Being designed for scaling does not mean that you are immune from making bad decisions. For example, network services can always be a bottleneck if you ask 10,000 hosts to fetch something from one place at the same time.

The fact that each CFEngine agent keeps a local copy of policy (regardless of whether it was written locally or inherited from a central authority) means that CFEngine will continue to function even if network communications are down.

1.2 CFEngine Components

The CFEngine software consists of a number of components: separate programs that work together (see Figure 1.1).

The components of CFEngine are:

- **cf-agent**: Interprets policy promises and implements them in a convergent manner. The agent can use data generated by the statistical monitoring engine **cf-monitord** and it can fetch data from **cf-serverd** running on local or remote hosts.
- **cf-execd**: Executes **cf-agent** and logs its output (optionally sending a summary via email). It can be run in daemon (stand-alone) mode, or it can be run from **cron** on a Unix-like system.
- **cf-serverd**: Monitors the CFEngine port: serves file data and starts **cf-agent** on receipt of a connection from **cf-runagent**. Note that no data can be passed to this daemon.
- **cf-runagent**: Contacts remote hosts and requests that they run **cf-agent**.
- **cf-monitord**: Collects statistics about resource usage on each host for monitoring and for anomaly detection purposes. The information is made available to the agent in the form of CFEngine classes so that the agent can check for and respond to anomalies dynamically.
- **cf-key**: Generates public-private key pairs on a host. You normally run this program only once, as part of the CFEngine software installation process.
- **cf-report**: Dumps the **cf-agent** database contents in various formats, should you become interested in its internal memory.

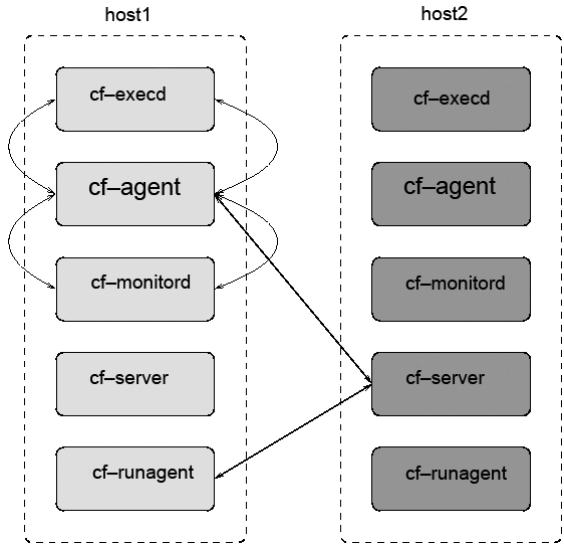


Figure 1.1: CFEngine Components and the Connections Between Them

Figure 1.1 illustrates the relationships among CFEngine components on different hosts. On a given system, **cf-agent** may be started by the **cf-execd** daemon; the latter also handles logging during **cf-agent** runs. In addition, operations such as file copying between hosts are initiated by **cf-agent** on the local system, and they rely on the **cf-serverd** daemon on the remote system to obtain remote data.

1.3 Getting Started

In this section, we'll get CFEngine installed and running. You should get the CFEngine components working with a trivial policy before trying to understand the details of the language, just to get the engine ticking over. Later, when you have understood its operation, you can build up your policy step by step.

1.3.1 Setting Up Your First CFEngine Host

You should start from a blank system. If you have been using CFEngine Community Edition and you have already developed a policy; set aside this policy during the installation process. You will be able to integrate it back later.

For performing these exercises, you can get a free license for CFEngine Enterprise, for managing up to 25 hosts, from http://cfengine.com/25free.

The Enterprise edition is provided in two packages: the main software package must be installed on every host (including the policy-server or hub). The expansion package is only installed on the policy hub. You should install and set up the hub first.

Verify that the machine's network connection is working. On the hub, verify that the package manager for your system is working (e.g., apt-get update) and install the package.

cfengine-3.xxx.[rpm | deb | etc]

Red Hat or SuSE families:

host# rpm -ihv packages

Debian family:

host# dpkg --install packages

On the hub, a public key has now been created in */var/cfengine/ppkeys/localhost.pub* as part of the package installation. As a commercial customer, you should send this public key to CFEngine Support as an attachment in the ticket system to obtain a license file license.dat. You do not need to do this for using the 25free license, as it is automatically enabled.

Save the returned license file to */var/cfengine/masterfiles/license.dat* on the hub before continuing.

Decide on the hostname and IP address of your hub (policy server); here we assume "10.10.10.1" is the address.

hub # /var/cfengine/bin/cf-agent --bootstrap --policy-server 10.10.10.1

Use the same command on all hosts, i.e., **do not bootstrap the policy server with a localhost address**. If you mistype the address of the hub, we recommend doing the following steps to re-bootstrap.

hub # /var/cfengine/bin/cf-agent --bootstrap --policy-server 10.10.10.1
hub # killall cf-execd cf-serverd cf-monitord cf-hub
*hub # rm -rf /var/cfengine/inputs/**
hub # rm -f /var/cfengine/policy_server.dat
hub # /var/cfengine/bin/cf-agent --bootstrap --policy-server 10.10.10.1

CFEngine will output diagnostic information upon bootstrap. Error messages will be displayed if bootstrapping failed: pursue these to get an indication of what went wrong and correct accordingly. If all is well you should see the following in the output:

-> Bootstrap to 10.10.10.1 completed successfully

CFEngine should now be up and running on your system. It will copy its default policy files into */var/cfengine/masterfiles* on the hub (policy server). When the clients are bootstrapped, they will contact the hub and copy them to their inputs directories. Because the policy server is a client of itself, those files will also be copied to */var/cfengine/inputs/* on the policy server.

1.3.2 Simple Policy Test

You continue by editing policy for hosts in the root file *promises.cf* in the masterfiles directory on the policy server.

Before doing this, let's just make sure that the software is working by executing a manually created, self-contained "hello world" promise. Create a file with the following content called, say, test.cf in your current directory.

Policy Example 2: Trivial policy for initial testing

```
body common control
{
bundlesequence => { "test" };
}
bundle agent test
{
reports:
   cfengine_3::
      "Danger, Will Robinson!";
}
```

Now try, as root, the command:
/var/cfengine/bin/cf-agent -f ./test.cf

You should see:
R: Danger, Will Robinson!

This is all you need to test CFEngine. The policy is a simple one: it simply promises to print out a message on any host running any version of cfengine_3. Test this now by running the agent. The agent will look for the *promises.cf file* by default, i.e., if you don't use the **-f** option on the command line.

You will not normally need to activate **cf-agent** manually. The background service **cf-execd** automatically schedules **cf-agent** to wake up and run every five minutes. However, you are always free to do so, without causing harm to the system. This is very useful for testing new policies during development.

Congratulations, you have now successfully used CFEngine.

Keep this in mind: Everything you do in CFEngine 3 is about making and keeping promises.

1.3.3 What's Next?

Starting from this simple policy being enforced on a single host, you can build up your CFEngine implementation, expanding it both to include more hosts and to place more aspects of system configuration and maintenance under CFEngine control. We will consider these two activities separately in the chapters that follow.

1.4 CFEngine Architecture

CFEngine does not have one and only one possible architecture. You are free to build any kind of architecture you like. Most users follow the same basic patterns, however, and build small enclaves of governance around "central" hubs. They may or may not then federate these hubs, or try to build a single framework for everything.

By standardizing around the idea of hubs, we can simplify the deployment of infrastructure, and we expect certain components to be in place:

- There is a single place where policy is written (usually around a version control system).
- There is a place where policy is tested.
- There is a place where new policies are dropped to be deployed to production.

In the default CFEngine model, policy is written around some kind of version control repository that is outside of your production system. You should never change the promises that are in "live" production without offline review. To do so would be to connect the possibility of human error directly to your production environment.

Subversion or git are fine possibilities for version control. Version control repositories provide access control to change policy too, so you can authorize only certain people to make changes.

To write a new policy, you edit a copy of the master policy and test if using the **cf-promises** syntax checker. Typing **cf-promises –inform** will also give you help in identifying possible errors that go beyond mere syntax, e.g., conflicting promises.

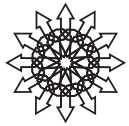
Once a policy is approved for deployment, you would drop it into the policy distribution point on the policy server:

/var/cfengine/masterfiles

This then gets copied by CFEngine itself to the policy cache

/var/cfengine/inputs

on each client, where the policy is kept until any update can be detected at the distribution point. CFEngine maintains binaries in */var/cfengine/bin* and policy under */var/cfengine/inputs,* and this makes it as robust as possible against network failures.



2. CFEngine Language

In this chapter, we'll consider what making promises for CFEngine looks like in more detail.

CFEngine's "API" or application programming interface is its promise language. You can't call CFEngine functions, like a library, as with typical programming languages, because CFEngine is not a "do it once" scripting language, but a self-maintaining design specification for your infrastructure. Think of it more as *active documentation* than as a programmed algorithm.

Only **cf-agent** can really keep promises in the way CFEngine is designed to work. The CFEngine agents have to be running as a service on each host to maintain these self-healing properties.

To use CFEngine, you write promises about the "desired state" you intend for your infrastructure, in the CFEngine language, and the agents do their best to implement it, without human intervention.

As a user of CFEngine, you naturally want to move beyond repeating copied "recipes" to a state of maximal understanding, since mastering tools is a prerequisite for ensuring the predictability—and therefore the security—of your site.

Everything in CFEngine has a very consistent grammar, because everything you express in CFEngine is considered to be a promise, and promises follow a generic pattern. At the highest level, the CFEngine grammar consists of multiple declarations of the following simple form:

```
bundle agent-name user-defined-name
{
...
promise-type:
   class-expression::                    Classes are optional, defaulting to the "any" class.
      "affected object" -> { list of stakeholders or promisees },
        attribute_group_1 => value1,
        attribute_group_2 => value2,
        …..
        attribute_group_1n=> value;
...
}
```

CFEngine has several component-agents, and each can have bundles of promises to keep, enclosed by curly braces. The bundle types are named: *common, agent, server, monitor, executor,* etc. Not all promise-types can be used in all bundles. Bundles for server, monitor and knowledge agent have their own capabilities and hence their own types of promises that only make sense there.

At the lower level, most rules follow this general pattern:

```
target option => value, option => value ...
```

where the various options control when and how the target item is validated and/or modified. You can also write at a higher level, and hide the details of object attributes under layers of abstraction.

2.1 General Notes about the Language

The following are the most important characteristics of the CFEngine policy language:

- It is a free-format language. Rules can extend over as many lines as is necessary or desired. Indentation is conventionally used for readability.
- Variables and lists are de-referenced using the following syntax: *${name}* or *$(name)*.
- Comments use the shell syntax; a # sign marks the remainder of the line as a comment.
- There is a lot of provision for adding metadata around promises, that is not directly functional, but which aids in the management of knowledge.

The default input file that CFEngine looks for is called *promises.cf* and is located in the current-users work-directory. For users other than root, the work-directory is a special directory called *.cf-agent* in your home directory. For the root user, the work directory is */var/cfengine.* Every **cf-agent** input file must have a part that explains what promises the agent will keep, and where to find the necessary descriptions.

```
body common control
{
 bundlesequence => { "some_promises", "another" };
 inputs => { "otherfile.cf", "/special/directory/file.cf" };
}
# definitions of promise-bundles
bundle agent some_promises
{
…
}
```

Here is a simple, high-level excerpt of CFEngine language:

```
bundle agent service_catalogue
{
services:
     webservers::
        "www";
        "mysql";
     any::
        "name_resolution";
        "ntp";
}
```

And here is a brief low-level excerpt, which illustrates how high-level concepts can be broken down into patterns of elementary promise primitives. It contains some initial settings and definitions and two policy rules: one of type **files** and one of type **packages.**

```
bundle agent some_promises
{
vars:
   "match_package" slist => {                              Define a list variable
                            "apache2",
                            "php5"
                            };
files:

   webservers.redhat::
      "/etc/sysconfig/apache2",                                        Edit a file
        comment => "Main web-site config is imported here",
        edit_line => append_if_no_line
                            (
                            "APACHE_CONF_INCLUDE_FILES=\"/repo/local.conf\""
        ),
      edit_defaults => std_defs;

packages:
   webservers.redhat::
      "$(packages)"                          For all packages in the list, ensure installed
        package_policy => "add",
        package_method => yum,
        action => if_elapsed("360");          Check only if 360 minutes since last check
}
```

This bundle is "executed" by **cf-agent,** if it is included in the `bundlesequence` meaning that **cf-agent** looks at the declarations and tries to keep the promises it finds.

The **files** section illustrates the use of comments and of a compound class expression: in this case, two classes joined by logical AND, denoted by a period. It is also allowed to use the less-readable ampersand (&). The promiser in this section is */etc/sysconfig/apache2.* This file is ensured to contain a line that imports some local Apache configuration definitions contained in the file */repo/local.conf.* It also computes an MD5 checksum for each file and compares it to a stored value in */var/* verifies that the Apache configuration file imports some local definitions contained in another file */repo/local.conf.*

2.2 Example of Using Templates

Templating files is a relatively primitive approach to adapting configurations to a context, but if that approach works for you, there is no reason to make it more complicated than that. File templating is easy in CFEngine 3. It starts with a files promise. Let's maintain a file called *file_based_on_template* by basing it on a template. We start by making a promise that the file will be based on a template.

```
bundle agent templating
{
files:
 "/home/mark/tmp/file_based_on_template"
      create => "true",
   edit_template => "/tmp/source_template";
}
```

Then we need to create the input file. This file can contain CFEngine variables. Suppose, for example, the source template file looked like this, with embedded CFEngine variables:

```
mail_relay = $(sys.fqhost)
important_user = $(mybundle.variable)
#...
```

These variables will be filled in by CFEngine if they are defined within your configuration scope.

The result will be something like this:

```
mail_relay = cfhost-104
important_user = Contents of variable
```

If you use the **edit_template** promise, you can embed directives to CFEngine context-classes and mark out regions of a file to be treated as an iterable block.

```
#This is a template file /templates/input.tmpl

These lines apply to anyone

[%CFEngine solaris.Monday:: %]
Everything after here applies only to solaris on Mondays
until overridden...

[%CFEngine linux:: %]
Everything after here now applies now to linux only.

[%CFEngine BEGIN %]
This is a block of text
That contains list variables: $(some.list)
With text before and after.
[%CFEngine END %]

nameserver $(some.list)
```

For example: if we use this template in a promise:

```
bundle agent test
{
vars:
   "var" slist => { "1", "2", "3"};
files:
   "/tmp/expander"
      create => "true",
   edit_template => "/templates/input.tmpl";
}
```

The result would look like this, on a Linux host:

```
#This is a template file /templates/input.tmpl

These lines apply to anyone
Everything after here now applies to Linux only.
This is a block of text
That contains list variables: 1
With text before and after.
This is a block of text
That contains list variables: 2
With text before and after.
This is a block of text
That contains list variables: 3
With text before and after.
nameserver 1
nameserver 2
nameserver 3
```

You can imagine using this in a less artificial example, to add hosts to a web server configuration:

```
[%CFEngine any:: %]
<VirtualHost $(sys.ipv4[eth0]):80>
     ServerAdmin           $(stage_file.params[apache_mail_address]
[1])
     DocumentRoot          /var/www/htdocs
     ServerName            $(stage_file.params[apache_server_name][1])
     AddHandler            cgi-script cgi
     ErrorLog              /var/log/httpd/error.log
     AddType               application/x-x509-ca-cert .crt
     AddType               application/x-pkcs7-crl    .crl
     SSLEngine             off
     CustomLog             /var/log/httpd/access.log
</VirtualHost>

[%CFEngine webservers_prod:: %]
[%CFEngine BEGIN %]
<VirtualHost $(sys.ipv4[$(bundle.interfaces)]):443>
     ServerAdmin           $(stage_file.params[apache_mail_address][1])
     DocumentRoot          /var/www/htdocs
     ServerName            $(stage_file.params[apache_server_name][1])
     AddHandler            cgi-script cgi
     ErrorLog              /var/log/httpd/error.log
     AddType               application/x-x509-ca-cert .crt
     AddType               application/x-pkcs7-crl     .crl
     SSLEngine             on
     SSLCertificateFile    $(stage_file.params[apache_ssl_crt][1])
     SSLCertificateKeyFile $(stage_file.params[apache_ssl_key][1])
     CustomLog             /var/log/httpd/access.log
</VirtualHost>
[%CFEngine END %]
```

2.3 Promise Types

We've already introduced several of the promise types supported by **cf-agent** configuration files (e.g., *promises.cf*). In this section, we will discuss the most widely used of these in some detail. Table 2.1 briefly describes the rule types that appear in the following subsections.

Type	Purpose
access	Grant access to remotely accessible files (server).
commands	Execute external shell commands.
classes	Define classes or contexts, e.g., web servers.
files	Verify/correct the attributes of files.
methods	Call another (parameterized) bundle as a subroutine.
packages	Verify the presence of/Install software packages.
processes	Monitor and manage processes.
services	Encapsulate a managed service.
storage	Check or mount storage devices.
vars	Define variables for use elsewhere.

Table 2.1: Some CFEngine Promise Types

Unfortunately, space limitations do not allow us to cover all of the available rule types here. Consult the CFEngine reference manual for complete information about all rules types and options.

