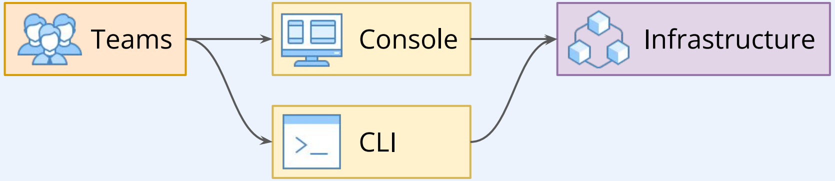
Terraform

What is Infrastructure as Code?

What is Infrastructure as Code?

Provisioning and managing infrastructure through code instead of manually

Creating resources manually



Teams

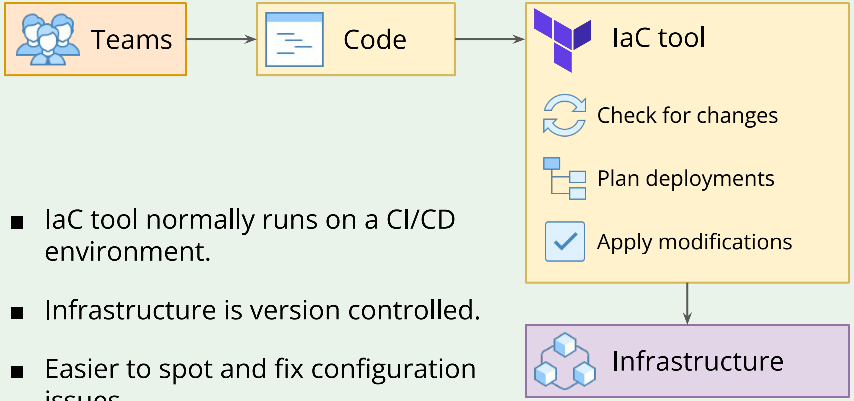
Console

CLI

Infrastructure

- Hard to reproduce large infrastructures.
- Hard to track and revert changes.
- Imperative rather than declarative.
- Very error-prone in terms of configuration.

Infrastructure as Code (IaC)



Teams

Code

IaC tool

Check for changes

Plan deployments

Apply modifications

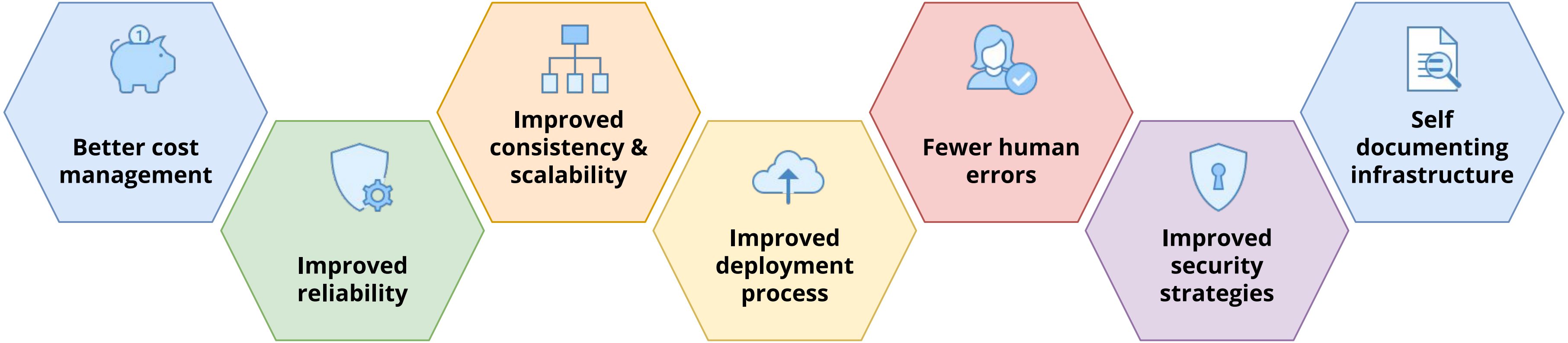
Infrastructure

- IaC tool normally runs on a CI/CD environment.
- Infrastructure is version controlled.
- Easier to spot and fix configuration issues.
- Declarative rather than imperative.

Terraform

Benefits of Infrastructure as Code

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

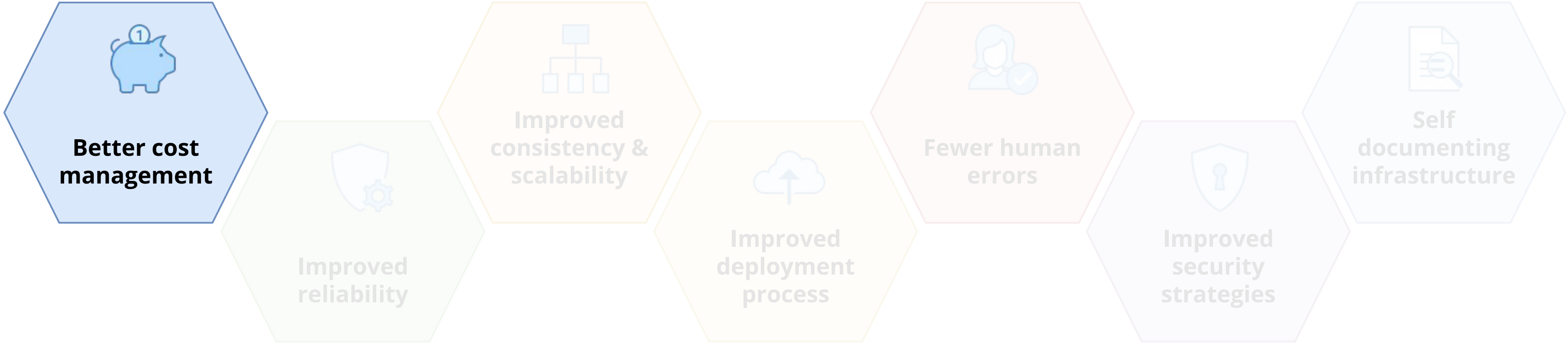
Improved deployment process

Fewer human errors

Improved security strategies

Self documenting infrastructure

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

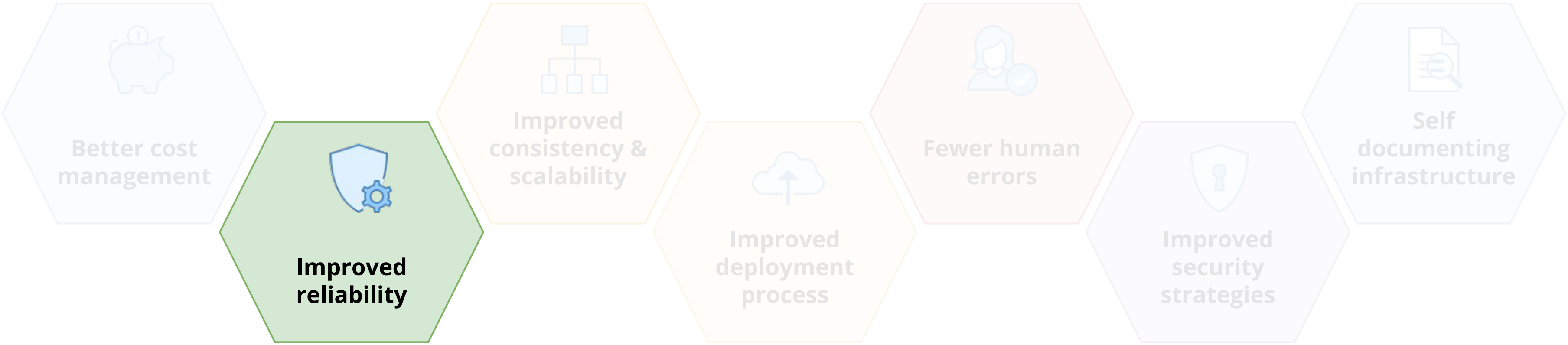
Fewer human errors

Improved security strategies

Self documenting infrastructure

- Resources, environments, and complex infrastructures can be easily created and destroyed.
- Automation considerably frees up the time of developers and infrastructure maintainers.
- Tagging strategies and requirements can be easily implemented across entire infrastructures.
- It becomes much easier to obtain an overview of all resources created by a specific IaC project.

