微服務建議指南

金融科技應用

目　錄

BROBRIDGE

圖目錄

版本控制

日期	版本號碼	作者/更新人員	備註
2019 年 6 月 30 日	v1.0	Fred	初始發行版本

Brobridge Co., Ltd
7F., No.58, Ln. 188, Ruiguang Rd., Neihu Dist., Taipei City 114, Taiwan (R.O.C.)
Tel +886 26579798
Fax +886 26579010

寬橋有限公司
臺北市內湖區瑞光路 188 巷 58 號 7F
電話：+886 26579798
傳真：+886 26579010

1 背景介紹

近年來，微服務（Micro-service）已然是熱門話題，也是各企業的重點達成的目標。歸功於微服務架構好處相當多，可以為企業及各種線上服務業者，帶來資訊服務處理容量的提升、更良好的軟體設計彈性、極佳的效能擴充性。

然而，微服務不是一個簡單的議題，並非單一從 IT 基礎建設（Infrastructure）或軟體開發（Software development）的角度，就能順利將舊系統轉換成新的架構，而是要同時考量兩者，雙管齊下才有機會成功轉置。

本文件將提出建議指南，協助有微服務化需求的企業或組織，評估自己當前的狀態，以更妥善的策略進行技術導入，無論最終是選擇需要授權的商業化工具和解決方案，還是採用社群的開放原始碼方案，都可以實現微服務化的轉置需求。

2 微服務帶來的改變

採用微服務架構，不單單是基礎設施（Infrastructure）的改變，也是軟體程式架構（Software Architecture）的改變，也會影響維運、開發與交付上線的工作規範。最終，微服務化的導入，將會改變資訊系統的整個面貌，在真正開始之前，建議先理解當全面微服務化後的樣貌。

2.1 微服務架構生態系統

在完整導入微服務架構後，所有相關的基礎設施和系統設計，都將會圍繞在「服務實例（Service Instance）」上進行，並以容器做為管控實例的解決方案。

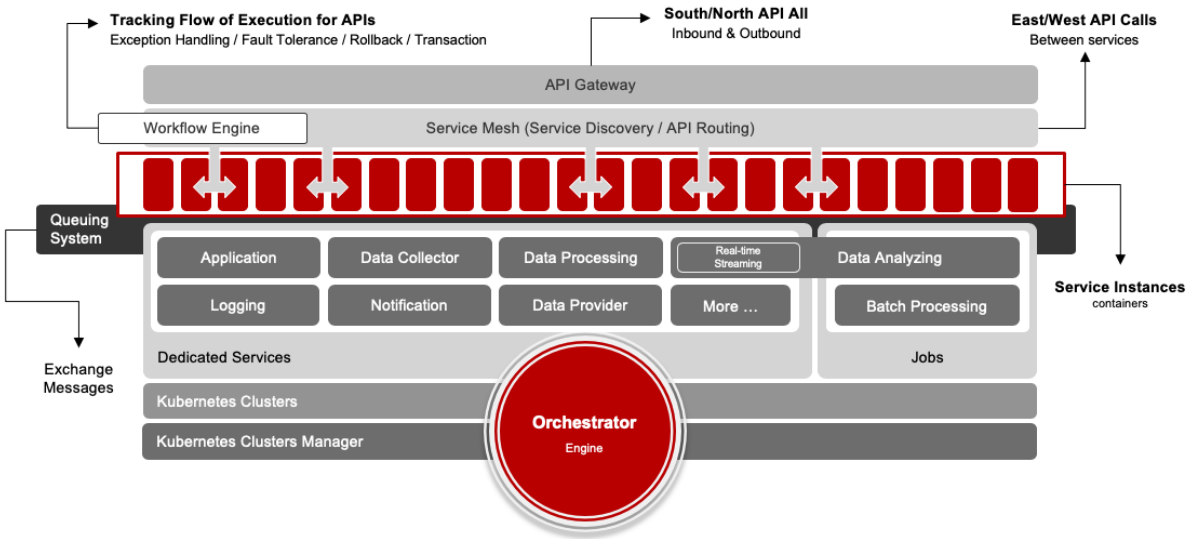

圖 1 完整的微服務架構生態

2.2 微服務 API 控管

所有的微服務都是依靠 API 與外界（南北向）以及服務之間（東西向）相互溝通，因此對外將採用 API Gateway 進行 API 的存取管控。而內部服務之間的東西向存取管控，則屬於服務網（Service Mesh）的設計範疇。

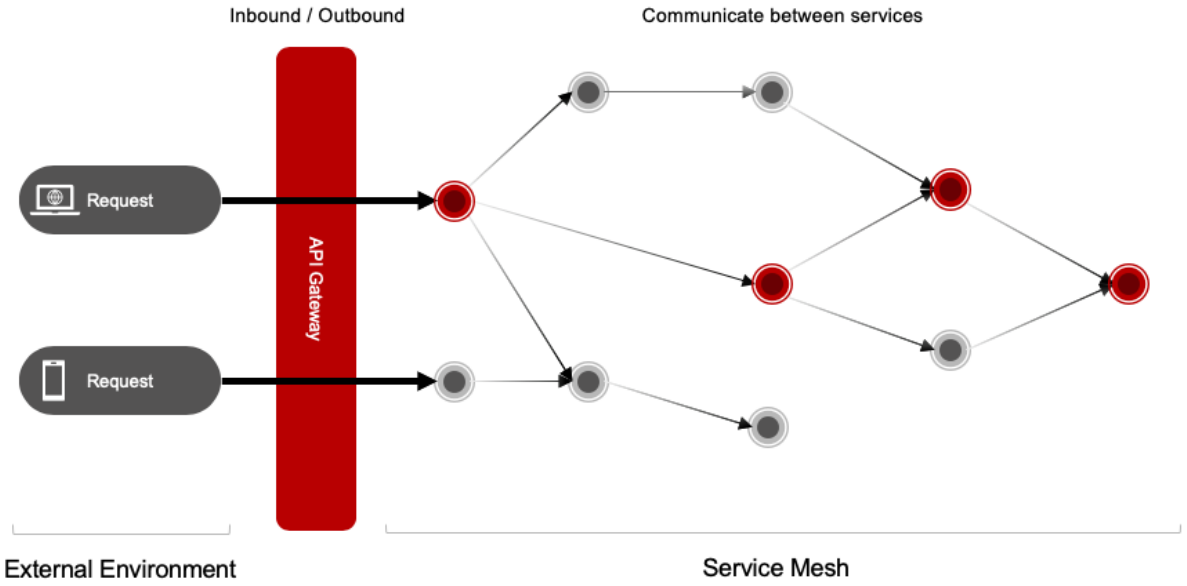

圖 2 API 呼叫的實際情境

2.3 微服務 API 閘道器（API Gateway）

應用程式對外的窗口，統一都由 API 閘道器（API Gateway）來管理，所有的進出呼叫存取，都需要經過權限等安全性檢查，並進行行為的監督和紀錄。一個符合微服務架構需求的 API 閘道器，除了基本的 API 存取控管、流量監控之外，通常需要跟容器進行整合，進一步結合服務網（Service Mesh）管控。現在業界的主流的解決方案，如：APIGEE、WSO2、Kong。

2.4 服務網格（Service Mesh）

傳統程式架構上，模組或元件之間的關係，可能透過程式內函式呼叫、系統 IPC 呼叫就可以達成，但在微服務架構上，應用程式被拆分成許多小型服務，以致模組或元件可能是

以服務實例的形式，分別運行在不同的容器或虛擬機之中，其服務之間的呼叫會透過 API 或 RPC（Remote Procedure Call）來進行溝通，而這些微服務間的溝通和呼叫路徑，便形成了如網一般的樣貌。

值得特別注意的是，服務網（Service Mesh）其實就是應用程式本身，也是的運作核心，當一個單體式（Monolith）的應用程式在解耦（Decoupling）之後，就會變成多個服務的分散式樣態。為了維護和監控應用程式，其中每個服務之間溝通的健康狀況，都需要視情況被調校和監督，甚至是調整其部署方式。這類的工作，通常會依賴相關的管理工具，例如：Brobridge Mesh Builder、Istio、Pivotal Service Mesh 等。

2.5 工作流程管控

由於在微服務架構中，每次的需求（Request）都是由分布在不同容器中的服務來分散處理，所以有導入流程管控引擎（Workflow Engnie）的必要，整個需求處理的工作流程被管控、追蹤，才能實現例外處理（Exception handling）、容錯（Fault Tolerance）、退回（Rollback）以及交易（Transaction）的工作。

2.6 容器及叢集系統的管理

微服務的實例通常都被打包在容器（Container）之中，利用著 Kubernetes(K8S) 容器管理平台進行部署及管控維護。因此，跟應用程式本身直接相關的高可用（High Availabilty, HA）需求，主要由 Kubernetes Cluster 負責處理。

但對於維運人員來說，Kubernetes Cluster 本身的穩定性、高可用性，就必須仰賴虛擬化技術來實現和管理。目前，可靠的解決方案如：Pivotal Container Service(PKS)、Brobridge Kubernetes Cluster Management 等，都可以實現以虛擬化技術來管理 Kubernetes Cluster。

2.7 彈性的資料管線架構

導入微服務架構後，許多工作被獨立切割出來運作，資料流有自己的獨立管道進行引流，因為訊息佇列系統（Message Queuing System）的幫助，資料被都被分門歸類在各種的主題上。應用程式若需要取得特定資料主題，只需要在資料流管道（Data Streaming Pipeline）中訂閱（Subscribe）即可。資料管線的設計，除了可以滿足傳統批次（Batch）型態的資料處理和分析工作，也可以實現即時資料流（Real-time Data Streaming）的需求。

