

# .NET Conf

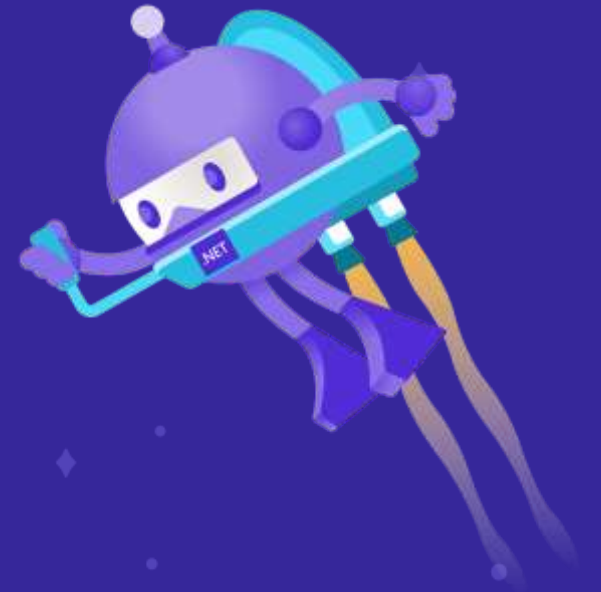
探索 .NET 新世界



# 非同步系統的服務水準保證

// 談非同步系統的 SLO 設計

Andrew Wu, 91APP / Chief Architect  
2020 / 12 / 19



# 講師簡介: Andrew Wu

現職: 91APP Chief Architect

負責帶領: 產品研發聯合處 / 架構設計部, 執行架構改善與規劃 ;  
核心服務設計開發 ; 以及大型整合專案的推動。



現任: Microsoft MVP ( Azure, 2016 ~ 至今 )



專長與個人期許:

談論各種軟體開發與設計的大小事，有 20 年的大型與雲端服務的開發經驗。  
喜歡研究各種技術背後的原理與實作細節，期許自己做個優秀的系統架構師。

主題以: .NET / C# / OOP / Container / Cloud Native / DevOps 為主軸，同時在  
部落格上也持續分享相關主題的一系列文章。期許能將這些實作經驗分享到社群。

回顧-2017

容器驅動開發

Container Driven Development.

Andrew Wu

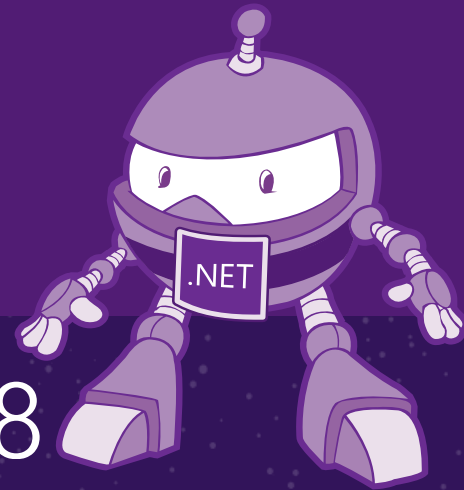


Learn. Imagine. Build.

.NET Conf

# Message Queue Based RPC

Andrew Wu, Chief Architect @ 91APP

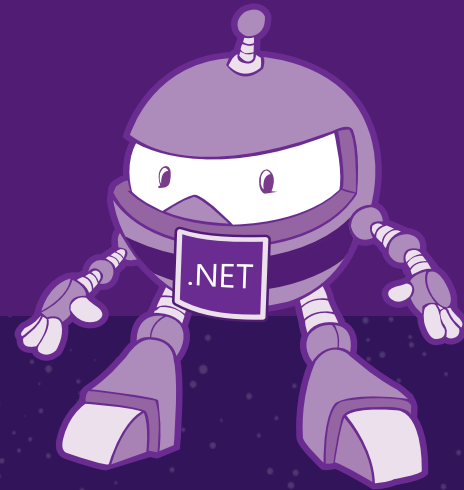


.NET Conf 2018



# 大規模微服務導入 #1

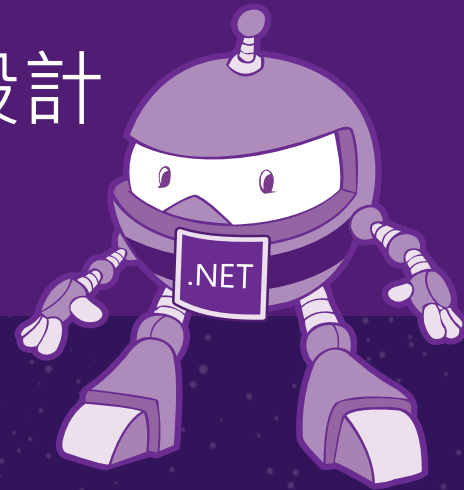
從零開始的系統架構設計概觀  
Andrew Wu, 2019/11/09





# 大規模微服務導入 #2

從零開始的微服務 .NET Core 框架設計  
Andrew Wu, 2019/11/09







# .NET Conf 2020

**91APP** 企業實戰系列





.NET Conf  
2020

Microsoft STUDY4 Build School



Ruddy Lee

## DevOps 教戰手冊：三步工作法

DevOps ( Development 和 Operations 的組合) , 是一種重視軟體開發人員和 IT 運維技術人員之間溝通合作的文化、運動或慣例。

在 DevOps 經過 10 年的市場洗禮之後，讓我們重新回來審視當年盡力在推廣開發作業 Dev 與維運作業 Ops 相結合的軟體界先驅們的思維過程「三步工作法」，這個議程希望你在聽完後能夠有「Aha, 原來如此！」的頓悟。



# .NET Conf 2020

## 非同步系統的服務水準保證；淺談非同步系統的 SLO 設計



Andrew Wu

91APP 的系統，在支撐所有客戶每年累計上百億的訂單背後，有一套負責管理與維運全公司非同步任務的系統，叫做 NMQ (NineYi Message Queue)。它乘載了我們每日超過百萬個 task，共有數百個不同類型的任務(job)處理。

在這麼龐大的 task 背後，我們該如何兼顧每種任務的可靠度與服務水準？舉例來說，光是發送簡訊這件事，我們就面對「行銷簡訊」的發送與「註冊簡訊」的發送。「註冊簡訊」期望在五秒內就送達，而「行銷簡訊」則有比較寬裕的發送延遲時間。如何兼顧系統複雜度、維運成本與服務水準，就是我們要面對的課題。

Ruddy 老師在闡述 DevOps 的精神時，都會耳提面命的告訴我們：「要先能夠量測，才能夠改善。」我們架構團隊在面對這挑戰的第一件事，就是先定義如何將服務水準量化，同時替這指標訂好目標，也就是所謂的 SLO (Service Level Objective)。這個 Session 我會來介紹我們架構團隊如何用系統化的方式來面對這個難題。





.NET Conf  
2020

Microsoft STUDY4 Build School



Steven Tsai

## 微型任務編排器 - 以 Process Pool 為例

你有服務上不了 k8s 的煩惱嗎？你手上是否也有那種不知何時會來的任務排程，有的一兩週才執行一次，有的又像洪水猛獸一般打的你措手不及失了魂，它們不定時不定量，讓你在配置資源上吃盡苦頭，也許，微型任務編排器正是你在找的東西！



# .NET Conf 2020

 Microsoft  STUDY4  Build School

## 刻意練習：如何鍛鍊你的抽象化能力

我們來聊聊開發大型軟體最重要的「抽象化」能力吧！

「抽象化」——考驗的是你如何從眼前的需求看出背後的脈絡，並且把這些脈絡用具體的模型 (也就是 class / object) 表達出來的過程。面對越複雜的系統，這樣的能力越發重要。在 91APP 的架構團隊，首要任務就是替所有研發團隊找出這些需求背後的脈絡，並規劃出合適的架構後，再由各個團隊合作開發完成。也因此用 code 來講故事的能力，在我們團隊內是不可或缺的。

這個 session 主要探討的是大型軟體開發團隊的基本功夫該如何培養。前段會由 Andrew 來說明我們如何找出複雜商業需求背後的脈絡，並且抽象化實作的過程。從這些過程，你可以了解為何這基本功夫這麼重要；後段則請我們的 team member Fion 現身說法，用我們內部的練習題來當作案例，示範如何透過不斷的練習來達成目的。



Fion Yu



# 前言

服務水準的概觀: SLA, SLO, SLI ?



## 維運管理重點

- 系統監控
  - 「能被量測的系統，才能被控制」
- 預防型維運管理
  - 設定目標
  - 量測指標
  - 提前改善



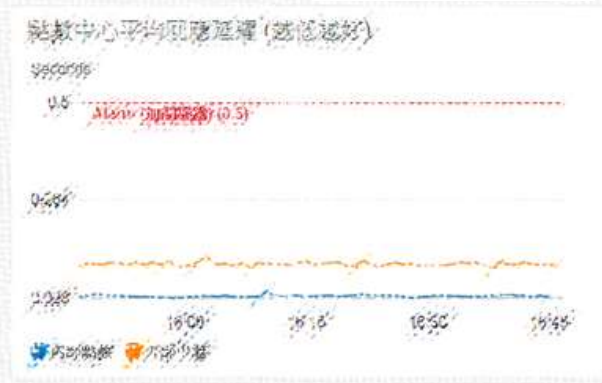
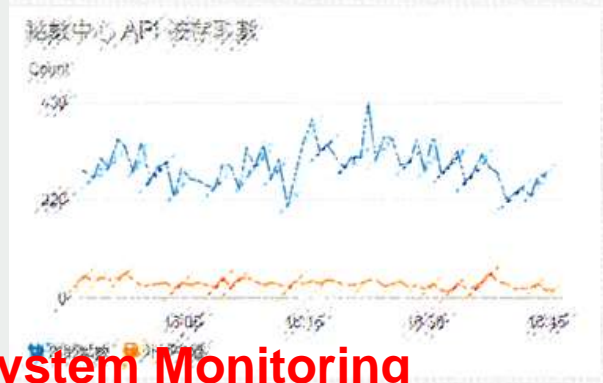
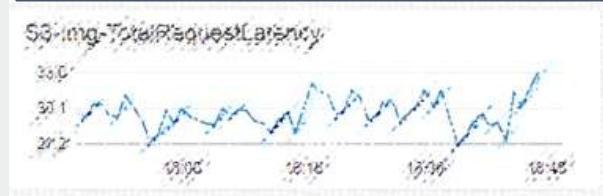
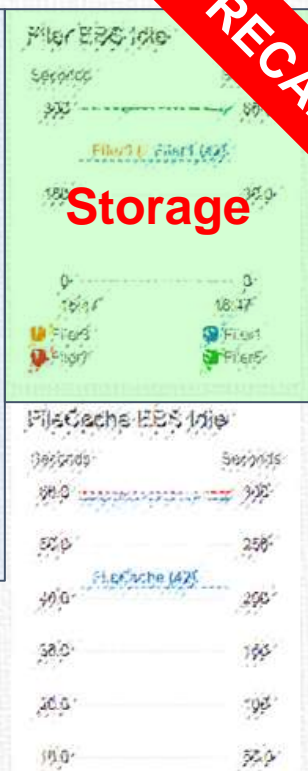
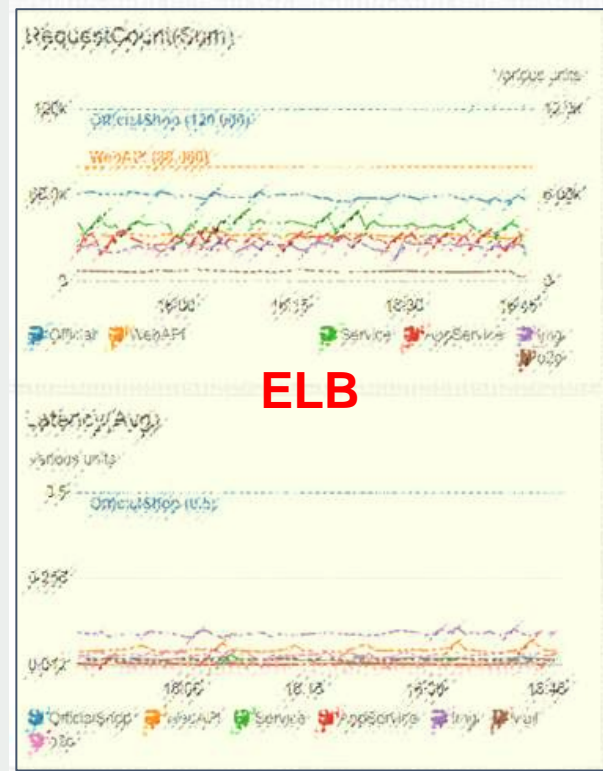
RECAP

