Ansible and CFEngine scalability

Vratislav Podzimek @ Northern.tech

I. Motivation

Scalability is an important feature of any infrastructure management solution. Either the to-be-managed infrastructure is big already or it is expected to grow as the business grows. Over time more and more resources are needed for CI/CD pipelines and more customers use the product(s). Generally, growing a business means more traffic and requests need to be handled by the infrastructure. Hence, scalability is an important metric for comparing infrastructure management tools when deciding which one to use. Or which *ones*.

In our previous white paper[1], we presented *Ansible* and *CFEngine* as two very different, but complementary approaches that together provide a great solution for infrastructure management thanks to the combination amplifying their strengths and eliminating their weaknesses. One of those weaknesses we mentioned was the limited scalability of *Ansible*. Some would call this *a known fact*, while others, including us, would call it *an assumption*. We are software engineers and so we didn't want to make claims based on assumptions. And even though there are some posts with measurements of *Ansible* performance across various numbers of managed hosts[2], we wanted to have some hard data based on a reproducible model.

II. Model

A. Scenario

In order to compare two infrastructure management tools, we needed a simple, small, yet still real-life example of what to configure on the given hosts. We decided to test deployment of a very basic setup which, however, represents a variety of real-life use cases – a high-availability proxy (aka load-balancer) with any number of backend web servers, all being separate virtual machines. Of course, a modern approach would involve pods, containers, and many other things instead of hundreds or thousands of separate virtual machines each running one service. But deploying services over a set of machines is the primary goal of infrastructure management. After all, every single machine should *serve* some purpose. As far as the tooling is concerned, it shouldn't matter if some hosts run a database, some others run a web server and some others run a caching layer. On all those machines the setup boils down to "prepare the environment for the service to run properly and securely, start the service and make sure everything keeps running and remains secure". Doing slightly different things on different hosts means the configuration description (*Ansible playbook*, *CFEngine policy*,...) needs to be more complex, but we wanted to keep things simple to reduce the effect of hidden variables and external dependencies in the model.

The desired configuration is described in the following specification:

- There is one machine serving as an HA proxy:
  - running *CentOS 7*,
  - with the *haproxy* software installed and the *haproxy* service running, serving content from all the backend web servers (described below) at port 80 using the round-robin strategy, and providing load-balancing statistics at port 8080,
  - with *iptables* installed, configured to block all incoming traffic except for ports 80 (HTTP traffic), 8080 (load-balancing statistics), 22 (SSH), and 5308 (CFEngine).
- There are N backend web servers (for N giving the scale factor):

[1] https://cfengine.com/wp-content/uploads/2020/09/AnsibleCFEngine_whitepaper_1.pdf

[2] E.g. https://sweetness.hmmz.org/2019-10-28-operon.html linked also from the previous white paper.

- half of them running *CentOS 7*, half of them running *Ubuntu 18*
- with the *Nginx* software installed and the *nginx* service running under the *nginx* user, serving content at port 8080 with *index.html* identifying the machine and the infrastructure management tool used to set it up,
- with *iptables* installed, configured to block all incoming traffic except for ports 8080 (HTTP content), 22 (SSH) and 5308 (CFEngine).

The above specification requires demonstration of basic capabilities of any infrastructure management solution like OS-specific configuration, package installation, rendering configuration files, working with users and services, etc. At the same time, it is very minimalistic with very few external dependencies and hidden variables (only the packages need to be installed from the repositories).

We implemented the above specification as *Ansible playbooks* as well as *CFEngine policy* in a publicly available repository[3]. Of course, there are multiple ways of achieving the same results, but we tried to keep things as simple and straightforward as possible. We compared *CFEngine Enterprise* with *AWX* which is the upstream project of *Ansible Tower*. We used the default configuration for both tools, except the number of forks used for the *ansible-playbook* process as explained later.

B. Hardware specification

There are two strategies for running scalability tests:

- using fake clients simulating the activities of real clients with lower resource requirements or
- using real clients and accepting the costs of resource requirements.

Both strategies can provide valid and valuable results, but the first approach is sensitive to hidden and often subtle differences between real clients and the simulated behavior. To eliminate that risk, increased by the fact that we are not *Ansible* experts, we decided to use the latter strategy, with real clients. Virtual machines in a public cloud, to be more precise.