2.4 Promise Bodies and the Standard Library

We'll begin by discussing some options that are available for many different promise types. To fully specify a promise we have to think about its body. The body of a promise is that part that contains the specification of how a promise will be kept (think "body" in the sense of "body of a contract" or "HTML body"). The body of a promise consists of a list of "constraints" that describe what limits will be placed on the affected object, and contraints have the form

```
body-subtype => value
```

where attribute is something like "perms" (for file permissions) or "signals" (for process termination), etc. A constraint assigns a value or range of allowed values for the body type, which becomes a part of the promise to keep. The right-hand side values can either be direct scalars or lists of data, or body-templates that are defined elsewhere. For example:

```
files:
   "/affected/file"
      create => "true",              scalar value
      perms => mo("644","root");     body-template called "mo"
```

The create constraint has a true/false value which can easily be entered in-line, but some constraints naturally consist of many detailed specifications that suggest hiding the details behind a simpler representation. The reusable body-template "mo" is such a case, as file permissions can have many complex attributes on different file systems. Thus we make a named object that will be declared outside of any promise bundle block as a separate syntax object. Here is a rather trivial example implementation

```
body perms mo(p,o)
{
mode => "$(p)";
owners => { "$(o)", "administrator", "wheel" };
}
```

In this simple case, using a template offers little extra benefit, except to reduce the overall amount of typing for command usage, but in cases like copy_from, the saving and contribution to clarity can be very large. More important than the saving of space is the clarity of intent that can be expressed by a judicious choice of words.

CFEngine provides a standard library of these body-templates in an effort to make code as readable and concise as possible, and without overloading the reader with details. If you don't like the standard library, however, you can make your own with the basic language. Being able to customize the language to local culture is part of the knowledge strategy for CFEngine. This includes whether or not parameters are exposed or hidden. So, for example, the case above could also be written:

```
files:
   "/affected/file"
      create => "true",              scalar value
      perms => system_settings;      body-template called "system_settings"
```

with definition:

```
body perms system_settings
{
mode => "644";
owners => { "root", "administrator", "wheel" };
}
```

The decision whether to use parameters, or not, is a didactic one. Do you want to see the data values in-line for clarity, or do you want to not see them for clarity? This is your choice. And if you don't like making these choices, simply use the standard library.

The standard library is included with the CFEngine distribution. To use it, you simply have to include it in your policy's inputs attribute, in the body common control structure. For example:

```
body common control
{
   inputs => { "cfengine_stdlib.cf" };
   bundlesequence => { ... };
}
```

2.4.1 Some Common Attributes

Most body-types are specific to a particular type of promise, because the attributes they describe only exist for those objects. Some body-types can be used in any promise, however, because they refer to constraints on the way in which promises in general are kept.

Action	Constraints taken to keep the promise
classes	Whether to set any class-context information about what happened
comment	A documentation of why we are making the promise
depends_on	A list of promise handles this promise depends on
handle	A short identifier by which to refer to the promise
ifvarclass	An optional class-context limiter
meta	Other user-defined metadata associated with the promise

Table 2.2

2.5 Creating Your Own Library

There are plenty of good reasons to use the CFEngine standard library: it offers a lingua franca for sharing simplified patterns with other users, and it helps you to communicate with other users in the same language. The downside of any standardization is that it becomes a straitjacket if it fails to match your needs or expectations. In that case, the CFEngine language was designed to make as few decisions for you as possible: you are free to make your own set of body-templates and/or bundles of promises.

Today, the standard library has become part of a larger body of standard solutions called the CFEngine Design Center, hosted on GitHub. There you will find a wealth of pre-defined data-driven methods and templates for immediate use, or for use as examples for local modification. We'll return to the Design Center in chapter 4.

2.6 Basic Policy Orchestration

The **bundlesequence** is just a master list; it does not have the sophistication of expression to truly organize the flow of your policy. It is more akin to a list of chapters in a book, or the movements in a symphony—it provides a convenient framework around the major parts of the story, but not all the details. To fully orchestrate your policy, we recommend using one or more of the bundles in your **bundlesequence** to promise *methods* and *services* abstractions that sketch out your desired infrastructure. This allows you the full power of the CFEngine promise syntax to orchestrate your checks and changes.

To illustrate this, consider the following partial example. Here we show how to set up a kind of "score" for your policy. In this more substantial excerpt of CFEngine code, you can see how the major flow structures can be organized. The code above is not complete, because it is missing a number of definitions that explain what "STIGs" means, for example. We can assume that such definitions exist and are defined elsewhere.

Let's comment on some of the features of this code as we go.

All code begins with the a control body of type common (meaning that it applies to all the agents). Like other body-templates, a control body template consists of attributes that explain the details being promised. Unlike user-defined templates, this refers to promises that are hardcoded into the CFEngine software itself.

```
body common control
{
bundlesequence => { "overture", "orchestrator", "finale" };
inputs         => { "cfengine_stdlib.cf" };
}
```

The inputs attribute tells the agent which additional files to import, as these will contain definitions to be used in making promises. The **bundlesequence** tells the agent the major order in which to execute top-level bundles.

In this case, we have invented three top-level bundles called "overture," "orchestrator," and "finale" to win over your imagination in thinking about the orchestration of policy.

```
bundle agent overture
{
methods:
    "name resolution"     usebundle => name_services;
    "security hardening"  usebundle => STIGs;
    "security hardening"  usebundle => PCI_DSS;
    "security hardening"  usebundle => tripwires;
    "security hardening"  usebundle => secure_applications;
 user_machines::
    "user management"     usebundle => regular_users;
    "user management"     usebundle => root_passwords;
 cloud_controllers::
    "private cloud"       usebundle => check_fixed_VMs;
}
```

In the first of these bundles, we use only "methods" promises. By using methods, like subroutines, we can defer the details of what to do with a layer of naming and abstraction that describes the major themes.

The definitions of these themes must be given as separate bundles elsewhere.

```
bundle agent orchestrator
{
services:
 backend_webservers::
   "www"            comment => "This is the standard web module";
   "web_main_site" comment => "This is the core business site config";
 dbservers::
   "mysql" comment => "Migrating to postgres on 2nd Feb";
   # "postgres" comment => "Postgres initiative, talk to Fred Smith";
 any::
   "ntp";
storage:
 user_machines::
   "/home"  mount => nfs("/export/home","storage_server.example.com");
 any::
   "/mnt/common"  mount => nfs("/export/common","storage_server.example.
com");
}
```

In the middle section, which we call "orchestrator," we use a different abstraction, namely one based in services, to start key services in our infrastructure. In fact, services are no different than methods promises, but the semantic distinction is helpful—services are something that everyone understands are relevant to business operations.

The storage promises also expose an aspect of system management that makes high level sense. In fact, we are free to get as high- or low-level as we like in these bundles. For the purposes of communication, however, we recommend a layer of high-level abstraction like this:

```
bundle agent finale
{
reports:
   www_in_anomaly::
      "Web traffic on $(sys.host) is abnormally high";
}
```

For the finale of our infrastructure symphony, we chose some simple reporting. Reports promises allow you to print messages, access variables, pipe data to log files, etc. The final section could be about summarizing some important details about what happened for all the promises. In general, we are not interested in hearing back from our hosts: if they are 10,000 in number, but if there are exceptional circumstances, we might be.

The example shown here uses a class-context **www_in_anomaly** which CFEngine defines automatically, if the level of incoming web traffic is more than two standard deviations above normal. You must be running the **cf-monitord** agent for this to be defined.

The purpose of this section was to show you that it can pay to not jump headfirst into low-level details. Instead, give some thought to how you want to communicate your infrastructure promises. The more understandable they are, the easier it will be to work across different silos of your organization, e.g., from business to IT, from development to operations.

2.7 Using Lists to Compress Policy

Separating promise patterns from data is a good way to minimize the amount of stuff you are writing down, and to make it easier for readers of your infrastructure design to see general patterns in what can easily become a chaos of details. For this purpose, list variables are your friends, as they make it very easy to apply a consistent standard or behaviour to a set of objects. For example, to install a list of packages, we only need a single promise unless there are specific version numbers to specify:

```
vars:

   "packages" slist => { "apache2", "php5" };                    Data

packages:

      "$(packages)"                                              Promise pattern
        package_policy => "add",
        package_method => apt;
```

Similarly, to start a number of virtual machines, all the same except for a name:

```
vars:

   "vm_list" slist => { "vm1", "vm2", "wm3", "vm4", "vm5 };      Data

guest_environments:

   host_system::
      "$(vm_list) on $(sys.host)"                                Promise pattern
        environment_resources => std_kvm("$(vm_list)"),
        environment_type      => "kvm",
        environment_state     => "create",
        environment_host      => "$(sys.host)";
```

2.8 An Access Control Paradigm for Policy

We can take this idea even further and say that most configuration issues can be divided into two cases, very much analogous to access control lists, where there are things we want (grant, add, install), things we don't want (deny, delete, remove) and everything else we don't care about (no micro-management of systems). Thus, many tasks can be framed as two lists: a whitelist and a blacklist.

For example, in an edit_line bundle to manage the Apache module list, we can make a list of what we do and don't want, and CFEngine will convergently maintain this state:

```
vars:
   "add_modules" slist => {
                          "dav",
                          "dav_fs",
                          "ssl",
                          "php5",
                          "dav_svn"
                          };
 "del_modules"   slist => {
                          "php3",
                          "jk",
                          "userdir",
                          "imagemap"
                          };
```

```
# Generic promises below here

field_edits:
   "APACHE_MODULES=.*"
      edit_column => quotedvar("$(add_modules)","append");
   "APACHE_MODULES=.*"
      edit_column => quotedvar("$(del_modules)","delete");
```

Alternatively, given the two lists above, defined externally, we could pass these to a generic method as arguments:

```
bundle agent xyz
{
vars:
 "add_modules" slist =>   {
                          "dav",
                          "dav_fs",
                          "ssl",
                          "php5",
                          "dav_svn"
                          };
 "del_modules" slist =>   {
                          "php3",
                          "jk",
                          "userdir",
                          "imagemap"
                          };

methods:
   "web servers"
      usebundle => fix_modules("@(xyz.add_modules)", "@(xyz.del_
modules)");
}

# Hide everything below this line in a library
bundle agent fix_modules(allow,deny)
{
files:
     "/etc/sysconfig/apache2"
     edit_line => apache_list("@(allow)", "@(deny)");
}
bundle edit_line apache_list(add,del)
{
field_edits:
     "APACHE_MODULES=.*"
       edit_column => quotedvar("$(add)","append");
     "APACHE_MODULES=.*"
       edit_column => quotedvar("$(del)","delete");
}
```

2.9 CFEngine Classes, Contexts and Decisions

Promises describe *what* objects will be configured and *how* they will be maintained. To describe *where* and *when* such promises apply, CFEngine uses the concept of contexts, which in turn are built from *classes*.

Historically, the term *class* was chosen for CFEngine's classification of environment properties before object orientation became very popular, so the term has become slightly overloaded with different meanings today. We have begun to refer to the them as *contexts* or class-contexts to avoid this confusion—but the name has stuck for most

CFEngine users. Class contexts refer to CFEngine's effort to discover or express facts and properties about host environments that classify them in different ways. We use *class expressions* to describe the *context* in which *promises* apply.

Classes or contexts are the discovered and cached results of tests that CFEngine makes about properties of the system. They are evaluated just before **an agent** starts executing. For example:

```
files:
   linux|solaris::
      "/etc/nsswitch.conf"
        comment => "Use a standard template for name services",
      edit_template => "/masterfiles/my_ns_template.in";
```

Context classifications can be:

- Detected at runtime about the host environment.
- Defined by you in a policy file, based on probes, tests or list data.
- Based on the results of special functions.
- Defined as a result of actions taken (or not taken) during promise-keeping.
- Based on measurements/observations taken by **cf-monitord**.
- Used in complex logical expressions to enable the conditional application of rules.

Classes of the first type (often called hard classes) include immutable characteristics of the operating system environment, the network environment, and the date and time of the **cf-agent** run. To see which classes are detected in your environment, run **cf-promises** with the **-v** option.

Here is an example of the output from one of our systems. We have reorganized and annotated the output for pedagogical purposes.

```
$ cf-promises -v
Defined Classes = (
snow snow_white_com white_com                          Hostname & domain variations
192_168_1_101 192_168_1 192_168 19                     IP address components
May Day26 Saturday Yr2012                              Date components
Hr12 Hr12_Q3 Min35 Min35_40 Q3                         Time of day components
linux linux_2_6_11_4_21_10_default                     OS and kernel version
SuSE SuSE_9 SuSE_9_3                                   Operating system specifics
64_bit i686                                            Hardware characteristics
fe80__20e_35ff_fe52_5b03 net_iface_eth0                MAC address & interface name
CFEngine_3 CFEngine_3_3                                CFEngine version variations
UserProcs_high_dev1 DiskFree_high_dev2                 System resource usage levels
…etc.
)
```

Classes, like other identifiers, can only consist of the characters a-z, A-Z, 0-9 or the underscore. When **cf-agent** converts data containing other characters such as dots or hyphens into classes, it converts all illegal characters to underscores. Hence fully qualified domain names such as *host.domain.tld*, when represented as classes, become **host_domain_tld**.

Most of these classes are self-explanatory. However, those relating to the time of day may be a bit opaque at first. First of all, these times always refer to when the current **cf-agent** run began, and not to the exact time when any specific rule is actually processed. The **Hr***nn* and **Min***nn* forms refer to specific hours of the day and minutes after the hour. The **Q***n* classes refer to the four 15-minute "quarters" of each hour: i.e., **Q3** refers

to the period from 30 to 44 minutes after the hour. Similarly, the form **Min***mm_nn* refers to the specified five-minute interval: e.g., **Min20_25** refers to the five minutes starting at twenty minutes past the hour.

Note that, because each agent detects its own private environment, the classes it experiences are local and are not seen by any other hosts on the network.

> ***Definition 10: Autonomy and locality.*** *By default, CFEngine does not give you an overview of the state of all the hosts running it. Each host is a closed and independent box. (CFEngine Enterprise edition includes a Mission Portal for this purpose.)*

If you are used to thinking in terms of centralized management, you might find this surprising or even a weakness, but how should CFEngine know the boundaries of your system? No one would want a system that automatically opened every host to knowledge about every other host. Since CFEngine allows every possible model from centralization to independence, it defaults to maximum privacy, and maximum scalability.

2.9.1 How Do I Define My Own Contexts?

CFEngine defines classes that cover generic aspects of systems. These are called hard classes because they are indisputable properties of the environment in which **cf-agent** is operating. In addition to these you might want to other define classes of your own (known as soft classes) based on abstract customizations of the local environment, such as group membership, geography or the existence of certain files or processes.

The **classes** section of *promises.cf* may be used to define classes. Here are some example class definitions:

```
classes:
     "WinXP"
                or => { "pc121", "pc122", "pc123_CFEngine_org" };
     "TheTouched"
   expression => fileexists("/usr/local/etc/touch_me");
```

The identifier name on the left-hand side becomes defined (logically true) if any of the classes on the right-hand side is defined. In the first case, the class name **WinXP** is a shorthand for the three specifically named hosts on the right-hand side. (CFEngine automatically defines classes based on the hostname, as well as on the FQDN of the machine in which it is running, so those classes would be defined if the current hostname is pc121, pc122, or if its FQDN is pc123.cfengine.org.)

In the second example, the class **TheTouched** becomes defined if the function evaluates to true: i.e., if the file */usr/local/etc/mark* exists. The fileexists() function is one of many predefined functions in the CFEngine policy language.

Once classes are defined, they can be used to label policy rules using the double colon notation:

```
files:
   TheTouched::
      "/usr/local/etc/mark"
        edit_line => append_if_no_line("Mark woz ere");.
```

2.9.2 Combining Classes

Classes are effectively Boolean variables—you make "if-then-else"-like decisions with them by making them label promises. For precise customization, you need to combine them like Boolean expressions. Classes are combined with the basic operators:

Logical Operation	Symbol	Alternative Symbol
NOT	**!**	
AND	**.** (dot)	**&**
OR	**|**	
Grouping	**( )**	

Table 2.3

Operator precedence is the usual one, as ordered in this table. We recommend using parentheses for grouping in complex expressions to avoid ambiguity as well as to improve configuration file readability.

Here are some examples of class expressions:

Class Expression	When True
`solaris.Monday.Hr01::`	*Solaris systems on Monday during the 1 AM hour.*
`aix|hpux::`	*AIX or HP-UX systems.*
`aix.!vader::`	*AIX systems other than system named vader*
`December.Day31.Friday::`	*New Year's Eve when a Friday*
`Day13.!Friday::`	*The 13th of the month when not a Friday*
`solaris|aix.Monday::`	*Solaris systems, or AIX systems on Monday*
`(solaris|aix).Monday::`	*Solaris or AIX systems, on Monday*
`something::`	*A class "something." has been defined*

Table 2.4

Classes remain in effect within a stanza until another class expression is encountered. However, they do not carry across stanza boundaries. Note that the **any** class may always be used to remove any class-based restrictions in effect. For example:

```
reports:
   solaris::
      "This message prints only on Solaris systems.";
   aix::
      "This message prints only on AIX systems.";
   any::
      "This message prints everywhere.";
```

2.9.3 Defining Classes with Functions

Classes can also be defined conditionally based on the return value of a variety of built-in functions or of an external command. Here is an example:

```
classes:
   "have_mark" expression => userexists("mark");
```

2.9.4 Outcome Classes Like Return Codes

CFEngine is not a script language with return codes from sub-shells, but you can still base certain promises on the outcome of others. Class contexts are also the generic

mechanism for doing this. After all, the success or failure of one part of an infrastructure just becomes part of the current context in which we operate. Outcome classes can be attached to any kind of promise through the *classes => body-template* syntax.