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

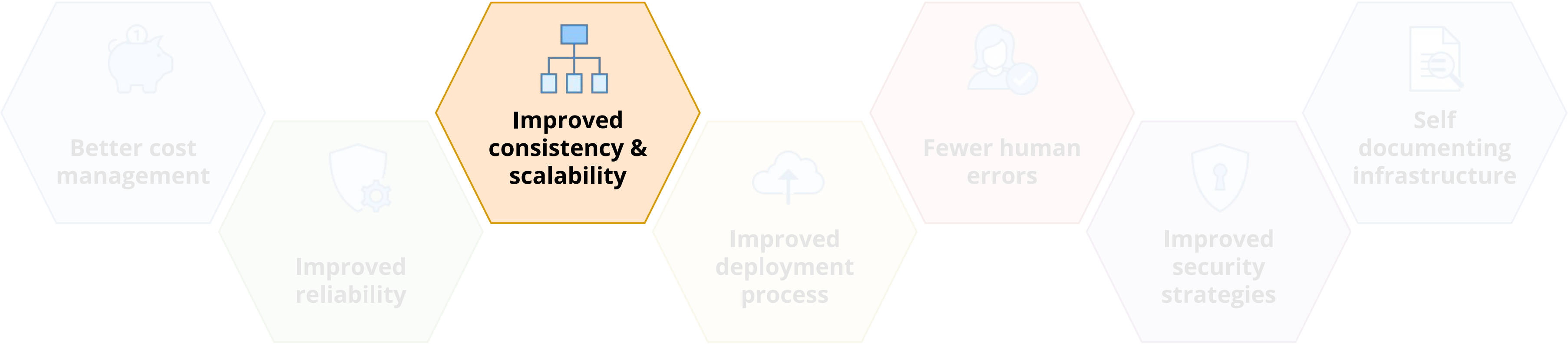
Fewer human errors

Improved security strategies

Self documenting infrastructure

- Well-developed IaC tools guarantee a consistent behavior.
- IaC tools provide multiple ways of deploying configurations: locally, as part of a CI/CD pipeline, triggered via API calls.
- IaC tools validate infrastructure configuration as part of the deployment process.

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

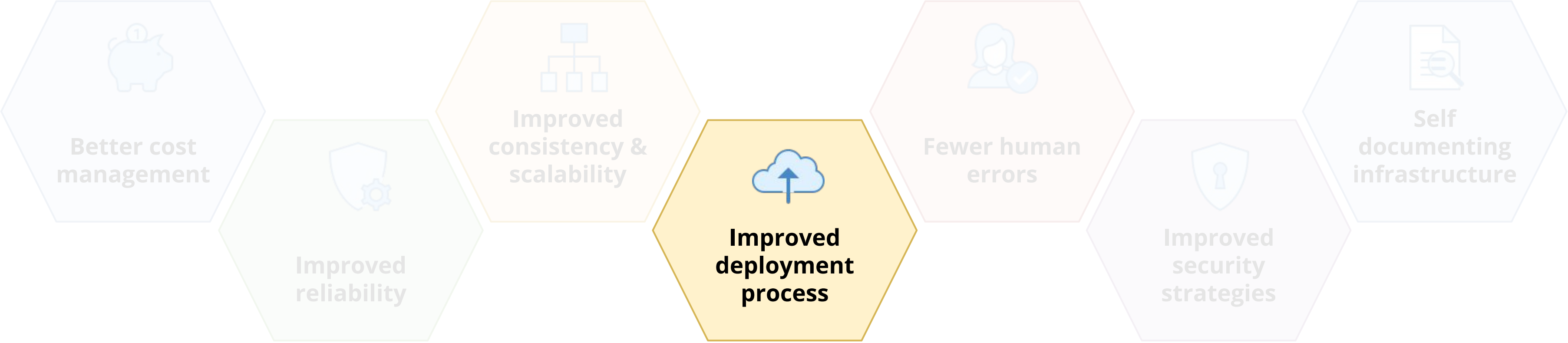
Fewer human errors

Improved security strategies

Self documenting infrastructure

- Infrastructure can be easily copied and deployed multiple times with the same structure.
- Modules can be created and made publicly or privately available.
- Different environments can be created based on the same / similar configuration files.
- Resource counts can be easily increased or decreased when needed.

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

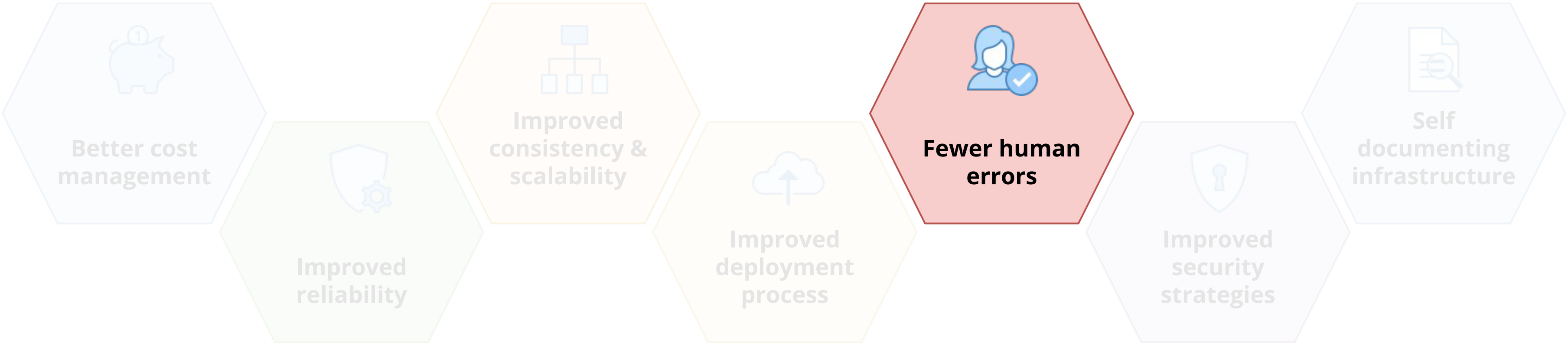
Fewer human errors

Improved security strategies

Self documenting infrastructure

- Automation saves a lot of time and effort when deploying infrastructure.
- Prevents configuration drift by identifying and reverting unexpected changes.
- Creating, updating and destroying resources becomes fully integrated with other CI/CD tasks.
- Changes to infrastructure are version controlled, being easier to revert in case of incompatibility or error.

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

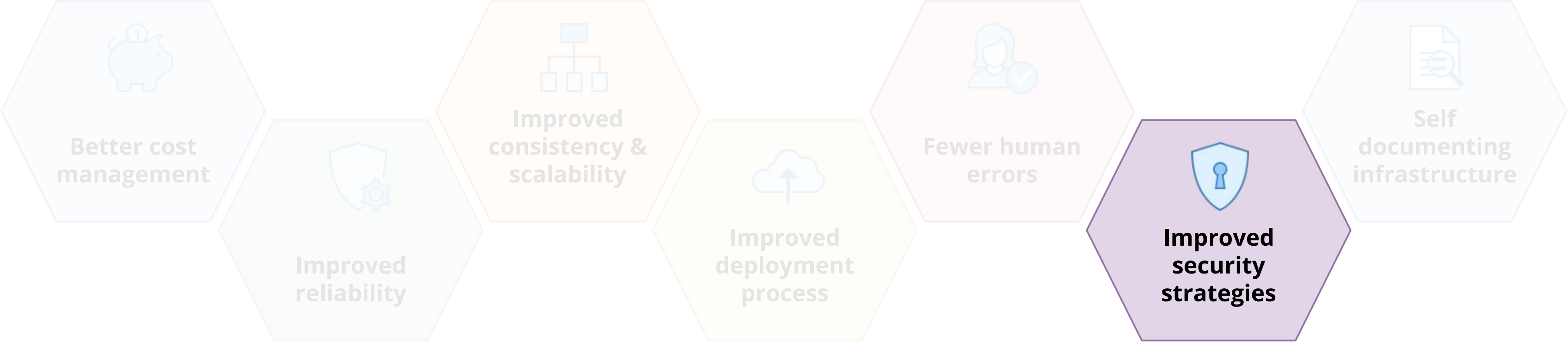
Fewer human errors

Improved security strategies

Self documenting infrastructure

- The planning stage shows all the changes that are expected to be carried out, and can be inspected by the engineers.
- Connecting and integrating different resources becomes more intuitive due to developer-friendly identifiers.
- Many IaC tools support validation and integrity checks with custom conditions.
- Many IaC tools support protection rules against deletion of critical resources.