We used our own tool, *cf-remote*[4], to spawn virtual machines in the *Google Cloud Platform* (*GCP*) and we chose the *n1-standard-4* machine for the *AWX* host, *CFEngine* hub, and the HA proxy and *e2-micro* machines for the backend web servers. The *n1-standard-4* machine type means 4 vCPUs, 15 GiB RAM, and local SSD. We could have used bigger machines for *AWX* and *CFEngine hub*, but we wanted to find the limits and compare the two tools without the need for spawning many thousands of client machines. The *e2-micro* machines used for the backend web servers only have 2 shared vCPUs, 1 GiB of on-demand RAM, and non-local SSD. Of course, these are the smallest and cheapest machines available in *GCP* which allowed us to limit the costs. At the same time, those machines are not handling any real traffic and so the resources they provide are fully available for the needs of the infrastructure management which should be very well covered.

III. Ansible scalability

With the VMs spawned, we ran the HA proxy playbook and then the web server playbook on increasingly bigger and bigger groups of hosts measuring the times it took to run the playbook. This simulated adding new hosts to the infrastructure and using fresh hosts for each group eliminated side effects potentially caused by previous changes done on the same hosts.

We repeated the measurements multiple times, each time with fresh virtual machines, to make sure we had enough results that were not affected by random anomalies in the cloud. The results are in the following figure 1.

As can be seen, there is a linear dependency between the number of hosts the playbook runs on and the time it takes to finish the whole *job* (a playbook run on a set of hosts). This dependency was expected due to the nature of how *Ansible* works – it executes each task from the given playbook on all the hosts

