

[https://youtu.be/KEtrWd6zUKI?list=PLIAzIZzCjtLJQzoK-1n6R\\_KVTXm4tHZqP](https://youtu.be/KEtrWd6zUKI?list=PLIAzIZzCjtLJQzoK-1n6R_KVTXm4tHZqP)

[https://youtu.be/6elw85w7QTU?list=PLIAzIZzCjtLJQzoK-1n6R\\_KVTXm4tHZqP](https://youtu.be/6elw85w7QTU?list=PLIAzIZzCjtLJQzoK-1n6R_KVTXm4tHZqP)

這份教學原始碼主要針對 **NTU 網頁程式設計** 課程的第五項作業，指導學生如何結合 **Next.js 全端框架**、**Pusher 即時通訊服務** 與 **Serverless 無伺服器架構** 來開發一個類似 X（原 Twitter）的社群應用。內容特別強調了 **Serverless 架構** 的趨勢，解釋開發者雖無需自行管理實體伺服器，但必須面對其「無狀態」且無法持續連線的限制，因此需引入 **Pusher** 作為中介代理，利用 **WebSocket** 技術實現雙向且即時的資訊推送。教授亦藉由馬斯克收購 Twitter 的案例，勉勵學生在運用 AI 輔助開發的時代，更應具備定義 **細緻規格書** 的耐心與邏輯表達能力，而非僅依賴工具生成程式碼。最後，課程引導學生透過 **Vercel** 進行雲端部署，並利用第三方服務處理影音多媒體與 **NoSQL 資料庫**，完整建構一個現代化的全端雲端服務。

這份教學原始碼主要探討在 **AI 輔助編碼** 的時代下，如何利用 **Next.js**、**Pusher** 與 **MongoDB** 等技術快速構建並部署一個具備實時通訊功能的全端應用程式。作者強調 AI 雖然大幅降低了開發門檻，但學生仍需具備系統化的專案管理能力，並提倡 **CI/CD（持續實作與持續部署）** 的精神，要求開發者在初期就將服務上雲以避免後期的技術債。文中詳細介紹了 **NoSQL 資料庫** 的四大類型與應用場景，特別是 **MongoDB Atlas** 的雲端配置，以及如何整合 **Cloudinary** 處理多媒體資料。最後，作者提醒開發者應將複雜專案拆解為多個 **Milestones（里程碑）**，並善用版本控制工具與錯誤日誌進行調試，才能在日益複雜的軟體架構中保持開發效率與穩定性。

Discuss Serverless 架構特性.

**Serverless（無伺服器架構）** \*\*開發者不需要親自管理伺服器。在這種架構下，開發者只需專注於撰寫功能（Function）\*\*，而底層的基礎設施、運算資源與自動化管理則交由雲端平台（如 Vercel、AWS Lambda）負責。

以下根據來源內容，詳細探討 Serverless 架構的主要特性：

### 1. 以功能為核心的開發（Function-driven）

Serverless 架構主要以 **Function** 為主要撰寫單位。開發者只需定義好 API 或是特定功能的邏輯，並將其部署到雲端。這種做法能實現**極致的開發體驗**，讓團隊將精力集中在產品創意、設計與使用者服務的開發上，而不用擔心後端的維護問題。

### 2. 按需分配與自動縮放（On-demand & Scalability）

與傳統伺服器 24 小時不間斷運行不同，Serverless 是**有需求（On-demand）時才啟動功能**。

- **水平擴展性**：當流量突然暴增（例如雙 11 活動）時，雲端平台能自動且容易地進行水平擴展，開發者不需要提前準備大量實體機器或手動設定環境。

- **資源回收**：當任務執行完畢後，系統會自動關閉功能，不佔用多餘資源。

### 3. 計價模式：按量付費（Pay-as-you-go）

傳統伺服器無論是否有人使用，通常都必須支付固定的維運費用。Serverless 則採取**按執行次數或執行時間計費**的模式。這意味著費用與實際流量成正比，對於流量波動較大或處於成長階段的服務來說，可以有效**降低成本**。

#### 4. 無狀態特性 (Stateless)

Serverless 的每一次執行都是**獨立且短暫的 (Short-lived)**。

- **不保留狀態**：功能執行完畢後就會關閉，因此它不會記得上次執行的結果。
- **狀態管理**：開發者必須自己負責狀態的保存，通常需要配合外部的資料庫（如 MongoDB）或緩存服務來儲存資料。

#### 5. 即時通訊的侷限與補償

由於 Serverless 伺服器會自動斷線且不具備持續連線的特性，它原生無法直接處理需要長連接的**即時雙向溝通**（如 WebSocket）。為了在 Serverless 架構中實現即時互動（如按讚通知、聊天室），通常需要引入像 **Pusher** 這樣的第三方雲端服務，充當「不睡覺的守衛 (Watch dog)」，在中間維持訊息的傳遞。

#### 6. 典型應用場景

Serverless 非常適合以下類型的應用：

- **API 服務**：處理不固定流量的請求。
- **靜態網站搭配動態功能**：提高管理效率。
- **短時間繁重工作**：如影像壓縮、驗證碼產生等。
- **定時背景任務**：如定時爬蟲抓取資料。

