Automated Configuration of Monitoring Systems in an Immutable Infrastructure

Adrián Medina-González(✉), Sodel Vazquez-Reyes, Perla Velasco-Elizondo, Huizilopoztli Luna-García, and Alejandra García-Hernández

Autonomous University of Zacatecas, 98000 Zacatecas, ZAC, Mexico
{adrian.medina, vazquezs, pvelasco, hlugar, alegarcia}@uaz.edu.mx

Abstract. Automated monitoring of Information Technology resources allows for the treatment of issues relating to availability, capacity, and other quality requirements. Currently, the use of monitoring systems in immutable infrastructures requires manually updating configurations every time a new server is launched, which is often time consuming and error prone. In this work, we propose a process to automate the configuration of monitoring systems in an immutable infrastructure. The process works for monitoring daemon services (Although Monit has the ability to monitor many other aspects of operating systems, this article only exposes the automation of the configuration of services or processes.) running in Debian-based operating systems and involves the use of technologies such as *Ansible*, *Monit*, and *Slack*. In contrast to manually updating configurations, the main advantages of the proposed method are: a reduction in time and user-friendliness when configuring the monitoring system.

Keywords: Immutable infrastructure · Configuration management
Service monitoring · *Ansible* · *Monit* · *Slack*

1 Introduction

Cloud computing allows users to access a variety of Information Technology (IT) resources through the network [1, 2]. Cloud computing has been adopted by many organizations because it is an easier, faster and cost-efficient alternative to deploying and running software systems without the need to purchase the underlying IT resources. Cloud computing services come in different forms, such as *Infrastructure as a Service* (IaaS), *Platform as a Service* (PaaS) and *Software as a Service* (SaaS) [3]. These forms of cloud computing allow users to treat IT resources as intangible and flexible, rather than physical and rigid [4].

An *immutable infrastructure* is a paradigm in which servers are never modified after they are deployed [5]. In contrast to traditional infrastructure paradigms, where servers are modified *in situ* by upgrading and downgrading packages, modifying configuration files, and deploying new code, immutable infrastructure entirely replaces outdated servers with new ones, rather than updating, or "mutating" them. The need for multiple mutations often results in complex configurations that make tasks such as

J. Mejia et al. (Eds.): CIMPS 2018, AISC 865, pp. 225–235, 2019.
https://doi.org/10.1007/978-3-030-01171-0_21

reproducing, replacing, scaling and recovering servers difficult [6]. In an immutable infrastructure, if servers need to be updated, fixed or modified, it is often easier, faster and more cost-efficient to create new servers with the required configurations from scratch. After validating them, the new servers are ready to use and the old ones are decommissioned. The main benefits of using an immutable infrastructure are the simplicity of implementation; reliability; stability; consistency; efficiency; and coherence. Together, these minimize the many points of conflict and failure of mutable infrastructures [6].

Service monitoring provides information, either in real time or for a specified time frame, which allows informed decision-making to prevent or correct failures and helps ensure that services provide their stated quality attributes, e.g. availability and capacity [1].

Today, the use of monitoring systems in immutable infrastructures often requires manually updating configurations every time a new server is launched. In Unix-based systems, the integration of a monitoring system into an immutable infrastructure is still under development and is not fully standardized. Consequently, such integration is not fully implemented or automated. Furthermore, updating configurations manually is often error prone and time consuming.

The main objective of this work is to propose a process for automating the configuration of monitoring systems in an immutable infrastructure, specifically, the process that automates the configuration of daemon services, via *Monit*,[1] that run on Debian-based operating systems and involve the use of technologies such as *Ansible*, *Monit* and *Slack*, as well as preconfigured and auto-generated configuration templates. In addition to contributing to the field of configuration management, and in contrast to manually updating configurations, the main advantages of the proposed method are: a reduction in time and user-friendliness when configuring the monitoring system.

This article is organized as follows; Sect. 2 explains the background concepts and technologies involved to in automatically supporting them. Next, Sect. 3 describes provides the details of how the background concepts and technologies are used in the proposed process. Section 4 presents a discussion and evaluation of this work. Finally, Sect. 5 includes a summary and outlines some avenues of future investigation.

