一個人的 DevOps

從程式碼到 Production

站在巨人肩膀上，一個人也能交付一座 Production

白天是資訊室主任
骨子裡是工程師

平常是一人全端開發者

今天的範例，是自己做的 Side Project (個人部落格)

國軍高雄總醫院 資訊室主任 鄭重男(Nickle)

一個人的焦慮

一個人，要扛 > 六個角色

PM · 前端 · 後端 · DBA · SRE · 資安

真正讓人放棄的，從來不是不會寫，是後面那一長串維運

大部分的 side project，都是死在這裡

同樣的流程，幾年內換了一個架構

2020前後 · 工具為中心

你是所有工具之間的那一坨膠水

Git、CI、測試、部署已經能串接；但上下文分散，流程判斷、錯誤處理、跨工具協調，仍然主要靠人補位

2025–2026 · Agent 為中心

規格、測試與工作流，變成人與 agent 的共同語言

從需求、改檔、跑測試、開 PR，到觸發 CI/CD，agent 開始串起整條 SDLC。你不再只是工具之間的膠水，而是流程的指揮者。

DevOpsDays Taipei 2026·台北文創·Nickle

開發時間	commits	記憶體	月租(USD)	上線至今
5 個月	235	1GB	< $10	0 停機

一個陽春的 VPS 配置、一個人的人力

撐一個每天更新、有背景工作、有即時推播的正式網站

- $4.00/mo
  1 vCPU - 512 MB RAM - 10 GB SSD - 500 GB Transfer
- $6.00/mo
  1 vCPU - 1 GB RAM - 25 GB SSD - 1000 GB Transfer
- $12.00/mo
  1 vCPU - 2 GB RAM - 50 GB SSD - 2 TB Transfer
- $18.00/mo
  2 vCPU - 2 GB RAM - 60 GB SSD - 3 TB Transfer

一個人，不等於一個人在戰鬥

我沒有變成超人

只是找到三個超強的隊友，把我不擅長、不想自己扛的部分，丟給它們

框架、AI、維運 – 三個巨人

DevOpsDays Taipei 2026·台北文創·Nickle

過去，後端要養一整座動物園

Sidekiq · Redis · Memcached · Cable

每多一隻，就多一份設定、監控、備份，和一個故障點
對一個人來說，這就是壓垮你的最後一根稻草

現在，一個 PostgreSQL 全包

Solid Queue　背景工作，落在你本來就有的那個資料庫

Solid Cache　快取，不再需要另一隻 Memcached

Solid Cable　即時更新，同樣跑在同一個 PostgreSQL

Redis 不見了，Sidekiq 不見了，Memcached 也不見了

架構圖從一堆方塊，縮成一個

服務數量下降，就是負擔下降

一個資料庫　要備份的，只有它

一個服務　半夜要監控的，只有它

一台機器　月底要付錢的，只有它

⅓

做同樣一個功能，Ruby 用的 token 大約是其他主流的三分之一

主流語言裡 token 效率排第二省 token，就是省錢、更快、context 更夠用

語言	寫法
Ruby（原版）	`users.select(&:active?).map(&:email)`
TypeScript	`users.filter(u => u.active).map(u => u.email)`
Python	`[u.email for u in users if u.active]`
Java	`users.stream().filter(User::isActive).map(User::getEmail).toList()`
C#	`users.Where(u => u.Active).Select(u => u.Email).ToList()`

沒用 Node，不跑前端建置流程
能少裝就少裝，能不編譯就不編譯

Importmap · Propshaft · Tailwind

瀏覽器直接吃 ES module

一個人開發，需要維護的東西越少越好

DevOpsDays Taipei 2026·台北文創·Nickle

AI Ascent
question for the room. Who writes 100%
of their code by hand?

用 AI 寫程式，沒有規格、沒有測試
就像閉著眼睛開車

車速很快，但方向盤根本不在你手上

問題不是在 AI 而是在你沒有給它方向及流程

AI 是放大器：你有紀律，它放大紀律；你很亂，它加倍放大混亂

DevOpsDays Taipei 2026·台北文創·Nickle

幫我做一個網站

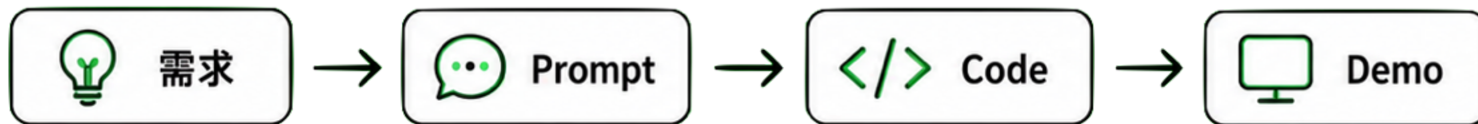
介面

API

資料

登入頁

Vibe Coding 讓「做出來」變得很快



動手前，先寫**規格**

Spec 成為我跟 AI 的共同語言，它做出來的東西就不會天馬行空

這套方法不是我發明的(Openspec)

