

# Modern Web Architecture Design Journey

Ant

yftzeng@gmail.com

2017-08-11



# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

懂一點程式開發

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

台灣駭客年會 2 次的演講經驗  
待過 3 年資訊安全產業

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

涉獵一點著作權、商標權及專利權  
撰寫過數篇發明專利

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

目前負責四間新創公司的技術選型  
協助數間友人新創公司的架構討論  
**無償**

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

目前負責四間新創公司的技術選型  
協助數間友人新創公司的架構討論

無償

血淚史

# 自介

程式開發 X 資訊安全 X 智慧財產權 X 創業

目前負責四間新創公司的技術選型  
協助數間友人新創公司的架構討論

# 技術選型

技術選型，選擇對公司或團隊最佳的技術組合

## 技術選型

理想

技術選型，選擇對公司或團隊最佳的技術組合

# 技術選型

現實

所謂技術選型，大多數情況仍由關鍵人物決定偏好  
例如 **CTO** 或架構師

# 技術選型

現實

所謂技術選型，大多數情況仍由關鍵人物決定偏好  
例如 **CTO** 或架構師



這是我遇過最多的抱怨！

# 技術選型

現實

所謂技術選型，大多數情況仍由關鍵人物決定偏好  
例如 **CTO** 或架構師



很多人認為【空降神兵】能解決問題。

公司內部？顧問？接案 / 外包？

但有時解決的反而是《提出問題的人》。

# 技術選型

現實

所謂技術選型，大多數情況仍由關鍵人物決定偏好  
例如 **CTO** 或架構師



天降神兵：.NET 不行，全部改 Java。

天降神兵：PHP 開發太慢，全部改 RoR。

天降神兵：AngularJS 沒人用，全部改 VueJS。

這些決策未必錯，但問題是《為什麼》  
以及更重要的：有無《信服人的理由》

偶爾任性是可愛，一天到晚任性是妖孽

~ ANT ~

# ① 架構先決

無視人員、流程只講技術，是耍性子  
架構會影響公司文化、商業擴展；思維更要超越程式碼層次

CTO/ 架構師不該只懂技術

## ② 沒有完美的架構， 只有最適的架構

無視場景只講架構， 是耍流氓

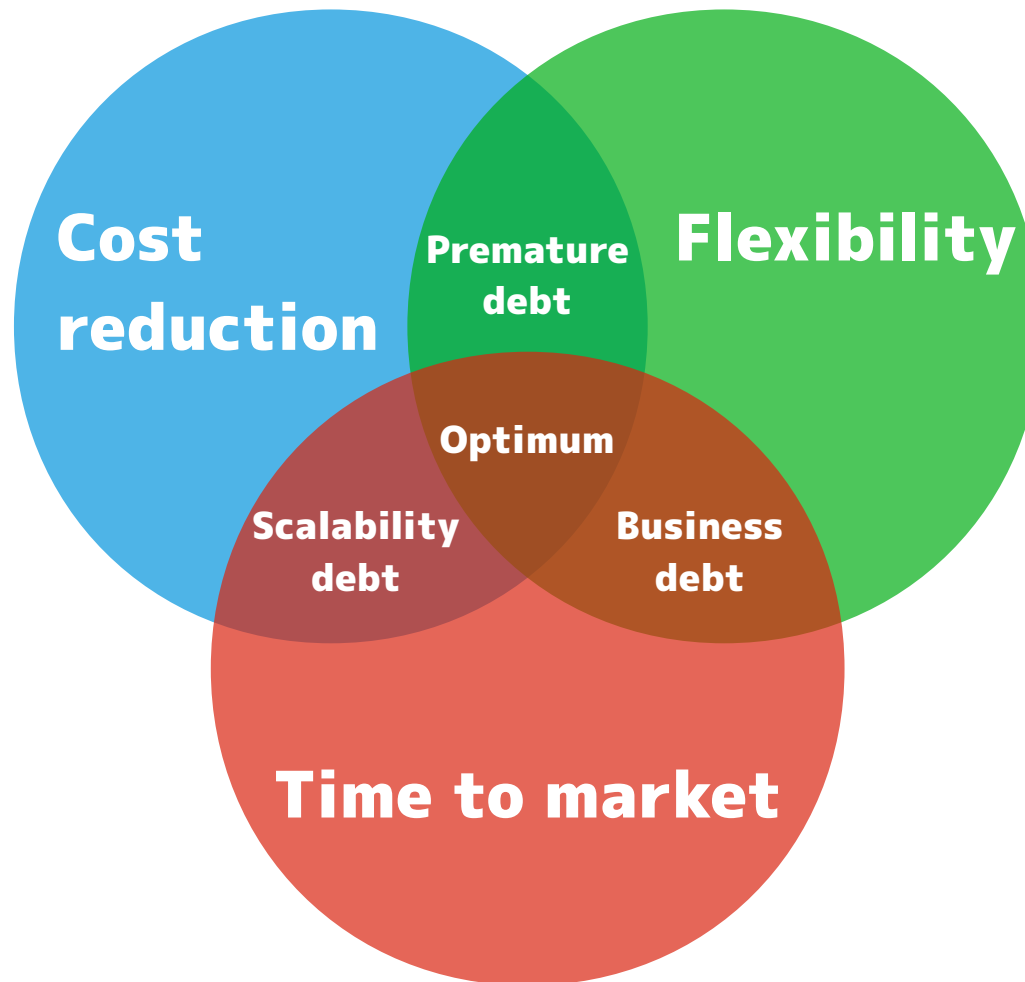
若無法舉出架構的缺陷， 代表你還無法掌握  
盲目套用別人的架構， 並不會讓你變得跟他一樣好