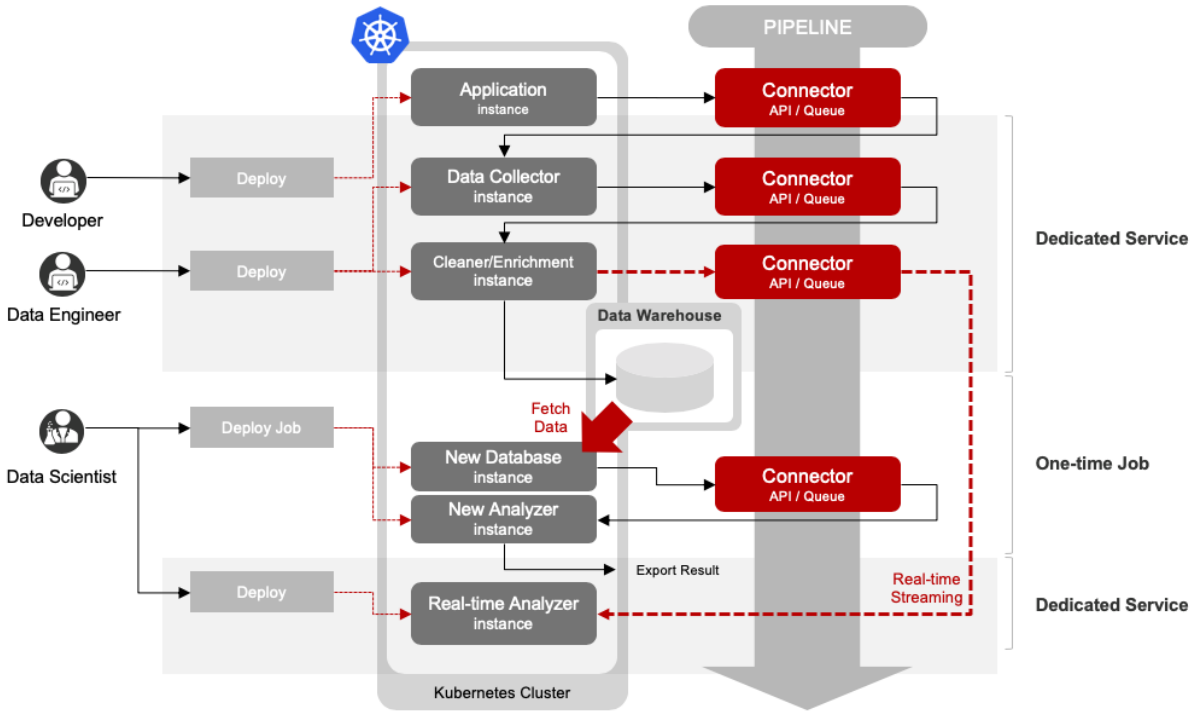

圖 3 不同類型任務與資料流之關聯性

2.8 常駐型服務（Dedicated Service）

常駐型服務指的是應用程式一旦部署到線上，就會長時間運作或待命，如需要提供對外服務的網站應用程式、監控程式、資料收集程式及即時分析程式皆屬此類。

2.9 單次型任務（One-time Job）

單次型任務指的是生命週期有限，應用程式從執行任務開始，直到工作做完就結束釋放資源，不會常駐於平台之中。在大數據應用中，單次型任務常用於批次資料的處理，然後計算出結果，為了讓計算的過程和結果的龐大數據量，不會嚴重衝擊既有的資料倉庫和資料庫系統，會建立臨時的容器環境來處理和儲存資料。

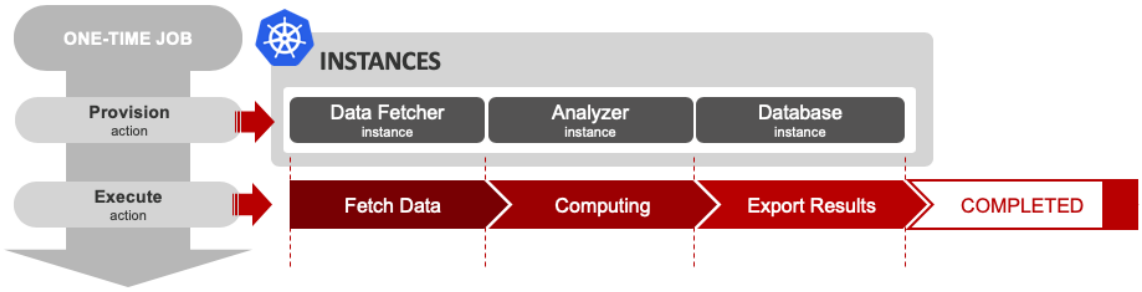

圖 4 單次行任務的生命週期

2.10 開發生命週期

以開發流程來說，相較於傳統的單體式（Monolith）架構，微服務架構（Microservice Architecture）的開發生命週期和維護顆粒度將會變小、更為獨立，對開發人員來說，在遵循開發規範下，只需要專心對付特定的功能、模組或特定領域的服務即可，無需考慮太多整體系統整合性（如：整合工作、整合測試、整體穩定性）的問題。

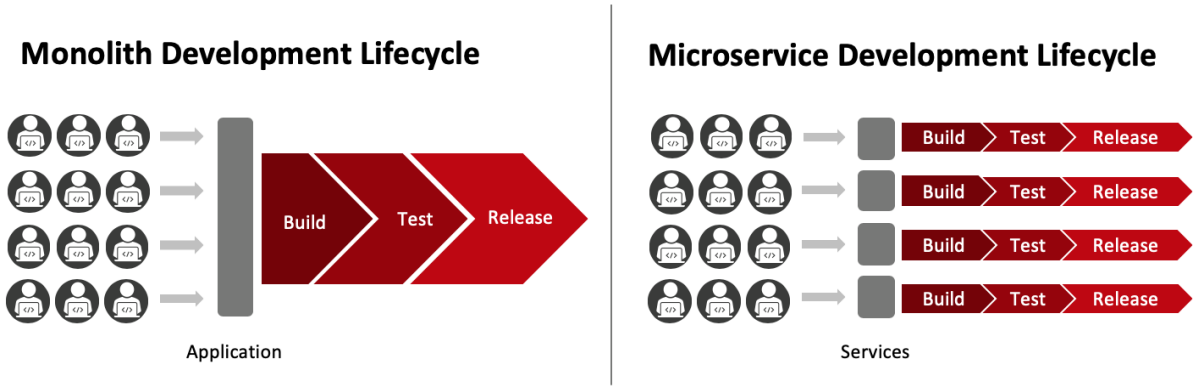

圖 5 傳統架構與微服務架構的開發生命週期之比較

由於所有服務元件相互之間獨立運作，任何的改動與升級，都會不會相互間影響，在最壞的情況之下，也只會影響部分系統以及依賴該系統的部分功能。在測試工作上，無需每次更新都再對整個系統從裡至外的測試和檢驗，只需要針對特定區塊的服務進行檢測即可。

對開發人員的負擔上，壓力會更少，更容易實作新功能、修復問題以及更快進行開發迭代的工作。但是對維運人員來說，必須面對更多「微型服務」的管理工作，且必須要在維運層面上保證整體平台的穩定性。

3 導入策略的考量及規劃

微服務的主要方向，是從架構層面上對整個應用程式或線上服務進行翻新，在開發層面意味著可能會面臨設計上的大規模翻修，舊有的體制和維運政策都可能改變，甚至在未來變得不再適用。這樣的結果，也意味著在轉換的過程中，可能會有許多衝擊和抉擇。

想要避免衝擊和影響，實務上微服務化的工作，除了既有系統的軟體開發改版規劃需要考量外，也需要適當的相關基礎建設輔助和制度才能完成，其牽涉範圍之廣，涵蓋軟體開發人員、系統維運人員以及企業的資訊單位決策者，基本上幾乎所有相關資訊人員都要納入考量範圍。

3.1 不同角色的考量

最佳的導入策略，必須讓多方人員都能同時逐步接受、轉換並信任新的架構和工具。不宜貿然引入大量工具，妄想一次性升級舊有系統的架構。主導微服務化的人必須知道，微服務架構所帶來的彈性極大，若非循序漸進妥當引入，其彈性所帶來的災難和意外，將難以預防和解決。

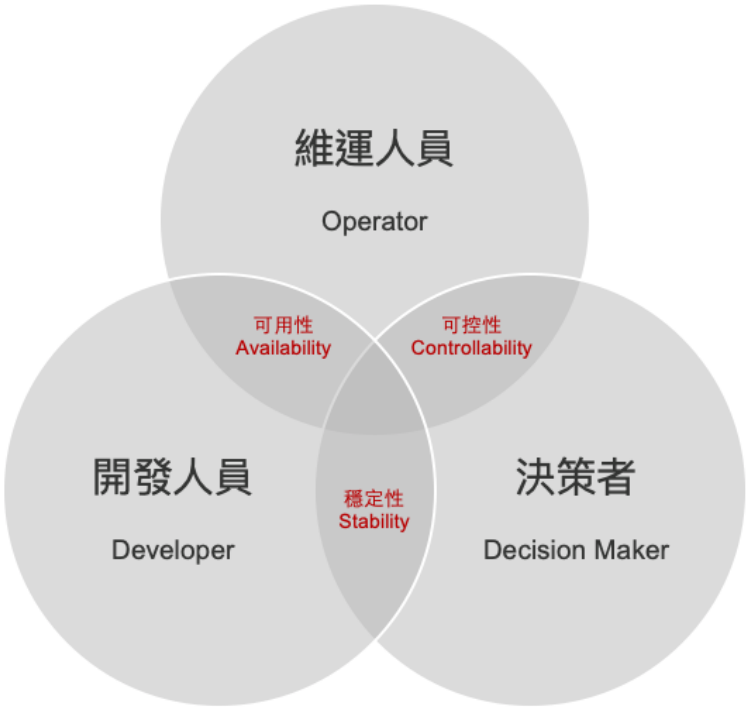

圖 6 不同角色之考量要點

不同人員的角度相互拉鋸，都會影響微服務架構最終的樣貌，甚至可能讓整個系統和制度扭曲、崩解，若導入一個工具會讓不同崗位的人員相互對立或是產生衝突，這就不是一個應該導入工具，或者是並非正確的導入時機。

所以在一切尚未開始之前，必須先瞭解每個每個人員角色所考量的重點以及困難，然後才能選擇一條較為不衝突的導入方案。

3.2 維運人員（Operator）

維運人員（Operator）肩負著整個資訊系統正常運作的重責大任，無論是軟體還是硬體問題，一旦發生而使系統不正常，維運人員是所有危機處理的第一關。因此，維運人員的首要任務重心，就是確保乘載應用程式的系統平台，有足夠可控性，以便於維運人員可以快速釐清問題所在，並有足夠方法做必要處置。
理論上，由於維運人員本身並不直接介入應用程式的開發，任何因為應用程式臭蟲所產生的問題，維運人員除了回報問題外，就是盡可能利用成熟的平台工具，來閃避問題的發生，或是減緩問題的擴大。

