Basler Versicherungen

bluesky CONSIDER IT DONE

KARAKUN

sympany versicherungen

magnolia®

JAVA USER GROUP CH

ELCA We make it work.

Gradle

OPTRAVIS Transfer Pricing Solutions

#BaselOne21

baselone.ch

# Getting started with CI/CD Pipelines for Cloud Infrastructure

Michael Lihs

/thoughtworks

## Michael Lihs

Infrastructure Consultant
@Thoughtworks

father of 👶 and 👧

🏡 Tübingen

🚴 cycling

@kaktusmimi

## What’s on the menu today

**1. Motivation**

Failure Patterns
Maturity Levels

**2. Challenges**

Blast Radius
Incremental Deployments

**3. Prepare your Infrastructure Code for Automation**

A Checklist for Automation

**4. Design your Infrastructure Pipeline**

Pipelines that don’t suck
Sample Pipeline

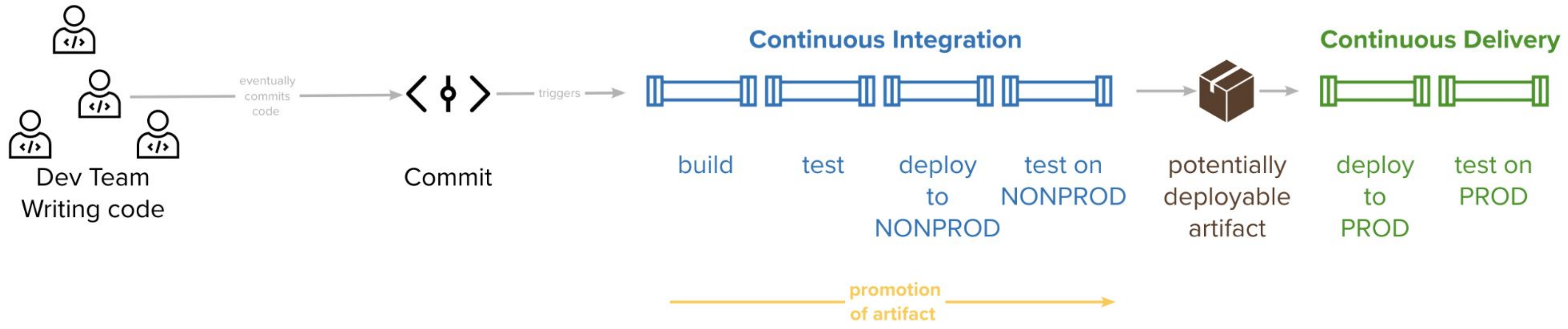
**5. Bootstrapping**

Bootstrapping
Tooling

**6. Summary**

A little cheat sheet

# CI/CD Pipelines - a Recap



## Why?

We have a situation: one of the team members applied **terraform locally** with a version of the terraform binary older than what was used in the pipeline. Due to that we ran into a **terraform state conflict** that resulted in terraform trying to **re-create all resources**

*A fellow Thougtworker*

/thoughtworks

# Infrastructure Automation Maturity Levels

**On your way to fully automated infrastructure provisioning**

- Reproducibility
- Reliability
- Traceability
- Focus on changes rather than applying them

**Web-UI-Driven**

**Script-Driven**

**Infrastructure-as-Code**

**Infrastructure Pipelines**

# Familiar Workflow

**1.**

**Write (infrastructure) Code**

**2.**

**Commit your changes**

**3.**

**Push**

**4.**

**Trigger Pipeline**

## Challenges

/thoughtworks

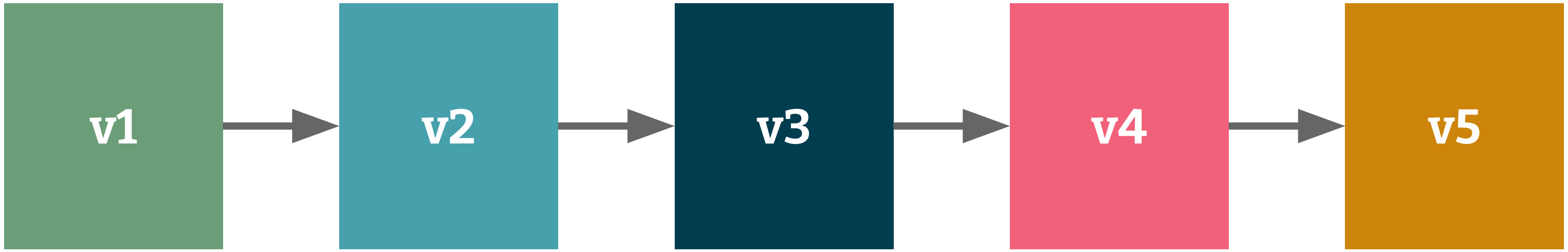
## Blast Radius

The term *blast radius* describes the potential damage a given change could make to a system. It’s usually based on the elements of the system you’re changing, what other elements depend on them, and what elements are shared.