沒有一個架構能適用所有場景

### ③ 架構是演進的，預想但不過早調優

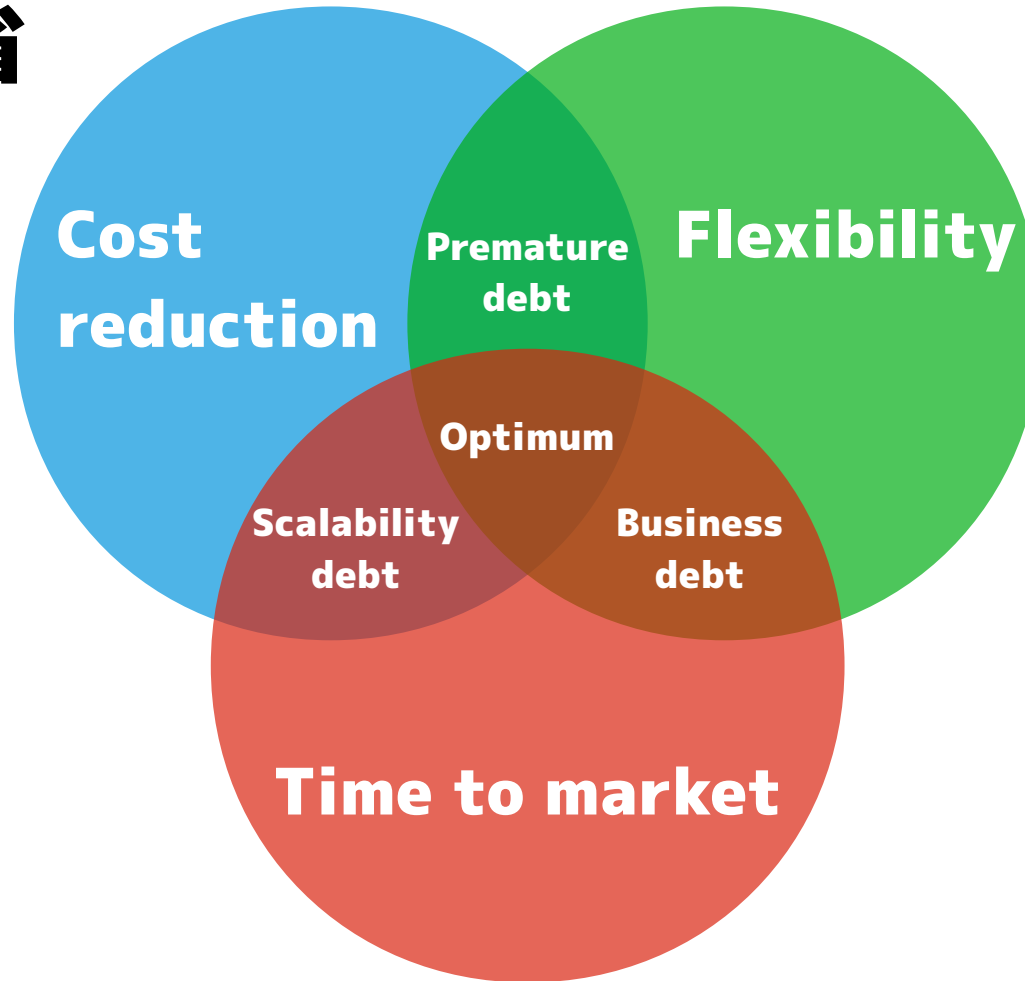
無視未來只求急行軍，是要短視  
兵馬未動，糧草先行，預想下一步，下下一步，甚至下下下一步 ...

歡迎進入八奇的思考領域



省

好



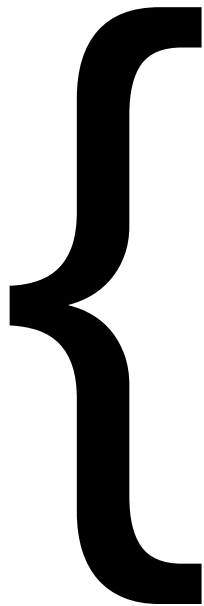
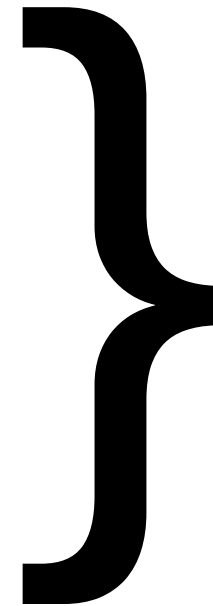
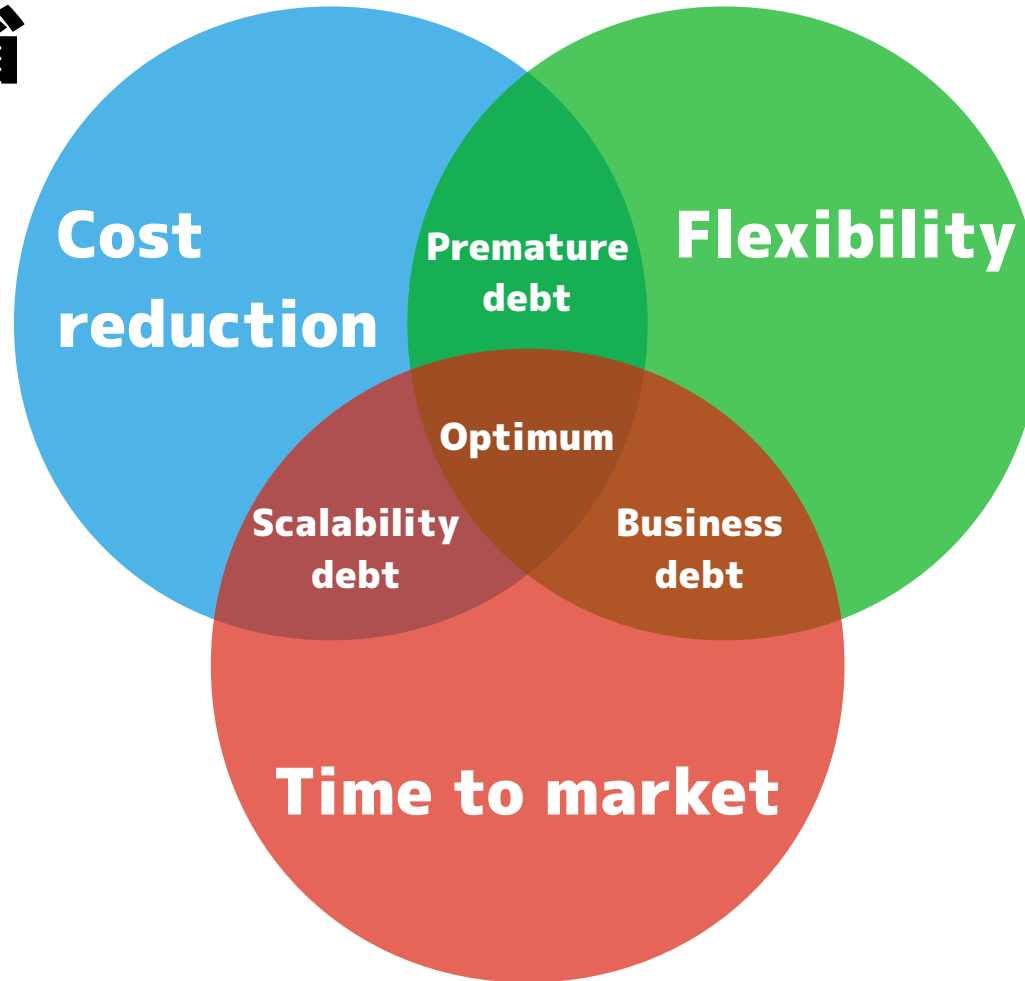
快

