淺談 Cloud Native 生態系下工程師逐漸喪失的那些技能

HungWei Chiu 2024/07/03

HungWei Chi

- Hwchiu
- 個人部落格
  - https://hwchiu.com
  - https://hwchiu.medium.com
- CNTUG (Cloud Native Taiwan User Group)

Cloud Native (雲端原生)

定義

雲端原生的實踐賦予組織能夠在運算環境中（公有雲、私有雲、混合雲）以規模化、可程式化和可重複的方式開發、構建和部署工作負載，以滿足其組織需求。其特徵為鬆耦合的系統，這些系統以安全、彈性、可管理、可持續和可觀察的方式相互操作。 雲端原生技術和架構通常由容器、服務網格、多租戶、微服務、不可變基礎設施、無伺服器和聲明式 API 等組成——此列表並非全部。

雲端原生的益處

搭配穩固的自動化，雲端原生的實踐讓組織以最少的勞動、清楚分離關注點，就能頻繁、可預測地進行重大變更。

CNCF關注點

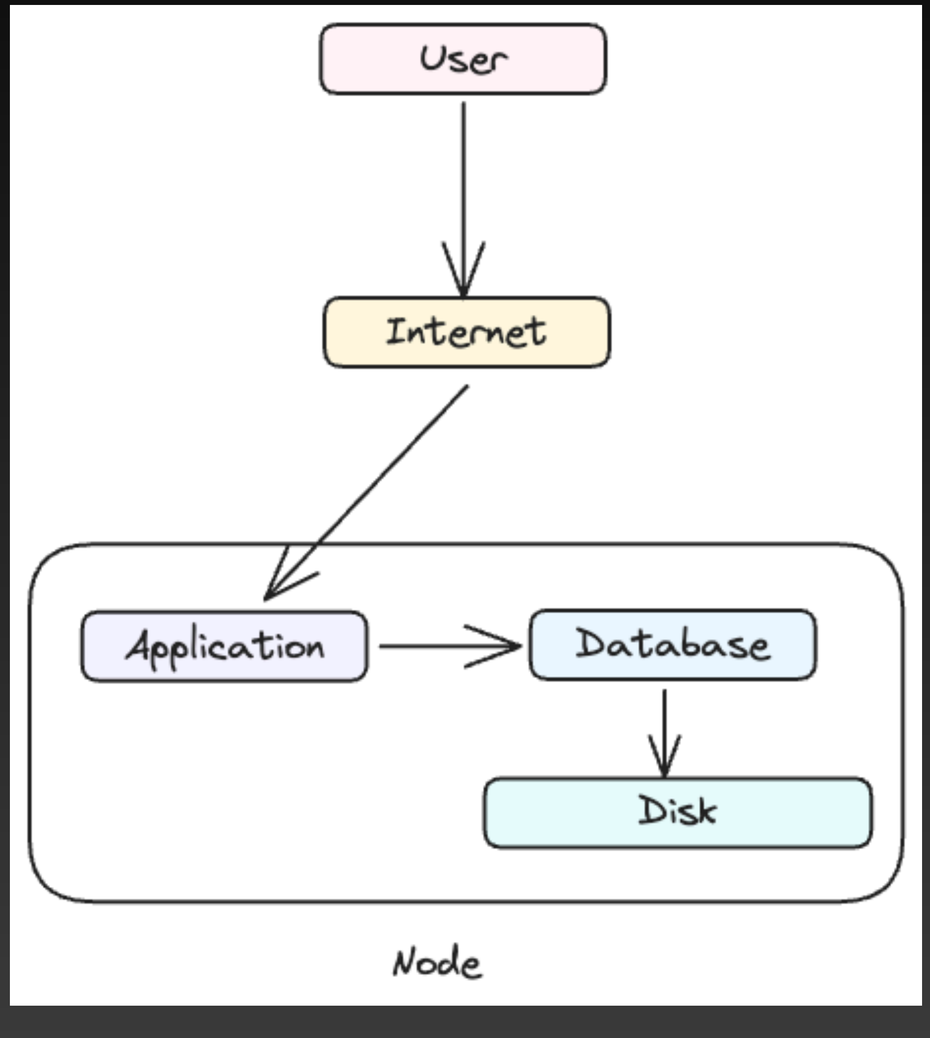
雲端原生運算基金會（Cloud Native Computing Foundation）旨在透過培育和維持一個開放原始碼、廠商中立的專案生態系統，以推動此範式的採用。我們將最先進的模式民主化，使這些創新對每個人都可用。

Cloud Native (雲端原生)

- 沒有非常清晰的定義與工具，但是大抵上脫離不了相關技術
- Cloud
- Orchestration System
  - VM
  - Container (Kubernetes)
- CI/CD
- Infrastructure as a Code
- ServiceMesh
- ...等

過去部署服務

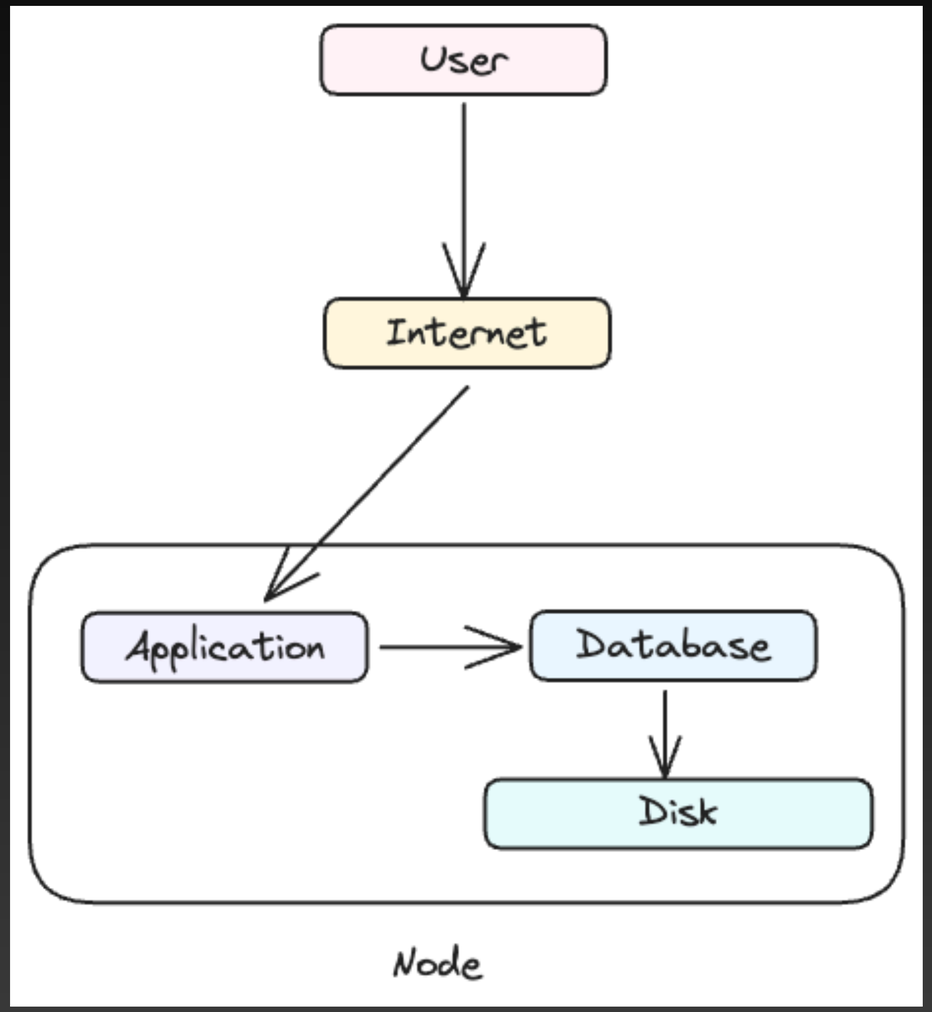
- 部署一個簡易的應用程式，需要
  - 一台機器
  - 一個應用程式
  - 一個資料庫
  - 網路設定
  - 儲存設定