The CFEngine reference manual details powerful possibilities for working with outcomes, but a simplified interface is provided by the standard library, as illustrated in these examples:

```
files:
   "/tmp/important_file_1"
      create => "true",
      classes => if_ok("nothing_to_do");

   "/tmp/important_file_2"
      create => "true",
      classes => if_repaired("I_was_repaired");

   "/tmp/important_file_3"
      create => "true",
      classes => not_kept("not_repaired");

   "/tmp/important_file_4"
      create => "true",
      classes => if_else("nothing_to_do","not_repaired");

reports:
   nothing_to_do::
      "All quiet on the Western front";
   I_was_repaired::
      "Alert! System was repaired, and is now as desired!"
   not_repaired::
      "Wake the sysadmin, repairs failed!";
```

If you want to go beyond these simple cases, you are free to make your own body-templates to keep the syntax clear and uncluttered. You can look at the definitions of *if_ok(), if_repaired()* and *not_kept()* in the *cfengine_stdlib.cf* library, to understand how they work.

2.10 Policy Ordering and Execution

CFEngine can make huge savings on execution time by taking charge of the order in which certain operations take place, grouping together similar items and avoiding contention. The order in which declarations are made is not necessarily related to the order in which promises are kept. CFEngine determines its own ordering, which is called *normal ordering*. We refer you to the reference manual to learn more on this.

You can override the order of operations in CFEngine to suit special needs (using classes, or the depends_on constraint with promise handles), but you should try to avoid thinking sequentially like a flow-chart. CFEngine's strength lies in taking away the worry of thinking algorithmically—we should think documentation.

2.11 Knowledge and Your Declarative Policy

As infrastructure becomes more ubiquitous and more complex, understanding it is increasingly a challenge, especially for new or inexperienced system engineers. CFEngine 3 was designed, more than ever, to invest in *knowledge* about infrastructure, not merely to build it. Although we are revisiting a period of focus on "Build," once technologies

for virtualization mature, the knowledge deficit will be even greater than before unless we take the time to document structures properly.

Each promise in a CFEngine policy may be wrapped in a wealth of metadata as part of the general promise model. This not only improves the general overview of infrastructure, but also the contextual information provided by CFEngine at runtime.

In the Enterprise edition of CFEngine, knowledge instrumentation takes on a whole different level of significance, with additional features. Even at the community level, it is worth knowing how to use the metadata to give improved error reporting and diagnostics.

2.11.1 Syntax for Encoding Knowledge

```
bundle component name(parameters)                    A container with a name/theme
{
what_type:                                                          Promise type
   where_when::                                                     Classes/context
      # Traditional comment                          Throw-away comment for designers
      "what/affected object" -> { "promisee", "stakeholder" },
        comment => "The intention ...",                        Comment for observers
        handle => "unique_id_label",              A way for refer to this whole promise
      attribute_1 => body_or_value1,                            The body of the promise
      attribute_2 => body_or_value2;
}
```

How much instrumentation you add will depend on your local culture and the permanence of your infrastructure. There is a fine balance between "less is more" and "metadata are empowering." This can be organization-dependent.

Consider the following example of how a promise, instrumented with metadata, can lead to a an easier diagnostic experience for users. Let's add a promisee, a handle and a comment to a silly mistake:

```
commands:

   "exoijdsfkn" -> { "mark@cfengine", "officer plod" },

          handle => "mojito",
       comment => "I was drunk when I wrote this";
```

The command clearly does not exist and does nothing. The output from this piece of nonsense would be the following:

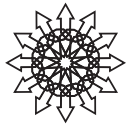
```
Proposed executable file "exoijdsfkn" doesn't exist
exoijdsfkn promises to be executable but isn't
I: Report relates to a promise with handle "mojito"
I: Made in version 'not specified' of "./test.cf" near line 12
I: The promise was made to (stakeholders): {'mark@cfengine','officer
plod'}
I: Comment: I was drunk when I wrote this
```

By adding metadata, we allow CFEngine to help supply meaningful error messages with real diagnostic potential. In the Enterprise edition of CFEngine, the possibilities go much further, enabling integration with a semantic web of system information.

2.11.2 Example of Knowledge Instrumentation

```
body common control
{
bundlesequence => { "overture", "orchestrator", "finale" };
inputs         => { "cfengine_stdlib.cf" };
}
#########################################################
bundle agent overture
{
methods:
     "name resolution"                              promise category wrt infrastructure
          handle => "name_resolution"                          handy identifier
        comment => "Ensure basic NIS/LDAP/DNS services are ok",    why
       usebundle => name_services;                                  what

 webservers_payment::
     "security hardening" -> {"mr.security@example.com",
                              "business@cfengine,com"} ,
          handle => "infosec_pci",
        comment => "Auditors need to see this in place",
       usebundle => PCI_DSS;

     "security hardening"  -> "mr.security@example.com",
          handle => "infosec_tripwire",
        comment => "Monitor unauthorized changes",
       usebundle => tripwires;

       "security hardening"  -> "mr.security@example.com",
          handle => "infosec_apps",
        comment => "Local application security documented here",
       usebundle => secure_applications;

 user_machines::
     "user management"
          handle => "users_regular",
        comment => "Configure login environment for non-priv users",
       usebundle => regular_users;

     "user management"
          handle => "users_root_passwd",
        comment => "Enforce root password policy and change",
       usebundle => roor_passwords;

 cloud_controllers::
     "private cloud"
     handle => "vms_permanent",
     comment => "Spawn and monitor permanent VMs",
     usebundle => check_fixed_VMs;
}
#########################################################
bundle agent orchestrator
{
services:
   webservers_backend::
     "www"
          handle => "www_basic",
        comment => "This is the standard web module";
     "web_main_site"
      depends_on => { "www_basic" },
        comment => "This is the core business site config";
```

```
   dbservers::
          "mysql" comment => "Migrating to postgres on 2nd Feb";
      # "postgres" comment => "Migrating to postgres on 2nd Feb";
      any::
         "ntp";
storage:
 user_machines::
      "/home"  mount => nfs("/export/home","storage_server.example.
com");
 any::
      "/mnt/common" mount => nfs("/export/common","storage_server.
example.com");
}
#############################################################
bundle agent finale
{

reports:
      www_in_anomaly::
        "Web traffic on $(sys.host) is abnormally high";
}
```



3. Services and Methods

Making your infrastructure description easy to understand is a major goal if you are going to succeed in dealing with complexity and scale. Abstraction is the basic tool for hiding complexity without avoiding it—to model the true needs of an environment, and not try to suppress them because we have failed to comprehend the problem.

CFEngine has been written by engineers for engineers; it takes a bottom-up point of view: giving powerful primitives that can be combined into an intuitive and appealing model of infrastructure.

3.1 The Method Abstraction

In section 2.6, we showed how methods can be used to divide up the orchestration of policy into a high level menu-like structure. Now let's look at what the details of a bundle might look like in one of these cases.

Principles for writing bundles:

- Each bundle should handle special cases internally.
- Each bundle should be self-healing.
- Data for the bundle may be either local (bespoke method) or passed as a parameter (reusable method).

Here is an example of what a web server module might look like for a PHP-enabled web server. This makes implicit use of the standard library body-templates, and it handles just two special cases: CentOS and Ubuntu. All of the data about package names on different operating systems are contained in the bundle so that this does not cloud the application of the method at the menu level.

```
bundle agent app_web_phpapache
{
vars:
   centos::   "php_pkgs" slist => { "httpd", "php" };
   ubuntu::   "php_pkgs" slist => { "apache2", "php5" };

packages:
   "$(php_pkgs)"
                  comment => "Install Apache webserver with PHP",
        package_policy => "add",
     package_method => generic,
              classes => if_ok("ensure_php_apache_running");

processes:
   centos.ensure_php_apache_running::
     ".*httpd.*"     restart_class => "start_httpd";
   ubuntu.ensure_php_apache_running::
     ".*apache2.*"   restart_class => "start_apache";
```

```
commands:
   start_httpd::        "/etc/init.d/httpd start";
   start_apache::    "/etc/init.d/apache2 start";
}
```

This bundle takes no parameters. It is merely a block of code, grouped for convenience, like a simple container. This should not be seen as a weakness: the aim, after all, is to have clear active documentation about intent, not to dazzle with programming acumen.

3.2 The Service Abstraction

Service orientation is a highly desirable and modern approach to organizing infrastructure. Almost anything can be turned into a service, if we want to think in that way—from web services to "compute" farms to storage arrays. In CFEngine 3 you are free to choose a method or a service abstraction for your major items; we only encourage you to make a clear interface to these for readability.

The **services** promise type has a special meaning on Windows systems, as Windows services are distinct from processes. On Unix, you can decide for yourself what you want the definition of a service to be: unless otherwise specified, CFEngine will look for a default bundle, e.g., in the standard library called

```
bundle agent standard_services(service,state)
```

This is assumed to contain promises that flesh out what starting and stopping services means. The defaults are hardwired to make it very easy to start services:

```
services:
   "www";
```

This would be enough to start a web service, assuming it had been defined in the **standard_services** bundle.

```
bundle agent orchestrator
{
vars:
   "mail" slist => { "milter", "spamassassin", "postfix" };
services:
   "www" service_policy => "start";
   "$(mail)" service_policy => "stop";
}
```

The services bundle is assumed to be in your standard library file, included as part of the common control body. It might look something like this:

```
bundle agent standard_services(service,state)
{
vars:
 suse|redhat::
   "startcommand[www]"     string => "/etc/init.d/apache2 stop";
   "stopcommand[www]"      string => "/etc/init.d/apache2 stop";
 debian|ubuntu::
   "startcommand[www]"     string => "/etc/init.d/httpd stop";
   "stopcommand[www]"      string => "/etc/init.d/httpd stop";
 linux::
   "startcommand[postfix]" string => "/etc/init.d/postfix stop";
   "stopcommand[postfix]"  string => "/etc/init.d/postfix stop";
```

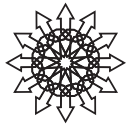
```
classes:
   "start" expression => strcmp("start","$(state)");
   "stop"  expression => strcmp("stop","$(state)");

processes:
   start::
      ".*$(service).*"
        comment => "Verify that the service appears in the process
table",
        restart_class => "restart_$(service)";
   stop::
      ".*$(service).*"
        comment => "Verify that the service does not appear in the process",
        process_stop => "$(stopcommand[$(service)])",
        signals => { "term", "kill"};

   commands:
      "$(startcommand[$(service)])"
           comment => "Execute command to restart the $(service)
service",
        ifvarclass => "restart_$(service)";
}
```

The services bundle does not have to be limited to process control. You could just as easily include the installation of the prerequisite packages in the same bundle. As you can see, there is great flexibility and power of control in the system through simple documentational interfaces.

3.3 Customizing Non-Standard Services

If you don't want to use a standard service description, of course, CFEngine allows you to customize the way you use the service abstraction.

```
bundle agent orchestrator
{
vars:
   "mail" slist => { "milter", "spamassassin", "postfix" };
services:
        "www" service_policy => "start",
              service_method => bespoke;

   "$(mail)" service_policy => "stop",
        service_method => bespoke;
}

#########################################################################

body service_method bespoke
{
service_bundle =>
     non_standard_services("$(this.promiser)","$(this.service_
policy)");
}

bundle agent non_standard_services(service,state)
{
reports:
   !done::
      "Test service promise for \"$(service)\" -> $(state)";
}
```



4. CFEngine Design Center

The Design Center is a public repository of reusable CFEngine configuration components, tools and examples, currently hosted on GitHub. It covers a broad range of different themes and contexts, and encourages a data-driven approach to infrastructure configuration, separating reusable knowledge from the specific parameters of usage.

The packaged methods in the Design Center are called infrastructure sketches, or more commonly, sketches. You will be able to use many of the sketches stored there "as is" to generate a good-enough level of automation for your specific needs. Other users will want to edit them as examples of good-practice and how others have solved similar problems, but customize them into locally perfected designs. Users are encouraged to contribute their code to the Design Center, so that it becomes a growing community resource.

The Design Center will play an increasingly large role in the use of CFEngine for most users. In most organizations, the task of designing the infrastructure and the task of implementing it will fall on different people. Implementers will not need to have the same knowledge as designers.

The Design Center and its tools are in the early stages of development—we are still researching the changes taking place in infrastructure engineering to make the best possible tools. Trends like DevOps play into the designs as a new generation of engineers grapple with new scale and complexity and a much more integrated IT infrastructure. This chapter offers a brief glimpse of what is going on, but some of the details are bound to change.

4.1 Getting Started with the Design Center

The Design Center contains three main types of content:

- *Sketches* are ready-to-use components that can be directly installed and used on a system. Sketches are managed using the **cf-sketch** utility.
- *Policy examples* are examples uploaded by CFEngine users. They are not meant to be ready to use but are simply to illustrate certain aspects of CFEngine policy writing, and to serve as starting points for your own policies.
- *Tools* that help in miscellaneous aspects of managing and interacting with CFEngine.

You can view these top-level categories of contents when you look at the repository on GitHub at https://github.com/cfengine/design-center:

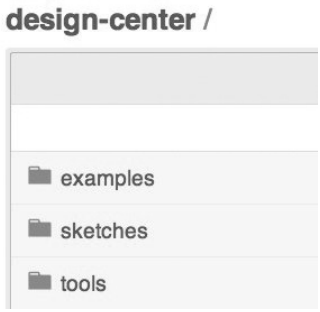


Figure 4.1

As a first step, you may want to check out the entire repository so that you can explore it at leisure. For this you need to use git:

```
$ git clone git://github.com/cfengine/design-center.git
$ cd design-center
$ ls -F
README.markdown  examples/        sketches/        tools/
```

Feel free to explore its contents. You can simply copy and use whatever you need from the examples and tools directories. For now we will focus on sketches, which are the most complex and useful part of its contents.

4.2 Getting Started with cf-sketch

cf-sketch is the main tool you will use for setting up sketches from the Design Center in your own systems. It allows you to search for, list, install, configure, activate and deactivate sketches. Here we will explore how to install and start using it. For a full reference manual, please refer to the documentation at https://github.com/cfengine/design-center/wiki.

Once you have checked out the Design Center repository as described above, you will find **cf-sketch** under the *tools/cf-sketch/* directory. Before using it, you need to make sure the following Perl modules are installed:

- LWP
- LWP::Protocol::https
- Term::ReadLine::Gnu

Depending on your system, these modules may already be installed, or may be available as packages in your operating system repository. Alternatively, you can use the cpan utility included with Perl to install them.

Once the dependencies are installed, you can install **cf-sketch** on your system, from the checked-out copy of the Design Center repository:

```
$ cd tools/cf-sketch
$ make install
```

You can now run **cf-sketch help** to see a summary of all the available options.

To better understand how **cf-sketch** works, it helps to understand the typical workflow for working with sketches:

1. Browse or search the Design Center repository for the sketch you need.
2. Install the sketch. This step downloads the sketch and installs the files locally, under /var/cfengine/masterfiles/sketches/ (this location may vary depending on your system and your user account).
3. Configure and activate a sketch. In this step you indicate the parameters to use with a sketch during its execution.
4. Generate a CFEngine "run file" that contains the actual CFEngine code needed to execute the activated sketches with the appropriate parameters.
5. Deactivate or uninstall sketches if necessary.

Let us now illustrate this workflow with an example. To start, run **cf-sketch**, which will put you in an interactive command mode:

```
# cf-sketch
Welcome to cf-sketch version 2.0.1.
CFEngine AS, 2012.

Enter any command to cf-sketch, use 'help' for help, or 'quit' or '^D'
to quit.

cf-sketch>
```

From this prompt you have access to all the commands needed to manage sketches. To view them, you can type the **help** command. Note that all the commands can also be run directly from the shell, without entering the interactive command mode, by typing the command you want to execute as arguments to the **cf-sketch** script.

First, we will find a sketch to install, by finding all sketches in the "utilities" category:

```
cf-sketch> search utilities

The following sketches match your query:

Monitoring::nagios_plugin_agent Run Nagios plugins and optionally take
action
Utilities::abortclasses Abort execution if a certain file exists, aka
'Cowboy mode'
Utilities::ipverify Execute a bundle if reachable ip has known MAC
address
Utilities::ping_report Report on pingability of hosts
VCS::vcs_mirror Check out and update a VCS repository.
```

Each sketch lists its name (e.g., VCS::vcs_mirror) and a short description of its functionality.