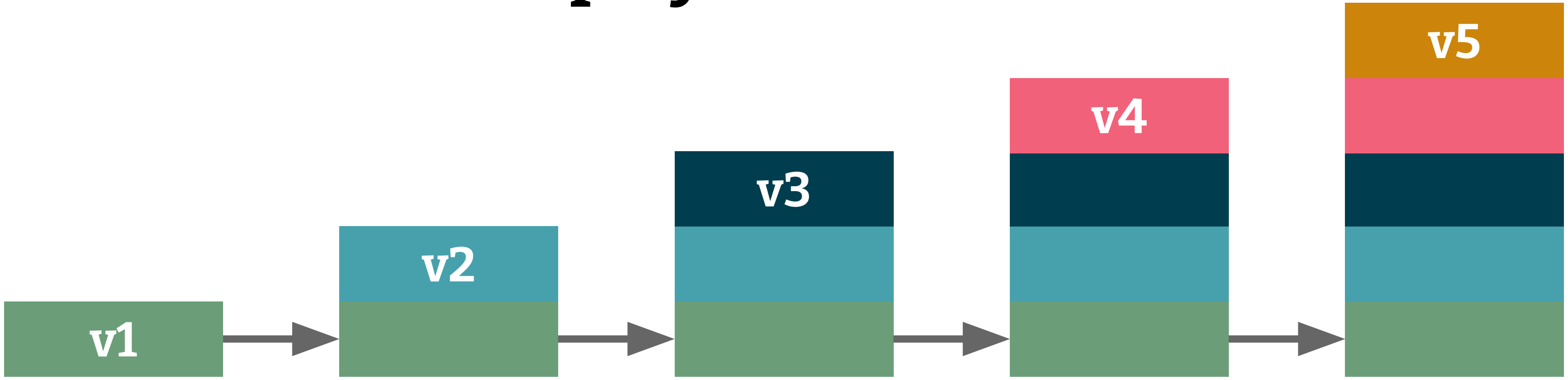
*Kief Morris, Infrastructure as Code 2nd Edition*