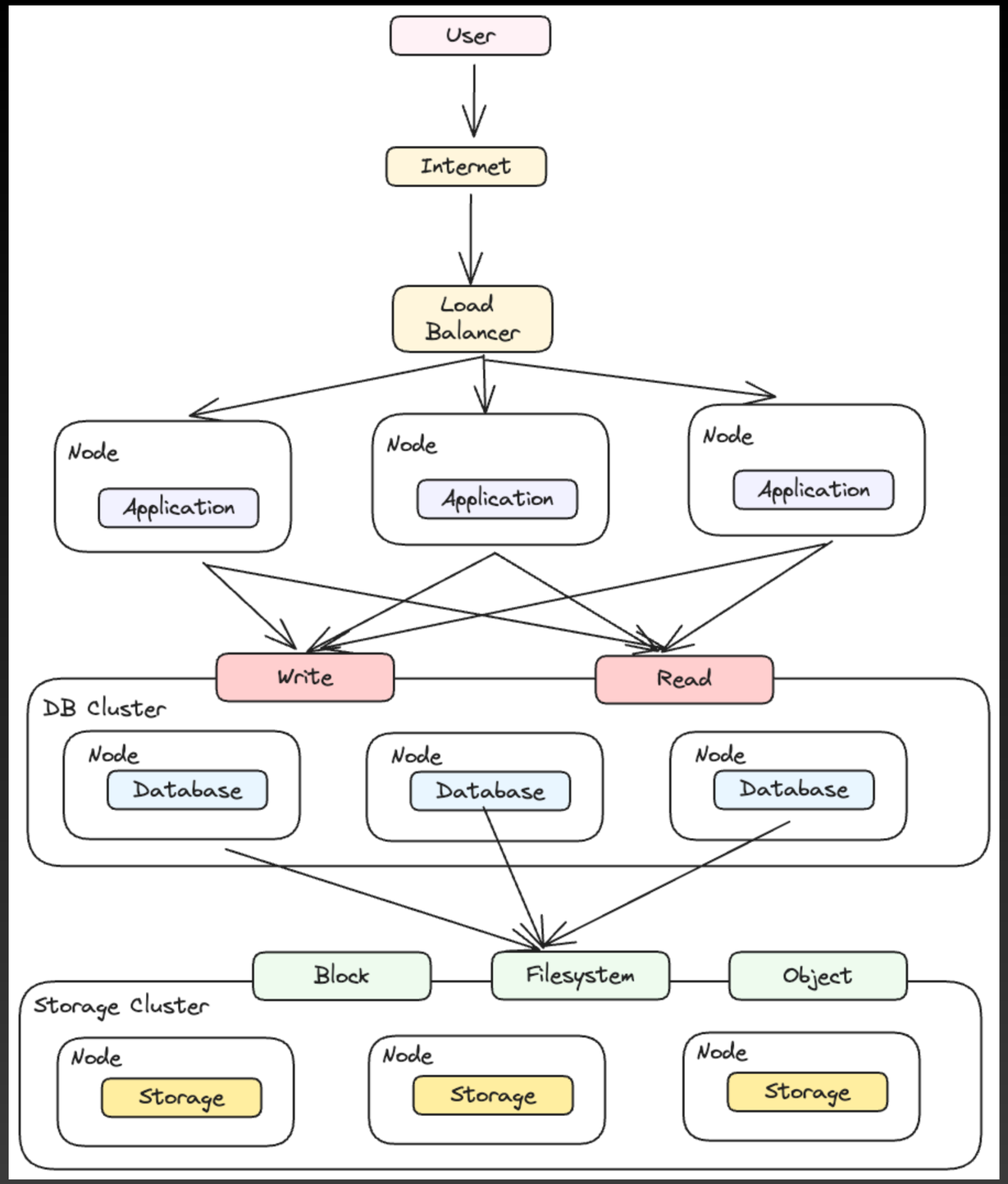


過去部署服務

- 如今的應用程式都追求
  - 高可用性
  - 避免單點故障 (Single Point Of Failure)
  - 可擴充性
    - 水平擴充
    - 垂直擴充
- 這些非業務需求都會使得架構變得龐大且複雜

過去部署服務


User
Internet
Application
Database
Disk
Node


User
Internet
Load Balancer
Node
Application
Node
Application
Node
Application
Write
Read
DB Cluster
Node
Database
Node
Database
Node
Database
Block
Filesystem
Object
Storage Cluster
Node
Storage
Node
Storage
Node
Storage

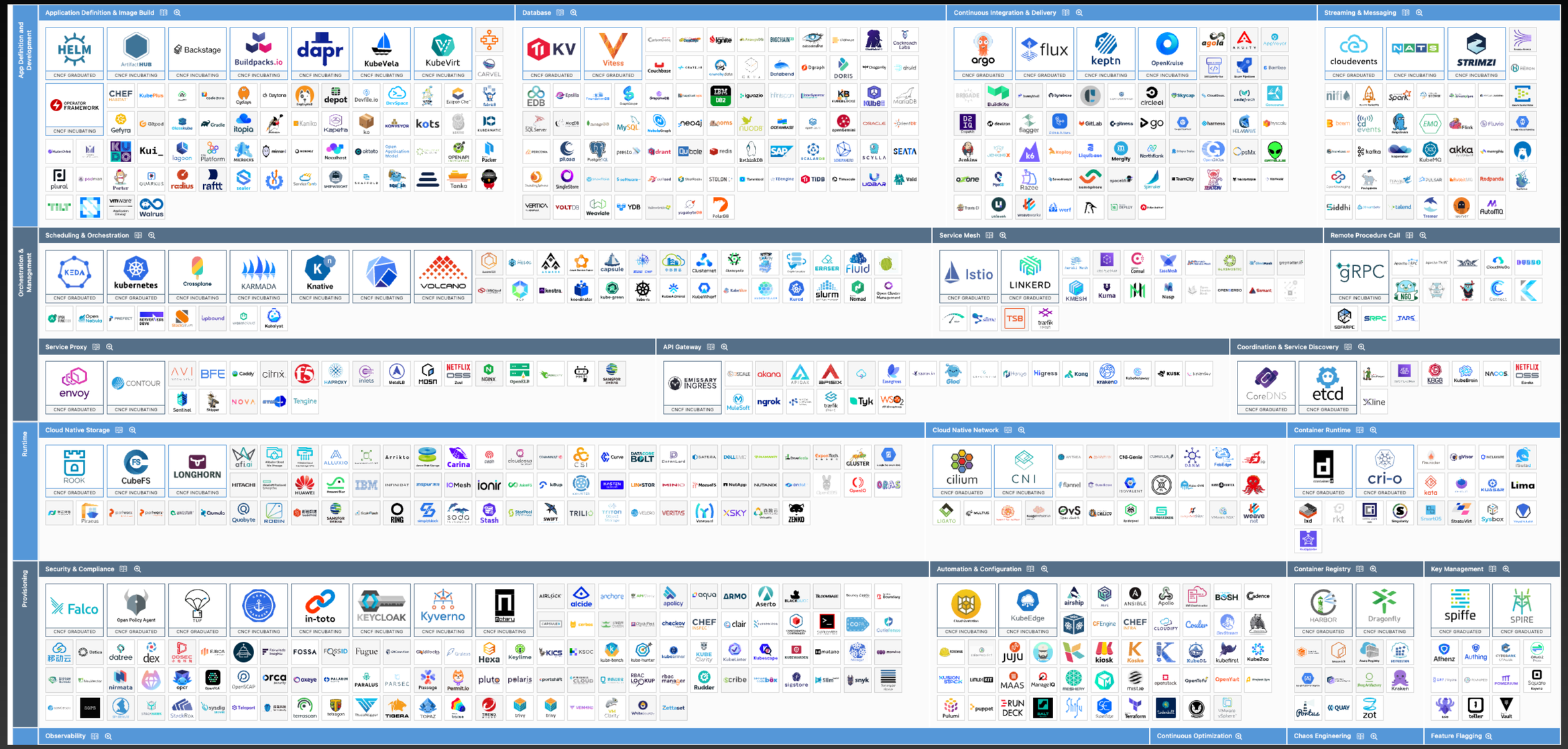
如今部署服務

- Cloud Provider 提供各種服務，簡化很多服務背後的架構，讓使用者可以專注在應用程式與業務本身
  - VM
    - 根據資源用量自動擴充
  - DB as a Service
  - VPC
  - Storage (Object, FS, Block)

Cloud Native 架構

- 隨者微服務與容器化等概念一起成長，現在愈來愈多架構都開始往 Kubernetes 平台邁進
- Kubernetes 本身也是一個簡化操作的平台，將儲存/運算/網路都給隱藏起來
- CNCF 有個著名的 landscape 來收集目前整個生態系的相關開源專案

Cloud Native 架構

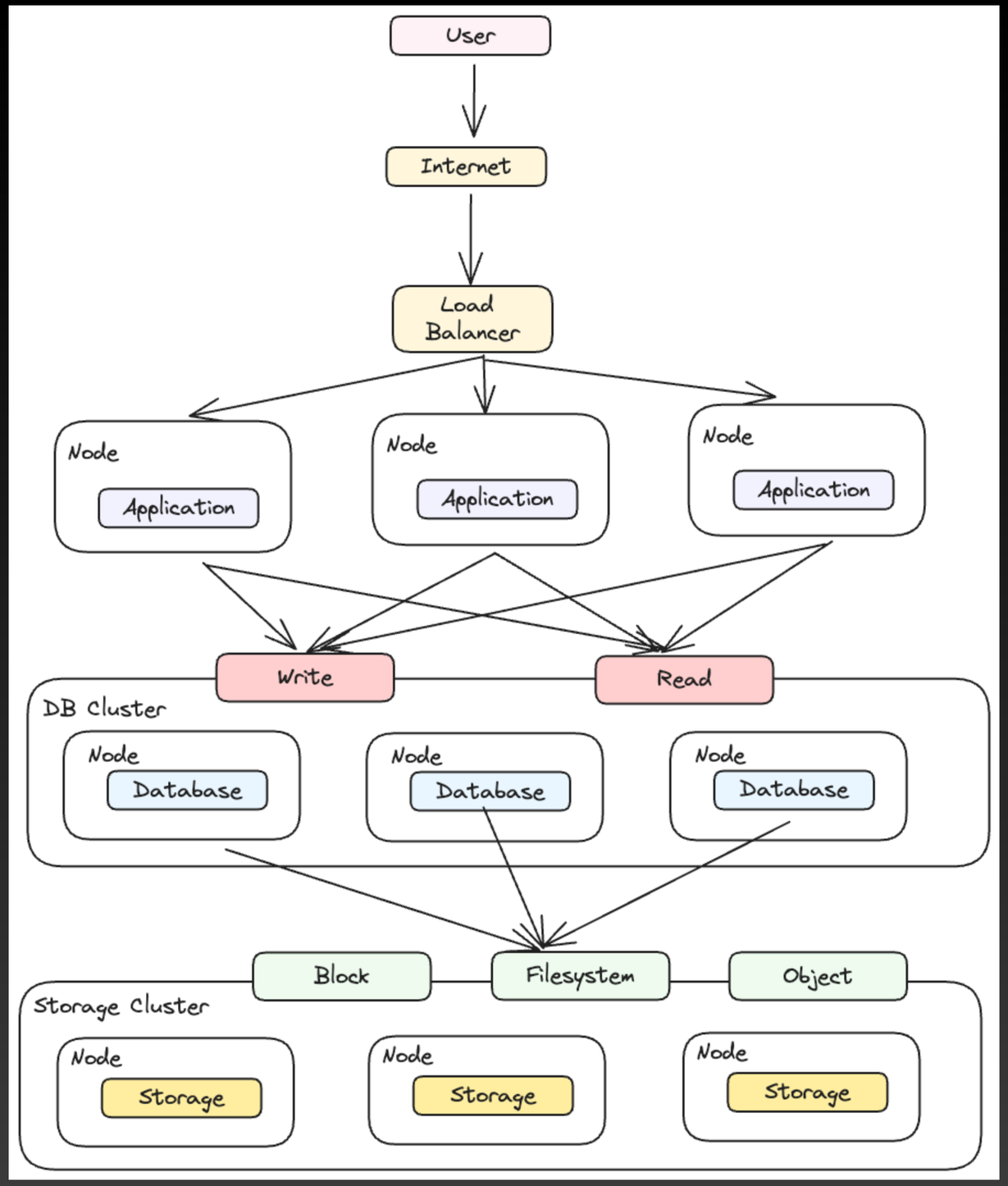


Cloud Native 架構

- 以過去的架構來說，想要把上述的所有服務都整合進去，要花非常多時間
- 設計架構
  - 處理彼此之間的網路存取
  - 處理各自的儲存空間需求
- 軟體
  - 綁定版本
  - 研究設定
  - 安裝部署