The VCS::vcs_mirror sketch allows us to use CFEngine to check out a copy of a git repository. We can get more detailed information about it using the **info** command.

```
cf-sketch> info VCS::vcs_mirror

The following sketches match your query:

Sketch VCS::vcs_mirror
Description: Check out and update a VCS repository.
Authors: Nick Anderson <nick@cmdln.org>, Ted Zlatanov <tzz@lifelogs.com>
Version: 1.11
```

```
License: MIT
Tags: cfdc
Installed: No
```

Once we know this is the sketch we want to install, we can install it.

```
cf-sketch> install VCS::vcs_mirror

Installing VCS::vcs_mirror
Unsatisfied dependencies: CFEngine::stdlib
Trying to find CFEngine::stdlib dependency
Found CFEngine::stdlib dependency, trying to install it
Installing CFEngine::stdlib
Checking and installing sketch files.
Done installing CFEngine::stdlib
Checking and installing sketch files.
Done installing VCS::vcs_mirror
```

Note that the CFEngine::stdlib sketch was installed automatically, as a dependency of VCS::vcs_mirror. The **info** command will now tell you where the sketch has been installed.

```
cf-sketch> info VCS::vcs_mirror

The following sketches match your query:

Sketch VCS::vcs_mirror
Description: Check out and update a VCS repository.
Authors: Nick Anderson <nick@cmdln.org>, Ted Zlatanov <tzz@lifelogs.
com>
Version: 1.11
License: MIT
Tags: cfdc
Installed: Yes, under /var/cfengine/masterfiles/sketches/VCS/vcs_
mirror
Activated: No
```

On a CFEngine policy hub, the sketches are installed under */var/cfengine/masterfiles/sketches* (substitute with */var/cfengine/inputs/sketches* if running on a CFEngine client, or with *$HOME/.cfagent/inputs/sketches* if running as a regular user instead of **root**). You can use the **list** command to show the installed sketches.

```
cf-sketch> list

The following sketches are installed:

1. CFEngine::stdlib (library)
2. VCS::vcs_mirror (not configured)
```

Note that the output says "not configured," which means that the sketch is installed, but it's not doing anything. We need to configure and activate it by providing the parameters the sketch needs for its execution. You can view these parameters and their types using the **info** command with the **–v** option.

```
cf-sketch> info -v VCS::vcs_mirror

The following sketches match your query:

Sketch VCS::vcs_mirror
Description: Check out and update a VCS repository.
Authors: Nick Anderson <nick@cmdln.org>, Ted Zlatanov <tzz@lifelogs.com>
Version: 1.11
```

```
License: MIT
Tags: cfdc
Installed: Yes, under /var/cfengine/masterfiles/sketches/VCS/vcs_mirror
Activated: No
Parameters:
    vcs: PATH
    path: PATH
    origin: HTTP_URL|PATH
    branch: NON_EMPTY_STRING (default: master)
    runas: NON_EMPTY_STRING (default: getenv("USER", "128"))
    umask: OCTAL (default: 022)
    activated: CONTEXT (default: any)
    nowipe: CONTEXT (default: !any)
```

You use the **configure** command to enter the interactive configuration mode, in which **cf-sketch** will prompt you for all the parameters necessary for the sketch. For our example, let's configure the sketch to check out a copy of the CFEngine core repository under /tmp/cfengine-core.

```
cf-sketch> configure VCS::vcs_mirror

Entering interactive configuration for sketch VCS::vcs_mirror.
Please enter the requested parameters (enter STOP to abort):

Parameter 'vcs' must be a PATH.
Please enter vcs: /usr/bin/git

Parameter 'path' must be a PATH.
Please enter path: /tmp/cfengine-core

Parameter 'origin' must be a HTTP_URL|PATH.
Please enter origin: https://github.com/cfengine/core.git

Parameter 'branch' must be a NON_EMPTY_STRING.
Please enter branch [master]: master

Parameter 'runas' must be a NON_EMPTY_STRING.
Please enter runas [getenv("USER", "128")]: root

Parameter 'umask' must be a OCTAL.
Please enter umask [022]: 022

Parameter 'activated' must be a CONTEXT.
Please enter activated [any]: any

Parameter 'nowipe' must be a CONTEXT.
Please enter nowipe [!any]: !any
```

Note that you can also provide a second argument to the **configure** command to indicate a JSON file from where the configuration parameters should be loaded instead of prompting for them. Most sketches include a params/ directory with some sample parameter files.

```
# cat /var/cfengine/masterfiles/sketches/VCS/vcs_mirror/params/
cfengine-core.json
{
 "activated": true,
 "path": "/tmp/cfengine-core",
 "origin":"https://github.com/cfengine/core.git",
 "branch": "master",
 "vcs": "/usr/bin/git"
}
```

We can verify that the sketch has been activated using the **list** command, with the **–v** option to show the parameter values:

```
cf-sketch> list -v

The following sketches are installed:

1. CFEngine::stdlib (library)
2. VCS::vcs_mirror (configured)
   Instance #1: (Activated on 'any')
     branch: master
     nowipe: !any
     origin: https://github.com/cfengine/core.git
     path: /tmp/cfengine-core
     runas: root
     umask: 022
     vcs: /usr/bin/git
```

A single sketch may be configured multiple times with different sets of parameters. The "activated" parameter must be a CFEngine class expression that determines where that configuration instance will be executed, so you can have the same sketch running with different parameters on different machines, differentiated by their activation condition.

We can now run the sketch by using the **run** command.

```
cf-sketch> run

Generated standalone run file /var/cfengine/masterfiles/standalone-cf-
sketch-runfile.cf

Now executing the runfile with: /var/cfengine/bin/cf-agent  -f /var/
cfengine/masterfiles/standalone-cf-sketch-runfile.cf

File /tmp/cfengine-core/.git/config was marked for editing but could not
be opened
I: Made in version 'not specified' of '/var/cfengine/masterfiles/
sketches/VCS/vcs_mirror/main.cf' near line 145
I: Comment: Expand Git config file from variable

Q: "...sr/bin/git clon": Cloning into '/tmp/cfengine-core'...
I: Last 1 quoted lines were generated by promiser "/usr/bin/git clone
-b master https://github.com/cfengine/core.git /tmp/cfengine-core"
```

This will write to a run file the CFEngine policy code necessary to execute all the active sketches with the appropriate parameters, and automatically execute it.

The run file generated by the **run** command contains a body common control declaration, to make it possible to execute it as a standalone policy. If you want to integrate the generated run file into your existing policy files (for example, by loading and running it from your promises.cf file), you can use the **deploy** command to generate the run file without the standalone bits.

```
cf-sketch> deploy

Generated non-standalone run file /var/cfengine/masterfiles/cf-sketch-
runfile.cf
This run file will be automatically executed from promises.cf
```

If you want to deactivate a sketch, you can use the **remove** command to remove a particular configuration instance or the entire sketch from your system.

```
cf-sketch> list -v

The following sketches are installed:

1. CFEngine::stdlib (library)
2. VCS::vcs_mirror (configured)
   Instance #1: (Activated on 'any')
     branch: master
     nowipe: !any
     origin: https://github.com/cfengine/core.git
     path: /tmp/cfengine-core
     runas: root
     umask: 022
     vcs: /usr/bin/git

cf-sketch> remove config VCS::vcs_mirror#1

Deactivated: VCS::vcs_mirror activation #1

cf-sketch> list -v

The following sketches are installed:

1. CFEngine::stdlib (library)
2. VCS::vcs_mirror (not configured)

cf-sketch> remove sketch VCS::vcs_mirror

Deactivated: all VCS::vcs_mirror activations
Successfully removed VCS::vcs_mirror from /var/cfengine/masterfiles/
sketches/VCS/vcs_mirror

cf-sketch> list

The following sketches are installed:

1. CFEngine::stdlib (library)
```

We encourage you to explore the documentation at https://github.com/cfengine/design-center/wiki/ for complete details and reference for using **cf-sketch**, and also for information about writing and contributing new sketches to the Design Center.

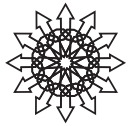
4.3 The Role of the Design Center in Your IT Infrastructure

The primary goal of the Design Center sketches is to make it very easy to reuse code written by others, and to promote sharing of CFEngine code within its community. However, another very important goal of the Design Center is to make it extremely easy for different sets of people to write and use the sketches. A sysadmin need not know the CFEngine language in detail (or at all) to be able to make use of Design Center sketches, as shown in the previous section. This makes it easy for sysadmins in your organization to use CFEngine effectively to manage IT infrastructure, while reusing the CFEngine knowledge from the community and from their colleagues. The **cf-sketch** tool allows you to easily draw sketches from multiple repositories, making it possible to have internal repositories in addition to the public one.

4.4 The Future of the Design Center

As of this writing, the main mechanism for accessing the Design Center is through the **cf-sketch** utility, which means a certain familiarity with the command line and with CFEngine is still needed. However, this is only the beginning! There is active work as of this moment to make it much easier to both develop and use Design Center sketches. These are some of the areas being explored, and which may be available even as you read this:

- Better sketch development tools, to aid experienced CFEngine users to create and share new sketches, and to convert their existing code to the appropriate sketch format.
- Design Center integration into the CFEngine Mission Portal interface, so that sketches can be managed directly and in conjunction with the other management and monitoring features provided by the Mission Portal.
- Alternative or improved front-ends to the Design Center, both command-line and graphical interfaces.
- Integration of Design Center into other tools that interact with CFEngine, such as the Vagrant CFEngine provisioner.



5. Building a CFEngine Infrastructure

In this chapter, we provide a quick roadmap for thinking about distributed management using CFEngine. If you are new to infrastructure automation, you will almost certainly be thinking about some form of centralized management. This is where most people start.

As IT systems and organizational complexity grow, centralization does not scale gracefully, either from a technical or a human perspective. At this point some form of federated management has to take over to cope with the challenge.

5.1 Roadmap for Centralized Policy

We shall assume that you have a central location for the definition of your policy. This is by no means necessary. Federated control is by far a preferable strategy in a large organization. If you don't, then you can repeat this procedure multiple times for each decentralized point of control.

There are several steps to be performed. The following order is recommended:

1. Set up policy source host first. We call this a policy server, and in the Enterprise edition this is also a reporting "hub" where status reports and inventory information are collated. A policy server is the computer that will store the master policy files that are used on every host running CFEngine. The default settings assume a policy dispatch point of */var/cfengine/masterfiles* on the policy server. In other words, this is where you drop approved changes to policy that will be collected.
2. Set up clients to install and update themselves.

This is a standard part of the default policy that comes with CFEngine 3, so unless you have any special needs, this should take care of itself.

Deciding policy is clearly a big task, but you can start simply, by getting a small policy running across your network and then build on this foundation. To get started, you need only to have a working prototype.

5.2 Federation of Control

The traditional view of network management is to apply a control over a network from some centralized, authoritative location: a master host. You can easily create this kind of architecture using CFEngine, but you are not limited by it. CFEngine's principle of autonomy makes it plausible to divide authority into regions, or have every host managed individually if that suits your needs. There is no compulsion to have centralized management, but it is easily implemented if that's what makes sense for you.

What are the advantages of centralized management?

- Having a single point of decision aids consistency.
- Changes of policy are easiest to implement from one place.
- Backup and version control of policy is convenient when policy is centralized.

What are the disadvantages?

- Central services are somewhat old-fashioned, making one think of marching armies rather than free market business.
- Local customization becomes awkward by forcing local knowledge to pass upward through a central authority.
- Centralization is inappropriate for security and privacy if you have completely independent departments or businesses that merely coexist and work together.

You can probably think of other reasons, and indeed you should think about this carefully. The key to sound management is in calculating the correct force to apply. Too much and you will bludgeon your departments into inappropriate conformity, but too little and they might run away to a place that you no longer understand. CFEngine's view is one of voluntary cooperation and hence voluntary consensus.

Federation of control is not only desirable but might actually be regulated in some industries. For example, in financial sectors, trading and banking are strictly separated by federal regulation, even if both activities are maintained in the same company. In this case, access to the different policies and infrastructure designs must be kept separate too.

This separation can be handled by a kind of staging (see section 5.3), but ideally, by complete physical separation.

5.2.1 Starbursts and Constellations

The most common architecture for management is the star network, i.e., a single central server (manager) surrounded by a number of clients (see figure below).

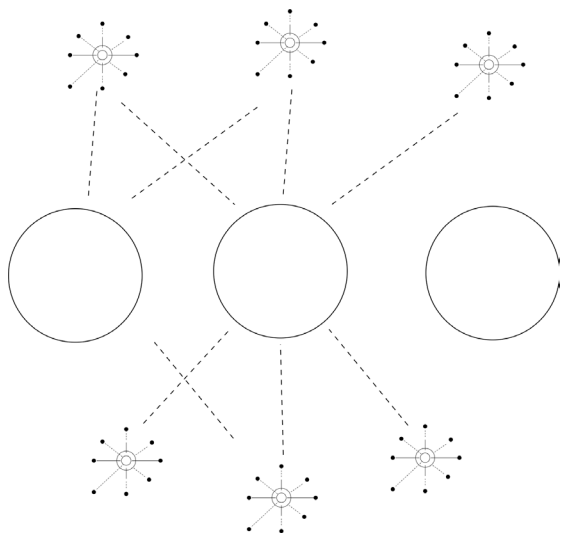


Figure 5.1

To federate this structure we can simply replicate it into a constellation of star networks. Using CFEngine, you can couple these weakly or keep them separate. CFEngine Enterprise allows you to go even further and obtain restricted reports at a very high level so that a few privileged monitors (without the power to change anything) could overlook the various parts of the network.

The large circles in this picture represent "observatories" from which privileged observers could, in principle, gain read-only access to a federated infrastructure. Note, once again, a key principle of autonomy in CFEngine forbids any external source from having authority over an agent. The way to view the constellation scenario is to imagine a collection of independently governed worlds, watched over by curious auditors. The power to change and grant access comes from below, not from above.

5.3 Staging Environments

In agile environments, it is common to have versioned stages of infrastructure, based on different policies. The different versions are run, either in separate environments, or mingled together in production by picking candidates to upgrade (so-called A-B testing).

We can create multiple environments with partitioned policies, just by keeping them in separate directories under */var/cfengine/inputs.* This can be achieved in a number of different ways.

The simplest way is to arrange for different hosts to collect their policies from different source files that are kept completely separate.

A variation on this is to use a class-determined variable to select the input specification like a switch:

```
body common control
{
 bundlesequence => { "environments" };
         inputs => { "environment_$(environments.active)/promises.cf" };
}
```

Here, the variable **$(environments.active)** can be defined in a common bundle called environments that uses class context to discover a host's membership in the different stages; CFEngine then inputs the relevant *promises.cf* file.

```
bundle common environments
{
 classes:

      "environment_development" or => {
                                       "hostname1_example_com",
                                       "ipv4_256_256_256_256",
                                      };

      "environment_testing"     or => {
                                       "hostname2_example_com",
                                       "ipv4_256_256_257",  # subnet
                                      };

      "environment_production"  or => {
                                       "hostname3_example_com",
                                       "ipv4_256_256_258",  # subnet
                                      };
```

```
  vars:
    environment_development::
       "active" string => "development";
    environment_testing::
       "active" string => "testing";
    environment_production::
       "active" string => "production";
}
```

A final approach might be to define the execution of **cf-execd** differently for different variables, once again effectively reading in a different policy for each class of host.

```
body executor control
{
exec_command => "$(sys.workdir)/bin/cf-agent -f
$(environments.active)_failsafe.cf && $(sys.workdir)/bin/cf-agent";
}
```

5.4 Test Environments Using Vagrant

As you are developing CFEngine policy, it is common to need to test it before full deployment. Apart from designating testing and staging environments as described in the previous section, you can do local testing using Vagrant (http://vagrantup.com/), a tool that allows repeatable, consistent creation and management of virtual machines. As of Vagrant 1.1, support for CFEngine is included. For Vagrant 1.0, you can download the CFEngine provisioner from https://github.com/cfengine/vagrant-cfengine-provisioner, which is the version we will use in this description.

The centerpiece of a Vagrant-managed environment is the Vagrantfile, which contains the instructions for creating and configuring VMs. A full description of the Vagrantfile syntax is outside the scope of this document, but you can find it at http://vagrantup.com/.

Using CFEngine on a Vagrant VM is as easy as indicating in the Vagrantfile that we want to provision the VM using the CFEngine provisioner, by adding this line inside the corresponding **config.vm.define** block:

```
config.vm.define :cfhub do |hub_config|
   ...
   hub_config.vm.provision CFEngineProvisioner
   ...
end
```

By adding this line, the default behaviour of the CFEngine provisioner is to ensure CFEngine is installed, and then configure CFEngine as a policy hub, bootstrapping to the VM's own IP address. When you run the **vagrant up** command, you will see the messages corresponding to the installation (if needed) and configuration of CFEngine on the VM:

```
$ vagrant up
...
[cfhub] Running provisioner: CFEngineProvisioner...
...
```

```
[cfhub] I am a CFEngine policy hub, bootstrapping to policy server at
10.0.2.15.
** CFEngine BOOTSTRAP probe initiated
...
[cfhub] CFEngine policy hub bootstrapped successfully.
[cfhub] Because I am a hub, I'm running cf-agent manually for the
first time to finish initialization.
```

Once the VM is running, you can log into it and verify that CFEngine is running:

```
$ vagrant ssh
...
$ ps ax | grep cf
 1198 ?        Ss     0:00 /var/cfengine/bin/cf-execd
 1201 ?        Ss     0:00 /var/cfengine/bin/cf-serverd
 1214 ?        Ss     0:00 /var/cfengine/bin/cf-monitord
$ ls /var/cfengine/
bin                     cf-serverd.pid     performance.tcdb.lock
cf3.lucid32.runlog      document_root.dat  policy_server.dat
cfagent.lucid32.log     inputs             ppkeys
cf_classes.tcdb         lastseen           promise_summary.log
cf_classes.tcdb.lock    lib                randseed
cf-execd.pid            masterfiles        reports
cf_lastseen.tcdb        modules            share
cf_lastseen.tcdb.lock   outputs            state
cf-monitord.pid         performance.tcdb
```

Of course, this is just the basics. Using the CFEngine provisioner, you can install custom policy files into the VM, instantiate multiple VMs in hub/client combinations, and even install and use the CFEngine Enterprise edition. For example, to create a hub/client pair, and make sure certain local files (contained in the **cf_files** directory) are copied to the hub for distribution to all the clients, you can use something like this:

```
config.vm.define :cfhub do |hub_config|
  hub_config.vm.box = "lucid32"
  hub_config.vm.network :hostonly, "10.1.1.10"

  hub_config.vm.provision CFEngineProvisioner do |cf3|
    cf3.policy_server = "10.1.1.10"
    cf3.files_path = 'cf_files'
  end
end

config.vm.define :cfclient do |hub_config|
  hub_config.vm.box = "lucid32"
  hub_config.vm.network :hostonly, "10.1.1.11"

  hub_config.vm.provision CFEngineProvisioner do |cf3|
    cf3.policy_server = "10.1.1.10"
  end
end
```

You can find complete Vagrantfile examples for different setups in the repository at https://github.com/cfengine/vagrant-cfengine-provisioner.

