Table of Contents

Examination Content

The exam may encompass a range of topics that we have covered throughout the course. To aid in your preparation, I would like to highlight some key areas that you might encounter:

- **Core Principles of Infrastructure as Code**: Reflect on the three core principles outlined in our course textbook. These principles form the foundational understanding of Infrastructure as Code and its application in real-world scenarios.
- **Understanding Terraform HCL**: Delve into how Terraform HashiCorp Language (HCL) operates, focusing on its declarative nature, idempotency, and the structuring of blocks. This knowledge is crucial for writing effective and efficient Terraform code.
- **Terraform Providers**: Investigate the role and functionality of Terraform providers in managing resources across various service providers.
- **Terraform Modules**: Understand what Terraform modules are, including their benefits and best practices in modularizing infrastructure code.
- **Terraform State File**: Examine the purpose and significance of the Terraform state file and what it represents in the context of infrastructure management.
- **Analyzing Terraform Configurations**: Be prepared to analyze examples of Terraform configurations, identifying their strengths and areas for improvement.
- **CI/CD in Infrastructure as Code**: Explore the concept of Continuous Integration and Continuous Deployment (CI/CD), focusing on its benefits and how it integrates with Infrastructure as Code practices.
- **GitHub and GitHub Actions**: Understand how GitHub and GitHub Actions can enhance team productivity and contribute to the delivery of reliable infrastructure.

As you prepare for this exam, I encourage you to revisit our course materials, engage in discussions with peers/studass in discord, and utilize the resources available to you. This examination is not just a test of your memory but an assessment of your ability to apply theoretical concepts in practical scenarios.

Wishing you all the very best in your preparations.

Core Principles of Infrastructure as Code

Reflect on the three core principles outlined in our course textbook. These principles form the foundational understanding of Infrastructure as Code and its application in real-world scenarios.

What is Infrastructure as Code?

Using code, you define the infrastructure that needs to be deployed in a descriptive model.
The code for the infrastructure becomes part of the project and is stored inside your version control system (or VCS).

Benefits of Infrastructure as Code

IaC solves many common problems with provisioning Infrastructure.

- New environments or infrastructure can be provisioned easily from your IaC configuration code. Infrastructure deployments with IaC are **repeatable**.
- Manually configured environments are difficult to scale. With environments provisioned using IaC, they can be **deployed and scaled rapidly**.
- If you want to make changes to the existing infrastructure that has been deployed with IaC, this can be done in code, and the changes will be **tracked**. -When IaC is used with a declarative tool (it describes the state you want your environment to look like), you can **detect and correct environment drift**. If a part of the infrastructure is modified manually outside of the code, it can be brought back in line with the desired state on the next run.
- Changes can be applied multiple times without changing the result beyond the initial application. This is known as **idempotence**.
- **Avoid manual configuration** of environments which can typically introduce mistakes due to human error.
- IaC is a means to achieve **consistency** across environments and infrastructure. The code can be reused.
- Infrastructure **costs are lowered** as the time to deploy and effort to manage, administer and maintain environments decrease.
- IaC can be used in Continuous Integration / Continuous Deployment (or CI/CD) pipelines. The main benefit of doing this is to **automate** your Infrastructure deployments.
- DevOps teams can **test applications** in production-like environments early in the development cycle.
- With your Infrastructure configuration code held in your version control system alongside your application source code, commonly in the same repository. Now everything can be **held together**.
- **Productivity will increase** due to a combination of all the benefits of using IaC.
- As the code is held in your version control system, it **gains all the benefits of the VCS**. More on that in the next section.

Challenges and Limitatoins with IaC

- Traditional infrastructure or operations teams within organizations may not be familiar with version control systems.
- The learning curve for IaC can be steep for some teams.

- Skills in IaC and DevOps are highly sought after, so therefore it may be difficult to hire people with these skills.
- Migration from traditional infrastructure to IaC can be difficult and time-consuming.

Four Key Metrics

> DORA's Accelerate research team has identified four key metrics that are highly correlated with software delivery and operational performance. These metrics are:

- *Delivery Lead Time*: The elapsed time it takes to implement, test, and deliver changes to the production environment.
- *Deployment Frequency*: How often you deploy changes to the production environment.
- *Change Fail Percentage*: What percentage of changes either cause and impaired service or need immediate correction (hotfix, rollback, fix forward, etc.)
- *Mean Time to Recovery*: How long does it take to restore a service when there is an unplanned outage or impairment.

Define Everything as Code

This principle can be read about in detail in the book in Chapter 4.

Core Practice: Define Everything as Code

Defining all your stuff as code is a core practice for making changes rapidly and reliably. Some concepts that support this is:

- *Reusability*: If you define a thing as code, you can create many instances of it. You can repair and rebuild things quickly, and other people can build identical instances of the thing.
- *Consistency*: Things built from code are built the same way every time. This makes system behaviour predictable, makes testing more reliable, and enables continuous testing and delivery.
- *Transparency*: Everyone can see how the thing is built by looking at the code. People can review the code and suggest improvements. They can learn things to use in other code, gain insight to use when troubleshooting, and review and audit for compliance.

Why Define Everything as Code?
The motivation for defining everything as code is to make it easier to change things. The more things you define as code, the simpler the change process gets. Implementing and managing your systems as code enables you to leverage speed and improve quality.

What Can You Define as Code?

Infrastructure code specifies both the infrastructure elements you want and how you want them configured. It can be used to define:

- *Infrastructure Stack*: Collection of elements provisioned from an infrastructure cloud platform.
- *Server Configuration*: Packages, files, user accounts, services, etc.
- *Server Role*: A server's role in a collection of server elements that are applied together in single server instance.
- *Server Image*: A server image definintion that generates an image for building mulitple server instances.
- *Application Package*: How to build a deployable application artifact, including containers.

- *Configuration and Scripts*: To deliver services, which include pipelines and deployment, or to configure operations services, such as monitoring and logging.
- *Validation Rules*: Automated tests and compliance rules that can be applied to infrastructure and applications.