2 Background

In this section, the background concepts of this work are explained, specifically, configuration management (CM) and service monitoring. We also provide some examples of technologies to automatically support them.

[1] Although Monit has the ability to monitor many other aspects of operating systems, this article only exposes the automation of the configuration of daemon services.

2.1 Configuration Management

Broadly, CM refers to the process of systematically handling changes to a system in a way that ensures the system maintains its integrity over time. Although this concept did not originate in the IT industry, it is now broadly used to refer to server CM.

Automation plays an essential role in server CM. It is the ideal mean to make a server reach a desirable state, previously not defined at all or defined by provisioning scripts written in a specific language and/or features of a CM tool. For servers, automation is, in fact, the heart of CM. There are many CM tools available on the market, each one with a specific set of scripting languages, features and different complexity levels. Popular tools include *Chef* [7], *Puppet* [8] and *Ansible* [9].

In this work, we use *Ansible* to handle CM. *Ansible* is an open source IT engine that automates CM, application deployment, cloud provisioning, intra-service orchestration, and many other IT tasks [10]. *Ansible* uses management and configuration scripts, called playbooks, to specify automation jobs on remote machines. Playbooks are written in a very simple scripting language called YAML [11], which allows users to describe them in a way that approaches plain English. *Ansible* uses a series of modules that can be run directly on remote hosts or through playbooks. These modules can control system resources, such as services, packages, and files, or manage the commands of the execution system [12].

2.2 Monitoring Systems

Service monitoring provides information, either in real time or for a specified time frame, which allows informed decision-making to prevent or correct failures and helps ensure that services provide their stated quality attributes, e.g. availability and capacity [1].

For CM, automation plays an essential role in service monitoring. There are a variety of monitoring system tools such as *Nagios* [13], *Icinga* [14], and *Monit* [15].

In this work, *Monit* has been selected to support service monitoring. *Monit* is a utility for managing and monitoring processes, programs, files, file systems, and directories on Unix-based systems [16]. *Monit* conducts automatic maintenance and repair and can execute meaningful causal actions in error situations, e.g., it can start a process if it does not run, restart a process if it does not respond, or stop a process if it uses too many resources.

3 The Proposed Process

In order to support the automated configuration of a monitoring system, we propose a three-step process that uses *Ansible*, *Monit*, and *Slack* to support CM, Monitoring, and Alerts respectively. Each will be performed by specific (sub)systems (Fig. 1). For simplicity, from now on daemon services will be referred simply as "services."

The implementation code for the process depicted in Fig. 1 can be downloaded from https://github.com/cracos/ansible-monit-slack. In the following sections, the details of this process are explained.

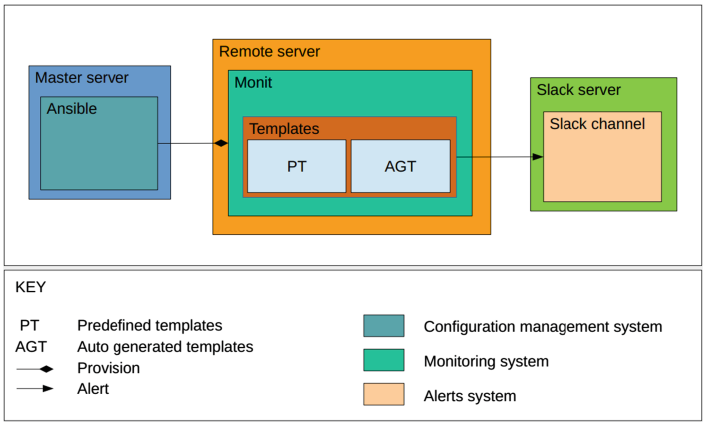

Fig. 1. Main elements of the proposed process to configure monitoring systems

3.1 Configuring a Monitoring System with *Ansible*

The first step in the proposed approach requires performing the CM of the monitoring system via *Ansible*, which is done using a variety of lists that contain information about the services to be (or not to be) monitored. In this work, two main types of lists are recognized: (i) user-provided lists; and (ii) auto-generated lists. Based on the information in these lists, *Ansible* will start to select or generate the corresponding configuration files. To reduce the probability of errors in configuration files, specific set operations are applied to the elements of these lists.