Benefits of Infrastructure as Code



Better cost management

Improved reliability

Improved consistency & scalability

Improved deployment process

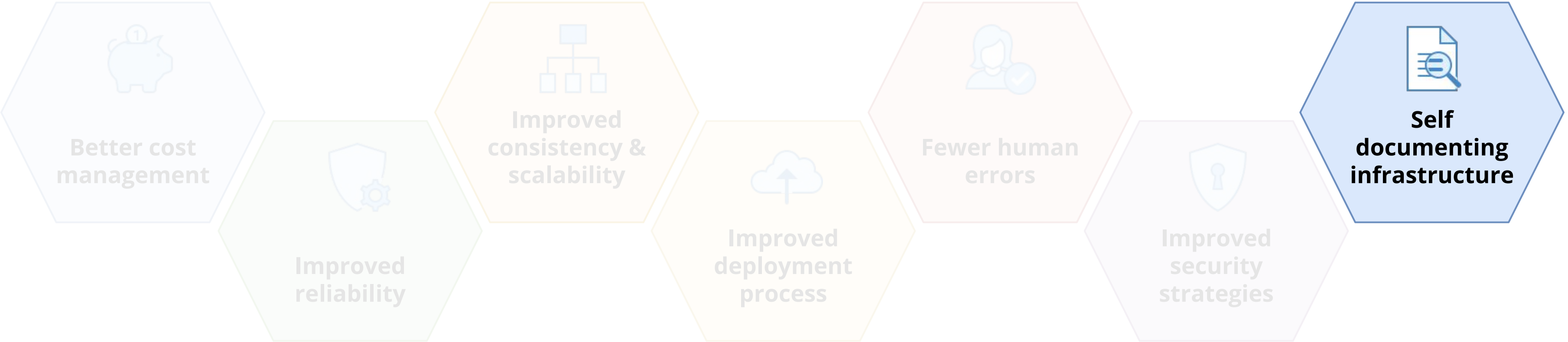
Fewer human errors

Improved security strategies

Self documenting infrastructure

- Validation and integrity checks can be used to ensure the infrastructure complies with security requirements.
- Shared infrastructure modules are normally maintained by teams with a strong focus on securing these resources.
- Security strategies (for example, IAM users, roles and their respective policies) can also be configured via IaC tools.
- The infrastructure configuration files can be inspected by security software for vulnerabilities.

Benefits of Infrastructure as Code


Better cost management
Improved reliability
Improved consistency & scalability
Improved deployment process
Fewer human errors
Improved security strategies
Self documenting infrastructure

- The created infrastructure is the infrastructure documented in the code.
- Many IaC tools allow detailed inspection of the resources created.
- Run logs are normally stored for a period of time, allowing inspection in case of any errors or unwanted changes.

Terraform

Why Terraform?

Why Terraform?

Advantages of Terraform over other Infrastructure-as-Code tools

Platform-agnostic

Terraform can be used with multiple providers, both in the cloud and on-premises.

High-level abstraction

Terraform can be used to manage resources across multiple providers.

Modular approach

We can create modules grouping resources, and then combine and compose these modules to build a bigger solution.

Parallel deployment

Terraform builds a dependency graph of resources and supports parallel changes.

Separation of plan and apply

We can execute only plan commands to inspect the potential changes before actually applying them.

Resource protection & validation

Terraform Provides several ways to protect resources against accidental changes or deletion, as well as ways of validating the deployed infrastructure.

State file

Terraform is very fast due to its implementation of a state file, saving the entire snapshot of the current deployment.

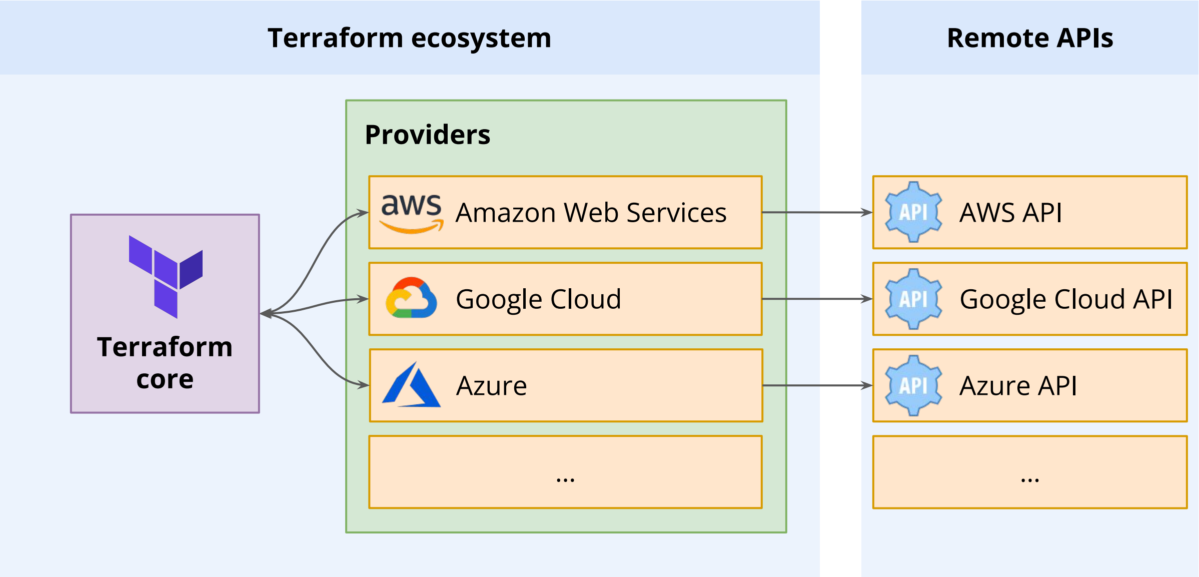
Terraform

Terraform's Architecture

Terraform's Architecture

How does Terraform manage resources in so many different places?

- **Terraform providers** allow Terraform to know how to interact with remote APIs.
- Providers provide the logic to interact with upstream APIs:
  - Read, create, update, and delete resources through that provider's API.
- Anyone can write and publish a provider for a remote API, as long as it follows Terraform's specifications.
- We declare the necessary providers in the Terraform project configuration, and Terraform installs those providers during initialization.