Version Control
If you're defining your stuff as code, then putting that code into a version control system (VCS) is simple and powerful. By doing this, you get:

- *Traceability*: You can see who changed what, when, and why.
- *Rollback*: You can revert to a previous version of the code if something breaks.
- *Correlation*: Keeping script, specifications, and configuration in version control helps when tracing and fixing gnarly problems.
- *Visibility*: You can see the history of changes to the code, providing situational awareness.
- *Actionability*: You can use the code history to trigger an automatic action for each change committed, triggers enable CI jobs and CD pipelines.

Infrastructure Coding Languages

There is a lot in the book about this, but I will not go into depth in this document, as the course focuses on HCL, which has its own section in this document (see Understanding Terraform HCL). If you want to read about this, check out the book (page 38 - 46).

The important thing to note here is the difference between *imperative* and *declarative* languages. *Imperative code* is a set of instructions that specifies how to make a thing happen. *Declarative code* specifies what you want, without specifying how to make it happen.

A big issue today is people mixing these two types of languages. This is a problem because it makes it harder to understand what the code does, and it makes it harder to change the code.

Implementation Principles

To update and evolve your infrastructure systems easily and safely, you need to keep your codebase clean: easy to understand, test, maintain, and improve.

Seperate Declarative and Imperative Code

- Code that mixes both declarative and imperative code is hard to understand and change.
- It's a design smell that suggests you should split the code into separate concerns.

Treat Infrastructure Code Like Real Code

- Often configuration files and utility scripts evolve into an unmanageable mess.
- You should treat infrastructure code like real code, with the same care and attention to detail, and the same engineering practices as application code.
- Design and manage code for infrastructure that is easy to understand and maintain.
- Follow code quality practices, such as code reviews, automated testing, and pair programming.
- Minimize *techincal debt*.
  - Technical debt is the cost of doing something quickly now, rather than doing it properly.

Code as Documentation

- Writing documentation is hard, and it's often out of date, so for some purposes, the code is the best documentation.
    - New joiners can browse the code to learn about the system.
    - Team members can read the code, and review commits, to see what other people have done and why.
    - Technical reviewers can use the code to assess what to improve.
    - Auditors can review code and version history to gain an accurate picture of the system.
- However, infrastructure code is not a substitute for documentation.
    - You should still write documentation that explains the system's purpose, how it works, and how to use it.
    - You can automatically generate useful material like architecture diagrams and API documentation from the code.

Continuously Test and Deliver All Work in Progress

This principle can be read about in detail in the book in Chapter 8.

Core Practice: Continuously Test and Deliver All Work in Progress

Effective infrastructure teams are rigorous about testing. They use automation to deploy and test each component of their system, and integrate all the work everyone has in progress. They test as they work, rather than waiting until they've finished.
The idea is to *build quality in* rather than trying to *test quality in*.
One part of this that people often overlook is that in involves integrating and testing *all work in progress*. CI involves merging and testing everyone’s code throughout development. CD takes this further, keeping the merged code always production-ready.

Why Continuous Testing and Delivery?

The motivation for continuously testing and delivering all work in progress is to make it easier to change things in the future. While initial investment in creating an automated test suite for infrastructure might seem daunting, the long term benefits heavily outweigh the upfront effort.

The traditional view of infrastructure development as a one-off activity is challenged by the reality that ongoing changes and optimizations are an integral part of maintaining a robust system. Continuous Delivery (CD) plays a crucial role in blurring the distinction between the "build" and "run" phases, emphazising the need for automated testing throughout the system's lifecycle. There will be a continuous need for patching, upgrade, fixing, and improving the system after it has been deployed.

What Continuous Testing Means

The main focus is to *build quality in*, rather than trying to *test quality in*. This means that you should test as you work, rather than waiting until you've finished. Finding problems more quickly means spending less time going back to investigate problems, and less time fixing and rewriting code. Fixing problems continuously avoids accumulating technical debt.

Immediate Testing

- Happens when you push your code

- Immediate testing is ideal, happening as you write code
  - Validation activities, such as linting, syntax checking, or running unit tests
- Pair programming is a great way to do this
  - Essentially a code review that happens as you work
  - Provides much faster feedback than code reviews that happens when you finished working on a story or feature

Eventual Testing

- Happens after some delay, perhaps after a manual review, or on a schedule

What Should You Test?

The short answer is *everything*. You should test everything that you can.

- The essence of CI is to test every change someone makes as soon as possible, and the essence of CD is to maximize the scope of that testing.
- Quality assurance is about managing the risks of applying code to your system.
- CD is about broadening the scope of risks that are immediately tested when pushing a change to the codebase, rather than waiting for eventual testing days, weeks, or even months afterwards.

Overview of things you may want to validate, whether automatically or manually:

- *Code Quality*: Is the code well structured, easy to understand, and easy to change?
- *Functionality*: Does the code do what it's supposed to do?
- *Security*: Ensure that the code is secure and that it doesn't introduce security vulnerabilities.
- *Compliance*: Systems may need to comply with regulations, industry strandards, contractual obligations, or organizational policies.
- *Performance*: Automated tools can test how quickly specific operations run, and how much resources they consume.
- *Scalability*: Create tests to prove that scaling works correctly.
- *Availability*: Automated testing can prove that your system would be available and resilient of potential outages.
- *Operability*: Automatically test any other system requirements needed for operations, such as monitoring, logging, and alerting.

Challenges with Testing Infrastructure

I heavily suggest reading the book for this section, as it's quite long (spans from page 110-115), but below follow a summary of the challenges with testing infrastructure.

