INFRASTRUCTURE AS A CODE:

Best Practices for DevOps Engineers

INNOVECS INNOVATE INSPIRE CARE

ABOUT AUTHOR

Ihor Kanivets, Advanced DevOps Engineer at Innovecs. He is experienced in Infrastructure as a Code. An automation process is something Ihor enjoys the most in his job. That's why he has chosen the DevOps engineering path. Creating a system that operates without human involvement is essential.

Throughout his carrier growth, Ihor has earned experience in the **following areas:** CI\CD (Jenkins pipelines, GitHub actions), Infrastructure as a Code (Terraform, Pulumni), Automation tools (CHEF, Ansible), Scripting (Bash, Python, Powershell), Cloud providers (AWS, Azure, GCP) and so on.

CONTENTS

INTRODUCTION

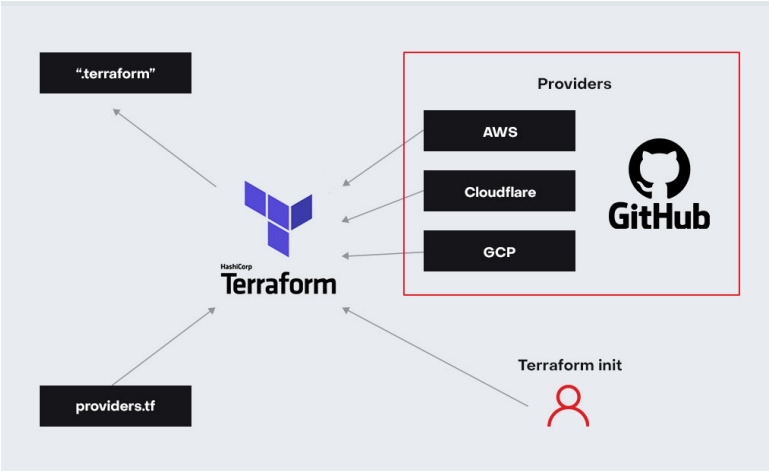
Today I'd like to share the best practices used in IAAC (Infrastructure as a code).

Some of them are known, and some may be quite new. In any case, this article will be useful for DevOps Engineers whose experience proves the necessity to cover the infrastructure with code; and beginners who are just starting to embrace Terraform.

IAAC stands for the infrastructure that is described by code. In the article, I offer you to take a look at IAAC with Terraform, which can work with both cloud and on-premise environments, but we will focus specifically on clouds.

You need to start working on the project with initialization, the terraform init command, which will load all the necessary providers, allowing you to work with various platforms (AWS, GCP, Azure, and CloudFlare just to name a few).

Figure 1



A provider is, to some extent, an instruction by which Terraform understands how to interact with a particular platform.

The major commands for working with Terraform are as follows:

1. terraform init (initializes the project and loads providers that are necessary for deployment in a given environment (Fig. 1));
2. terraform plan (allows you to see exactly what resources terraform wants to create or change or delete);
3. terraform apply\destroy (allows you to deploy\remove the resources you saw in the terraform plan step).

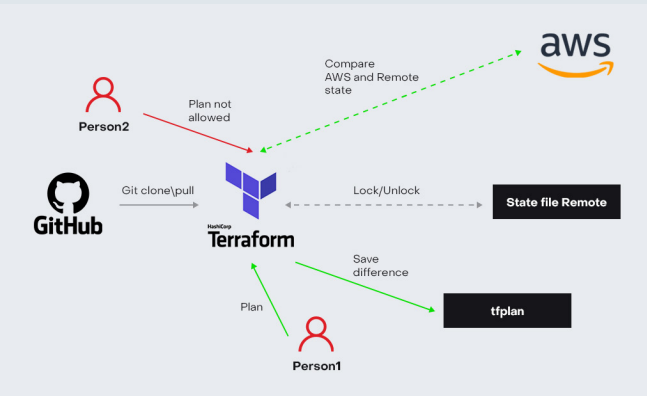
The convenience of Terraform is that the entire team can work on the project. It supports the lock mechanism. We will explore the lock mechanism scheme in the article below.

USE REMOTE STATE FILES AND THE LOCK MECHANISM

To save changes, you should use a remote state file containing the infrastructure's current state. Terraform also supports a local state file. However, in this case, you will not be able to work as a team on the same terraform project, because your teammates will not know what you are doing, and you will not know what they are doing. It also jeopardizes overwriting or deleting each other's resources. During the first run of the terraform plan command, we create a remote state file (having previously specified the path in terraform), reflecting the infrastructure's current state.