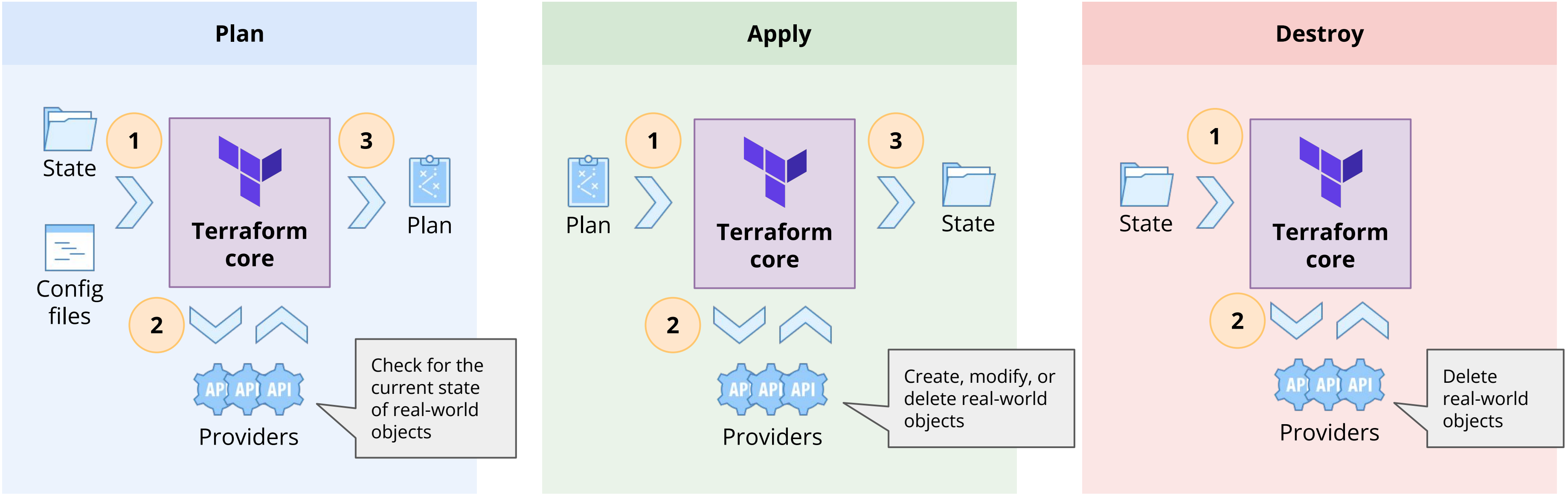


Terraform

Provisioning Infrastructure

Provisioning Infrastructure

How does Terraform provision and manage infrastructure?



Plan

State

Config files

1

Terraform core

3

Plan

2

Providers

Check for the current state of real-world objects

Apply

Plan

1

Terraform core

3

State

2

Providers

Create, modify, or delete real-world objects

Destroy

State

1

Terraform core

2

Providers

Delete real-world objects

Terraform

Terraform Block

Terraform Block

Used for configuring the Terraform project (backend, providers, required versions)

- Only constants are allowed within the `terraform` block. Input variables or resource references are not allowed.
- `cloud` block: used to configure Terraform Cloud.
- `backend` block: used to configure a state backend for the project.
- `required_version` key: used to specify the accepted versions of Terraform for the current project.
- `required_providers` block: specifies the required providers for the current project or module, including their accepted versions.

```
terraform {
  required_version = "1.7.0"

  backend "s3" {
    // ...
  }

  required_providers {
    aws = {
      source  = "hashicorp/aws"
      version = "~> 5.33.0"
    }
  }
}
```

Version constraints

`=` : allows only the specified version.

`!=` : excludes an exact version.

`>=, <=, >, <` : allow versions for which the comparison is true.

`~>` : allows only the rightmost digit to increment.

```
required_version = ">=1.7.0"
required_version = ">1.5.0, <1.7.0"
required_version = "~>1.5.0"
required_version = "~>1.5"
```

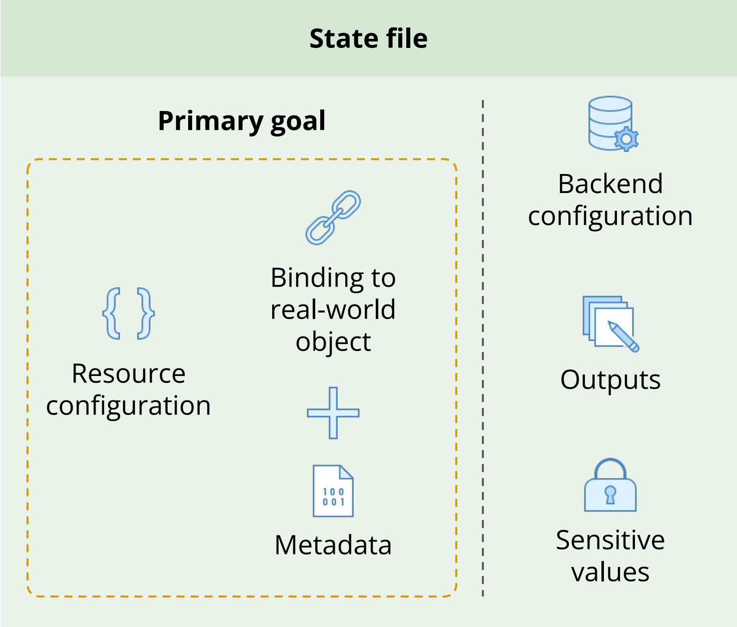
Terraform

Terraform State

Terraform State

Terraform State maps resources from the configuration to real-world objects

- State is always required in Terraform.
- **Important!** The state contains extremely sensitive data, so be careful regarding who has access to it.
- The state file also stores metadata, such as resource dependencies, so that Terraform knows in which order resources must be created, updated or deleted.
- Before any planning operation, Terraform refreshes the state with the information from the respective real-world objects.
  - This is essential to avoid configuration drift. If a real-world object has been modified outside of Terraform and the respective configuration has not been updated, Terraform will revert the changes.
- State can be either stored locally (default) or in several remote backends (S3, Google Cloud Storage, Terraform Cloud, among others).
- **State locking:** locks the state while executing write operations to prevent concurrent modifications and state corruption.



State file

Primary goal

Resource configuration

Binding to real-world object

Metadata

Backend configuration

Outputs

Sensitive values

Terraform

Backends

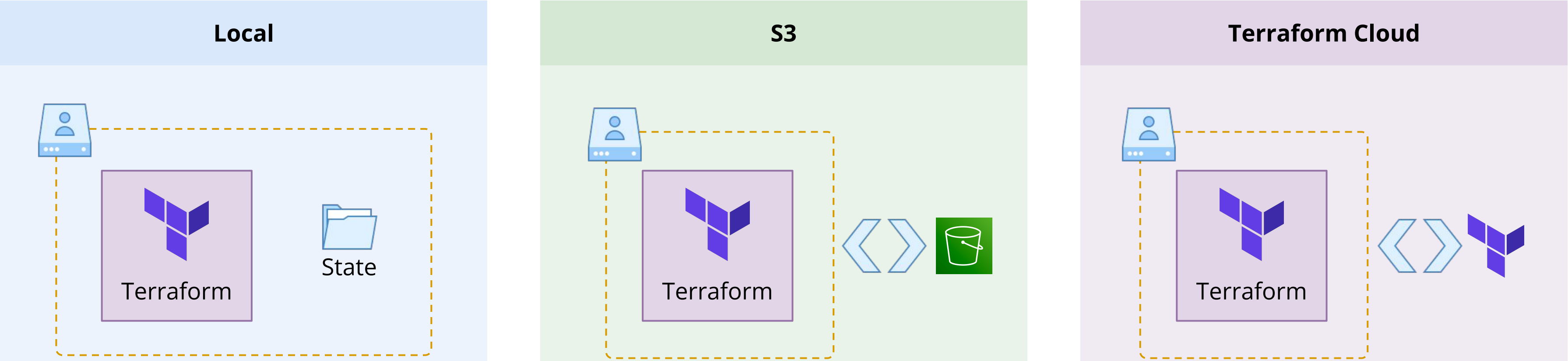
Backends - Overview

Backends define where Terraform stores its state file

- There are multiple types of backends, which can be place into three categories:
  - **Local:** the state file is stored in the user's local machine.
  - **Terraform Cloud:** the state file is stored in Terraform Cloud. Offers additional features.
  - **Third-party remote backends:** the state file is stored in a remote backend different from Terraform Cloud (for example S3, Google Cloud Storage, Azure Resource Manager / Blob Storage, among others).
- Different backends can offer different functionalities (for example, state locking is not available for all remote backends) and require different arguments (we'll discuss more of that in the next slide).
- A Terraform configuration can provide only one backend.
- The `backend` block cannot use any input variables, resource references, and data sources.
- Remote backends require authentication credentials in order for Terraform to properly access the files.
- When changes are made to the configured backend, we must rerun the `terraform init` command.
- Terraform offers the possibility of migrating state between different backends.

Backends - Local and Remote

Terraform provides different options for backends, both local and remote



Local backends: are stored in the same machine as the Terraform project.

S3 backends: are stored in an S3 bucket in AWS.

Terraform Cloud backends: are stored in a Terraform Cloud project and workspace.