所以對於維運人員來說，「可控性（Controllability）」和「可用性（Availability）」是導入新架構或新系統時的必要考量。若可控性太差，維運人員只能雙手一攤，將會直接衝擊企業的商業營運；而可用性太差，無論大小問題都會變得嚴重，直接給開發人員帶來極大壓力。

3.3 開發人員（Developer）

開發人員（Developer）主要工作，是實作應用程式的各項功能，確保應用程式是以足夠品質的原始程式碼所組合而成，並使臭蟲數量降到最低，根絕所有的技術性問題，其所在乎的是應用程式之下平台的穩定性（Stability）以及可用性（Availabilty）。

平台有足夠穩定性，才能確保應用程式的開發、運行，能在理想、理論的環境狀態下運作，開發人員除了可以專心於程式架構、功能設計之上，當有問題發生時，更能容易定位問題發生點。而平台可用性，才能讓開發人員不必擔心太多維運問題，讓部署（Deployment）、測試（Testing）等例行工作，可以自動化或交由維運人員（Operator）、其他非開發人員處理，也能有更多緩衝時間來進行臨時的問題處理。

對於開發人員，能夠專心於開發工作，不用擔心測試流程、上線部署，更不用擔心運行後穩定性問題，是最重要的。

3.4 決策者（Decision Maker）

決策者關注的是整體系統的穩定性及可控性，確保整個 IT 服務可以穩定支撐企業的業務發展，也能針對各種系統維運的潛在風險，進行監督、控制和調節。由於在微服務架構的導入過程，多少會衝擊既有已經穩定運行的系統、體制和人員業務內容，導致整個系統的不穩定風險提高，這是決策者最不樂見的狀態。

若想要減少不穩定的風險，提高可控性是主要訴求，在正式導入微服務之前，提前導入正確的管理工具和解決方案，然後輔以訓練和新工作規範制定，以確保新平台、流程、人員為可控的狀態，是最佳的做法。

3.5 人員參與度規劃

微服務架構主要是軟體開發人員的工作，而在軟體程式架構的開發開始之前，有一些前置作業要先準備，主要是管理工具的導入、人員的訓練以及既有系統的相關轉置工作。若希望順利完成前置作業，其參與人員、參與比重及進度，需要參照前一章節的「角色考量」來適度的規劃。

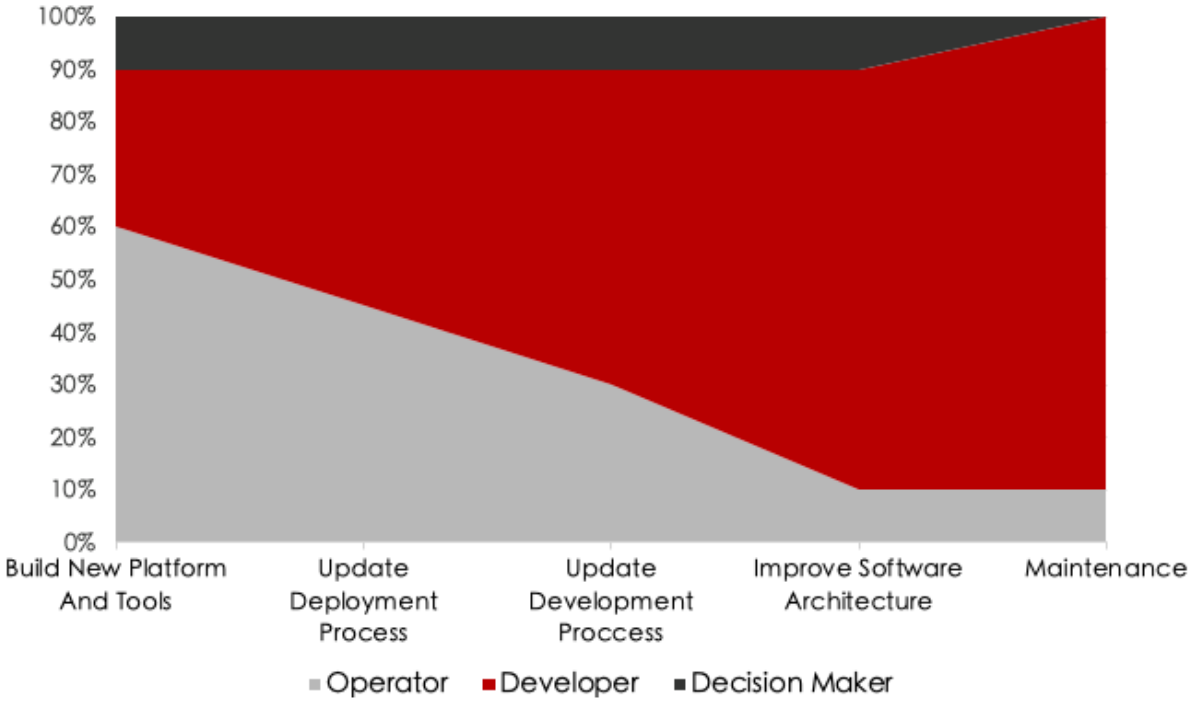

圖 7 不同導入階段的人員參與度分佈

一般來說，建議採取人員參與度規劃：

- 導入新平台及工具

  以維運人員為主要參與對象，開發人員協助調整和討論，以及建立相關維運政策與開發規範。
- 更新部署流程

  開發人員與維運人員分工將舊有應用程式納入新平台管理，使既有系統以容器形式運行和管空。
- 更新開發流程

  訓練開發人員轉移至新的開發測試流程，讓開發、測試及部署流程能順利整合，並以自動化取代大多數人力工作。
- 更新軟體程式架構

  主要由開發人員執行，開發人員具備新平台與流程的基礎使用能力，可以開始全面改進舊的程式架構，轉換成微服務架構。
- 維運與維護

維運人員負責基本例行維運工作，而開發人員則既繼續於新的架構上開發功能和修復問題。關於應用程式的主要工作，一樣回歸到開發人員身上。

4 舊系統微服務化五階段

舊系統有許多包袱，很難一下就全部轉成微服務架構，除了對軟體開發人員來說是一件艱鉅的挑戰之外，變成微服務架構後，數量更多的「子系統」也會造成維運人員的恐慌和管理困難。

建議採用四個步驟，階段性導入微服務的相關工具和轉置開發工作：

1. 基礎建設（Infrastructure）
2. 生產環境的自動化部署（Deployment Automation）
3. 測試環境（Testing Environment）
4. 持續交付及持續部署流程（CI/CD）
5. 舊系統微服務化的軟體開發（Application Development）

而所有工作，可以分為三個步驟完成：

1. 基礎設施的建置與導入
2. 測試環境及部署的流程整合
3. 舊系統軟體程式架構的變更開發

4.1 階段一：基礎建設（Infrastructure）

大多數企業，在這幾年多半已經從實體機環境轉換到虛擬化環境（physical to virtual, P2V），以便於管控「作業系統（Operating System）」層級的執行環境。在過去單體（Monolithic）架構設計下的應用程式，往往一個作業系統上能夠執行一個實例（Instance）就已經足夠。

然而，在微服務架構下，單體架構會被拆開變成多個微小服務，應用程式的實例數量會因此進一步提升。此時，需要擁有能管控「應用程式」層級的環境，才能應付「大量微型服務」的管理需求，且儘可能減少運算資源的消耗。

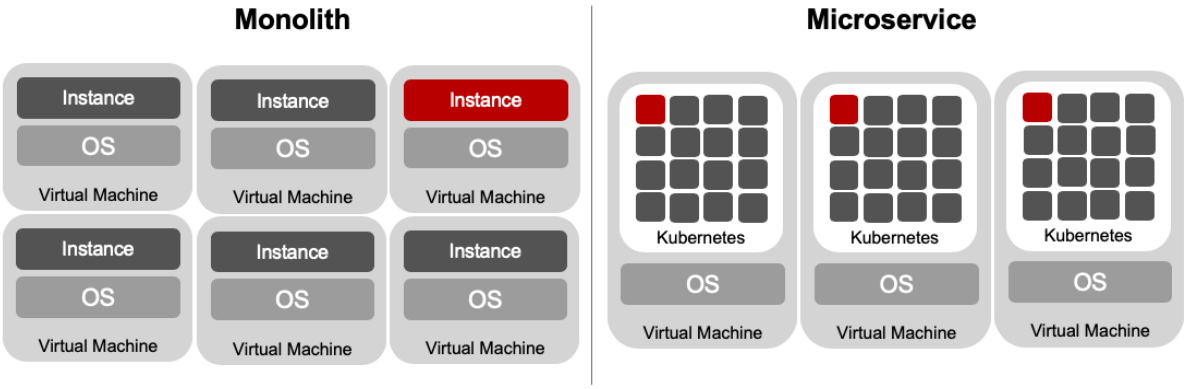

圖 8 單體式架構與微服務架構的實例管理

在這樣的需求下，通常就會建議導入容器技術（Container technology）以管理「巨量的服務實例」，而現在業界主流趨勢，皆採用 Kubernetes(K8S) 做為容器管理平台。

4.1.1 實現高可用性（High Availability, HA）

傳統單體式的應用程式架構搭配虛擬化的基礎設施，在實現高可用性（High Availabilty）的手段，多半仰賴著虛擬機本身的備援機制，透過複製多台虛擬機，當有問題發生導致某一台虛擬機不穩定時，以虛擬機切換的方式來達成。這樣的實現方式，主要針對於整個「作業系統環境」進行備援和容錯處置。

而採用容器技術後，關注的重心不再是作業系統環境而是「服務實例」本身，針對高可用性的實現，變成是針對服務實例（或應用程式本身）進行備援和容錯處置。這種架構之下，服務實例很容易在不同的虛擬機、實體機，甚至是跨雲的環境中進行快速開發。

