# Infrastructure as Code and Configuration Management with Puppet

William Berg
DD2482 Essay

May 7, 2023

## Contents

*"I certify that generative AI, incl. ChatGPT, has not been used to write this essay. Using generative AI without permission is considered academic misconduct."*

# 1 Introduction

IBM succinctly defines IT infrastructure as "the combined components needed for the operation and management of enterprise IT services and IT environments" [1]. If this sounds very comprehensive, that's because it is! These IT infrastructure components refer to all of the hardware and software elements that work together to provide some service. On the hardware side, they include physical resources like servers, as well as networking hardware like routers and switches. On the software side, this could mean web servers, databases, and virtualization software. The point is that all of these components have to interoperate effectively and reliably to ensure client satisfaction. To reach a state where all of this is achieved, a lot of work needs to be done. The different components need to be set up, connected if necessary, configured, etc. Traditionally, all of the components of an organization's infrastructure had to be managed manually, which naturally means that all of this work had to be done manually. Such an approach can lead to several problems, I will highlight a few:

1. Manual configuration is a very error-prone procedure. Making a mistake in this process can be very costly if large-scale infrastructure ends up failing as a consequence of human error.

2. In addition to potential costs incurred in the case of unexpected errors, it is also the case that manual work is generally more expensive than automated work. An increased financial burden is often a consequence of traditional infrastructure management.

3. Manual work takes longer. Critical tasks such as network management, setting up and configuring servers, and deployment will be more time-consuming.

4. A lack of infrastructure transparency can occur when everything is done manually. It can be hard for maintainers to properly visualize what the infrastructure actually looks like.

An organization fraught with such problems will not be able to operate at peak performance. In the coming section, I will introduce the concept of infrastructure as code as a means to remedy these issues.

# 2 Infrastructure as Code

Red Hat define Infrastructure as code (IaC) as "the managing and provisioning of infrastructure through code instead of through manual processes" [2]. With this, we should already get an inkling that many of the problems mentioned in the previous sections may be resolved by adopting this approach. All of those problems stemmed from the fact that this "managing and provisioning of infrastructure" was done manually after all. But what does managing and provisioning through code mean? Well, in the case of IaC, this entails creating

definition files that specify the desired configuration of infrastructure resources. The code in these files generally follows one of two approaches: the declarative or the imperative approach. The imperative approach is perhaps more familiar to your typical system administrator, as it involves programmatically defining the commands that are required to reach the wanted system state. When these files are processed, you can be sure that those commands are executed. This differs from the declarative approach, which seems to be the favored method among contemporary IaC tools. With the declarative approach, you simply *declare* the desired state of the system, and the IaC tool will perform the required configuration to reach that state automatically.

Another difference in IaC methods is the concept of push vs. pull with regard to how managed nodes receive their configurations. With the pull approach, the managed machine pulls its configuration from the controlling server, while with the push approach, the controlling server pushes the configuration to the managed machine.

When these definition files have been created, the manual work is done and they can be applied to create the desired infrastructure. Granted, you will probably have to create multiple such files for the different components of your infrastructure, but having to only do this once saves organizations a lot of time - and by extension, money. You also have the assurance that your previously time-consuming infrastructure tasks can be repeated without unexpected behavior. In this way, IaC enforces consistency by avoiding manual configuration [3]. This also means that the engineers of the organization can focus on the development of products that are actually delivered to end users, instead of the underlying infrastructure. Finally, I want to expand on the last issue raised in the previous section, namely that it may be difficult to maintain a proper view of the infrastructure. This problem is indirectly addressed by these definition files, as they, just by virtue of existing, provide maintainers with infrastructure "documentation". It can be seen as a form of self-documentation, if you will [4]. Furthermore, because IaC tools define configurations in files, they can be put under version control just like any other file, thereby allowing for a comprehensive view of the infrastructure history.

DevOps is all about automation for speed and reliability. The infrastructure as code workflow is relevant in this regard since it allows developers to automate infrastructure tasks that are repetitive and prone to error. In the next section, I will focus on an IaC tool called Puppet as a means of doing so.

