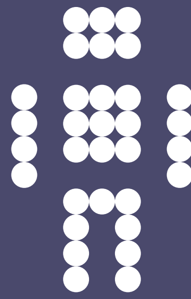
CFEngine



CFEngine 3 Best Practices

A CFEngine Handbook

CFEngine AS

Copyright © 2008- CFEngine AS

Table of Contents

1 Policy Style Guide 1
1.1 Arranging files 1
1.2 Where to define variables and classes 1
1.3 How to choose and name bundles 1
1.4 How to decide when to make a bundle 2
1.5 When to use a paramaterized bundle or method 2
1.6 When should classes be in `common` bundles? 3
1.7 When should variables be in `common` bundles? 3
1.8 When should variables be in local bundles? 3

2 Policy Dos and Don'ts 5
2.1 Never do 5
2.1.1 Never change system policy when humans are absent 5
2.1.2 Never embed simple shell commands 5
2.1.3 Never manage more than one cron job 5
2.2 Avoid 5
2.2.1 Avoid writing custom scripts 5
2.2.2 avoid running CFEngine without lock protection 6
2.3 Recommended (Try to) 6
2.3.1 Try to combine tests and operations during file searches 6
2.3.2 Try to make many small changes 6
2.4 Always do 6
2.4.1 Always document promises 6
2.4.2 Always keep coding to a minimum 7
2.4.3 Always use lists to make the same promise about multiple objects 7
2.4.4 Always use existing templates 8
2.4.5 Always use the system variables for system resources 8
2.4.6 Always use variables as pointers to paths and servers 9

3 Common Workflows 11
3.1 Anomaly Monitoring 11
3.2 Batch Jobs 12
3.3 Garbage Collection 13
3.4 Knowledge Updating 13
3.5 Name Service 13
3.6 Policy Distribution 14
3.7 Services 15
3.8 Security 16
3.9 Software Management 16

4 Quality Assurance around CFEngine 19
4.1 Policy changes 19
4.2 The policy decision flow 21
4.3 Version control and rollback 22
4.4 Delegating responsibility 22

1 Policy Style Guide

1.1 Arranging files

Base your files on high level services as you do with bundles, See Section 1.3 [How to choose and name bundles], page 1. The purpose of breaking up policy into files is to limit the scope of the policy to manageable amounts, making it easier to understand. It will only be easier to understand if the casual user can immediately locate promises from the name of the file.

You can place related files in subdirectories of the inputs to localize them. This also makes updating more efficient, as fewer objects need to be checked.

> The Enterprise Knowledge base allows you to search for promises, but everything will make more sense if promises are found in an intuitive place.

1.2 Where to define variables and classes

> Note that all CFEngine variables are globally accessable, by using their fully qualified name :`$(bundle.variable)`, or `@(bundle.variable)`, so placing variables in one place or another does not affect their accessability.

Variables should be defined as close to the place where they are used as possible. The user will expect to find variables defined:

- In the current bundle, first and foremost.
- In some common bundle for generic, global data.

Variables that define global aspects of configuration, e.g.

- Well known path names, e.g. document root.
- Site specific data, e.g. the email address of the support unit.

can be defined in `common` bundles. This places them in a neutral context.

The only reason to define a variable far from its place of use would be when writing generically re-usable methods and passing data as parameters, See Section 1.5 [When to use a paramaterized bundle or method], page 2. However, re-usability can make rules harder to understand.

1.3 How to choose and name bundles

Use the name of a bundle to represent a meaningful aspect of system adminstration, We recommend using a two or three-part name, that explains the context, general subject heading and special instance. Names should be service oriented and should guide non-experts to understand what they are about. e.g.

- app_mail_postfix
- app_mail_mailman
- app_web_apache
- app_web_squid

- app_web_php
- app_db_mysql
- garbage_collection
- security_check_files
- security_check_processes
- system_name_resolution
- system_xinetd
- system_root_password
- system_processes
- system_files
- win_active_directory
- win_registry
- win_services

1.4 How to decide when to make a bundle

Put things into a single bundle if:

- They belong to the same conceptual aspect of system administration.
- They do not need to be switched on or off independently.

Put things into different bundles if:

- All of the promises in one bundle need to the checked before all of the promises in another bundle.
- You need to re-use the promises with different parameters.