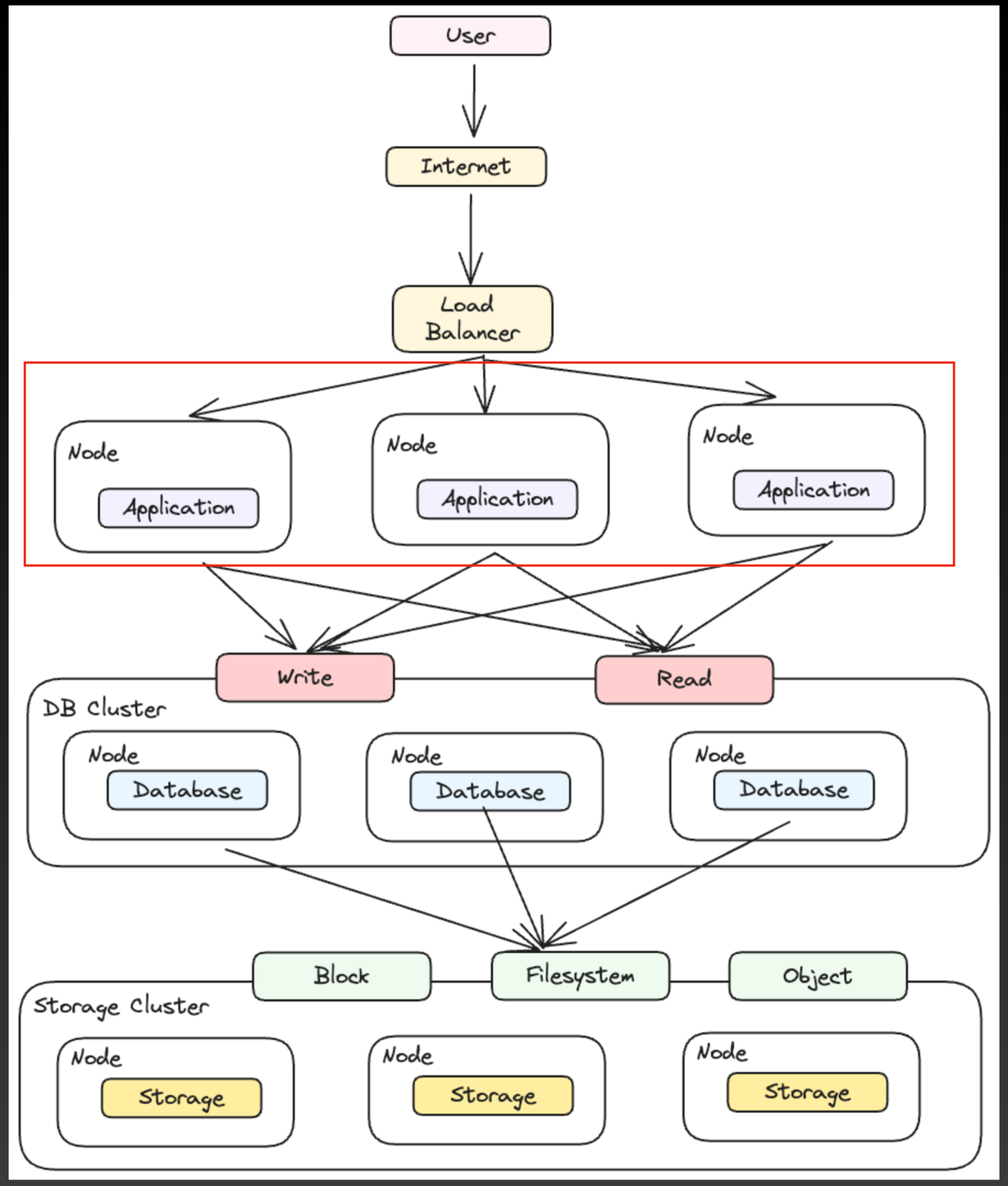
Cloud Native 架構

- 以右圖架構為範例，手工打造的步驟是什麼？



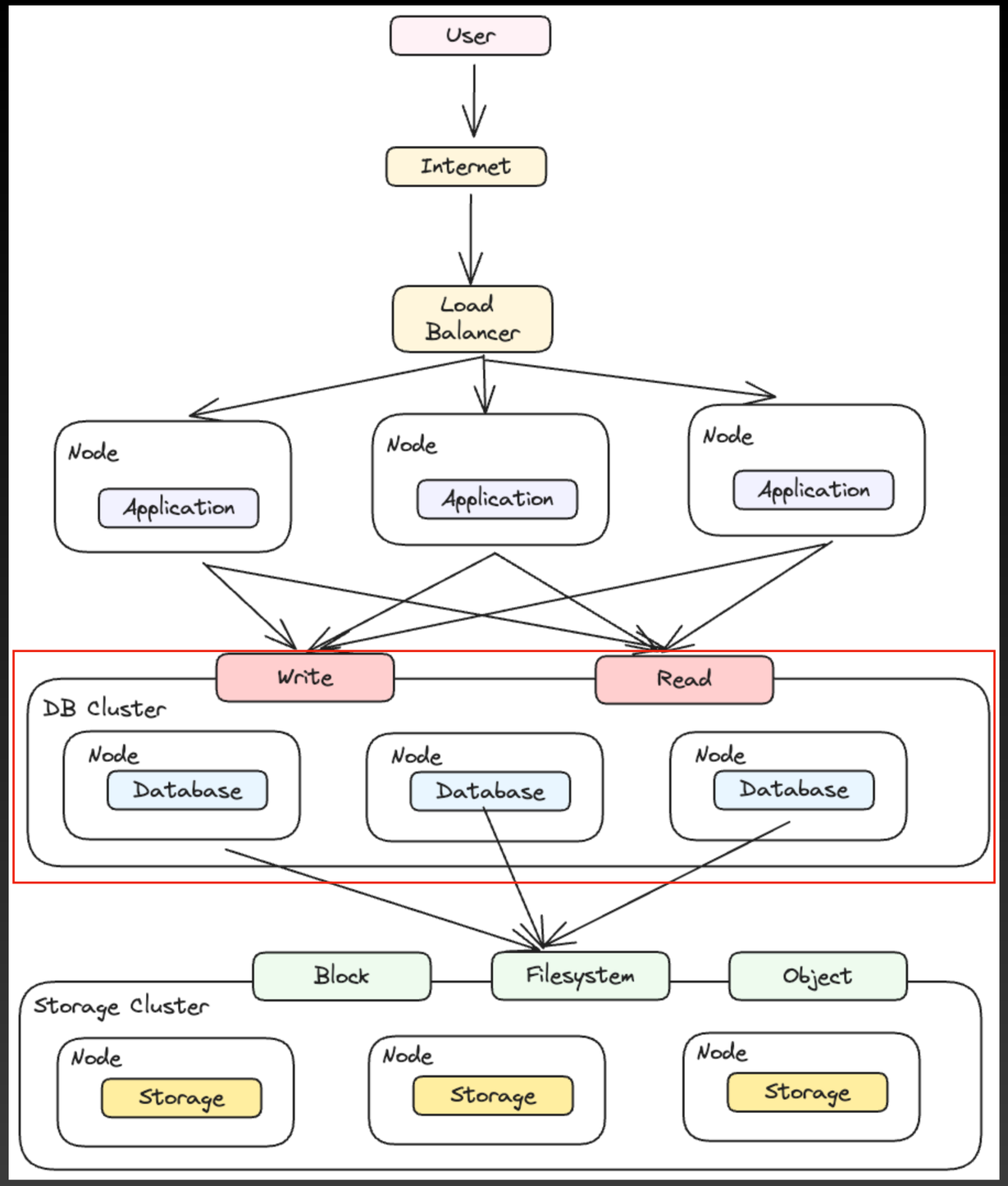
Cloud Native 架構

- Application
  - 對 OS 的基本操作要理解
  - 安裝 OS 與相關功能
  - 安裝與部署應用程式



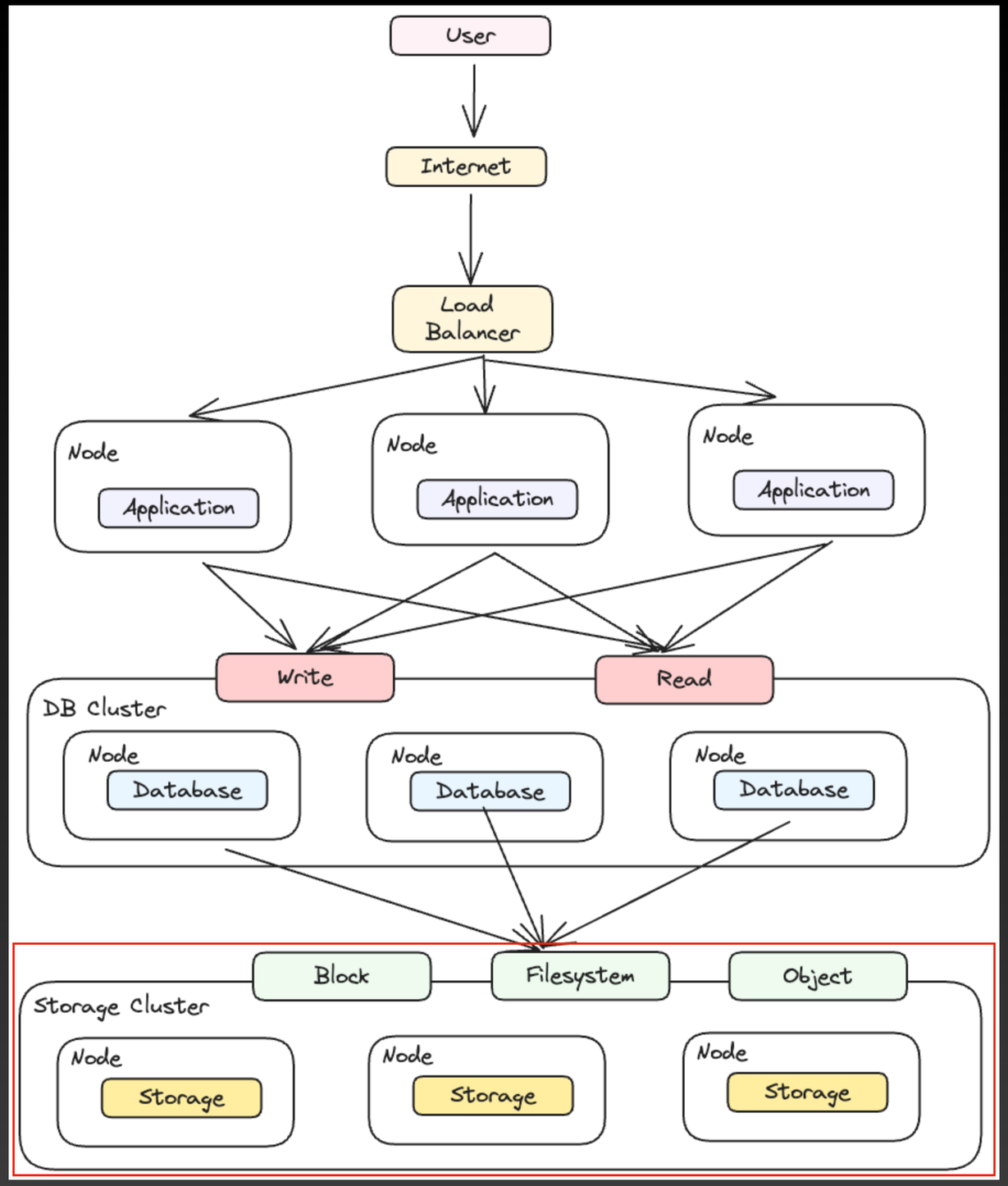
Cloud Native 架構

- Database
  - Database 的選擇障礙
  - 多節點的 DB，如何處理多讀單寫的架構
  - 如何安裝與設定
    - 不同 DB 的方式完全不同，因此驗證與測試會花很多時間