Image Source: commons.wikimedia.org

## Application Deployment

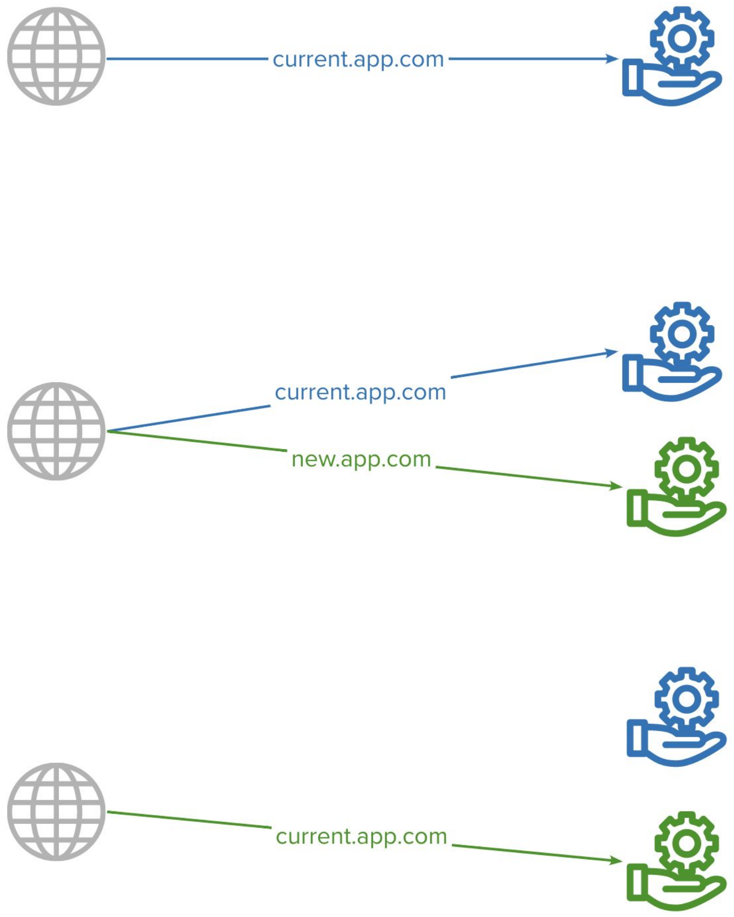


## Infrastructure Deployment



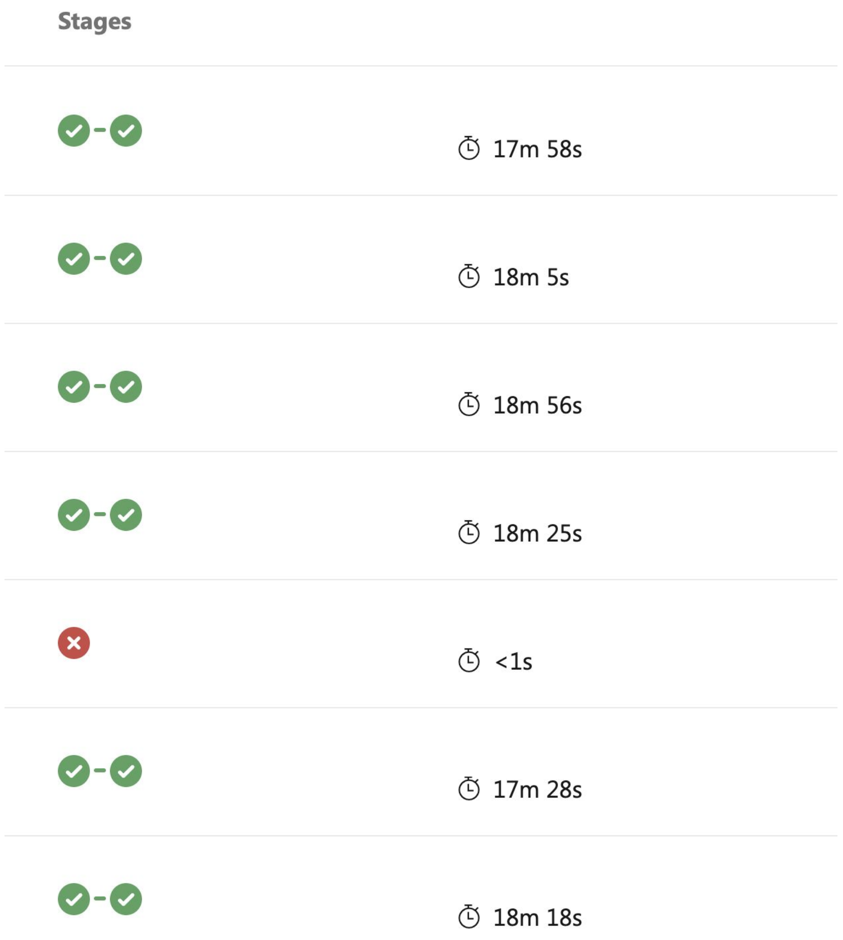
## Blue-green deployments and roll-backs

With infrastructure code, there is no easy roll-back of changes. Having infrastructure as code allows for re-creating every state of your setup - but it doesn't prevent you from potentially losing state.



## Feedback Cycles

With infrastructure pipelines we usually face long feedback loops. This easily leads to developers working around using the pipelines and can bring you into trouble if you “quickly need to fix something in production”.



| Stages | |
|---|---|
| ✓-✓ | 17m 58s |
| ✓-✓ | 18m 5s |
| ✓-✓ | 18m 56s |
| ✓-✓ | 18m 25s |
| ✗ | <1s |
| ✓-✓ | 17m 28s |
| ✓-✓ | 18m 18s |

## Prepare your Infrastructure Code for Automation

/thoughtworks

## Keep it in Version Control

Changes in the code base will be the trigger for all further steps in your pipeline.

This also holds for the pipeline itself!

```
$ git add -p
$ git commit -m "..."
$ git push origin master
```

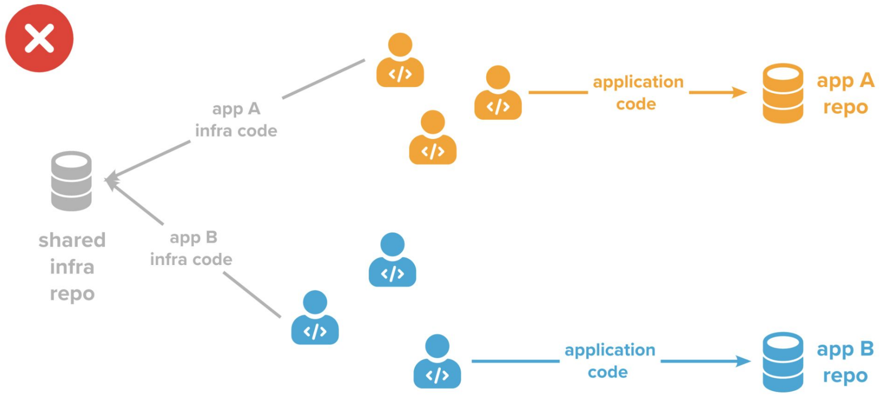
## No Secrets in your Source Code

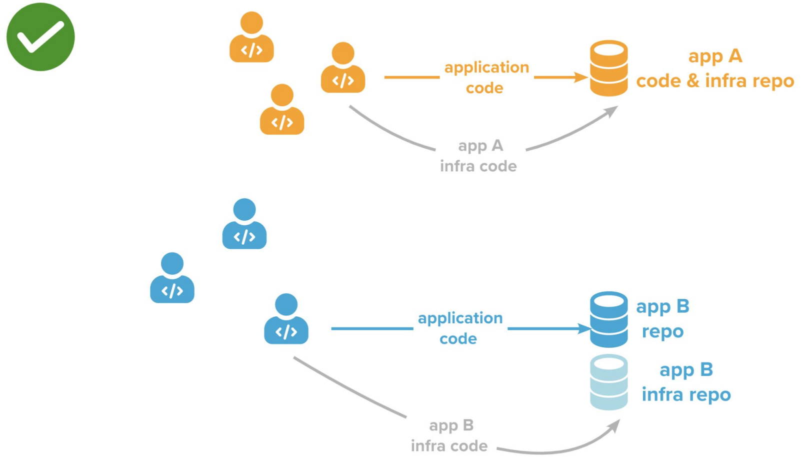
Manage your secrets in Vaults or as Pipeline Variables. Can be tricky for bootstrapping your automation.

```
$ export PULUMI_TOKEN=$(
    az keyvault secret show \
    --vault-name "mainvault" \
    --name "pulumi-access-token"
)
```

# Shape your infrastructure code around your applications

Keeping your infrastructure code together with your applications code helps you in building smaller blocks that are related to the applications and changes in either parts of the code will be easily managed.