架構債

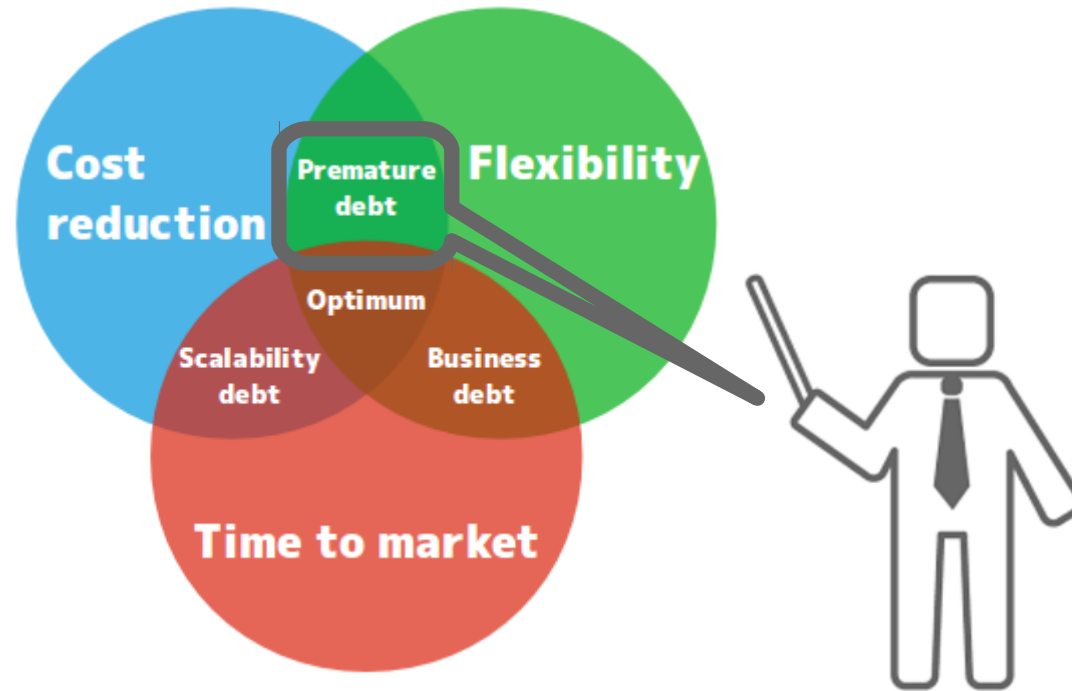
# Architectural debt

省

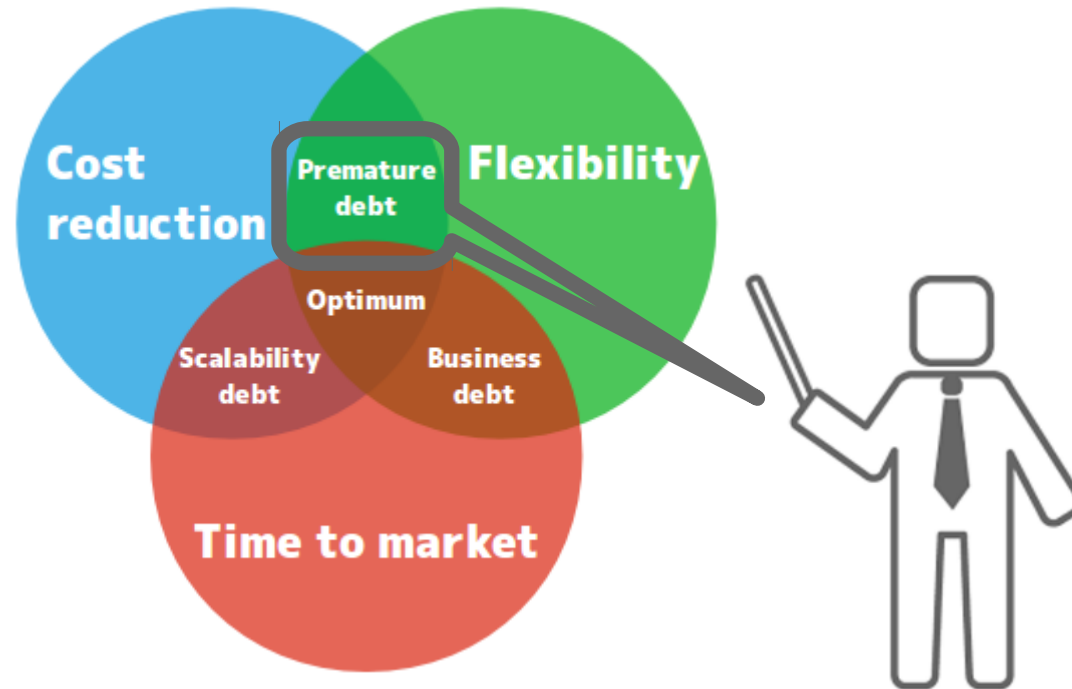
好



快

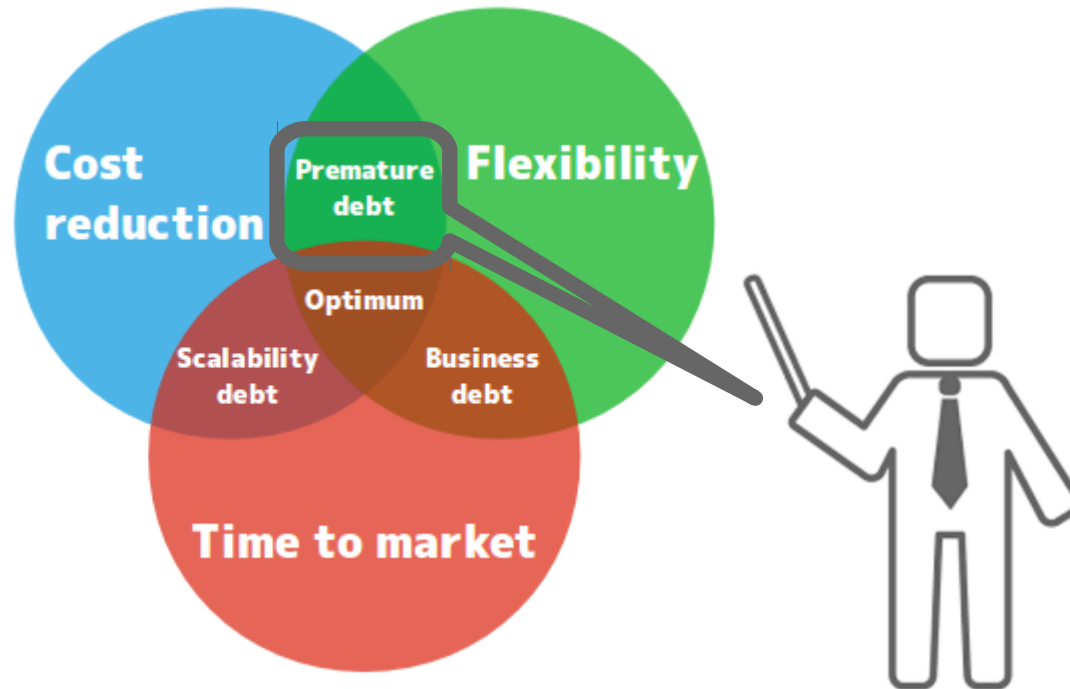