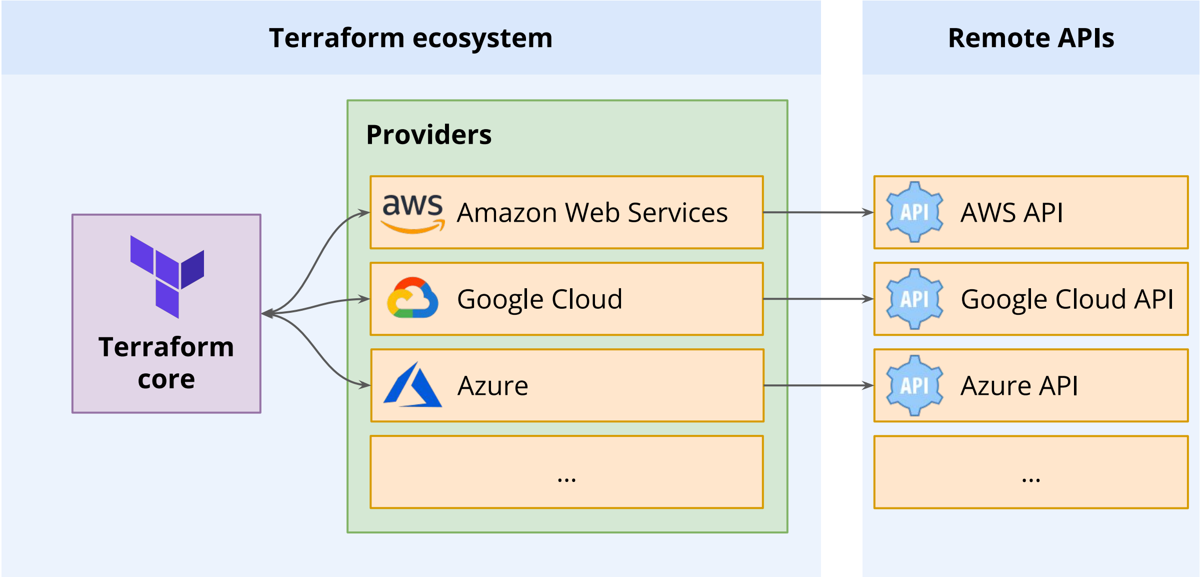
Terraform

Providers

Providers Overview

Providers are how Terraform interacts with remote APIs and platforms

- Each provider adds a set of resource types and data sources that the Terraform project can use.
- Providers are developed and maintained separately from Terraform. They fit into Terraform's plugin architecture.
- The Terraform configuration must declare all the providers it requires.
- Provider configurations belong to the root module of a Terraform project. Child modules receive their provider configuration from the parent module.
- We can use the same version constraints as when specifying the Terraform version, and we can create a dependency lock file to ensure that the exact versions of providers are installed.



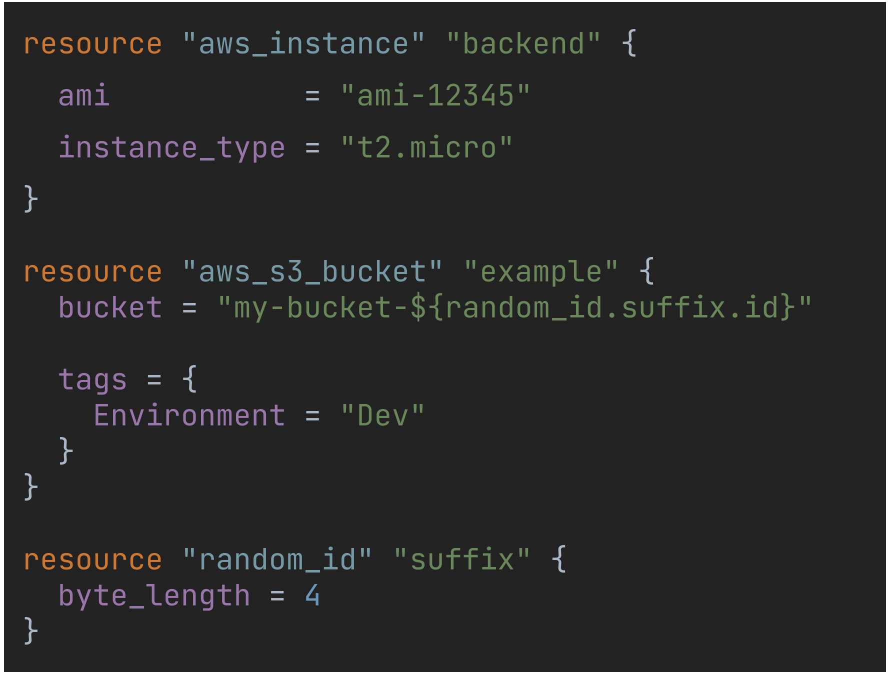
Terraform

Resources

Resource Blocks

Resources are used to configure real-world infrastructure objects

- They are used to manage any infrastructure that we want to manage with Terraform, and are the most important blocks within Terraform.
- Resource blocks represent things like virtual networks, compute instances, DNS records, storage disks, among others.
- The arguments depend on and vary based on the resource type.
- The combination of resource type and resource name must be unique within a module. This is how Terraform links the resource to its respective real-world object.
- If no other provider is specified, Terraform will use the default provider to provision the infrastructure. We can use the `provider` meta-argument to explicitly pass a different provider.
- Terraform offers a few local-only resources, such as generating random strings or IDs, private keys, or self-issued TLS certificates.
- We can create multiple instances of resources by using Terraform loops (`for_each`, `count`).



```
resource "aws_instance" "backend" {
  ami           = "ami-12345"
  instance_type = "t2.micro"
}

resource "aws_s3_bucket" "example" {
  bucket = "my-bucket-${random_id.suffix.id}"

  tags = {
    Environment = "Dev"
  }
}

resource "random_id" "suffix" {
  byte_length = 4
}
```

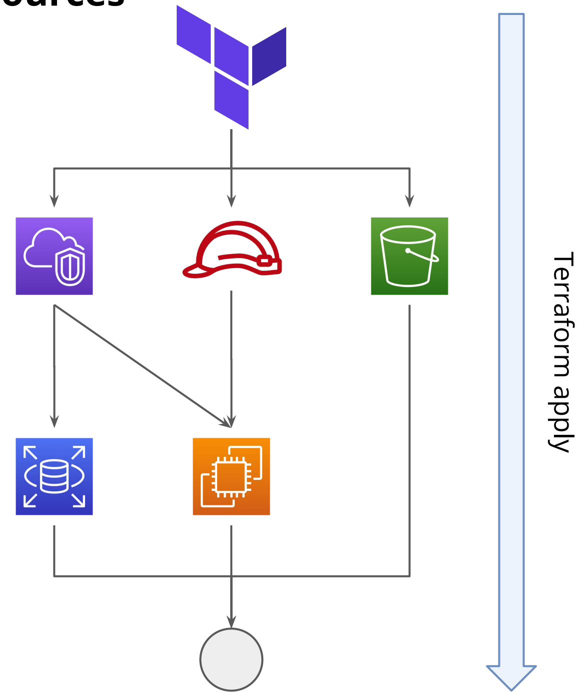
Terraform

Resource Dependencies

Resource Dependencies

Terraform supports both parallel and sequential creation of resources

- Certain resources can be created in parallel, while other resources depend on each other and must be created in a certain order.
- Terraform supports both parallel and sequential management of resources.
- Terraform inspects the expressions to automatically establish implicit dependencies between resources.
- Additionally, we can define explicit dependencies via the `depends_on` meta-argument.
- If the operation on an upstream resource fails, Terraform will not continue the operations on downstream resources.
- We can also force Terraform to replace a parent resource in case a child resource is modified by using the `replace_triggered_by` meta-argument.



Terraform

Meta-Arguments

Meta-arguments

Meta-arguments allow us to configure Terraform's behavior in many ways

`depends_on`

Used to explicitly define dependencies between resources.

`count` and `for_each`

Allow the creation of multiple resources of the same type without having to declare separate resource blocks.

`provider`

Allows defining explicitly which provider to use with a specific resource.

`lifecycle`

`create_before_destroy`

Prevents Terraform's default behavior of destroying before creating resources that cannot be updated in-place. The behavior is propagated to all resource's dependencies.