In general, keep the number of bundles to a minimum. This is a knowledge management issue. Clarity comes from differentiation, but only if the number of things is small.

1.5 When to use a paramaterized bundle or method

If you need to arrange for a *managed convergent collection* or *sequence* of promises that will occur for a list of (multiple) names or promisers, then use a bundle to simplify the code.

Write the promises, which may or may not be ordered, using a parameter for the different names, then call the method passing the list of names as a parameter to reduce the amount of code.

```
bundle agent testbundle
{
vars:

 "userlist" slist => { "mark", "jeang", "jonhenrik", "thomas", "eben" };

methods:

 "any" usebundle => subtest("$(userlist)");

}
```

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

bundle agent subtest(user)

{
commands:

 "/bin/echo Fix $(user)";

files:

 "/home/$(user)/."

    create =>  "true";

reports:

 linux::

  "Finished doing stuff for $(user)";
}
```

1.6 When should classes be in `common` bundles?

- When you need to use them in multiple bundles (because classes defined in `common` bundles have global scope).

Note, if you are converting from CFEngine 2 you should know the following. In CFEngine 2, all classes were global and it was common to define all classes in a big unmanageable list. That meant that there was a chance of class name collisions. CFEngine 3 has both local and global classes, allowing you to limit the scope of classes and define them more in context.

1.7 When should variables be in `common` bundles?

- For rationality, if the variable does not belong to any particular bundle, because it is used elsewhere. (Qualified variable names e.g. `$(mybundle.myname)`are always globally accessible, so this is a cosmetic issue.)

1.8 When should variables be in local bundles?

- If they are not needed outside the bundles.
- If they are used for iteration (without qualified scope).
- If they are tied to a specific aspect of system maintenance represented by the bundle, so that accessing `$(bundle.var)` adds clarity.

2 Policy Dos and Don’ts

This chapter lists a number of recommended practices.

2.1 Never do

2.1.1 Never change system policy when humans are absent

Never make system changes when humans are unavailable, e.g. just before going offline for the weekend. No matter how careful you have been, mistakes can be made and you need to have at least 24 hours experience with a running policy to lend it your trust.

2.1.2 Never embed simple shell commands

Do not embed simple shell commands with CFEngine `commands` promises, like this:

```
commands:

   # Don’t do this!

   "/bin/rm -r /tmp/xyz*";
   "/bin/mkdir /tmp/abcd";
```

WHY? Embedded shell commands like this cannot be managed by CFEngine, so none of the protections that CFEngine offers can be applied to the process. Moreover, this starts a new process, adding to the burden on the system.

Most importantly, this approach works like a covert channel, making changes that are not directly visible to CFEngine.

2.1.3 Never manage more than one cron job

When you run CFEngine, there is no reason to maintain separate cron jobs. Instead, use CFEngine’s time classes to work like a user interface for cron. This allows you to have a single, central CFEngine file which contains all the cron jobs on your system without losing any of the fine control which cron affords you. All of the usual advantages apply:

- It is easier to keep track of what cron jobs are running on the system when you have everything in one place.
- You can use all of your carefully crafted groups and user-defined classes to identify which host should run which programs.

WHY? This gives you a single point of definition for batch jobs. It encapsulates jobs under CFEngine’s tutelage, for improved control and security. Finally, CFEngine can collate and summarize the outputs from multiple scripts in a rational monitoring process.

2.2 Avoid

2.2.1 Avoid writing custom scripts

Do not spend your time writing scripts to embed within CFEngine. If you are doing this, you are not using the potential of CFEngine and you are not benefitting from the protections

and efficiencies that CFEngine offers. Custom scripts should be for your specific business operations, not for system maintenance.

If you are tempted to use scripts to achieve your needs, consider using `methods`, and if necessary consult with support personnel for advice.

2.2.2 avoid running CFEngine without lock protection

CFEngine's adaptive locking is an important system protection. You should not run CFEngine continuously without this protection, e.g. by running with the '`-K`' flag set, of by setting `ifelapsed` to zero for a promise.

WHY? System inconsistencies can result and unnecessary resources will be consumed.

2.3 Recommended (Try to)

2.3.1 Try to combine tests and operations during file searches

Searching through files on a disk is one of the most time consuming operations for a computer. If you have to do it, make sure that you are getting the most for your CPU-cycles and combine operations in a single promise. This allows CFEngine to optimize the resource use of the system.