---

為了讓你更好理解，我們可以把 Serverless 比喻成「計程車服務」：

傳統架構就像是「**自己養車**」，不論你有沒有出門，都要負責車庫、保養、保險和冷氣費，且車子閒置時成本依然存在。而 Serverless 就像是「**叫計程車 (Uber)**」：你不需要管車子怎麼維修、停哪裡，只有在你想去某個地方（呼叫 Function）時，車子才會出現並為你服務，而你也只需要根據搭乘的里程和時間（執行次數與量）付費，下車後（功能執行完）車子就會離開，不佔用你的空間。

比較傳統伺服器與 Serverless 的計價方式差異。

為何 Serverless 架構需要 Pusher 來實現即時通訊？

簡述 Pusher 產品中 Channels 與 Beams 的功能區別。

Discuss Vercel 雲端部署。

**Vercel** 是一個專為前端開發者設計的雲端部署平台，其核心理念是提供「**極致的開發體驗**」與「**自動化部署**」。它與全端框架 **Next.js** 由同一家公司開發，因此對 Next.js 提供最完整且友好的支援。

以下根據來源內容，詳細討論 Vercel 雲端部署的特性與操作流程：

##### 1. 核心設計理念

Vercel 的創辦團隊具有強烈的前端與設計師背景（UI/UX），他們希望解決開發者不想管理後端基礎設施的痛點。透過 Vercel，團隊可以將精力集中在產品的**創意、設計與使用者服務**上，而將後端的部署與維運交給機器自動化處理。

## 2. 自動化 CI/CD 工作流

Vercel 最強大的功能之一是與程式碼託管平台（如 **GitHub**）的深度整合：

- **專案綁定**：開發者只需將 Vercel 專案與 GitHub 的 Repository 進行綁定。
- **自動部屬**：一旦完成綁定，每當開發者執行 `git push` 將更新推送到 GitHub 時，Vercel 就會自動感應並在雲端重新進行部屬。
- **持續整合與部屬**：這種機制讓開發者完全不需要手動干預部屬流程，實現了高效的 **CI/CD**（持續整合/持續部屬）。

## 3. 多層次的雲端架構處理

當請求進入 Vercel 部署的專案時，平台會根據程式碼的特性自動分流處理：

- **靜態網頁（Static Pages）**：透過 **CDN 與快取（Cache）** 功能直接回傳，這是最有效率的處理方式。
- **動態功能**：透過 **Serverless Functions**（無伺服器功能）或 **Edge Functions** 進行運算處理。
- **狀態管理**：若需要儲存資料，Vercel 會引導開發者連線到外部的資料庫（如 MongoDB）。

## 4. 部屬方式與工具

除了透過 GitHub 自動化部署，開發者也可以使用以下工具：

- **Vercel CLI（命令列介面）**：開發者可以在本地端安裝 Vercel CLI，透過指令如 `vercel login` 進行登入，並使用 `vercel` 指令快速設定並部屬專案。
- **環境變數設定**：在部屬過程中，開發者必須在 Vercel 後台設定必要的**環境變數（Environment Variables）**，例如資料庫連線字串（DB URL）或第三方服務（如 Pusher、Facebook、Cloudinary）的 API Key，以確保雲端服務能正常運行。

## 5. 優勢與適用性

- **易用性**：相比於傳統的雲端平台（如 Digital Ocean）或其他的部署工具（如 Railway），Vercel 的部署流程更加直覺且完整。
- **成本效益**：Vercel 提供一定額度的免費使用量，非常適合學生作業、個人專案或初步成長的創業專案。

---

為了讓你更好理解，我們可以把 Vercel 部署比喻成「自動發報系統」：

傳統部署像是「**手動派報**」，你寫好報紙後，要自己搬運、租倉庫、找派報員，還要擔心天氣好壞。而使用\*\*就像是裝了一個「感應式印刷機」\*\*：你只要在辦公室把稿子寫完並按下儲存（Git Push），印刷機就會自動感應到新內容，立刻自動排版、印刷並透過全自動無人機（CDN）送到全世界讀者的手上，你完全不需要踏出辦公室一步。

為什麼 Serverless 架構需要 Pusher 服務？

傳統 HTTP 與 WebSocket 的通訊機制有何不同？

寫程式規格（Specification）對軟體工程師的重要性為何？

Discuss Pusher 即時通訊.