If two or more people are working on the project, one of whom is running Terraform plan\apply, then the other one trying to execute one of the specified commands at the same moment will receive a notification that somebody has been already working with this project, and must wait a bit until the deployment\plan ends and the lock is removed. The Lock mechanism allows you to prevent each other's resources from being consumed. You can find out more about the lock mechanism via the **link**.

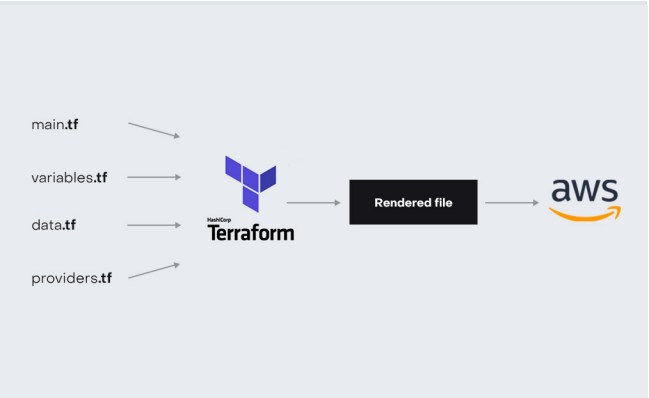
Figure 2



Sometimes it happens that you start doing terraform plan\apply and you face a connection issue. In this case, the state file remains locked because terraform plan\apply did not complete successfully and the state file was not unlocked. To unlock it, use the terraform force-unlock <lock-id> command if you locked the state file. And you didn't, wait for the deployment to finish. If the state file remains locked for a long time, ask your teammate whether you can unlock the state file (the terraform message will reveal the person who locked the state file).

Separate terraform files based on the logical resources distribution:

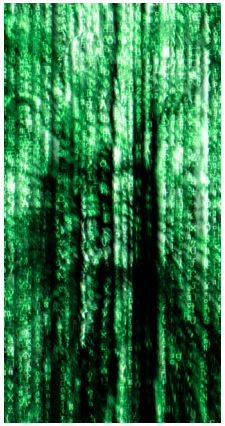
Figure 3



Collect all variables in one variables.tf file, providers in providers.tf file, data resources in data.tf file, and use them in main.tf file. Thanks to this, teammates can easily understand the terraform code, because everyone will be on the same page and quickly figure out where the code needs to be changed to change the infrastructure in one direction or another.

Terraform does not require the logical distribution of files. Whether there are 20 files with the .tf extension, or 1 .tf extension file with a thousand lines of code — at the terraform plan stage, everything is collected into a single file, which is later used for deployment. It is much easier to review 10 files of 15 lines each than one file of 150 lines, so I recommend splitting the files.

USE VARIABLES INSTEAD OF HARDCODING



If a value is used more than once in the code, it makes sense to use variables. This way you manage only one place where you specify the variable.

Picture a situation where we hardcoded a certain value into thousands of code lines. If the value change is necessary, it must be rewritten in all places where we hardcoded it. We can specify this value once in the variables .tf file to avoid this long process: use a variable instead of hardcode, and voila! By adding a variable in one place in the variables.tf file, we will change it in thousands of places.

For example, if there are three variables: vpc_id, vpc_cidr, subnet_group, let's combine them into one object variable network, which will have three attributes (vpc_id, vpc_cidr, subnet_group). Why do we need this? During terraform deployment, we use API calls under the hood. Each individual API call is a certain amount of time spent on processing. The more API calls, the more time for terraform plan\deploy. And everything seems to be fine if it is about deploying one instance to the cloud. We won't notice much difference whether we have one variable with three attributes or three different variables. Imagine that in a project with 1000 resources, the time to plan and apply terraform increases if we use three separate objects instead of one object variable. And in the code, it looks better when we combine variables into logical objects.

Hence, we offload work with Terraform and reduce response time. We can refer to the object variable as follows: var.network.vpc_id, var.network.vpc_cidr, var.network.subnet_group.

My advice is to combine variables into logical chains

Naming a variable is a crucial point in the work.

