軟體開發中，我們要求嚴謹的 Code workflow，有 Review 還有 release 流程

但 infra 卻沒有這些步驟：只是上去 console 點一點，把 infra 開起來，把很穩的 code，透過很穩的 CI/CD 放到沒有 review 流程的 infrastructure 上運行，然後祈禱他會很穩。結果就是

* code 好，卻跑不好
* infra 鬼故事多
  + 不改就不會壞
  + 新功能，穩定性，二選一，或是兩個都沒有

**Terraform workflow**

導入 terraform 後，團隊成員都可以各自撰寫 terraform 來描述 infrastructure，然後各自 plan 各自 apply，這樣個工作流程在小規模的 infrastructure 中是可以運作的。然而隨著團隊規模增加，infra 越來越複雜，這樣協作性低的工作流程，便會面離許多挑戰。

Terraform 官方文件也針對[團隊導入的階段做說明](https://www.terraform.io/docs/cloud/guides/recommended-practices/part3.1.html" \t "_blank)，也描述[實務上 infra 管理的困難](https://www.terraform.io/docs/cloud/guides/recommended-practices/part1.html#fundamental-challenges-in-provisioning)，這邊只簡述摘要 infrastructure 管理的兩大困難:

* 技術 infra 越來越複雜
  + 實務上會累積許多不一致（ex. infra 團隊常見：我以為是這樣，但怎麼實際 cloud 上是長這樣勒？）
  + Terraform 是簡化的工作流程抽象框架(workflow-level abstraction)，固定工作流程（init，plan，apply...）
* 團隊組織越來越複雜
  + 分工，各個團隊各自負責，但又要溝通協作
  + 平行開發，注重效率，但整合又要要避免衝突與錯誤

這些問題是所有 IaC 工具都試圖解決與優化的部分。Infrastructure as Code 不只是將 web console 上的手動點擊操作，替換成程式碼操作而已，使用 IaC 來管理 infrastructure，除了更改使用方式，還有許多延伸的優勢，例如搭配使用程式碼的管理工具，進一步提升 IaC 程式碼的整體效率，達成 infrastructure 管理的元件化，標準化與自動化。

這邊要先說明，不是自動化就是好

* 步驟標準化可以降低錯誤機率
* 自動化可以達到更高效率
* 然而，技術棧的選擇應該以團隊為本

底下我們會介紹 IaC 高效開發的範例，導入 gitflowd 開發流程，來管理 IaC，使用到的工具有：

* 版本管理工具，這邊使用 GitHub
  + 引入開發流程，ex. diff, review, PR, release
* [搭配 CI/CD 工具，大概在第十章左右會討論到](https://ithelp.ithome.com.tw/articles/10-ci-cd-automation-github.md)
* 工作流程全自動化，可以考慮 [Github Action （第十章）](https://ithelp.ithome.com.tw/articles/10-ci-cd-automation-github.md" \t "_blank) 或是 [Atlantis（第十一章）](https://ithelp.ithome.com.tw/articles/11-atlantis.md) 等工具

**Core workflow**

* [Github flow](https://guides.github.com/introduction/flow/)
* [Terraform Team Workflow](https://www.terraform.io/guides/core-workflow.html#working-as-a-team)
  + Terraform 推薦自家的 Terraform Cloud 作為範例介紹
  + 我們底下會使用 Github，事實上任何模式的 git-flow 都能套用

回顧一下 Github 具體的開發流程

* create branch
  + Write：編輯.tf 檔案
  + Plan：本地 validate, plan，檢查結果
    - pre-commit：terraform fmt
  + (Optional) Apply：最好有 dev 環境可以讓 infra 自由測試
* add commits
* Create Pull Request & Review
* Apply to stag / prod

**local branch**

本地開發

* 取得新的 feature request 後，進行 .tf 檔案的更改
* 更改完後執行 terraform plan，確定 plan 與預期相同
* commit
* 推上 remote branch

在 commit 之前，還有一些事情可以透過 pre-commit hook 處理，例如：

* 確認 .tf 都是有效的，不會有錯誤的程式碼，因為人為疏失被 push 到遠端
* 希望在 remote branch 上的 .tf coding-style 與格式都相同

上面這兩件事，terraform cli 都已內建 (都內建了，不做真的說不過去）

* terraform validate
* terraform fmt

**Pre-commit**

使用 ㄠpre-commit 工具，每個開發人原本機都需要 [安裝 pre-commit](https://pre-commit.com/#install)，然後依據 .pre-commit-config.yaml 的設定，自動安裝 pre-commit 中指定的 script

$ sudo port install pre-commit

*# brew install pre-commit*

$ pre-commit *--version*

pre-commit 2.13.0

然後使用 [gruntwork 準備的 pre-commit script](https://github.com/gruntwork-io/pre-commit" \t "_blank)，進行以下幾個 pre-commit script

repos:

- repo: https:*//github.com/gruntwork-io/pre-commit*

rev: v0.1.12 # Get the latest from: https:*//github.com/gruntwork-io/pre-commit/releases*

hooks:

- id: terraform-fmt

- id: terraform-validate

- id: tflint

- id: shellcheck

- id: gofmt

- id: golint

執行 pre-commmit install 來安裝 script 到本地

$ pre-commit install

pre-commit installed at .git/hooks/pre-commit

手動驅動執行 pre-commit run，來檢查新增的 git changes  
或是執行 pre-commit run --all-files，來檢查所有檔案

git add .

pre-commit run

pre-commit run terraform-fmt

pre-commit run --all-files

Terraform fmt............................................................Passed

Terraform validate.......................................................Passed

tflint...................................................................Passed

Shellcheck Bash Linter...................................................Passed

gofmt....................................................................Passed

golint...................................................................Passed

設定完成後，以後每次 commit 前就會自動執行，不用擔心忘記 fmt 就 commit 歪七扭八的 code

pre-commit 還可以額外增加許多功能，這些功能我們在之後會介紹，例如

* terragrunt test：使用 golang 來執行 infra 的測試
* infracost：使用 tool 來計算新的 plan 在公有雲上的費用

**Remote Branch & Pull Request**

在本地執行過基本 fmt，validate，lint 與 local test 後，我們把相對穩定的 branch 推到 Github 上。這已確定遠端的 code 有一定品質。

CI 可以接收 Branch Push event

CI 可以接收 Pull Request event，只要有 Pull Request 產生：

* 針對 git diff 部分進行驗證，ex. terraform fmt -recursive，或是更完整的測試
* 顯示新的 plan
* 直接將 Pull Request 的 infrastructure 使用 test framework 產生到公有雲上，測試
  + CI 最後再 destroy 所有測試用 resource

Pull Request 除了自動化測試以外，當然還有人工 Review

* 當然需要 Review .tf 程式碼本身
* bot / github action 可以執行 plan 結果，然後備註在 Pull Request 中，輔助 Review
* 其他如 lint，效能與執行時間，都可以顯示在 Pull Request 中

Review 是 Github flow 的核心

* IaC 的 infra 才容易做完整的 Review
* Review 的延伸意涵，是在整合團隊的程式碼，並傳遞團隊文化
  + coding style，如何提升品質，什麼品質是可以接受的...等等
* 經過完整測試 + 人工 Review 的檔案才會 merge 到 master 中

**Master Branch**

所有 master 的 .tf code 都

* 經過 apply 測試（到test framework）
* 經過人工 Review

都是穩定的程式碼，可以自動 apply 到 stag 環境，stag 上有相對穩定的 infrastructure 搭載穩定的 app code，這時讓 QA team 進駐做更詳細的測試，可以發現更細部的錯誤

* infrastructure 的變更造成錯誤
* 新 infrastructure 跟 app 版本不匹配，例如：app 需要下一版的 infrastructure 的新元件
* 效能 performance 測試，效能比起上一版本 infrastructure 是否提升
* 搭配 app 做壓力測試，app 端壓力升高時，infrastructure 是否稱得住
* 服務的可用性測試，infrastructure 是否能支撐
* ...無盡的測試

**Release**

依照產品時程準備 release candidate

* 根據 product release 的時程，cherry-pick master 的功能到 release branch
* 打上當前 release candidate tag，例如：v1.0.0-rc
* 與 app 版本搭配，進入 pre-prod 環境發布與測試
* 如果有抓出錯誤，打上 hotfit commit 修復

確定 release 版本，打上 release tag，例如：v1.0.1

* 準備 changelog
* 準備 infrastructure changelog，本次 apply 是否會影響正在線上運作的服務
* app 與 infrastructure 配合 release

**CI: Github Action**

Github Action 是 Github 支援的 CI 工具，十分方便，而且是免費使用。我們這邊直接使用 Github Action 來做，目的在教學單純化，以及可以節省成本。[drlook/terraform-github-action](https://github.com/dflook/terraform-github-actions" \t "_blank) 有非常多開源範例，這邊實際套用

tree .github

.github

└── workflows

├── lint.yaml

├── fmt.yaml

└── plan.yaml

workflow 有

* lint: 在所有 push branch 上坐 validate 與 fmt
* fmt: 在 master branch 上執行 fmt，然後自動發 PR 校正回歸到 master（可被 pre-commit fmt 取代）
* plan: 在 PR 的內容

**CD: deploy to azure**

<https://thomasthornton.cloud/2021/03/19/deploy-terraform-using-github-actions-into-azure/>

* create ad service principal for terraform
  + test terraform run with service principal
* Config Github Aciont

**CI: Terraform Cloud**

[Terraform 官方提供的 Github Action 整合說明](https://learn.hashicorp.com/tutorials/terraform/github-actions)，需要依賴 Terraform Cloud。放在這邊讓大家參考

**Github action Alternatives**

* 各種 CI 工具皆可，例如：
  + circleCI, DroneCI, travisCI

考量

* 安全性
  + Sass platform
  + on-premises / self-hosted
    - Self-hosted Github Enterprise
* 各家的功能支援程度
* 團隊熟悉程度

**References**

* <https://thomasthornton.cloud/2021/03/19/deploy-terraform-using-github-actions-into-azure/>