**Pusher** 是一個專門提供**即時通訊（Real-time Communication）**功能的雲端服務平台。在現代網頁開發，尤其是結合 Serverless（無伺服器）架構時，它扮演了極其關鍵的「仲介者」角色。

以下根據來源內容，詳細討論 Pusher 的技術特性、必要性及應用：

### 1. 解決 HTTP 協議的「懶惰」限制

傳統的 **HTTP 協議**是單向且被動的。來源中形容 HTTP 非常「懶惰」：只有當客戶端（Client）主動發送請求時，伺服器才會回應。

- **無法主動通知**：在 HTTP 架構下，如果使用者 A 發送了訊息，伺服器無法主動告訴使用者 B 有新訊息，除非使用者 B 不斷地刷新頁面或進行「輪詢（Polling）」。
- **雙向溝通需求**：對於遊戲、社群平台（如 X/Twitter）或聊天室，需要**雙向且即時的**互動，這時就必須引入像 Pusher 這樣的技術。

### 2. Serverless 架構中的「不睡覺守衛」

在 Vercel 等 Serverless 平台上，後端功能（Functions）是「隨開隨關」的，無法維持長時間的連線。

- **維持持久連線**：Pusher 透過 **WebSocket** 技術建立持久性的連線。
- **守衛角色（Watchdog）**：當 Serverless 伺服器因為執行完畢而「睡覺」斷線時，Pusher 就像是一個 **24 小時不睡覺的守衛**，在中間幫雙方（或多方）維持訊息的持續傳遞。它解決了無伺服器架構無法進行長連線即時通訊的痛點。

### 3. 主要產品與功能區分

Pusher 提供兩種主要的通訊模式，開發者需根據需求選擇：

- **Pusher Channels（推薦用於雙向溝通）**：這是透過 WebSocket 實現的**雙向即時溝通**，適合用於即時貼文更新、按讚通知、聊天室互動等場景。
- **Pusher Beams（單向推播）**：主要用於**單向的推播通知（Push Notifications）**，例如你在手機上收到 LINE 的廣告或訊息預覽，這屬於伺服器主動推給客戶端的反向溝通。

### 4. 開發與部署流程

使用 Pusher 的過程相對直覺，但需要精細的設定：

- **帳號與 App 建立**：開發者需在 Pusher 官網註冊並建立一個專屬的 App（選擇 Channels 形態）。
- **API 金鑰管理**：建立後會獲得一組 **API Key**，開發者必須將這些金鑰複製到專案的環境變數（`.env` 檔案）中。

- **雲端整合：**當專案部署到 Vercel 時，這些金鑰會讓雲端的 Serverless Functions 能夠正確與 Pusher 伺服器溝通，進而達成全網域的即時互動。
- 

為了讓你更好理解，我們可以把 Pusher 比喻成「全天候運作的電信轉接中心」：

你的 Serverless 後端就像是一個「**只有接到電話才進辦公室**」的職員，處理完事情就下班回家休息了。如果這時有人想傳急件給他（即時訊息），根本找不到人。而**\*\* 就是那個「24 小時值班的轉接中心」\*\***。不論職員（伺服器）在不在辦公室，轉接中心都幫你拉好了專線。一旦有新訊息，轉接中心會立刻把訊息送到對應的客戶端手上，確保訊息傳遞不會因為職員下班（伺服器斷線）而中斷。

為什麼 Serverless 架構需要配合 Pusher 實現雙向溝通？

在 Pusher 產品中，Channels 與 Beams 有何區別？

傳統伺服器與 Serverless 架構在計價方式上有何不同？

Discuss 軟體規格撰寫。

撰寫軟體規格是一門極致的「**正功夫**」，它不僅是定義功能，更是軟體開發成敗的關鍵基礎。根據來源內容，我們從以下幾個面向探討軟體規格撰寫的重要性與實踐：

### 1. 規格定義的深度與細緻度

許多人會認為像 Twitter (X) 這樣的應用程式邏輯很簡單，但實際上，一個看似簡約的服務背後隱藏著無數的細節（即「妹嘎」），例如：

- **介面與體驗：**按鈕的順序、UI 的配置如何影響 UX。
- **資料邏輯：**資料該如何儲存、如何排序，以及註冊、登錄、發文、閱讀與即時互動等基本功能的細節處理。
- **邏輯判斷：**規格撰寫者必須仔細思考、判斷每一個功能流程，否則一旦實作錯誤，事後修改將會造成巨大的開發成本。

### 2. 克服「缺乏細節」的通病

來源中指出，學生（或初學者）在撰寫規格時，常面臨**缺乏耐心與細緻度**的問題。

- **常見誤區：**往往只寫了一兩頁大方向的說明就結束，雖然方向正確，卻沒有照顧到執行層面的細節。
- **精進方向：**真正的專業在於能夠將一個東西的規格寫得**清楚且富有邏輯**。

### 3. AI 時代的新核心能力：自然語言程式設計

在現代開發環境中，「**自然語言程式設計（Natural Language Programming）**」已成為重要的技能。即便開發者未來可能不需要親自撰寫每一行程式碼，但必須具備以下能力：

- **精準描述：**能夠用文字（如英文或中文）將規格描述得非常詳盡。

- **避免 AI 漏洞：** 當你將需求交給 AI（如 Cursor）執行時，如果一次給予過多細節但結構定義不佳，AI 容易在實作中「漏掉幾個洞」，而事後要補這些漏洞往往非常困難。