- **Tests for Declarative Code Often Have Low Value**
  - Declarative code typically declares the desired state for some infrastructure
    - A suite of low-level tests of declarative code can become a bookkeeping exercise.
    - Everytime the code is changed, you need to change the test to match
- **Testing Infrastructure Code Is Slow**
  - To test infrastructure code, you need to apply it to relevant infrastructure. And provisioning an instance of infrastructure is often slow, especially when you need to create it on a cloud platform.
  - Solutions for this include a number of strategies:

        - Divide infrastructure into more tractable pieces
        - Clarify, minimize, and isolate dependencies
        - Progressive Testing
        - Choice of ephemeral or persistent instances
        - Online and offline tests
- **Dependencies Complicate Testing Infrastructure**
    - The time needed to set up other infrastructure that your code depends on makes testing even slower.
        - You can use stubs and mocks to simulate dependencies
        - There is a growing number of tools that allow you to mock the APIs of cloud vendors.
        - These won't tell you if your network structure is working correctly, but they should tell you whether they're roughly valid

Testing

There are a few different ways to test infrastructure code. The book goes into quite the detail about this, but this is just a summary of the different methods mentioned. For more detail, please review the book (p. 115 - 127).

- **Progressive Testing**
    - Running test suites in a sequence, starting with the fastest and simplest tests, and ending with the slowest and most complex tests.
    - Models include the Test Pyramid and Swisss Cheese Model
    - The guiding principle is to get fast, accurate feedback.
        - This means running faster tests with a narrower scope and fewer dependencies first.
        - This way small errors are quickly visble so they can be fixed and retested.
    - **Test Pyramid**
        - The key idea of the test pyramid is that you should have more tests at the lower layers, which are the earlier stages in your progression, and fewer tests in the later stages.
    - **Swiss Cheese Model**
        - The idea is that a given layer of testing may have holes, like one slice of Swiss cheese, that can miss a defect or risk. But when you combine multiple layers, it looks more like a block of Swiss cheese, where no hole goes all the way through.
- **Infrastructure Delivery Pipelines**
    - A CD pipeline combines the implementation of progressive testing with the delivery of code across environments in the path to production
    - There are several pipeline stages; *Trigger*, *Activity*, *Approval*, and *Output*.
    - You can have several scopes when you are testing in a stage, such as scope for *Dependencies*, and *Components*.
    - **Delivery Pipeline Software and Services**
        - Gives a way to configure pipelines stages, and trigger stages from different actions.
        - Support any action you may need for your stages, and handle artifacts and help trace and correlate specific version of instances of code.
        - Some Piplelines system options are:
            - *Build Server*: Jenkins, Team City, Bamboo, or Github Actions
            - *CD Software*: Define each stage as part of a pipeline, and code versions and artifacts are associated with the pipeline so you can trace them forward and backward

            - *SaaS Services*: CircleCI, TravisCI, and CodeShip
            - *Cloud Platform Services*: AWS CodeBuild (CI) and AWS CodePipeline (CD), and Azure Pipelines.
            - *Source Code Repository Services*: Github Actions, GitLab CI and CD.
- **Testing in Production**
    - As systems increase in complexity and scale, the scope of risks that you can practically check for outside of production shrinks.
    - Things you cannot replicate outside of production include but are not limited to:
        - *Data*: The production system may have larger data sets than you can replicate, and will have more unexpected data and combinations.
        - *Users*: Users are fare more creative at doing strange things because of their sheer number.
        - *Traffic*: If your system has a nontrivial level of traffic, you can't replicate the number and types of activities it will regularly experience.
        - *Concurrency*: Testing tools can emulate multiple users using the system at the same time, but they can't replicate the unusual combinations of things that your users do concurrently.
    - The following can help manage risks of testing in production:
        - *Monitoring*: Effective monitoring gives confidence that you can detect problems caused by your tests so you can stop them quickly.
        - *Observability*: Gives visibility into what's happening within the system at a level of detail that helps you to investigate and fix problems quickly.
        - *Zero-Downtime Deployment*: Deploy and roll back changes quickly and seamlessly, to mitigate the risk of errors.
        - *Progressive Deployment*: Having different versions of components running in production at the same time, and gradually shifting traffic to the new version.
        - *Data Management*: Production tests should not make inapproriate changes to data or expose sensitive data.
        - *Chaos Engineering*: Lower risk in production by deliberately injecting failures and other problems into the system to test its resilience.

Build Small, Simple Pieces That You Can Change Independently

This principle can be read about in detail in the book in Chapter 15.

Core Practice: Build Small, Simple Pieces That You Can Change Independently

You struggle when your systems are large and tighty coupled. The larger the system, the harder it is to change, and the easier it is to break.
When you look at the codebase of a high-performing team, you'll see that their system is composed of small, simple pieces. Each piece is easy to understand and easy to change, as well as having clearly defined interfaces. Each component can independently be deployed and tested (in isolation), and can be changed without affecting other components.

As a system expands, the challenges of managing changes increase, leadingn to greater complexity and risk. The resulting process hinders the system's ability to evolve, fostering techincal debt and compromising quailty. Composing systems from smaller pieces facilitates a faster rate of change without sacrificing quality.

Designing for Modularity

The goal of modularity is to make it easier and safer to make changes to a system. Some ways to achieve modularity is removing duplication of implementation, and simplifying implementation by providing components that can be assembled in different ways. You can also acheive modularity by ensuring that changes made to a smaller component will have a limited impact on other components.

Characteristics of Well-Designed Components

Designing components involves the art of deciding how to group or separate elements within a system, with a focus on understanding their relationships and dependencies. Two crucial design characteristics are coupling and cohesion, where the goal is to achieve low coupling (minimizing the impact of changes between components) and high cohesion (ensuring strong relationships within a component).

Coupling measures how often a change in one component requires a change in another, with the aim of achieving low or loose coupling for flexibility and ease of change.

Cohesion describes the internal relationships within a component, emphasizing that components with high cohesion are easier to change due to their smaller, simpler, and cleaner nature.

Four Rules of Simple Design:

- Pass its tests (do what it is supposed to do)
- Reveal its intention (be clear and easy to understand)
- Have no duplication
- Include the fewest elements possible

Rules for Designing Components

Avoid Duplication

- DRY (Don't Repeat Yourself) is a principle that states that every piece of knowledge should have a single, unambiguous, authoritative representation within a system.
- Reuse increases coupling so a good rule of thumb for reuse is to be DRY within a component, and wet across components.

Rule of Composition

- Make independent pieces to have a composable system.
- It should be easy to replace one side of a dependency relationship without disrupting the other side.