[3]https://gitlab.com/Northern.tech/CFEngine/web-cluster

[4]https://github.com/cfengine/core/tree/master/contrib/cf-remote

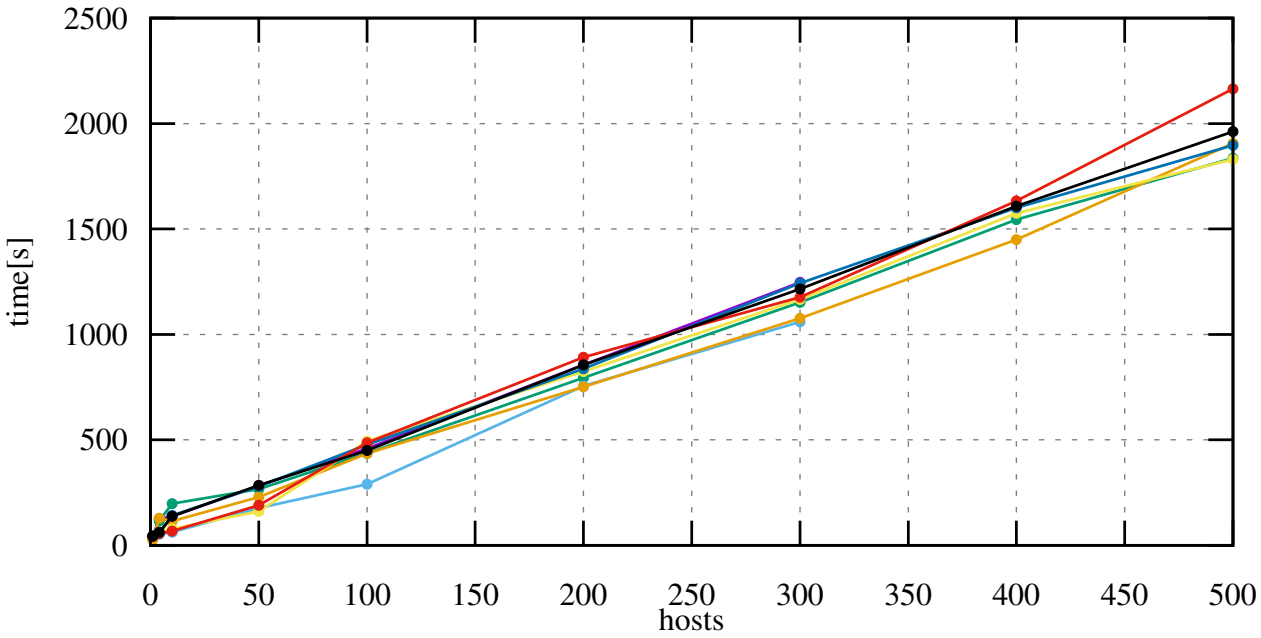


Fig. 1: Ansible playbook run lengths

from the set and then moves to the next task. The bigger the number of hosts, the longer it takes to execute every task on all of them. There is some level of parallelism given by the number of forks the *ansible-playbook* process creates to run the tasks on the hosts. The default is 5 and there seems to be no auto-scaling based on hardware detection in *AWX*. We used 20 for our measurements because the default yielded very long run times for the jobs and higher numbers didn't shorten the times (on the HW configuration we used). *AWX* also supports *job slices* which is another level of parallelism (multiple sub-jobs are started for the job, each with the specified number of forks). This defaults to 1 again with no auto-scaling and higher numbers didn't make the total times shorter on the HW configuration we used.

The testing playbook is very simple, with only 15 tasks – installing and removing a few packages, changing a couple of configuration files, creating one user, and starting two services. Still, running it on 500 fresh hosts was repeatedly taking around 30 minutes (1800 seconds). And again, given the nature of how *Ansible* works, this time would become longer with every extra task added to the playbook.

IV. CFEngine scalability

With the same specification implemented in *CFEngine policy*, we were able to deploy 3500 hosts using identical hardware configuration and the default 5 minute intervals for agent runs and report collection. It took around 5 minutes to install *CFEngine* on the machines and bootstrap them to the *CFEngine hub* (using `parallel ssh`) then they all fetched and evaluated the policy in a time a bit over 5 minutes at which point the deployment was done. In another 5 minute interval, the reports from the policy evaluation, together with lots of information about the hosts, were collected by the *CFEngine hub* and thus available to the administrator in the web user interface.

If we look at the following chart 2, showing the load in the last minute (reported by `uptime`) on the *CFEngine hub* machine over time, we can see that in every 5 minute interval there is still a window of time when the report collection is done and the load drops below 1. In other words, even with 3500 hosts bootstrapped to a 4 vCPU virtual machine we used, there was still some free resource capacity and so even more hosts could be bootstrapped to such hub machine without overloading it to a malfunctioning state.