#### 4. 規格撰寫與溝通管理

撰寫規格本質上也是一種**向上管理與問題解決**的訓練：

- **從問答題變是非題：** 在與老闆或 AI 溝通時，最糟糕的是丟出一個開放式的「問答題」（例如：這不能跑怎麼辦？）。
- **展現思考痕跡：** 好的規格與溝通應該是經過思考後給出「選擇題」，甚至是最省力的「是非題」，這代表你對問題本身已經投入了足夠的力氣去理解與定義。

---

**為了讓你更好理解，我們可以把撰寫軟體規格比喻成「編寫建築藍圖」：**

如果你只是隨手畫一個房子的外觀（大方向對了），建築工（AI 或開發者）雖然能蓋出形狀，但可能會發現漏了插座、排水管位置不對或門開錯方向。一份好的規格就像是**詳盡的精密藍圖**，它精確到每一根管線的走位和每一顆螺絲的規格。只有藍圖畫得夠細、夠清楚，蓋出來的房子才不會在完工後才發現「漏水」或「結構不穩」，導致必須拆掉重蓋。

比較傳統伺服器與 Serverless 架構的計費方式差異。

為何在 Serverless 架構中需使用 Pusher 來實現即時通訊？

試述在 Vercel 部署專案時，與 GitHub 綁定的優點。

Discuss 全端框架應用。

根據來源內容，**全端框架（Full-stack Framework）**如 **Next.js** 已成為現代網頁開發的主流趨勢。這類框架整合了前端與後端的邏輯，改變了以往需要分別維護兩個獨立框架的開發模式。

以下詳細討論全端框架應用的關鍵特性與組成要素：

##### 1. 現代化全端架構的整合趨勢

全端框架（如 Next.js）之所以被廣泛採用，是因為它能提供更完整的開發體驗，並優化服務架構。

- **整合開發模式：** 不同於以往將前端與後端拆分為兩個獨立框架，現代全端框架允許開發者在同一個專案中處理 UI 邏輯、API 路由及資料庫互動。
- **雲端部署優化：** 像 Vercel 這樣的平台對全端框架提供高度支援，能自動根據程式碼特性分流處理靜態頁面（透過 CDN）或動態功能（透過 Serverless Functions）。

##### 2. 全端應用程式的核心功能組件

要打造一個如 X (Twitter) 般的完整全端應用，除了框架本身，還需要整合多種雲端服務以達成「無伺服器（Serverless）」部署：

- **身份驗證（Authentication）：** 整合如 **NextAuth** 的服務，支援 Google、Facebook 或 GitHub 等多方登錄，這不僅提升使用者體驗，也方便開發者進行跨帳號的互動測試。

- **資料處理與資料庫**：對於具有高頻率新增、刪除及互動特性的社群應用，**NoSQL**（如 **MongoDB**）相比傳統的 **SQL** 表格結構更具彈性與效率。
- **即時通訊 (Real-time Interaction)**：傳統 **HTTP** 協議具有「被動回應」的限制，無法主動通知使用者新訊息。全端應用通常會引入 **Pusher** 這種第三方服務來達成 **WebSocket** 雙向溝通，實現按讚通知或即時貼文更新。
- **多媒體管理**：為了維持部署環境的輕量化，全端應用常將影音處理交給 **Cloudinary** 等第三方專門服務。

### 3. 開發實作的挑戰與要求

開發全端應用不僅是撰寫程式碼，更涉及複雜的**規格定義與邏輯思考**：

- **規格細緻度**：一個看似簡單的功能（如註冊、流覽個人網頁、發表文章）背後都有極多細節需要照顧，開發者必須具備寫好、寫對規格的能力。
- **CI/CD 自動化流程**：透過將全端專案與 **GitHub** 綁定，開發者只需推送程式碼（**git push**），系統便會自動在雲端完成部署與更新，實現持續整合與部署。
- **無狀態管理的挑戰**：在 **Serverless** 環境下，全端應用的功能執行是獨立且短暫的，開發者必須自行負責狀態（**State**）的存儲與提取，通常需配合雲端資料庫來達成。

### 4. 適用場景

全端框架特別適合開發 **API 服務**、**具有動態功能的網站**、或是需要**快速水平擴展 (Scaling)** 的應用程式，因為其架構能根據流量需求自動分配運算資源，達成按量計費的成本優勢。

---

**為了讓你更好理解，我們可以把全端框架應用比喻成一間「全功能的移動式餐車」：**

傳統開發模式像是你必須分別經營一間「中央廚房」（後端）和一間「店面」（前端），兩者之間還需要專門的運輸車隊來傳遞食物。而**全端框架**就像是一輛裝備齊全的**餐車**，廚房與櫃檯整合在一起，讓你可以直接在同一個空間完成料理並交給顧客。當生意太好（流量增加）時，你可以快速叫來更多相同的餐車（自動擴展）來支援，而不需重新蓋一座工廠。

為何 **Serverless** 架構適合處理 **API 服務**？

**Pusher** 的 **Channels** 與 **Beams** 功能有何差異？

在 **Vercel** 部署時如何處理環境變數？

**Discuss** 專案部署要求。

根據來源內容，專案部署（**Deployment**）在現代全端開發中不再只是最後一步，而是整個開發生命週期的核心。針對課程專案（如作業五與期末專案），部署的要求涵蓋了技術實作、時間時程以及維運管理等層面：