```
files:

   "$(site)/app/webroot/img/inside/extmans"
       comment => "Copy the images for the private html documents",
     copy_from => cp("$(kbase)"),
         perms => p("root","644"),
   file_select => by_name(".*.png"),
  depth_search => recurse("1"),
        action => ifelapsed("60");
```

2.3.2 Try to make many small changes

Changes to policy should always be part of a serious and considered plan. They should not be *ad hoc*. That said, consideration of changes should not be so time-consuming that it cripples human resources, or leads to change-avoidance because it seems daunting.

It is better to make many small changes than few large changes. Large changes involve many interdependencies, which make them fragile to unexpected contingencies. The risk of large changes is high. The risk of small changes is low.

CFEngine makes it easy to make small changes frequently, without operational repercussions. As long as humans are on hand during the change to observe possible side-effects this.

2.4 Always do

2.4.1 Always document promises

Always add comment attributes to your promises to explain the intention.

```
files:
```

```
 # This is a throw-away comment, below is a full-bodied promise

   "/tmp/testfile"                      # promiser

     comment => "This is for keeps...", # Live comment
      create => "true",                 # Constraint 1
       perms => p("612");               # Constraint 2
```

If a promise has a stakeholder that is worthy of special mention, then use the promisee fields to add the name of this person.

```
files:

   "/tmp/testfile" -> { "stakeholder@company.com" },

     comment => "This is for keeps...", # Live comment
      create => "true",                 # Constraint 1
       perms => p("612");               # Constraint 2
```

If a promise depends on another promise being run before it, use the `depends_on` fields to document the handle of the other prior promise. This allows tracing of the impact chain.

```
files:

 "/tmp"

      handle => "make_temp",
     comment => "This is for keeps...", # Live comment
      create => "true",                 # Constraint 1
       perms => p("612");               # Constraint 2


   "/tmp/testfile"

  depends_on => { "make_temp" },
     comment => "This is for keeps...", # Live comment
      create => "true",                 # Constraint 1
       perms => p("612");               # Constraint 2
```

2.4.2 Always keep coding to a minimum

If you are coming to CFEngine from another scripting langauge, you will probably be tempted to add a lot of 'logic' to your CFEngine program, testing whether things are true and trying to control the order of things. This is not necessary. You should think of each promise as being a self-contained 'nugget' that requires little additional coding. The more coding you add, the more fragile a configuration becomes.

The hardest part of using CFEngine for programmers is letting go of the reins.

2.4.3 Always use lists to make the same promise about multiple objects

If you have a number of system resources that all make the same promise, then use lists to iterate over the resources in a single promise, rather than coding the same promise many times.

```
vars:

  "watch_files" slist =>  {
                          "/etc/passwd",
                          "/etc/shadow",
                          "/etc/group",
                          "/etc/services"
                          };
files:

   "$(watch_files)"

      comment      => "Change detection on the above",
      changes      => change_management_trip_wire;
```

2.4.4 Always use existing templates

Familiarize yourself with the current `CFEngine_stdlib.cf` file in the software distribution. This contains many body templates, e.g.

```
local_cp() remote_cp() secure_cp() if_elapsed() recurse()
```

Use these pre-existing body templates whenever possible, rather than inventing new ones. For example:

```
bundle agent update
{
files:

 "/path/to/copy"

    comment => "Update the policy files from the master",
    perms => u_p("600"),
    copy_from => local_cp("$(master_location)","localhost"),
    depth_search => recurse("inf");

}
```

WHY? The comprehensibility of your code to consultants and new employees is enhanced by standardization of practice. If the global CFEngine community uses the same set of idioms, then communicating policy will be simpler.

2.4.5 Always use the system variables for system resources

CFEngine provides indirection (pointers) to particular resources, through its '`sys`' variable context. These variables adapt to the operating system and user id under which CFEngine is run. Your policy will be more readily portable and you will need to code fewer exceptions if you use CFEngine's automatically adapting primitives, e.g. instead of writing '`/etc/resolv.conf`' for the name-service configuration file, use `$(sys.resolv)`.

```
files:

  "$(sys.resolv)"

     comment       => "Add lines to the resolver configuration",
     create        => "true",
     edit_line     => resolver,
```

```
     edit_defaults => std_edits;
```

2.4.6 Always use variables as pointers to paths and servers

You should avoid coding paths and names of resources directly in promises. Use instead a local or possible global variable to point to the resource instead. This brings consistency to the coding, often shortens the references, and provides a *single point of definition* for change.

