# Ansible|CFEngine

## Solution for the whole infrastructure lifetime

Vratislav Podzimek @ Northern.tech

## I. Problem definition

One of the so-called UNIX principles says that programs, or tools in general, should be written in a way that they *do one thing and do it well* (DOTADIW)[1]. Such programs or tools should then be combined to do multiple things that solve more complex problems. A simple example is a pipeline like this:

```
grep matching ./file | sort | uniq
```

which prints unique lines in `./file` containing the word `matching`. `grep` filters the matching lines and `uniq` makes sure that duplicate lines are not printed. However, `uniq` only merges **adjacent** matching lines and therefore `sort` needs to be used to group such lines together. Hence, three tools are used to do what initially looked like one thing to do. Of course there are many tools which would be able to do the same task by running just one of them (`AWK`, `Perl`, `Python`, etc.), but their complexity and ease of use is very different from that of `grep`, `uniq` and `sort`. And the solution based on one of the complex tools would still be based on the decomposition of the problem into smaller sub-problems to solve. Moreover, `grep`, `uniq` and `sort` are perfectly reusable for solutions of various other problems.

As described above, following the DOTADIW principle brings many benefits. However, the key seems to be what the *one thing* is or should be. *Ansible* and *CFEngine* are both *configuration management* tools and these days it is probably more precise to call them *infrastructure management* tools because with virtual machines, containers and software-defined everything, the infrastructure itself is managed by these tools as part of *configuration* in the extended meaning of the word. In relation to the DOTADIW principle, a natural question arises: Is infrastructure management a *one thing* that one tool should be doing well? The obvious answer is no because infrastructure management consists of a multitude of small things that need to be done – managing users, packages, configuration files, processes, services, etc. And then managing users in itself is about modifying files, creating directories, setting permissions, etc. Both *Ansible* and *CFEngine* follow the UNIX principle by using and combining many (system) components that *do one thing and do it well* to do the bigger sub-tasks of infrastructure management. However, on the other end of the scale from elementary bits that need to be done over bigger and bigger and more complex tasks there is the final element – the human whose objective it is to keep the infrastructure up and running as expected. And from their perspective the question whether infrastructure management is a *one thing* to do boils down to: *Is there a single tool that can **efficiently** do all the tasks they, as the human responsible for infrastructure management, need to do?*

## II. Deploy, maintain, grow

The ideal scenario of infrastructure management, or maybe any management in general, is to *start* a new project and then *maintain* it and *grow* it. And then, eventually, the project reaches an end of life and something new comes in as a replacement. These phases are by their nature very different. They result in different tasks, they require different approaches, and consequently, different tools. For example, when a team starts working on a new project, the manager (or technical leader) initially assigns work to the members of the team and confirms with all of them that they are all all set to start working on their tasks. The manager plays the active role and the communication is synchronous. However, when everybody is all set to start working on their tasks, they need to be independent and they have to play the active role

[1] https://en.wikipedia.org/wiki/Unix_philosophy#Do_One_Thing_and_Do_It_Well

asking the manager about what to do next etc. The manager, on the other hand, checks if the work goes as expected. So from the initial micro-management (*start*), the team and the project go into *maintenance* where everybody works on what they were assigned to do, checking from time to time if any changes have been made in the project and reporting the progress and results to the manager. And if the team *grows*, some micro-management is done again, focused on the new member of the team until they become part of the maintenance process of the whole team.

When deploying an infrastructure, the scenario is similar. First, all hosts are set up and checked if they are running as expected, then they are maintained and supposed to keep running as expected without direct control. If a new host is being added, it is again actively set up and checked. In the maintenance phase, it is highly beneficial if the hosts act as autonomous agents and keep running as expected without direct control and constant corrections from a single place, let alone from the human responsible for the infrastructure. And the tools required to do things efficiently differ between the phases. For example, if the machines are installed and set up physically, let's say in offices, the initial configuration can be done by the human when the machine is being set up. The same applies to when the infrastructure grows and a new machine is being added. However, if there are hundreds or thousands of such machines running then and a change needs to be made in the configuration, it is practically impossible to do it manually or, generally, in a one machine at a time way. The ideal solution is to make the change in one place and then let all the machines reflect the change. And again, once the change is done, the machines are expected to work according to the new configuration without external control or constant corrections. They should behave as autonomous agents "responsible" for their own configuration (if this can be said about machines).