#### 1. 核心部署工具與流程

專案必須實現在雲端環境運行，而非僅限於本地端測試。

- **平台選擇**：來源建議使用 **Vercel** 進行部署，因其與 **Next.js** 框架高度整合且提供完整的 **Serverless** 支援。

- **自動化 CI/CD**：開發者需將 **GitHub Repository** 與 **Vercel 專案綁定**。一旦綁定完成，每次執行 `git push` 時，系統應能自動感應並在雲端完成重新部署。
- **環境變數配置**：部署時必須正確設定所有第三方服務的 **API Key** 與**環境變數**，包括資料庫（MongoDB URL）、認證服務（NextAuth）、即時通訊（Pusher）以及多媒體管理（Cloudinary）。

## 2. 時程與里程碑（Milestones）要求

為了避免開發者到最後一刻才處理部署問題，來源定義了明確的進度要求：

- **第 12 週：原型部署（Prototyping）**。學生需完成初步部署，即使功能尚不完整，也必須有可運行的網址供檢查（佔總分 3 分）。
- **12 月 2 日前：期末穩定版**。必須有一個穩定的版本上線，供其他同學進行互評（Review），此部分涉及社群互動的回饋（佔總分 5%）。

## 3. 評分與互動要求

部署的成功與否直接影響成績評定：

- **部署失敗的懲罰**：若部署失敗導致必須在本地端測試，成績將會大幅扣減（最高分僅能得 4 分，至少扣 2 分）。
- **雲端互評機制**：專案 review 將直接在雲端部署環境進行。為了模擬真實社群互動，每個人的專案至少需獲得 **5 位使用者的互動回饋**。

## 4. 技術細節與常見陷阱

部署要求中特別強調了細節處理，以確保服務在生產環境正常運作：

- **固定網址設定**：在設定 Facebook 或 GitHub 登入時，必須使用 Vercel 提供的 **固定 Domain**，而非帶有 Hash Key 的特定部署版本網址，否則更新後登入功能會失效。
- **開啟客戶端事件**：若使用 Pusher 進行即時通訊，必須在後台手動開啟 **"Enable client events"**，否則即時互動功能（如聊天室）可能無法正常運作。
- **日誌監控（Logging）**：開發者應學會查看 Vercel 後台的 **Runtime Logs** 以及瀏覽器的 Console 訊息，以便在雲端環境出錯時能提供精確資訊進行除錯。

## 5. 開發心態與管理

來源強調，成功的部署需要極大的**耐心與細緻度**。

- **不要 Panic**：面對雲端環境中機器或技術的各種不可控狀況，開發者應保持冷靜並仔細閱讀規格。
- **分階段開發**：建議將複雜專案切分成多個 **Milestones**，確保每個階段都能通過完整測試並成功部署（即 CICD 精神），而非一次性的大規模更新。

---

為了讓你更好理解，我們可以把專案部署要求比喻成「餐廳開幕」：



本地開發像是「在家試煮」，就算廚房亂一點或少個盤子也沒關係。但**專案部署**就像是「正式開張一間餐廳」。你不但要把菜做好，還得確認水電瓦斯（環境變數）都接好了、招牌（Domain）掛對了、服務生（Pusher）隨時待命。最重要的是，你不能在開幕當天才測試烤箱會不會動；你必須提前兩週就先進行試營運（Prototyping），確保客人（Reviewers）進來時，系統不會當場崩潰。

比較傳統伺服器與 Serverless 架構的計費差異。

說明在 Pusher 中開啟 Client Events 的重要性。

為什麼軟體開發建議採用 CICD 與原型開發？

Discuss CICD 開發精神.

**CI/CD（持續整合與持續部署）**在現代軟體開發中不僅是工具的應用，更代表了一種「**精進且持續迭代**」的開發精神。根據來源內容，這種精神的核心在於透過自動化流程，確保團隊協作的穩定性與產品上線的高效率。

以下是針對 CI/CD 開發精神的詳細探討：

### 1. 持續整合（Continuous Integration, CI）：協作與穩固

CI 的核心精神是**多人協作下的透明度與同步**。

- **頻繁 Check-in**：開發團隊成員應該隨時將更新的程式碼提交（Check-in）到共同的開發框架中，而不是各自閉門造車。
- **確保系統不崩潰**：透過頻繁提交，開發者必須確保自己的改動不會破壞整個系統，讓其他團隊成員隨時都能拿到最新的、可運行的程式碼進行開發。
- **單元測試的重要性**：為了確保 CI 順暢，開發者應學會將複雜功能切分為簡單的 APP 或單元進行測試，否則難以在整合過程中除錯。