(i) User-provided lists:

- *Blacklist* (BL): *Ansible* configuration file where the user specifies the services that they do not wish to monitor.
- *Whitelist* (WL): *Ansible* configuration file where the user specifies the services that they do wish to monitor.

These lists can be generated in two main ways:

(a) Exclusion approach: requires the user to provide the BL. It is assumed that *Monit* will automatically monitor services that are running, except for those on the blacklist.
(b) Selective approach: requires the user to provide both the BL and WL.

(ii) Auto-generated lists using specific scripts:

- *List of predefined templates* (PT): generated from default CM information, i.e. port numbers.
- *List of services in execution* (SE): generated from active services that are running in the system.
- *Exclusion list by predefined templates* (ELPT): generated by performing the following operation (SE $\cap$ PT) $-$ BL, as shown in Fig. 2.

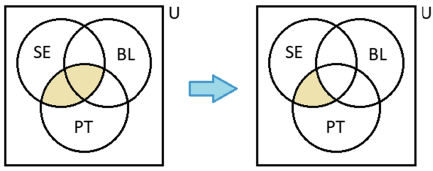

Fig. 2. Generation of the exclusion list using predefined templates

- *Exclusion list using basic templates (ELBT)*: generated by performing the following operation (SE − PT) − BL, as shown in Fig. 3.

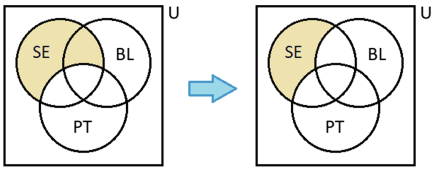

Fig. 3. Generation of the exclusion list using basic templates

- *Selective list using predefined templates (SLPT):* generated by performing the following operation {[(WL ∩ PT) ∩ SE] − BL}, as shown in Fig. 4.

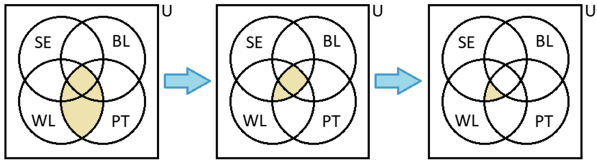

Fig. 4. Generation of the selective list using predefined templates

- *Selective list using basic templates (SLBT):* generated by performing the following operation {[(WL − PT) ∩ SE] − BL}, as shown in Fig. 5.

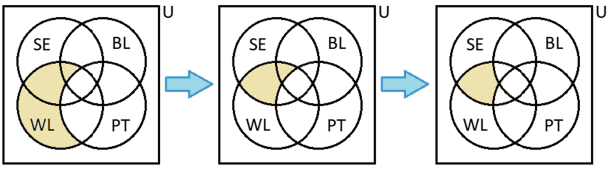

Fig. 5. Generation of the selective list using basic templates

3.2 Configuration of Services for *Monit*

The second step of the proposed approach is to perform the configuration in the Monitoring System, taking into consideration the configuration specified in *Ansible*. Service monitoring is performed using *Monit*.

Monit is configured by writing directives, a.k.a. rules, in a set of configuration files. The main configuration file is called *monitrc,* which consists of declarations of global definitions. In Debian-based systems this file is located in the `/etc/monit/` directory. *Monit* can use process-specific configuration files. These files are available, but not enabled, in the `/etc/monit/conf-available/` directory. To enable the desired configuration, symbolic links, a.k.a. symlinks, are created from the `/etc/monit/conf-enabled/` directory to the `conf-available` directory. This causes the configurations of the services to be monitored, added and loaded the next time *Monit* is started.

Predefined templates are stored in the `conf-available/preconfigured_templates/`[2] directory, while auto-generated templates are stored in `conf-available/autogenerated_templates/`. Auto-generated templates are stored for services that do not have a predefined template; when it is necessary to enable a service, only the corresponding symlink is created in the directory `conf-available`.

Configuration files can be structured in two different ways. The first uses the ".pid" file of the service to be monitored, which is a file that contains the process identification number (PID[3]). This is illustrated by the following example:

```
check process <unique name> with pidfile <path/to/pidfile.pid>
```

`<unique name>` is the unique name of the service to be monitored, and `<path/to/pidfile.pid>` is the absolute path where the service's .pid file is located. This form of configuration is recommended, as it defines the exact PID of the service of interest.