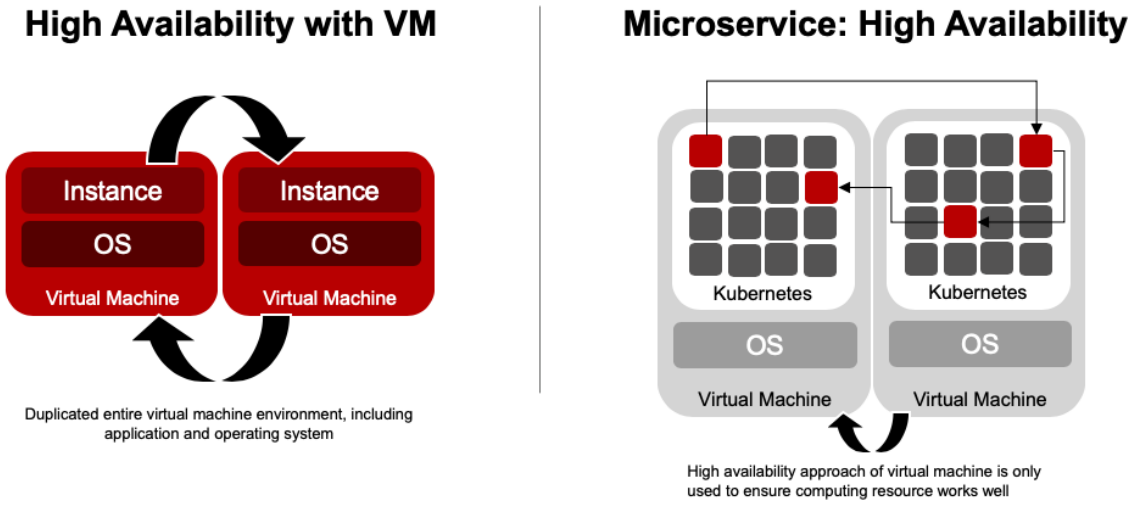

圖 9 傳統架構與微服務架構的高可用性設計

4.1.2 實現運算資源高利用率

傳統環境上，無論應用程式規模大小如何，作業系統都無差別的佔據了許多運算資源（CPU、記憶體、硬碟空間等），這導致有許多運算資源的浪費，此外超出應用程式需求的預防性虛擬機資源配置，也讓資源總是空閒。

在導入容器化後，由於應用程式會被包裝成容器，無需再跑在獨立的虛擬機之中，可大量減少虛擬機及 Guest OS 的數量，也免除了每個虛擬機保留的作業系統資源消耗，藉此提高運算資源的利用率。另外，緊密的服務實例管理安排，也能近一步減少閒置資源的數量。理論上，同樣的運算資源，在導入容器化後，能夠乘載更多的應用程式實例數量。

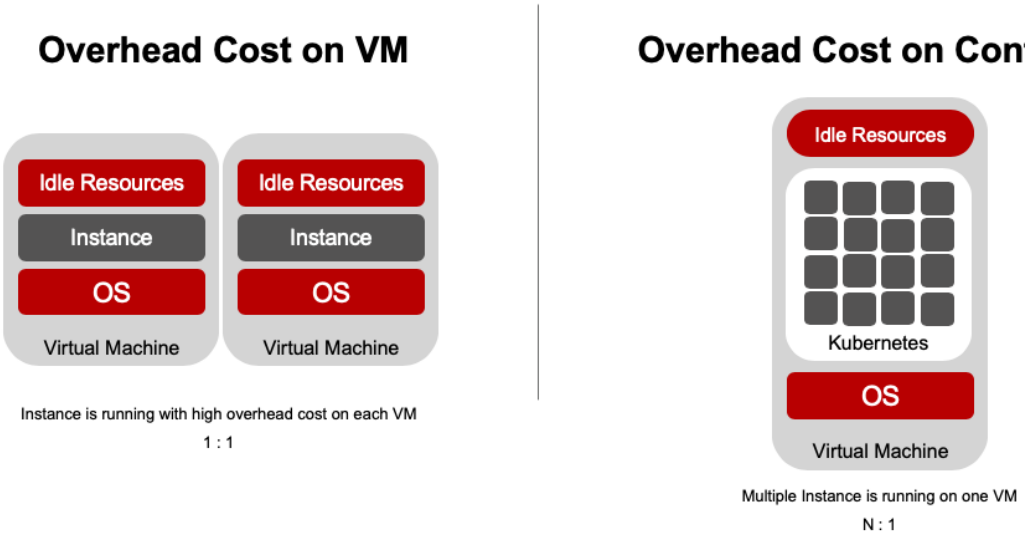

圖 10 傳統虛擬機架構與容器架構下，運行應用程式實例時的消耗成本

4.1.3 實現跨虛擬機與容器的網路管理

跨虛擬機與容器的網路管理，在企業級的使用情境下是一個困難點。導入容器化技術後，由於許多實例會跑在同一個節點上，這使得網路的規劃上相當困難。尤其是多半的網路管理工具，都只能管理到「節點」層級的網路政策，以前在「實例即虛擬機」時雖然夠用，但當「多個實例在同一個虛擬機」的情境，這些工具就顯得不足。

此外，實務上不太可能所有系統都容器化，導致現實中會同時有虛擬機和容器的存在，甚至是有多座 Kubernetes Cluster 存在，這種相對複雜的使用情境，在企業中也相當常見，使得網路管理更為困難。

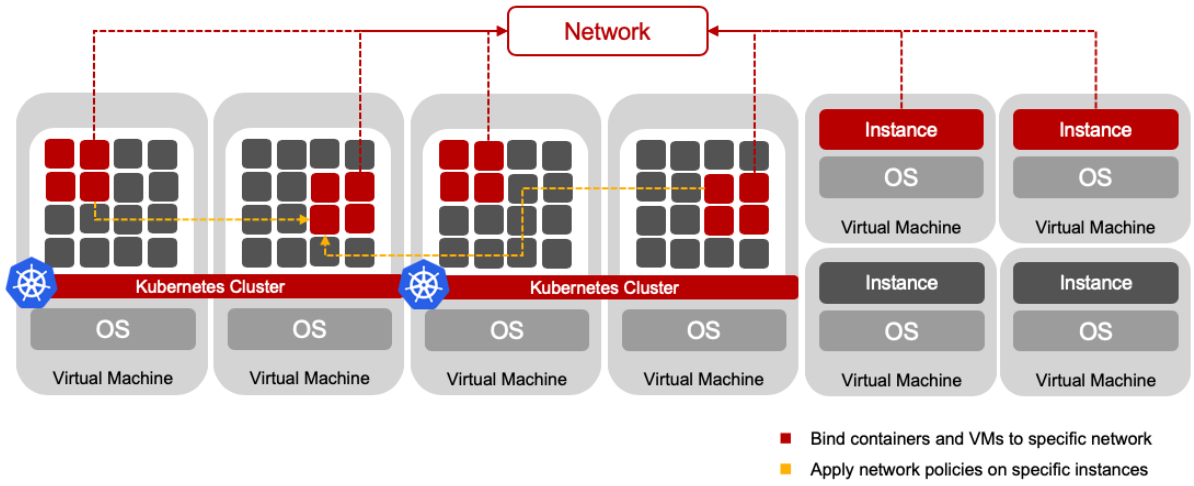

圖 11 資安及網路管理的需求

另外一個困難點是，在真正引入微服務架構後，許多的實例（無論是使用容器技術還是虛擬機技術）都是自動產生部署，難以手動一個個設定其網路政策及防火牆規則，即使真的有工具可以細緻的去管理所有的網路，也因為難以同時整合既有的容器和虛擬機自動化環境，使維運人員、資安人員大為頭痛。

網路是在導入容器技術時相當重要的議題，若有這類需求，業界較完整的解決方案有 VMware NSX-T，可以同時解決虛擬機、容器、實體機的網路虛擬化管理工作。

4.1.4 實現多叢集管理架構

若不是有管理需求，通常一個 Kubernetes 叢集就足以乘載所有的容器，對於小型專案或小型組織來說，搭建一套叢集就能滿足需求。但在企業級應用中，將所有雞蛋放在一個籃子裡風險相當高，無論是從資安層面還是從維運管理層面來看，在資源配置上也相對不夠彈性。這時多 Kubernetes 叢集的管理架構，就顯得相當需要，可以依據業務、服務層級、安全等考量，實際分割不同的容器運作區域而不相互影響。

實務上的例子，將開發、測試、正式環境分開就是典型的多叢集應用場景。而 GPU 的資源管理，通常也是常見的應用場景，因為 GPU 資源有限，為了管理方便，配置 GPU 的資源時會為其獨立設置一座 Kubernetes 叢集，將所有使用到 GPU 的應用程式，集中部署在這個叢集之中，與其他類型應用程式分開。

不過，手工架設一套 Kubernetes 叢集相當費時，架設完成後管理多個叢集也是一個莫大痛點，維運人員往往在管理這些叢集時，都遭遇了許多困難，例如網路架構問題、資安權限問題或是系統升級問題。

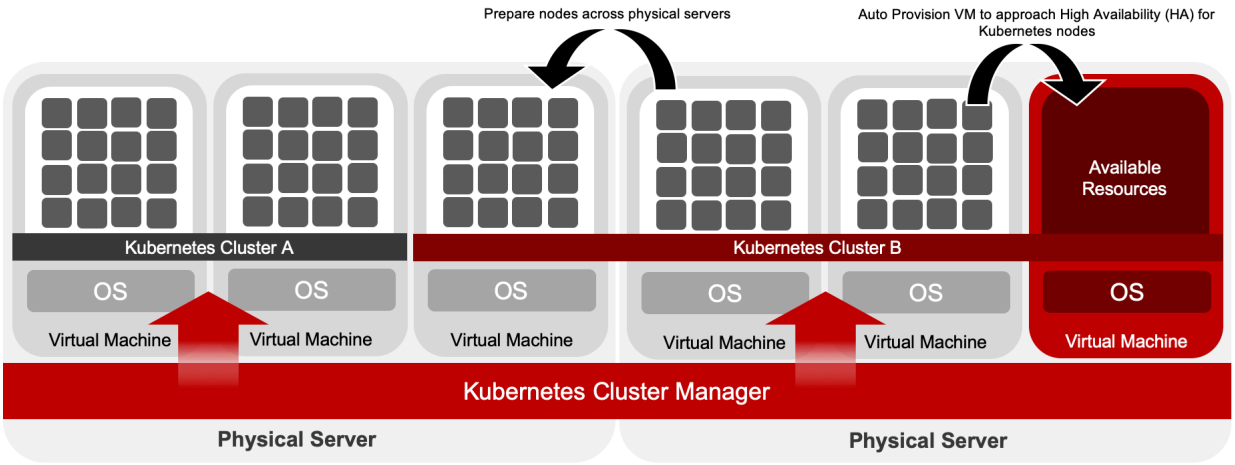

圖 12 善用虛擬化技術的多 Kubernetes 群集管理

如果採用了成熟穩定的虛擬化技術來管理 Kubernetes 叢集，則可以帶來更多的彈性，善用虛擬化技術進行補強和整合，過去的虛擬化網路管理機制、資安機制，就都能快速引入，而且可以快速部署多個叢集，實現許多自動化的維運工作（如：節點的 High Availability 機制）。