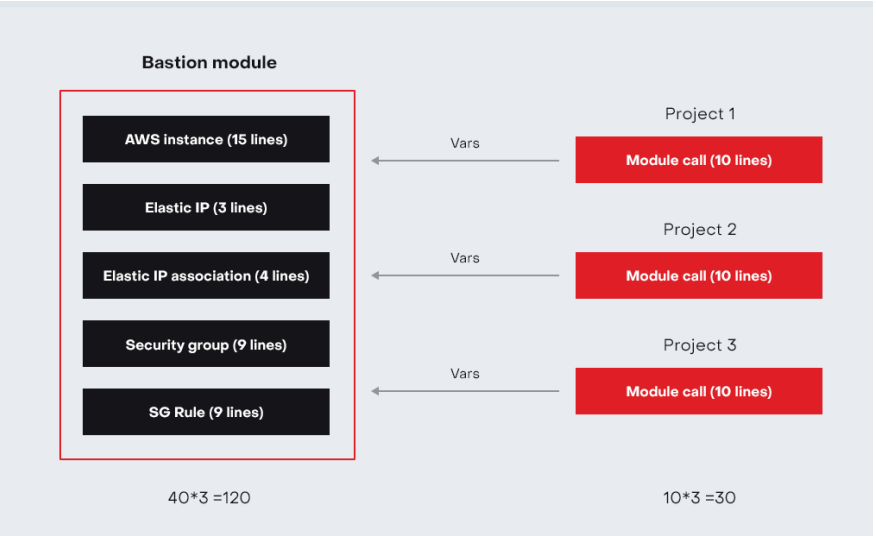
Naming a variable is a crucial point in the work. This is especially true if you've written code that the whole team will work with. Spending 5 minutes on a variable name could save an hour of team time in the future.

USE MODULES

A module is a logical grouping of resources used in different projects. Usually, the module resources are identical for different projects, only the variable values differ.

By using modules, we avoid code duplication. In case we need to create 15 identical instances (which require security_group, security_group_rules, public_ip, etc.) in different projects, we can write one module and call it 15 times. Instead of 400 lines of code, we will have only 15. Thus, we will create 15 different resources, but with the help of one module.

Figure 4



Code is easier to manage thanks to modularity.

Let's assume we have created a bastion module used in 15 projects. For example, a new office was opened in the company, and we need to add another IP address to bastion security_group. Without the module, we would need to add a new IP address to each bastion in 15 projects. Every manual change to the code is a potential human error. But since we use a module, it will be enough to add a new IP address in one place (this will help avoid the human factor mentioned earlier and reduce the time to adjust the variable in one place instead of 15), deploy the code, and that's it. Such are the merits of modularity.

USE THE 'COUNT' FUNCTION TO CREATE IDENTICAL RESOURCES OR TO RESOLVE AN 'IF CASE' FOR A RESOURCE

The ‘count’ function can help us both to create identical resources with different names and to resolve an ‘if condition’ within a resource.

You can learn about the purpose of the count function in the official terraform documentation, and below I would like to talk about how the ‘count’ function can help make our module more flexible or a resource that requires an if condition based on certain data. For example, there are two DNS providers — Cloudflare and Route53 in Amazon. Different systems use different providers. But we have a module that creates a web server and accompanying instance resources (security group, security_group_rule, public_ip, and others).

If we don’t use the count function with the if condition, we just need to duplicate the code and create two modules. In the first case, it will be a module with a Cloudflare entry, and in the second — a module with a Route53 entry. But duplicating an entire module that has 500 lines of common code and only 10 lines that differ is not the best practice.

To address this situation, we can use the count function as intended. Thus, when calling the module, we indicate that the provider is equal to Cloudflare. If so, the function will return 1 to us and thus create a Cloudflare record in Cloudflare. Next, we go to the next Route53 resource and check with the count. Accordingly, the condition will be 0, and we will not create an entry in Route53. If we specify the Route53 provider, then the opposite will happen: unlike the first situation, a Route53 dns record will be created, and Cloudflare will not be. This is an essential option inside Terraform that helps make the module flexible.