## Modularize your Infrastructure Code

Building your infrastructure code in small reusable modules allows for re-use of code and facilitates testing.

```
infrastructure
    └── modules
        ├── aks-cluster
        ├── redis-cache
        ├── storage-accounts
        ├── users
        └── vaults
```

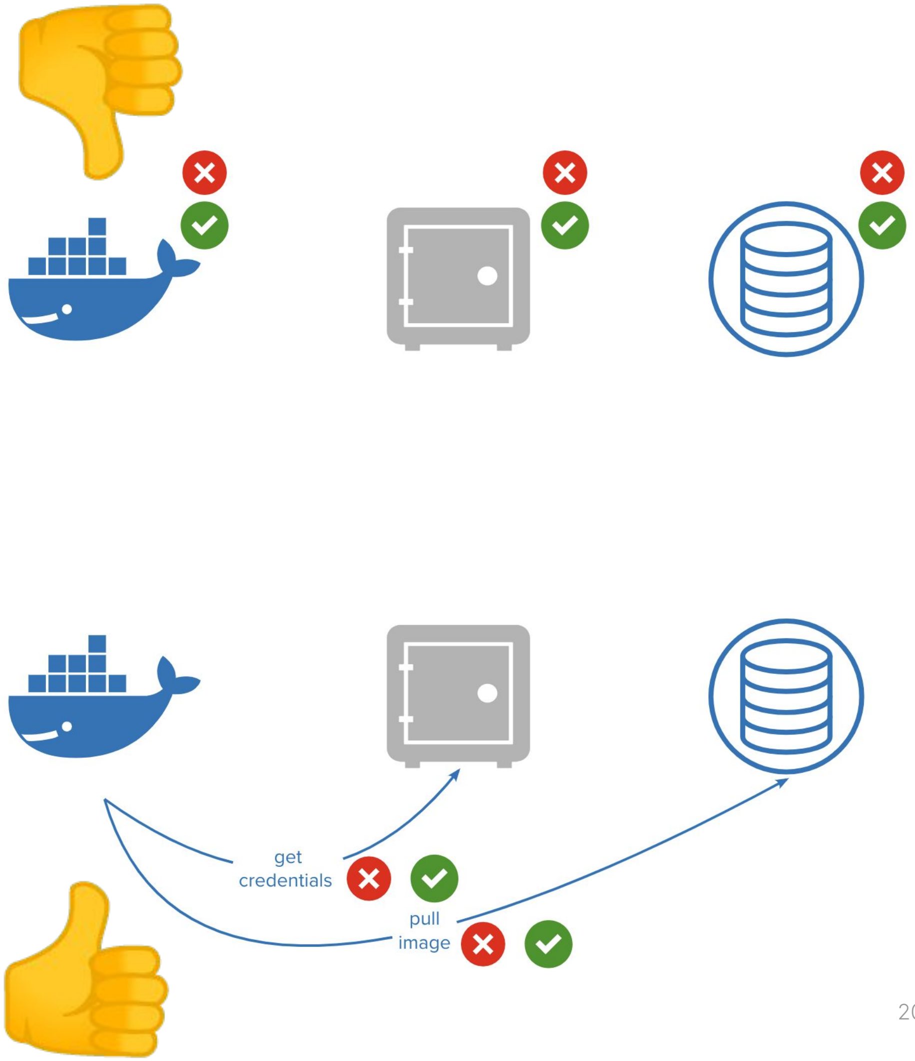
## Protect stateful resources

E.g. introduce locks for databases. Require at least 2 runs of the pipeline for a resource to be deleted.

```
resource "lock" "pgsql" {
  count        = 1
  name         = "pgsl-lock"
  scope        = pgsql_id
}
```

# Tests

Automated tests are the safety net in any automation.