4.1.5 外置儲存架構

基於容器技術的應用程式版本管理，通常內容保證不可變（Immutable），這意味著所有容器映像檔（Container Image）每次被啟動時，都會生成一樣的環境和處於相同的狀態。所以，如果我們有狀態或資料需要儲存，通常不會存於容器之中，而是存放於外置的儲存裝置或平台中，每次容器被啟動時，就會將外置的儲存空間掛載，以存取過去的資料。

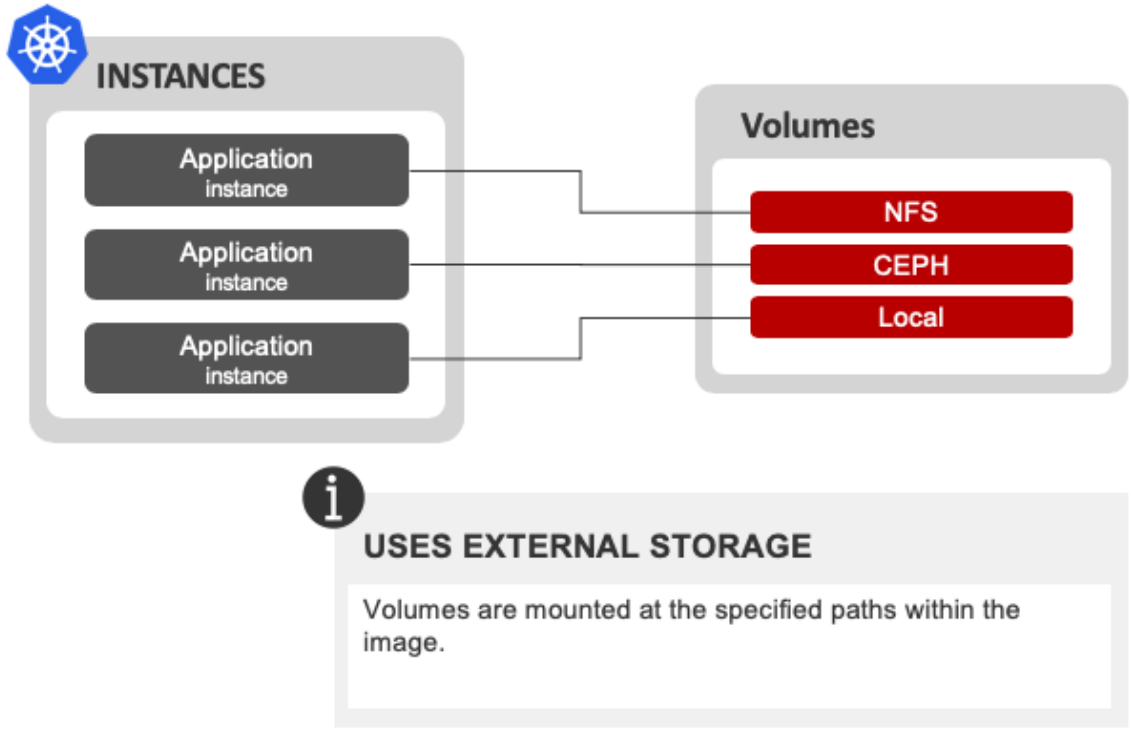

圖 13 Kubernetes 支援多種類型的儲存機制

Kubernetes 可整合掛載的外置儲存平台有很多，如 Ceph 就是常見的儲存叢集系統，或是任何允許透過 NFS(Network File System) 協定來掛載的儲存裝置。當然，若是沒有特別建置外置的儲存叢集系統，也可使用本機端的硬碟空間。

4.1.6 實現監控管理架構

容器平台的監控需要涵蓋叢集本體（Cluster）、節點（Nodes）、容器（Containers），很多情況，甚至需要做到更細緻，例如實例個體的監控、應用程式的狀態收集。為了滿足這樣多元的監控需求，彈性的監控管理機制就是必要的，CNCF 生態中的 Prometheus 專案就是一個針對這樣需求所設計的監控平台。

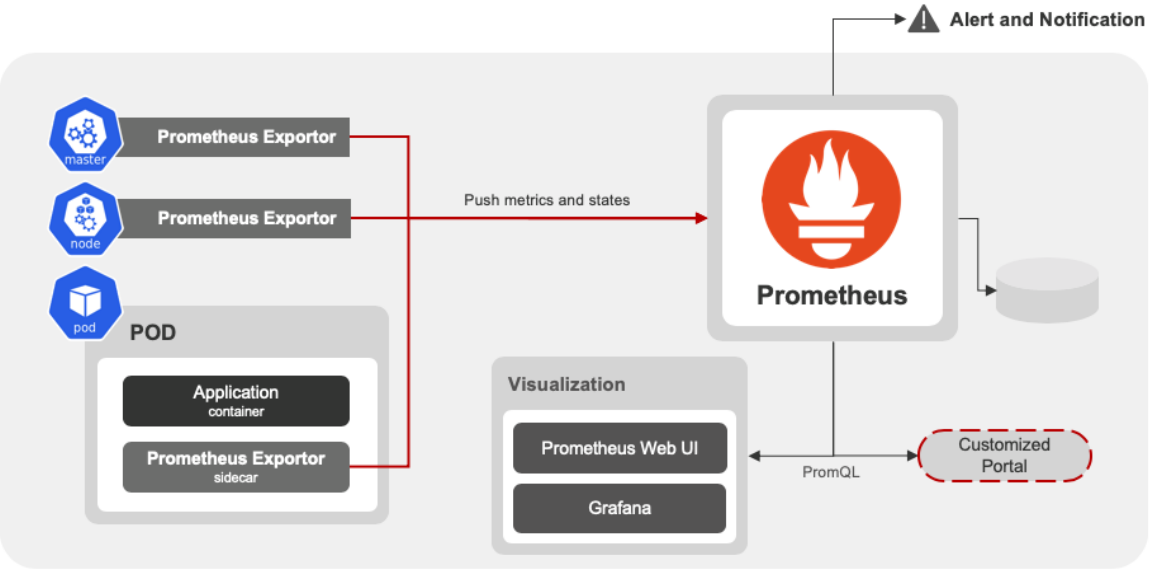

圖 14 採用 Prometheus 的監控管理架構

Prometheus 提供標準的 Exportor 介面，讓 Kubernetes 的各個元件都能推送系統資源、狀態、日誌等資訊，回到 Prometheus Server 進行收集。針對特殊的應用程式，若有監控需求，也可訂製 Exportor 進行監控資訊回送。另外，告警和通知是監控系統不可缺少的重要功能，Prometheus 也實現了 Alert Manager，可以串接電子郵件等告警訊息推播通道。

對於視覺畫呈現和管理工具，除了可以使用官方的 Promethues Web UI 之外，也可以整合業界常見的 Grafana 進行資料呈現。若有需要進階的資料查詢或是整合第三方應用程式，Prometheus 亦提供了 PromQL 的查詢語言和 API。

4.1.7 實現基礎設施容器化

對於基礎設施（Infrastructure）的管理維運人員來說，在微服務架構下，應用程式的實例管理是第一課題，傳統虛擬化技術在顆粒度上已經無法滿足，容器管理技術（Container technology）就有導入的必要。而對於已經全面虛擬化的企業來說，應該要關心的是如何從虛擬化轉容器化（virtual to container, V2C）。

無論是從實體機還是虛擬機至容器化，更新基礎設施的主要目標都相同：

1. 建立容器的管理平台與容器資源池

2. 導入容器化系統維運與管理

4.1.8 建置 Kubernetes(K8S) 容器管理平台

Kubernetes（簡稱：K8S）是近年來容器管理平台的首選，旨在提供「跨主機集群的自動部署、擴展以及運行應用程式容器的平台」，該系統由 Google 設計並捐贈給 Cloud Native Computing Foundation。

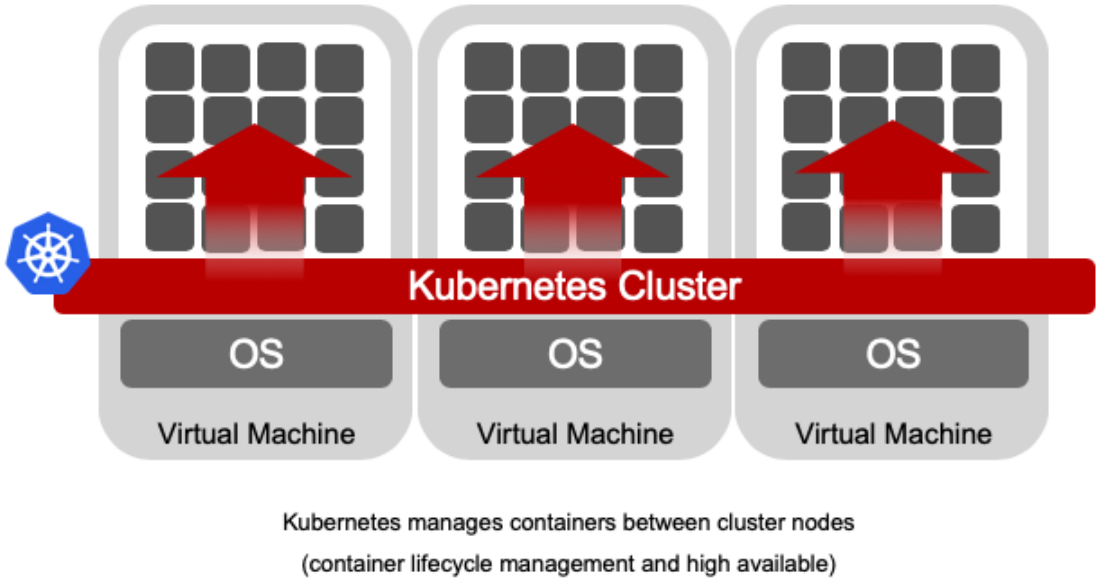

圖 15 運用 Kubernetes Cluster 管理容器生命週期及實現高可用性

建置 Kubernetes 叢集最小需求為三台節點（一台 Master Node、兩台 Worker Node），可以選擇在實體機（Bare Metal）或是在虛擬機（VM）上安裝。通常建議部署在虛擬機之上，可以繼承過去虛擬機的各種備份和高可用機制，來確保節點穩定和安全運行。

