Automated tests for infrastructure code

Amar Hodzic
amarh@kth.se

Niklas Wessman
nwessman@kth.se

April 2021

Contents

1 Introduction

A common DevOps practice is to use *Infrastructure-as-Code* to manage IT infrastructure. This principle has the benefit of consistency. The code produces the same infrastructure each time, in contrast to earlier days when infrastructure needed to be set up manually, which made it time-consuming and hard to achieve the same results each time. Infrastructure code, like all types of code, needs testing. We want to know that the code does what it should and does not introduce problems into our infrastructure. Thus, creating the need for good testing practices. Infrastructure-as-Code has some special aspects that need to be taken into considerations when creating tests. This essay aims to introduce the concepts of software testing and explain how they can be applied to infrastructure code.

2 Background

Infrastructure, in the context of DevOps, usually refers to the services that cloud providers offer. These services range from cloud-based platforms, applications to storage. The appeal of using cloud providers, instead of investing in the hardware yourself, is the predominantly used pay-per-use model and the possibility to scale and provision resources as needed. [1]

Traditionally, the provisioning of infrastructure was done manually by the operations team which is slow, time-consuming and prone to errors [2]. As the paradigm has shifted to faster delivery of code releases, so too has the demand on the production systems to be stable and updated at all times. As a consequence of the effort to speed up and streamline the handling of infrastructure, organizations started to adapt Infrastructure-as-code (IaC) to overcome the development-operations barrier [3].

2.1 What is Infrastructure as Code?

Infrastructure-as-Code promotes the managing of knowledge and experience, of infrastructure provisioning, in reusable infrastructure code scripts [2]. It allows DevOps teams to test applications early in the development cycle in production-like environments which prevents common issues during deployment [4]. IaC is a way of automating infrastructure with practices based on software development and a focus on consistency, reusability and transparency [5].

Consistency

By building from code, consistency is achieved each time it's built. This in turn allows for predictability in system behaviour and makes testing more dependable, which enables continuous delivery.

Reusability

When things are defined in code, it becomes easy to scale up and create multiple instances of it. If problems occur or fixes are needed, the code can be updated and all instances redeployed.

Transparency

By encoding expertise and knowledge in code, everyone can read it, learn and suggest improvements. It can also provide building blocks for other implementations and solutions.

2.2 Why should it be tested?

Infrastructure code, like any other type of code, is prone to errors. These errors can range from syntactical or logical to run-time errors among other types which can be discovered with tests. Testing can also reveal edge cases and their behaviour as well as the impact of changes. By testing the code, confidence in it grows which can empower developers to dare fix bugs and add new features [6].

2.3 Challenges with automated tests

With automated tests come new types of problems that could halt the process it aimed to streamline. If the organisation is not used to the approach of test automation, it will create the need to spend time and resources to learn what tools to use and how they should be implemented to fit the organisation's needs. If the software is not built from the start in a test-driven approach, introducing these concepts later in the development stages could lead to technical problems when trying to apply the automation tools to the project [7].

For tests to be useful, they need to be good. If the tests are poorly made, they will not contribute to the development practice and could instead create a false sense of security. With bad test practices, the tests may miss crucial use-cases. If passing the tests is the only metric for measuring code quality, it could lead to bad code being deployed to production [7].

3 Types of tests

There are different types of tests of infrastructure code that range in sophistication and effort. There is a trade-off between the types of tests used and the robustness of the system. On one hand, there is static analysis and unit testing which can give some confidence in the code and on the other hand there is end-to-end testing which gives strong confidence in the code. However, the type of test will also dictate the amount of money and time needed to be spent, as illustrated in figure 1. The pyramid provides a loose framework for the foundation when creating tests for a code-base. Meaning that static analysis and unit testing should probably stand for the bulk of the testing. The tests will generally also be more brittle and flaky higher up in the pyramid [8]. Flaky tests fail sporadically with a seemingly non-deterministic behaviour [9]. The bigger tests become and the more of the system they encompass, the bigger the chance of intermittent failures.