The different phases also raise different questions. When deploying new machines (either when the infrastructure is being set up or a new machine is being added), the biggest focus is on *what **should** be running on the machines*. And things that are set up to be running are then supposed be running as expected for the lifetime of the machine or until its role changes. During maintenance, however, the biggest focus should be on *what **should not** be running on the machines*, either because of costs or because of data security concerns. And while when deploying, it is great to have a fine and direct control over how and when things that are supposed to be running are started and checked at the given moment, later, when the machines are running, it is much better to have the machines themselves making sure that things that are expected to be running are running and things that are supposed to **not** be running are not. Without any external direct control or corrections.

## III. The right tool(s) for the job(s)

As explained above, deployment and maintenance are two different sub-problems or sub-tasks of infrastructure management. And as such, they require different approaches and different tools. Going back to the *DOTADIW* UNIX principle, infrastructure management is not a *one thing* one tool should do well. It is a similar situation like in sports or racing – a super-fast sprinter will never have the endurance for long-distance competitions and a decathlon athlete will never be as good in a specific discipline as an athlete specializing on that particular discipline. A perfect approach would be to use 10 different specialized athletes for the 10 different disciplines of a decathlon. Unlike in decathlon, this is possible with tools for infrastructure management and so instead of using one tool for two very different problems where it only excels in one of them, two tools can be combined to solve the two problems in the most efficient way thanks to their specialization.

*Ansible*, with its push-based approach and direct and fast control of things, is a perfect choice for deployment, both when setting up the infrastructure and when adding new machines. It excels at running sequences of commands or operations over a set of machines making sure they are executed in the given order, with synchronization across the machines and providing fast results back. This is very useful for deployment when, for example, a load balancer should ideally not start sending traffic to a new content server before the content server is fully set up. And the responsible human needs fast feedback that both

the content server setup and the subscription of the content server to the load balancer succeeded. Or, if something fails, what were the error messages.

*CFEngine*, with its pull-based approach and treating machines as autonomous agents that are responsible for their own setup even without external control, is a perfect choice for long-term maintenance. For example, in case the hub (policy server) becomes unreachable, the autonomous machines remain in a state compliant with the policy even if corrections are needed to do so. And the out of the box monitoring and reporting capabilities of *CFEngine* allow the responsible humans to check that the infrastructure is running as it is supposed to be.

The combination eliminates the weaknesses of the two tools and allows their strengths to be fully utilized. The limited scalability of *Ansible*[2] is not an issue for deployment and the asynchronous nature of *CFEngine* is not an issue for maintenance. On the other hand, the direct and synchronous control of *Ansible* is perfectly suited for deployment of sets of machines while the autonomous behavior of *CFEngine* agents is a perfect fit for a long-time maintenance of thousands of machines. When combined, the full potential of the two tools can be unveiled.