### 2. 持續部署（Continuous Deployment, CD）：及早發現問題

CD 的精神在於**打破「最後一刻才部署」的惡習**。

- **雲端即時同步**：只要程式碼一有更新（例如執行 `git push`），系統就會自動感應並部署到雲端（如 Vercel），。
- **測試環境的真實性**：許多問題在本地端（Local）測試時不會發生，到了雲端環境才顯現。CD 精神要求開發者**及早將產品放上網**，即使功能尚不完整，也要維持一個隨時可測試的雲端版本，。
- **快速應變**：透過 CD，開發者能更快地根據雲端運行結果進行調整，避免到專案末期才發現架構不符需求而必須「砍掉重練」，。

### 3. AI 時代的敏捷節奏

在 AI 工具（如 Cursor）的輔助下，CI/CD 的節奏被大幅壓縮。

- **縮短開發週期**：傳統開發的一個衝刺週期（Sprint）可能是兩週，但在 AI 時代可能縮短到**兩天**。

- **里程碑管理 (Milestone)**：開發者必須將複雜專案切分成多個細小的里程碑，確保每個版本都是可 Demo、可測試的，這正是 CI/CD 精神的具體實踐。

#### 4. 工程師的心理素質與專業

CI/CD 流程中，自動化工具雖然強大，但最終仍取決於開發者的態度。

- **耐性與細緻度**：面對複雜的環境變數 (API Keys) 與部署問題，工程師需要有極高的耐性與休養，不能因為機器出錯就輕易陷入恐慌 (Panic)。
- **掌握背後邏輯**：即便 AI 能代寫程式碼，開發者仍需理解背後的部署配置 (如 `vercel.json`) 與架構，才能在自動化流程出錯時有效溝通並除錯。

---

為了讓你更好理解，我們可以把 CI/CD 精神比喻成「雲端共寫文件」：

傳統開發像是「大家各自在家寫草稿，最後一天才貼到一張大布告欄上」，結果往往發現內容重複、邏輯打架或版面全亂。而\*\* 就像是在「Google Doc 上共同編輯」\*\*：每個人寫一段話 (CI)，系統就自動儲存並確認文字沒有跑版，並且所有人都能即時看到最新版本；同時，這份文件會自動同步推播給讀者看 (CD)，讓你隨時能根據讀者的反饋 (雲端測試) 即時修正文字，而不必等到整本書印好才發現錯誤。

為什麼使用 serverless 架構開發 API 服務較具優勢？

在 Vercel 部署時，如何確保網址能指向最新版本？

為什麼開發社群互動服務時，NoSQL 比 SQL 更合適？

Discuss NoSQL 資料庫應用。

在現代全端開發中，**NoSQL (Not Only SQL)** 資料庫因其靈活性與擴展性，已成為處理高頻互動應用 (如社群平台 X/Twitter) 的首選技術。與傳統的關聯式資料庫 (SQL) 不同，NoSQL 採用非表格化的資料結構，能更有效地處理頻繁的新增、刪除與動態變化的資料需求。

以下根據來源內容，詳細討論 NoSQL 資料庫的應用特性、類型及其實作考量：

##### 1. 為何選擇 NoSQL？（對比傳統 SQL）

- **動態架構 (Schema-less)**：傳統 SQL 在建置時必須定義好固定的表格欄位 (Schema)，若上線後要新增欄位，通常需要進行複雜的「資料遷移 (Data Migration)」。
- **而 NoSQL (如 MongoDB) 允許資料欄位動態變化**，甚至一開始 Schema 是空的也沒關係，能隨著應用需求隨時調整。
- **高效處理頻繁變動**：傳統表格結構將資料連續存儲，若經常需要刪除或修改中間的資料，效率較低。NoSQL 則設計用於處理大量的 **User Generated Content (使用者生成內容)**，適合高頻率的社群互動場景。

##### 2. 四大常見 NoSQL 類型與應用場景

來源將 NoSQL 歸納為四種主要類型，各司其職：

- **文件存儲 (Document Store)**：

- **代表：** MongoDB、Google Firebase。
- **特性：** 資料以類似 JSON 物件的文件 (Document) 格式存儲。
- **應用：** 最適合社群網站、部落格或內容管理系統。
- **鍵值對存儲 (Key-Value Store)：**
  - **代表：** Redis。
  - **特性：** 存取速度極快，通常作為 **快取 (Cache)** 使用，存放在記憶體 (In-memory) 中。
  - **應用：** 提升資料庫查詢效率，減少對硬碟存取的依賴。
- **寬行存儲 (Column-base Store)：**
  - **代表：** Cassandra、HBase。
  - **特性：** 結合了 NoSQL 的彈性與適合大數據分析的結構。
  - **應用：** 大數據分析、需要頻繁執行 CRUD (增刪查改) 且兼顧統計需求的場景。
- **圖形資料庫 (Graph-base Store)：**
  - **代表：** Facebook 使用的技術架構。
  - **特性：** 利用「圖形 (Graph)」來串連使用者與文章之間的關係，並配合 **GraphQL** 進行增量更新。
  - **應用：** 處理上億人規模的高速互動、推薦系統及社群關係網絡。

### 3. Serverless 架構下的 NoSQL 實作：MongoDB Atlas

在 Serverless (如 Vercel) 環境中，由於雲端功能 (Functions) 具備**無狀態 (Stateless)** 且執行完即關閉的特性，資料無法存在本地端伺服器。