Single Responsibility Principle

- Any given component should have responsibility for a single part of the system's functionality.
- This principle is about keeping each component focused, so that it's contents are cohesive and easy to understand.

Design Components around Domain Concepts, not Technical Concepts

- Building components solely around technical concepts may lead to increased coupling, as shared components tie together all the code they use.
- It is more effective to design components around domain concepts, such as application servers or build servers, as they encapsulate functionality that can be reused across various applications or teams,

promoting better modularity and reducing code coupling.

Law of Demeter

- Also known as the *Principle of Least Knowledge*, this law states that a component should only interact with its immediate dependencies, and should have no knowledge of how other components are implemented.

No Circular Dependencies

- A provider component should never consume resources from one of its own direct or indirect consumers.

Finally, use **Testing** to drive **Design Decisions**.

- Testing is emphasized as a crucial aspect of infrastructure code development, making testability an essential design consideration for infrastructure components.
- The ability to continuously test and deliver code is contingent on maintaining clean system designs characterized by loose coupling and high cohesion.
- Automated testing acts as a catalyst for better design by necessitating a well-organized and modular codebase.

Modularizing Infrastructure

Whilst being a large part of the chapter, I believe this part mostly consists of **how** to modularize your infrastructure, and not the concepts around it, so I will not go into detail about this. If you want to read more about it, please refer to the book (p. 256 - 270).

https://aidanfinn.com/?p=22549 (Add pros and cons of modularizing infrastructure)

Benefits of Modularization

- You write less code because the code is written once and can be reused.
- Code is standardised, and you can go from one workload or client to another, and know how the code works.
- Governance is built into the code, such as naming conventions, tagging, and security.
- Smaller code is easier to troubleshoot and understand.
- Breaking your code into smaller modules makes collaboration easier.

Challenges of Modularization

- No matter how well you write a module, it will always require updates.
    - New code means new versions.
- Trying to create a one-size-fits-all module is hard.
- The code length is compounded by code complexity.
- You will have to create a code release and versioning system that must be maintained.

Drawing Boundaries Between Components

The following is listed when it comes to drawing boundaries between components:

> To divide infrastructure, as with any system, you should look for seams. A *seam* is a place where you can alter behavior in your system without editing in that place. The idea is to find natural places to draw boundaries between parts of your systems,where you can create simple, clean integration points.

Align Boundaries with Natural Change Patterns

- **Natural Change Patterns**: Optimize component boundaries by understanding their natural patterns of change, treating these patterns as seams or natural boundaries.
- **Learn from Historical Changes**: Analyze historical changes, especially finer-grained ones like code commits, to identify components that frequently change together. This understanding informs efforts to refactor for increased cohesion and reduced coupling.
- **Focus on Small, Frequent Changes**: While higher-level work like tickets or stories can provide insights into system changes, prioritize understanding smaller, frequent changes. This approach enables incremental changes within larger change initiatives and helps identify components that can be modified independently.

Align Boundaries with Component Life Cycles

- **Diverse Life Cycles**: Acknowledge varied life cycles for different components, like dynamic server clusters and less frequently changing database storage.
- **Simplify Management**: Organize resources, particularly stacks, based on life cycles to streamline management; for example, separate stacks for application servers and database storage.
- **Risk Mitigation**: Prevent potential issues by placing elements with distinct life cycles in separate stacks, reducing the impact of updates or failures.
- **Optimized Testing**: Align stack boundaries with life cycles for efficient automated testing in pipelines, ensuring faster feedback and streamlined testing processes.
- **Cost Management**: Isolate components with challenging rebuild processes, such as data storage, into separate stacks for flexible and cost-effective infrastructure management.

Align Boundaries with Organizational Structures

- **Conway's Law Principle**: Conway's Law asserts that systems often mirror the organizational structure that creates them, with teams finding it easier to integrate and set boundaries within their owned components.
- **Implications for System Design**: Design systems with two key implications in mind - avoid creating components requiring changes from multiple teams and consider organizing teams to align with desired architectural boundaries through the "Inverse Conway Maneuver."
- **Team Structure Alignment**: In infrastructure design, aligning with the organizational structure, often organized around product lines or applications, can enhance efficiency. Even for shared infrastructure like a Database as a Service (DBaaS), designing infrastructure to manage separate instances for each team can reduce disruptions and negotiation complexities.

Create Boundaries That Support Resilience

- **Resilient Deployment**: Emphasize independent deployability for components like infrastructure stacks, offering the ability to rebuild or repair in case of failure.
- **Automated Rebuilding**: Shift from manual interventions to automated rebuilding processes triggered by the same mechanisms used for changes and updates, reducing dependency on manual expertise.

- **Quick Rebuild and Recovery**: Design components with a focus on swift rebuilding and recovery, especially by organizing resources based on their life cycle considerations.
- **Example of Efficient Design**: The example of splitting infrastructure with persistent data illustrates efficient rebuilding processes, incorporating automated steps for data saving and loading.
- **Optimizing Recovery**: Division of infrastructure into components aligned with their rebuild process simplifies and optimizes recovery efforts, further supported by redundancy strategies and running multiple instances of infrastructure parts.

Create Boundaries That Support Scaling

- **Scaling Strategies**: Scaling systems often involve creating additional instances of components, adding them during peak demand, and considering deployment in different geographical regions.
- **Cloud Platform Automation**: Cloud platforms can automatically scale server clusters based on load changes, with serverless computing executing instances of code only when necessary.
- **Identifying Scaling Bottlenecks**: While server clusters can scale automatically, other elements like databases and storage devices may become bottlenecks. Parts of the software system can also present scaling challenges.
- **Considerations Beyond Infrastructure**: Scaling strategies should account for potential bottlenecks in both infrastructure components, like databases, and other parts of the software system.