`replace_triggered_by`

Replaces the resource when any of the referenced items change.

`prevent_destroy`

Terraform exits with an error if the planned changes would lead to the destruction of the resource marked with this.

`ignore_changes`

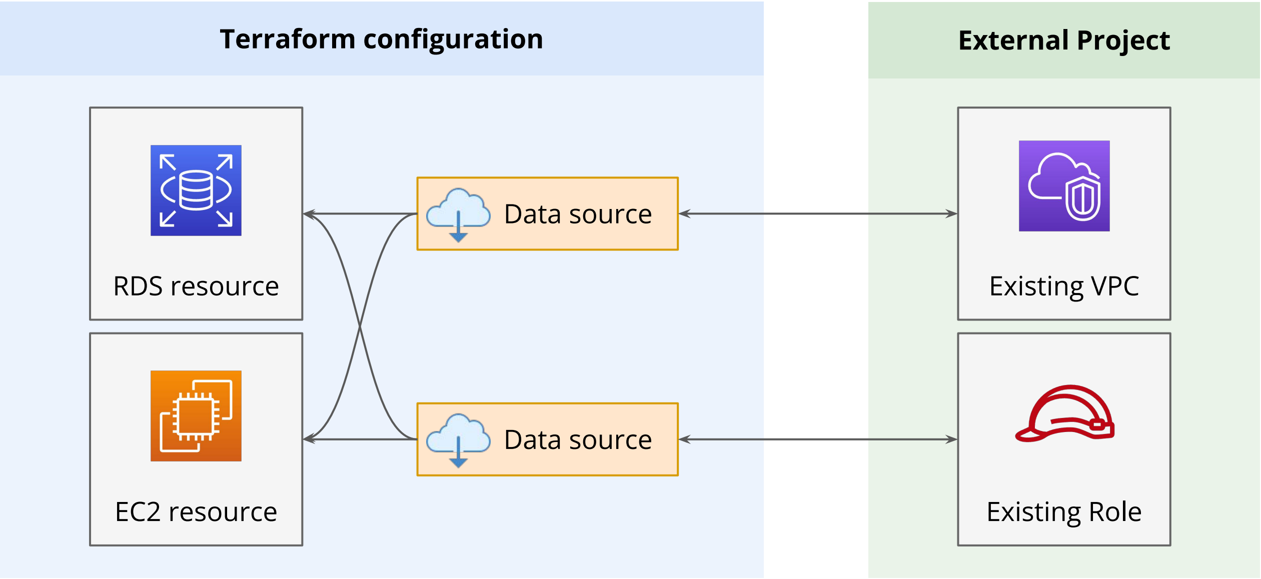
We can provide a list of attributes that should not trigger an update when modified outside Terraform.

Terraform

Data Sources

Data Sources

Query or retrieve data from APIs or other Terraform projects



Use-case

Retrieve data from infrastructure that is not managed by the current Terraform project.

Terraform

Input Variables

Input Variables

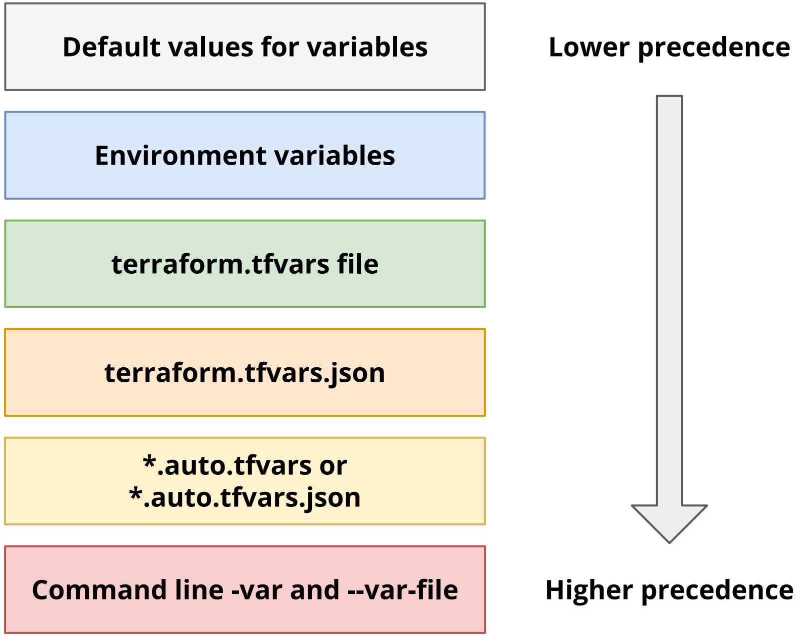
Customize values based on variables to create reusable and composable code

Motivation for input variables:

- To be able to customize aspects of Terraform configurations and modules without having to alter the source code.

Notes around variables:

- It's convention to declare them inside of `variables.tf` file, and we use them via `var.<NAME>`
- When defining a variable, we can set the type, provide a description, give a default value, set `sensitive` to a boolean value, and provide validation rules.
- When we run `terraform plan` or `apply` and don’t provide command-line arguments for the variables, it will ask us to provide the values for each of the variables.
  - If we provide defaults, Terraform will not ask for these values.



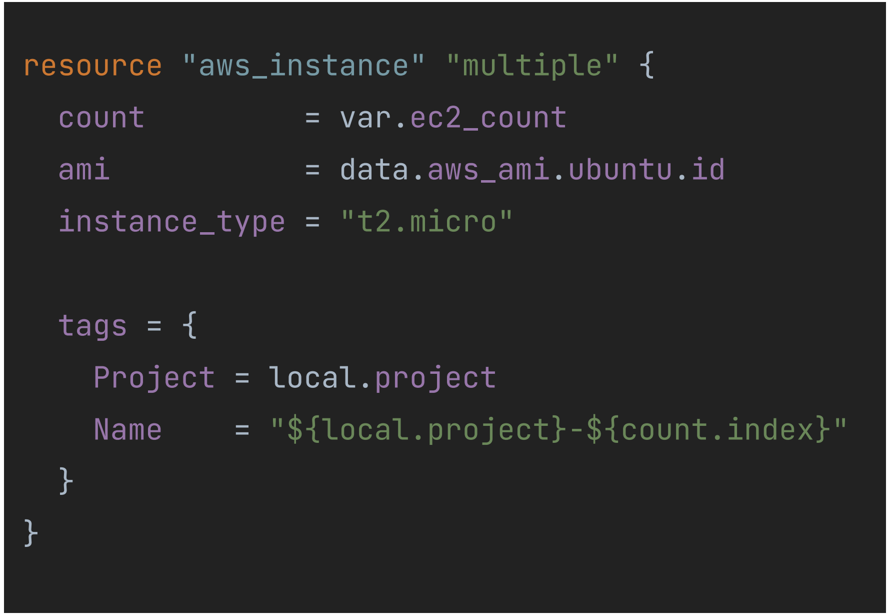
Terraform

Creating Multiple Resources

Creating Multiple Resources

Avoid code duplication by leveraging count and for_each meta-arguments

- `count`: used to define the number of instances of a specific resource Terraform should create.
    - It can be used with modules and with resources.
    - Must be known before Terraform performs any remote resource actions.
    - `<TYPE>.<LABEL>[<INDEX>]` refers to a specific instance of a resource, while `<TYPE>.<LABEL>` refers to the resource as a whole.
    - We can use `count.index` in the resource's arguments to retrieve the index of the specific instance.