高見龍的 Spectra

開發流程也站在大神的肩膀上

https://spectra.5xcamp.us

change 的生命週期

討論、提案、實作、歸檔

01 Discuss 先討論清楚這次要做什麼

02 Propose 寫成提案：為什麼、規格、拆好的任務清單

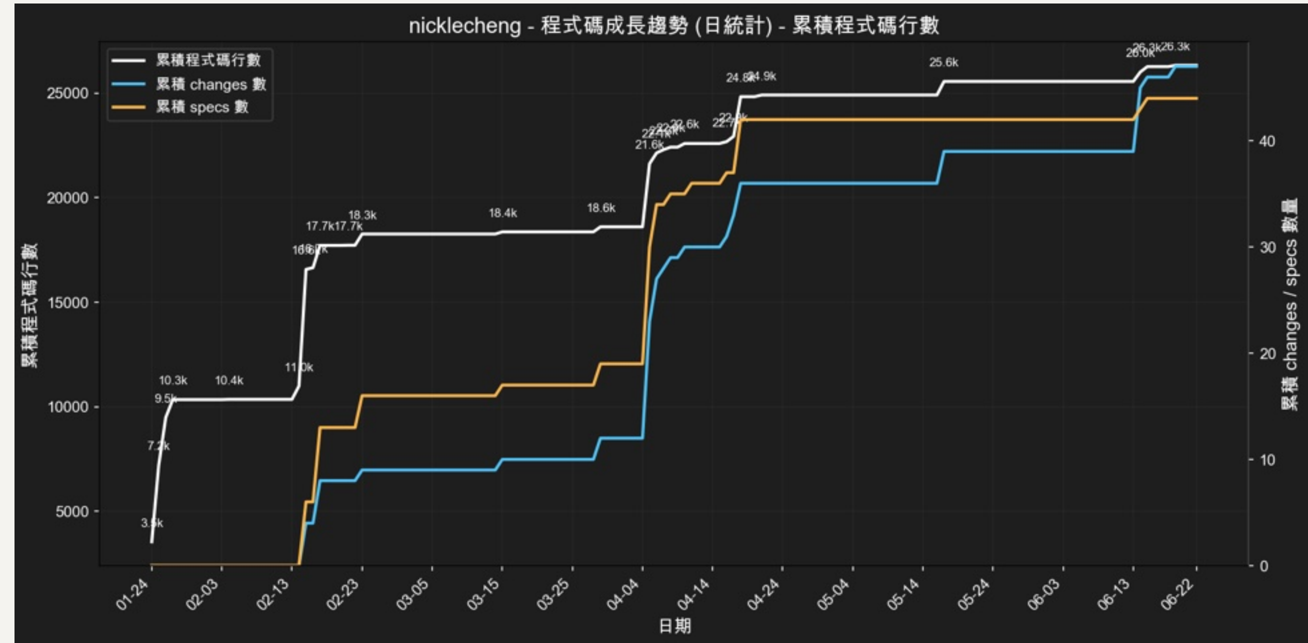
03 Apply AI 照清單實作、先寫測試再寫實作

04 Archive 歸檔，知識沉澱、決策可回溯

歸檔最關鍵 — 半年後回來看，看規格就知道當初為什麼這樣設計

紀律，可以量化

47



五個月，完成並歸檔 47 個 change、收斂 44 份規格

平均每 2~3 天，一個功能就從寫規格、測試、實作完整循環

功能領域	Change	Spec	小計
Wiki & 知識圖譜	22	13	35
About & 個人檔案	8	9	17
內容 / 文章互動	5	4	9
推播 & 通知	2	4	6
SEO / 評論	2	4	6
PWA & 行動體驗	4	2	6
後台管理	2	3	5
技能 Skill	2	2	4
LLM & 翻譯	0	2	2
PDF 生成	0	1	1
總計	**47**	**44**	91