5.5 Using the cf-runagent Command

The **cf-runagent** command connects to a remote **cf-serverd** with a simple signal asking the server to initiate an immediate execution of **cf-agent** to verify its current promises. It is not permitted to send new policy to the remote host, for security reasons; however, the remote host might opt (as part of its ordinary policy) to update its policy from a command location. This command is useful for testing and for running **cf-agent** without having to log onto the host.

With the help of **roles** promises in **cf-serverd**, it is possible to make a Clark-Wilson-like model for Role Based Access Control of simple commands through this interface, trusting the public key identity of users rather than a login shell.

Since **cf-runagent** addresses remote hosts from a local host, there is an ambiguity in whether options are intended for the **cf-runagent** command itself, or whether they are meant to be passed on to the agent on the remote hosts. It is not possible to send the **-f** option to the remote agent, to ask it to run a different policy file. This option is stripped by the server on receipt to prevent an unauthorized attempt to change policy.

Remote classes are processed by the remote **cf-serverd** service, and specify classes which must be satisfied by the remote host in order to invoke the remote command.

Here are some examples, all of which use the host list in *cf-runagent.hosts*:

```
--background,      -b value - Parallelize connections (50 by default).
--define-class,    -D value - Define a list of comma-separated classes to be sent to a remote agent.
--select-class,    -s value - Define a list of classes to be used to select remote agents by constraint.
--remote-options, -o value - Pass options to a remote server process.
--diagnostic,    -x       - Activate internal diagnostics (developers only).
--hail,    -H value - Hail the following comma-lists of hosts, overriding default list.
```

For example, to run **cf-agent** on all Solaris hosts in the background:

```
cf-runagent --background --select-class solaris
```

To connect to a single host and execute promise checks in verbose, dry-run mode:

```
cf-runagent -hail myhost.example.com -remote-options "-dry-run --verbose"
```

5.6 Dealing with Firewalls

Some users want to use CFEngine's remote copying mechanism through a firewall, in particular to update the CFEngine policy on hosts inside a DMZ (so-called demilitarized zone). Firewalls are often shrouded in myth and mystery: magical force fields that protect us against Klingon torpedoes. It is important to see the firewall security model together with the CFEngine security model. Amongst the difficulties one faces, the firewall administrator is not often the same as the CFEngine administrator and does not trust anyone or anything. You might have to convince this person to make changes that help you out, so it is important to understand the consequences of your security strategy.

Any piece of software that traverses a firewall can, in principle, weaken the security of the barrier. On the other hand, a strong piece of software might have better security than the firewall itself. Consider the example in Figure 5.1.

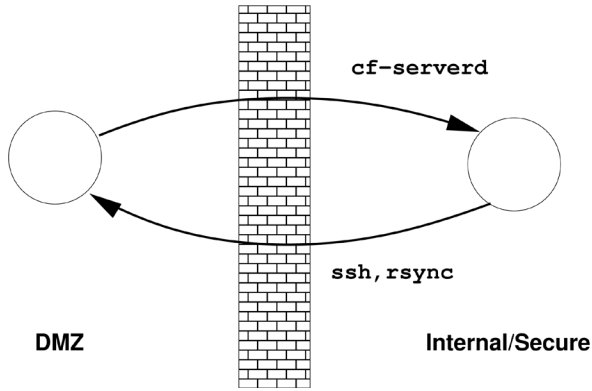


Figure 5.2: A CFEngine Host Outside a Firewall

We label the regions inside and outside of the firewall as the "secure area" and "demilitarized zone" for convenience. It should be understood that the area inside a firewall is not necessarily secure in any sense of the word unless the firewall configuration is understood together with all other security measures.

Our problem is to copy files from the "secure" source machine to hosts in the DMZ in order to send them their configuration policy updates. There are two ways of getting files through the firewall:

- An automated CFEngine solution, i.e., a pull operation generated outside with a target inside the secure area.
- A manual push to the outside of the wall from the inside.

One of the main aims of a firewall is to prevent hosts outside the secure area from opening connections to hosts in the secure area. If we want **cf-agent** processes on the outside of the firewall to receive updated policies from the inside of the firewall, information has to traverse the firewall.

> ***Definition 11: Conflicting trust models.*** *CFEngine's trust model is fundamentally at odds with the external firewall concept. CFEngine says: "I am my own boss. I don't trust anyone to push me data." The firewall says: "I only trust things that are behind me." The firewall thinks it is being secure if it pushes data from behind itself to the DMZ. CFEngine thinks it is being secure if it makes the decision to pull the data autonomously, without any orders from some potentially unknown machine. One of these mechanisms has to give if firewalls are to co-exist with CFEngine.*

From the firewall's viewpoint, push and pull are different: a push requires only an outgoing connection from a trusted source to an untrusted destination; a pull necessarily requires an untrusted connection being opened to a trusted server within the secure area. For some firewall administrators, the latter is simply unacceptable (because they are conditioned to trust their firewall). But it is important to evaluate the actual risk. We have a few observations on this score to offer at this point:

- It is not the aim of this note to advocate any one method of update. You must decide for yourself. The aim here is only to evaluate the security implications. Exporting data from the secure area to the DMZ automatically downgrades the privacy of the information.

- The CFEngine security model assumes that the security of every host will be taken seriously. A firewall should never be used as a substitute for host security.
- Knowing about CFEngine but not your firewall or your secure network, it is only possible to say here that it seems, to us, safe to open a hole in a firewall to download data from a host of our choice, but we would not accept data from just any host on your company network on trust. It would be ludicrous to suggest that an arbitrary employee's machine is more secure than an inaccessible host in the DMZ.

5.6.1 Option: A Policy Mirror in the DMZ

You can compromise by creating a policy mirror in the DMZ. This is the recommended way to copy files, so that normal CFEngine pull methods can then be used by all other hosts in the DMZ, using the mirror as their source. The policy mirror host should be as secure as possible, with preferably few or no other services running that might allow an attacker to compromise it. In this configuration, you are using the mirror host as an envoi of the secure region in the DMZ.

Any method of pushing a new version of policy can be chosen in principle: CVS, FTP, RSYNC, SCP. The security disadvantage of the push method is that it opens a port on the policy mirror, and therefore the same vulnerability is now present on the mirror, except that now you have to trust the security of another piece of software too. Since this is not a CFEngine port, no guarantees can be made about what access attackers will get to the mirror host.

5.6.2 Option: Pulling through a Wormhole

Suppose you are allowed to open a hole in your firewall to a single policy host on the inside. To distribute files to hosts that are outside the firewall it is only necessary to open a single tunnel through the firewall from the policy mirror to the CFEngine service port on the source machine. Connections from any other host will still be denied by the firewall. This minimizes the risk of any problems caused by attackers.

To open a tunnel through the firewall, you need to alter the filter rules. A firewall blocks access at the network level. Configuring the opening of a single port is straightforward. We present some sample rules below, but make sure you seek the guidance of an expert if necessary.

Cisco IOS rules look like this:

```
ip access-group 100 in
access-list     100 permit tcp mirror host source eq 5308
access-list     100 deny   ip  any any
```

Linux **iptables** rules might look something like this:

```
iptables -N newchain
iptables -A newchain -p tcp -s mirror-ip 5308 -j ACCEPT
iptables -A newchain -j DENY
```

Once a new copy of the policy is downloaded by CFEngine to the policy mirror, other clients in the DMZ can download that copy from the mirror. The security of other hosts in the DMZ is dependent on the security of the policy mirror.

5.6.3 Frequently Asked Questions about the Pull Method

- *Doesn't opening a port on a machine on the inside of the firewall make it vulnerable to both denial-of-service and buffer overflow attacks?*
  Buffer overflow attacks are extremely unlikely in CFEngine by design. The likelihood of a bug in CFEngine should be compared to the likelihood of a bug existing in the firewall itself.

  Denial-of-service attacks can be mitigated by careful configuration (see separate FAQ item). **cf-serverd** reads a fixed number of bytes from the input stream before deciding whether to drop a connection from a remote host, so it is not possible to buffer overflow attack before rejection of an invalid host IP.

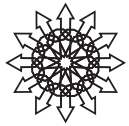
  Another possibility is to use a standard VPN to the inside of the firewall. That way one is concerned first and foremost with the vulnerabilities of the VPN software.

- *Doesn't opening the firewall compromise the integrity of the policy information by allowing an attacker the chance to alter it?*
  The CFEngine security model, as well as the design of the server, disallows the uploading of information. No message sent over the CFEngine channel can alter data on the server. (This assumes that buffer overflows are impossible.)

- *Couldn't an IP spoofer manage to gain access to data from the policy server that it should not be able to access?*
  Assuming that buffer overflow attacks and DOS attacks are highly improbable, the main worry with opening a port is that intruders will be able to gain access to unauthorized data. If the firewall is configured to open only connections from the policy mirror, then an attacker must spoof the IP of the policy attacker. This requires access to another host in the DMZ and is non-trivial. However, if the attacker succeeds, the worst he/she can do is to download information that is available to the policy mirror. But that information is already available in the DMZ since the data have been exported as part of the policy, thus there is no breach of security. (Security must be understood to be a breach of the terms of predefined policy.)

- *What happens if the policy mirror is invaded by an attacker?*
  If an attacker gains root access to the mirror, he/she will be able to affect the policy distributed to any host in the DMZ. The policy mirror has no access to alter any information on the policy source host. Note that this is consistent with the firewall security model of trusted/untrusted regions. The firewall does not mitigate the responsibility of securing every host in a network regardless of which side of the firewall it is connected.

Unfortunately, these decisions are often made as a matter of principle rather than considered judgment.



6. From Simple to Advanced

In this chapter, we present a brief gallery of example configurations for a few different scenarios. We begin with a simple case, and then move on to harder cases. What you'll see is that, unlike in an imperative language, the complexity of the policy is unrelated to the complexity or power of the behaviour.

6.1 Name Service Configuration

As a simple example, let's look at setting up the system resolver. This is a text editing problem. It could be handled by templating, but many sites have complicated requirements for host resolution that require special options and selection of nameserver based on network location and other criteria.

```
bundle agent name_services

{
vars:

   # data, can be kept here locally in this bundle

 "searchlist"  slist => {
                        "search example.com",
                        "search cfengine.com"
                        };

 !am_name_server::

    "nameservers" slist => {
                           "10.0.0.35",
                           "192.168.2.16",
                           "192.168.3.103"
                           0  },
                 policy => "overridable";

  am_name_server::

    # This will make the policy automatically adapt

    "nameservers" slist => {
                           "127.0.0.1",
                           },
                 policy => "overridable";

classes:

   "am_name_server"

      # This is basically a grep to know if we are a nameserver
      expression => reglist("@(nameservers)","$(sys.ipv4)");
```

```
files:

  # This is where the actual editing is done, using stdlib

  "/tmp/resolv.conf" # "$(sys.resolv)"
            comment => "Customize the resolver config to point to an
efficient nameserver",
             create => "true",
          edit_line => resolvconf("@(this.searchlist)","@(this.
nameservers)");

}
```

The output for a host that is not in the list of nameservers is thus:

```
search search iu.hio.no
search search cfengine.com
nameserver 10.0.0.351
nameserver 192.168.2.16
nameserver 192.168.3.103
```

The output for a host in the list of nameservers is thus:

```
search search iu.hio.no
search search cfengine.com
nameserver 127.0.0.1
```

If we examine the definition of the resolvconf bundle in the standard library:

```
bundle edit_line resolvconf(search,list)

 # search is the search domains with space
 # list is an slist of nameserver addresses

{
delete_lines:

  "search.*"     comment => "Reset search lines from resolver";
  "nameserver.*" comment => "Reset nameservers in resolver";

insert_lines:

  "search $(search)"    comment => "Add search domains to resolver";
  "nameserver $(list)"  comment => "Add name servers to resolver";
}
```

we see that **resolvconf** does not remove lines that a user has added by hand, unless they are nameservers or searchlist directives. This shows how we can manage "just enough" without being too heavy-handed. A template solution here would overwrite any local settings, which might not be desirable.

6.2 Phased Deployment—Inter-host Orchestration

As a more complex example of CFEngine, the following shows how the distributed system allows very powerful workflows to be orchestrated, without much more complexity than the simple cases. Orchestrating ordered sequences of bulk change is a major headache in many systems, and one often has to hack around an imperative approach.

We can make wrappers to handle the distributed ordering of workflows using CFEngine's client-server capabilities. This can be done peer-to-peer for simple cases, and with some Enterprise hub aggregation, we can also define multi-host waves of tasks.

6.2.1 Basic Communication Methods for Orchestration

The example below illustrates the basic syntax constructions for communication using systems. We can pass class data and variable data between systems in a peer-to-peer fashion, or through an Enterprise hub. You can run this with a server and an agent just on localhost to illustrate the principles.

In this example, we pass variable data between hosts. The generic peer function remotescalar can address any other host running cf-serverd. The abbreviated interface hubknowledge assumes that it should get data from a hub.

Both these functions ask for an identifier; it is up to the server to interpret what this means and to return a value of its choosing. If the identifier matches a persistent scalar variable (such as is used to count distributed processes in CFEngine Enterprise) then this will be returned preferentially. If no such variable is found, then the server will look for a literal string in a server bundle with a handle that matches the requested object.

```
body common control
{
bundlesequence => { "overture" };
inputs => { "cfengine_stdlib.cf" };
}

body server control

{
allowconnects          => { "127.0.0.1" , "::1",};
allowallconnects       => { "127.0.0.1" , "::1", };
trustkeysfrom          => { "127.0.0.1" , "::1",};
}

########################################################

bundle agent overture
{
vars:

  "remote" string => remotescalar("test_scalar","127.0.0.1","yes");

  "know" string => hubknowledge("test_scalar");

  "count_getty" string => hubknowledge("count_getty");

processes:

  # Use the enumerated library body to count hosts running getty

  "getty"

       comment => "Count this host if a job is matched",
       classes => enumerate("count_getty");

reports:

 !elsewhere::

    "GOT remote scalar $(remote)";
    "GOT knowledge scalar $(know)";
    "GOT persistent scalar $(xyz)";

}
```

```
#########################################################

bundle server access_rules()
{
access:

  "value of my test_scalar - $(sys.host)"
    handle => "test_scalar",
    comment => "Grant access to contents of test_scalar VAR",
    resource_type => "literal",
    admit => { "127.0.0.1" };

  "XYZ"
    resource_type => "variable",
    handle => "XYZ",
    admit => { "127.0.0.1" };

}
```

You can run this example on a single host, running the server, the agent and the hub (if you have Enterprise CFEngine). The output will be something like this:

```
host$ ./cf-agent -f ~/test.cf -K
R: GOT remote scalar value of my test_scalar - myhost
R: GOT knowledge scalar value of my test_scalar - myhost
R: GOT persistent scalar 1
```

Run job or reboot only if n out m systems are running. The self-healing chain - inverse Dominoes, Basic communication methods for orchestration, Distributed Orchestration between hosts with CFEngine Enterprise

6.2.2 Run Job or Reboot Only if N Out M Systems Are Running

The ability to base local promises on global knowledge seems superficially attractive in some cases. As a strategy, this way of thinking requires a lot of caution. We have to assume that all knowledge gathered about an environment is subject to errors, latencies and a dozen other uncertainties that make any snapshot of remotely assessed current state subject to considerable healthy suspicion. This is not a weakness of CFEngine—in fact CFEngine has mechanisms that make it as reliable as you are likely to find in any technology—rather it is a fundamental limitation of distributed systems, and it is strongly dependent on the architectures you build.

In the following example, we show how you can make certain decisions based on global, uncertain knowledge, allowing for the fact that the information is uncertain. In other words, we aim to err on the safe side. In this case we ask how could we reboot systems after an upgrade only if doing so would not jeopardize a Service Level Agreement to have at least 20 machines running at all times. Since the globally counted instances of a running process cannot be greater than the actual number, this particular problem satisfies the constraint of erring on the side of caution.

```
#############################################################
#
#  Keep a special promise only if at least n or m hosts
#  keep a specific promise
#
#  This method works with Enterprise CFEngine
#
#  If you want to test this on localhost, just edit /etc/hosts
```

```
#  to add host1 host2 host3 host4 as aliases to localhost
#
###############################################################

body common control
{
bundlesequence => { "n_of_m_symphony" };
inputs => { "cfengine_stdlib.cf" };
}

###############################################################

bundle agent n_of_m_symphony
{
vars:

  "count_compliant_hosts"  string => hubknowledge("running_
myprocess");

classes:

  "reboot" expression => isgreaterthan("$(count_compliant_
hosts)","20");

processes:

  "myprocess"

       comment => "Count this host if a job is matched",
       classes => enumerate("running_myprocess");

commands:

  reboot::

     "/bin/shutdown now";
}

#########################################################

bundle server access_rules()
{
access:

  "value of my test_scalar, can expand variables here - $(sys.host)"
    handle => "test_scalar",
    comment => "Grant access to contents of test_scalar VAR",
    resource_type => "literal",
    admit => { "127.0.0.1" };

  "running_myprocess"
    resource_type => "variable",
    admit => { "127.0.0.1" };

}
```

The code for this is surprisingly simple, but effective nonetheless. A generalization of this approach can be used to make phased deployments across waves of hosts containing minimum numbers of hosts.