```
bundle agent update
{
vars:

 # A standard location for the source point (single point of definition)

 "master_location" string => "$(sys.workdir)/masterfiles";

files:

 "$(sys.workdir)/inputs"

    comment => "Update the policy files from the master",
    perms => u_p("600"),
    copy_from => u_cp("$(master_location)","localhost"),
    depth_search => recurse("inf");

}
```

3 Common Workflows

This chapter concerns 'workflow processes' that should typically be dealt with on systems. A workflow process is represented by a *promise bundle* in CFEngine. None of the proposals here should be considered mandatory in any sense, but they do represent the norm.

We refer users to the CFEngine solutions guide for implementation details of specific solutions.

3.1 Anomaly Monitoring

Purpose:

The purpose of anomaly monitoring is to understand the stability of a system, both in terms of its run-time performance and its architectural structure. Sudden changes on a system can be separated from the normal slow variations.

Remarks:

Anomaly detection is enabled and performed by the `cf-monitord` daemon. Reporting of anomalies is not automatic however. Alerts must be promised explicitly. This is normally handled by a `reports` promise.

Change detection of the file system is handled by `files` promises in `cf-agent`.

Example:

```
bundle agent anomalies
{
vars:

  "sysdir" string => "/tmp";
  "files" slist => { "passwd", "shadow" };

classes:

  "no_$(files)" not => fileexists("$(sysdir)/$(files)");

files:

  # backup

  "/var/cfengine/inputs/$(files)"

      copy_from => emergency_save("$(sysdir)/$(files)");

  # restore

  "/tmp/$(files)"

       copy_from => emergency_save("/var/cfengine/inputs/$(files)"),
      ifvarclass => "no_$(files)";

reports:

 rootprocs_high_dev2::

   "RootProc anomaly high 2 dev on $(mon.host) at $(mon.env_time)
```

```
    measured value $(mon.value_rootprocs) av $(mon.average_rootprocs)
    pm $(mon.stddev_rootprocs)"

      showstate => { "rootprocs" };

 entropy_www_in_high&anomaly_hosts.www_in_high_anomaly::

   "HIGH ENTROPY Incoming www anomaly high anomaly dev!! on $(mon.host)
    - measured value $(mon.value_www_in) av $(mon.average_www_in) pm
     $(mon.stddev_www_in)"

      showstate => { "incoming.www" };

 entropy_www_in_low.anomaly_hosts.www_in_high_anomaly::

   "LOW ENTROPY Incoming www anomaly high anomaly dev!! on $(mon.host)
      at $(mon.env_time)
     - measured value $(svalue_www_in) av $(average_www_in) pm $(stddev_www_in)"

     showstate => { "incoming.www" };

 # etc.

}
```

3.2 Batch Jobs

Purpose:

Batch jobs are run on systems in order to perform basic house keeping functions such as updating databases or executing business related tasks.

Remarks:

Batch jobs should not be run every time CFEngine runs, so you need to limit the execution of each one carefully, using:

- Classes Classes for time and location.
- Locks The `ifelapsed` parameter determined how much time has to have elapsed before the job can be executed again.

Example:

```
bundle agent example
{
commands:

   # Exec on the first quarter after noon on Mondays

    Hr12.Q1.Monday::

      "/path/myscript -arg1 -arg2";

   # Exec every second quarter past hour, every day

    Q2::

      "/path/otherscript";
```

```
}
```

3.3 Garbage Collection

Purpose:

Garbage collection is required on systems to prevent temporary or antiquated files from consuming all available storage resources. It is impossible for a system to survive in the long term without throwing some data away.

Remarks:

Needless to say, care should be exercised when deleting anything from the system. There are many strategies to select carefully what is to be deleted. The `file_select` constraint is your friend.

Example:

```
bundle agent garbage_collection
{
files:

  "$(sys.workdir)/outputs"

    comment => "Garbage collection of any output files",
    delete => tidy,
    file_select => days_old("3"),
    depth_search => recurse("inf");

  "/tmp"

    comment => "Garbage collection of any temporary files",
    delete => tidy,
    file_select => days_old("3"),
    depth_search => recurse("inf");