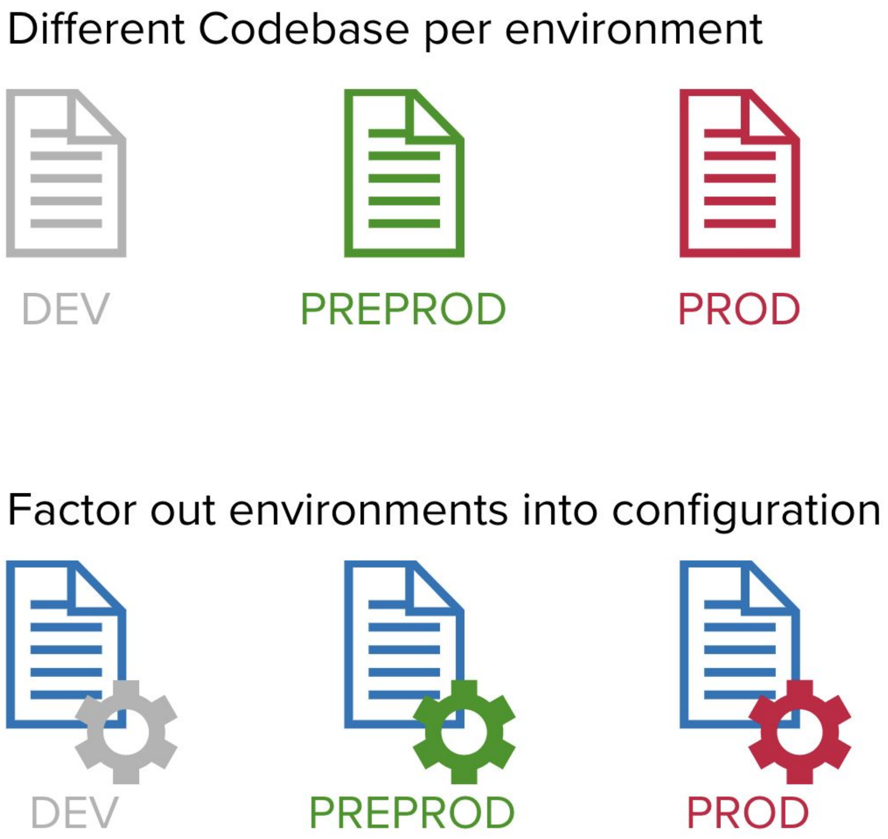


# Proper testing

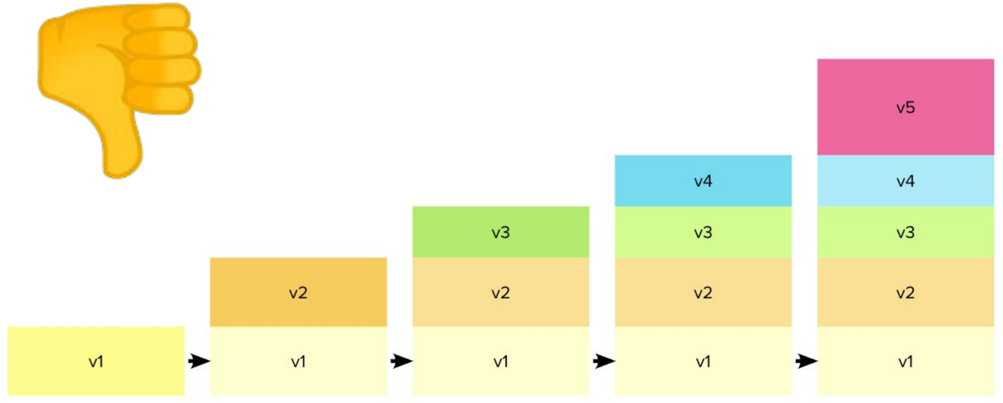
- **Avoid testing your IaC tool**
  - There is no use in checking whether a resource has been created
  - Compares to “don’t test your framework”
- **Focus on higher level flows**
  - Test the collaboration of multiple resources
  - Example: access a database with a set of credentials taken from a Vault
- **Unit Testing** (if your IaC tool allows you to use a “real programming language”)
  - Test the small pieces of glue code (i.e. mock out cloud provisioning)
  - Examples: propagation of variables, conditions, module outputs…
- **Journey tests** can be used as smoke tests in more production like environments
  - Know about a provisioning gone wrong before your user does
  - Relating bugs to changes allows for fixing things easier

# Prepare your code for Staging

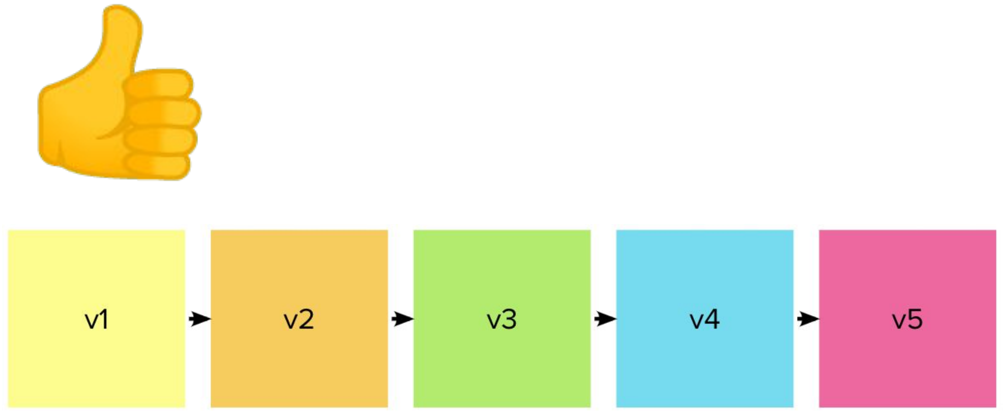
Having the same build steps across multiple environments, ensures that when changes are promoted to production, they behave as expected.