The alternative, which is used when the service to be monitored does not have a PID file, requires *Monit* to perform a process search using a regular expression or search pattern to find the process to be monitored. The following is an example:

```
check process <unique name> matching <regex>
```

`<regex>` is a regular expression or search pattern specifying the name of the service to be monitored. This form of configuration is only useful if the name of the process is unique. This alternative is used for generating basic templates. Each process has methods to start, stop, or restart its service that are used by *Monit* to execute an action on that service. For example:

---

[2] Templates are stored in a subdirectory to allow better organization.

[3] PID is an identification number that is automatically assigned to each process when it is created on Unix-like operating systems. Generally, it is stored in a ".pid" file, which allows other programs to find the PID of a running script.

```
check process <unique name> with pidfile <path/to/pidfile.pid>
start program = "/bin/xyz start"
stop program = "/bin/xyz stop"
restart program = "/bin/xyz restart"
```

Proactive monitoring can anticipate abnormal service events and take immediate action to prevent major incidents on the server. For example, if a server is consuming a lot of resources, *Monit* can stop or restart the server and send a notification. A simple example of how to tell *Monit* what to do, depending on the type of abnormal situation, is shown below:

```
check process <unique name> with pidfile <path/to/pidfile.pid>
start program = "/bin/xyz start"
stop program = "/bin/xyz stop"
if cpu > 80% for 4 cycles then alert
if cpu > 80% for 4 cycles then restart
```

In this example, if the process is consuming 80% or more processor resources for four *Monit* review cycles,[4] it will send an alert message and restart the process. Using this approach, a variety of checks can be enabled, such as detecting if a process is not running or detecting if the start of a service fails.

It is not always possible to have a predefined configuration template or enough information to generate it. For these situations, auto-generation of basic configuration templates was implemented. These templates contain information generated from a series of checks that are common for services. The checks are used to enable proactive monitoring of a service.

The logic of auto-generation of basic configuration templates is depicted in Fig. 6. First, the data of the services in the system is obtained based on the elements of the exclusion or selective lists, as seen in Fig. 6(a). Then, using the *Ansible* module "*Systemd*", information about running services is obtained, as in Fig. 6(b). From this information, the Control Group (Cgroup[5]) of the processes of interest is extracted. Then, from the Cgroup, the PID of the process is retrieved, shown in Fig. 6(c). Finally, using the `ps`[6] command, the information on how the process was launched is obtained, as in Fig. 6(d). This information is used to generate the search pattern that *Monit* will use to avoid the possibility of generating erroneous configurations with unwanted processes.

[4] By default, when *Monit* completes a programmed job, it goes to sleep for a configured period, then it wakes up and start monitoring again in an endless loop.

[5] Cgroup is a hierarchical grouping of processes managed by the OS kernel, and exposed through a special file system.

[6] The ps command is provides information about the processes that are running in the system.

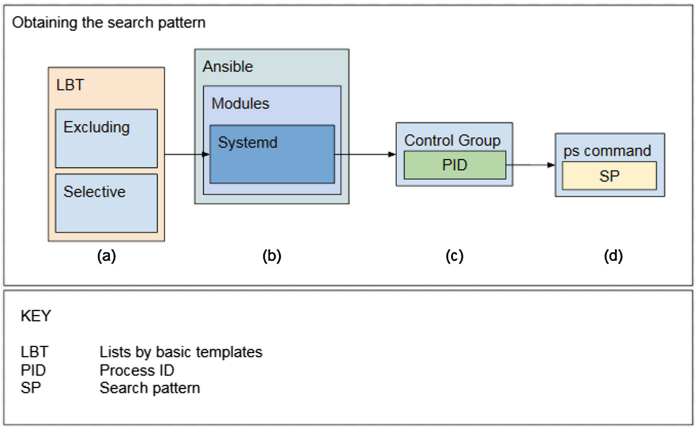

Fig. 6. The logic of auto-generation of basic configuration templates.