若是希望有更佳的彈性和多叢集配置的準備，並且與企業內既有的虛擬網路架構或虛擬機整合，可以採用業界的相關解決方案來部署 Kubernetes 叢集，如：Pivotal Container Service(PKS)。

4.1.9 轉置既有系統至容器平台

在微服務導入初期，雖然難以一次性將所有系統微服務化，但可以先將既有系統轉置於容器之中，主要目標先針對容器化系統管理及維運模式，進行轉型和適應。具體的轉置方法，是將應用程式重新包裝成「容器映像檔（Container Image）」，然後直接部署至 Kubernetes 叢集之中。

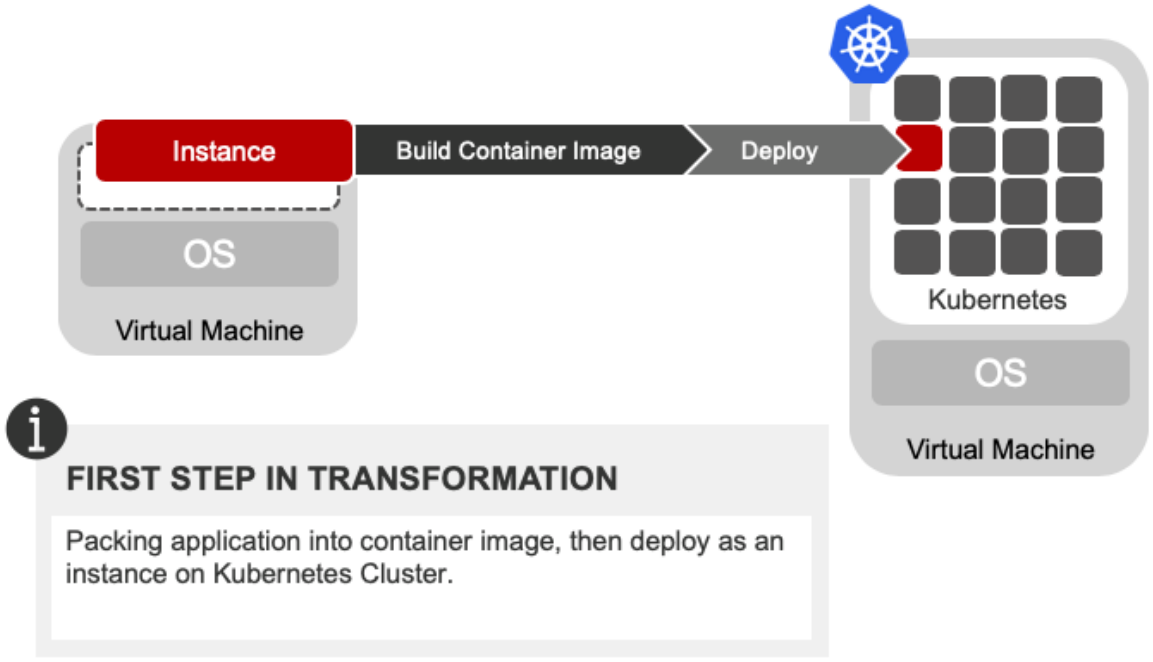

圖 16 容器化是轉置既有系統的第一步

需要特別注意的是，轉置之前我們將針對應用程式本身進行評估，確定其所需的運算資源、儲存資源以及網路環境等，這裡無須考慮「作業系統」本身的消耗，只需要考慮應用程式執行期的實例即可。

4.2 階段二：生產環境的自動化部署（Deployment Automation）

一旦既有系統都轉置於容器平台，所有系統的修改更新，都將與過去實體機或虛擬化平台不相同。一般來說，容器部署的流程為：

1. 程式原始碼打包編譯
2. 容器映像檔生成
3. 部署新版本容器映像檔至生產環境
4. 釋放舊版容器環境

部署流程包括許多繁雜步驟，對於開發人員與上版人員來說，都不是件輕鬆的事。尤其是，若開發與部署上版不是同一個人執行，過程中如何交接程式原始碼和確保部署設定正確，難以釐清責任。

因此，容器部署流程自動化是必要的規劃，使開發人員只需於程式原始碼，置入相應的容器設定檔，即可以觸發容器部署機制的自動化工作。搭建自動化部署機制的目標，主要是使開發人員開始正確使用容器平台，熟悉容器部署的方法，以及減少維運人員對容器平台的管理工作。

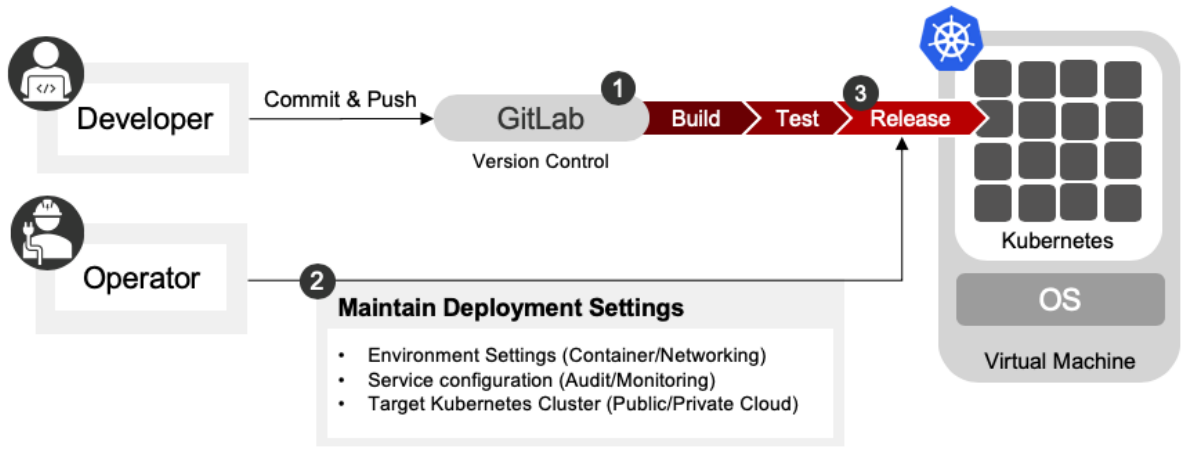

圖 17 容器部署流程自動化

若要完成自動化部署機制的建置，需要完成下列建置工作：

1. 搭建程式碼版本控制平台（Version Control System）
2. 準備容器部署設定範本（Deployment Configuration）
3. 準備部署自動化執行腳本（Deployment Scripts）

4.3 階段三：測試環境（Testing Environment）

生產環境已經容器化，但開發和測試環境可能還未轉置到容器平台之上，這將導致開發人員需要費心同時管理舊的開發、測試環境設定，亦需維護容器平台的相關設定。此時，最好的解決方案，便是搭建容器化的測試環境，實現從開發到生產環境皆為容器的統一平台。採用容器技術實現的測試環境，進一步解決下列測試資源不夠的問題：

1. 開發及測試資源爭奪
2. 同一時間只能部署一個版本
3. 工作任務可能相互衝突
4. 任務管理困難

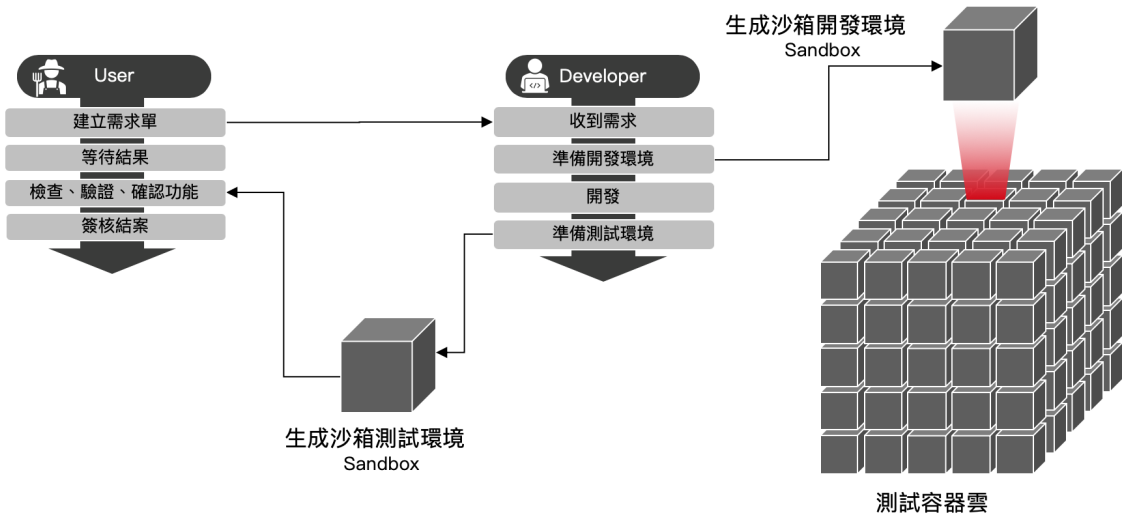

圖 18 採用容器技術實現的測試環境

藉由容器管理的優勢，每個開發工作都可以建立臨時的沙箱測試環境（如：SIT、UAT 等），待測試完成並推上正式環境（Production）後，自動釋放測試環境。在這樣的架構，可以使多個版本的測試和開發工作同時進行，並可以在不影響其他開發人員的情況下，重現系統當下環境，進行測試和偵錯。這是傳統集中式 SIT/UAT 環境，所遭遇的最大痛點。

4.4 階段四：持續交付及持續部署流程（CI/CD）

有了全容器化的系統平台，可以進一步整合開發、測試直到生產環境，實現一個流水線開發交付及部署的全自動化部署流程。從此開發人員只需要專注開發和推送成果至版本控制系統，其餘的測試環境準備（SIT/UAT）、正式環境上版（Production），都可以快速自動完成。

持續交付主要可以實現頻繁、可靠的軟體發佈，同時還能降低維運風險。它可以簡化和自動執行軟體交付流程，以便快速、可靠地推動應用發佈，同時還提供針對整個應用發佈流程的管理監控以及可視化，並可跨軟體發展生命週期的管理，在多種部署平臺進行調度。

通常持續交付及持續部署流程 CI / CD 會伴隨著企業使用的版本管理系統以及軟體發佈系統進行流程整合，以下為以 gitlab 為版控系統的 CI/CD 流程規劃範例：