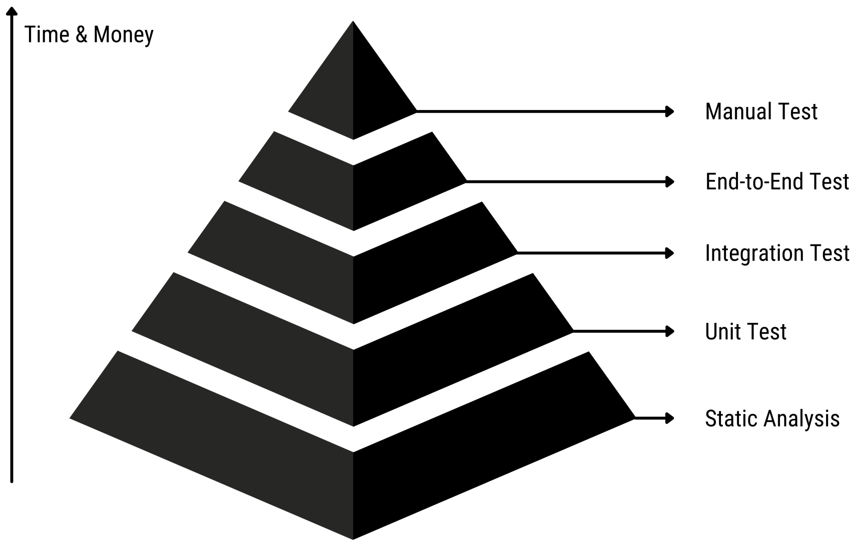

Figure 1: An overview of the different tests and their relation to time and money needed.

It is important to use a different environment than the production environment when doing this type of testing, so it does not interfere with the deployed product.

3.1 Static analysis

The first line of defence against errors is static analysis tools. These tools assess the quality of the code at a surface level. With the help of compilers, parsers and interpreters, syntactical and structural errors that could lead to the machine not being able to parse the code are discovered. This could for example be the use of an undeclared variable, missing bracket or indentation error.

When it comes to static analysis in infrastructure code, the point is to find these errors and inconsistencies without executing and deploying any infrastructure. There are several ways to accomplish this with popular tools and frameworks such as Terraform and Packer.

3.2 Unit test

A unit test is used to test individual functionalities by targeting isolated parts of a software, i.e. units. A unit in this case could be a function, object or procedure, in infrastructure code the equivalent is modules. These tests are used to assert the correctness of the code and make sure that each unit performs as expected [10]. Unit testing is important because it helps find bugs and errors in the code early in the development. By pinpointing individual units, problems are mitigated that could be hard to track down later in development when the bugs materialize in other parts of the programs connected to the unit.

The problem with unit testing in the context of IaC is that it is hard to test units without deploying the code since they usually require external resources or API calls [8]. Therefore, to test it we have to deploy it to a real environment and validate that it works there, and after we have asserted that it works as expected we then need to undeploy the infrastructure.

3.3 Integration test

Integration tests pick up from where unit testing left off and is instead the process of combining modules or programs to assert that they work correctly in the complete system [11]. There are different types of strategies that are commonly used to do integration testing:

- Big bang testing: Here all software and hardware elements are combined at once to create a complex system that is tested fully, in contrast to an approach of building the system in stages. [11]
- Sandwich testing: This is a combination of two other popular testing methods called Top-Down testing and Bottom-Up testing. It tests the system both from the top layer going down and from the bottom layer going up, finally meeting in the middle. One of the negative aspects of using this type of approach is that the combination of two different strategies makes it an expensive operation. [12]

The same steps of deploying, validating and undeploying used for unit testing are of course needed here as well. The difference is that we need to set up our environment so that we can deploy each module that is needed for the whole integration test and not only the individual parts. Equally important is then to clean up and undeploy when done.