有經驗的技術團隊



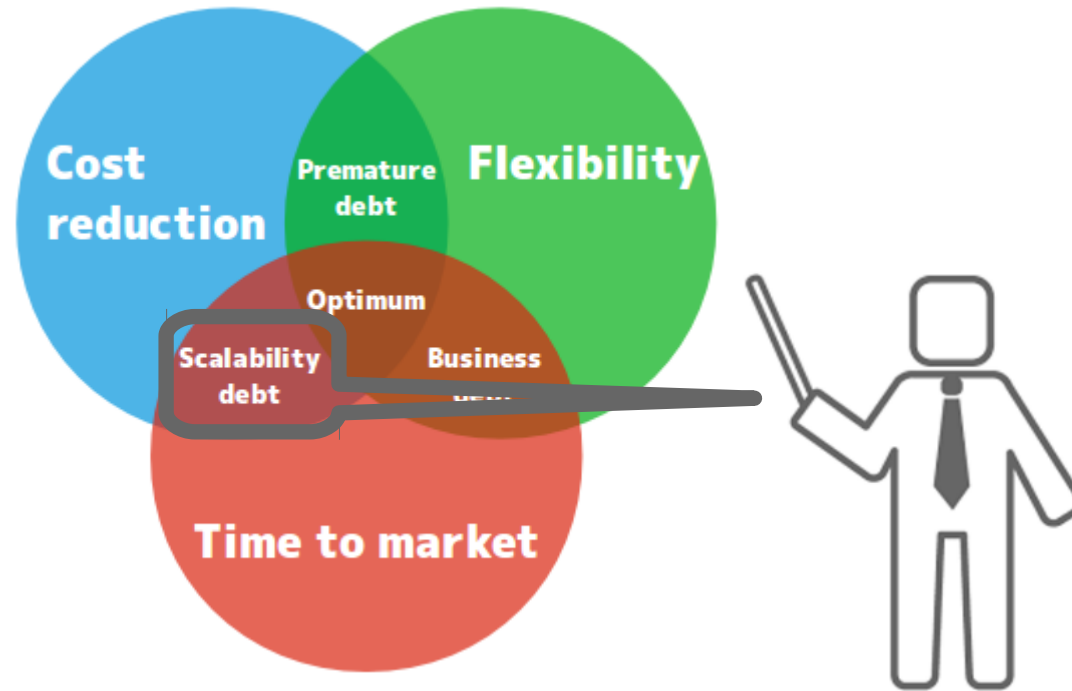
有經驗的技術團隊

喜歡採用最新、最好、最大的架構

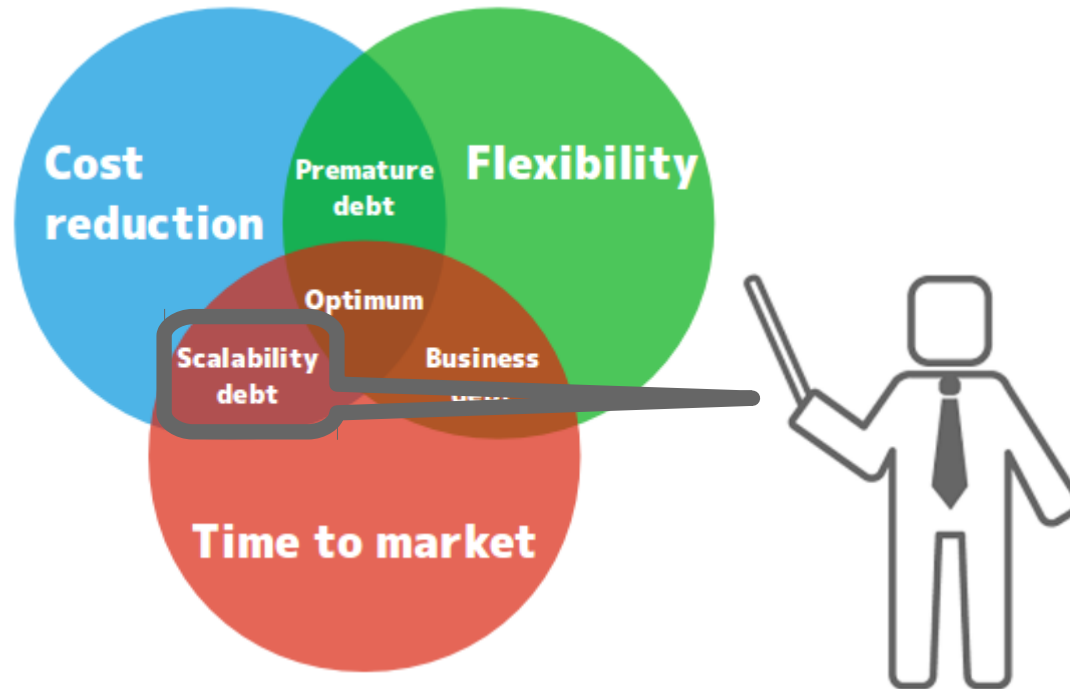


**You Are Not Google**  
**Facebook**  
**Amazon**

別為了高大尚，而迷失自我