Storage

Cache

Web  
Server

ELB



System Monitoring

Task Count

13.3k

13.3k

100%

SendTemplateMailPriorityLo

SendTemplateMailPriorityLo

SendTemplateMailPriorityLo

9.63k

9.63k

100%

UpdateTransactionInfoBalancePoint

UpdateTransactionInfoBalancePoint

UpdateTransactionInfoBalancePoint

1.57k

1.57k

100%

CreateLoyaltyPointTransactionInfo

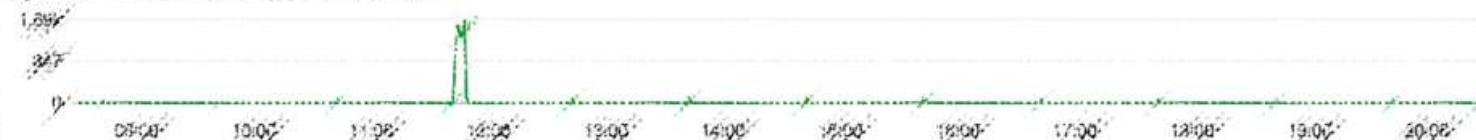
CreateLoyaltyPointTransactionInfo

CreateLoyaltyPointTransactionInfo

SendTemplateMailPriorityLow



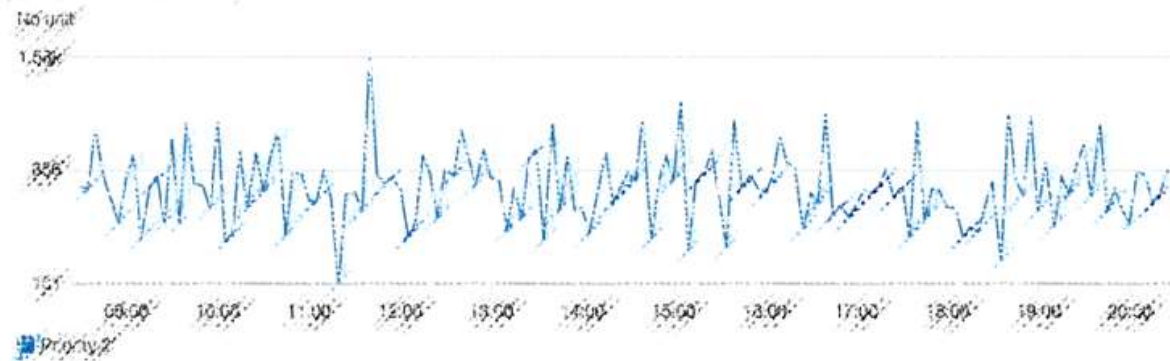
UpdateTransactionInfoBalancePoint



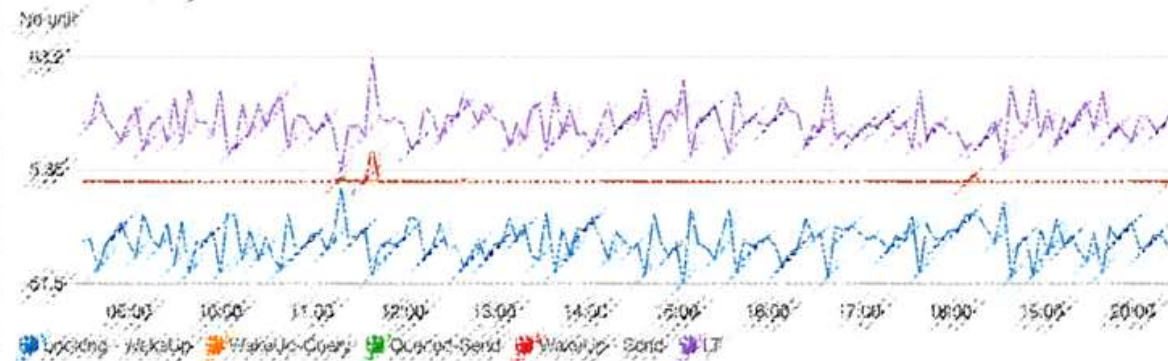
CreateLoyaltyPointTransactionInfo



Switch - LT (P\*P)




Switch - Priority-2







# 預防型維運管理

- 決定服務等級目標 - Service-Level Objective (SLO)
    - 99% 前台每秒User訪問延遲 < 300ms
  - 測量服務當前狀態 - Service-Level Indicator (SLI)
    - 目前狀況：99% 前台每秒User訪問延遲 < 75ms
  - 決定服務等級領先目標
    - 綠燈：99% 前台每秒User訪問延遲 < 150ms
    - 黃燈：99% 前台每秒User訪問延遲介於150ms 到 200ms
    - 紅燈：99% 前台每秒User訪問延遲 > 200ms
  - 定期每月、每季Review領先目標
    - 針對黃紅燈項目列出Action Item
- 