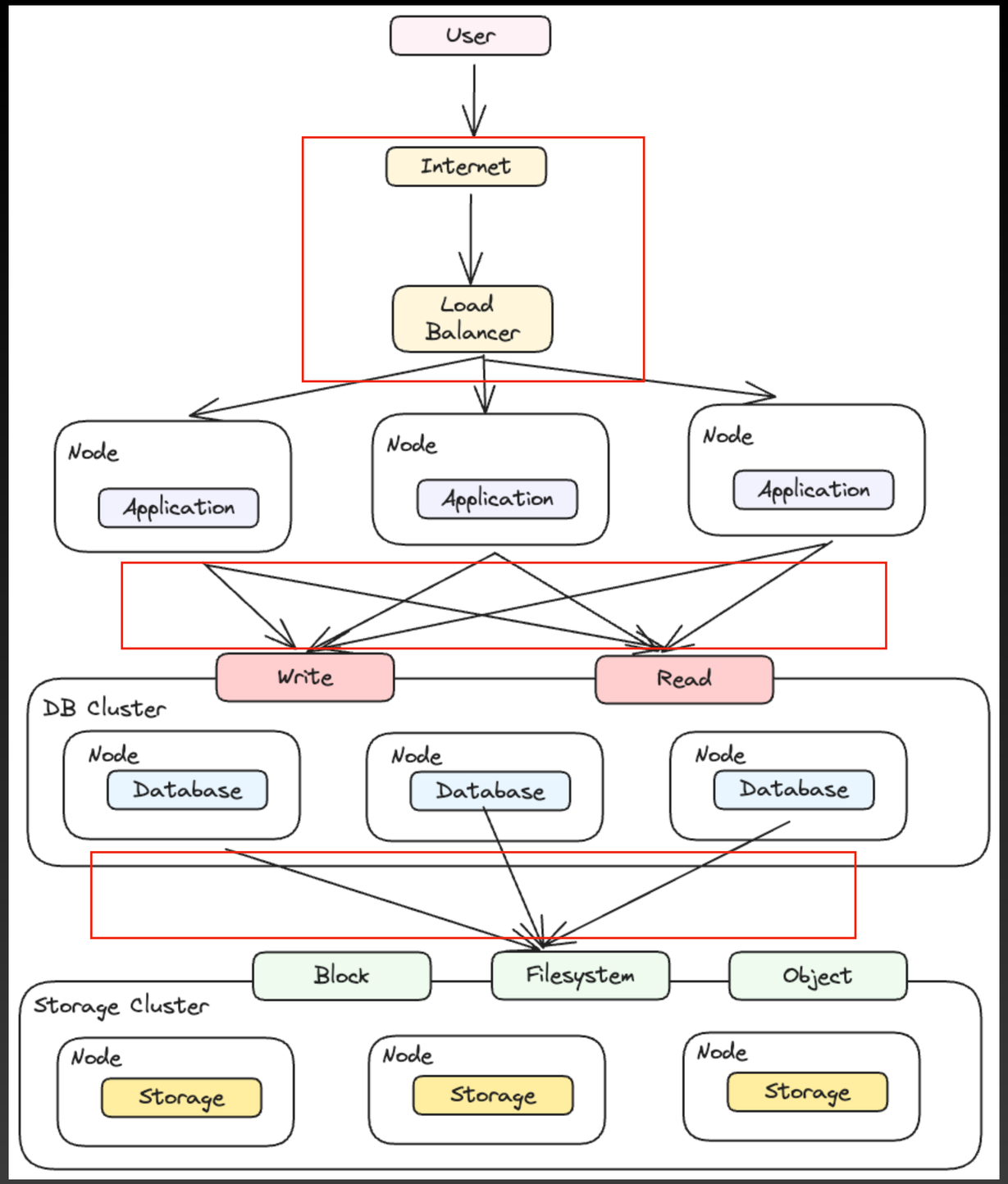
Cloud Native 架構

- Storage
  - 選擇一套儲存設備
    - 單節點
    - 分散式
  - 不同的設定與架構也不同，安裝方式曠日費時



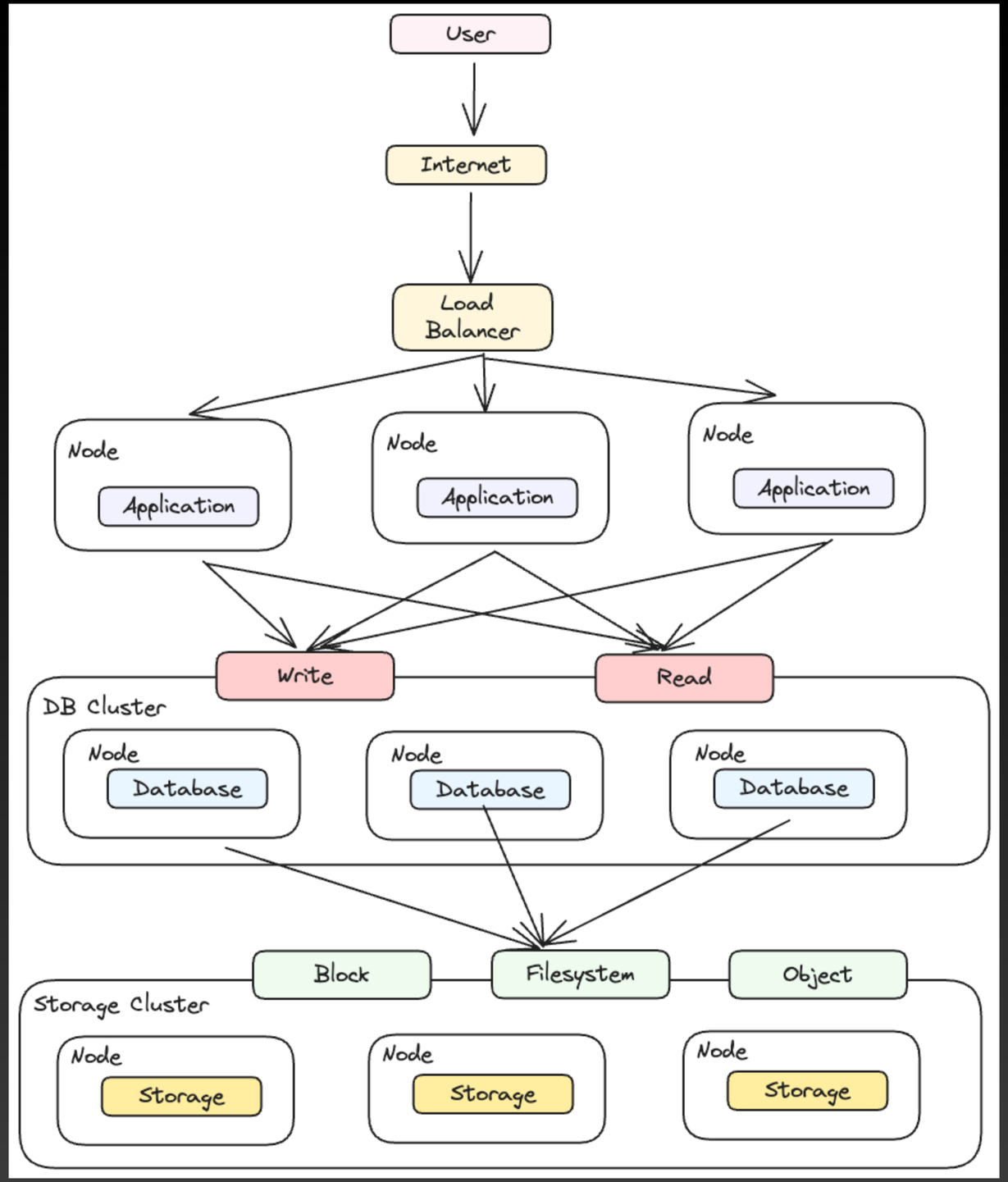
Cloud Native 架構

- Networking
  - 從 Load Balancer 到所有服務中間的網路存取
  - 防火牆
  - DNS



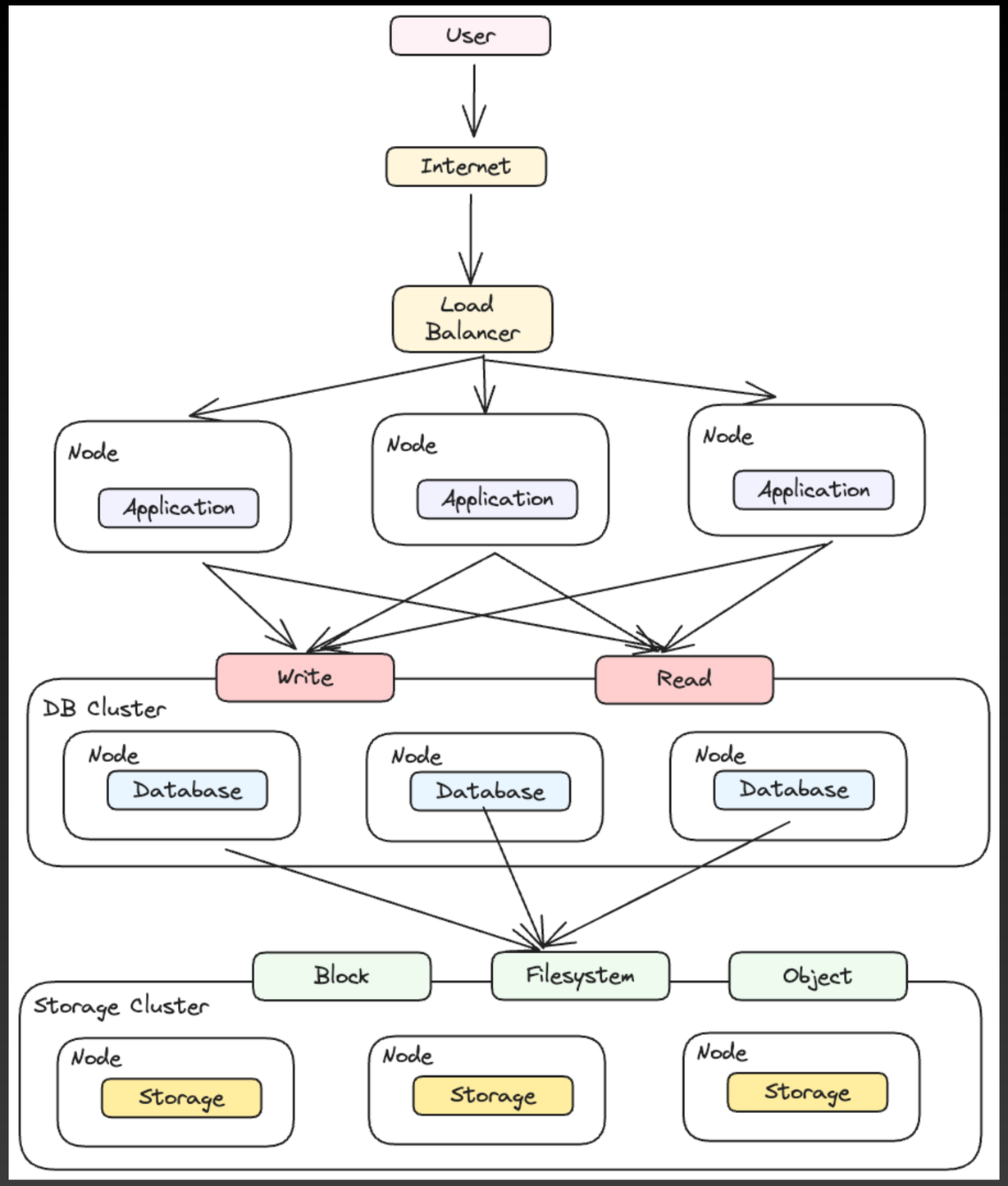
Cloud Native 架構

- 如果是現在的雲端架構，要花多少時間完成？
  - 申請 VPC
  - 申請 VM
  - 申請 DB
  - 申請 Storage
- 整體時間可能不需要一天，串接很快。



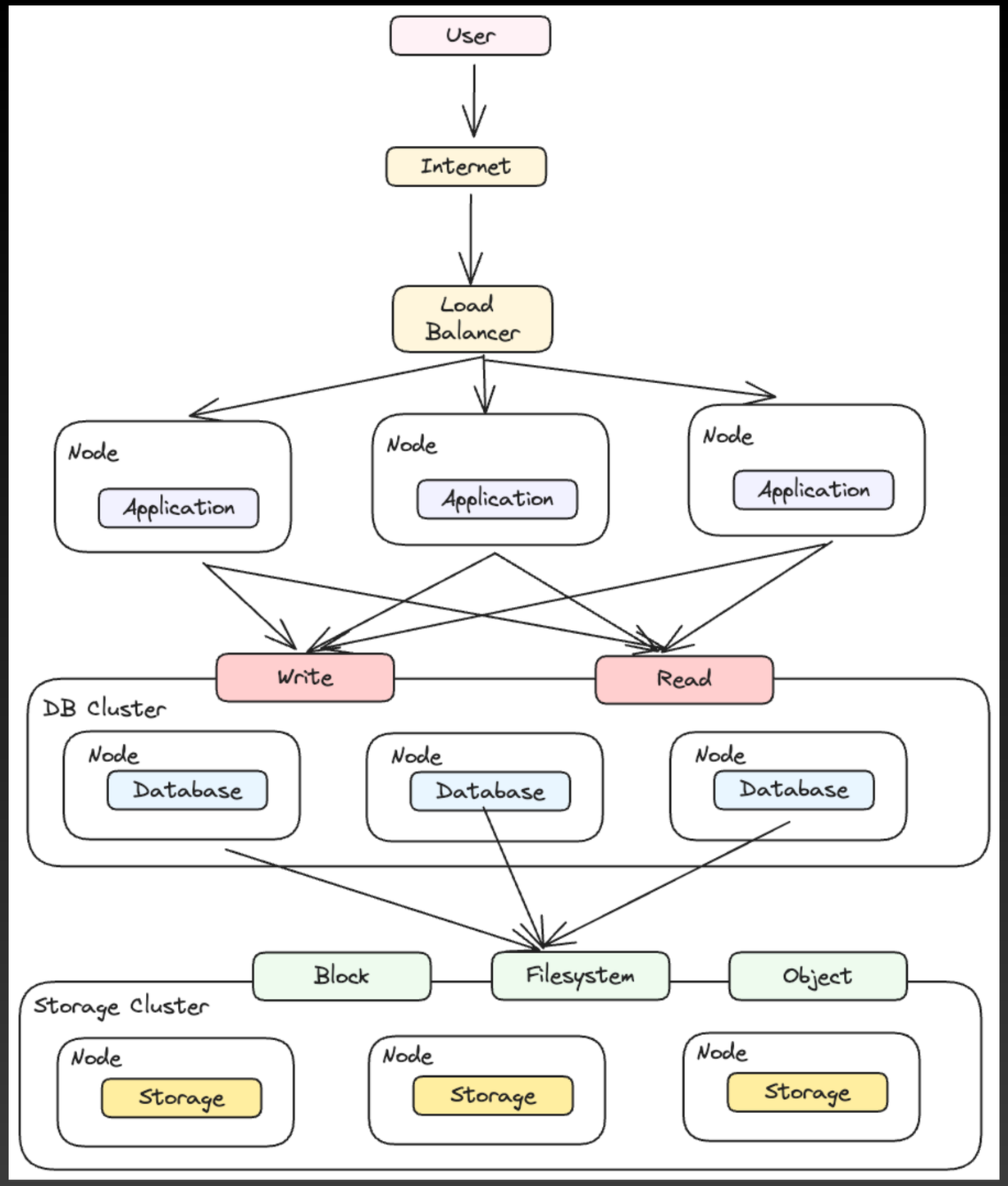
Cloud Native 架構

- 那如果要脫離雲端服務，採取盡量統一的架構，譬如 Kubernetes 那大概會需要多久？
  - 準備節點
  - 安裝 Kubernetes
  - 安裝上述所有服務



Cloud Native 架構

- 三個指令安裝
  - Application
  - Database
  - Storage
- 所有操作都是 YAML，順利的話不用半天服務就全部架設完成



Cloud Native 架構