}
```

3.4 Knowledge Updating

Purpose: Remarks: Example:

3.5 Name Service

Purpose: Every computer needs to know how to perform name directory lookups in the Domain Name Service. On Unix systems this requires it to manage the '`/etc/resolv.conf`' file.

Remarks:

Always use the `$(sys.resolv)` variable to refer to the file.

Example:

```
bundle agent name_resolution

{
files:

  "$(sys.resolv)"  # test on "/tmp/resolv.conf" #

     comment       => "Add lines to the resolver configuration",
     create        => "true",
     edit_line     => resolver,
     edit_defaults => std_edits;

}

bundle edit_line resolver

{
delete_lines:

  "search.*";
  "nameserver 80.65.58.31";

insert_lines:

  "search CFEngine.com" location => start;
  "nameserver 212.112.166.18";
  "nameserver 212.112.166.22";
}
```

3.6 Policy Distribution

Purpose:

In a centralized model of policy suggestion, policy updates are downloaded from a single point of definition, from one or more policy servers. Maintaining this flow of communication from 'central command' is what maintains that centralized command.

Remarks: It is not mandatory to centralize management, but usually there needs to be some automated process.

Example:

```
vars:

 "master_location" string => "/var/cfengine/masterfiles";

 "policy_server"   slist => { "62.109.39.150" },
                   comment => "IP address to locate your policy host.";

files:

  "/var/cfengine/inputs"

    handle => "update_policy",
    perms => system("600"),
    copy_from => u_scp("$(master_location)",@(policy_server)),
    depth_search => recurse("inf"),
```

```
          file_select => input_files,
          action => immediate;
```

3.7 Services

Purpose: Keeping services up and running, or taking down services that should not be running is both a matter of productivity and security.

Remarks: Example:

```
bundle agent services
{
vars:
 "serlist" slist => { "dhcp", "ntp", "sshd" };

  "sindex" int => readstringarray
                    (
                    "service",
                    "$(g.workdir)/inputs/fixservices-array",
                    "#[^\n]*",
                    ":",
                    "10",
                    "4000"
                    );

methods:

         "any" usebundle => fixservice
             (
             "$(service[$(serlist)][0])",
             "$(service[$(serlist)][1])",
             "$(service[$(serlist)][2])",
             "$(service[$(serlist)][3])",
             "$(service[$(serlist)][4])"
             );
}

bundle agent fixservice(service,tfiles,mfiles,procs,restart)
{
files:

 "$(tfiles)"
      perms => system("0600","root","root"),
  copy_from => mycopy("$(g.masterfiles)/config/$(mfiles)","$(g.phost)"),
    classes => cdefine( "$(service)_restart", "failed");

processes:

  "$(procs)"

      restart_class => canonify("$(service)_restart");

commands:

  "$(restart)"

      ifvarclass =>  canonify("$(service)_restart");
```

```
}
```

3.8 Security

Purpose: Security is a vast topic. You need to start with a security policy and then translate this into promises about the system. For instance you might promise file permissions and access rules. You might promise change monitoring or anomaly detection.

Remarks: This is an open ended topic. Security should be discussed as a human process, since most breaches come from within the system. CFEngine can then be used to implement hardening measures, and monitoring of important assets.

Example:

```
vars:

  "system_files" slist => {
                          "/etc/passwd",
                          "/etc/group",
                          "/etc/services"
                          };

  "secret_files" slist => {
                          "/etc/shadow"
                          };

files:


   "$(secret_files)"

      comment      => "Check permissions are secret on the above",
      perms        => mo("o-rwx","root");

   "$(system_files)"

      comment      => "Check permissions are correct on the above",
      perms        => mo("644","root");
```

3.9 Software Management

Purpose: Installing software and updating

These days most systems have some kind of package based management system. These vary in their intelligence from self-updating robots to simple dumb file repositories. CFEngine can manage the installation and subsequent customization/configuration.

Remarks: Installing software from some kind of source is only the first step. Thereafter, special settings must be harmonized with security policies and operational requirements.

Example:

```
vars:

  "match_package" slist => {
                           "apache2",
```

```
                            "apache2-mod_php5",
                            "apache2-prefork",
                            "php5"
                            };

packages:

  "$(match_package)"

     package_policy => "add",
     package_method => yum,
     classes => ok("software_ok");
```

4 Quality Assurance around CFEngine

A powerful tool like CFEngine can do great good, or cause enormous damage if used carelessly. It is essential to have a strict discipline when making changes. This is a human quality assurance process.

Your general rule of thumb should be: make small changes, not big releases.

4.1 Policy changes

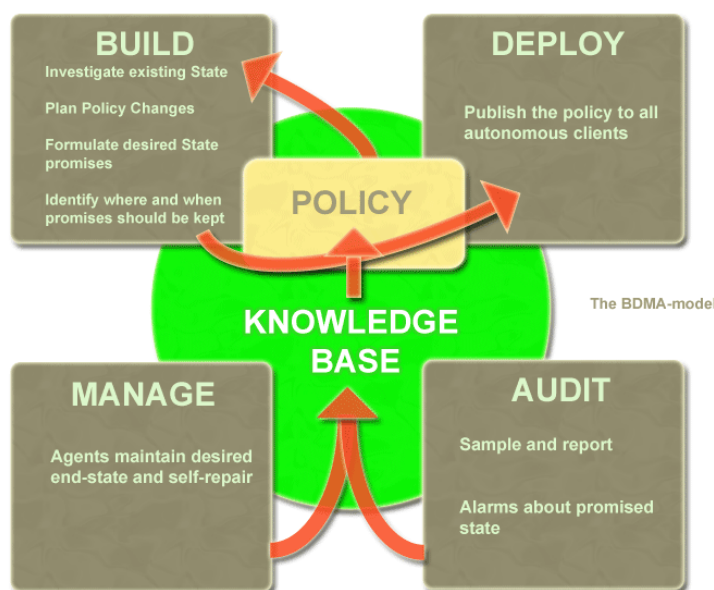
Changes to policy should always be part of a serious and considered plan. They should not be *ad hoc*. That said, consideration of changes should not be so time-consuming that it cripples human resources, or leads to change-avoidance because it seems daunting.

It is better to make many small changes than few large changes. Large changes involve many interdependencies, which make them fragile to unexpected contingencies. The risk of large changes is high. The risk of small changes is low.

CFEngine makes it easy to make small changes frequently, without operational repercussions. As long as humans are on hand during the change to observe possible side-effects this.

Consider the following issues in quality assurance:

- Create a schedule and policy for major changes.
- Plan to acquire the complete set of components for release.
- Assign human roles as well as machine roles for changes.
- Label new policy release items uniquely for tracking.
- Always document the policy changes using the comment fields.
- Test prior to releasing into the production environment.
- Test in the production environment on a small number of machines whenever possible.



There are four commonly cited phases in managing systems, summarized as follows (see figure):

- Build
- Deploy
- Manage
- Audit

These separate phases originate with a model of system management based on transactional changes. CFEngine's conception of management is some different, as transaction processing is not a good model for system management, but we can use this template to see how CFEngine works differently.

Build A system is based on a number of decisions and resources that need to be 'built' before they can be implemented. Building the trusted foundations of a system are the key to guiding its development. You don't need to decide every detail, just enough to build trust and predictability into your system.

In CFEngine, what you build is a template of proposed promises for the machines in an organization such that, if the machines all make and keep these promises, the system will function seamlessly as planned. This is how it works in a human organization, and this is how is works for computers too.

Deploy Deploying really means implementing the policy that was already decided. In transaction systems, one tries to push out changes one by one, hence 'deploying' the decision. In CFEngine you simply publish your policy (in CFEngine parlance these are 'promise proposals') and the machines see the new proposals and can adjust accordingly. Each machine runs an agent that is capable of implementing policies and maintaining them over time without further assistance.

Manage Once a decision is made, unplanned events will occur. Such incidents usually set off alarms and humans rush to make new transactions to repair them. In CFEngine, the autonous agent manages the system, and you only have to deal with rare events that cannot be dealt with automatically.