# SLA



## SERVICE LEVEL AGREEMENT

the agreement you make  
with your clients or users

# SLOs



## SERVICE LEVEL OBJECTIVES

the objectives your team must  
hit to meet that agreement

# SLIs



## SERVICE LEVEL INDICATORS

the real numbers on  
your performance

# Case Study





# 狀況：帳號註冊的驗證簡訊發送

## 情境：

會員在註冊帳號的過程中，需要驗證手機號碼。91APP 系統會發出驗證簡訊，使用者收到後輸入驗證碼，即可完成手機號碼驗證。

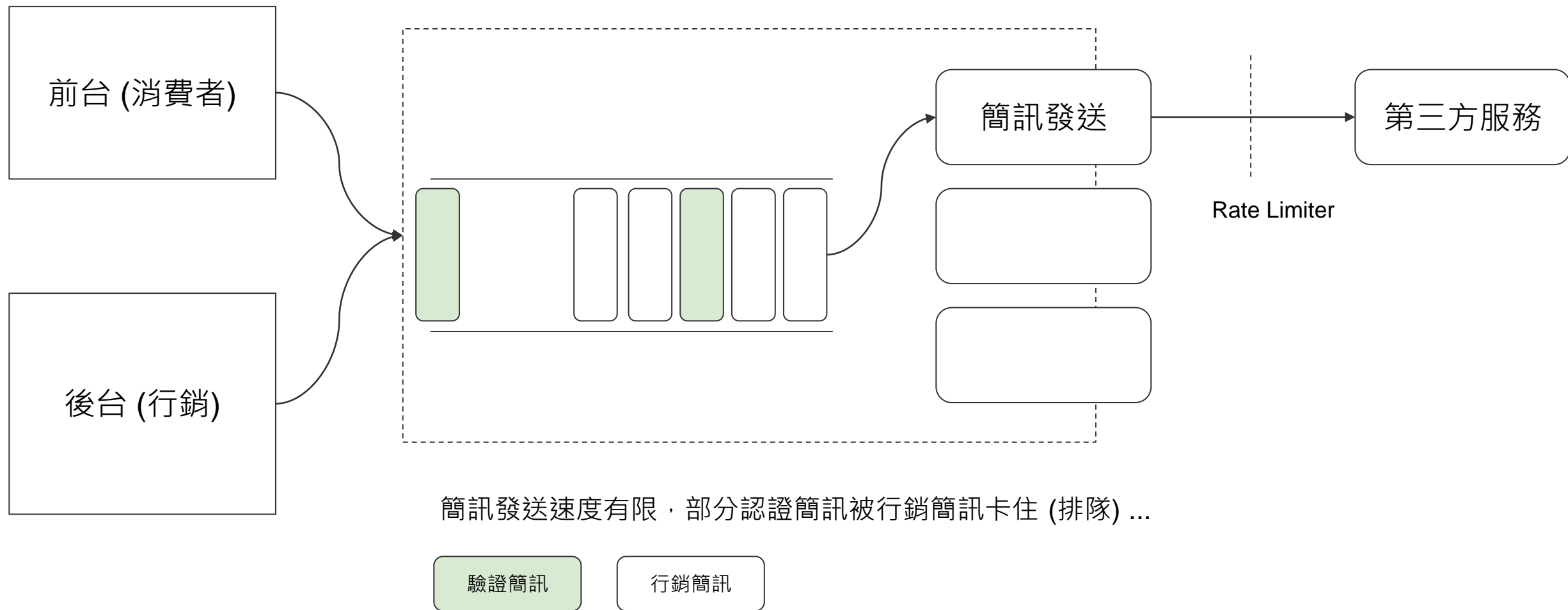
## 要求：

為了顧及使用者的體驗，系統必須在 **5 sec** 內完成發送的作業。驗證碼必須在 **5 min** 內輸入才有效。

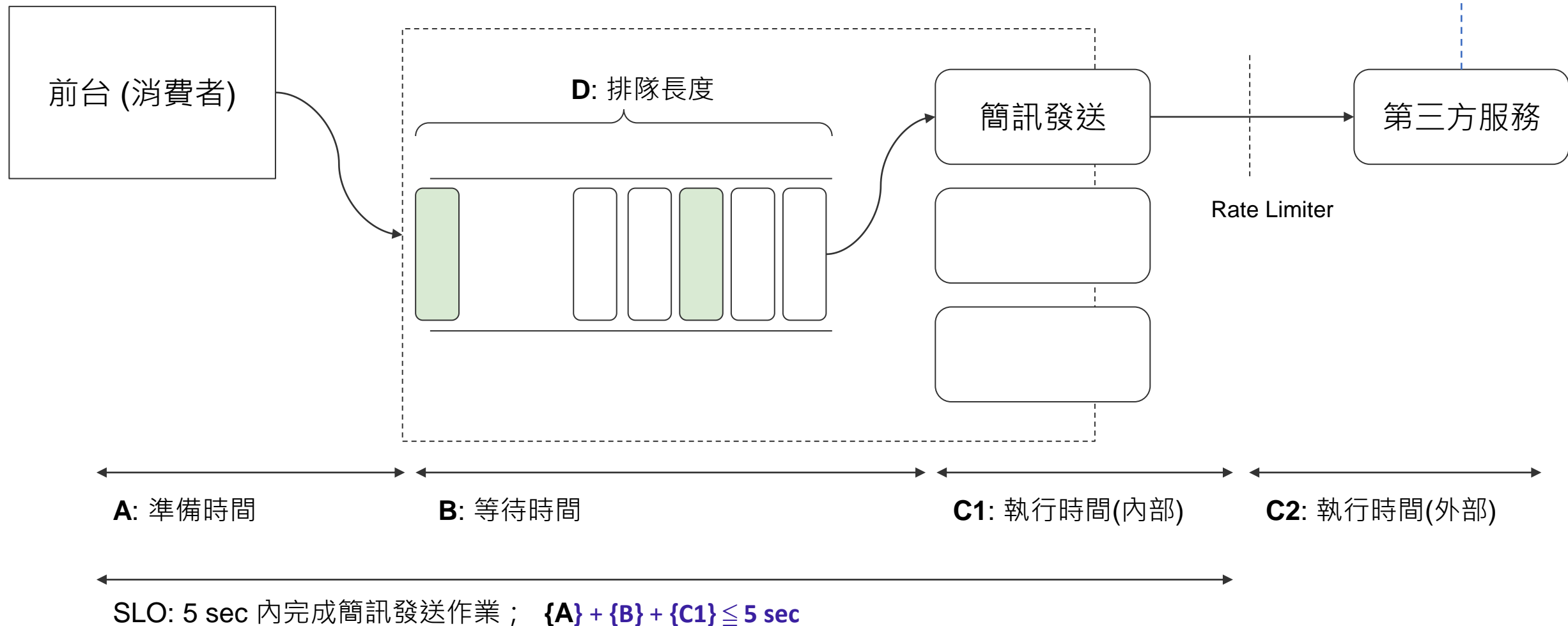
## 挑戰：

1. 對時間敏感的任務，必須盡量避免被干擾
2. 要顧及外部系統的處理能力

# Case: 搶購前會有大量的會員註冊 & 行銷簡訊發送...



# 我們監控了哪些指標? (SLI)



## 診斷：有監控數據 (診) 才能找出效能瓶頸的所在 (斷)

如果：

( A ) 的數值過高：前端系統產生驗證簡訊的速度太慢；

( B ) 的數值過高：訊息在 Queue 裡面排隊花太多時間

**Queue 堆積太多行銷簡訊**

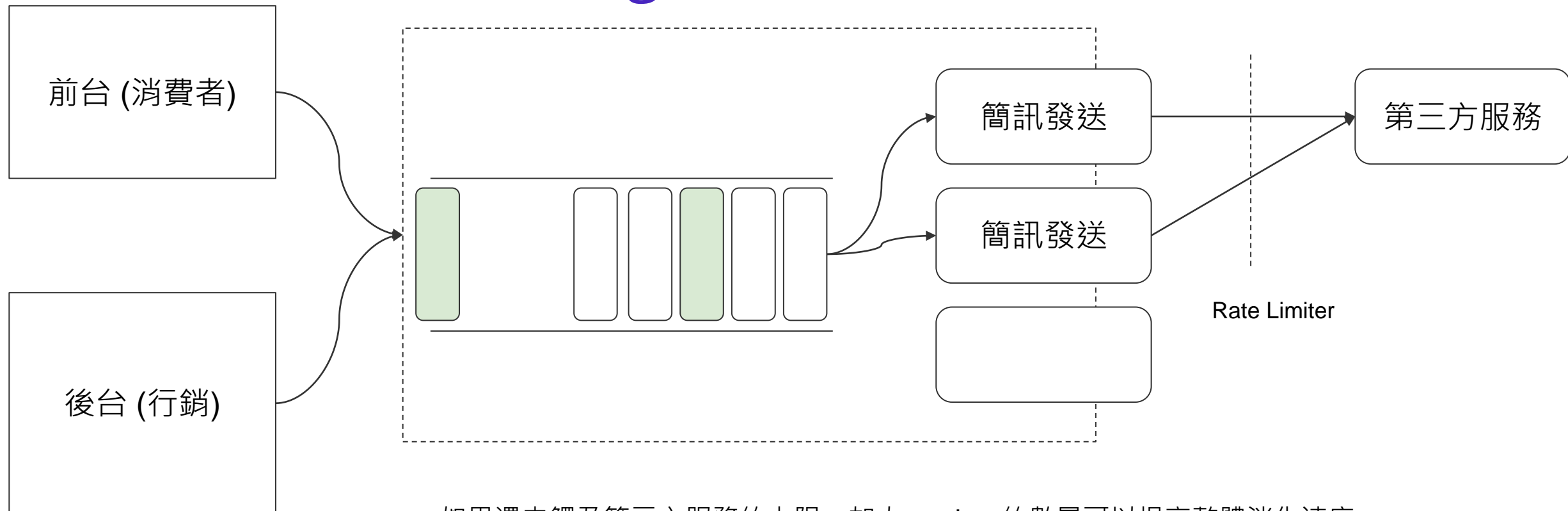
( D ) 過高, 訊息堆積太多

( D ) 不高, 訊息消化太慢

( C1 ) 數值過高：訊息消化太慢 (查詢資料庫，套用訊息內容等等)

( C2 ) 數值過高：第三方的處理效能太慢

## Solution #1, Message Worker Scaleout ...

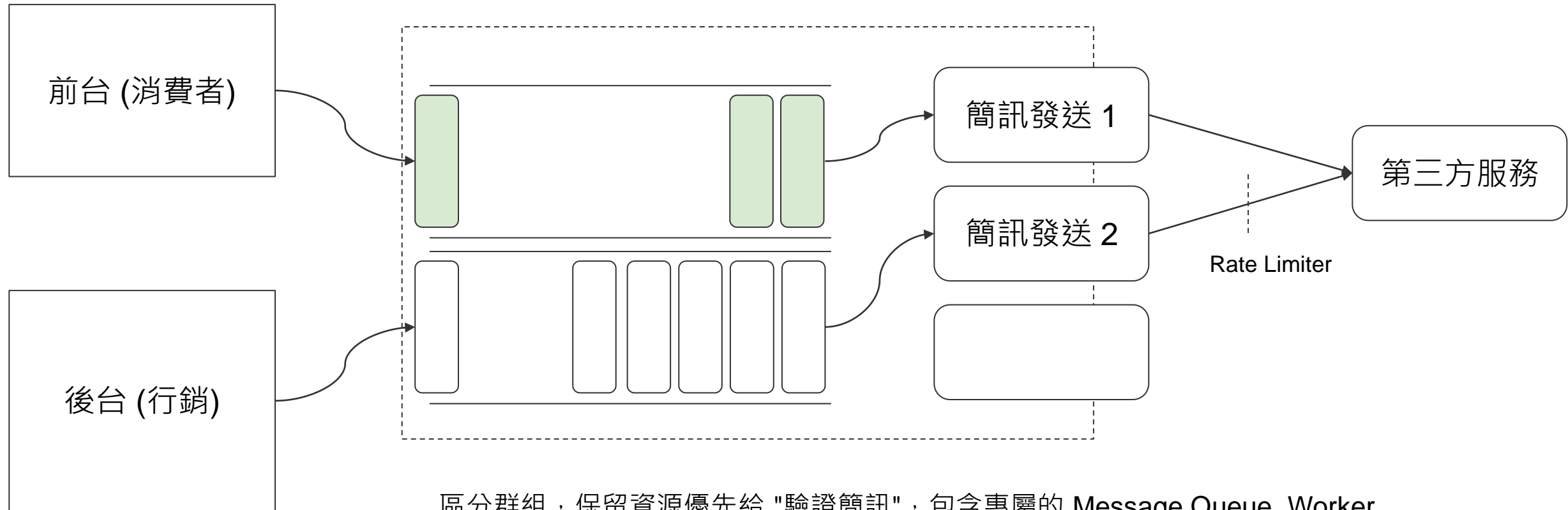


如果還未觸及第三方服務的上限，加大 **worker** 的數量可以提高整體消化速度。

代價: 耗用兩倍的運算能力

=> **錢沒花在刀口上**，因為不需要被加速的行銷簡訊也被加速了...

## Solution #2, 降低 Queue Length 的方法 => 分群組



區分群組，保留資源優先給 "驗證簡訊"，包含專屬的 Message Queue, Worker, Rate Limiter ...

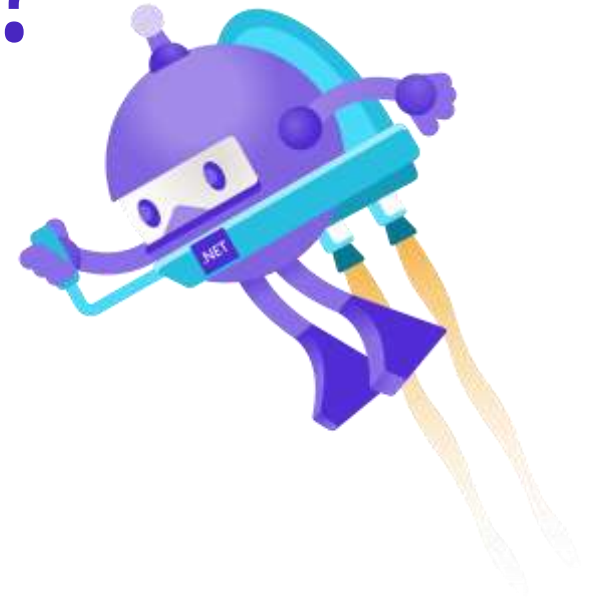
資源花在刀口上，完全用於加速驗證簡訊的發送。



# Q: 我當下怎知該怎麼做?

想辦法擁有“上帝視角”。

把你需要的指標，放到監控系統內



目標導向: 從開發的第一天，就弄清楚你期待的 SLO ...

定義: 消費者按下 "發送驗證簡訊"，**5 秒內** 就要送到手機上

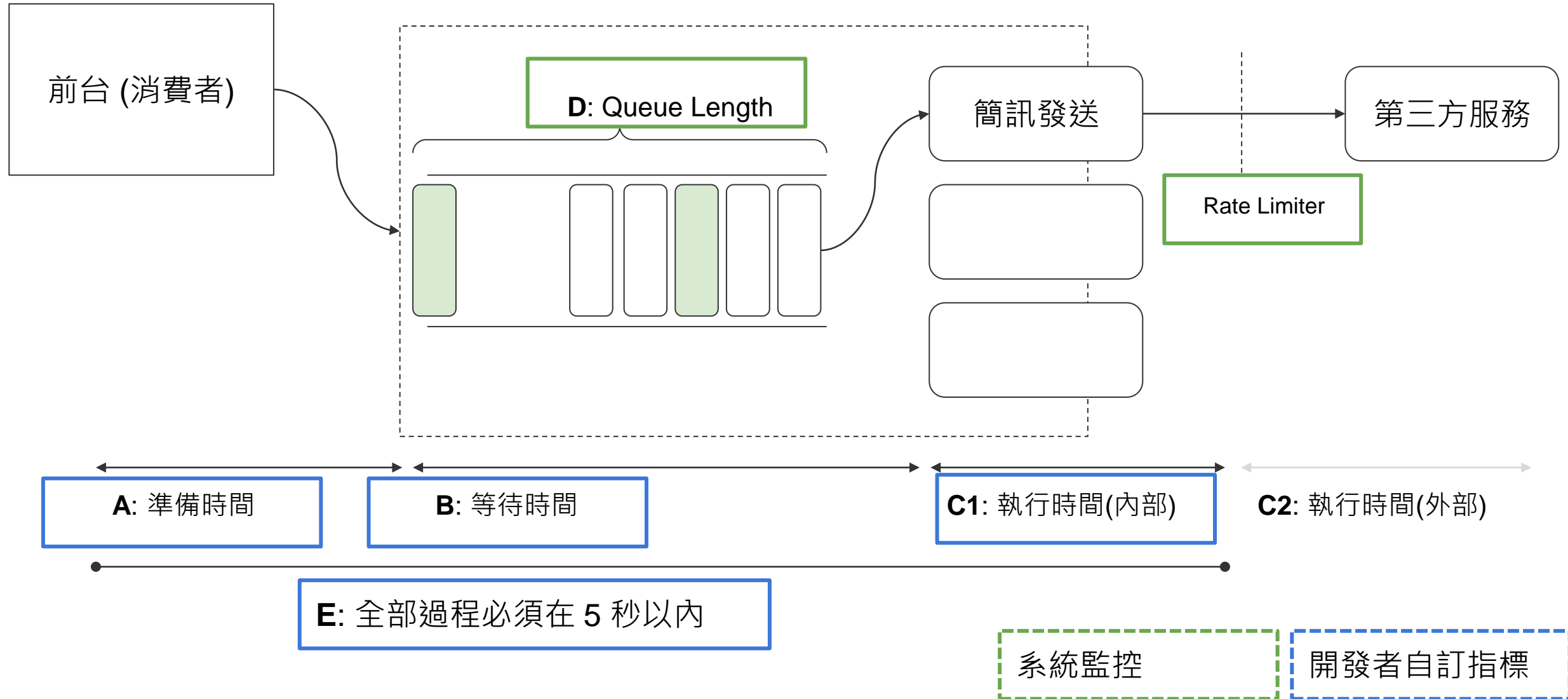
拆解: 這 5 秒內總共要完成哪些事情?