- 以 Kubernetes 為首的生態系，將上述的所有麻煩都給簡化，並且封裝成一個又一個的檔案，開發者只要透過幾行指令，搭配 YAML 檔案就可以搭建上述所有需求

Cloud Native 架構

- Kubernetes 本身架構複雜，安裝需要多種指令與架構，但是
  - 公有雲提供 Kubernetes Service，簡化所有操作
  - 各種發行版本再次簡化其架構
    - K0s
    - K3s
    - MicroK8s
    - ...等

Cloud Native 架構

- 準備一個有下列功能的 Cloud Native 架構環境只需要不到20行的指令
  - Kubernetes 且多節點管理
  - 有 GitOps 協助應用程式的 CD
  - 有 Prometheus/Grafana, EFK 協助監控
  - 有 Istio 提供的 Service Mesh
  - 有 Ceph 提供的檔案系統
  - 有 MariaDB 提供的資料庫
  - 有 Vault 提供的 Key Management

雙面刃

- 對團隊來說，能夠用更簡單的力氣去準備環境，把精力專注於業務發展是個好事情
  - 讓公有雲或是這些開源專案幫你搞定底下的 Infrastructure
- 過度簡化的操作與隱藏的細節，會使得團隊完全無法掌握底層細節與關係
  - 環境沒問題前，都不會有問題
  - 環境有問題，無法下手除錯

雙面刃

- 傳統手工真的麻煩，曠日又廢時，但是這些廢時中間的花費其實都是工程師成長的養分
  - 必須要去看文件要去知道各種細節才有辦法把這些服務創建好
  - 各種踩雷的經驗未來某一天都會派上用場