- **雲端託管 (DBaaS)：** 開發者通常使用 **MongoDB Atlas** 等雲端版本。它透過 API 或連接字串 (Connection String) 與 Serverless Functions 溝通。
- **層級結構：** 在 MongoDB 中，結構層級為：**伺服器 > 資料庫 (Database) > 集合 (Collection, 相當於 Table) > 文件 (Document, 相當於 Row) > 欄位 (Field)**。

### 4. 實作考量與挑戰

- **資料邏輯設計：** 雖然 NoSQL 具備靈活性，但開發者仍需在初期想清楚資料邏輯。例如在開發聊天室時，需要區分 user、message、conversation 以及處理群聊成員的 conversation member，以避免後續修改困難。
- **安全性與存取控制：** 使用雲端 NoSQL 時，必須設定網路存取權限 (Network Access)，例如限制僅有特定的部署伺服器或本地 IP 可以連線，並使用環境變數 (.env) 妥善保管資料庫金鑰。

---

為了讓你更好理解，我們可以把資料庫的選擇比喻成「筆記本與表格」：

**SQL** 就像是一張印好的正式報表，每個格子都已經畫好標題，如果你寫到一半發現想多加一欄資訊，就必須整張撕掉重印（資料遷移）。而 **NoSQL** 就像是一本空白筆記本，你可以根據每一筆記錄的需求，隨意增加註記或改變排版。這對於每天都有新花樣、訊息量龐大的社群媒體（如 X 或 Facebook）來說，空白筆記本（NoSQL）提供的彈性顯然比死板的報表（SQL）更能應付快速變動的需求。

比較傳統伺服器架構與 Serverless 架構的計價差異。

說明 Pusher 在 Serverless 環境中扮演的角色。

簡述 NoSQL 資料庫中 Document Store 的特性。

Discuss 即時通訊實作。

在現代 Web 開發中，實作**即時通訊（Real-time Communication）**是打造社群平台（如 X 或 Messenger）的核心。根據來源內容，即時通訊的實作涉及協議選擇、架構限制的突破以及第三方服務的整合。

以下詳細討論即時通訊實作的關鍵要素：

#### 1. 通訊協議：HTTP 與 WebSocket 的差異

實作即時通訊的首要任務是理解底層協議。

- **HTTP 的侷限性**：來源形容 HTTP 協議非常「懶惰」，它採用被動回應模式，只有當客戶端主動發送請求時，伺服器才會回傳資料。這導致伺服器無法主動通知使用者 B 有來自使用者 A 的新訊息。

- **WebSocket (WS) 的優勢**：WebSocket 是一種雙向通訊協議，它會在網頁上開啟一個「窗口」，且連線會維持開啟（Remain open）。這使得資料傳輸能達到及時性，讓伺服器能主動推播訊息給客戶端。

#### 2. Serverless 架構中的挑戰與解決方案

在 **Serverless（無伺服器）** 架構（如部署在 Vercel 上）中，實作 WebSocket 面臨重大挑戰：

- **無狀態與自動斷線**：Serverless 功能是「有需求才啟動」，執行完邏輯後就會自動關閉，不保留狀態，也無法維持 24 小時的長連線。

- **Pusher 的角色**：為了在會「睡覺」的 Serverless 環境中維持通訊，必須引入 **Pusher** 這種雲端即時通訊服務。Pusher 充當一個「不睡覺的守衛（Watchdog）」，在中間幫雙方維持持久連線與訊息傳遞。

#### 3. Pusher 的產品選擇與設定

實作時需根據需求選擇 Pusher 的產品：

- **Pusher Channels**：透過 WebSocket 提供**雙向及時溝通**，適用於聊天室、按讚通知或及時貼文更新。

- **Pusher Beams**：僅用於**單向推播**（如手機廣告或訊息預覽），屬於反向的單向通知。

- **關鍵設定**：在 Pusher 後台實作聊天室功能時，必須手動開啟「**Enable client events**」，否則及時互動功能可能無法正常運作。

#### 4. 資料與多媒體處理邏輯

即時通訊不僅是訊息傳遞，還包含資料儲存與效能優化：

- **NoSQL 資料庫**：由於社群應用涉及頻繁的新增、刪除與互動，使用 **MongoDB** 等 NoSQL 資料庫比傳統 SQL 表格更具彈性與效率。
- **多媒體分離**：為了讓部署環境保持輕量，圖片或影音資料不應直接存在自己的伺服器，而是交由 **Cloudinary** 等第三方 CDN 服務管理，資料庫僅儲存網址。

## 5. 功能實作細節

一份完整的即時通訊服務（如來源提到的 Messenger Clone）應包含：

- **訊息流管理**：使用 Array of messages 存儲對話內容。
- **狀態顯示**：利用 Pusher 實作「已讀狀態」與「正在輸入中（Typing indicator）」的及時反饋。
- **對話邏輯**：區分單聊與群聊。群聊邏輯較為複雜，需精確管理成員（Conversation Member）的加入與離開，以避免資料權限衝突。