6.2.3 The Self-Healing Chain—Inverse Dominoes

A self-healing chain is the opposite of a domino event. If a part of the chain is broken or "down," it will be revived. If these events depend on one another, then the resuscita-

tion of this part causes all of the subsequent parts to be repaired too.

Let's start with the more common case of the independently repairable services, such as one might find in a multi-tier architecture: database, web servers, applications, etc.

The following example can be run on multiple hosts or on a single host, using the aliases described in the example. It illustrates coordination through the use of CFEngine's remoteclasses function in the Enterprise edition to get confirmation of the self-healing structure. In fact, the verification of the self-healing is optional if one trusts the underlying system.

The first section defines the workflow:

```
#############################################################
#
# The self-healing tower: Anti-Dominoes
#
#  This method works with CFEngine Enterprise
#
#  If you want to test this on localhost, just edit /etc/hosts
#  to add host1 host2 host3 host4 as aliases to localhost
#
#############################################################

body common control
{
bundlesequence => { "weak_dependency_symphony" };
inputs => { "cfengine_stdlib.cf" };
}

body server control
{
allowconnects          => { "127.0.0.1" , "::1", @(def.acl) };
allowallconnects       => { "127.0.0.1" , "::1", @(def.acl) };
}

#############################################################

bundle agent weak_dependency_symphony
{
methods:

   # We have to seed the beginning by creating the tower
   # /tmp/tower_localhost

   host1::
    "tower"  usebundle => tier1,
               classes => publish_ok("ok_0");

   host2::
    "tower"  usebundle => tier2,
               classes => publish_ok("ok_1");

   host3::
    "tower"  usebundle => tier3,
               classes => publish_ok("ok_2");

   host4::
    "tower"  usebundle => tier4,
               classes => publish_ok("ok_f");
```

```
classes:

 ok_O:: # Wait for the methods, report on host1 only

  "check1" expression => remoteclassesmatching("ok.*","host2","yes",
"a");
  "check2" expression => remoteclassesmatching("ok.*","host3","yes",
"a");
  "check3" expression => remoteclassesmatching("ok.*","host4","yes",
"a");

reports:

  ok_O::
   "tier 1 is ok";
  a_ok_1::
   "tier 2 is ok";
  a_ok_2::
   "tier 3 is ok";
  a_ok_f::
   "tier 4 is ok";

  ok_O&a_ok_1&a_ok_2&a_ok_f::
   "The Tower is standing";

  !(ok_O&a_ok_1&a_ok_2&a_ok_f)::
   "The Tower is down";
}
```

The second part defines the content of the phases:

```
bundle agent tier1
{
files:

  "/tmp/something_to_do_1"
      create => "true";
}

bundle agent tier2
{
files:

  "/tmp/something_to_do_2"
      create => "true";
}

bundle agent tier3
{
files:

  "/tmp/something_to_do_3"
      create => "true";

}

bundle agent tier4
{
files:

  "/tmp/something_to_do_4"
      create => "true";
}
```

Finally, there is a little overhead:

```
bundle server access_rules()
{
access:

  "ok.*"
    resource_type => "context",
    admit => { "127.0.0.1" };

}

#############################################################

body classes publish_ok(x)
{
promise_repaired => { "$(x)" };
promise_kept => { "$(x)" };
cancel_notkept => { "$(x)" };
persist_time => "2";
}
```

If we execute this simple test on a single host, or allow it to be executed on distributed hosts, the chain forms and quickly stands up the system into a tower of dependencies.

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/self-
healing-chain.cf -K
R: tier 1 is ok
R: tier 2 is ok
R: tier 3 is ok
R: tier 4 is ok
R: The Tower is standing
```

If we break the tower, by giving it an impossible promise to keep, e.g., changing the name of the directory in tier 3 to something that cannot be created (for this illustration, we run in non-privileged mode and choose a directory name we do not have permission to create), then tier 3 will fail and the output looks like this:

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/self-
healing-chain.cf -K
Unable to make directories to /xtmp/something_to_do_3
 !!! System reports error for cf_mkdir: "Permission denied"
R: tier 1 is ok
R: tier 2 is ok
R: tier 4 is ok
R: The Tower is down
```

Clearly, whatever tier 3 is really supposed to do, any promise failure would result in the same behaviour. If we then correct the policy to make it repairable, the output heals quickly:

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/self-
healing-chain.cf -K
R: tier 1 is ok
R: tier 2 is ok
R: tier 4 is ok
R: The Tower is down
R: tier 3 is ok
R: The Tower is standing
```

6.2.4 A Domino Sequence

A different kind of orchestration is a domino cascade, which starts from some initial trigger and causes a change in one host that causes a change in the next, etc. These examples show how this can easily be carried out by CFEngine. Domino cascades can be done with Community or Enterprise editions, but are limited to single machines in each step.

The basic principle is shown below. This example has deliberately been made general enough to demonstrate on a single host with several aliases. If each host can be guaranteed to have a unique name and address, we could simplify the `hand_over` wrapper.

Note: to simulate this on a single host, start the server and agent with this same file as input, and make aliases to localhost in /etc/hosts as described in the example.

```
#############################################################
#
#  Dominoes
#
#  This method works with either Community or Enterprise
#
#  If you want to test this on localhost, just edit /etc/hosts
#  to add host1 host2 host3 host4 as aliases to localhost
#
#############################################################

body common control
{
bundlesequence => { "dominoes_symphony" };
inputs => { "cfengine_stdlib.cf" };
}

#############################################################

bundle agent dominoes_symphony
{
methods:

   # We have to seed the beginning by creating the dominoes
   # /tmp/dominoes_localhost

   host1::
    "dominoes"  usebundle => hand_over("localhost","host1","overtu
re");

   host2::
    "dominoes"  usebundle => hand_over("host1","host2","first_
movement");

   host3::
    "dominoes"  usebundle => hand_over("host2","host3","second_
movement");

   host4::
    "dominoes"  usebundle => hand_over("host3","host4","final_
movement"),
                classes => if_ok("finale");

reports:

  finale::
```

```
    "The visitors book of the Dominoes method"
       printfile => visitors_book("/tmp/dominoes_host4");

}
```

The wrapper which handles the inter-process communication for the methods is:

```
bundle agent hand_over(predecessor,myalias,method)
{

 # This is a wrapper for the orchestration

files:

  "/tmp/tip_the_dominoes"

        comment => "Wait for our cue or relay/conductor baton",
      copy_from => secure_cp("/tmp/dominoes_$(predecessor)","$(predec
essor)"),
        classes => if_repaired("cue_action");

methods:

  cue_action::

   "the music happens"

       comment => "One off activity",
     usebundle => $(method),
       classes => if_ok("pass_the_stick");

files:

  pass_the_stick::

   "/tmp/tip_the_dominoes"
       comment => "Add our signature to the dominoes' tail",
     edit_line => append_if_no_line("Knocked over $(myalias) and did:
$(method)");

   "/tmp/dominoes_$(myalias)"

    comment => "Dominoes in position to be beamed up by next agent",
     copy_from => local_cp("/tmp/tip_the_dominoes");

}
```

Finally, the methods themselves are defined:

```
bundle agent overture
{
reports:

  cfengine_3::

    "Singing the overture...";
}

bundle agent first_movement
{
reports:

  cfengine_3::

    "Singing the first adagio...";
}
```

```
bundle agent second_movement
{
reports:

  cfengine_3::

    "Singing second allegro...";

}

bundle agent final_movement
{
reports:

  cfengine_3::

    "Trumpets for the finale";

}

#############################################################

bundle server access_rules()
{
access:

  "/tmp"

        admit => { "127.0.0.1" };

  "did.*"
    resource_type => "context",
    admit => { "127.0.0.1" };

}

body printfile visitors_book(file)
{
file_to_print   => "$(file)";
number_of_lines => "10";
}
```

When executed, this produces output only on the final host in the chain, showing the correct ordering out operations. The sequence also passes a file from host to host as a coordination token, like a baton in a relay race, and each host signs this so that the final host has a log of every host involved in the cascade.

```
R: Singing the overture...
R: Singing the first adagio...
R: Singing second allegro...
R: Trumpets for the finale

R: The visitors book of the Dominoes method
R: Knocked over host1 and did: overture
R: Knocked over host2 and did: first_movement
R: Knocked over host3 and did: second_movement
R: Knocked over host4 and did: final_movement
```

The average time for such a cascade to complete will be half the length of the chain multiplied by the run-interval, if normal cf-execd splaytime is used. Without any splaying, the average time will be the run interval multiplied by the chain length. The completion time could be increased by using cf-runagent.

6.2.5 A Chinese Dragon Star Pattern

The Chinese dragon darts back and forth between different hosts, forming a chain of events and leaving a trail behind it. This pattern is much like the Domino pattern, except that it follows a star pattern with a hub at the centre. The orchestrated sequence of events follows the dragon from its lair to the first satellite host, then back to its lair to record the journey, then out to the next satellite, then back to its lair, etc.

A prototypical application for this kind of pattern is taking servers, one by one, off a load balancer (in the dragon's lair), and then upgrading them before reinstating them and moving on to the next host.

The pattern can be divided up into specific user-promises and generic library methods. We'll show both for completeness on the understanding that you would not need to write the library code each time. The basic orchestration is like the previous examples, with methods to be executed for each phase. Each of these has to be specified. Here, we make trivial methods that just print out a message for the purpose of illustration.

Also, as the methods execute, a logfile is collected, and is actually passed around between the hosts (the dragon), picking up entries as it goes. So the result is a summary of the execution to be printed as a file at the end.

```
#############################################################
#
#  Chinese Dragon Dancing
#
#  This method works with either Community or Enterprise
#  and uses named signals
#
#  If you want to test this on localhost, just edit /etc/hosts
#  to add host1 host2 host3 host4 as aliases to localhost
#
#############################################################

body common control
{
bundlesequence => { "dragon_symphony" };
inputs => { "cfengine_stdlib.cf" };
}

#############################################################

bundle agent dragon_symphony
{
methods:

   # We have to seed the beginning by creating the dragon
   # /tmp/dragon_localhost

    "dragon"  usebundle => visit("localhost","host1","chapter1");

    "dragon"  usebundle => visit("host1","host2","chapter2");

    "dragon"  usebundle => visit("host2","host3","chapter3");

    "dragon"  usebundle => visit("host3","host4","chapter4"),
                classes => if_ok("finale");
```

```
reports:

  finale::

    "The dragon is slain:"
       printfile => visitors_book("/tmp/shoo_dragon_host4");
}
```

The next parts define the phases themselves, as CFEngine methods:

```
# Define the phases

bundle agent chapter1(x)
{
# Do something significant here

reports:

 host1::
    " ----> Breathing fire on $(x)";
}

################################

bundle agent chapter2(x)
{
# Do something significant here

reports:

 host2::
    " ----> Breathing fire on $(x)";

}

################################

bundle agent chapter3(x)
{
# Do something significant here

reports:

 host3::
    " ----> Breathing fire on $(x)";

}

################################

bundle agent chapter4(x)
{
# Do something significant here

reports:

 host4::
    " ----> Breathing fire on $(x)";

}
```

Finally, there is the library of wrappers which can be reused and needn't be altered. We include this here so you can see how CFEngine uses simple peer-to-peer file copying to pass messages between the hosts, and generate a runlog as it goes. This is a highly elegant solution to the problem of distributed orchestration.

```
#############################################################
# Orchestration wrappers
#############################################################

bundle agent visit(predecessor,satellite,method)
{

 # This is a wrapper for the orchestration will be acted on
 # first by the dragon's lair and then by the satellite

vars:

  "dragon's_lair" string => "host0";

files:

  # We start in the dragon's lair ..

  "/tmp/unleash_dragon"

        comment => "Unleash the dragon",
      rename => to("/tmp/enter_the_dragon"),
        classes => if_repaired("dispatch_dragon_$(satellite)"),
     ifvarclass => "$(dragons_lair)";

  # if we are the dragon's lair, welcome the dragon back, shooed from
the satellite

  "/tmp/enter_the_dragon"

        comment => "Returning from a visit to a satellite",
      copy_from => secure_cp("/tmp/shoo_dragon_$(predecessor)","$(pre
decessor)"),
        classes => if_repaired("dispatch_dragon_$(satellite)"),
     ifvarclass => "$(dragons_lair)";

  # If we are a satellite, receive the dragon from its lair

  "/tmp/enter_the_dragon"
        comment => "Wait for our cue or relay/conductor baton",
      copy_from => secure_cp("/tmp/dragon_$(satellite)","$(dragons_
lair)"),
        classes => if_repaired("cue_action_on_$(satellite)"),
     ifvarclass => "$(satellite)";

methods:

   "check in at home"
        comment => "Edit the load balancer?",
      usebundle => switch_satellite(" -> Send dragon to
$(satellite)"),
        classes => if_repaired("send_the_dragon_to_$(satellite)"),
     ifvarclass => "dispatch_dragon_$(satellite)";

   "dragon visits"
        comment => "One off activity that the nodes carry out while
the dragon visits",
      usebundle => $(method)("$(satellite)"),
        classes => if_repaired("send_the_dragon_back_
from_$(satellite)"),
     ifvarclass => "cue_action_on_$(satellite)";
```

```
files:

   # hub/lair hub signs the book too and schedules the dragon for next
satellite

     "/tmp/dragon_$(satellite)"
         create => "true",
        comment => "Add our signature to the dragon's tail",
      edit_line => sign_visitor_book("Dragon returned from
$(predecessor)"),
     ifvarclass => "send_the_dragon_to_$(satellite)";

   # Satellite signs the book and shoos dragon for hub to collect

   "/tmp/shoo_dragon_$(satellite)"
         create => "true",
        comment => "Add our signature to the dragon's tail",
      edit_line => sign_visitor_book("Dragon visited $(satellite) and
did: $(method)"),
     ifvarclass => "send_the_dragon_back_from_$(satellite)";

reports:

  cfengine_3::

    "Done $(satellite)";

}

#############################################################

bundle agent switch_satellite(name)
{
files:

   "/tmp/enter_the_dragon"
        comment => "Add our signature to the dragon's tail",
      edit_line => append_if_no_line("Switch new dragon's target
$(name)");

reports:

  cfengine_3::
   " X Switching new dragon's target $(name)";
}

#############################################################

bundle edit_line sign_visitor_book(s)
{
insert_lines:

  "/tmp/enter_the_dragon"
        comment => "Import the current visitor's book",
    insert_type => "file";

  "$(s)" comment => "Append this string to the visitor's book";
}
```

```
#############################################################

bundle server access_rules()
{
access:

  "/tmp"

        admit => { "127.0.0.1" };

  "did.*"
    resource_type => "context",
    admit => { "127.0.0.1" };

}

#############################################################

body printfile visitors_book(file)
{
file_to_print   => "$(file)";
number_of_lines => "100";
}
```

Let's test it on a single host, equipped with aliases to see the entire flow. Without the trigger, this simply yields:

```
R: Done host1
R: Done host2
R: Done host3
R: Done host4
```

Now, if we create the trigger:

```
host$ touch /tmp/unleash_dragon
```

The result is:

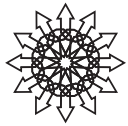
```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/dragon.cf
-K
R:  X Switching new dragon's target  -> Send dragon to host1
R: Done host1
R: Done host2
R: Done host3
R: Done host4

host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/dragon.cf
-K
R:  ----> Breathing fire on host1
R: Done host1
R:  X Switching new dragon's target  -> Send dragon to host2
R: Done host2
R: Done host3
R: Done host4
host$

host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/dragon.cf
-K
R:  ----> Breathing fire on host1
R: Done host1
R:  X Switching new dragon's target  -> Send dragon to host2
R:  ----> Breathing fire on host2
R: Done host2
R:  X Switching new dragon's target  -> Send dragon to host3
```

```
R: Done host3
R: Done host4
```

```
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/dragon.cf
-K
R:  ----> Breathing fire on host1
R: Done host1
R:  X Switching new dragon's target  -> Send dragon to host2
R:  ----> Breathing fire on host2
R: Done host2
R:  X Switching new dragon's target  -> Send dragon to host3
R:  ----> Breathing fire on host3
R: Done host3
R:  X Switching new dragon's target  -> Send dragon to host4
R: Done host4
host$ ~/LapTop/cfengine/core/src/cf-agent -f ~/orchestrate/dragon.cf
-K
R:  ----> Breathing fire on host1
R: Done host1
R:  X Switching new dragon's target  -> Send dragon to host2
R:  ----> Breathing fire on host2
R: Done host2
R:  X Switching new dragon's target  -> Send dragon to host3
R:  ----> Breathing fire on host3
R: Done host3
R:  X Switching new dragon's target  -> Send dragon to host4
R:  ----> Breathing fire on host4
R: Done host4
```

The output of the run log is tagged on to the end:

```
R: The dragon is slain:
R: Switch new dragon's target  -> Send dragon to host1
R: Dragon returned from localhost
R: Dragon visited host1 and did: chapter1
R: Switch new dragon's target  -> Send dragon to host2
R: Dragon returned from host1
R: Dragon visited host2 and did: chapter2
R: Switch new dragon's target  -> Send dragon to host3
R: Dragon returned from host2
R: Dragon visited host3 and did: chapter3
R: Switch new dragon's target  -> Send dragon to host4
R: Dragon returned from host3
R: Dragon visited host4 and did: chapter4
```



7. Monitoring, Reporting and Security

Traditionally, many engineers have viewed IT management as being principally about monitoring; the goal was to present alarms about incidents to humans so that they could file tickets and respond manually. This has been part of a long-running mind-set of having low trust in automation, and using humans to service the needs of machines. CFEngine is not about monitoring in the classical sense, but it *is* about re-humanizing the system administration experience, so that machines work for humans.

CFEngine provides its own kind of monitoring to simplify the problem of infrastructure maintenance, mainly so that it can verify its own promises. The aim is to build trust in the automation by building in as much of a guarantee of correctness as is possible. The monitoring it does is somewhat different from that of well-known tools like Nagios, for example.

CFEngine's monitoring is designed to be scalable rather than universal in nature. That said, the Enterprise edition of CFEngine competes quite well in a number of scenarios with several monitoring solutions that are based on much more resource expensive technologies, thanks to an innovative lazy-evaluation approach that reduces the amounts of data by orders of magnitude.

7.1 Autonomic Computing and Knowledge

Whatever we lack in understanding of the issues, we must make up for by trial and experience in the field. Having genuine insight into the systems we manage can help both the experienced and the inexperienced engineer to learn about behaviour.

CFEngine uses advanced machine-learning techniques to capture the multiple behaviours of systems cheaply, i.e., with a minimum of resources. This information integrates with a much larger plan to gather knowledge about systems and present that knowledge to humans in a meaningful way. The aim is not to make humans sit and watch line-traces of different metrics, but rather to hide such distracting information until it is actually needed; then to offer depth of insight and semantic relevance.

CFEngine Enterprise has an extended strategy for being knowledge-oriented that encompasses topics from how-to, through diagnostics, to causal inference. Making maximal use of the information gathered from a system for different audiences is where real business value can be added to the raw technology.

7.2 Reports Promises

The simplest response to an anomaly is to issue an alert. The **reports** section of **cf-agent**'s configuration is used for this; **reports** simply prints messages based on class membership, and every entry within this stanza must be preceded by an explicit class specification. Because the class **any** is always defined, you are not allowed to place an alert in this class, since it would always give rise to a message. This is probably an

expensive mistake if you have 10,000 hosts, so **cf-agent** deliberately makes it a little difficult for you.

You can test reports using some simple rules like these:

```
classes:
  jambalaya expression => "any"; # make a dummy class that is always
true

reports:
  jambalaya::
      "Gumbo!"
```

In general, reports are most useful if they are made only in unusual circumstances. For example, let's see what happens when a variable crosses a threshold:

```
bundle agent finale
{
vars:
 # Take these from the list of standard or custom measures
 "service" slist => { "www_in", "rootprocs", "otherprocs", "temp0" };
classes:
  "monitoring" expression => "any";
  "threshold_$(service)"
           comment => "Set a hard threshold of 50 units on all these",
           expression => isgreaterthan("$(mon.
value_$(service))","50");
reports:
 monitoring::
   "Service $(service) at $(mon.value_$(service)) over threshold"
      ifvarclass => "threshold_$(service)";
}
```

The output might look something like this

```
R: Service rootprocs at 152 over threshold
R: Service otherprocs at 68 over threshold
```

Whether you consider this information to be about security or merely about mapping out resources makes no difference. It is not necessary for CFEngine to make this distinction—after all, this is a matter for policy.

7.3 The cf-monitord Daemon

The basis of resource monitoring is the **cf-monitord** daemon. **cf-monitord** requires no configuration, although you can extend it with promises of its own in the CFEngine Enterprise. **cf-monitord** makes efficient use of tools and resources already existing on your system for monitoring—commands like **ps** and **netstat**. If you have **tcpdump** installed it can use it to monitor traffic too. It updates measurements locally on each host every 2.5 minutes, a carefully measured balance between too often (heavy on resources) and too seldom (missing important information). Nothing is sent over the network unless you have the Enterprise edition of CFEngine and have configured a reporting hub.

Using a smart-learning algorithm, **cf-monitord**:

- Learns the behavioural trends in each computer over weeks.
- Evaluates the current state of the resources against learned averages.
- Classifies the current state against learned averages into a basic ontology of performance.

When **cf-monitord** is active it records data in */var/cfengine/state* on each distributed host. Later, when **cf-agent** starts, it reads classes and variables about the current state of resources from a file *env_data* in the same directory. These variables have names of the form:

```
$(mon.value_tcp_in)                    Most recent measured value
$(mon.av_www_out)                      Weekly mean value
$(mon.dev_userprocs)                   Weekly standard deviation
```

The classes have the form:

```
rootprocs_high_anomaly                 Class related to the number of root processes.
loadavg_high_dev2                      Class related to the load average.
```

The first two items show the use of the **_in** and **_out** suffixes with metrics that distinguish incoming and outgoing connections on network ports. The latter two items illustrate the automatic classes that are defined based on comparing current resource usage with normal values. The second and third components of these classes, which function syntactically as suffixes to the metric keyword, have the following meanings:

```
low, normal, high                      Current value is <, ≈ or > normal.
dev1, dev2, anomaly    How far current value deviates from norm: 1, 2 or ≥3 standard deviations.
```

Like everything in CFEngine, reports about anomalies are subject to policy decisions. Here is an example **cf-agent** configuration for alerting about unusual activity:

```
bundle agent anomalies
{
reports:

rootprocs_high_anomaly::
   "RootProc anomaly high 3 dev on $(sys.host) at
        $(mon.env_time) measured value $(mon.value_rootprocs)
         av $(mon.av_rootprocs) pm $(mon.dev_rootprocs)"
      showstate => { "rootprocs" };

entropy_www_in_high.anomaly_hosts.www_in_high_anomaly::

   "HIGH ENTROPY Incoming www anomaly high anomaly dev!! on $(sys.
host)
        at $(mon.env_time) measured value $(mon.value_www_in)
        av $(mon.av_www_in) pm $(mon.dev_www_in)"

      showstate => { "incoming.www" };

}
```

Table 7.1 lists the available metrics tracked by **cf-monitord**.

Metric	Description	Port	_in and_out accepted?
CFEngine	CFEngine-related traffic	5308	yes
diskfree	Amount of free disk space		no
dns	Domain nameserver-related traffic	53	yes
ftp	File transfer protocol traffic	21	yes
icmp	Total ICMP traffic (e.g., **ping**)		yes
irc	Internet relay chat protocol traffic	194	yes
loadavg	Current load average		no
netbiosdgm	NetBIOS datagram service	138	yes
netbiosns	NetBIOS name service	137	yes
netbiosssn	NetBIOS session service traffic	139	yes
nfsd	NFS daemon traffic	2049	yes
otherprocs	Number of non-root process		no
rootprocs	Number of processes owned by root		no
smtp	SMTP traffic	25	yes
ssh	Secure shell remote login protocol	22	yes
tcp	Total TCP traffic	all	yes
tcpack	TCP packets with ACK flag set	all	yes
tcpfin	TCP packets with FIN flag set	all	yes
tcpmisc	All other TCP packets	all	yes
tcpsyn	TCP packets with SNY flag set	all	yes
udp	Total UDP traffic	all	yes
users	Number of logged in users		no
www	Web traffic (HTTP)	80	yes
wwws	HTTP protocol over TLS/SSL (HTTPS)	443	yes

Table 7.1 Standard Anomaly Measures

7.4 Measurement Promises

In the Enterprise edition of CFEngine, the monitoring daemon can play an even greater role in collecting targeted information about your environment. One of the reasons that conventional monitoring systems are expensive, in terms of resources, is that they run a lot of local scripts and probes to extract information about the environment. On modern operating systems, a lot of information is available in files and file-like interfaces. GNU/Linux, especially, is well equipped with data that can be read using a regular file-open. CFEngine is able to use its text-editing and extraction technologies to extract data with very little overhead and feed these into measurement streams and variable values that can be used either to define policy conditions, or to inform architects about trends for future planning.

7.4.1 Collecting Values

CFEngine can handle pipes to local programs as well as ordinary files. This allows us to extract system data in cases where there is no simple text interface. It might be similar, for example, to discovering the free disk on different Unix-like operating systems:

```
measurements:
  "/bin/df"

      handle => "free_disk_watch",
      stream_type => "pipe",
      data_type => "slist",
      history_type => "static",
      units => "device",
      match_value => file_system;
```

The data aggregated in this way are turned into graphs and variables, using the identifier given in the promise-handle, so that we can refer to

```
$(mon.value_free_disk_watch),
```

and variations described below.

7.4.2 Aggregating Values

In the following example, we can make a probe to add up the total memory usage from a process like httpd. Because CFEngine reads the process table regularly, there it caches the value for minutes at a time, which means we don't have to execute a bunch of shell commands to see it. A simple regular expression to parse the data gives us what we need. Using a match_value body, CFEngine can aggregate the multiple hits as either a sum or an average. This is very useful for monitoring memory leaks.

```
measurements:

"/var/cfengine/state/cf_otherprocs"
  handle => "httpd_memory",
stream_type => "file",
data_type => "int",
 history_type => "weekly",

match_value => get_rss("httpd");
```

The body referred to here is defined as follows:

```
body match_value get_rss(xxx)  # match RSS
{
select_line_matching => ".*$(xxx).*";
extraction_regex => "root\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\s+\S+\
s+(\S+).*";
#select_multiline_policy => "sum"; # sum or average over multiple
lines
}
```

7.4.3 Scanning Log Files

Measurement promises have many uses, not just extracting process information. For instance, they can be used to scan local log files in a distributed way. A lot of script kiddies will probe your site for vulnerabilities, using dictionaries of account/password combinations, looking for unguarded accounts or accounts with default passwords.

Most of these scans are harmless, because a well-maintained site will not use the default passwords that these hackers seek to exploit.

CFEngine can help you find out when you are being scanned. Because "sshd" logs its message through "syslog," we again need to filter lines based on the service name. On our system, authorization messages are routed to */var/log/auth.log*, and we would monitor it like this:

```
bundle monitor watch_break-in_attempts
{
measurements:
    "/var/log/auth.log"
             # This is likely what you'll see when a script kiddie probes
                                                              # your system
         handle => "ssh_username_probe",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*Invalid user.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
       # As scary as this looks, it may just be because someone's DNS
          # records are misconfigured - but you should double check!

         handle => "ssh_reverse_map_problem",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*POSSIBLE BREAK-IN ATTEMPT!.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
           # Someone is trying to log in to an account that is locked
                                       # out in the sshd config file

         handle => "ssh_denygroups",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*group is listed in
DenyGroups.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
           # This is more a configuration error in /etc/passwd than a
                                             # breakin attempt...

         handle => "ssh_no_shell",
    stream_type => "file",
      data_type => "counter",
    match_value => scan_log(".*sshd\[.*because shell \S+ does not
exist.*"),
   history_type => "log",
         action => sample_rate("0");

    "/var/log/auth.log"
     # These errors usually indicate a problem authenticating to your
                                             # IMAP or POP3 server
```

```
          handle => "ssh_pam_error",
     stream_type => "file",
       data_type => "counter",
     match_value => scan_log(".*sshd\[.*error: PAM: authentication
error.*"),
    history_type => "log",
          action => sample_rate("0");

     "/var/log/auth.log"
          # These errors usually indicate that you haven't rebuilt your
           # database after changing /etc/login.conf - maybe you should
          # include a rule to do this command: cap_mkdb /etc/login.conf

          handle => "ssh_pam_error",
     stream_type => "file",
       data_type => "counter",
     match_value => scan_log(".*sshd\[.*login_getclass: unknown class.*"),
    history_type => "log",
          action => sample_rate("0");
}
```

7.5 The mon Variable Context

CFEngine gives you access to the measurements assembled by the monitoring daemon from within a policy, so that you can adapt the automatic behaviour of the system to its current and normal states. All of the information observed by the monitor about one of your hosts is available to you as information.

The data from the monitoring daemon are all in a variable context called "mon". For example:

`$(mon.value_users)`	*The current value of users active*
`$(mon.av_users)`	*The average value of users at this time of day*
`$(mon.dev_users)`	*The deviation/spread around the average*
`$(mon.value_rootprocs)`	
`$(mon.av_rootprocs)`	
`$(mon.dev_rootprocs)`	
`$(mon.value_otherprocs)`	
`$(mon.av_otherprocs)`	
`$(mon.dev_otherprocs)`	
`$(mon.value_diskfree)`	
`$(mon.av_diskfree)`	
`$(mon.dev_diskfree)`	
`$(mon.value_loadavg)`	
`$(mon.av_loadavg)`	
`$(mon.dev_loadavg)`	

Data from the system discovery are in the "sys" context. For example:

`$(sys.class)`	*The main system hard classification*
`$(sys.cpus)`	*The number of CPU cores on the system*
`$(sys.crontab)`	*The location of the crontab file on this OS*
`$(sys.date)`	*The current data string*
`$(sys.doc_root)`	*The default document root of web servers on this OS*
`$(sys.flavor)`	*The exact operating system release*
`$(sys.flavour)`	*As above*
`$(sys.fqhost)`	*The fully qualified name of the host*
`$(sys.fstab)`	*The system's fstab filename and location*
`$(sys.hardware_addresses)`	*A list of MAC addresses for this host*

```
$(sys.hardware_mac[interface_name]) The MAC address for a given interface
$(sys.interfaces)                   A list of active network interfaces in this host
$(sys.ip_addresses)                 A list of IP addresses for this host
$(sys.ipv4[interface_name])         The different octets of the IPv4 address
$(sys.ipv4_1[interface_name])
$(sys.ipv4_2[interface_name])
$(sys.ipv4_3[interface_name])
```

For example, the open ports:

```
Port lists in mon

@listening_ports={'80','5308','631','22','53','1194'}
@listening_tcp6_ports={'631','22','53','80'}
@listening_tcp4_ports={'5308','631','22','53','1194'}

Address bindings in mon

tcp6_port_addr[631]=::1
tcp6_port_addr[22]=::
tcp6_port_addr[53]=::
tcp6_port_addr[80]=::
tcp4_port_addr[5308]=0.0.0.0
tcp4_port_addr[631]=127.0.0.1
tcp4_port_addr[22]=0.0.0.0
tcp4_port_addr[53]=0.0.0.0
tcp4_port_addr[1194]=127.0.0.1
```

A simple way to show the open ports on a host is to make the following reports promise. This assumes that the monitoring daemon is running:

```
reports:
  monitoring::
  "Open tcp4 port on $(mon.listening_tcp4_ports)";
  "Open tcp6 port on $(mon.listening_tcp6_ports)";
```

Nothing else is required. The output has the form:

```
R: Open tcp4 port on 5308
R: Open tcp4 port on 631
R: Open tcp4 port on 22
R: Open tcp4 port on 53
R: Open tcp4 port on 1194
R: Open tcp6 port on 631
R: Open tcp6 port on 22
R: Open tcp6 port on 53
R: Open tcp6 port on 80
```

All of the data measures by the monitoring daemon are available through variables, and are reported to the Mission Portal in the Enterprise edition for further knowledge integration.

7.6 Security-Related Scanning

In addition to the port scanning and the log scanning shown earlier, file change monitoring is a key aspect of security management. It is about detecting when file information on a computer system changes. You might or might not know that files are going to change. Expected changes are not usually a problem, but unexpected change can be problematic or even sinister.

The bulk of information on a computer is its file data. Change detection for file systems uses a technique made famous in the original open source program Tripwire, which collects a snapshot of the system in the form of a database of file checksums (cryptographic hashes) and then periodically rechecks the system against this database to see what has changed. Using cryptographic hashes is an efficient way of detecting change as it reduces file contents to a unique number, just a few bytes long, which can be stored for later comparison to detect change.

If as much as a single bit of information changes, the file hash will change by a noticeable amount. This is a very simple (even simplistic) view of change, but it is effective at warning about potential incursions to the system.

A cryptographic hash (also called a digest) is an algorithm that reads (digests) a file and computes a single number (the hash value) based on its contents. If so much as a single bit in the file changes, then the value of the hash will change. You can compute hash values manually—for example:

```
host$ openssl md5 /etc/passwd
MD5(/etc/passwd)= 1fbd82252c441d0e9539f8f7271ec2fe
```

There are several kinds of hash function. The most common ones are MD5 and SHA1. Recently, both of these algorithms have been superseded by the newer SHA2 set. Note that the FIPS 140-2 US government standard for encryption does not recognize the MD5 hash algorithm. The default algorithm for Enterprise grade CFEngine is now SHA256.

To use hash-based change detection we use files promises and the changes feature; first we ask CFEngine to compute file hashes for specified files and enter them into a database. Then, the same promise on subsequent runs will re-collect the data and compare the result to show what has been stored in the database.

Here is a simple CFEngine promise that checks for changes in /usr/local:

```
files:
  "/usr/local"
     changes      => detect_all_change,
     depth_search => recurse("inf");
```

This example uses the standard library template *"detect_all_change,"* which uses the two best hashing algorithms available to cross-reference change on any aspect of a file, including content, permissions and other metadata. This method is probably overkill for most scenarios, and the lesser *detect_content* is a cheaper alternative that just uses MD5 hashes to monitor file contents.

The first time this promise is kept, CFEngine collects data and treats all files as unchanged. It builds a database of the checksums. The next time the rule is checked, **cf-agent** recomputes the checksums and compares the new values to the "reference" values stored in the database. If no change has occurred, the two should match. If they differ, then the file as changed and a warning is issued either on the command line (if you are testing manually) or by email.