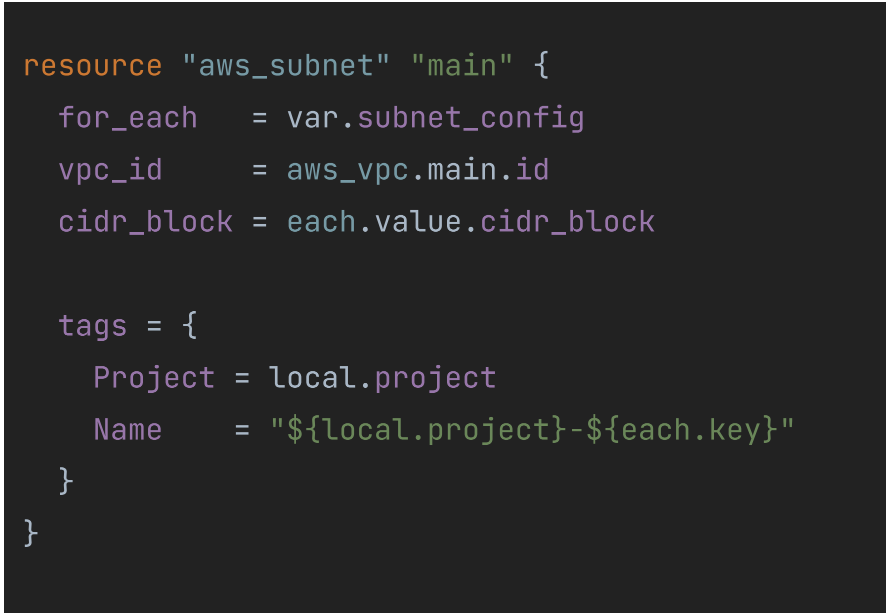
```
resource "aws_instance" "multiple" {
  count         = var.ec2_count
  ami           = data.aws_ami.ubuntu.id
  instance_type = "t2.micro"

  tags = {
    Project = local.project
    Name    = "${local.project}-${count.index}"
  }
}
```

Creating Multiple Resources

Avoid code duplication by leveraging count and for_each meta-arguments

- for_each: accepts a map or a set of strings and creates an instance for each entry in the received expression.
  - We can access the key and value via the each object. Key and value are the same if the received value is a set.
  - We should not use sensitive values as arguments to the for_each meta-argument.
  - The for_each value must be known before Terraform performs any remote operations.
  - We can chain for_each resources into other for_each expressions if we need to create multiple resources based on a map or set.



```
resource "aws_subnet" "main" {
  for_each   = var.subnet_config
  vpc_id     = aws_vpc.main.id
  cidr_block = each.value.cidr_block

  tags = {
    Project = local.project
    Name    = "${local.project}-${each.key}"
  }
}
```

Terraform

Modules

Modules

Organize, encapsulate, and re-use Terraform components

- Modules are used to combine different resources that are normally used together. They are just a collection of `.tf` files kept in the same directory.
  - **Root module**: the set of files in the main working directory.
  - Root modules can then call other modules (child modules), defined either locally or remotely.
- The goal is to develop modules that can be reused in various ways.
- Why use modules?

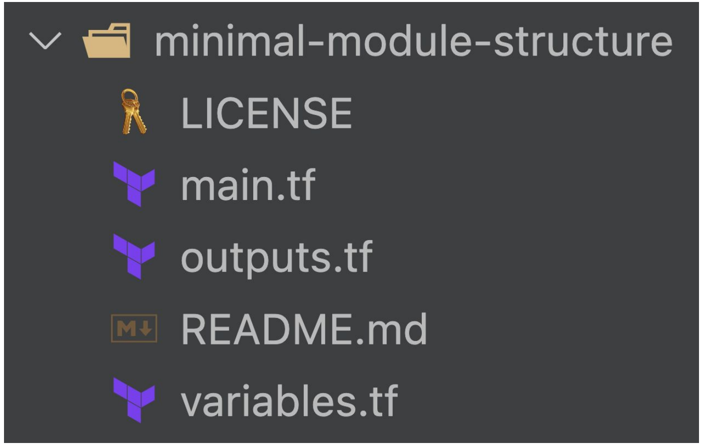
Organize configuration	Encapsulate configuration	Re-use configuration	Ensure best practices
Group related parts of the infrastructure to make the code easier to understand and improve maintainability.	Encapsulate sets of resources to prevent unintended changes and mistakes that can happen in complex code bases.	Modules make it much easier to reuse entire sets of components, thus improving consistency, saving time, and preventing errors.	Provide configuration and infrastructure best practices, and publish modules both publicly and privately for use by other teams.

Terraform

Standard Module Structure

Standard Module Structure

Which files are recommended, and what they are used for



```
minimal-module-structure
    LICENSE
    main.tf
    outputs.tf
    README.md
    variables.tf
```

`main.tf`

Main entry point for module resources. More complex modules should split the resources into multiple files with relevant names.

`outputs.tf`

File containing all outputs from the module. Used by Terraform Registry to generate documentation.

`variables.tf`

File containing all variables for the module. Used by Terraform Registry to generate documentation.

`README.md`

File containing documentation for the module. Used by Terraform Registry to generate documentation.

Terraform

Building Modules

Building Modules

When to create modules and which best practices to follow?

- Creating a module is as simple as creating a directory and a couple of terraform files within that directory.
- When to build modules?
  - When useful abstractions of our infrastructure can be identified.
  - When certain groups of resources always need to be created together and strongly depend on each other.
  - When hiding the infrastructure details of a certain part of our infrastructure will lead to better developer experience.
- Which best practices to follow?
  - **Use object attributes:** group related information under object-typed variables.
  - **Separate long-lived from short-lived infrastructure:** resources that change rarely should not be grouped together with resources that change often.
  - **Do not try to cover every edge case:** this can quickly lead to highly complex modules, which are difficult to maintain and configure. Modules should be reusable blocks of infrastructure, and catering to edge cases goes against that purpose.
  - **Support only the necessary configuration variables:** do not expose all the internals of the module for configuration via variables. This hurts encapsulation and makes the module harder to work with.

Building Modules

When to create modules and which best practices to follow?

- Which best practices to follow?
  - **Output as much information as possible:** even if there is no clear use for some information, providing it as an output will make the module easier to use in future scenarios.
  - **Define a stable input and output interface:** All used variables and outputs create coupling to the module. The more coupling, the harder it is to change the interface without breaking changes. Keep this in mind when designing the module's interface.
  - **Extensively document variables and outputs:** this helps the module's users to quickly understand the module's interface and to work more effectively with it.
  - **Favor a flat and composable module structure instead of deeply nested modules:** deeply nested modules become harder to maintain over time and increase the configuration complexity for the module's users.
  - **Make assumptions and guarantees explicit via custom conditions:** do not rely on the users always passing valid input values. Thoroughly validate the infrastructure created by the module to ensure it complies with the requirements the module must fulfill.
  - **Make a module's dependencies explicit via input variables:** data sources can be used to retrieve information a module needs, but they create implicit dependencies, which are much harder to identify and understand. Opt for making these dependencies explicit by requiring the information via input variables.
  - **Keep a module's scope narrow:** do not try to do everything inside a single module.

Terraform

Publishing Modules

Publishing Modules

Make your Terraform module available to others via Terraform Registry

- Anyone can publish a module, as long as the following conditions are met:
  - The module must be on GitHub and must be a public repo. This is required only for using the public registry; for private ones, this can be ignored.
  - Repositories must use a naming format: `terraform-<PROVIDER>-<NAME>`, where `PROVIDER` is the provider where resources are created, and `NAME` is the type of infrastructure managed by the module.
  - The module repository must have a description, which is used to populate the short description of the module. This should be a simple one sentence description of the module.
  - The module should adhere to the standard module structure (`main.tf, outputs.tf, variables.tf`). The registry uses this information to inspect the module and generate documentation.
  - Uses `x.y.z` tags for releases. The registry uses these tags to identify module versions. Release tag names must be a semantic version, and can be optionally prefixed with a `"v"`.
- Published modules support versioning, automatically generate documentation, allow browsing version histories, show examples and READMEs, and more.

Terraform

Object Validation

Object Validation

More reliable infrastructure through pre and postconditions, and check assertions

Preconditions

```
lifecycle {
  precondition {
    condition     = ...
    error_message = "..."
  }
}
```

- Used from within resources and data blocks
- Cannot reference the resource itself.
- Can be used to check the validity of data blocks or variables that the resource references.

