Table of Contents

- Examination Content
- Core Principles of Infrastructure as Code
  - Define Everything as Code
    - What Can You Define as Code?
    - Infrastructure Coding Languages
    - Implementation Principles
  - Continuously Test and Deliver All Work in Progress
    - Why Continuous Testing and Delivery?
    - Challenges with Testing Infrastructure
    - Testing
  - Build Small, Simple Pieces That You Can Change Independently
    - Designing for Modularity
    - Modularizing Infrastructure
    - Drawing Boundaries Between Components
- Understanding Terraform HCL
- Terraform Providers
- Terraform Modules
- Terraform State File
- Analyzing Terraform Configurations
- CI/CD in Infrastructure as Code
- GitHub and GitHub Actions

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.

The 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 correctoin (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. Thismakes system behaviour predictable, makes testing more reliable, and enables continouous 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 contro 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 merg- ing 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 port 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 continous need for patching, upgrand, 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 accumilating 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 tahn code reviews that happens when you finished workin on a story or feature

Eventual Testing

- Happens after some delay, perhaps after a manuel 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 visislb eso 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).

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

Terraform Providers

Terraform Modules

Terraform State File

Analyzing Terraform Configurations

CI/CD in Infrastructure as Code

GitHub and GitHub Actions