3.3 Notifying Alerts via *Slack*

The third step of the proposed approach requires defining an Alert System. By default, when something abnormal happens, *Monit* raises automatic alerts that are sent to the user through pre-configured emails. However, *Monit* can also be configured to send notifications to almost any messaging service [17]. In the method described in this article, *Slack* is used as a messaging service because it allows all notifications to be centralized and takes action for specific messages [18]. The configuration to generate alert notifications through *Slack* was included in each configuration template for *Monit* services, either preconfigured or self-generated, as described below.

Configuring *Slack* to send alerts requires using the `exec` command in the *Monit* configuration files. The following example is a variant of one presented above:

```
check process <unique name> with pidfile <path/to/pidfile>
start program = "/bin/xyz start"
stop program = "/bin/xyz stop"
if cpu > 80% for 4 cycles then exec "/etc/monit/SlackAlert.sh"
```

Here, the configuration specifies that whenever the processor goes over 80% usage, a notification will be sent to a *Slack* channel. *Incoming WebHooks*[7] are used in this work. A Webhook provides a unique URL to which the message text is sent [19]. Using this method, *Monit* alerts can be received in a specific *Slack* chat room.

3.4 Generating Logs

When the monitoring system is deployed successfully, a log file is generated containing a summary of all the services and how they were configured. The log file can be found in the path `/etc/monit/rs.log`.

[7] They are a simple way to post messages from applications in Slack.

4 Discussion and Evaluation

There are four main aspects from which we discuss and evaluate the proposed approach: (i) CM tools; (ii) Monitoring tools; (iii) Messaging tools and (iv) Time and error reduction when configuring the monitoring system.

4.1 CM Tools

As mentioned previously, there are many CM tools available in the market: popular choices include *Chef* [7], *Puppet* [8], and *Ansible* [9]. We chose *Ansible* as our CM because it compatible with, and provides an easy way to configure, *Monit*. *Ansible* CM playbook syntax is built on top of YAML, which is a scripting language designed to be easy for humans to read and write. That is, it promotes user-friendliness. Also *Ansible* is agentless, which means there is no need to pre-install an agent or any other software on hosts (i.e. remote servers). It requires a server to have the SSH utility and Python 2.5 or later installed [20].

4.2 Service Monitoring Tools

Today, widely-used tools for monitoring services include *Nagios* [13], *Icinga* [14], and *Monit* [15]. A real benefit of *Monit* is its easy configuration and syntax. While developers have to dig through a bunch of files in order to create a simple check in *Nagios*, *Monit* allows them to simply use one (human-readable) configuration file in the correct path for it o work. *Nagios* and *Icinga* are not user-friendly to new users and involve complicated configurations because their installation and configuration require extensive knowledge of the underlying operating system.

4.3 Messaging Tools

The most important reason people use the *Slack* instant messaging tool is that it can be easily integrated with tools such as *Trello* [21], *GitHub* [22], *Dropbox* [23], *MailChimp* [24], and many more. Furthermore, *Slack* allows centralized events integrated into chat rooms.

4.4 Time Reduction

As we will discuss in Sect. 5, we are planning to perform a systematic study to quantitatively evaluate the reduction in time and error of the proposed approach. However, we have performed some experiments and the results seem very promising. We performed the described process to deploy instances of the monitoring system dealing with one to twenty two services and it took from 9 to 28 min to do it, respectively. When common changes to the existing configurations of these instances were performed (e.g. changes in port numbers, services' files dependencies, checks) it took from 9 to 13 min when dealing with one to twenty two services.

4.5 Other Thoughts

There is a repository on *GitHub* (https://github.com/pgolm/ansible-role-monit) with an *Ansible-Monit* configuration system. This system is similar to the one proposed in this paper but does not implement the auto-configuration of services and integration of alert notifications with instant messaging tools as ours does.

5 Summary and Future Work

The use of monitoring systems in immutable infrastructures sometimes requires manually updating configurations every time a new server is launched, which is often time consuming and error prone. In this paper, we described a process to automate the configuration of monitoring systems in an immutable infrastructure. The process works for monitoring daemon services running in Debian-based operating systems and involves the use of technologies such as *Ansible, Monit*, and *Slack*.

Future work in this area includes performing a systematic study to validate the benefits of the process presented in this paper. The study will be performed in two IT companies in the state of Zacatecas, Mexico: Sharing Economy Tools, which is a Mexican company that offers software development services, and Tinkerware, which is a Mexican company that provides infrastructure automation solutions and services to software development companies.