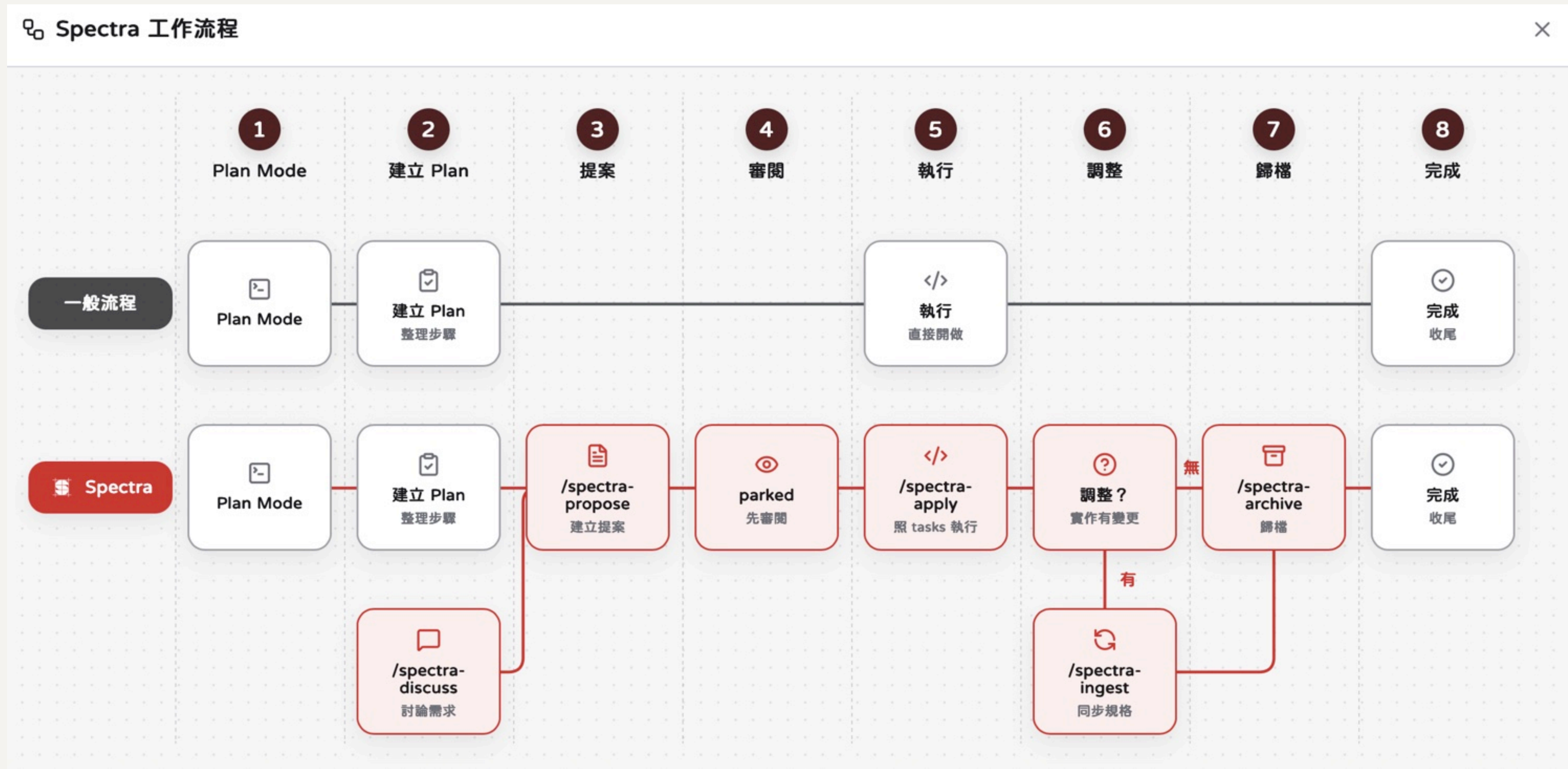
規格守方向，測試守行為

幻覺進不了 PR

因為程式碼跑過測試才送出

— DHH，談 Rails 的未來

Spectra 工作流程



/spectra-discuss 目前 wiki 文章排序會先依分類再依發布日期排序，不方便使用者閱讀，你有好的建議？

/spectra-propose change_name

/spectra-apply change_name

/spectra-archive change_name

我給的是一份規格，不是一句指令

我介入的點不多，主要在看方向對不對

DevOpsDays Taipei 2026·台北文創·Nickle

1GB	<$10	1台
記憶體	月租 USD	不用第二台

跑正式服務，不一定要很貴的基礎設施

部署只需 一行 kamal deploy

```
$ kamal deploy
  → build Docker image
  → 推送到伺服器
  → 切換新版本
  ✓ Deployed successfully
```