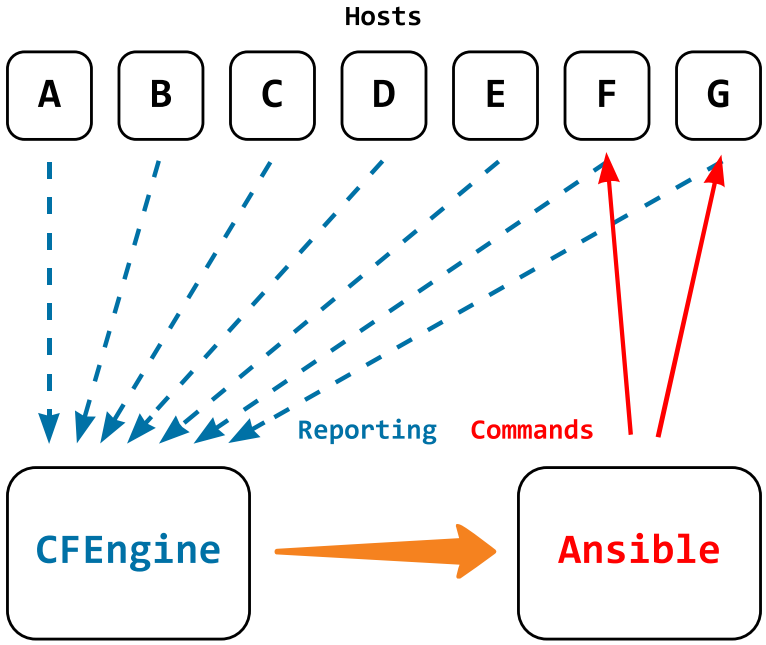


Fig. 1: Ansible and CFEngine managing hosts

## IV. Ansible|CFEngine

One of the keys for utilizing the DOTADIW principle, and maybe one of the biggest inventions brought by the UNIX operating systems, is the ability to efficiently combine the DOTADIW tools into pipelines. The *pipes* between these tools, represented by the symbol `|`, make sure that results from one tool are efficiently served to next tool in the pipeline. So what would be such a *pipe* between *Ansible* and *CFEngine*? The deployment comes first and is then followed by maintenance. Hence, at the end of deployment done with *Ansible*, the machine should be handed over to *CFEngine* for maintenance. This can be done with this *Ansible playbook* snippet (RHEL/CentOS6-specific for simplicity):

```
---
- hosts: all
  become: true
  vars:
    pkgs_url: https://cfengine-package-repos.s3.amazonaws.com/enterprise/Enterprise-3.15.2/
```

[2]https://sweetness.hmmz.org/2019-10-28-operon.html

```
    pkg: agent/agent_rhel6_x86_64/cfengine-nova-3.15.2-1.el6.x86_64.rpm
    cfengine_hub: 10.0.0.1

  tasks:
  - name: Install CFEngine
    yum: name="{{ pkgs_url }}{{ pkg }}" state=present

  - name: Bootstrap
    shell: "grep -q {{ cfengine_hub }} /var/cfengine/policy_server.dat ||
             /var/cfengine/bin/cf-agent -B {{ cfengine_hub }}"
```

Adding this to the playbook used for deployment of new machines makes sure the machine is then maintained by *CFEngine*.

### A. Hosts and categorization

Using the above `Ansible|CFEngine` pipeline also solves one of the most common challenges of infrastructure management, however surprising it may seem – knowing what is running in the infrastructure. How many machines, which operating systems, which software at what versions, etc. *CFEngine* provides answers to these questions out of the box just by installing it to the machines and bootstrapping them to a *CFEngine* hub.

Moreover, *CFEngine* also categorizes machines out of the box by assigning so called *classes* to them. It's easy to see all the *classes* assigned to a given machine. And with a simple script[3] utilizing the *CFEngine* API it is also easy to get dynamic and up to date lists of hosts for all or given classes. If a sequence of operations then needs to be performed on specific hosts, let's say all CentOS 6 hosts, in a command and control manner, *CFEngine* can serve the list of such hosts to *Ansible* which then only connects to the specific hosts instead of connecting to all hosts and skipping the steps on the ones not matching the criteria (being CentOS 6 hosts). Which means a great improvement in performance and scalability.And whenever possible, it is best to utilize the *CFEngine*'s categorization in a declarative way, describing the desired state for the particular class (category) of hosts instead of directly performing operations on specific lists of hosts.

### B. Policy

Deployment and maintenance should use the same set of policies. Ideally, a new machine should be set up in compliance with a policy that is then maintained over time. Combining *Ansible* and *CFEngine* means the policy needs to be expressed in two different languages understood by the two tools. *CFEngine* can, however, work with YAML files using the built-in `readyaml()` function. And generally, *CFEngine* works well with *data-driven* policy where data files define which users should be present in the systems, which packages should be installed, etc. Like in the following *packages.yaml* file:

```
- name: basic user packages
  package:
    name:
      - openssh
      - mutt
      - pass
      - lynx
      - ed
      - git
    policy: present
    version: latest
```

and the following CFEngine policy snippet:

[3] https://github.com/cfengine/core/blob/master/contrib/enterprise/report.pl

```
bundle agent __main__
{
  vars:
    "pkg_data" data => readyaml("$(this.promise_dirname)/packages.yaml");
    "indices" slist => getindices("pkg_data");


  packages:
    "$(pkg_data[$(indices)][package][name])"
      policy => "$(pkg_data[$(indices)][package][policy])",
      version => "$(pkg_data[$(indices)][package][version])";
}
```

This approach can be used to share data between the two tools or even for working with *Ansible* YAML files from *CFEngine*.

## V. Summary

In the above paragraphs we have seen that infrastructure management has two very different phases which require different approaches and different tools. *Ansible* and *CFEngine* are amazing tools for the two different jobs, and when combined together, their weaknesses are eliminated while their strengths are magnified. They create a great combination bringing the best of both worlds – direct command and control approach and cooperation of autonomous agents. A combination that can make fast and 100%-controlled changes while maintaining large infrastructures scaling to thousands of machines. And all that is needed is to not ask the question *Ansible OR CFEngine*, but rather do *Ansible*|*CFEngine*, where *Ansible* and *CFEngine* are just two different bits combined together with a bit-wise OR operation, and use them as the *Ansible*|*CFEngine* pipeline of two tools that each does one thing and does it well.