In this effort we worked with daemon services in Debian-based operating systems. We plan to explore the feasibility to automate monitoring of services that are not daemon. We plan also explore the use of these proposed approach with non-Debian-based operating systems. In addition, we are considering incorporating more of the monitoring options offered by *Monit* into our process, such as *host*, or *filesystems*.

Acknowledgments. The authors would like to extend thanks to Tinkerware and Sharing Economy Tools, and especially to Alfonso Álvarez Sánchez and Agustín Rumayor Barraza, members of each of these companies, who provided insight and expertise that greatly assisted the process presented in this paper.

References

1. Fatema, K., Emeakaroha, V.C., Healy, P.D., Morrison, J.P., Lynn, T.: A survey of cloud monitoring tools: taxonomy, capabilities and objectives. J. Parallel Distrib. Comput. **74**(10), 2918–2933 (2014)
2. Youseff, L., Butrico, M., Da Silva, D.: Toward a unified ontology of cloud computing. In: Grid Computing Environments Workshop, GCE 2008 (2008)
3. Syed, H.J., Gani, A., Ahmad, R.W., Khan, M.K., Ahmed, A.I.A.: Cloud monitoring: a review, taxonomy, and open research issues. J. Netw. Comput. Appl. **98**, 11–26 (2017)
4. Fowler, M.: The disposable infrastructure // Speaker deck (2017). https://speakerdeck.com/mlfowler/the-disposable-infrastructure. Accessed 25 Nov 2017
5. Stella, J.: An introduction to immutable infrastructure - O'Reilly media (2015). https://www.oreilly.com/ideas/an-introduction-to-immutable-infrastructure. Accessed 18 June 2018

6. Virdó, H., DigitalOcean: Immutable infrastructure | DigitalOcean (2017). https://www.digitalocean.com/community/tutorials/what-is-immutable-infrastructure. Accessed 12 Apr 2018
7. Chef Software Inc.: Chef - Automate IT infrastructure (2018). https://www.chef.io/chef/. Accessed 20 June 2018
8. Puppet Enterprise: Deliver better software, faster (2018). https://puppet.com/. Accessed 20 June 2018
9. Red Hat Inc.: Ansible (2018). https://www.ansible.com/. Accessed 26 Nov 2017
10. Red Hat Inc.: How Ansible works | Ansible.com (2018). https://www.ansible.com/overview/how-ansible-works. Accessed 20 Mar 2018
11. YAML: The official YAML Web Site (2018). http://yaml.org/. Accessed 20 June 2018
12. Red Hat Inc.: Working with modules—Ansible documentation (2018). https://docs.ansible.com/ansible/latest/user_guide/modules.html. Accessed 17 June 2018
13. Nagios Enterprises LLC.: Nagios - Network, server and log monitoring software (2018). https://www.nagios.com/. Accessed 20 June 2018
14. Icinga Open Source Monitoring: Icinga – Open source monitoring (2018). https://www.icinga.com/. Accessed 20 June 2018
15. Tildeslash Ltd.: Monit (2017). https://mmonit.com/monit/. Accessed 16 Mar 2017
16. Tildeslash Ltd.: Monit manual (2018). https://mmonit.com/monit/documentation/monit.html. Accessed 20 Mar 2018
17. Tildeslash Ltd.: M/Monit | Slack Notification (2018). https://mmonit.com/wiki/MMonit/SlackNotification. Accessed 17 June 2018
18. Slack Technologies: What is slack? – Slack help center (2018). https://get.slack.help/hc/en-us/articles/115004071768-What-is-Slack. Accessed 20 Mar 2018
19. Slack Technologies: Incoming webhooks (2018). https://api.slack.com/incoming-webhooks. Accessed 17 June 2018
20. Hochstein, R.M.L.: Ansible: Up and Running, 2nd edn. O'Reilly, Newton (2017)
21. Trello. https://trello.com/. Accessed 20 June 2018
22. GitHub. https://github.com/. Accessed 20 June 2018
23. Dropbox. https://www.dropbox.com/. Accessed 20 June 2018
24. MailChimp: Marketing platform for small businesses (2018). https://mailchimp.com/. Accessed 20 June 2018