```
cf3: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
cf3: SECURITY ALERT: Checksum (sha256) for /etc/passwd changed!
cf3: !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
```

7.7 Patterns and Anomalies

The patterns revealed in the graphs produced by the Mission Portal can give us humans a gut impression of how a computer is being used by its users, and how the resources of the system are being consumed, over a time scale of several weeks. This information is useful for understanding performance in broad terms. If we see a large load at certain times of day or week, for instance, we could use this information to alter the configuration of resources at certain times to cope better with the load.

Let's clear up a misconception about anomaly detection: CFEngine does not care specifically about anomaly detection in the sense of *intrusion detection*. This was a flirtation popularized in the late 1990s, largely abandoned because there is no provable correlation between anomalies and intrusions. CFEngine *is* interested in the detection of resource anomalies, regardless of their cause. It can autonomically monitor the usage of a configurable subset of system resources: disk, CPU, number of users, network services, etc. Why would a tool like CFEngine bother to do this?

There are obviously advantages to monitoring such resource usage. We can better tune and configure a system if we know how it works on a good day, and compare this to how it fails on a bad day. On the other hand, in general, if you don't know what you are looking for in a system, there is little point in collecting reams of data about it, so a compressed scalable summary, such as provided by CFEngine Enterprise is an excellent place to start learning and building a low-cost relationship with your system.

To organize your understanding about the system, examine the following checklist on each host in your organization:

- *Which measurements are predictable, i.e., which have a clear trend with small variations?* These are stable and predictable features, a sign that things are efficient and under control.
- *Which measurements are dominated by uncertainty, i.e., have large error bars with no visible pattern?* This occurs if the resource usage is only sporadic and irregular. It could be because resource usage is so low that you cannot see any pattern. The resource is not playing any role in your organization.
- *Which measurements have top-heavy distributions?* This might signal a resource that is being throttled by something, perhaps a performance limitation.
- *Which combinations of anomalies are most common in your system?* This will tell you the potential sources for instability and unpleasant surprise in the future.

7.8 The Enterprise Mission Portal

In the Enterprise edition of CFEngine, access to the wealth of information provided by the agents around your network is simplified. The Mission Portal is a web interface that is served by an aggregator called **cf-hub**. The hub process adds a highly efficient aggregation process that can bring together five-minute updates from each managed CFEngine node into a central database, so that the data can be presented as reports and graphs.

7.9 Vital Signs from the cf-monitord

Part of the great untapped potential of the CFEngine Community edition is to present an adequate visualization of the machine-learning databases that store what is known about the behaviour and trend information over weeks and months on the agents. The Mission Portal collates and graphs these trends in a variety of ways.

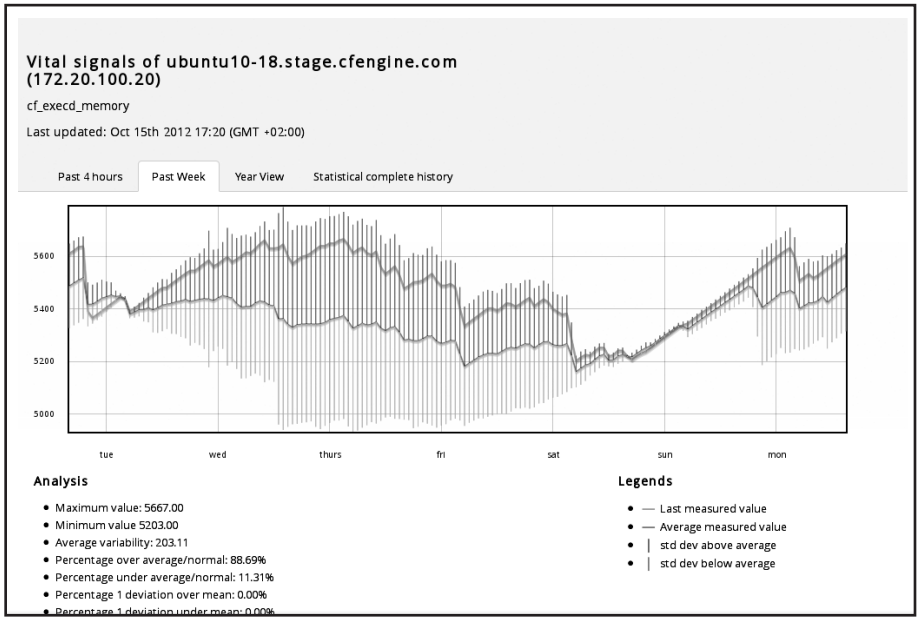
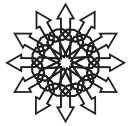


Figure 7.1: An example of a vital signs graph from the Mission Portal

In particular, a feature that distinguishes CFEngine from other monitoring tools is that it presents what is happening right now and compares it to what has typically happened in the past. This is much more useful than setting arbitrary alarm thresholds, as the patterns of behaviour on hosts are typically highly dynamic.

While the agents can respond to this information locally, without the need for any kind of aggregation, the human insight gained from seeing these reports should not be underestimated for capacity planning.



8. The CFEngine Management Process

Today businesses are more service oriented than before, as testified to by the increasing interest in good practices such as ITIL (the IT Infrastructure Library), COBIT (Control Objectives for Information and related Technology) and NGOSS/eTOM (New Generation Operational Support Service/enhanced Telecom Operations Map) as championed by the Tele-Management Forum. These process models have certain requirements of practice. How does CFEngine fit into this kind of process scenario?

8.1 Process Requirements

ITIL describes the following practices for good management:

- *Manager*: A service manager should be appointed to coordinate the management of services, specifically, to oversee configuration management within the organization. The manager should determine the requirements of the "service client"—in this case the configuration requirements of the organization. The configuration management workforce should be competent, i.e., well trained.
- *Documentation*: Policies should be documented. Performance towards goals should be reported. Security controls should be documented. CFEngine accomplishes these in a number of ways. The configuration language itself is designed to document the configuration policy to a large extent. You should also discuss the interaction of configuration options between different locations, processes and hosts. The big picture is only available to a configuration manager or engineer. Clear documentation is a sign of good engineering—but, as we all know, knowing what is good documentation for future contingencies is harder than we think.
- *Service implementation*: The service provider promises to deliver the agreed service. It must allocate the appropriate funds and resources to make this happen. In this case, an organization has to install CFEngine with an appropriate schedule for configuration management.
- *Monitoring*: The service provider promises to monitor its operation and seeks continuous improvement of service provision. This includes testing of the service. Monitoring here does not refer to the performance or configuration monitoring of CFEngine itself, but rather the extent to which the current configuration policy is effective in driving the larger goals of the organization.
- *Change and revision control*: Service Level Agreements should adapt and be subject to revision control. In this case, this means that we should frequently review the policies, expectations and CFEngine schedules. As changes are made to the configuration policy, those changes should be documented and versioned using a revision control system. For example, the Subversion revision control system is both convenient and easy to use.
- *Continuity*: ITIL requires a plan for continuity of an enterprise. Redundancy in critical dependencies must be established in order to provide the ability to continue to function in the absence of key dependencies and personnel. The

critical dependencies in CFEngine are, by design, minimal—as long as computers are running, CFEngine should be running. Humans who understand the policy itself are a dependency, and one can interpret this requirement as the need for at least two staff members who understand the CFEngine installation. The network might also be a dependency in some cases, although CFEngine is designed to work under unreliable, partial-communication conditions. If certain sources of data are required, e.g., servers for file copying, then failover servers can be provided.

8.2 Revision Control and Rollback

CFEngine does not currently version configurations internally, except to retain older versions of files that are changed during copying and editing. Tools for versioning policy at a high level will most likely be developed in the future.

One of the ideas that frightens system administrators about autonomic computing is that if a mistake is made (in policy or implementation), there is no clear way of "rolling back the change" to undo the damage. If you are thinking in this way, you are trapped by dangerous and costly preconceptions. System administrators often like to maintain the idea of a version control on their system configurations, as they generally believe that they are in control of every aspect of their configurations. This is false.

There is a basic conflict between the idea of policy and version control. Policy-based configuration management is about control of final state, and the scope of the changes involved in reaching it. This approach to the state is not versioned. Either a system is correct or it is incorrect.

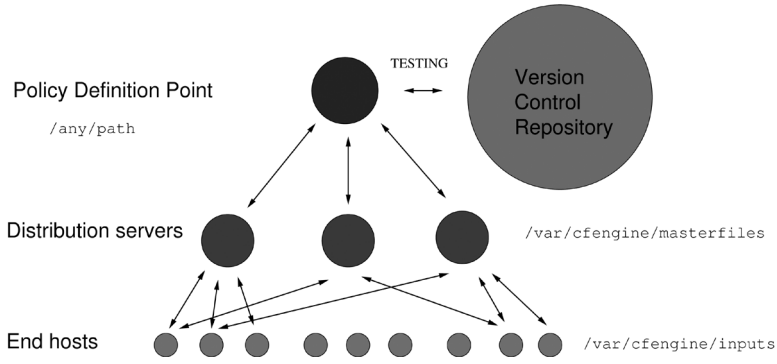


Figure 8.1: The role of Revision Control

We wish to caution readers: just because you undo the last changes you made on a computer does not mean that you will get back to the state you were in previously, because runtime (operational) changes and consequences are not necessarily reversed by a reversal of configuration.

So what do we recommend?

- Keep your source configuration files under a version control system like Subversion or Git so that you can track changes in your own thinking.
- Test new versions before rolling them out.
- When tested, update the master policy source with the new version.
- If, for whatever reason, the new policy has problems, either modify the policy again or go back to a previous version. In each case, CFEngine implements

changes in a forward direction, converging towards its policy, even if the policy has been rolled back.

This view of rollback and versioning might be an unfamiliar way of thinking to you, but it avoids several problems. The approach we advocate has the following properties:

- It avoids uncontrolled effects from ad hoc undo operations.
- It avoids complete reinstallation, e.g., versioning from image backup.
- By leaving the changes to CFEngine, you are certain that the end result is that which you wrote in your policy.

In this approach, you are encouraged to be forward thinking rather than taking a defensive backing off strategy. We think that anyone implementing configuration management should have sufficient expertise to be confident about their changes after testing.

8.3 DevOps and BizOps

The old cliché of a system administrator working in isolation to keep IT systems running is rapidly disappearing in favour of a new era of cooperation between IT services and business as cooperating units in an organization. As the tools for operational infrastructure have improved, the pace of Internet commerce has accelerated to the point where separation of these roles without meaningful cooperation becomes a hindrance: IT companies have learned crucial lessons about aligning business goals (development, in the case of Internet commerce) with the operations that deploy them.

The ability to manage and deploy infrastructure, including platforms and applications, in a timely fashion has been greatly accelerated by developments in configuration management and virtualization, especially Infrastructure as a Service "cloud" (IaaS). Still more important than the technologies themselves is the idea of an alignment of working cultures to support the needs of business, without overriding the expertise of IT engineers.

The term "DevOps" was coined by Patrick DeBois to express this synergy, and it was broken down into four main aspects of cooperation by John Willis, labelled CAMS for short:

- Culture—People are the key to cooperation. Human culture lies at the heart of integrating business processes, and this is where cooperation must begin.
- Automation—Having tools that bring about the right division of labour between humans and automation, allowing each to do what they are good at, is essential for scaling effort for rapid growth.
- Measurement—We need the insight into our progress to know whether or not business and IT are in fact aligned.
- Sharing—Sharing is a key part of knowledge management. It is how we spread expertise and learn from the experiences of others.

This list echoes and addresses the four barriers to progress, commonly cited in business literature:

- Not knowing the goal
- Not knowing how to achieve the goal
- Not being able to measure progress
- No responsible process or person

Ultimately these issues are all about knowledge—what we know and communicate in order to instigate change. CFEngine 3 was designed to support and invest in knowledge-based approaches to infrastructure.

8.4 The Role of Knowledge

Look at the figure below. We refer to this as the knowledge ladder for organizations. The maturity of a company in carrying out its operations is related to its internal expertise, by the way in which it climbs the knowledge ladder.

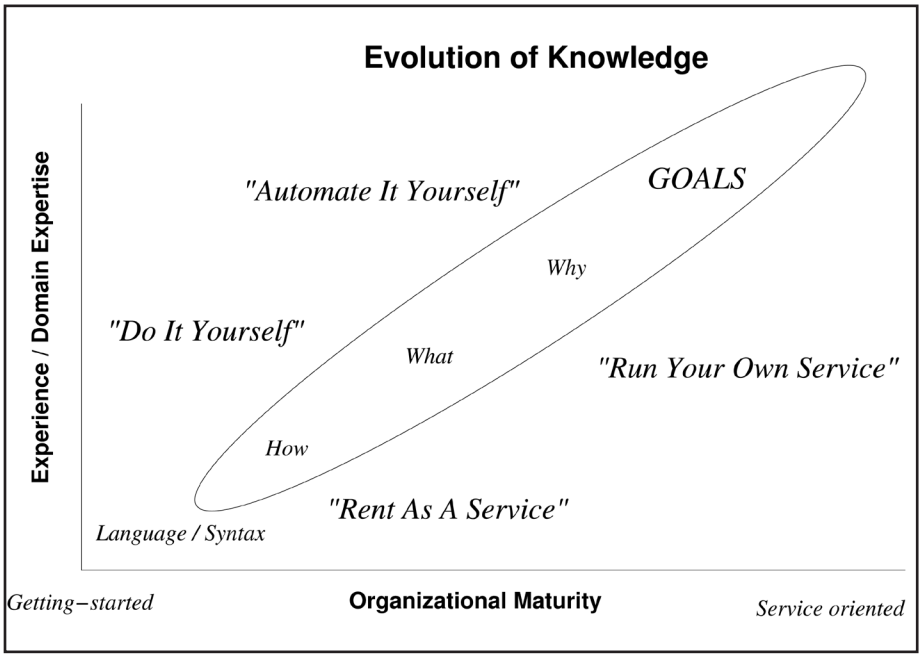


Figure 8.2: The knowledge ladder

If your company is young and immature, just getting started, then you are at the bottom left of this diagram—playing with tools to try to make things happen. You don't have a clear idea of what to do about your infrastructure because you don't know what is possible. Eventually, as you grow in knowledge, you will focus more on what it is you are trying to do (the how will be familiar) and even why you are trying to do it. Ultimately, you would like to be driven by a sense of purpose that comes from the raison d'etre of the organization itself: the company goals.

Now, if maturity of business grows faster than internal expertise, you will tend to outsource to save time and cost. On the other hand, if you have strong technical expertise, you will probably build everything yourself, as you will be able to do this cheaper in many cases. In either case, the goal is a balance between business maturity that drives service orientation, and a knowledge-based approach that favours keeping expertise close and in-house. One should never forget the most important question of all, which applies whether you are outsourcing or automating: *what happens when the service fails? Do we have the knowledge to recover?*

CFEngine 3 has been designed with this knowledge ladder in mind. It does not matter how you climb the ladder in your own organization, for the approach you choose

should build on your own internal culture. Although the basic possibilities are present in the Community edition of CFEngine, our aim with Enterprise CFEngine is to enhance this journey, with sound tools, in a way that respects the principles of DevOps, and hence ultimately the people behind your organization. Some of the available features of the knowledge integration in CFEngine Enterprise include:

- Source information:
  - A clear description of intended state (promises)
  - A clear description of actual state and estimate of uncertainty (vital signs, compliance, etc.)
  - Commentary and metadata around the above, to annotate relationships
  - Context-addressable documentation that can be linked to user need
  - Auto-generated manual pages and documentation at all levels
- Repetitive/habitual features:
  - Completely regular syntax, emphasizing one pattern for everything
  - Encourage incremental change
  - Allow dry-run to build trust by "looking/modelling ahead"
  - Web UI, command UI
- Human and machine reasoning features:
  - Clear separation of intended state into "what" and "how" affected
  - Model-oriented approach: promises, with a simple semantic syntax
  - Semantic index explains context and meaning of references, not just names
  - Story inference—tell me a narrative about something in the system and its influences
  - What questions can I ask about the system?

There are far too many things to say about CFEngine's knowledge strategy in this short overview. We recommend to you the Special Topics Guides on the CFEngine website to learn more about the possibilities.

Epilogue

Infrastructure engineering, assisted by configuration management, is still developing. As the IT industry changes before our very eyes, the challenges of infrastructure management change alongside it. CFEngine 3 was designed very much to support this rapid evolution, based on guiding principles that respect information flow and security, and scaling to large and complex scenarios with a minimum of cost. The story is far from over yet, and this book merely scratches the surface. We have yet to talk about embedded computing and the expansion of the true cloud of a society of users, expressing their freedoms to move around and consume services, for example. As a CFEngine user, we ask you to remember the shadow of the future: when you build ideas today, you are investing in principles that will have to last you for the decade to come. That decade will bring more surprises and turmoil than ever before, but we can meet the challenges with a new kind of expertise and cooperation. We hope that CFEngine will bring value to you on that journey.