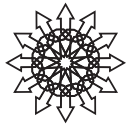

1. Introducing CFEngine

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.

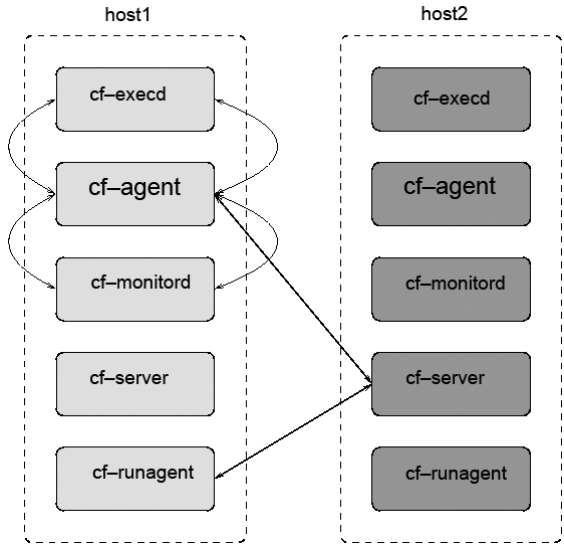

Figure 1.1: CFEngine Components and the Connections Between Them

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.