Different Codebase per environment
DEV
PREPROD
PROD
Factor out environments into configuration
DEV
PREPROD
PROD

## Immutable infrastructure

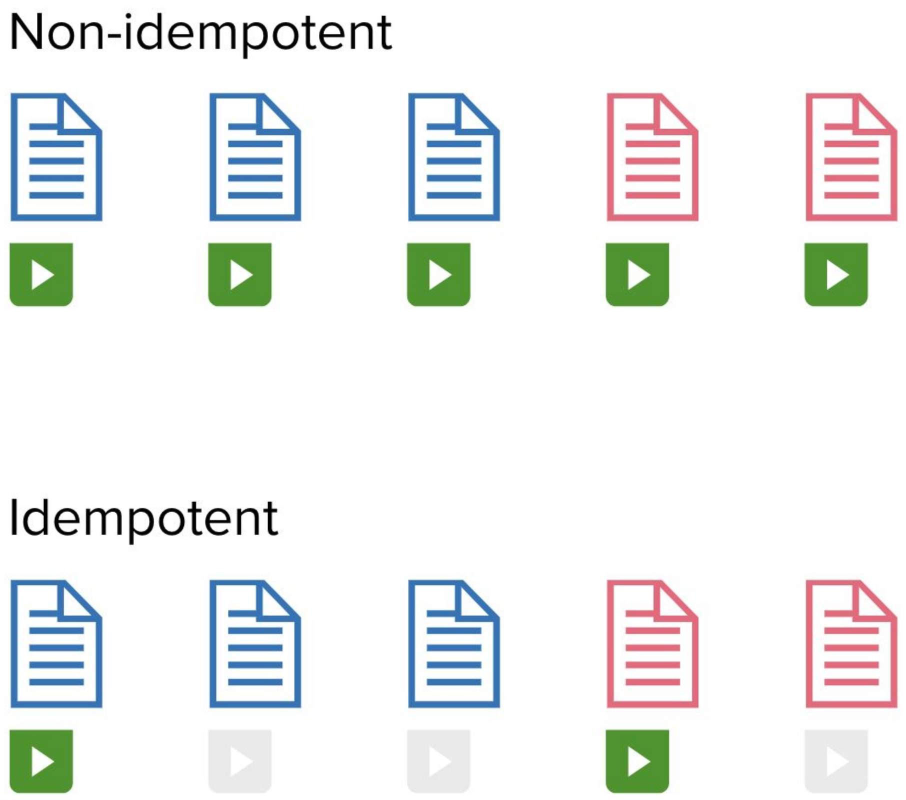


If you have the choice, try to use immutable infrastructure resources - doing so comes with the security that rebuilt infrastructure will always be the same.



## Your infra code should be idempotent

No matter how many times you apply the same code, there should be no changes beyond the result of the initial application.

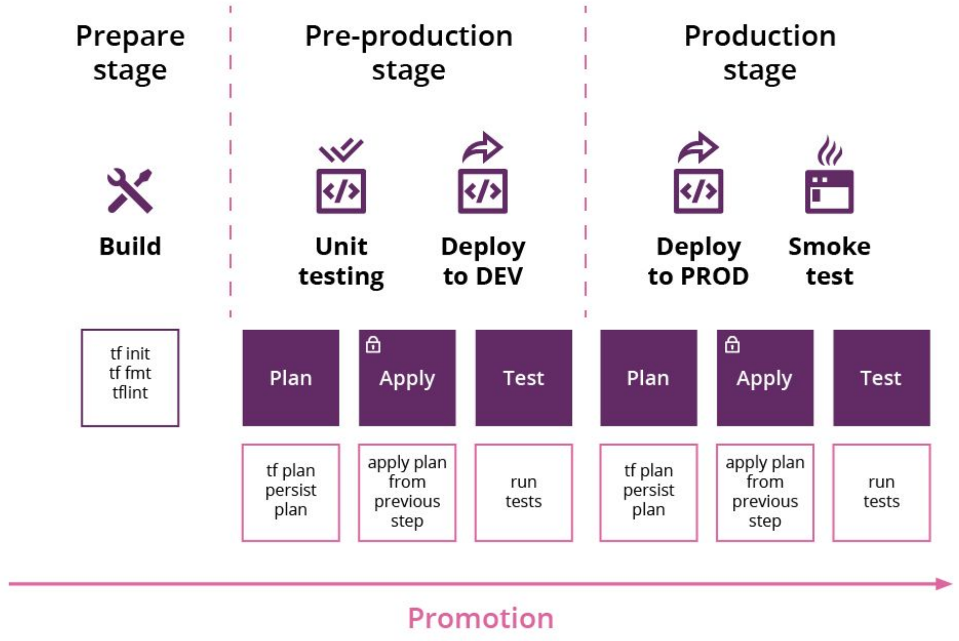


## Design your Infrastructure Pipeline

/thoughtworks

# Sample Pipeline

Click to add subtitle



## Prepare Stage

In this stage we want to **validate** our infrastructure code. The outcome of the stage is a **package** that contains all the artifacts we need for applying our infrastructure changes, e.g. validated infrastructure code.