Align Boundaries to Security and Governance Concerns

- **Security, Compliance, Governance Focus**: Prioritize security, compliance, and governance to protect data and ensure service availability, with different parts of the system adhering to specific rules and standards.
- **Regulatory Infrastructure Division**: Structure infrastructure based on applicable regulations and policies, providing clarity for implementing measures specific to each component.
- **Tailored Change Processes**: Customize change delivery processes to align with governance requirements, enforcing reviews, approvals, and generating change reports for streamlined auditing.

Understanding Terraform HCL

Delve into how Terraform HashiCorp Language (HCL) operates, focusing on its declarative nature, idempotency, and the structuring of blocks. This knowledge is crucial for writing effective and efficient Terraform code.

About HCL

Taken from Terraform Language Documentation.

The main purpose of the Terraform language is declaring resources, which represent infrastructure objects. All other language features exist only to make the definition of resources more flexible and convenient.
A *Terraform configuration* is a complete document in the Terraform language that tells Terraform how to manage a given collection of infrastructure. A configuration can consist of multiple files and directories.

The syntax of the Terraform language consists of only a few basic elements:

```
resource "azurerm_virtual_network" "example-network" {
  name                = "example-network"
  location            = azurerm_resource_group.example.location
  resource_group_name = azurerm_resource_group.example.name
  address_space       = ["10.0.0.0/16"]

  tags = var.common_tags
}

<BLOCK TYPE> "<BLOCK LABEL>" "<BLOCK LABEL>" {
  # Block body
  <IDENTIFIER> = <EXPRESSION> # Argument
}
```

- *Blocks* are containers for other content and usually represent the configuration of some kind of object, like a resource.
  - Blocks have a *block type*, can have zero or more *labels*, and have a body that contains any number of arguments and nested blocks.
  - Most of Terraform's features are controlled by top-level blocks in a configuration file.
- *Arguments* assign a value to a name. They appear within blocks.
- *Expressions* represent a value, either literally or by referencing and combining other values. They appear as values for arguments, or within other expressions.

The terraform langauge is declarative, describing an intended goal rather than the steps to reach that goal. The ordering of blocks and the files they are organized into are generally not significant; Terraform only considers implicit and explicit relationships between resources when determining an order of operations.

Declarative Language

As mentioned in the section about Infrastrucure Coding Languages, declarative code specifies what you want, without specifying how to make it happen.

Idempotency in Terraform

Idempotency is the property of a system that ensures that if you run the same code multiple times, you get the same result.

Structuring Blocks in Terraform

As provided in the documentation about Standard Module Structure, the standard module structure is a directory with any number of files ending in **.tf** and **.tf.json** that contain Terraform configuration code. Furthermore, each directory that contains Terraform configuration must contain a file named **main.tf**, which serves as the entry point for the module.
In addition to this, each each terraform file is built from blocks, that are containers for other content and usually represent the configuration of some kind of object, like a resource.

Resource Blocks

The syntax for these can be found in this documentation about Resource Blocks
Resource blocks have two strings before the block: the resource type and the resource name.
Depending on your provider, more values may be required to create a resource.

Resources are the most important element in the Terraform language. Each resource block describes one or more infrastructure objects, such as virtual networks, compute instances, or higher-level components such as DNS records.

Data Blocks (Data Sources)

Data sources allow Terraform to use information defined outside of Terraform, defined by another separate Terraform configuration, or modified by functions.

A data source is accessed via a special kind of resource known as a data resource, declared using a **data** block:

```
data "azurerm_resources" "example" {
  resource_group_name = "example-resources"
}
```

Provider Blocks

Will de described in detail in the Terraform Providers section.

```
terraform {
  required_providers {
    azurerm = {
      source  = "hashicorp/azurerm"
      version = "=3.0.0"
    }
  }
}

provider "azurerm" {
  skip_provider_registration = true # This is only required when the User, Service
Principal, or Identity running Terraform lacks the permissions to register Azure
Resource Providers.
  features {}
}
```

Variable Blocks

The Terraform language includes a few kinds of blocks for requesting or publishing named values.

- Input Variables serve as parameters for a Terraform module, so users can customize behavior without editing the source.
- Output Values are like return values for a Terraform module.
- Local Values are a convenience feature for assigning a short name to an expression.

Variables should always have a **description**, to help users understand what they are for.
They can also have a **default** value, which will be used if no value is set when calling the module.

Input Variables

When you declare variables in the root module of your configuration, you can set their values using CLI options and environment variables.

Each input variable accepted by a module must be declared using a **variable** block:

```
variable "resource_group_name" {
  default = "myTFResourceGroup"
}
```

Output Values

Output values make information about your infrastructure available on the command line, and can expose information for other Terraform configurations to use.
Output values have several uses:

- A child module can use outputs to expose a subset of its resource attributes to a parent module.
- A root module can use outputs to print certain values in the CLI output after running **terraform apply**.
- When using remote state, root module outputs can be accessed by other configurations via a **terraform_remote_state** data source.

In addition to the **value** argument, an output can also have a **sensitive** argument, which marks the value as sensitive and thus prevents it from being shown when the plan is applied.

```
output "resource_group_id" {
  value = azurerm_resource_group.rg.id
}
```

Local Values

A local value assigns a name to an expression, so you can use the name multiple times within a module instead of repeating the expression.

```
locals {
  service_name = "forum"
  owner        = "Community Team"
}
```

Local values can be helpful to avoid repeating the same values or expressions multiple times in a configuration, but if overused they can also make a configuration hard to read by future maintainers by hiding the actual values used.

Module Blocks

Will be described in detail in the Terraform Modules section.

Terraform Best Practices

As can be read in this article, how you structure your Terraform configuration affects your workspace design and scope. Some considerations, such as the file structure of your configuration and version-control repositories, will depend on your team's needs and preferences. There are best practices that we recommend to all Terraform users that help teams develop Terraform configuration more efficiently.

Terraform Providers

Investigate the role and functionality of Terraform providers in managing resources across various service providers.
As specified in this documentation about Providers, Terraform relies on plugins called providers to interact with cloud providers, SaaS providers, and other APIs.

Configurations need to declare which providers they require so that Terraform can install and use them.