沒有 Kubernetes、沒有一堆 YAML、沒有維運團隊待命

部署的心理負擔，大概跟 git push 差不多

零停機

新容器**健康**，才接手流量

01 起新容器　在舊的旁邊，先啟動新版本

02 健康檢查　打 /up 確認它真的活著、能服務

03 切流量　確認 OK，才把使用者切過去

04 收舊的　全程永遠有一個健康容器在服務

使用者完全無感 — 根本不知道剛上了一個新版

把 1GB 記憶體榨乾

了解你的限制，一個一個調

```
WEB_CONCURRENCY=1        # 單 worker，不開多 process
RAILS_MAX_THREADS=5      # 用執行緒吃並行
JOB_CONCURRENCY=2        # 背景工作壓低，別吃爆記憶體
# 直接在自己機器建 image、本機推送，省下外部 registry
```

調這些參數沒有魔法，就是了解限制之後讓 AI 幫忙調出來

便宜機器也是可以調校的

HTTPS，也幫你包好

憑證、續約，**不用你管**

Kamal 內建反向代理，自動申請並續約 Let's Encrypt

不用裝 Nginx、不用跑 Certbot

三個巨人都在做同一件事：把瑣碎、易忘、出錯會痛的事，從你身上拿走

不是每個系統都需要 Kubernetes

對一個人、一個中小型應用，一上來就搬 K8s 是過度設計

先用剛剛好的方案，需要再長大，Kamal 就是剛好

護城河 · CI 與資安

沒有同事幫你 review，**機器就是把關**

一個人開發，沒有人會攔你，你來自己就是最後一道防線

人會累、會分心、會偷懶

唯一靠得住的，是每次都自動跑，不靠人力

一條 bin/ci 串所有檢查

一個指令，全部跑完

01 RuboCop 程式風格
02 bundler-audit 套件已知漏洞
03 importmap audit 前端相依漏洞
04 Brakeman 靜態安全分析
05 Minitest 完整測試

● 完整狀況清楚了。三個 gem 都是 patch 級更新，且都落在 Gemfile 現有版本約束內，更新風險很低：

Gem	現況	修補版	相依	約束
concurrent-ruby	1.3.6	1.3.7	間接	–
faraday	2.14.2	2.14.3	直接	~> 2.12 ✓
nokogiri	1.19.3	1.19.4	直接	>= 1.16.5 ✓

security gate 擋下部署，需要先決定怎麼處理才能繼續。

□ Security gate

security gate (bundler-audit) 偵測到 3 個 gem 有未修補漏洞，擋下部署。要怎麼處理？

❯ 1. 更新 3 個 gem
2. 加入忽略清單
3. 本次跳過 gate

```
bundle update concurrent-ruby faraday nokogiri --conservative
→ concurrent-ruby 1.3.6 → 1.3.7
→ faraday         2.14.2 → 2.14.3 (High DoS)
→ nokogiri        1.19.3 → 1.19.4
→ 重新 kamal deploy

（改動 Gemfile.lock，建議事後 commit)
```

Rails 連 CI 工具都不用另外搭，任何一關沒過，就無法送出

上線後的防護

限流、信任名單、全程加密

限流與封鎖　rack-attack 擋掉暴力打站、狂掃。

信任名單　TRUSTED_IPS 確保自己人不被自己的規則誤傷。

全程 HTTPS　幾層加起來，對個人網站已是夠用的護城河。

難的不是會不會做，是有沒有意識到要做 – 線上裸奔

老實說，一個人是有極限的

bus factor = 1

適用 個人專案、新創 MVP、內部工具、中小型應用

巨人幫你扛掉八成的重複勞動，這些場景一個人撐得住

不適用 高可用、強合規、大流量的系統

不是一個人該逞強的場景，老實找個團隊

一人專案，系統沒人接，剩下那兩成的判斷與責任，永遠是你的

終點，還可以再往後一點

上線不是結束，維運也交給 agent

定時跑排程、自動更新 wiki、整理與發佈文章 — 睡覺的時候，它還在工作

從想法 → 開發 → Production → 上線後的自動維運與內容經營，一個人加一群 agent (Nanoclaw) 在跑

我不是超人，是選對巨人

框架扛基礎設施

AI 扛開發紀律

維運工具扛部署營運

你負責機器取代不了的部分

判斷與創造