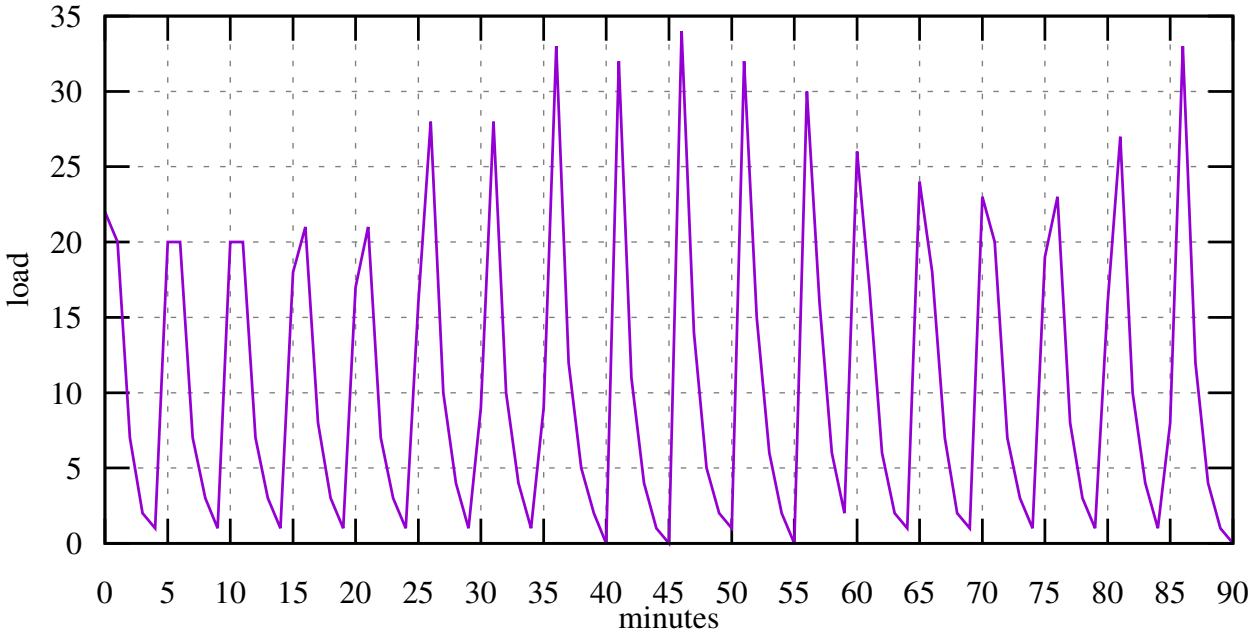


Fig. 2: System load on the CFEngine hub

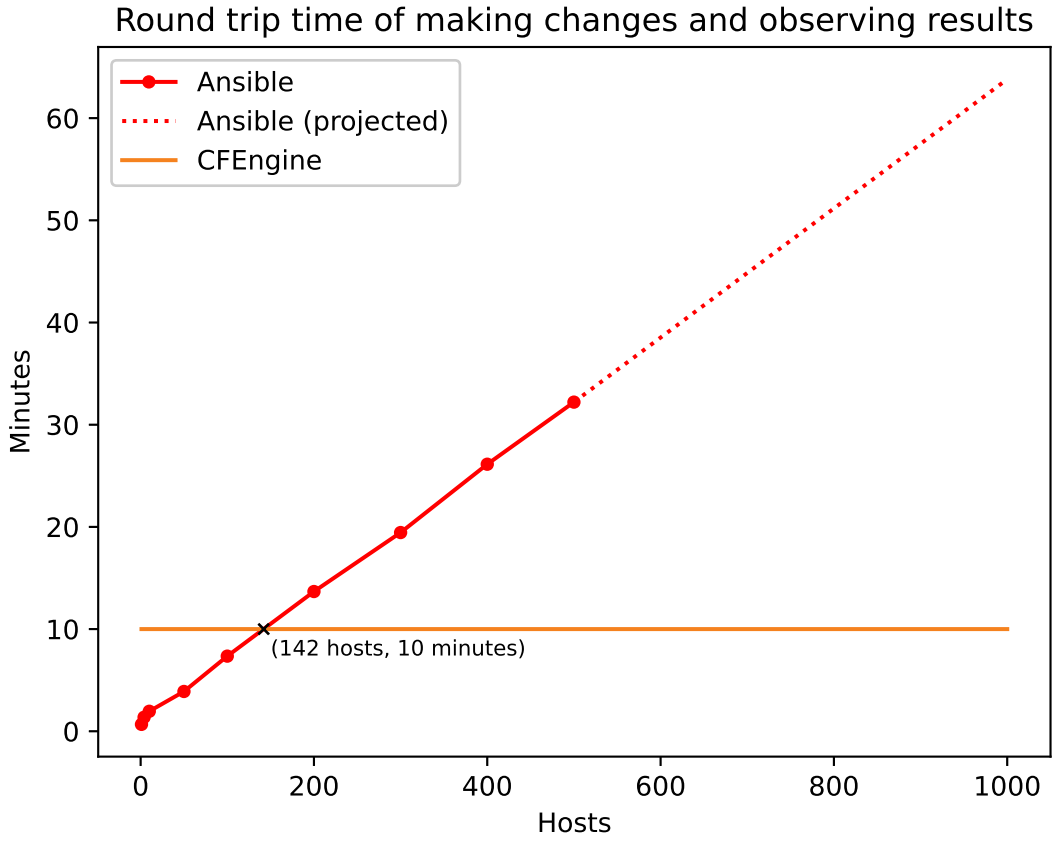
The chart also shows how *CFEngine hub* works when collecting reports – it tries to collect the reports from all the bootstrapped hosts as soon as possible at the specified times, relying on the Linux kernel schedulers that make sure all the *cf-hub* processing threads and the *PostgreSQL* backend are assigned CPU and I/O capacity and all the data gets through. This results in a very high load reported by the operating system, but thanks to the clever schedulers in the kernel the system doesn't choke and instead fully utilizes all resources. As long as it gathers and processes reports from all the hosts before the next specified report collection time, everything works as expected. The bigger the number of hosts bootstrapped to the hub, the higher the peak load is and the longer it takes for it to drop.