# 3 Puppet

## 3.1 Language Components

Let's start this section with a very pertinent question: what is Puppet? Well, according to their own documentation (a resource that will be used extensively for this section), Puppet is software that "provides tools to automate managing your infrastructure" [5]. The previous section placed a lot of emphasis on the definition files used for automating infrastructure configuration; in Puppet, such files are called *manifests*. Manifest files contain code written in Puppet's declarative configuration language (Ruby-based) that describes how resources should be configured. Let's look at an example:

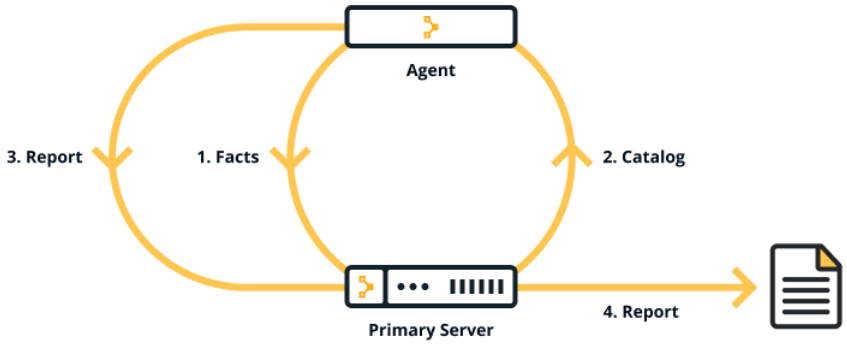
```
service { 'ntp':
  name      => ntpd,
  ensure    => running,
  enable    => true,
  subscribe => File['ntp.conf'],
}
```

Note the declarative approach in action here; you are not specifying the commands to execute to reach the desired state, instead, you are simply telling Puppet what you want. (It should be noted however that Puppet does provide limited support for the imperative approach as well.) In the Puppet language, manifest files are written by defining *resources* and *classes*. Resources are system elements such as files, packages, users, services, etc., whereas classes are just a logical grouping of related resources. The most important part of writing Puppet code is resource declaration, indeed every other construct in the language exists as syntactic sugar to simplify this process. When you declare a resource in a manifest file, you tell Puppet its desired state. In the code above, for example, you are telling Puppet to manage the ntp resource. More specifically, you are telling Puppet to configure ntp to ensure that the service is running, that it is enabled at startup, and that it is refreshed whenever its configuration file is modified. Declaring resources and creating manifest files like this is really what Puppet is all about.

Manifests are kept inside a directory tree called a *module*, and each module manages a single infrastructure task. When it comes to actually configuring nodes, however, the manifest files and modules are not used directly. Instead, Puppet manifests and configuration information are compiled into node-specific *catalogs*, which are then applied against the node in question. So, manifests contain the source code that defines a desired configuration, while catalogs are the compiled versions of that configuration that are actually applied to target nodes.

## 3.2 Architecture

Now that we have taken a look at Puppet's most important components, let's give an overview of Puppet's architecture. Puppet has support for running in a stand-alone architecture or client-server architecture. I will focus on the client-server architecture since that is Puppet's most common mode of operation. In Puppet, clients are referred to as agents and run the Puppet agent application, while servers are referred to as masters and run the Puppet server application. In this architecture, the clients are the nodes we want to manage while the servers are the nodes doing the managing. A "puppet run" facilitates automatic configuration management and describes the process by which a Puppet agent retrieves and applies configuration from the Puppet master. This process starts at the Puppet agent, where Puppet's inventory tool Facter collects information about that agent. This includes system information such as its IP address and operating system. These facts are then sent to the Puppet master in a manifest file, where, upon retrieval, a catalog is compiled using the received manifest as well as other relevant manifests stored in modules on the server. All managed Puppet agents request their own node-specific catalogs and then apply the described state when the catalog is received. Since the Puppet agents initiate communication with the Puppet master to receive their configuration, it should be obvious that Puppet employs the pull method to configure managed nodes (although support for the push method does exist). When this is done, the agents send a status report back to the Puppet master that outlines any errors as well as changes made. Finally, based on the results of the received report, the Puppet master also generates a report for logging purposes. This process is depicted in the image below:




By creating manifest files and scheduling these "puppet runs" at regular intervals, configuration management can be automated for a very large number of machines. Puppet is undoubtedly a useful piece of software for accomplishing this; however, it is far from the only tool used to automatically manage infrastructure. In the next section, we will broaden our horizons by taking a look at the state of the art in IaC technology.