盤點: 我能掌握哪些指標?

改善: 如何處理我無法掌握的指標? (為了維運而設計)

行動: 善用監控的服務，透過 logs 分析，或是 metrics API 來達成

# SLI: 我們監控了什麼?

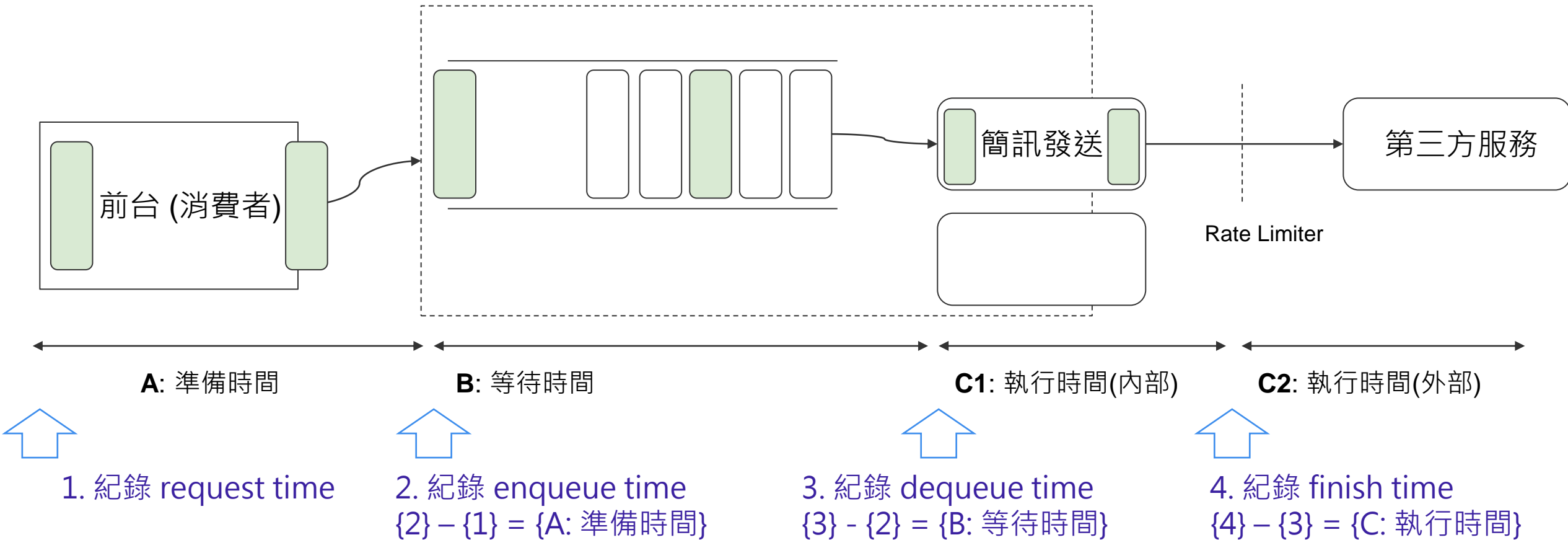


# Next:

統一收集與管理運用這些指標



# 怎麼標示與取得這些指標? ( for developer )



上述紀錄，經過運算後都可以透過 SDK 送出監控數據。

# Code Sample (Azure Application Insight API)

ApplicationInsights.TelemetryClient.TrackMetric 不是傳送計量的慣用方法。您應該一律預先彙總一段時間的計量，再加以傳送。使用其中一個 GetMetric(..) 多載來取得可供存取 SDK 預先彙總功能的計量物件。如果您要執行自己的預先匯總邏輯，您可以使用 TrackMetric ( # A1 方法來傳送產生的匯總。如果您的應用程式需要每次傳送個別遙測項目 (未隨時間彙總)，則可能有事件遙測的使用案例；請參閱 TelemetryClient.TrackEvent (Microsoft.ApplicationInsights.DataContracts.EventTelemetry)。

Application Insights 可以將未附加至特定事件的計量繪製成圖表。例如，您可以定期監視佇列長度。當您使用計量時，個別測量的重要性就不如變化和趨勢，因此統計圖表很有用。

為了將計量傳送至 Application Insights，您可以使用 TrackMetric(..) API。您有兩種方式可以傳送計量：

- 單一值：每次在應用程式中執行一個測量，都會將對應值傳送至 Application Insights。例如，假設您有一個描述容器中項目數的計量。在特定期間內，您先將 3 個項目放入容器中，再移除 2 個項目。因此，您會呼叫 TrackMetric 兩次：先傳遞值 3，再傳遞值 -2。Application Insights 會代替您儲存這兩個值。

- 彙總：使用計量時，每個單一測量並不重要。相反地，在特定期間內發生的狀況摘要才重要。這類摘要稱為\_彙總\_。在上述範例中，該期間的彙總計量總和為 1，而計量值的計數為 2。使用彙總方法時，您只會在每段期間叫用 TrackMetric 一次，並傳送彙總值。這是建議的方法，因為它可以藉由傳送較少資料點至 Application Insights，同時仍收集所有相關資訊，來大幅降低成本和效能負擔。

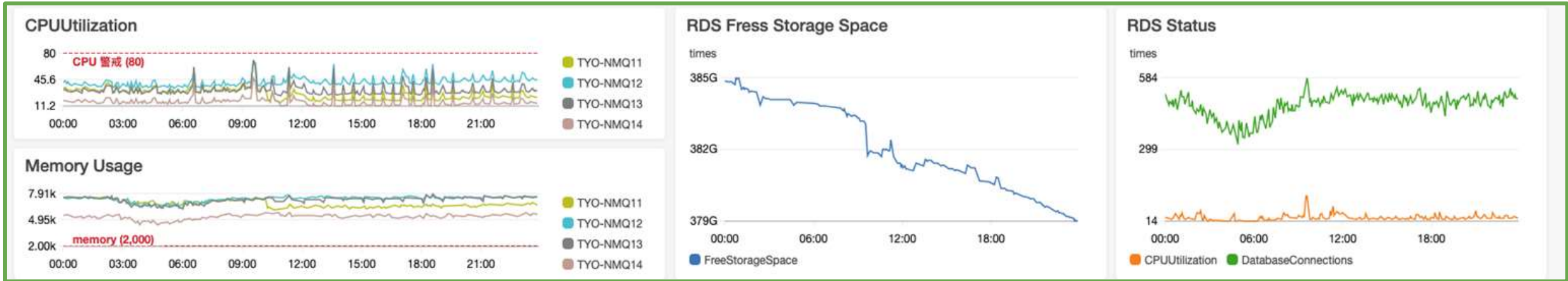
C#

```
var sample = new MetricTelemetry();
sample.Name = "metric name";
sample.Value = 42.3;
telemetryClient.TrackMetric(sample);
```

為了平衡效能與費用，經過 Buffer (in-memory) 緩衝是必要的，即使只有緩衝 1 sec 都能發揮極大的效益。如果不設置 buffer, 那麼費用與 RPS (request per second) 成正比 (變動費用)。若設置 1 sec 的 buffer, 則費用變成與 worker 數量成正比 (固定費用)。



# 2020 雙十一 監控 dashboard: 我們監控了什麼?



# 指標: Group Queue Process Status



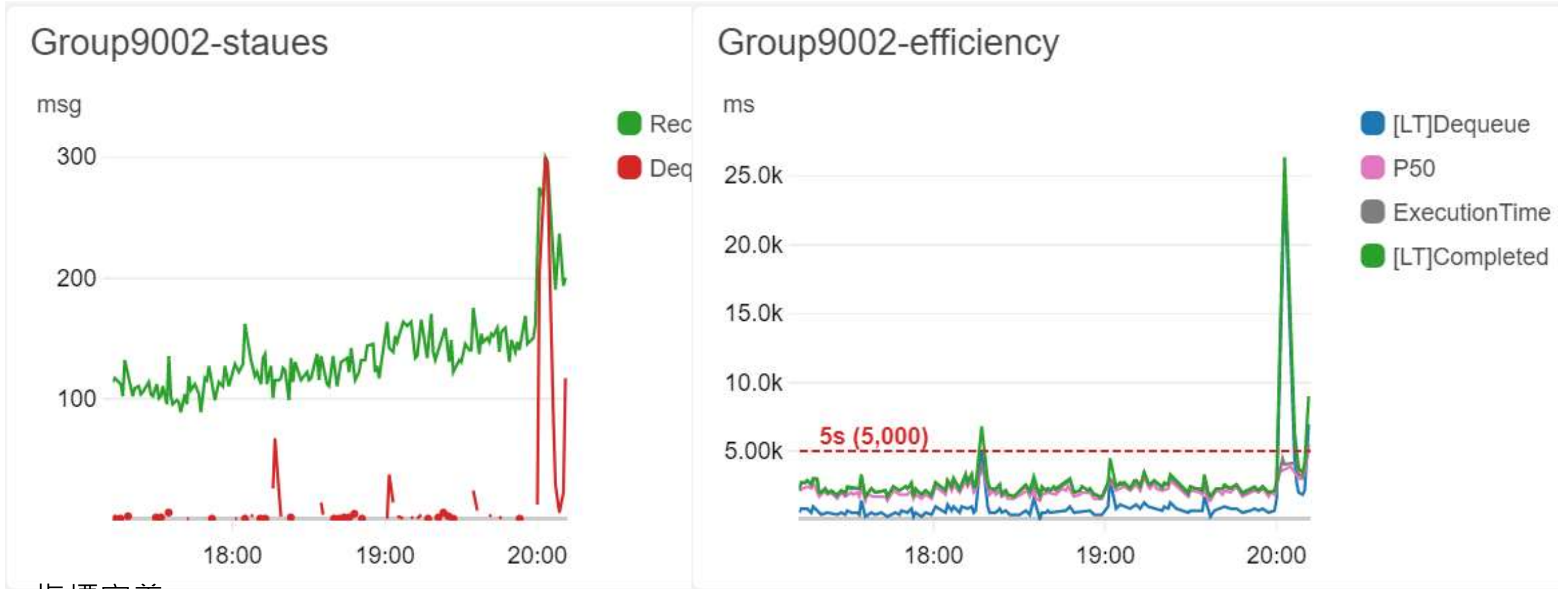
指標定義:

- Received:
- Dequeue-Over-SLO:

Task 從 Message Queue 取出執行的數量統計

所有從 Queue 取出的 Task 中, 取出當下就已經超過 SLO 要求的數量  
( **A** + **B** > **5** sec, 持續 **3** 分鐘狀況沒解除就會發送警告通知 )

# 指標: Group Queue Process Efficiency



指標定義:

- Efficiency: 每個任務從 create task 開始，到 task complete 為止的時間 (C)

# Q: 有了上帝視角之後?

訂定面對各種狀況的應對方式



# 案例1, 突然有大量的簡訊發送任務, 都超出 SLO 的要求...



指標定義:

- Received:
- Dequeue-Over-SLO:

Task 從 Message Queue 取出執行的數量統計

所有從 Queue 取出的 Task 中, 取出當下就已經超過 SLO 要求的數量  
( **A** + **B** > **5** sec )

# 碰到這種狀況，你會...?

先列出所有你想的到，[可能] 的解決方式有:

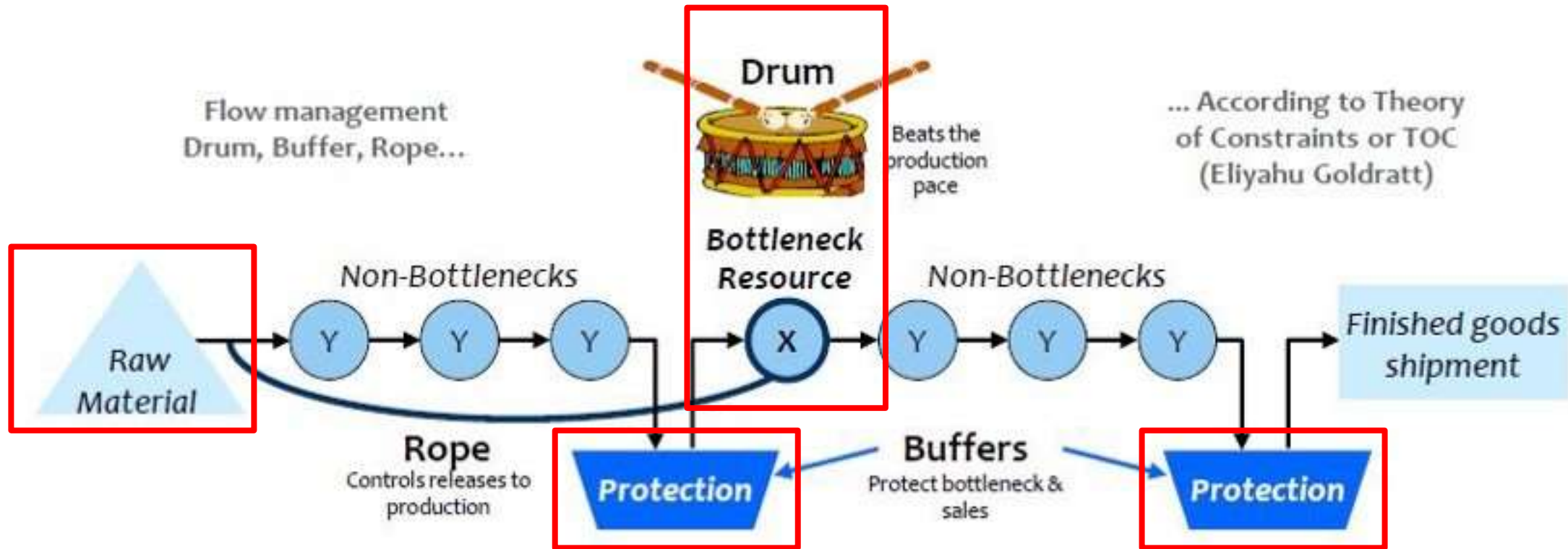
1. 加開 Worker, 增加 instance 個數
2. 改善 Queue 的效率
3. 改善 Task 的效率
4. 改善來源端 ( Task Create ) 的效率
5. 降低來源端 ( Task Create ) 的速度 ...

**是否有較有系統的思考方式，能精準的找出應對方式?**



# 從限制理論來看 (TOC, Theory Of Constraints)

最佳控制點



控制原料，避免生產過快造成堆積。  
可由 X 的狀況來控制原料的進貨狀況。

如果偵測到可能超過設計負荷，應該拉動繩子來告訴前端停止原料的採購，避免更大規模的損耗。

確保 X 能夠不中斷的順利進行  
必須隨時確保 Buffers 沒被清空。  
不過過多的庫存也是浪費。

確保 X 能夠不中斷的順利進行  
必須隨時確保 Buffers 不會滿載。

[『高效率團隊』如何運用限制理論 \(Theory of Constraints\) 於軟體開發 \[EMBA 雜誌\] 我該如何解決問題的瓶頸](#)

# 解決瓶頸的步驟:

1. **找到你的瓶頸** ; (特徵: 瓶頸的前一關通常都會出現庫存堆積的狀態)
2. **充分利用瓶頸** ; (既然是瓶頸, 就別讓他停下來, 盡可能讓他維持100% 的產能)
3. **非瓶頸協助瓶頸** ; (犧牲自己, 將處理能力轉給瓶頸)
4. **提升打破瓶頸** ;
5. 回到步驟一 ( 找出下一個瓶頸 ) , 周而復始。

延伸思考: 生產線的 [瓶頸], 就是整體效能的控制點。

1. **改善** 瓶頸效能, 就能改善整體效能
2. **控制** 好 [瓶頸] 的速度, 就能控制整條產線的速度
3. 若無法改善 [瓶頸], 則要有能力在生產線失控 (過多 WIP) 前從源頭停止。

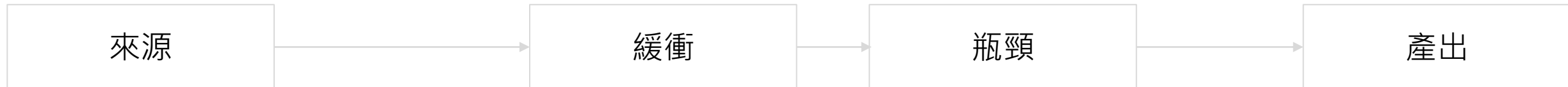
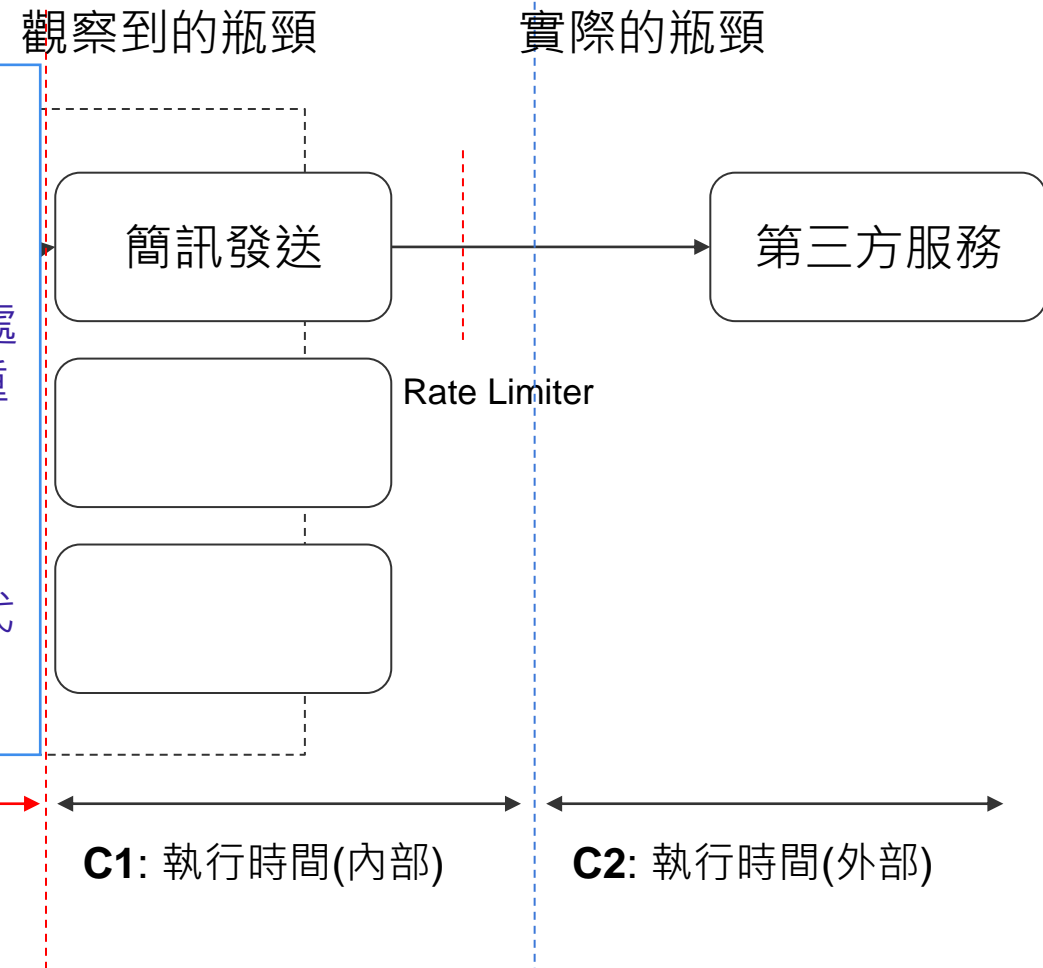


# 先從數據指標，還原實際的狀況

{ A } + { B } 延遲攀升, 加上 { D } 顯示堆積的狀況也提高，  
代表下一關 ( C1 ) 所在之處就是系統的瓶頸。

**對策1:** 為了充分利用“瓶頸”，別讓瓶頸閒置，同時讓瓶頸只處理關鍵的任務。從源頭剔除非優先的 SMS, 將關鍵的資源用在最重要的訊息上。

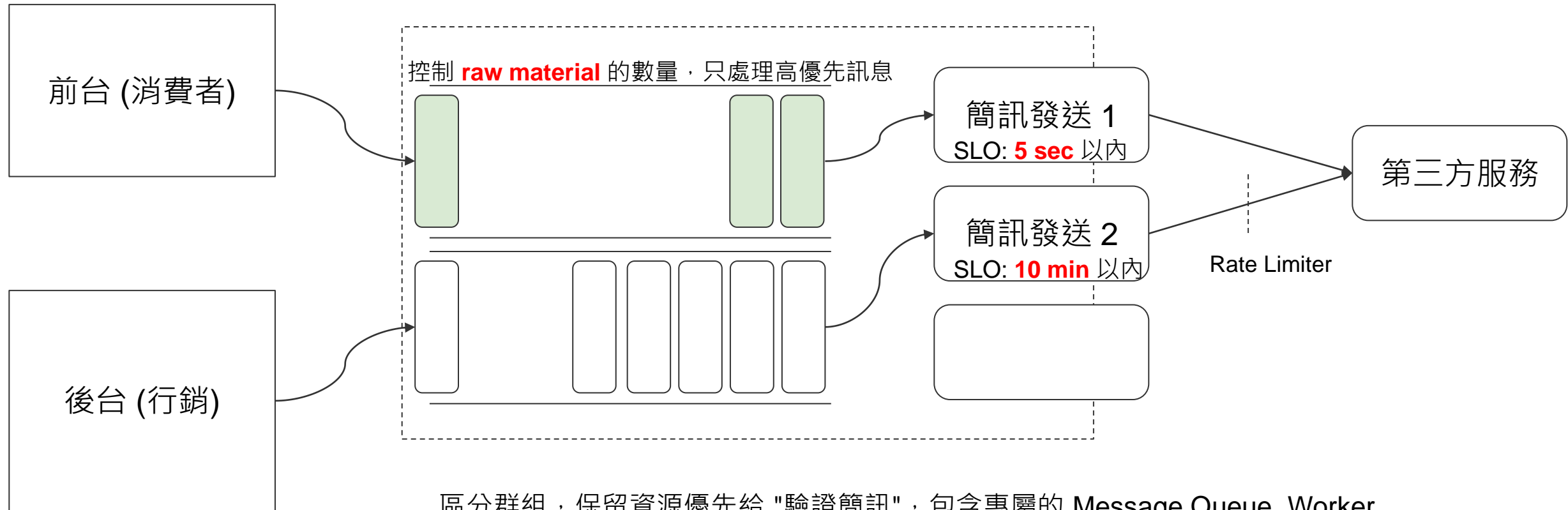
**對策2:** 若訊息堆積的情況過度嚴重，已經可以預期接下來每個 SMS 都必定無法滿足 SLO 時，就應該通知最前端，必須準備替代措施了。



# 能夠選擇的做法

1. 控制來源: 前端切換其他方式，進行手機號碼驗證 (例如: 改用撥號驗證)
2. 控制來源: 排除非關鍵的任務進入這條生產線 (例如: 行銷簡訊)
3. 降低 SLO 的要求 (非關鍵通知不需要 5 sec 送達)
4. 擴大 91APP 與簡訊商的安全容量  
(與第三方廠商確認後，放寬 Rate Limit 限制)
5. 擴充 Worker 的處理能力
6. 改寫 Task, 做好最佳化改善執行速度

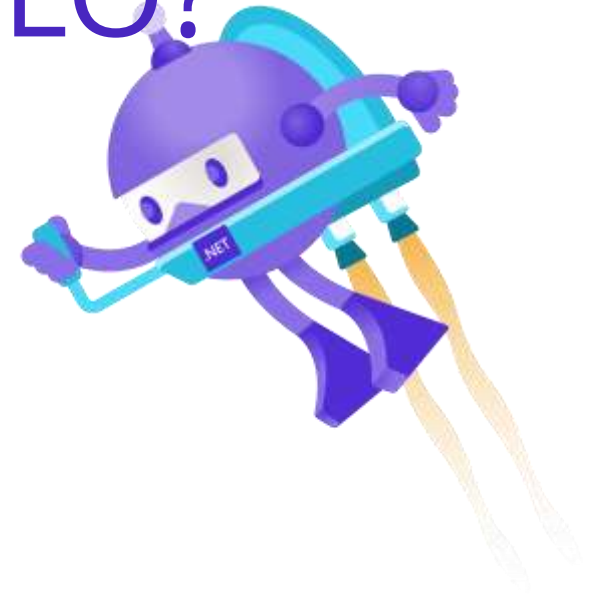
## Solution #2, 降低 Queue Length 的方法 => 分群組



區分群組，保留資源優先給 "驗證簡訊"，包含專屬的 Message Queue, Worker, Rate Limiter ...