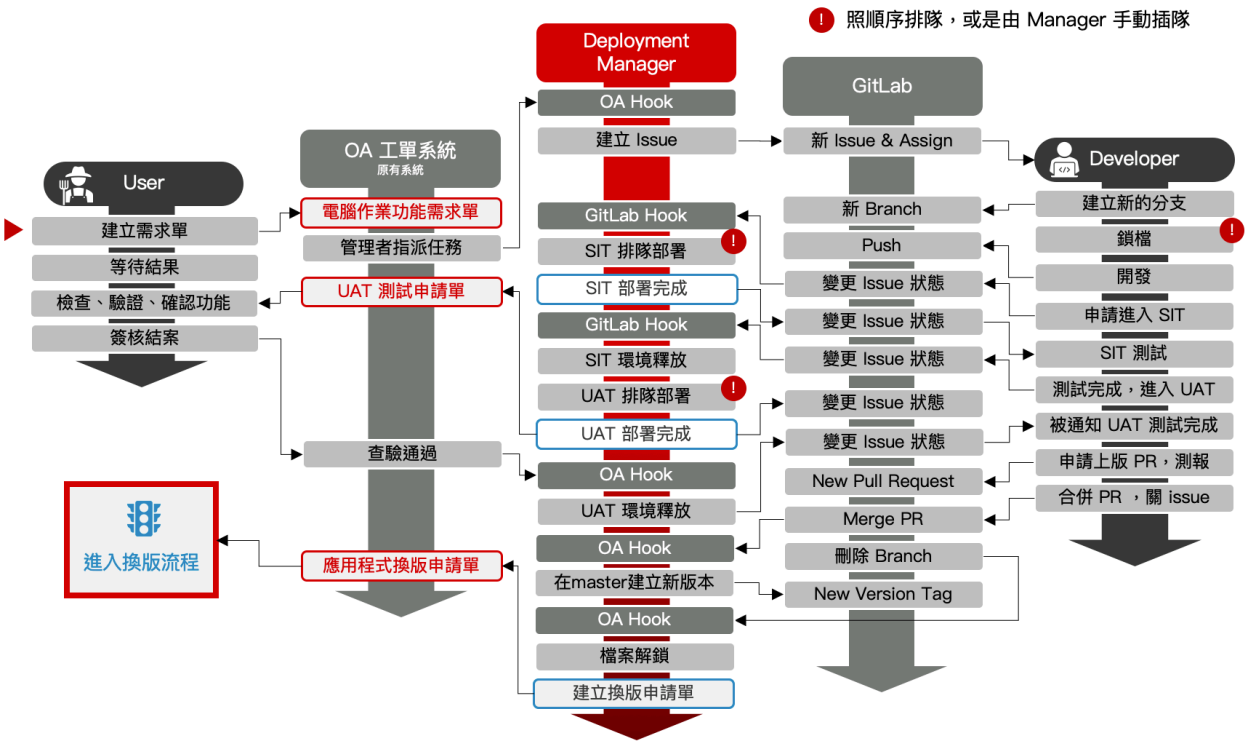

圖 19 開發及測試系統流程規劃

自動化軟體發佈與持續交付，關注於以下三種主要功能：

1. 發佈管道自動化 Pipeline Automation
2. 工件管理 Artifact Management
3. 發佈儀表板 Release Dashboard

4.4.1 發佈管道自動化 Pipeline Automation

一個自動化的可重複的並且可靠的應用發佈過程，可以設定工作流和管控策略來進行編譯、部署和測試，讓開發人員輕鬆、靈活地建立發佈流程，並且可以對發佈過程的各個階段進行建模，在每個階段內部可以定義任務來完成自動化。

除了在階段內的任務，定義策略，進而確定如何在階段之間推進工件。透過一組現有標準或自訂的標準來選擇自動推進工件到下一個階段。

發佈管道進行調度（Pipeline Orchestration），可利用現有的工具和流程來調度發佈過程。一般開發團隊可以使用自己的調度框架，或是採用常件的調度框架像是 jfrog artifactory。這種“整合並擴展”的方法可以讓你充分利用現有的投資，並且開發人員可以繼續使用他們最喜歡的工具。

4.4.2 工件管理 Artifact Management

開發團隊經常需要二進位工件管理系統，因此經常有各種不同的工件系統，如 Nexus, Yum 或 Artifactory。Artifactory 現在作為所有其他工件系統的代理在不同單位中使用，所有抓取最新和正確的工件的過程由 Artifactory 處理。

4.4.3 統一發佈儀表板 Unified Release Dashboard

統一的發佈儀表板可以提供一個跨各階段的發佈狀態的一致性視圖，它可以追蹤工件以確認他們被持續使用，同時還可以查看任務的執行細節，如下圖：

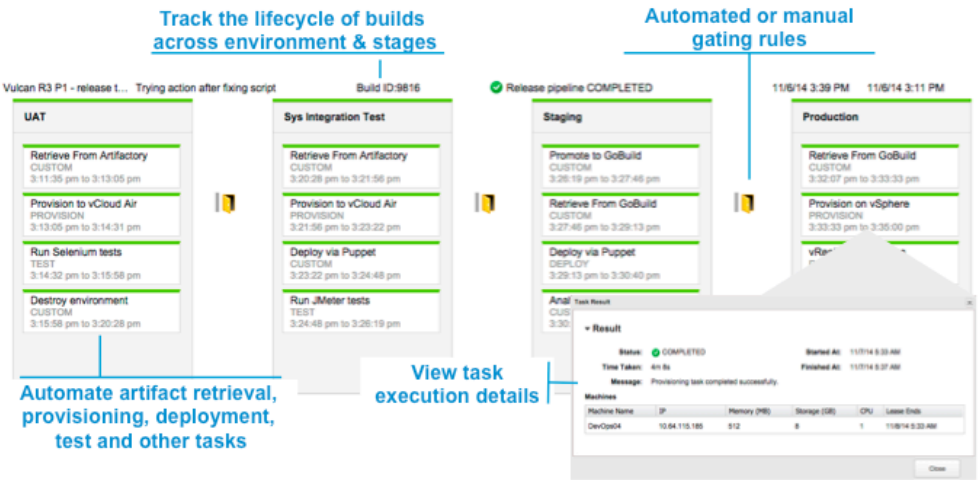

圖 20 統一發佈儀表板 Unified Release Dashboard

最終，我們可以將整個 CI/CD 的流程與輪廓勾勒出來，如下圖：

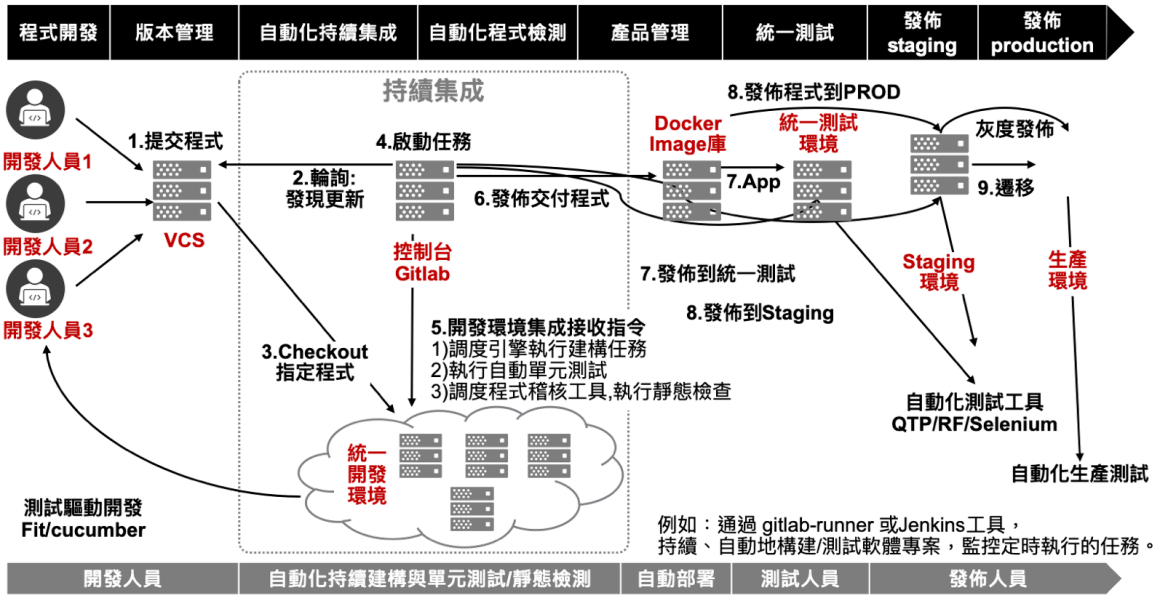

圖 21 CI/CD 流程圖

4.5 階段五：舊系統微服務化的軟體開發（Development）

到了這個階段，所有舊系統雖已容器化，但還不能稱之為微服務架構，應用程式的實例仍然還是單體（Monolith）架構，這代表整個系統還沒有真正從微服務架構的好處中獲益。需要開始進一步評估和執行軟體程式架構的轉置工作。

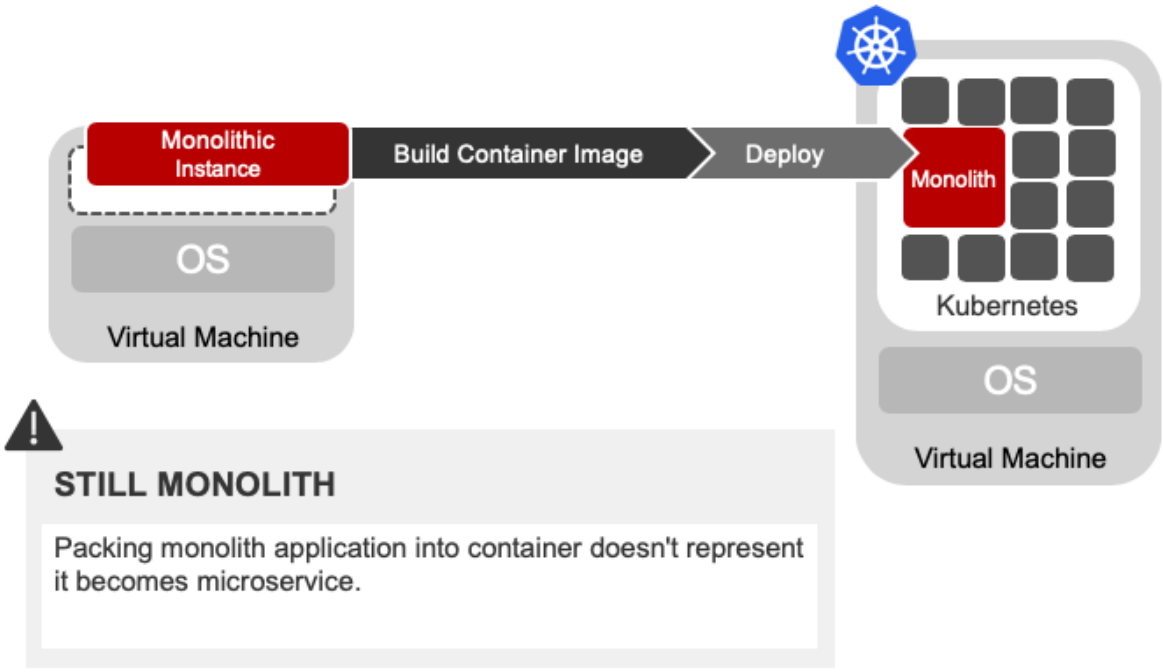

圖 22 單體架構容器化包裝後，仍然還是單體架構而非微服務

一切基礎設施都準備完成，參與人員的技能需求和新的工作規範制定已經能基本達標，下一步，即可開始準備將舊系統，逐步拆解、重新設計以轉換成真正具有各種效益的微服務架構。由於微服務的轉置開發工作，會完全基於容器平台之上，在開始微服務化的開發工作前，請先進行最後的內部評估確認：