Figure 5

```
resource “cloudflare_record” “app” {
  count = var.provider == “cloudflare” ? 1 : 0

  zone_id = var.dns.hosted_zone
  name    = var.network_config.hosts[count.index]
  value   = var.alb.endpoint
  type    = var.cloudflare_type
  proxied = false
}

resource “aws_route53_record” “app” {
  count = var.provider == “route53” ? 1 : 0

  zone_id = var.dns.hosted_zone
  name    = var.network_config.hosts[count.index]
  type    = “CNAME”
  ttl     = 300
  records = [var.alb.endpoint]
}
```

DATA RESOURCES OF EXISTING RESOURCES

Consider the situation when **terraform** creates a new resource, and you need a certain attribute of the resource you created manually. For example, you want to create a security_group inbound rule for an existing instance.

Figure 6

Here you need the IP address of the instance.
You have several options:

1. Hardcode (this is not a best practice and I don't recommend it, but this option still exists).

2. Use a data_resource that will allow you to get the IP address of the instance (this is the best practice I highly recommend).

Why is hardcoding bad and using data_resource good?

Let's imagine the case where the IP address of the instance is changed for some reason. If we use data_resource, then every time terraform plan is started, the instance information is updated, and terraform offers to replace the old IP with a new one. If we're using hardcode, then terraform will know nothing about it until you make changes to the code. And if you have 100 such changes in the project, it will create more work and take a lot of time. To use data_resource, you must specify the ID of the resource whose attribute you want to get. The ID is different for each resource, therefore, make sure to check the terraform documentation. This approach is effective when part of the infrastructure is not yet described by code, and part is in the process of being covered.

If the infrastructure is already described by the code, then instead of making an API call to AWS (data_resource), we can use the data resource to the state file (terraform_remote_state), which is stored in the s3 bucket. To do this, we make a request and get all the resources to create the current infrastructure.

Figure 7

DATA RESOURCES OF EXISTING RESOURCES

To sum up, we have two options:

1. a more complicated one, when we make 15 API calls in AWS, collect all the necessary information and thus continue to create the infrastructure (this is a working option in a case where we want to get certain attributes of resources that are not yet covered by terraform);

2. simpler and more convenient — we make one API call to the s3 bucket and get these 15 resources (terraform_remote_state) that we need (infrastructure covered by terraform). In this context, it is important to understand that the project's state file must contain outputs for the attributes we need. In this way, we offload Terraform flow, gently increasing performance and reducing response time. Respectively, if we have 700 such resources (data_resource), the time to obtain them multiplies. If we use data_remote_state, and have all 700 outputs, in this case, instead of 700 API calls, we make just one.

Overall, if you have the opportunity to use remote_state_ file from another project, do it and unload terraform. If not, use data_resource, but don't hardcode the value.

WHEN USING CODE INFRASTRUCTURE AVOID MAKING MANUAL CHANGES, BUT EVEN IF YOU DO — MARK IT IN TERRAFORM

If you have already started using Infrastructure as a code, stay away from manual changes.

Terraform often helps to bring the infrastructure back to working condition. Let's say someone accidentally made a mistake and your site won't open. To understand the reason for it, you need to do debugging, which is time-consuming. Instead, you can make a terraform plan that will show you the changes made before that. In this case, the terraform app will tackle your issue in minutes.

But when production is down and terraform plan\apply didn't bring the infrastructure back to a working state, don't waste time fixing the terraform code: better solve the problem manually first, and later specify everything in terraform in calm mode. This will reduce your downtime, which in turn will decrease your losses. If the changes are not related to a production issue, then everything must be done through the Terraform code.

When changes are made manually, the situation looks like this: one made tag changes in the instances, and the other changed the database retention period to 4 or 8 days. When we face a production issue, instead of one change, which is the root cause, we will see that terraform wants to make 4 changes in different places. In this case, instead of one resource, you need to review 4, and this will all take time. And if you changed not 4 resources, but 20, then this is even longer. Of course, you have to fight with manual changes. Although I also face this in my work, it eats up my time and the time of my colleagues. Therefore, it is more appropriate to cover the infrastructure with code and register everything in Terraform.

USE IMPORT RESOURCES IF THE PROJECT IS NOT PART OF TERRAFORM AND YOU WOULD LIKE TO COVER IT WITH CODE

If your project or a certain part of it is completely covered by terraform, it's just great. But there are occasions when resources are partly created by terraform, and partly by hand. If we need to cover all terraform resources, the terraform import command will help in this case. Terraform has a detailed description of how to import certain resources. Terraform runs the import command and thus takes over the infrastructure management. The import algorithm is the following:

1. Write terraform code for the resource you will import.
2. Run the terraform plan command (the command should show you that the resource you want to import will be created).
3. Import the resource using the terraform import resource_id command (the resource_id is different for each resource, check the terraform documentation).
4. Run the terraform plan command again (if the import was done correctly, terraform should display the message: 'The infrastructure is up to date').