- **Download Dependencies**
- **Check syntactical correctness**
- **Run linters & formatters**
- **Create a promotable artifact**

## Prepare stage

### Validate & package your code

In this stage we want to **validate** our infrastructure code. The outcome of the stage is a **package** that contains all the artifacts we need for applying our infrastructure changes, e.g. validated infrastructure code.

- Download Dependencies
- Check syntactical correctness
- Run linters & formatters
- Create a promotable artifact

# Pre-production Stage

**Apply, test & promote your package**

The purpose of this stage is to provision our infrastructure in a **production-like environment**. Therefore we generate a report of changes that will be made to our infrastructure, then we **apply** these changes and finally we can **test** our (new) infrastructure.

- Plan changes
- (potentially manual) Approval
- Apply changes
- Run tests

# Production Stage

## Apply, test & promote your package

We **repeat the exact same steps** that we ran in the previous stage - but in our production environment - with the added safety of having run them in pre-production.

- Plan changes
- (potentially manual) Approval
- Apply changes
- Run (smoke) tests

# Pipeline Security

A CI/CD system that can execute infra changes is a very powerful entity. Make sure only the right people can access it and that it is handled securely.

- Apply **least privilege principles** to the roles that run your pipeline.
- **Split state** and run multiple pipelines
- Use a **secure CI/CD system** (i.e. no internet-facing privately hosted Jenkins)
- Prepare for **backup and recovery**
- Implement proper **credential rotation** i.e. prepare to quickly revoke keys
- Raise **security awareness** amongst your devs (e.g. share "gone wrong stories")

## Pipelines “that don’t suck”

recommended read

**Build pipelines that don't suck**

By Mario Fernandez

**Reliable** - a reproducible process to provision our infrastructure

**Fast** - quick feedback cycles if the pipeline succeeds or fails

**Specific** - concrete feedback on what went wrong when errors happen

**Pipelines as Code** - no UI-based modifications of any kind

**Version Controlled** - ideally together with your (infrastructure) code

**Visual** - help identify issues quickly

## Trunk-based development

A development workflow that allows for continuous integration

- Infrastructure code usually does not allow for CI-ing feature branches
- Rather work in small batches in terms of change sets
- Apply them immediately
- Get fast feedback whether changes still work
- Shift feedback left e.g. via pair programming

## Bootstrapping your infrastructure pipeline

/thoughtworks

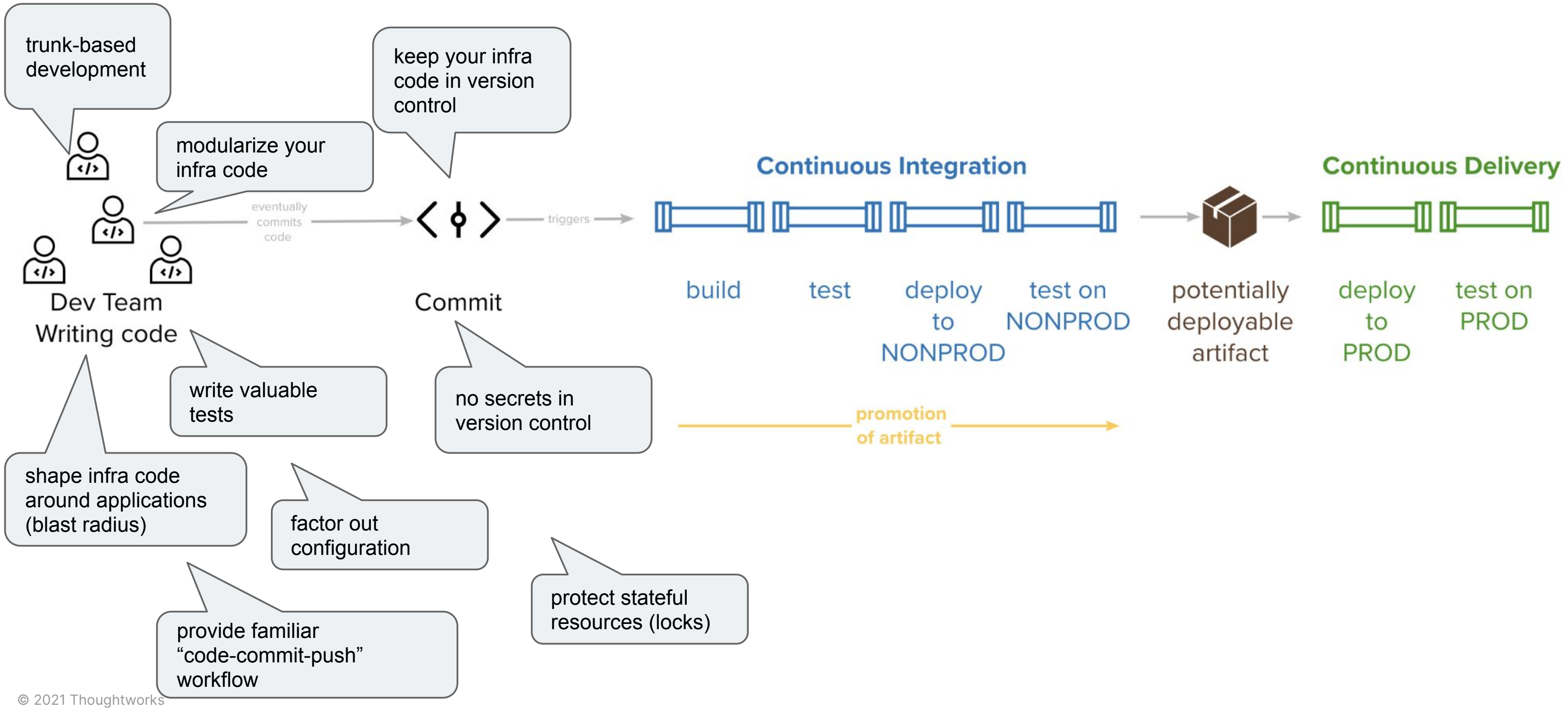
# Bootstrapping your Infrastructure Automation

```
$ ./do.sh

  Usage: ./do.sh

   bootstrap     bootstraps the PacMan infrastructure
                 automation idempotently
```