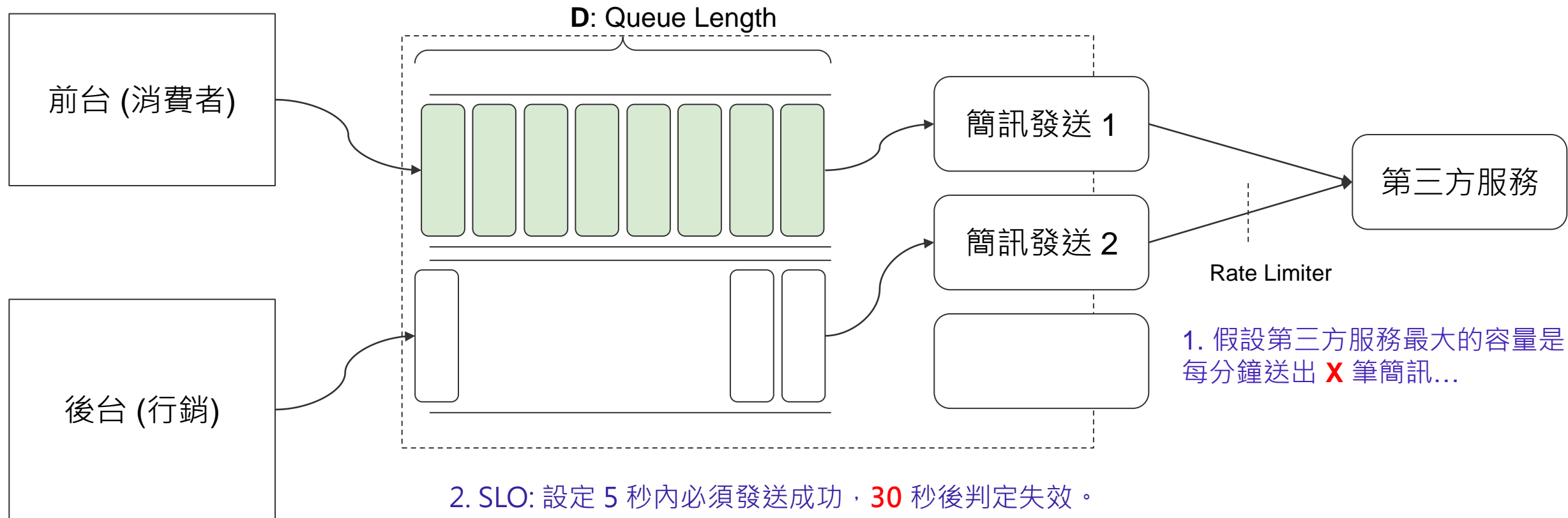
資源花在刀口上，完全用於加速驗證簡訊的發送。

Case: 如果長時間無法滿足SLO?



# 假想狀況：服務量滿載，驗證簡訊註定無法如期發送..

3. 只要 Queue Length 超過 **30X** 筆堆積的訊息，即使  
第三方服務全速運轉，新加入的訊息註定無法在 30 秒內送出...



(追加: 若超過 30 秒，則該簡訊直接判定失效，不再需要發送了。)

送出的驗證碼只在 5 分鐘內有效，延遲過久的簡訊就算成功送出，也沒有意義了...



## Think: 註定無法在期限內送出的訊息，還需要發送嗎？

背景: 驗證簡訊通常都有輸入的期限，一般而言是 5 分鐘。若考慮使用者操作習慣，通常超過 30 秒還沒發送成功的驗證碼，就不會被輸入了。

因此，若確定會延遲超過 30 秒才能送出的簡訊，那麼你還需要處理嗎？

Q: 如何從數據判讀這種狀況？

Q: 發生當下，前台系統能知道嗎？

Q: 當下該如何處理？

Q: 事後改善該如何處理？

## Q: 如何從數據判讀這種狀況?

若 { **Queue Length** } 超過上限，代表接下來的簡訊會超過允許的時間；  
若 { **發送速率** } > 處理的速率 { **Rate Limit** }，代表情況在持續惡化中 (越累積越多)。

## Q: 發生當下，前台系統能知道嗎?

**Design For Operation**；無法得知。除非你在開發的時候就有考慮到這個問題，或是搭配監控系統，偵測到這種狀況時自動調整系統 configuration / feature toggle, 來告訴系統實際的狀況。

## Q: 當下該如何處理?

需要靠“繩子”做為回饋機制 (生產者 / 消費者問題)。例如:

1. 若狀況發生，停用前台發送驗證簡訊的功能 (配合熔斷機制、健康偵測機制)。
2. 提供替代的方案，例如播號驗證 (提供 routing)。
3. 啟用備援方案，例如備援用的第二家簡訊商。

## Q: 事後改善該如何處理?

對外: 針對瓶頸改善。擴大 91APP / 第三方 之間的乘載量上限  
對內: 從產品設計與開發, 就建立回饋機制, 進行自動化控制  
(例如: 自動啟用備援服務, 自動啟用熔斷機制等等)

# SLO 與成本息息相關

Think: 你願意為 “service level” 付出多少成本?  
// 還記得前面的案例: “錢要花在刀口上” 嗎?



## (進階) 內部共用平台，該如何界定費用認列問題？

91APP 內部的非同步平台 NMQ，透過 Process Pool 能夠大幅提高運算資源的利用率，降低成本。不過高度共用，也帶來費用分攤認列的難題。

SLO 帶來解套的契機。第一步請先搞清楚你為了“SLO”花了多少錢：

1. NMQ 所有 nodes 的費用掛在架構團隊身上
2. NMQ 本身提供 usage logs, 並且提供合理的團隊費用攤提費用規則與試算。
3. Task 提交時同時規範 SLO, 提供監控與調度工具給各個團隊使用。
4. 需要擴充資源時，架構團隊提出須追加的成本與團隊確認。由團隊評估該進行哪一項？擴充資源, 程式碼優化, 調低 SLO 期待等決策。



# 回歸團隊，找到 SLO 與 COST 的平衡點

## **(Infra) Infrastructure / Platform:**

用更有效率的基礎建設，降低維持 SLO 所需要的 COST。  
(例如 Process Pool)

## **(Dev) Business Logic / Application Developer:**

從服務本身開始優化效能，降低成本。

## **(Ops) Operation Team:**

找出 SLO 與 COST 之間的關聯，定義最佳的平衡點。善用各種監控指標 (例如: 系統效能監控 搭配 轉換率的監控；)

# 試著評估不同的 SLO 所需要投入的 COST

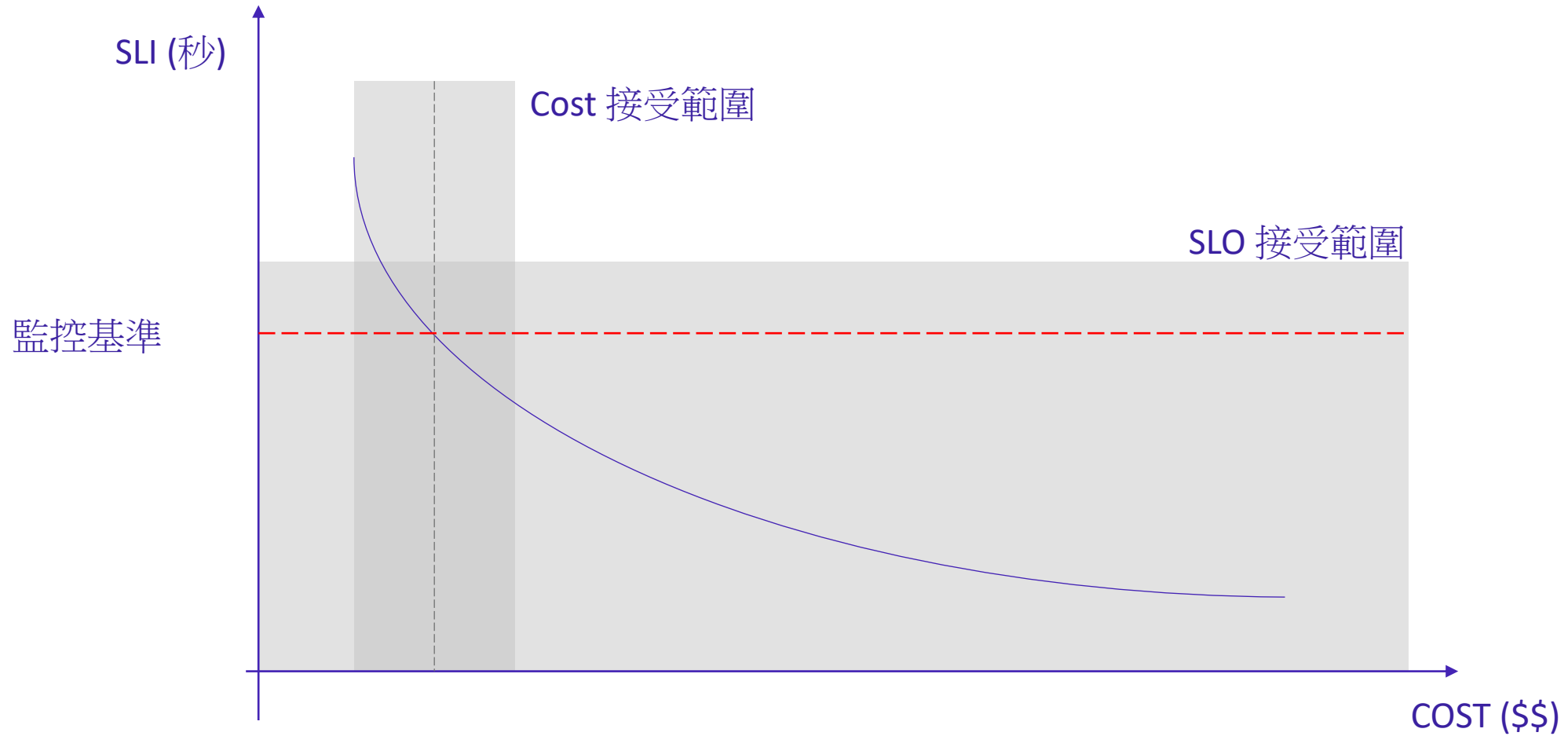
簡訊驗證: SLO = **5 sec**

1. Message Queue: **1x**
2. 簡訊商承諾 (基本費用): **1x**
3. Message Worker: **1x**
4. 備援管道費用: **1x**
5. ...

簡訊驗證: SLO = **1 sec**

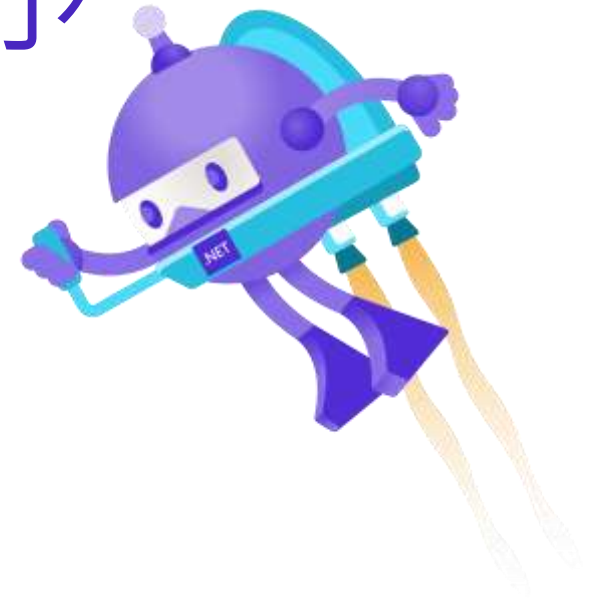
1. Message Queue: **3x**
2. 簡訊商承諾 (基本費用): **2x**
3. Message Worker: **5x**  
(有了 Process Pool: **1.2x**)
4. 備援管道費用: **10x**