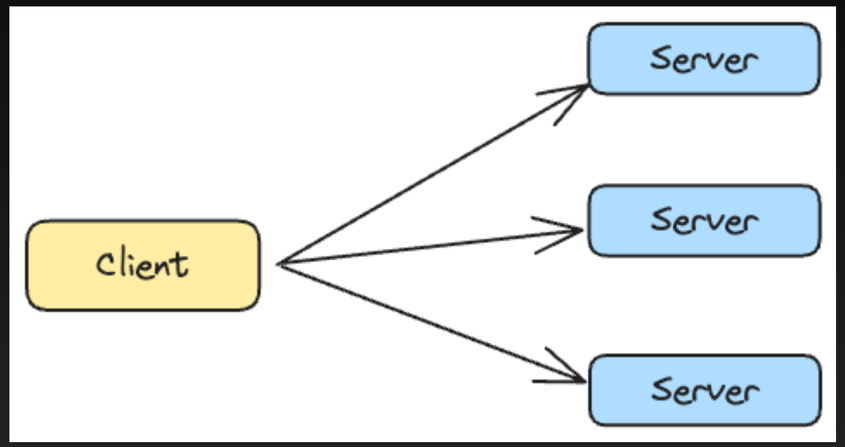
雙面刃

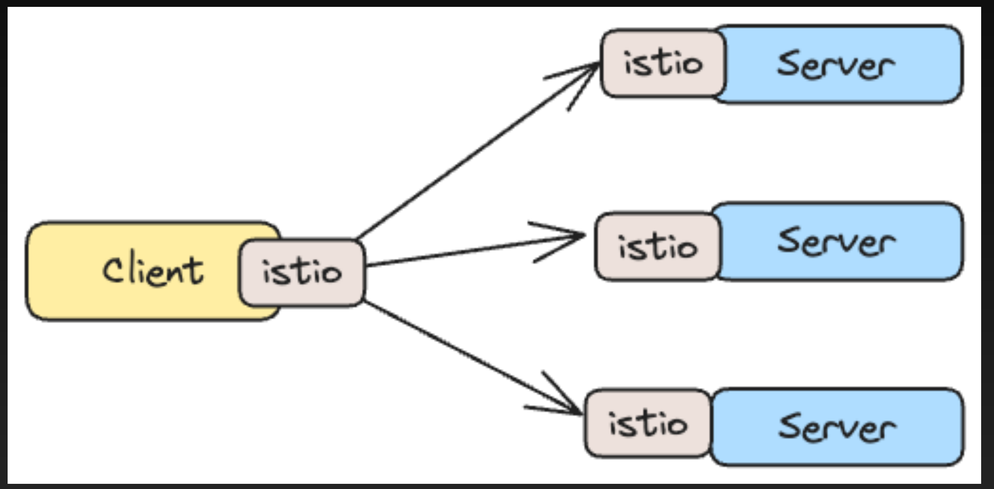
- 隱憂
  - 過度簡化，工程師只要會使用基本指令就可以部署環境，不需要有過深的背景能力去評估架構與專案
    - 過多的開源專案，不知道怎麼選擇與評比，很容易淪為
      - 誰熱門，誰星星多就用誰

雙面刃

- 隱憂
  - 專案功能愈來愈強，使用門檻降低，發生問題時無法除錯
  - Service Mesh
    - 提供各種
      - mTLS
      - Traffic Management
      - Circuit Breaker
      - ...etc

雙面刃





- 功能很多，好處很多
  - mTLS 流量自動加密
  - 針對 HTTP (Layer7) 去處理封包 (不同負載平衡演算法，流量分攤，金絲雀部署)
  - 針對 Client/Server 去進行存取權限控管
- 只要撰寫幾行 YAML 就可以輕鬆獲得上述所有優點

雙面刃

- 好好的一條 TCP 連線被拆成三條 TCP 連線
  - Timeout 問題有三邊要處理
  - TCP 參數也是有三邊要處理
  - 網路除錯還不知道要怎麼查

雙面刃

- 隱憂
  - 生態系發展迅速，容易會有 軟體/專案/版本 的焦慮
    - 擔心用的版本太舊，軟體不夠新潮，專案不夠突出
  - 最多的時間都在處理不同專案之間的整合
    - 每天就各種 YAML 修正與各種嘗試
  - 惡的循環
    - 嘗試新環境，新軟體，新專案
    - 整合所有軟體，搭建出一個適合的工作流程與環境
    - 不就後又有一個新的專案，又想要嘗試跟整合

雙面刃

- 隱憂
  - 一切的安裝都過於簡單，所有的工作時間都被低估
  - 認為部署環境只需要一天就好
    - 雖然實務上是真的很快，但是這些只是表面的快
    - 出問題就會發現自己跟白紙沒兩樣，無能為力
  - 最後工程師都沒有時間去學習這些底層與實作的概念
    - 時間都花費再
      - 看 YAML/寫 YAML
      - 看公有雲的文件

雙面刃

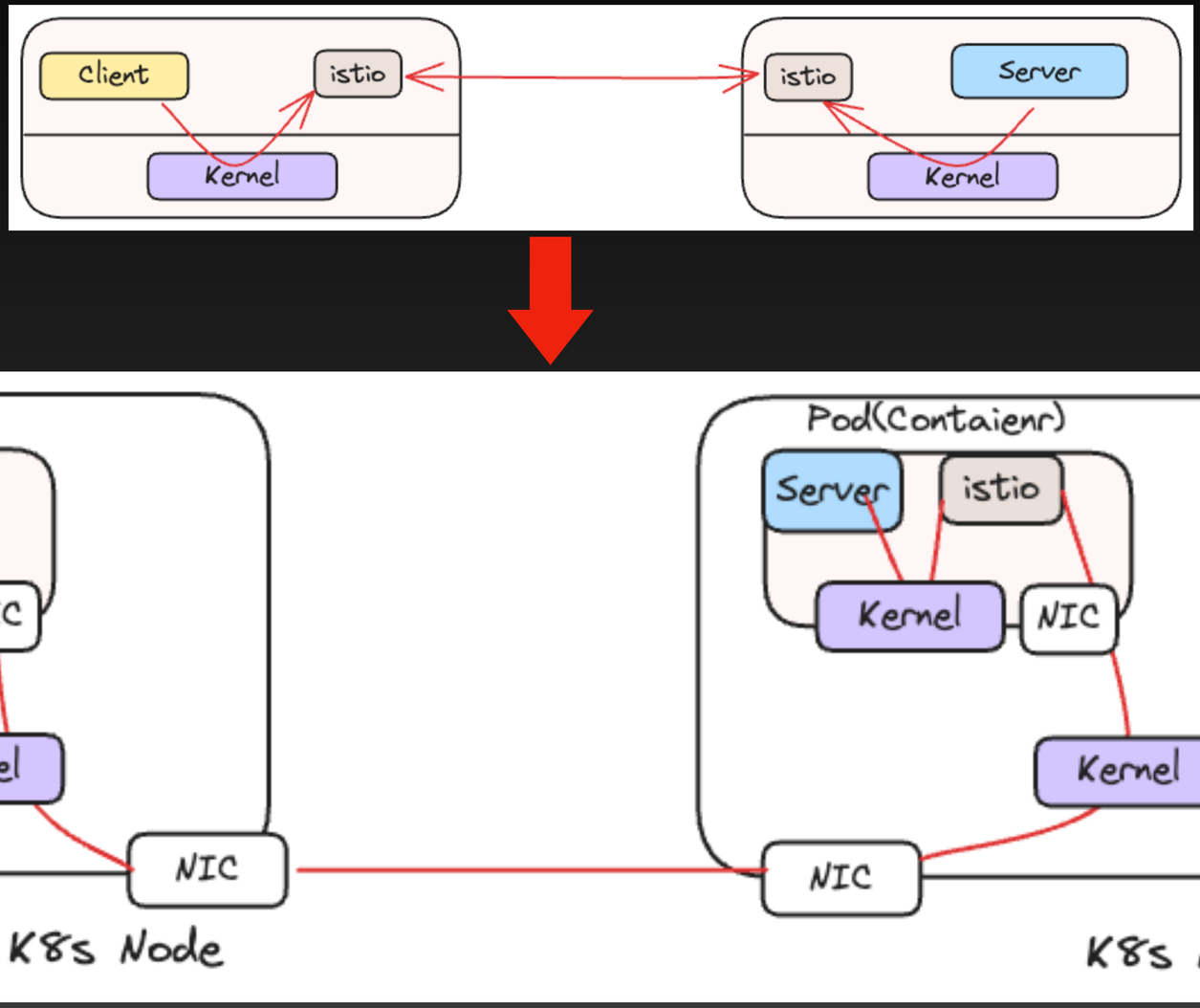
- 隱憂
  - 最擔心的是，當生產環境發生問題，團隊能夠用什麼方式找到問題
  - 最常見的就是
    - 用 YAML 除錯
    - 看官網找指令
  - 遇到複雜問題時，就沒辦法從作業系統的角度去評估與檢視問題，更別說相關工具

喪失之技能

- 網路概論
  - TCP/IP 等模型相對容易
- Kubernetes 已經先將容器網路給複雜化
  - 除錯已經很難了
- 為了 istio 的功能而導入 istio
  - 網路複雜度難上加難
  - 沒有足夠的技能幾乎沒有辦法執行有效的分析與歸納

喪失之技能

- 熱愛研究底層的人已經愈來愈少
- 然而這些東西都是大規模環境下不可或缺的技能
  - 環境愈刻苦，愈需要透過這些底層知識去最佳化與調整



喪失之技能

- 儲存設備
  - 公有雲環境用得太順手，如 EFS, EBS, S3 等
  - 沒有辦法去思考什麼情況下要用 Object, FileSystem 與 Block Device
    - 很多時候就是跟專案跑，專案用什麼，就用什麼
  - 當專案有多種選擇時，就不知道該怎麼選

喪失之技能

- 基本硬碟與 I/O 概論
  - 雲端服務勾選點一點，就有足夠的效能
- 但是回歸節點本體，這些東西實務上都有機會讓你的機器運作得更好
  - RAID
  - LVM (Logical Volume)