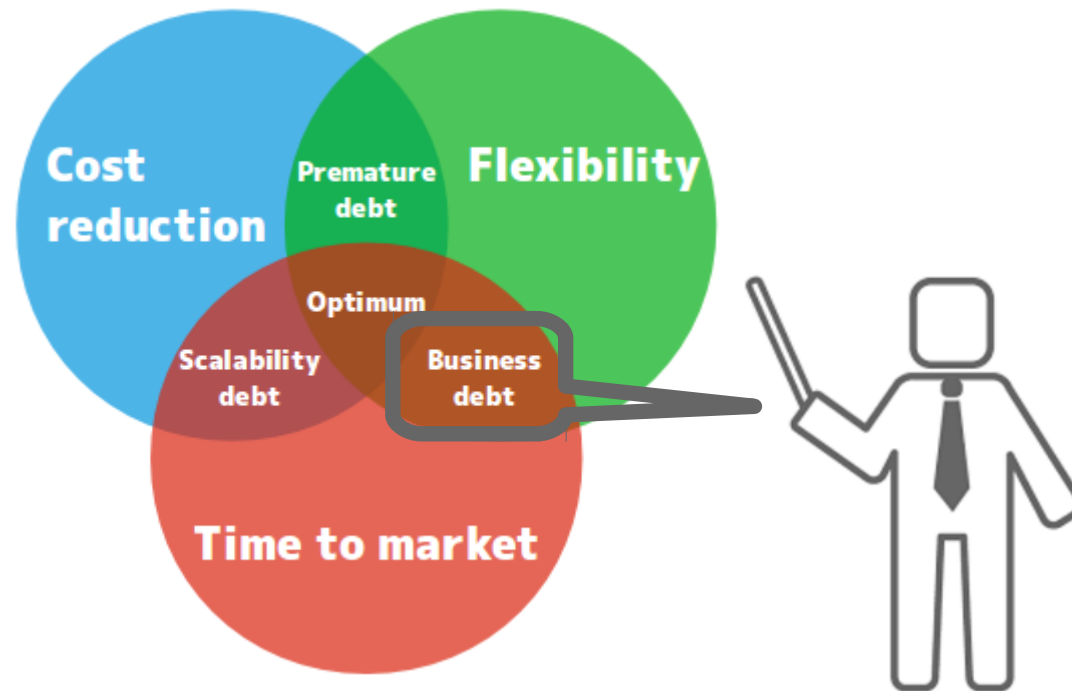
欠缺架構經驗的團隊



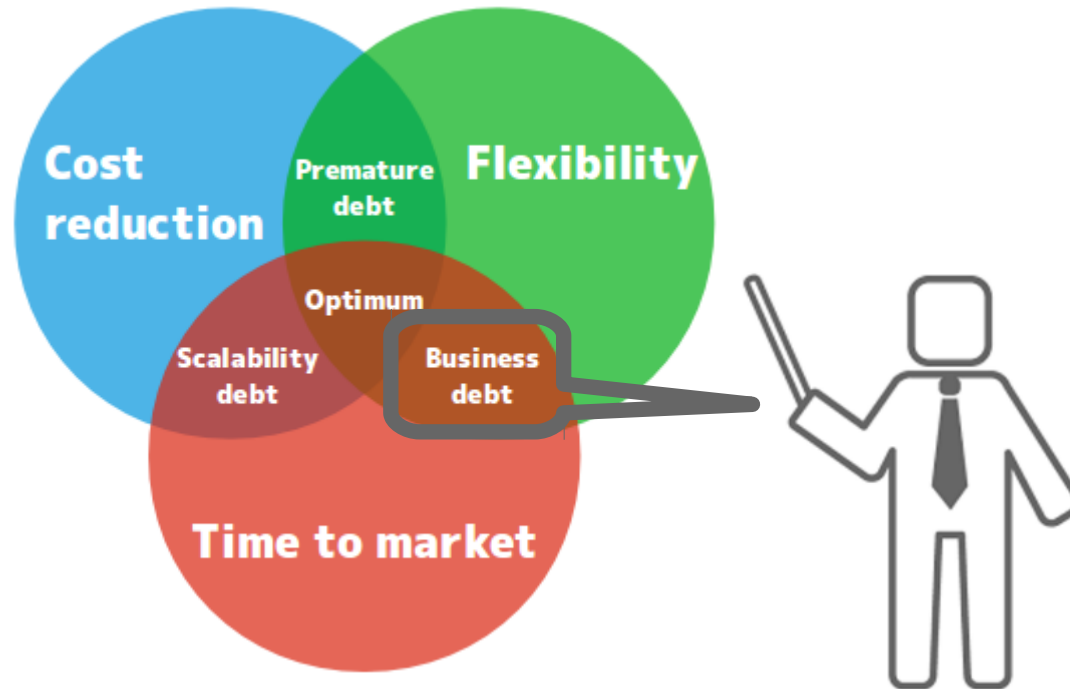
問題

**Non-reusable Code**  
**Non-HA / NonScalability**  