Providers Overview

Each Terraform module must declare which providers it requires, so that Terraform can install and use them.
Provider requirements are declared in a **required_providers** block.

```
terraform {
  required_providers {
    mycloud = {
      source  = "mycorp/mycloud"
      version = "~> 1.0"
    }
  }
}
```

Each provider has two identifiers:

- A unique *source address*, which is only used when requiring a provider.
- A *local name*, which is used everywhere else in a Terraform module.

What Providers Do

Each provider adds a set of resource types and/or data sources that Terraform can manage.
Every resource type is implemented by a provider; without providers, Terraform can't manage any kind of infrastructure. Most providers configure a specific infrastructure platform (either cloud or self-hosted).
Providers can also offer local utilities for tasks like generating random numbers for unique resource names.

Where Providers Come From

Providers are distributed separately from Terraform itself, and each provider has its own release cadence and version numbers.
The Terraform Registry is the main directory of publicly available Terraform providers, and hosts providers for most major infrastructure platforms.

Provider Documentation

Each provider has its own documentation, describing its resource types and their arguments.
The Terraform Registry includes documentation for a wide range of providers developed by HashiCorp, third-party vendors, and our Terraform community. Use the "Documentation" link in a provider's header to browse

its documentation.
Provider documentation in the Registry is versioned; you can use the version menu in the header to change which version you're viewing.
For details about writing, generating, and previewing provider documentation, see the provider publishing documentation.

Terraform Modules

Understand what Terraform modules are, including their benefits and best practices in modularizing infrastructure code.

Modules are containers for multiple resources that are used together. A module consists of a collection of **.tf** and/or **.tf.json** files kept together in a directory.
Modules are the main way to package and reuse resource configurations with Terraform.

Modules Overview

As your infrastructure grows in Terraform, the complexity of your Terraform codebase grows with it. There is no limit to the complexity of a single Terraform configuration, and you can keep writing and updating configuration files in a single directy, but you will encounter some problems:

- Understanding and navigating the configuration files will become increasingly difficult.
- Updating the configuration will become more risky, as an update to one section may cause unintended consequences to other parts of your configuration.
- There will be an increasing amount of duplication of similar blocks of configuration, for instance when configuring separate dev/staging/production environments, which will cause an increasing burden when updating those parts of your configuration.
- You may wish to share parts of your configuration between projects and teams, and will quickly find that cutting and pasting blocks of configuration between projects is error prone and hard to maintain.
- Engineers will need more Terraform expertise to understand and modify your configuration. This makes self-service workflows for other teams more difficult, slowing down their development.

Root Module

Every Terraform configuration has at least one module, known as its *root module*, which consists of the resources defined in the **.tf** files in the main working directory.

Child Modules

A Terraform module (usually the root module of a configuration) can call other modules to include their resources into the configuration. A module that has been called by another module is often referred to as a child module.

Child modules can be called multiple times within the same configuration, and multiple configurations can use the same child module.

```
module "servers" {
  source = "./source"

  servers = 5
}
```

What are Modules for?

Organize Configuration

- Modules make it easier to navigate, understand, and update your configuration by keeping related parts of your configuration together.

Encapsulate Configuration

- Encapsulation can help prevent unintended consequences, such as a change to one part of your configuration accidentally causing changes to other infrastructure, and reduce the chances of simple errors like using the same name for two different resources.

Re-use Configuration

- Using modules can save time and reduce costly errors by re-using configuration written either by yourself, other members of your team, or other Terraform practitioners who have published modules for you to use.
- You can also share modules that you have written with your team or the general public, giving them the benefit of your hard work.

Provide Consistency and Ensure Best Practices

- Not only does consistency make complex configurations easier to understand, it also helps to ensure that best practices are applied across all of your configuration.
- There have been many high-profile security incidents involving incorrectly secured object storage, and given the number of complex configuration options involved, it's easy to accidentally misconfigure these services.

Self Service

- Modules make your configuration easier for other teams to use. The Terraform Cloud registry lets other teams find and re-use your published and approved Terraform modules.
- You can also build and publish no-code ready modules, which let teams without Terraform expertise provision their own infrastructure that complies with your organization's standards and policies.

Using modules can help reduce these errors. For example, you might create a module to describe how all of your organization's public website buckets will be configured, and another module for private buckets used for logging applications.

If a configuration for a type of resource needs to be updated, using modules allows you to make that update in a single place and have it be applied to all cases where you use that module.

What is a Terraform Module?

- A set of Terraform configuration files in a single directory.
- Even a simple configuration with only a single directory is a module.
- When you run terraform commands directly from such a directory, it is considered the *root module* of your configuration.

A simple set can look something like this:

```
.
├── LICENSE
├── README.md
├── main.tf
├── variables.tf
├── outputs.tf
```

Or it can be more complex, like this:

```
.
├───keyvault
│   ├─main.tf
│   ├─outputs.tf
│   └─variables.tf
├───network
│   ├─main.tf
│   ├─outputs.tf
│   └─variables.tf
├───storageaccount
│   ├─main.tf
│   ├─outputs.tf
│   └─variables.tf
├───virtualmachine
│   ├─main.tf
│   ├─outputs.tf
│   └─variables.tf
├─main.tf
├─locals.tf
├─outputs.tf
├─terraform.tfvars
├─providers.tf
├─variables.tf
└─README.md
```

Module Best Practices

1. Name your provider **terraform--**.
2. Start writing your configuration with modules in mind.
   - You'll find the benefits of using modules outweigh the time it takes to make them properly, even for modestly complex Terraform configurations.
3. Use local modules to organize and encapsulate your code.
   - Organizing your configuration in terms of modules from the beginning will reduce the burden of maintaining and updating your configuration as the infrastructure grows in complexity.
4. Use the public Terraform Registry to find useful modules.
   - More quickly and confidently implement your configuration by relying on work of others to implement common infrastructure scenarios.
5. Publish and share modules with your team.

- Most infrastructure is managed by a team of people, and modules are important way that teams can work together to create and maintain infrastructure.

Terraform State File

Examine the purpose and significance of the Terraform state file and what it represents in the context of infrastructure management.