Postconditions

```
lifecycle {
  postcondition {
    condition     = ...
    error_message = "..."
  }
}
```

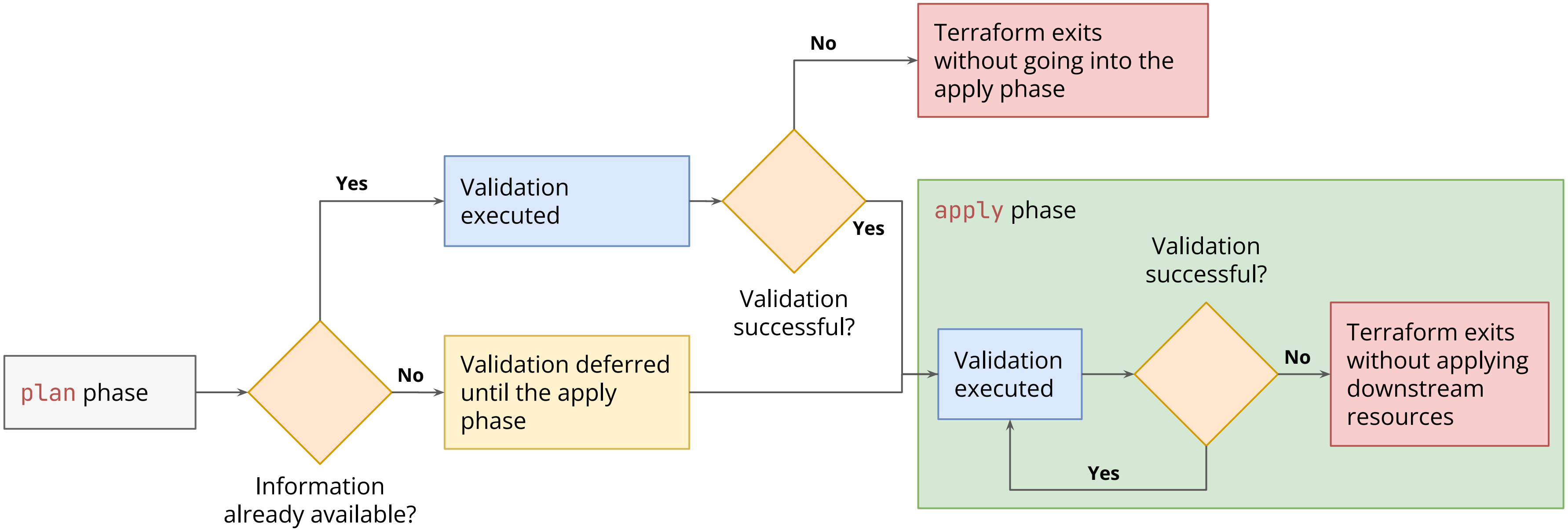
- Used from within resources and data blocks
- Can reference the resource itself.
- Can be used to check the validity of the resource's configuration.

Check assertions

```
check "my_custom_check" {
  assertion {
    condition     = ...
    error_message = "..."
  }
}
```

- Used from outside resources and data blocks
- Can reference information from across the current Terraform project.
- Results only in a warning and does not stop the apply process.

Object Validation - Pre and Postconditions


plan phase
Information already available?
Yes
No
Validation executed
Validation deferred until the apply phase
Validation successful?
No
Yes
Terraform exits without going into the apply phase
apply phase
Validation executed
Validation successful?
No
Yes
Terraform exits without applying downstream resources

The apply phase executes until there are no more changes to be applied.

Terraform

State Manipulation

State Manipulation

Recreate, import, refactor, and untrack infrastructure within Terraform

Tainting

Force the recreation of a resource that is tracked by a Terraform configuration. Can be used when a certain resource goes into an invalid state, but the configuration is correct and hasn't changed.

Importing

Import existing resources into a Terraform project, and start managing them with Infrastructure as Code.

Refactoring

Rename resources without recreating them, and move them inside and outside of modules whenever needed. Prevents recreation due to changing resource addresses in Terraform.

Untracking

Remove a resource from a Terraform configuration without actually destroying that resource. Useful when we want to manage the resource independently of the Terraform project.

Generating Configuration

Leverage Terraform's code generation feature to generate a best-effort configuration based on existing resources. Can be used when importing resources into Terraform.

Fine-grained State File Changes

Force state unlocking, and pull and push the state file from remote backends to perform careful, fine-grained editing in case something is wrong with it.

Terraform

Workspaces

Workspaces

Use a single code-base for different environments

- Workspaces allow us to leverage the same configuration directory to create different environments.
  - As a result, we can reduce code duplication and avoid installing multiple copies of modules and providers.
- When using the CLI in a workspace, resources from other workspaces are not considered.
- Different workspaces correspond to different state data. Terraform stores them in different `.tfstate` files.
- Terraform always has at least one workspace called `default`.
  - The default workspace is created when we initialize the Terraform project.
  - It is used by default when we do not specify any other workspace.
- Most, but not every, remote backend support workspaces.
- We can use `terraform.workspace` to access the current workspace, and change options based on the selected workspace.
  - **Recommendation:** do not use `terraform.workspace` for conditional operations. Instead, receive the information via variables.
- **This is different from Terraform Cloud's workspaces.**

Terraform

Terraform Cloud

Terraform Cloud

Execute Terraform code in a secure remote environment instead of locally

- Terraform Cloud is a solution from HashiCorp to manage Terraform projects remotely. It offers several features:
    - Access control to approving infrastructure changes
    - Secure state and secret storage
    - Graphical representation of managed resources
    - Private registry for sharing modules within an organization
    - Policy controls for managing the contents of Terraform projects
    - Run history
- Offers a free and a paid tier.
- Resources are organized into workspaces, and workspaces can be organized into projects.
- Workspaces store resource definitions, environment and input variables, and state files.

Terraform Cloud - Workspaces

Organize workspace information in a single place

- They are a required component of Terraform Cloud projects.
- In addition to managing information in a central place, workspaces are also a major component of role-based access to Terraform projects in Terraform Cloud.
- Paid plans offer workspace-based health checks that support drift detection and continuous infrastructure validation, as well as other advanced features.
- Workspaces can be linked to branches of VCS repositories, and we can specify which files and directories within the VCS repository trigger runs.

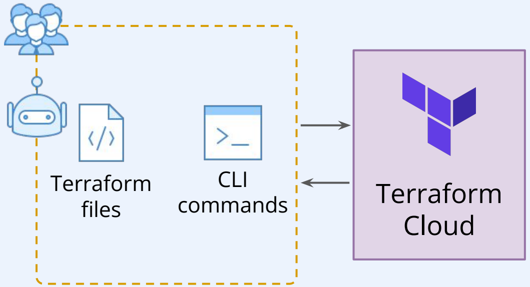
	Local Terraform	Terraform Cloud
Configuration	Saved on disk	VCS repository, API/CLI
Variables	Passed via the different methods we explored (`.tfvars`, CLI arguments, etc.)	Saved in workspace
State file	Saved locally or by using a remote backend	Saved in workspace
Secrets	Passed via environment variables or through the prompt	Saved in workspace

Terraform Cloud Workflows

Terraform Cloud Workflows

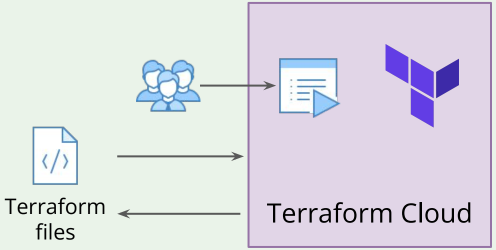
Trigger runs from the CLI, UI, connected VCS repositories, or the Terraform API

CLI



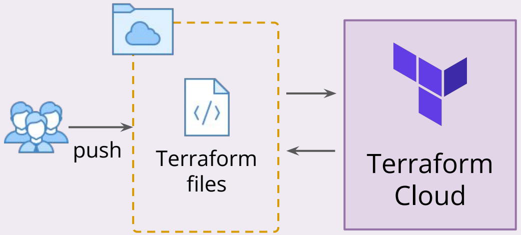
CLI-driven workflow: developers or CI/CD systems login to Terraform Cloud via the CLI and interact with the remote environment via CLI commands.

UI / API



UI or API-driven workflow: developers or systems manually trigger and manage runs by interacting with Terraform Cloud's UI or API.

VCS Repository



VCS-driven workflow: developers push code to a remote repository, which then triggers the terraform plan and apply commands.