**Legacy Code**

債築債



(起初) 不缺錢的團隊



# Fund raising !

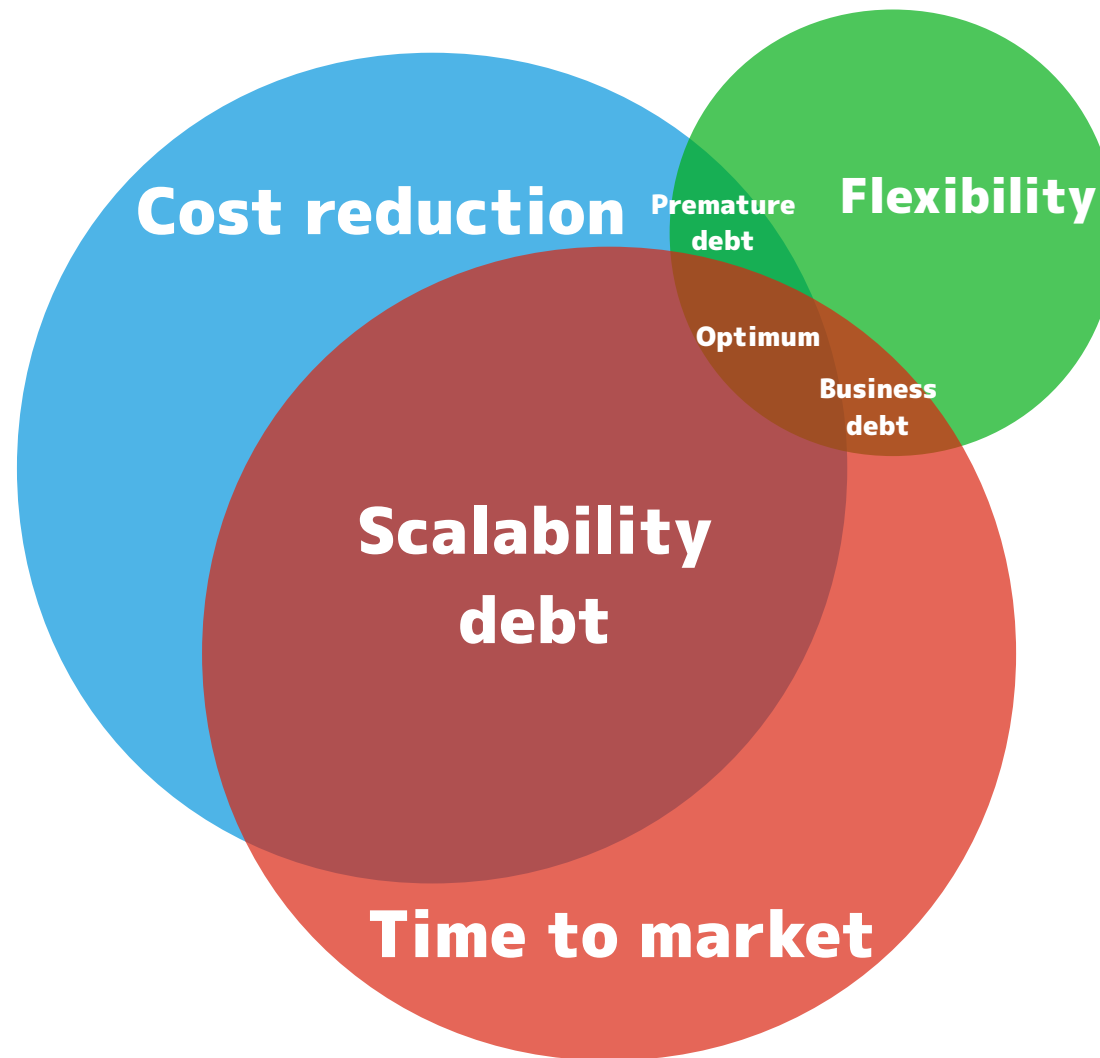
or

奔耗

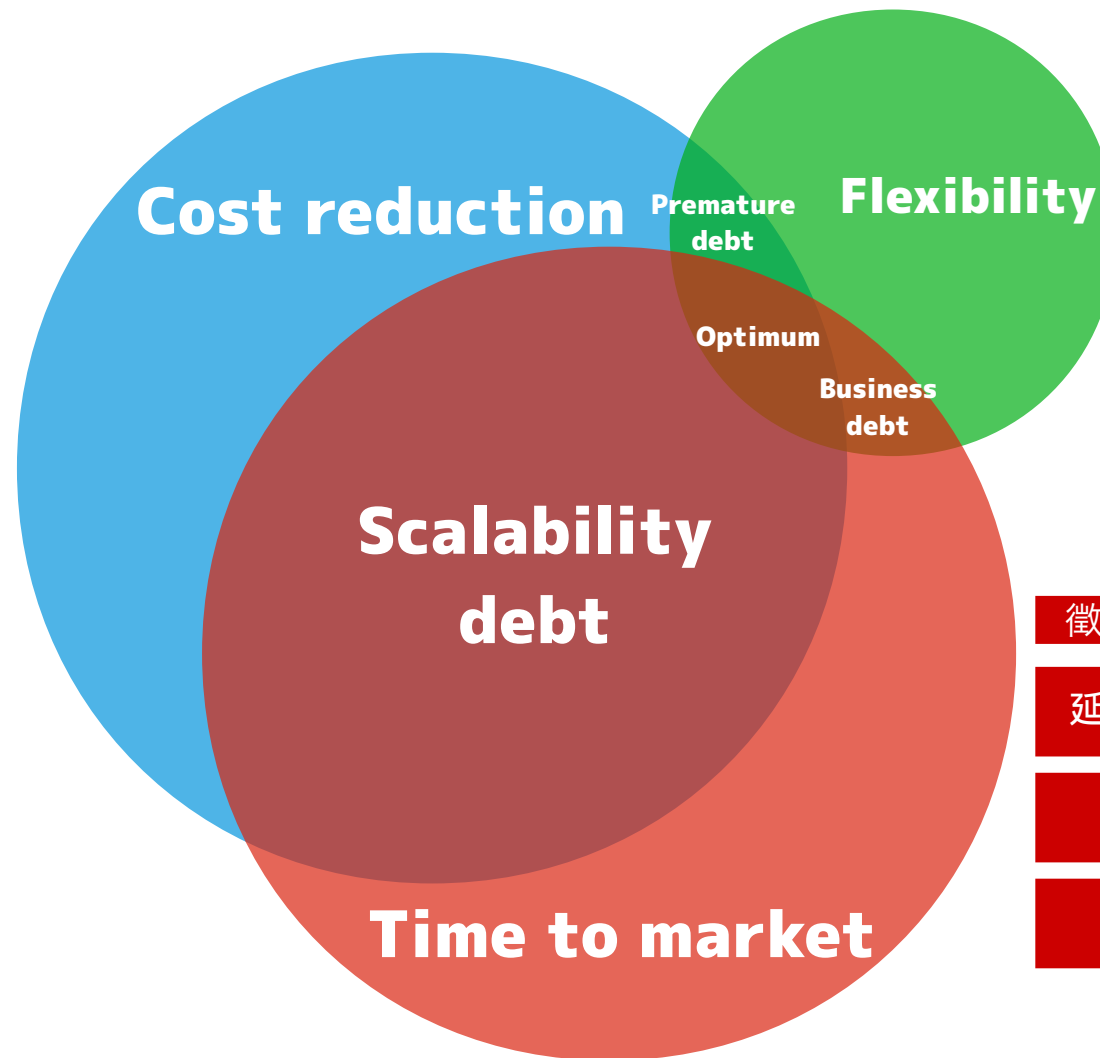
# Burnout !?