In covering the infrastructure with code, you implement terraform step by step.

Accordingly, next time all changes can be made using Terraform, and not manually.

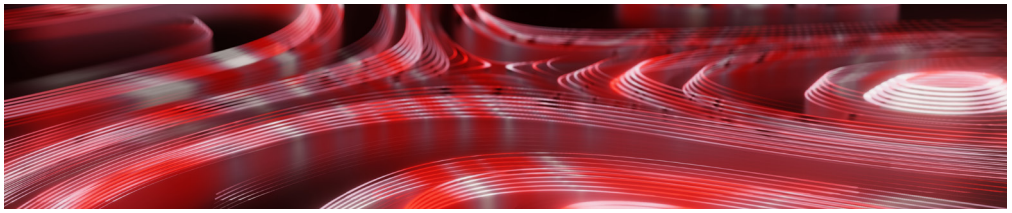
The import command can be used when we have two or three projects — Backend, Frontend, and DevOps — that are already covered by terraform. Yesterday it might belong to Backend whereas today it may belong to DevOps. Therefore, we need to import from Backend to DevOps. We should remove this resource from the Backend (terraform state remove); thus, we can manage Terraform both here and there. It is imperative that one resource is managed by only one terraform state file; otherwise there will be constant overwriting.

Suppose your resource is managed using two terraform state files DevOps and Backend, and default_tags are used in each project. Every time during terraform apply, tags will be recorded for the DevOps project and the Backend. To avoid double work, we store the resource in only one Terraform state file.

Before working with **remote tf.state**, we need to give our team a heads-up so that no one touches state file until you are done working with it.

After finishing work with **remote tf.state** inform teammates that you're done and others can work on the file. A happy ending of resources import is the result of the terraform plan command "infrastructure is up to date".

MAKE SURE THERE ARE NO ERRORS IN TERRAFORM FMT AND TERRAFORM VALIDATE SYNTAX



When we write Terraform code, it can all be in an unstructured form. Then you can use the **terraform fmt --recursive** and give it a streamlined look. This is especially convenient if you have 30 files that can be formatted at the same time. It's difficult to track all this with your eyes, especially if the project consists of thousands of code lines, and you always want to have the code readable. The **terraform validate command** is also helpful: developing terraform code allows you to check whether there is enough **terraform** of all the attributes to execute terraform plan\apply.

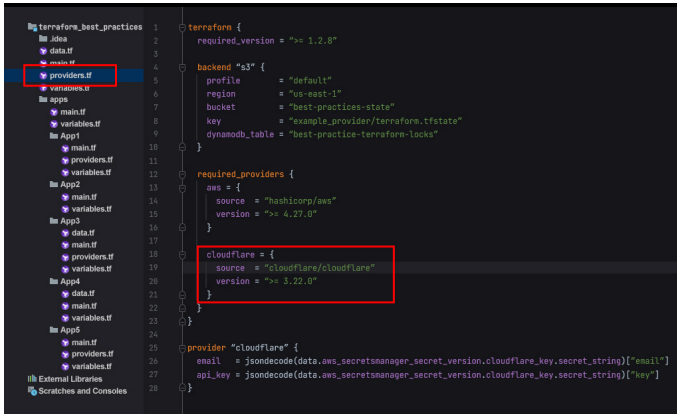
USE THE PROVIDER VERSION ONLY IN THE ROOT OF THE PROJECT, SO YOU DON'T NEED TO CHANGE IT IN DIFFERENT PLACES

As I mentioned at the beginning of the article, terraform uses providers to work with different platforms. A provider is a driver (instruction) that tells terraform how to communicate with the platform. There can be several providers in one terraform project. Imagine that you need to create a web server that should be accessible at example.terraform-learn.com.

To create a web server, you can use AWS\GCP\Azure etc. cloud providers. Each cloud has its own provider. Upon creating a server, you need to make a DNS record that will direct traffic to your web server. To create a DNS record in CloudFlare, you need to use Cloudflare Provider.