As can be seen in Figure 1 above, integration testing can be expensive. The testing stage can often be conducted quickly but instead, it is the steps of deploying and undeploying that can be a costly procedure. To mitigate this problem we can take advantage of our testing environment by letting it run after we have deployed it and run multiple tests before we undeploy and clean it.

3.4 End-to-end test

End-to-end testing (E2E) is meant to test the whole system from start to finish with all the components that make it up. More specifically, the point is to create an environment similar to the production environment and test all actions and flows that will be performed in production. Although E2E testing gives a high level of confidence in the code, it is usually costly in both time and money. It is also inherently prone to flakiness due to external dependencies such as API calls and the number of resources used.

If we assume that a resource (e.g. Amazon EC2) has a probability of failure of 0.1% and the E2E test uses 50 such instances, then the chance of the test failing due to failure in one or more of the instances is 5%. Bigger systems can have hundreds of resources deployed for E2E testing which quickly leads to brittle test. [8]

Another approach is to do *incremental* E2E testing. With this approach, a persistent environment for E2E testing is deployed and left running. Then, whenever a part or module of the system is updated, only that part is redeployed and validated against the whole system. The benefit is that the whole system is deployed once and after each update to the code, only the affected resources are redeployed [8]. This leads to a lower number of resources being deployed across all iterations of the code which mitigates some of the flakiness caused by failures.

4 Conclusion

There are a lot of perks of using Infrastructure-as-Code, but it relies on it being well-integrated and tested. As has been explored in this essay, the fundamentals of software testing can be found in infrastructure code testing as well. The largest difference is that the infrastructure has to be deployed to be tested. One approach to mitigate some of the problems is to let the environment be deployed and run multiple tests before undeploying. For E2E testing, deploying once and only redeploying modules when they change will save time and reduce flakiness. By using smart testing practices, the full power of Infrastructure-as-Code can be achieved.

References

[1] "What is a cloud service provider?" [Online]. Available: https://azure.microsoft.com/en-us/overview/what-is-a-cloud-provider/

[2] M. Artac, T. Borovssak, E. Di Nitto, M. Guerriero, and D. A. Tamburri, "DevOps: Introducing Infrastructure-as-Code," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. Buenos Aires: IEEE, May 2017, pp. 497–498. [Online]. Available: http://ieeexplore.ieee.org/document/7965401/

[3] W. Hummer, F. Rosenberg, F. Oliveira, and T. Eilam, "Testing Idempotence for Infrastructure as Code," in *Middleware 2013*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 8275, pp. 368–388, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-45065-5_19

[4] "What is Infrastructure as Code?" [Online]. Available: https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code

[5] K. Morris, *Infrastructure as Code*, 2nd ed. S.l.: O'Reilly Media, 2021, oCLC: 1156171206.

[6] S. Foote, "Gaining Code Confidence Through Testing," Apr. 2015. [Online]. Available: https://www.informit.com/articles/article.aspx?p=2346351

[7] "Common Problems of Test Automation in Modern Days," Oct. 2018. [Online]. Available: https://testautomationresources.com/software-testing-basics/problems-test-automation/

[8] Y. Brikman, "Automated Testing for Terraform, Docker, Packer, Kubernetes, and More," Dec. 2019. [Online]. Available: https://www.infoq.com/presentations/automated-testing-terraform-docker-packer/

[9] "Flaky tests." [Online]. Available: https://docs.pytest.org/en/stable/flaky.html

[10] Guru99, "Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE." [Online]. Available: https://www.guru99.com/unit-testing-guide.html

[11] *ISO/IEC/IEEE 24765:2017 Systems and software engineering — Vocabulary*, 2nd ed. ISO/IEC JTC 1/SC 7 Software and systems engineering, Sep. 2017. [Online]. Available: https://www.iso.org/obp/ui/#iso:std:iso-iec-ieee:24765:ed-2:v1:en

[12] "Sandwich Testing," Mar. 2020. [Online]. Available: https://www.professionalqa.com/sandwich-testing