利逐債

# Architectural debt



# Architectural debt



徵兆

延展性不足以支撐業務成長

架構升級積重難返

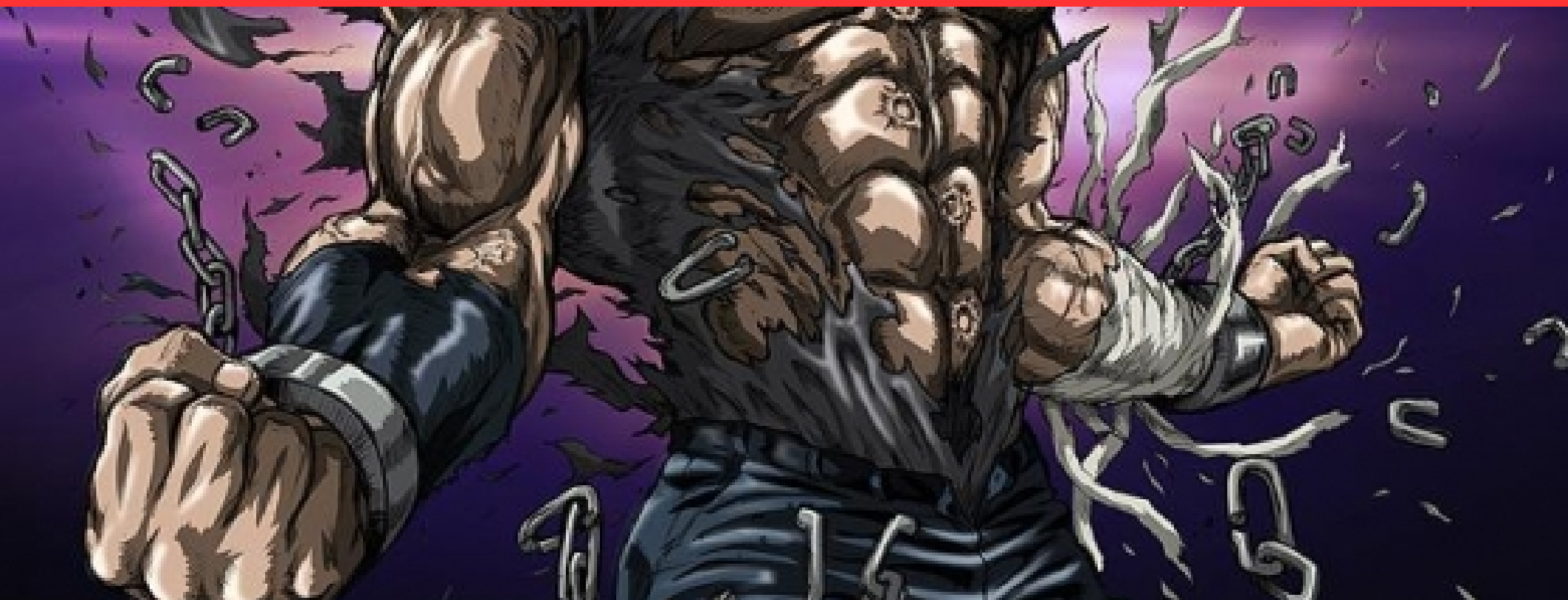
約束（硬限制）未滿足

**救世主？**





**Agile? 敏捷教練？**



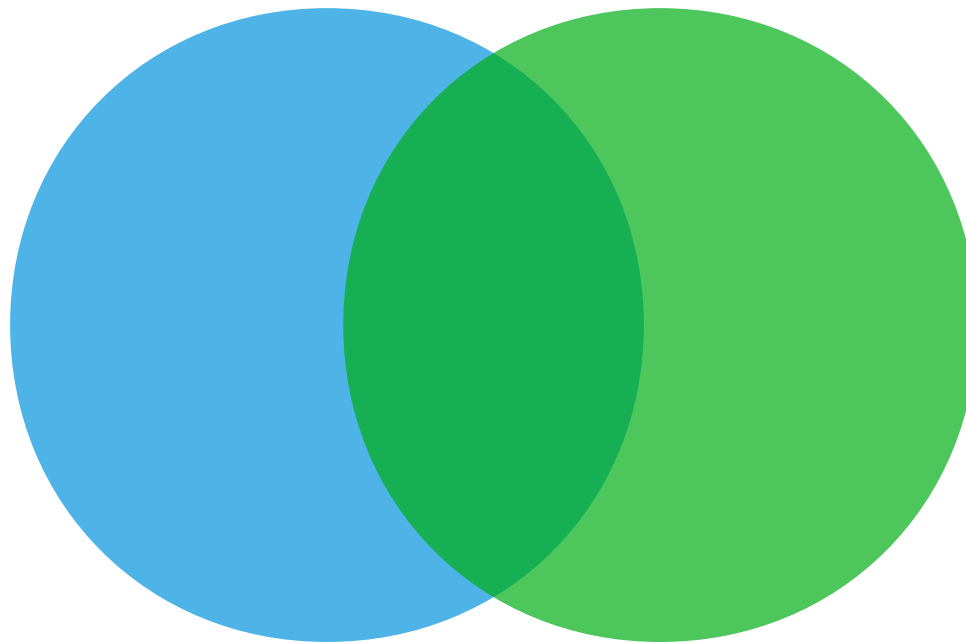


**Agile? 敏捷教練?**

**Architect? 架構師?**



## **Agile Architect**



Agile 與 Architect 是有交集的

Good architecture brings agility

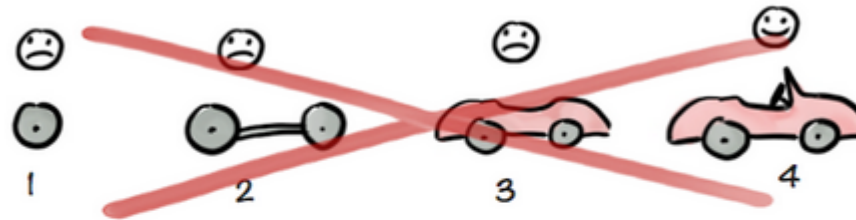
好的架構帶來敏捷

- Saugatuck -

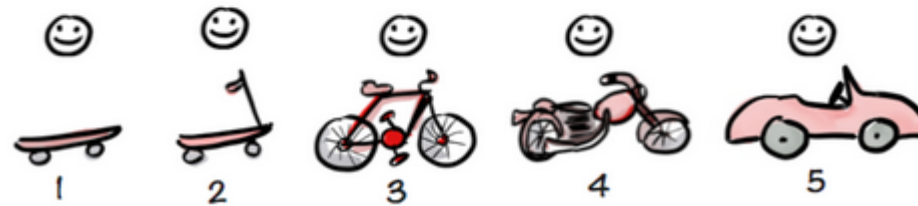
# Minimum Viable Product

最小可行產品

Not like this....



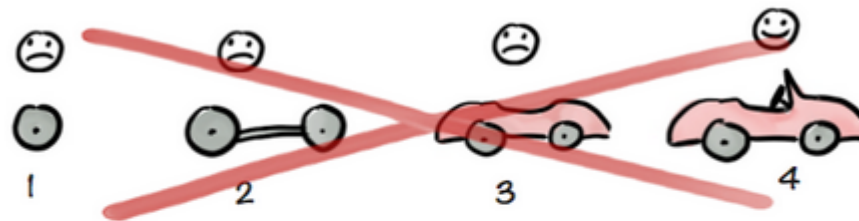
Like this!



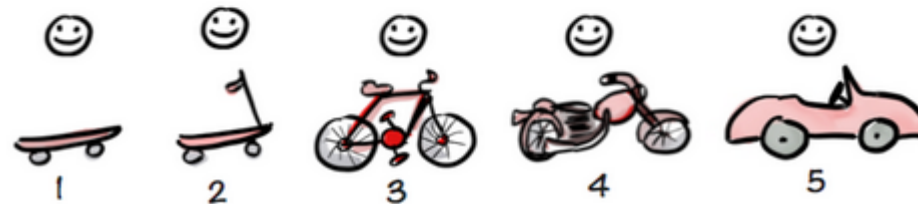
# Minimum Viable Product

最小可行產品

Not like this....



Like this!