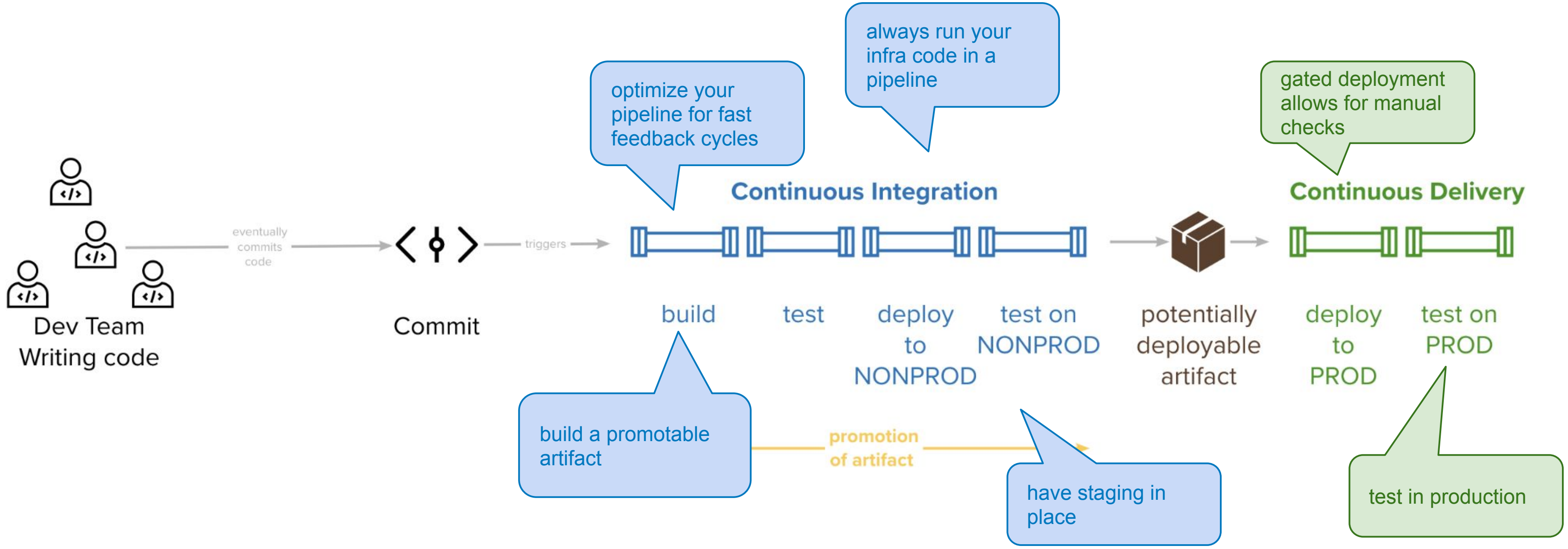
```
$ export AZURE_TECH_USER_ID='...'
$ export AZURE_TECH_USER_PASSWORD='...'
$ ./do.sh bootstrap
   ✅ create storage account for Terraform state
   ✅ create secret variables group
   ✅ create Terraform pipeline
```

## Summary

# Main Takeaways


trunk-based development
modularize your infra code
keep your infra code in version control
eventually commits code
triggers
Continuous Integration
Continuous Delivery
Dev Team Writing code
Commit
build
test
deploy to NONPROD
test on NONPROD
potentially deployable artifact
deploy to PROD
test on PROD
write valuable tests
no secrets in version control
promotion of artifact
shape infra code around applications (blast radius)
factor out configuration
protect stateful resources (locks)
provide familiar “code-commit-push” workflow

# Main Takeaways



## Thank you for your attention 👍

**Michael Lihs**
Infrastructure Consultant

michael.lihs@thoughtworks.com | @kaktusmimi

/thoughtworks

## References

- Kief Morris, Infrastructure as Code - 2nd Edition
- Alaa Mansour & Michael Lihs, Infrastructure Pipelines
- Structuring Hashicorp Terraform Configuration for Production
- Running Terraform in Automation
- Test-Driven Development for Infrastructure