# 了解 SLO 需要的 COST 之後...



# 更重要的交易後處理程序

比起時效，確保順利完成還更重要的任務



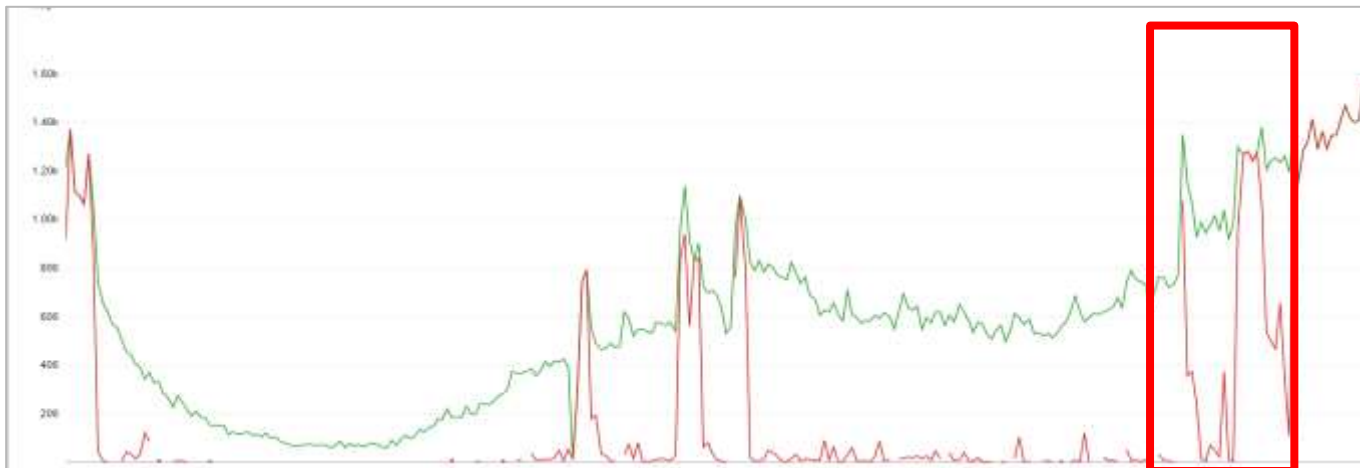
## 案例2, 交易後任務隨著流量增加，開始出現延遲



Efficiency

1. Task 執行的效率不佳

原因



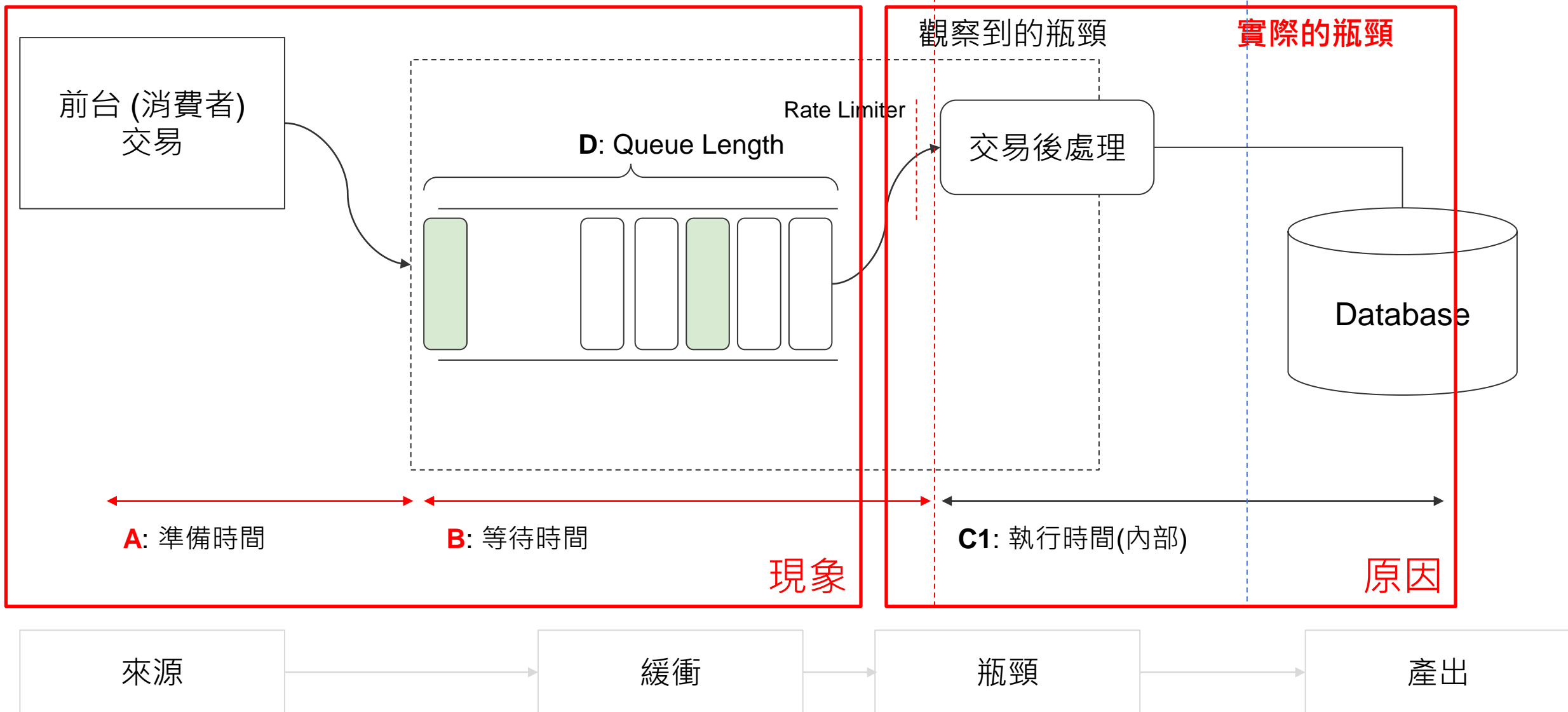
Received / Dequeue-Over-SLO

2. 導致消化速度低於產生速度，於是 task 開始在 queue 累積，光是排隊時間就超過額定 SLO

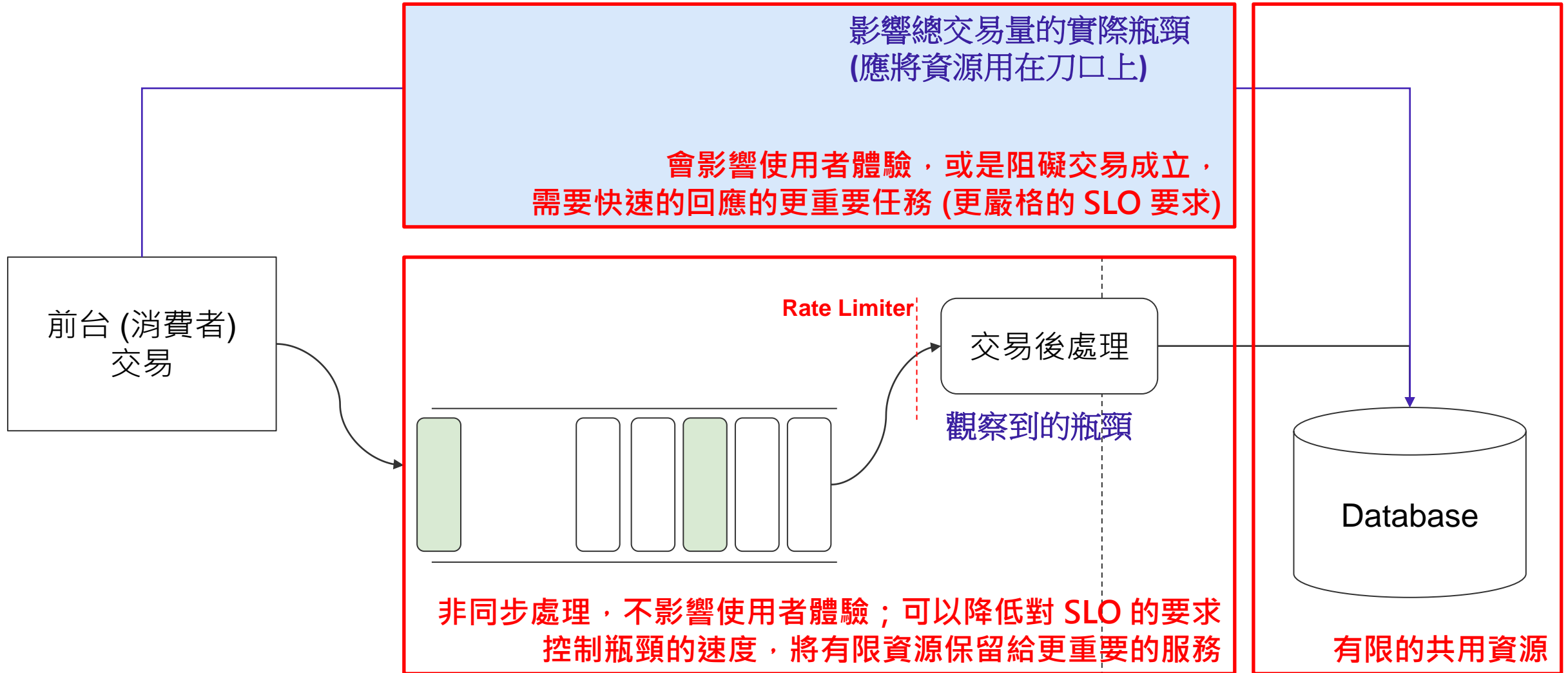
現象



# 先從數據指標，還原實際的狀況



# 綜觀全局；避免局部優化影響整體產出。



# 解決瓶頸的步驟:

1. 找到你的瓶頸； (特徵: 瓶頸的前一關通常都會出現庫存堆積的狀態)
2. 充分利用瓶頸； (既然是瓶頸，就別讓他停下來，盡可能讓他維持100% 的產能)
3. **非瓶頸協助瓶頸；**
4. 提升打破瓶頸； 調用非瓶頸的資源，確保瓶頸能充分發揮效益
5. 回到步驟一（找出下一個瓶頸），周而復始。

延伸思考: 生產線的 [瓶頸]，就是整體效能的控制點。

1. 改善瓶頸效能，就能改善整體效能
2. 控制好 [瓶頸] 的速度，就能控制整條產線的速度
3. 若無法改善 [瓶頸]，則要有能力在生產線失控 (過多 WIP) 前從源頭停止。