---

為了讓你更好理解，我們可以把即時通訊實作比喻成「對講機系統」：

傳統 HTTP 就像是「寫信掛號」，收件人（伺服器）只有在收到信時才會回信，你得一直跑門口看有沒有回音。而在 Serverless 架構下，你的辦公室（伺服器）甚至會定時熄燈下班。實作\*\* 就像是安裝了一套「全天候衛星對講機」\*\*：不論你的辦公室是否開燈，衛星（Pusher）永遠都在線上，當有人講話時，它會立刻把聲音傳到你的對講機（客戶端），確保通訊永不中斷。

為什麼 Serverless 架構適合開發 API 服務？

請比較 HTTP 與 WebSocket 的通訊機制差異。

在 Vercel 部署時，如何確保網址指向最新版本？

Discuss 專案管理實務.

在軟體開發中，**專案管理實務**並非單純的排程，而是一套結合了規格定義、開發節奏控制、自動化流程與溝通技巧的綜合藝術。根據來源內容，即使在 AI 工具（如 Cursor）普及的時代，專案管理的複雜度反而可能因工具進步而提升，開發者仍需掌握核心管理精神。

以下是專案管理實務的關鍵面向：

### 1. 規格定義的「正功夫」

撰寫軟體規格被視為開發者的核心功力。

- **細緻度與耐心**：專業的規格不應只是大方向的描述，而需涵蓋 UI/UX 順序、資料儲存邏輯、按鈕判定等無數細節。缺乏細節的規格會導致實作錯誤，事後修改的成本極高。
- **針對 AI 的規格切分**：由於 AI（如 Cursor）有上下文視窗（Context Window）的限制，管理專案時不能將龐大需求一次丟給 AI，否則容易出現漏洞且難以補救。必須按照邏輯結構定義好規格，引導 AI 逐步實作。

### 2. CI/CD 與敏捷開發節奏

現代專案管理強調 **CI/CD（持續整合與持續部署）** 的精神，確保產品始終處於可運行狀態。

- **持續整合（CI）**：團隊成員應頻繁提交程式碼（Check-in）到共同框架中，並確保每一次更新都不會破壞系統（Break the system），讓他人能隨時拿到最新的穩定版本。
- **持續部署（CD）**：提早並頻繁地將專案部署到雲端（如 Vercel），而不是等到開發完成才進行。這能幫助開發者及早發現本地端與雲端環境之間的差異或架構設計錯誤。
- **AI 縮短的 Sprint**：傳統開發週期（Sprint）約為兩週，但在 AI 輔助下，一個循環可能縮短至兩天。專案管理應適應這種高速節奏，確保每個版本都是可 Demo、可測試的。

### 3. 里程碑（Milestones）與單元測試

將複雜專案切割成多個 **Milestones** 是成功的關鍵。

- **階段性任務**：例如在第 12 週完成原型部署（Prototyping），確保基礎設施運作正常，再逐步加入功能。
- **單元測試概念**：引入新技術（如即時通訊服務 Pusher）時，應先建立簡單的測試專案進行「單元測試」，確認功能無誤後再整合進複雜系統，以降低除錯難度。

### 4. 版本控制與團隊協作

有效的版本管理能避免專案陷入混亂：

- **分支管理（Branching）**：建議開設 develop 分支進行開發與團隊同步，只有確認穩定的版本才合併（Merge）到與部署連結的主分支（如 main），以防開發中的程式碼直接影響線上服務。
- **標籤與紀錄**：適當地使用 Git Tag 標記版本，以便在專案出錯時隨時回溯到特定狀態。

### 5. 溝通與向上管理

專案管理也包含對人（老闆、助教或 AI）的溝通：

- **精準提問**：溝通應從「問答題」進步到「選擇題」，甚至「是非題」。這代表開發者已對問題進行過沙盤推演，並提供邏輯判斷供對方決策，能大幅提升管理效率。
- **日誌監控**：管理者應學會利用 Runtime Logs 和瀏覽器控制台（Console）監控服務狀態與使用者行為，將數據作為優化產品品質的依據。

---

**為了讓你更好理解，我們可以把專案管理實務比喻成「大型樂高建築工程」：**

傳統管理像是照著說明書一步步蓋，但現代全端開發更像是「**一邊設計一邊蓋**」。你不能等蓋到五層樓高才測試地基穩不穩（這就是 CD 部署精神），而是每蓋好一個區塊就要推出去展示（Milestones）。如果發現零件不合，你必須學會把藍圖切小，先蓋一小個模組測試（單元測試），並確保每位工人的進度都能隨時對齊，不會互相卡住（CI 整合）。最終，專案經理的角色不是親手去拼每一塊積木，而是確保那張「精密藍圖（規格）」畫得夠準，並能隨時調整工序來應對突發狀況。

試比較 Serverless 架構與傳統伺服器架構的計價方式差異。

為什麼在 Serverless 環境下建議使用 Pusher 提供即時通訊？

進行專案開發時，CI 與 CD 分別代表什麼核心概念？