Thus for some number of bootstrapped hosts, the system reaches the point where it isn't able to collect and process all the data before the next report collection begins and the unfinished tasks start piling up. This *CFEngine* scalability limit depends on the hardware used and the amount of data the bootstrapped hosts report back to the hub. The reports are using a delta mechanism, so unchanged data is not being sent over and over. For this reason, it is impossible to tell what the real highest possible number of bootstrapped hosts for a hub with a given HW configuration is. In the simple scenario we tested, this limit would be over 5000 hosts as the load data suggests. We couldn't, unfortunately, test such a setup because we hit the limits of the cloud setup we used. For a real-life policy producing more reporting data a bit more powerful hardware might be needed, but having 5000 hosts bootstrapped to one *CFEngine* hub is generally not a problem and is officially supported by *CFEngine Enterprise*.

In contrast to how with *Ansible* adding more tasks to the playbook increases the total time of running the playbook on the given set of hosts, the autonomous *CFEngine* hosts evaluate the policy independently and extending the policy only results in longer runs of the *cf-agent* process on the hosts. And similar to how reporting can scale up to the given report collection interval, the *cf-agent* runs can scale up to the agent run interval (5 minutes by default). The length of the agent run depends on the actions the policy triggers. Some actions can potentially take a long time, for example installing many packages with slow connection to the repositories, however, there are mechanisms to avoid triggering such actions too often.

V. Conclusions

A simple test setup deploying an HA proxy machine with several backend web servers has shown us the differences in scalability between *Ansible* and *CFEngine*. While the exact numbers are interesting, one has to take into account the specificity of the setup – the *Ansible* playbook and *CFEngine* policy together with the hardware used for the tests and measurements. More interesting are the general observations demonstrating how the different architectures of *Ansible* and *CFEngine* affect the scalability of the two infrastructure management solutions. The measurements also show that the general assumption that *Ansible* is good for quickly pushing changes to the managed hosts and getting instant feedback in contrast to *CFEngine* where hosts only pull and evaluate new policy at a certain interval, only applies up to a certain number of hosts. For specific playbooks and with a different hardware setup, the numbers would vary, but even our simple scenario has shown that the linear growth means that the time required to run a playbook on a set of hosts quickly becomes impractical with a growing infrastructure. The typical (default) 10 minute "round trip time" of *CFEngine* hosts pulling the new policy, evaluating it, and reporting the results back, on the other hand, means that changes happen faster and sooner with a certain number of hosts, ~150 or more (in our scenario) [5].



As we suggested and explained in our previous white paper, *Ansible* and *CFEngine* are two very different but complementary solutions for infrastructure management. This analysis proves the point by demonstrating the big differences between the two tools in scalability where one can handle an order of magnitude more hosts in a 10 minute interval and the other can handle a small number of hosts in an order of magnitude shorter time. In other words, making some changes to a small number of hosts or

[5] As mentioned earlier, it is important to note that, apart from the number of forks for *Ansible*, we stuck with default values and configuration. Both tools could be tweaked and optimized for different host counts and environments.

deploying them is faster to do with *Ansible*. Pushing a change to a big number of hosts, maintaining them according to a given policy or deploying them is faster to do with *CFEngine*. A typical infrastructure management process involves both of those activities and so an optimal solution is the combination of the two tools.

VI. References

1) Vratislav Podzimek (2020): *Ansible|CFEngine, Solution for the whole infrastructure lifetime*, https://cfengine.com/wp-content/uploads/2020/09/AnsibleCFEngine_whitepaper_1.pdf
2) David Wilson (2019): *Operon: Extreme Performance For Ansible*, https://sweetness.hmmz.org/2019-10-28-operon.html
3) O. Tange (2011): *GNU Parallel - The Command-Line Power Tool*, ;login: The USENIX Magazine, February 2011:42-47.