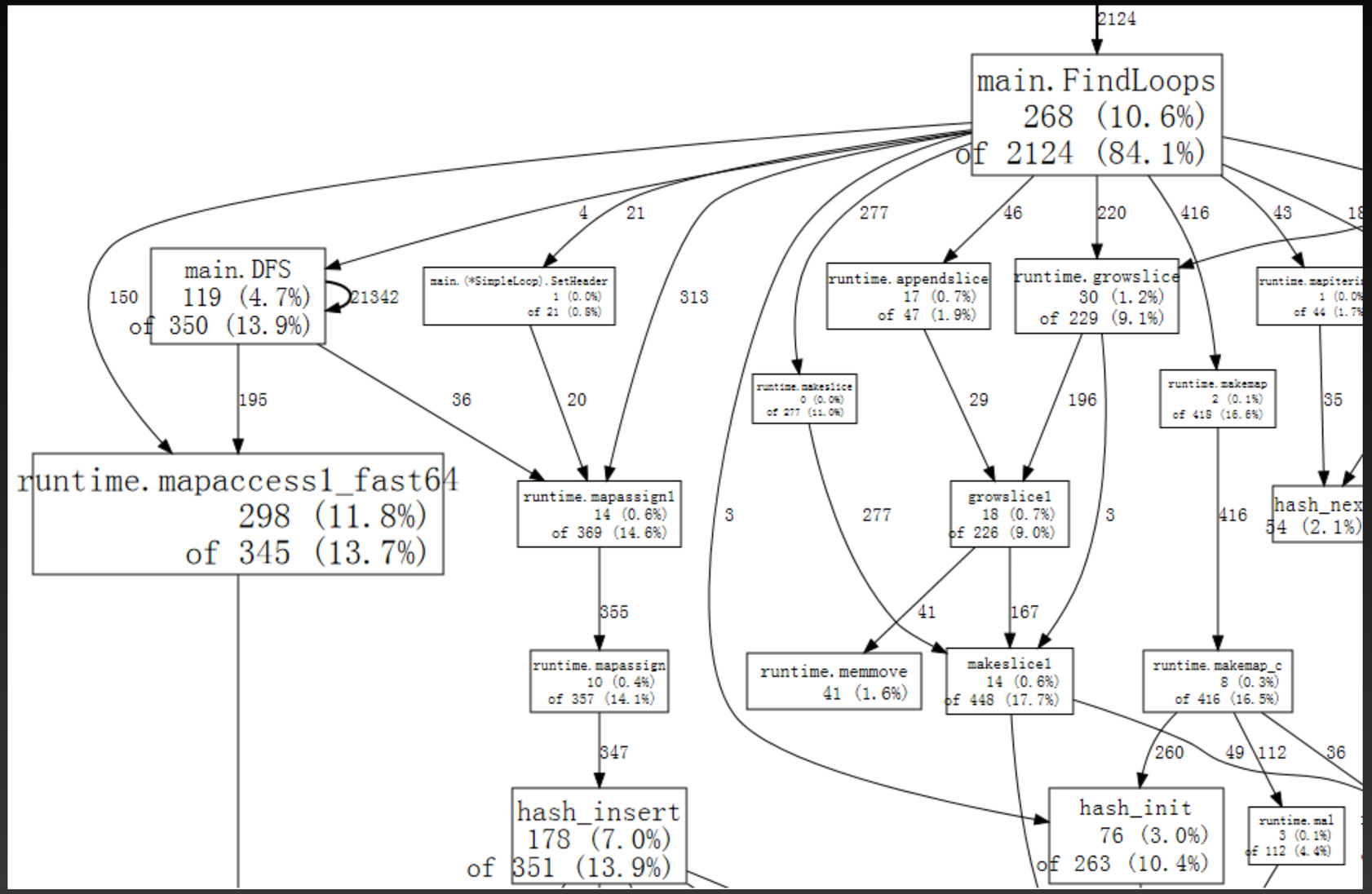
喪失之技能

- 應用程式效能問題
- Cloud Native 的架構通常可以讓維運人員很輕鬆的調度應用程式
  - 根據 CPU/Memory 等用量去自動調整數量與大小
- 水平伸展與垂直伸展可以快速解決當前業務問題
  - 但是終究會有極限，因為真正的瓶頸有可能會換地方
- 有問題先怪底層資源不夠，再來怪硬體不夠強，都沒有仔細研究自己系統效能的瓶頸是哪邊，以及要怎麼改善

喪失之技能

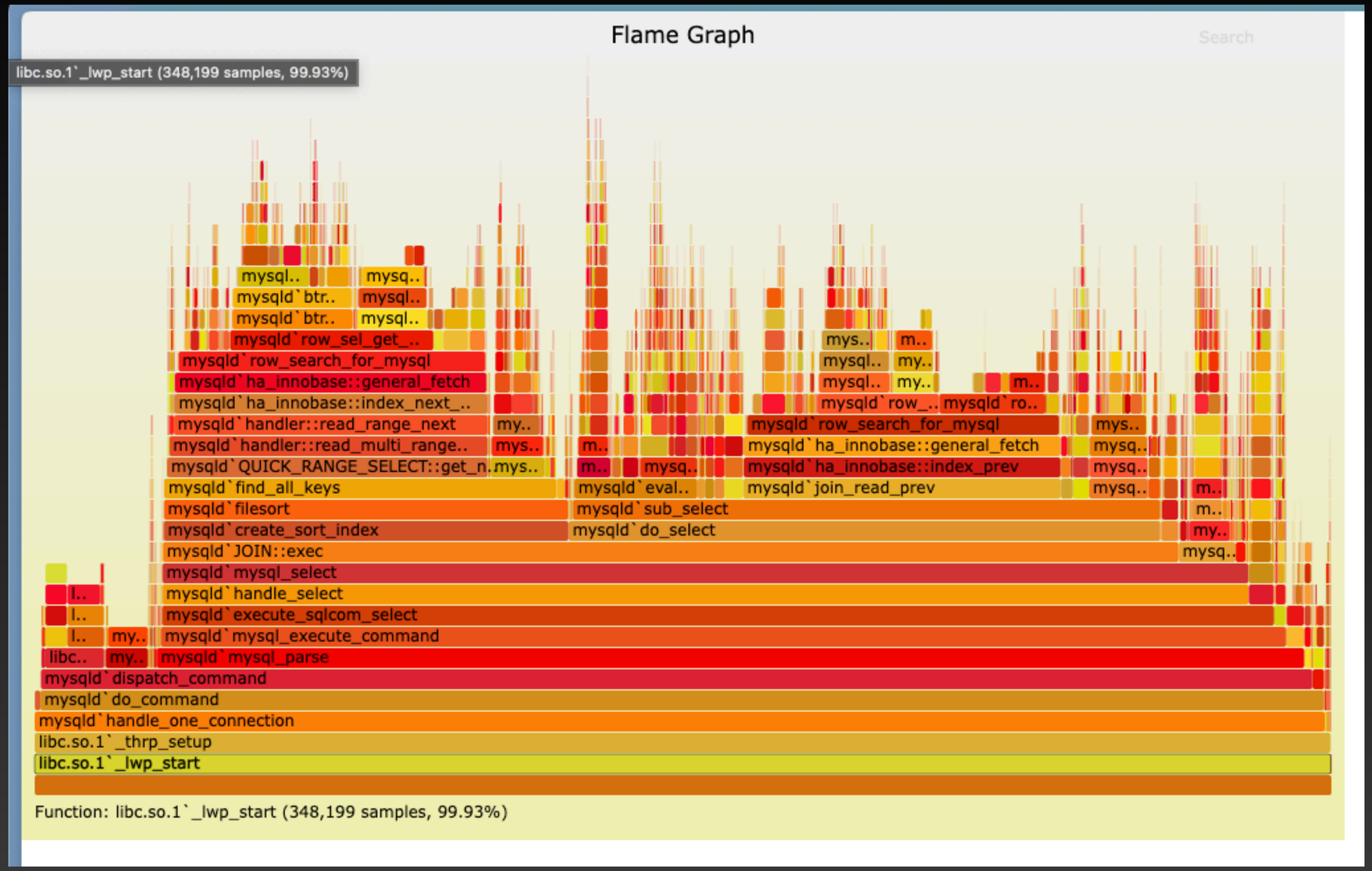
- 應用程式效能問題
- 過度依賴這些機制，反而會喪失去學習應用程式與最佳化的機會
- 如何透過工具去找尋效能之瓶頸？
  - Profiling (不同程式語言與框架)
  - 火焰圖
  - 各種 OS 指標觀測

喪失之技能



https://go.dev/blog/pprof

喪失之技能



https://www.brendangregg.com/FlameGraphs/cpu-mysql-updated.svg

喪失之技能

- 軟體技術每年一直推陳出新，正常情況下技術本來就會要一直往上追
- 最擔心的是因為沒有基礎知識，所以連如何有系統性地去分析問題都沒有辦法
- 遇到問題就是
  - Copy & Paste & Google
    - Try & Error
      - 找到答案 解決 -> 不知道原因
      - 找不到答案 無法解決 -> 沒辦法給予一個好的分析

喪失之技能

- 監控面板的雜訊化
- 團隊很容易進入一種，監控面板愈多愈豐富就愈好用的情況
  - 開發人員也很容易直接拿網路上現成的面板來使用
    - 但是並不了解每一個 Metrics 背後所代表的含義以及應該的解讀方式
  - Kubernetes + Prometheus 的組合，現在要創立一個豐富的監控面板只要幾分鐘，網路上 Copy & Paste 就好

喪失之技能

- 監控面板的雜訊化
- 最知名的範例就是，幾乎每個團隊都有針對節點 CPU 的監控，但是能夠精準回答下列兩個指標的意義與用途的少之又少
  - Load Average
  - CPU Throttling
- 這兩個指標數怎麼樣叫做不正常？ 不正常下一步要看什麼？
- 最擔心的就是看到跟 CPU 有關，有問題先喊 CPU 不夠，要求加更多 CPU

喪失之技能

- 不理解系統指標的隱憂就是，發生問題的時候，沒有辦法精準地找到 Root Cause
- 團隊更傾向用感覺跟運氣找問題，看哪些指標這些時段內有飆漲，然後想辦法找個理由把故事拼湊出來
  - 用一堆指標找趨勢，看圖說故事
    - 不是不行，重點是事後有沒有辦法找到原因並且學習改進

喪失之技能

- 上述所有的問題於公有雲環境已經很明顯
- 當環境遷移到地端自行架設機房時，會更為嚴重．
- 或是環境規模夠大，很多開源專案已經不敷使用，這時候就會陷入無力感
  - 拔也拔不掉，修也修不掉

喪失之技能

- 網路
  - 沒有公有雲方便的防火牆與路由設定
  - 所有的網路協定
    - Layer 2 (Switch, STP, Bridge)
    - Layer 3 (Routing, BGP, OSPF, VLAN)
    - MTU, MSS
  - 甚至硬體交換機相關的設計都要自行處理

喪失之技能

- 儲存
  - 要如何提供 Object, FileSystem, Block 等不同類型的儲存設備？
  - 這些專案背後的最佳化很仰賴對 Storage 領域與底層 OS 合作的理解，若過去太依賴雲端服務很容易都忽略這些細節
    - 一行 YAML 指令安裝的通常都是可以動，但是效能方面一定不能滿足需求
- Ceph 為範例，安裝變得超級簡單，但是最佳化跟運行上的調整則是非常困難