# 4 State of the Art

While Puppet is a popular choice for managing infrastructure as code, there are several other tools available that offer similar functionality. In this section, we will take a look at two of them. One of the more well-known options is Ansible, which is most often described as a configuration management and automation tool. It uses a simple YAML syntax to describe the desired infrastructure state. This code is written in the Ansible version of a Puppet manifest, called a "playbook". A playbook consists of a list of "plays" that each executes a set of one or more tasks. Ansible provides support for both imperative and declarative code in its playbooks, but it is often described as imperative. This makes Ansible slightly different from the primarily declarative Puppet. Another notable difference is that Ansible is agentless, meaning that it does not require any software to be installed on managed nodes. This differs from Puppet, where each client must have the Puppet agent software installed in order to pull configuration from the Puppet master. This also leads us to the final difference I wish to highlight. How can managed Ansible nodes pull their configuration from the primary server without any software telling them to do so? The answer is that they can't; instead, Ansible uses the push method, where the primary server pushes the configuration to target hosts. [6][7]

Chef is another tool similar to both Puppet and Ansible. In Chef, the definition file is called a "recipe", which is a collection of resources and attributes that define the configuration of a single component or service. This is very similar to the Puppet manifest file. Furthermore, recipes can be combined into "cookbooks", which are used to manage the configuration of entire systems; something akin to the Puppet module. Chef recipes are written in Ruby, which is also very similar to Puppet's own Ruby-based language. Much like Puppet, Chef also provides support for both the imperative and declarative approaches to configuration management, but prefers the imperative approach. Chef also uses the pull method for configuring its clients, but unlike Puppet, provides no support for the push method. The Chef architecture is also similar to Puppet's, both tools are agent-based with Puppet clients running the Puppet agent program and Chef clients running the Chef agent program.

It's worth noting that there are many more IaC tools available, and if the reader is interested, they can for example look up Terraform [8], a very popular alternative to the tools listed in this essay. For a visual overview of the section, I recommend referring to Table 1.

Table 1: Comparison of IaC Tools

| **Tool** | **Agent Type** | **Approach** | **Configuration delivery** |
|---|---|---|---|
| Puppet | Agent-based | Primarily declarative | Primarily pull |
| Chef | Agent-based | Primarily imperative | Pull |
| Ansible | Agentless | Primarily imperative | Push |

# 5 Reflection

Infrastructure as code and the associated toolsets have certainly had a great impact on the management of IT infrastructure in large organizations. At the beginning of this essay, I highlighted several problems with the traditional approach and explained how the IaC process can help mitigate them. However, it is important to note that employing such an approach requires a certain level of expertise and knowledge. Development teams must have a solid understanding of the underlying infrastructure components, as well as the programming languages and tools used to define and manage their resources. In other words, engineers and developers will need to invest a lot of time to learn all of these new technologies (Puppet, Ansible, Chef). Furthermore, they will have to switch to an entirely new workflow, and perhaps engage in large-scale IT migration if necessary; all with the hope that it will be beneficial in the long run. Granted, this is a problem associated with adopting DevOps practices in general, but it is nonetheless important to reflect on these issues from the IaC perspective.

# 6 Conclusion

In this essay I have introduced the problems with manually managing IT infrastructure tasks and presented the concept of infrastructure as code as a potential solution. I then continued to introduce Puppet, an IaC tool used to automate the management and configuration of IT infrastructure. I also compared similar tools in a section on the state of the art and reflected on the potential challenges of adopting the IaC workflow. Given all of this, the key take-away message that I want to leave the reader with is that IaC tools like Puppet offer an automated approach to infrastructure management, which can help improve time efficiency, prevent errors, and save on cost.

# References


[1] IBM. *What is IT Infrastructure*. 2023. URL: https://www.ibm.com/topics/infrastructure (visited on 05/05/2023).

[2] Red Hat. *What is Infrastructure as Code (IaC)*. 2022. URL: https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac (visited on 05/05/2023).

[3] Microsoft. *What is Infrastructure as Code (IaC)?* 2022. URL: https://learn.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code (visited on 05/05/2023).

[4] Chiradeep BasuMallick. *What Is Infrastructure as Code? Meaning, Working, and Benefits*. 2022. URL: https://www.spiceworks.com/tech/cloud/articles/what-is-infrastructure-as-code/ (visited on 05/05/2023).

[5] Puppet. *Welcome to Puppet 8.0.0*. 2023. URL: https://www.puppet.com/docs/puppet/8/puppet_index.html (visited on 05/05/2023).

[6] M. Altun. *Imperative Approach and Declarative Approach in Automation*. 2021. URL: https://medium.com/clarusway/imperative-approach-and-declarative-approach-in-automation-72cdfe82b1ae (visited on 05/07/2023).

[7] Red Hat. *Ansible vs. Puppet: What you need to know*. 2022. URL: https://www.redhat.com/en/topics/automation/ansible-vs-puppet (visited on 05/07/2023).

[8] *Terraform*. https://www.terraform.io/. Accessed: 2023-05-07.