The following documentation about State, provides useful insight as to what the state file is, and what it does.

State

The configuration state is used by Terraform to map real world resources to your configuration, keep track of metadata, and to improve performance for large infrastructures.

- Stored in a local file named "terraform.tfstate" by default, but it can also be stored remotely, which works better in a team environment.
- Used to determine which changes to make to your infrastructure.
- Store bindings between objects in a remote system and resource instances declared in your configuration.

While the format of the state files are just JSON, direct file editing of the state is discouraged.
Terraform expects a one-to-one mapping between configured resource instances and remote objects.

Purpose of Terraform State

State is a necessary requirement for Terraform to function.
Terraform may be able to get away without state, but doing so would require shifting massive amounts of complexity from one place (state) to another place (the replacement concept).

Mapping to the Real World

Terraform requires some sort of database to map Terraform config to the real world.

- This makes it possible for Terraform to know what real world objects it is managing.
- It maps the configuration to real world objects with instance ID's that correlate to the declarative resources to the real world objects.

Terraform expects that each remote object is bound to only one resource instance in the configuration.

- If a remote object is bound to multiple resource instances, the mapping from configuration to the remote object in the state becomes ambiguous, and Terraform may behave unexpectedly.

Metadata

Alongside the mappings between resources and remote objects, Terraform must also track metadata such as resource dependencies.

- Terraform typically uses configuration to determine dependency order.
- However, when you delete a resource from configuration, Terraform has to know how to delete that resource from the remote system.
- Terraform retains a coy of the most recent dependencies within the state.

    - This is to ensure that Terraform can determine the correct order of destruction from the state.
- Terraform also stores other metadata for similar reasons, such as a pointer to the provider configuration that was most recently used with the resource in situations where multiple aliased providers are present.

Performance

Terraform stores a cache of the attribute values for all resources in the state. This is the most optional feature of Terraform state and is done only as a performance improvement.

- Terraform must know the current state of resources in order to effectively determine the changes that it needs to make to reach the desired state.
- Terraform can query your providers and sync latest attributes from all your resources for small infrastructures.
- For larger infrastructures, querying every resource is too slow.
    - Many cloud providers do not provide APIs to query multiple resources at once.
    - Cloud providers almost always have API rate limiting so Terraform can only request a certain number of resources in a period of time.
- Terraform stores the latest known attributes for all resources in the state.
    - If you use the **-refresh=false** flag, this cache is used instead of querying the provider.

Syncing

Terraform stores the state in a file in the current working directory where Terraform was run by default.

- When using Terraform as a team, it is important for everyone to be working with the same state.
- This ensures operations will be applied to the same remote objects.
- Remote state is then used to store the state in a remote data store such as Terraform Cloud, Amazon S3, or HashiCorp Consul.

With a fully-featured state backend, Terraform can use remote locking as a measure to avoid two or more different users accidentally running Terraform at the same time, and thus ensure that each Terraform run begins with the most recent updated state.

Analyzing Terraform Configurations

Be prepared to analyze examples of Terraform configurations, identifying their strengths and areas for improvement.

For this I suggest reading your own terraform code, and checking out what could be done better, or how it looks, to ensure you know what do expect.

There are however, some tips and tricks that can be used to analyze Terraform configurations, which can be found in this documentation about Static Analysis.

1. **Type of analysis**: There are different types of static analysis that can be performed on Terraform code, including syntax checking, code formatting, and security scanning. It's important to choose tools and methods that are appropriate for the type of analysis you want to perform.
2. **Compatibility with Terraform**: Not all static analysis tools and methods are compatible with Terraform. It's significant to decide tools and methods that are specifically designed for use with Terraform, or that have been tested and proven to work with Terraform.
3. **Integration with the development process**: It's often most effective to integrate static analysis into the development process, rather than trying to perform it as a separate step. Choose tools and methods that can be easily integrated into your existing workflows, such as by using a plugin or extension for your code editor or continuous integration tool.
4. **Accuracy and reliability**: It's important to choose tools and methods that are accurate and reliable, and that produce results that are actionable and easy to understand.
5. **Cost and resource requirements**: Consider the cost and resource requirements of different tools and methods, including any licensing fees, hardware or software requirements, and maintenance costs.

Some things that can be mentioned here to analyze Terraform configurations are:

1. **Code review**: Manually reviewing your Terraform code as part of a code review process can also help identify issues and improve the quality of your code. Using pull requests with GitLab, GitHub, Azure DevOps, and Bitbucket, allows you to get feedback on your code changes from multiple reviewers and helps ensure that your code is of high quality before it is merged into the repository.
2. **Terraform commands**: Terraform validate and Terraform Plan can be used to check the syntax and overall structure of your Terraform configuration. They can help you identify issues such as missing required fields, invalid values, and syntax errors, and can be used to ensure that your configuration is correct and well-formed.
3. **Code scanners**: Terrascan and/or Tfsec can be used to analyze your Terraform code for security vulnerabilities and best practices violations. It uses a combination of rule sets and custom checks to identify potential issues with your code and provides detailed feedback to help you fix any issues that are found.

CI/CD in Infrastructure as Code

Explore the concept of Continuous Integration and Continuous Deployment (CI/CD), focusing on its benefits and how it integrates with Infrastructure as Code practices.

This blog post explains the concept of CI/CD pipelines in a very good way.
CI/CD pipelines enhance the software delivery process by automating key stages such as building, testing, delivery and deployment.

What is a CI/CD Pipeline?

A CI/CD pipeline is used to automate software or infrastructure-as-code delivery, from source code to production.

CI stands for **Continuous Integration**, and **CD** stands for **Continuous Delivery** or **Continuous Deployment**.

- **CI** covers the build and test stages of the pipeline. Each change in code should trigger an automated build and test, allowing the developer of the code to get quick feedback.
- **CD** takes place after the code successfully passes the testing stage of the pipeline.
  - Continuous delivery refers to the automatic release to a repository after the CI stage.
  - Continuous deployment refers to the automatic deployment of the artifact that has been delivered.