喪失之技能

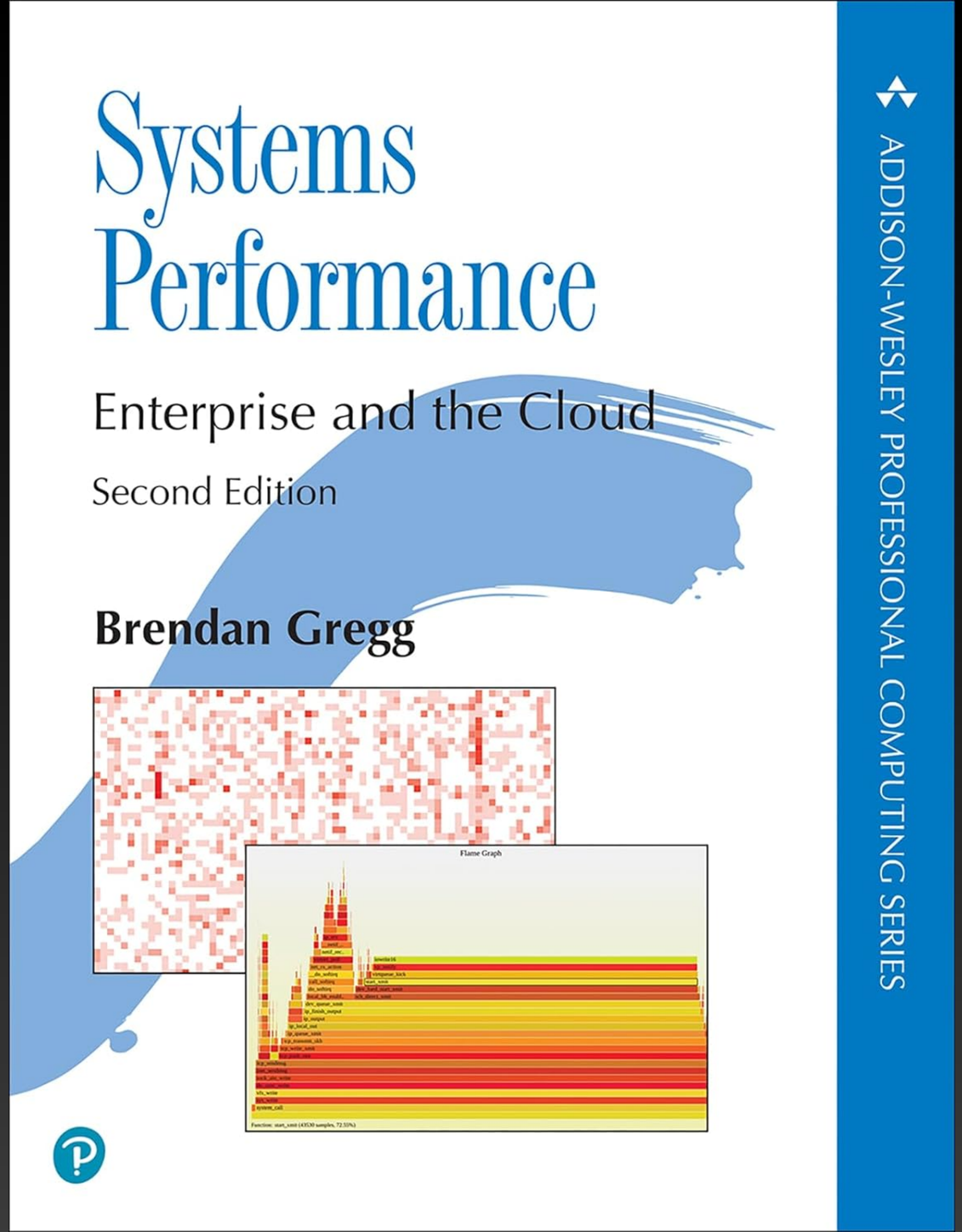
- 運算
  - 硬體資源有限，不可能無限制的擴充節點與服務
  - 如何就有限的資源內，節省系統資源用量，找出瓶頸與浪費的地方是難點

怎麼面對？

- 工具好用歸好用，還是要花時間去培養背後知識
- 不要過度專注於 “果”，而是要去思考 ”因“
  - 理解每個專案實作的技術與架構，釐清彼此的優缺點
  - 很多專案背後都有複雜的演算法，這些演算法也都決定了這些專案的瓶頸與優劣
    - 枯燥乏味，但是會讓你更加理解整個軟體系統，未來可以通用到其他的開源專案

怎麼面對？

- 唸書
  - System Performance
- 推薦 Brendan Gregg 的所有文章與著作
  - 以底層為核心概念，讓你重新掌握這些效能有關的概念
  - 遇到問題該用什麼思路，用什麼工具
  - 不同指標之間的解讀

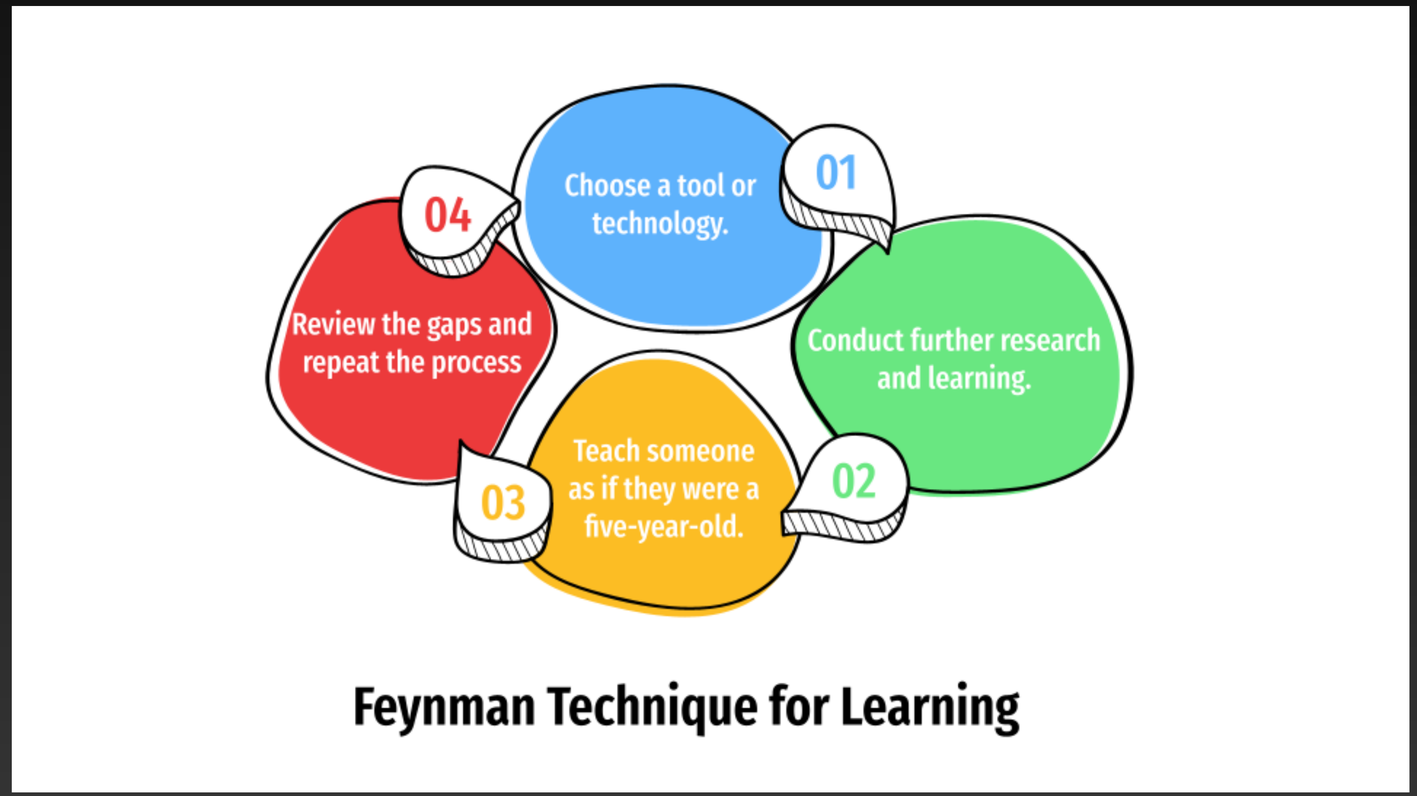


怎麼改善?

- 雲端世代，用好跟學好是兩件事情
- 服務可以上線 != 團隊可以維運
  - 不要低估軟體的難易度
  - 部署易，理解難
- 鼓勵團隊去深度學習，給予團隊時間與環境去培養技能
  - 費曼學習法

怎麼改善?

- Step 1
  - 選擇一個領域並且嘗試學習
- Step 2
  - 分享自己所學
- Step 3
  - 檢視過程是否有什麼不足
- Step 4
  - 簡化流程與概念
- 反覆上述流程



https://testsigma.com/blog/feynman-learning-technique/

怎麼改善?

- 舉例：
  - 學習 istio + service Mesh
- 進行分享
  - 一開始一定會分享的很粗淺，很 wiki 的講法
- 透過與講者的回饋，理解到自已不足的地方
  - 封包怎麼走？ 怎麼除錯？ 跟其他解決方案的差異是什麼？
- 根據理解不足的地方，重新學習，再次分享
  - 反覆所有流程，確保自己有辦法將腦中所學分享出去
    - 別人聽得懂
    - 問題可以回答

怎麼改善?

- 舉例
  - 組織技術分享會
    - 透過問與答互相討論，找出彼此不理解或是忽略的地方
  - 實體會議效果更好
    - 不做投影片
    - 直接寫白板，架構直接畫，邊畫邊討論
  - 講者可以精進自己對該領域的能力
  - 聽眾也可以學習，藉此避免團隊的技術瓶頸
    - 一個技術只有一個人會是一個潛在的問題

怎麼面對？

- 建議不要走讀書會的形式
  - 因為每個人都不熟，沒有人可以幫助大家補充以及判斷正確與否
  - 很容易就是大家針對表面的東西念過去，但是沒有辦法內化
    - 也無法透過既有環境來參考學習