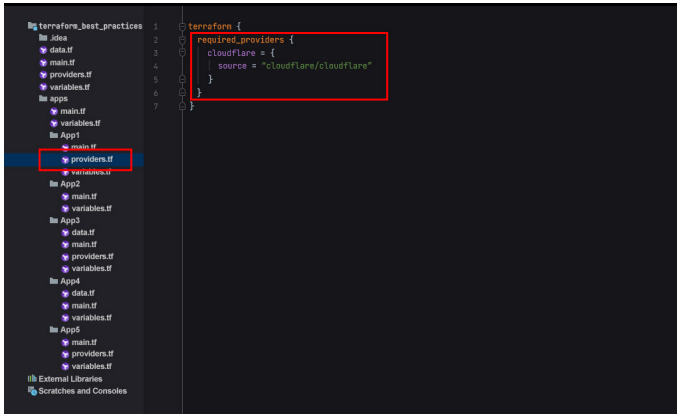
If your project has many applications, the best practice is to declare the provider version at the root of the project.

Figure 8



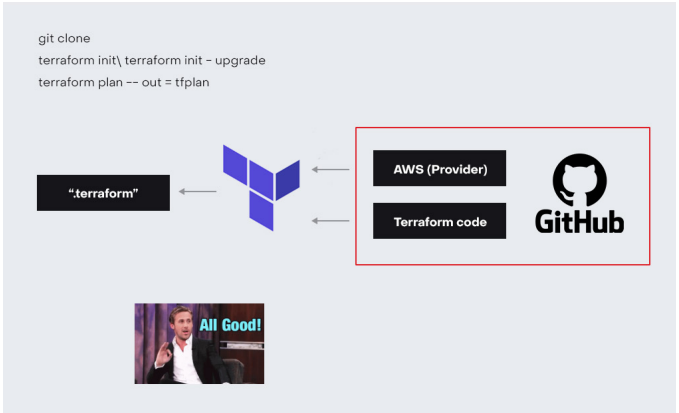
Next, refer to each application's root version if the provider is required. In other words, you manage the provider version only at the project root.

Figure 9



Now, a few words may save you some time searching for a problem I once faced. Let's suppose you have a terraform project. You are working with it at the moment, and you are doing well.

Figure 10

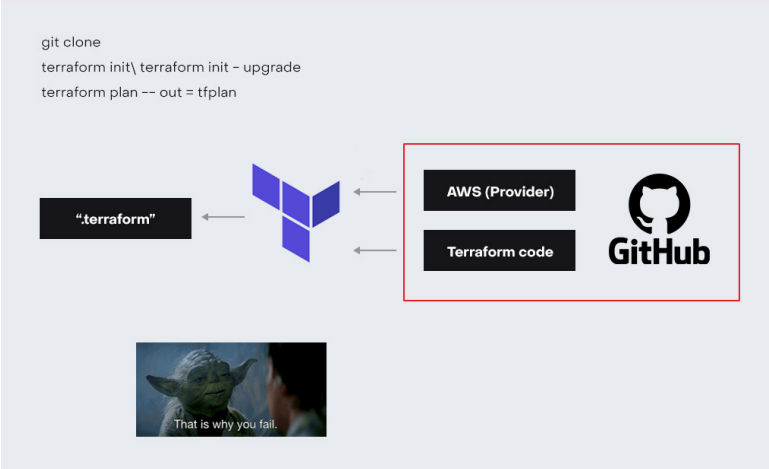


USE THE PROVIDER VERSION ONLY IN THE ROOT OF THE PROJECT, SO YOU DON'T NEED TO CHANGE IT IN DIFFERENT PLACES

The next day you perform the same commands:

1. git clone;
2. terraform init\terraform init -upgrade;
3. terraform plan.

Figure 11



And suddenly, you get an error.

At first, you think there's a problem with your code. You go to GitHub, and check it, and there've been zero changes since yesterday when we had everything running perfectly. And here you are wondering what happened. You go to your colleagues, ask them to make a terraform plan, and everything works out for them. What kind of magic is this? You keep debugging and can't make any sense.

Now let's sort through and figure out the magic of the error if you haven't changed anything. Spoiler alert! Something has been changed, and it's not magic.

1. Indeed, the code has not changed since everything was working correctly.
2. Indeed, terraform plan is running on your colleague's computer.