Audit In traditional configuration systems, the outcome is far from clear after a one-shot transaction, so one audits the system to determine to discover what actually happened. In CFEngine, changes are not just initiated once, but locally audited and maintained. Decision outcomes are assured by design in CFEngine and maintained automatically, so the main worry is managing conflicting intentions. Users can sit back and examine regular reports of compliance generated by the agents, without having to arrange for new 'roll out' transactions.

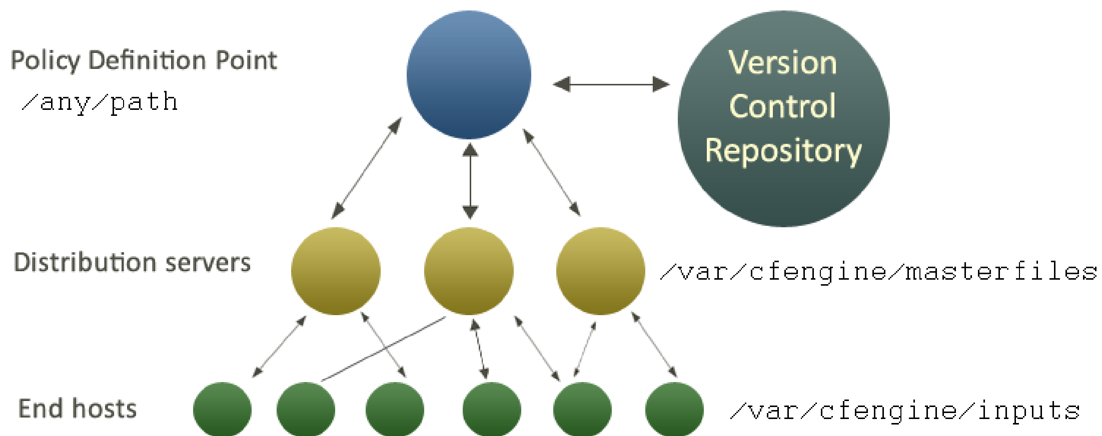
ROLL-OUT and ROLL-BACK? You should not think of CFEngine with a roll-out system, i.e. one that attempts to force out absolute changes and perhaps reverse them in case of error. Roll-out and roll-back are theoretically flawed concepts that only sometimes work in practice. With CFEngine, you publish a sequences of policy revisions, always moving forward (because like it or not, time only goes in one direction). All of the desired-state changes are managed locally by each individual computer, and continuously repaired to ensure on-going compliance with policy.

4.2 The policy decision flow

CFEngine does not make many absolute choices. Almost everything about its behaviour is matter of policy and can be changed. However, a structure for use, like the following, is recommended (see figure).

In order to keep operations as simple as possible, CFEngine maintains a private working directory on each machine referred to in documentation as WORKDIR and in policy by the variable `$(sys.workdir)`. By default, this is located at '`/var/cfengine`' or '`C:\var\CFEngine`'. It contains everything CFEngine needs to run.

The figure below shows how decisions flow through the parts of a system.



- It makes sense to have a single point of coordination. Decisions are therefore usually made in a single location (the Policy Definition Point). The history of decisions and changes can be tracked by a version control system of your choice (e.g. SubVersion).
- Decisions are made by editing CFEngine's policy file '`promises.cf`' on one of its included children. This process is carried out off-line.
- Once decisions have been formalized and coded, this new policy is copied *manually* (a human decision) to a *decision distribution point*, which by default is located in the directory '`/var/cfengine/masterfiles`' on all policy distribution servers.

  In this introduction, we shall assume that there is only one central policy distribution server, a specially-appointed server which is referred to simple as the `policy server`.
- Every client machine contacts the policy server and downloads these updates. The policy server can be replicated if the number of clients is very large, but we shall assume here that there is only one policy server.

Once a client machine has a copy of the policy, it extracts only those promise proposals that are relevant to it, and implements any changes without human assistance. This is how CFEngine manages change.

WHY DO THIS? CFEngine tries to minimize dependencies by decoupling processes. By following this pull-based architecture, CFEngine will tolerate network outages and will recover from deployment errors easily. By placing the burden of responsibility for decision at the top, and for implementation at the bottom, we avoid needless fragility and keep two independent quality assurance processes apart.

4.3 Version control and rollback

CFEngine does not provide specific tools for versioning promise specifications. It is recommended to use a tool such as subversion for this. CFEngine does allow you to track changes and keep versions of non-trivial changes, such as file content changes.

Subversion maintains revision numbers on files. It is useful to be able to refer to version names or numbers also in CFEngine. A version string can be added to files as follows:

```
body common control
{
version => 1.2.3
}
```

This defines the version number of a set of configuration files which is referred to in reference messages from CFEngine.

When CFEngine saves a current version of a file that it is modifying or replacing, by default such files are given a new extension and remain within the same directory which they were encountered. Alternatively, one can specify a repository directory to which such files can be moved instead. The repository location is specified in the `control` section:

```
body agent control
{
default_respository => "/var/cfengine/repository";
}
```

Files moved to the repository are given names reflecting their full path, with slashes replaced by underscore characters. For some, this creates a clearer overview of the changes that have occurred.

4.4 Delegating responsibility

In a large organization, you delegate responsibility for different issues to different teams. CFEngine has no meta-access control mechanism which can decide who may write policy rules on what issue. To create such a mechanism, there would have to be a monitor which could identify users, and an authority mechanism that would disallow certain users to write rules of certain types about certain objects on certain hosts. Although it is *possible* to create such a system, it would be both technically difficult, very cumbersome to use and would add a whole new level of complexity to policy and potential error to the configuration process.

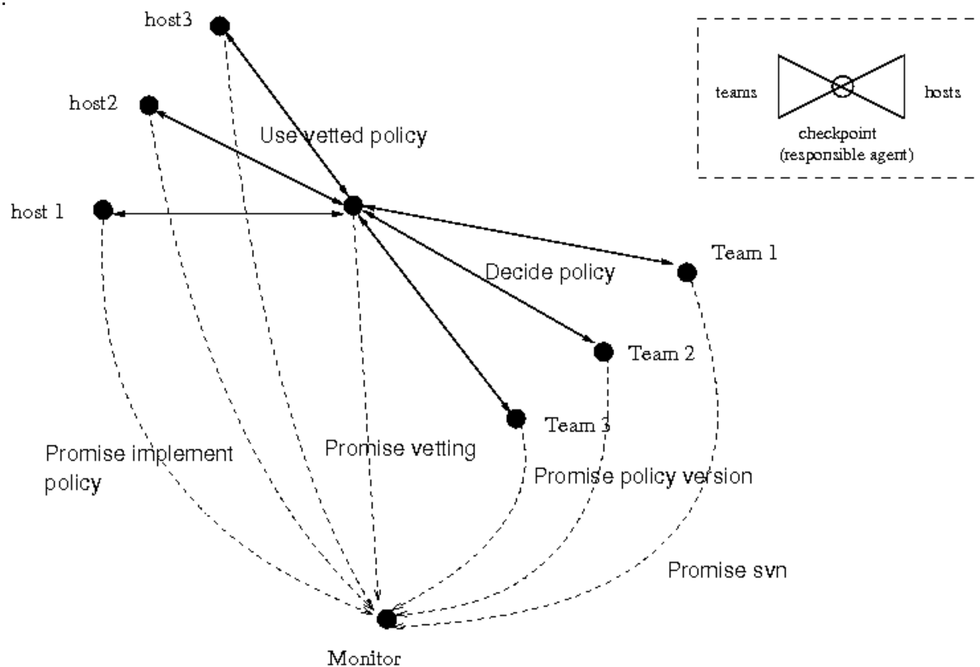
To keep matters as simple as possible, we avoid this and propose a different approach. Promise theory (CFEngine's basis) reveals a straightforward answer to model the security implications of this (see the figure of the bow-tie structure). A simple method of delegating is the following.

1. Delegate responsibility for different issues to admin teams 1,2,3, etc.

2. Make each of these teams responsible for version control of their own configuration rules.
3. Make an intermediate agent responsible for collating and vetting the rules, checking for irregularities and conflicts. This agent must promise to disallow rules by one team that are the responsibility of another team. The agent could be a layer of software, but a cheaper and more manageable solution is the make this another group of one or more humans.
4. Make the resulting collated configuration version controlled. Publish approved promises for all hosts to download from a trusted source.

A review procedure for policy promises is a good solution if you want to delegate responsibility for different parts of a policy to different sources. Human judgement is irreplaceable, and tools can be added to make conflicts easier to detect.

Promise theory underlines that, if a host of computing device accepts policy from any source, then it is alone and entirely responsible for this decision. The ultimate responsibility for the published version policy is the vetting agent. This creates a shallow hierarchy, but there is no reason why this formal body could not be comprised of representatives from the multiple teams.



Run several CFEngines? Another way to delegate CFEngine control for users that only require limited privileges would be to run several agents as non-root users. This only works however if the tasks delegated are very self-contained and require no special privilege.