Pipeline refers to a set of automated processes that are executed in a specific order, consisting of build, test, delivery, and deployment stages.

A build server (or build agent) is normally used to enable CI/CD runs. These often take the form of cloud-hosted virtual machines or containers.
When using containers, each step of the pipeline is executed in a separate container, which is then destroyed after the step is completed.

- This also enables pipelines to make full use of all the benefits containerization orchestration affords, such as resilience and scaling where required.

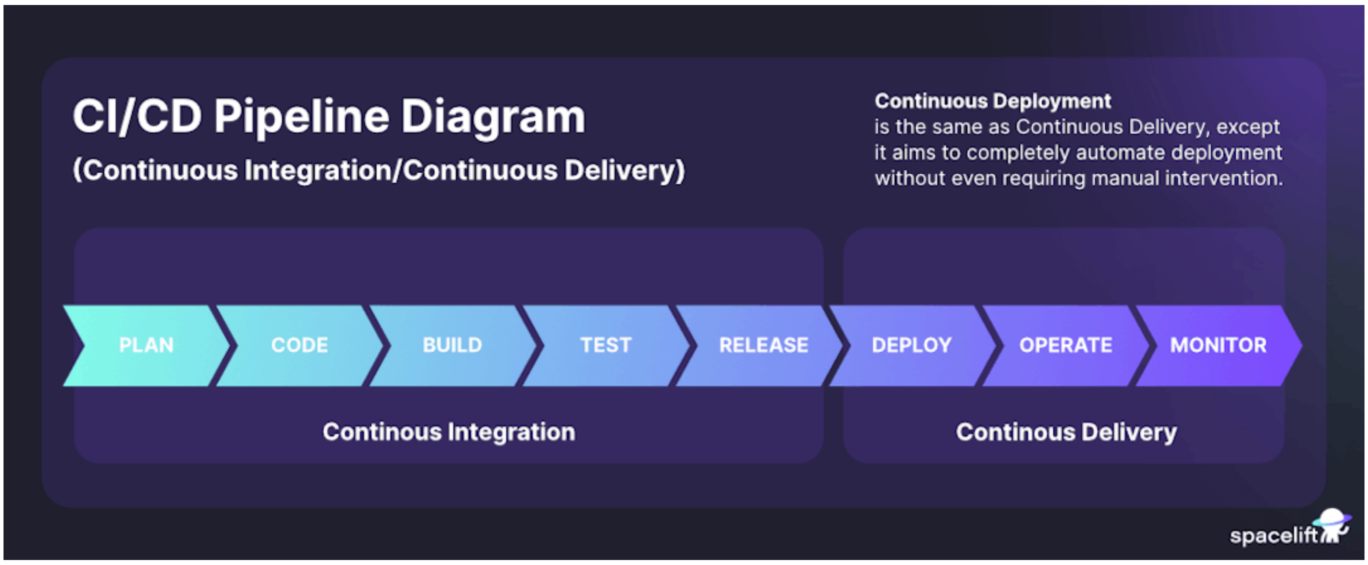
Stages of a CI/CD Pipeline

Pipeline runs are usually triggered automatically by a change in the code, but can also be run on a schedule, run manually by a user, or triggered after another pipeline has run.

The **four parts** of a CI/CD pipeline are:

- **Build**: The build stage is where the code is written, typically by multiple people working on the same codebase.
  - The artifact is a deployable version of the code that can be deployed to a production environment.
- **Test**: The test stage is where the artifact is tested to ensure that it is ready for deployment.
  - This stage can include unit tests, integration tests, end-to-end tests, smoke tests, and compliance tests.

- **Deliver**: The delivery stage is where the artifact is delivered to a repository, ready for deployment.
- **Deploy**: The deployment stage is where the artifact is deployed to a production environment.


CI/CD Pipeline Diagram
(Continuous Integration/Continuous Delivery)
Continuous Deployment is the same as Continuous Delivery, except it aims to completely automate deployment without even requiring manual intervention.
PLAN
CODE
BUILD
TEST
RELEASE
DEPLOY
OPERATE
MONITOR
Continous Integration
Continous Delivery
spacelift

CI/CD Pipeline Benefits

Adopting the use of CI/CD pipelines brings many **benefits**, including:

- **Reduced costs**: Reducing the time it takes the app or infrastructure to get coded and deployed means less human resources are taken up.
- **Reduced time to deployment**: The entire process, from coding to deployment, is streamlined, reducing the total process length and making it more efficient.
- **Enabling continuous feedback**: The pipeline provides feedback on the code, allowing developers to make changes and improvements quickly.
- **Improving team collaboration**: The team has visibility into problems, can view feedback, and make changes where appropriate.
- **Audit trails**: Each stage of the pipeline generates logs and can enable accountability and traceability.

Infrastructure as Code and CI/CD Pipelines

CI/CD can be applied to infrastructure as code (IaC) to automate the provisioning and management of infrastructure. By using a CI/CD pipeline for IaC, teams can automate the process of testing and deploying changes to their infrastructure. This can help reduce errors and downtime, as modifications are thoroughly tested before they are deployed to production.

A good CI/CD Pipeline for IaC is:

- **Fast**: The pipeline should be fast enough to allow developers to get feedback quickly.
  - Optimizing a pipeline to run as quickly as possible ensures that the developer of the code can get feedback on the success or failure of the pipeline run quickly, making them more efficient, reducing the chance of distractions and ‘context switching’.
- **Reliable**: A pipeline should run reliably each time, without any errors, or unexpected intermittent errors.
  - A reliable pipeline will produce the same result, given the same input.
- **Accurate**: The pipeline should be accurate enough to ensure that the code is tested and deployed correctly.

    - Ironing out errors in repetitive tasks is where CI/CD pipelines can really shine to improve the overall quality of a product.

GitHub and GitHub Actions

Understand how GitHub and GitHub Actions can enhance team productivity and contribute to the delivery of reliable infrastructure.

https://docs.github.com/en/actions

https://spacelift.io/blog/infrastructure-as-code-with-github-actions

https://learn.microsoft.com/en-us/devops/deliver/iac-github-actions