USE THE PROVIDER VERSION ONLY IN THE ROOT OF THE PROJECT, SO YOU DON'T NEED TO CHANGE IT IN DIFFERENT PLACES

It's all about the provider.

Between the points in time when everything worked and then stopped, the provider version, for instance, CloudFlare changed, and today you cannot make a plan for creating a DNS record. You'll probably say: ok, but why everything works for my colleague?

That's a good question. Your colleague has it working because the terraform init\terraform init-upgrade commands have not been run, and your colleague is not using the latest version of the provider. You, in turn, cloned the fresh code and initialized terraform, which took the last provider (yesterday it was 4.21.0, today it is 4.22.0), unless, of course, your provider is not fixed in terraform. If your colleague does terraform init\terraform init -upgrade, one will also have this problem.

Figure 12

MAKE SURE THERE ARE NO ERRORS IN TERRAFORM FMT AND TERRAFORM VALIDATE SYNTAX

A good practice is to use tags for resources. In this case, you will always know who owns the resource and whether it is covered by terraform. Also, when you look at costs, it's easy to identify who to ask why the infrastructure ate up so much money with the tags. There are two approaches to covering resources with tags:

1. Add tags to each resource (this is a good practice, but not the best).

2. Use default_tags in the root of the project. This is the best practice. In this case, all resources inside the project will have default tags. If, in addition to the default tags, you need to add any unique tags, use the merge construct. Even if we skip some resource and do not cover it with tags, terraform will not skip it and cover it.

Figure 13

```
variable "default_tags"  {
  default     = {

      Environment = "Test"
      Owner       = "TFProviders"
      Project     = "Test"


}
  description     = "Default Tags for Auto Scaling Group"
  type            = map(string)
}


resource "aws_autoscaling_group"   "example"  {
  # ... other configuration ...
  # This configuration combines some "default" tags with optionally provided additional

  tags = merge(
    var.default_tags,
    {
      Name = "MyASG"
    },
  )
}
```

Based on tags, you can design an entire automated system that will use a script to scrape the state of the instance and other things by tags.

MAKE A TERRAFORM PLAN WITH THE --OUT=TFPLAN KEY AND THEN APPLY

When creating or changing infrastructure, we use the terraform plan command.

This command shows us what terraform is going to do in infrastructure. If we do a terraform plan, terraform shows us what will be done directly in the console. If we use the --out=tfplan key, in this case, terraform will write all the changes that will be made to the local file.

We can perform terraform plan both with and without a key. It will work the same way. The only difference is that using the key will create a local file with all the changes that will be made, and if we do not use the above key, then all changes will be shown only in the console. If something goes wrong, you can always open the tfplan files and see what was planned to be done and what was actually done.

***

Bottom line: the infrastructure must be covered with code. Follow the Clean Code Principles when writing code. Avoid duplicating code — sometimes it's much faster to Copy\Paste and forget, but in this case, the quantity of your code will increase, not its quality. When developing the code and network in 'main', use PR Review, because, as a rule, the truth is born in the process of discussion, and your colleague, who has already faced a similar situation, can tell you the best option.

Innovecs is a global digital transformation tech company that presences in the US, the UK, the EU, Latin America, Israel, Australia, and Ukraine. Specializing in software solutions, the Innovecs team has experience in Gaming & Entertainment, Supply Chain & Logistics, FinTech, Healthtech, and Software & Hightech.

TOP 6 facts about Innovecs:

1. 850 employees globally, 9 countries of presence;
2. We have TikTok with 10M+ views and our tech talents starring;
3. Our 12 lifestyle communities have 53% employees' participation rate;
4. We're on the list of the 5,000 fastest-growing private companies in the U.S. by Inc. Magazine, with 127% growth over the last three years;
5. One of 100 Best Global Outsourcing Companies, recognized by IAOP;
6. Received the AWS Select Consulting Partner status in the AWS partner network.

For over a decade, the distinguishing feature of Innovecs has been its DNA and values — Innovate, Inspire, Care. The company's culture revolves around these concepts, creating multiple opportunities for the self-development of our people — both professionally and personally. Driven by the belief that only happy teammates promote creativity and innovation, Innovecs makes every effort to ensure the well-being of each team member. Regardless of the offices' location, all of them are united by the universal benefits system.

We are looking for DevOps Engineers.
Get acquainted with the list of vacancies and join us.