# 能夠選擇的做法

1. 將 SLO 調低到合理的程度 (原本定義過於嚴格)

現象・短期決策

2. 擴充 Worker 的處理能力

3. 改寫 Task, 做好最佳化改善執行速度

根本原因・長期改善

4. 擴大資料庫的處理能力

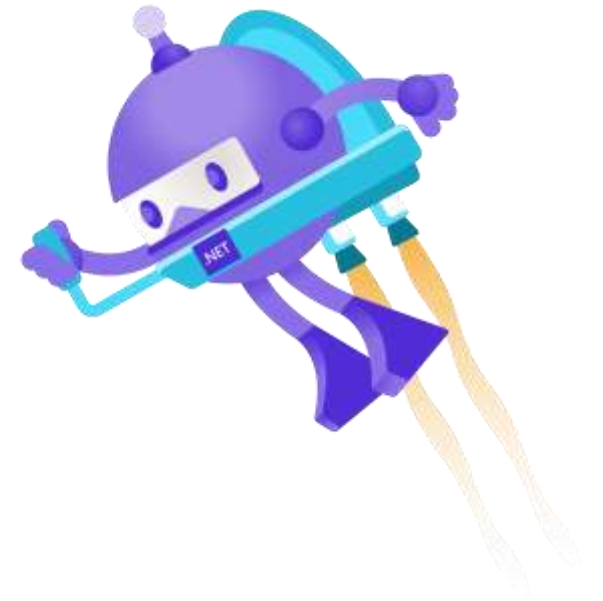
5. 限制 Worker 的處理能力 ( 將資料庫處理能力保留給交易 )

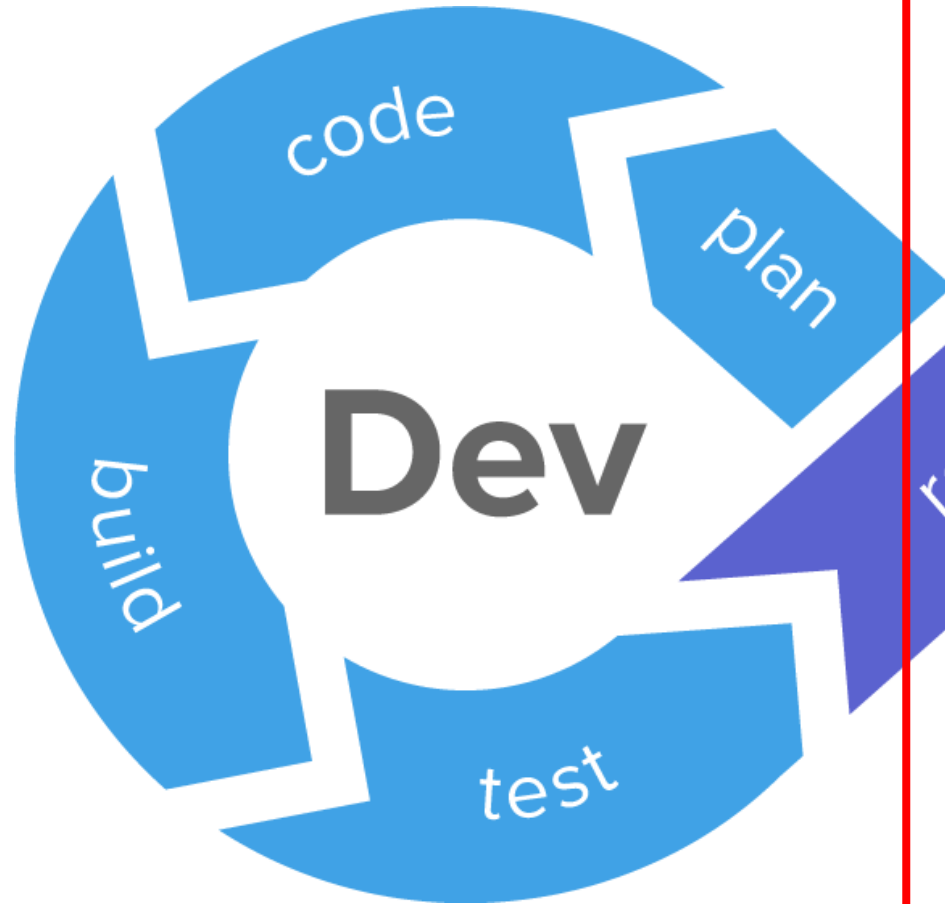
現象・短期決策

6. ....

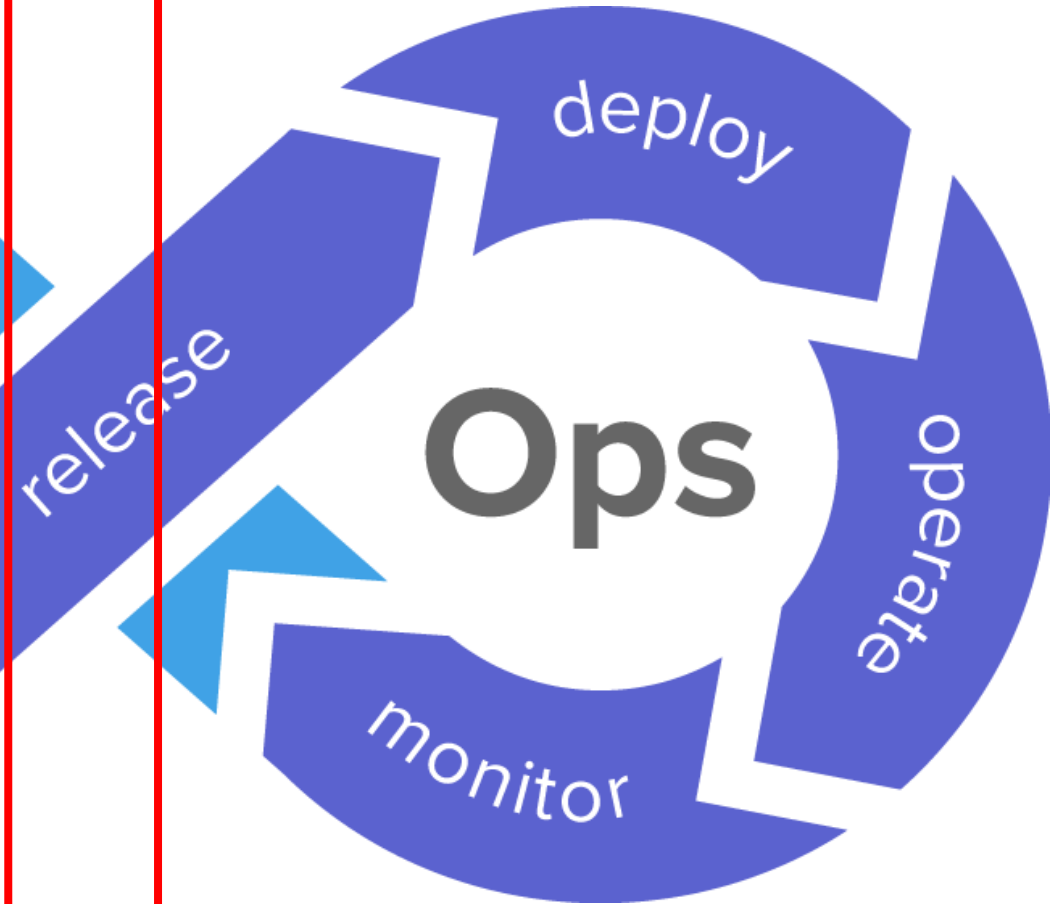
# 總結：

滿足 SLO 要求的改善過程，就是落實 DevOps 的精神



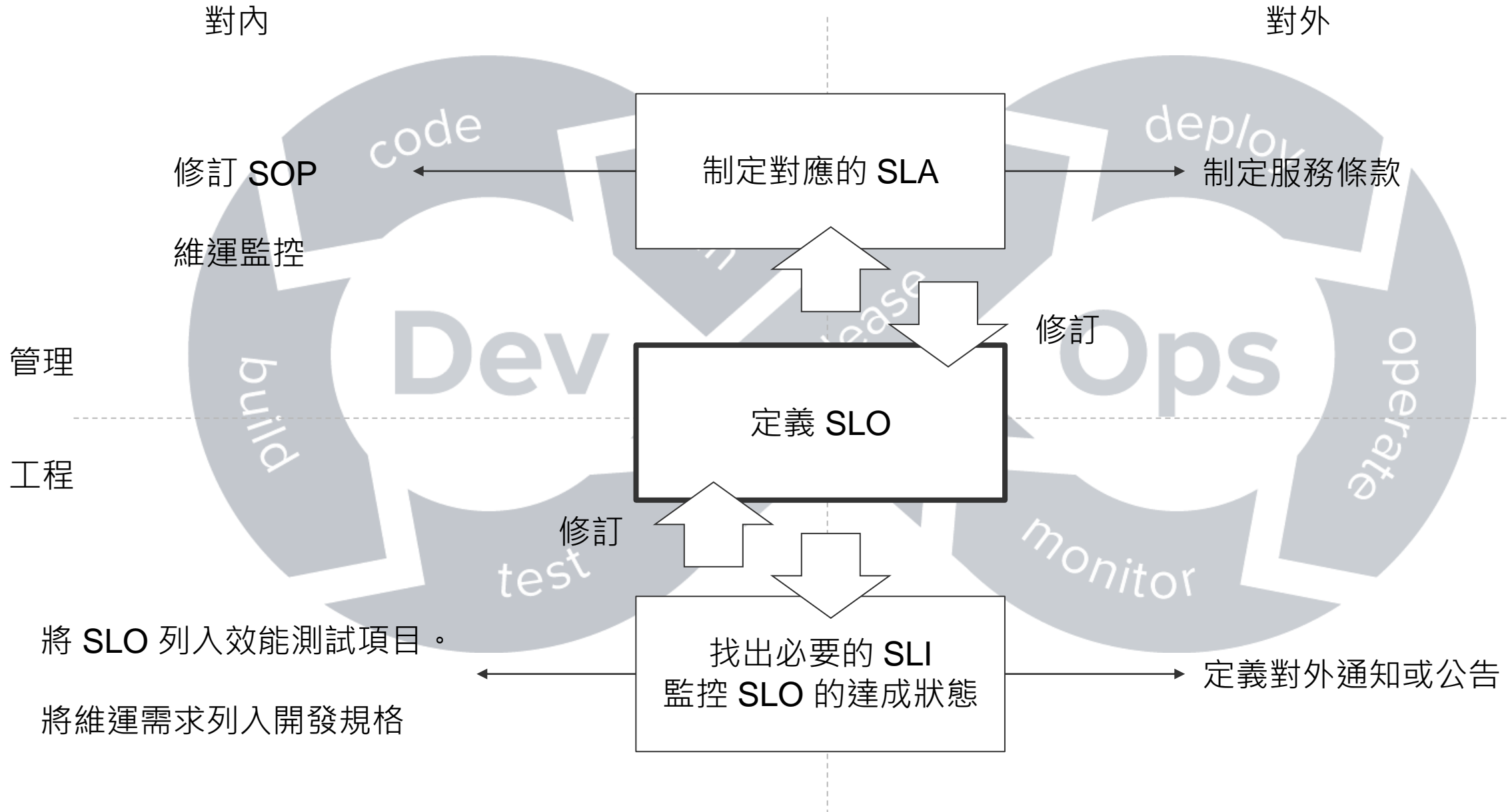


為了維運而設計開發



以滿足 SLO 為目標





# Thanks for joining!

Ask questions on Twitter using #dotNETConf



91APP 線上考題



91APP Tech 粉絲團



# 現場或對線上考題有任何問題  
# 歡迎到 91APP 攤位與講師交流 ♥

.NET Conf  
2020

# 特別感謝

91APP  
Technical Network



KK TIX



  
HackMD



 Microsoft

 Build School

STUDY4  
為 學 習 而 生

以及各位參與活動的你們