1. 軟體開發人員已經可以自由於容器平台上，進行開發測試及上版工作。
2. 維運人員已經懂得透過容器管理平台，管理應用程式及服務狀態。
3. 既有的基礎設施，架構上可以容易擴充容器資源池（如：增加硬體）。

若準備就緒，即可開始進行微服務的架構開發，此時軟體開發人員可以遵循相關的設計方法，如：領域驅動設計（Domain-driven design, DDD），對舊系統進行改造、切割，然

後將工作獨立成一個個微服務實例部署在容器平台之上。而跟資料流有關的工作，則開始引入 CQRS 等概念，佈建佇列系統（Queuing System）如 Kafka 或 RabbitMQ 等，進行資料管線的規劃和設計。

4.6 典型案例

導入微服務架構雖有一些標準指導原則，但卻沒有最終標準樣貌，通常需要因應不同的應用場景及既有系統狀態，做出不同的設計和執行方法。不過，仍然有一些實務上的最佳實踐案例執得參考，並視情況引用。

4.6.1 資料採集與大數據應用

資料採集與大數據應用是一個相當常見的場景，由於收集、分析資料需求日益增加，傳統圍繞在「分析工具」的資料收集和分析過程，慢慢不敷使用。面對當下即時資料流的處理分析新趨勢，以及處理單元、處理流程的各種高頻率變更、升級和調校，制式工具和架構缺乏足夠彈性和擴充性來滿足。很多大型企業，甚至會碰到更多的資料來源和跨境、跨區、跨系統的需求，系統整合就是一個更為大的難題。

微服務架構可以提供彈性的方法，讓資料工程師、資料分析師以及應用開發者，可以更容易交換數據、設計資料處理流程。

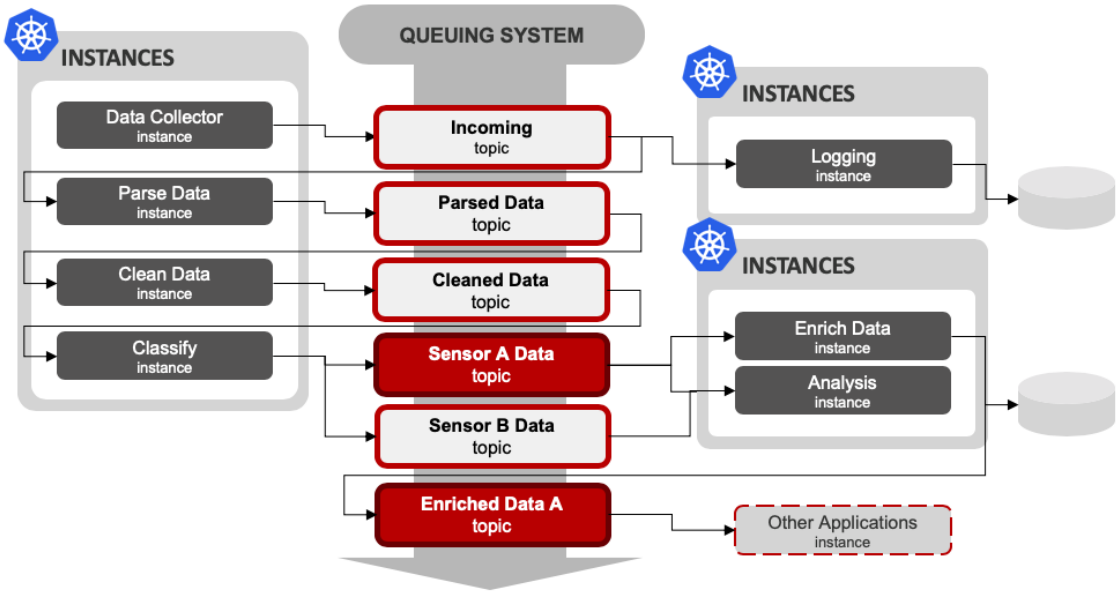

圖 23 資料管線和資料分流

4.6.2 批次資料供應管線

將原始資料從外部採集，經過一連串清洗、整理然後分門別類以資料集形式儲存在資料倉庫（Data Warehouse），應用程式或是分析程式可以選擇需要的資料集整批做後續處理，或是以 API 形式輸出使用。所有的處理流程，都是一個個微服務實例，可以動態調整或更新處理規則。

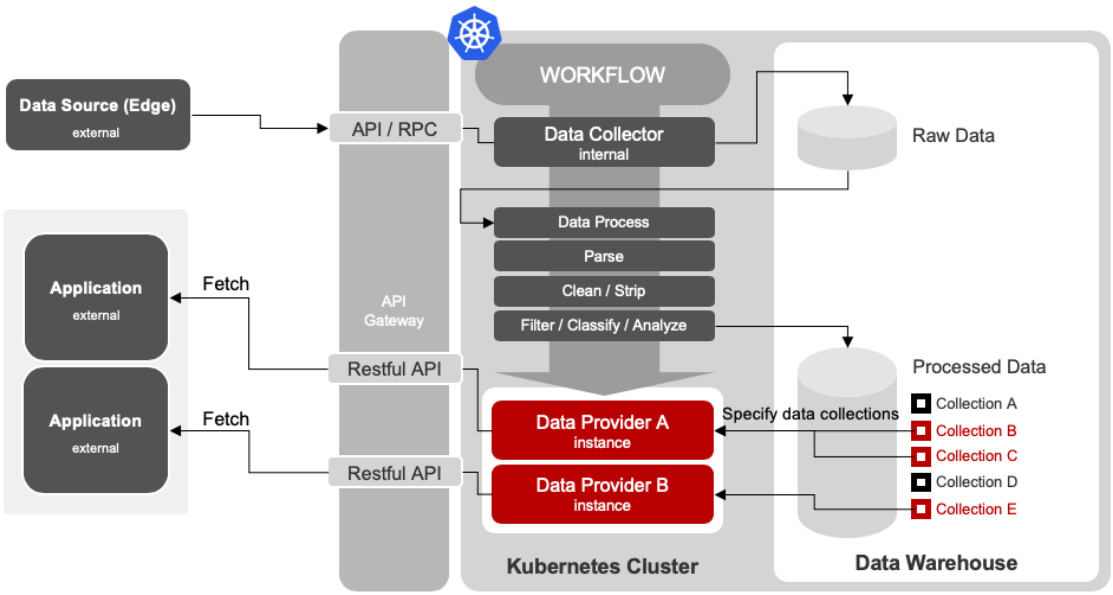

圖 24 批次資料處理的供應管線

4.6.3 即時資料流供應管線

即時資料流通常每次性處理的資料量不多，但會持續性的流入資料，每筆資料會被清理、整理、導流，通常會搭配佇列系統（Queuing System）做資料管線，讓資料依照主題（Topic）的形式被派送。有需要的應用程式或分析程式，可以藉由「訂閱主題（Subscribe）」，來取得流入的即時資料。

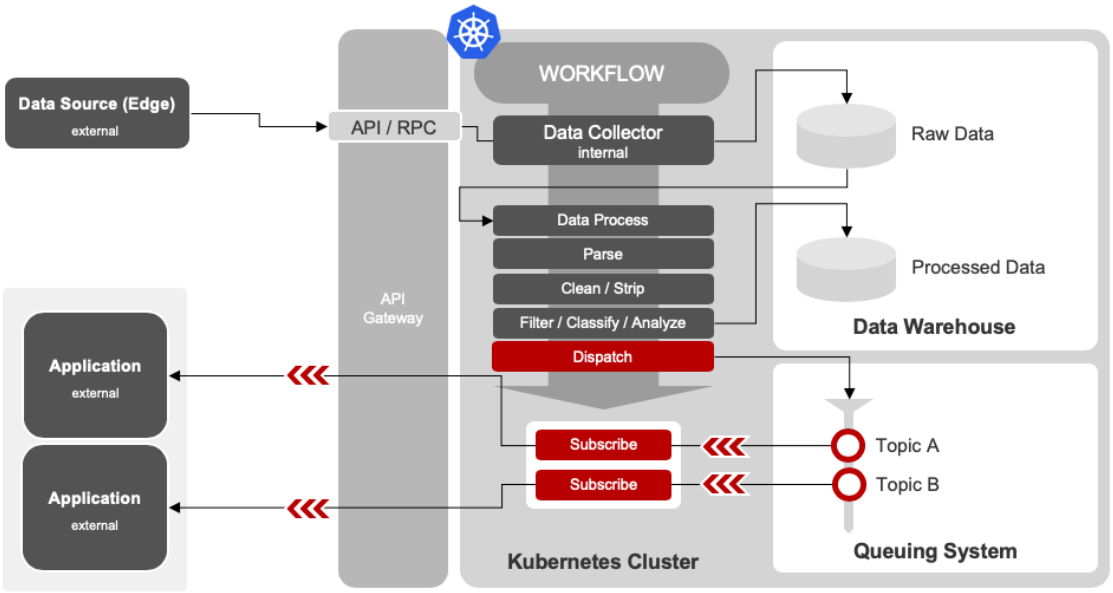

圖 25 即時資料流的供應管線