MVP 若沒有控制好，技術債將迅速增長

先前程式少能復用  
遺舊程式難以去除  
架構變動牽扯不清

$$E = mc^2$$

$$E = mc^2$$

$$\text{Errors} = (\text{more code})^2$$

$$E = mc^2$$

$$\text{Errors} = (\text{more code})^2$$

錯誤隨著程式碼數量呈平方次成長

# 最後廢碼定理

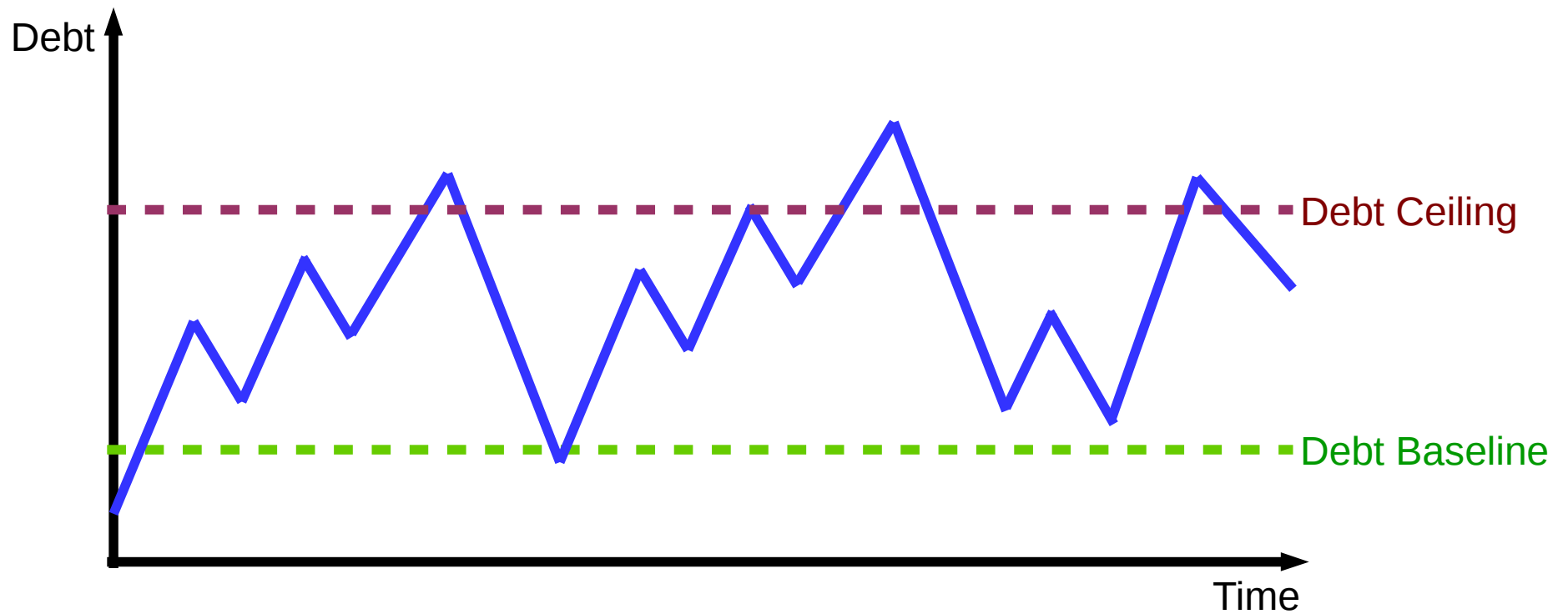
程式廢碼產生的速率，隨著接近專案最後死線呈等比倍增

真相只有一個，但死相可以有很多個

~ ANT ~

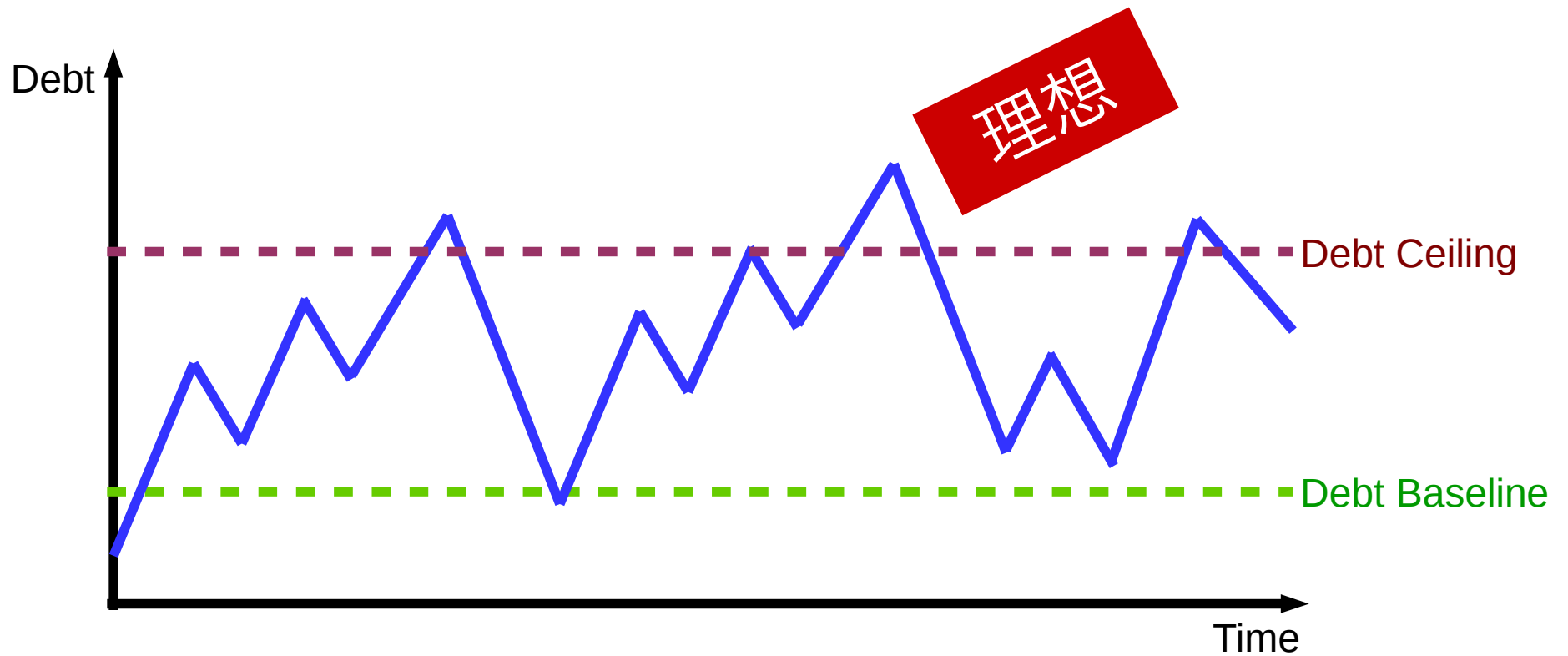
# Minimum Viable Product

理想與現實



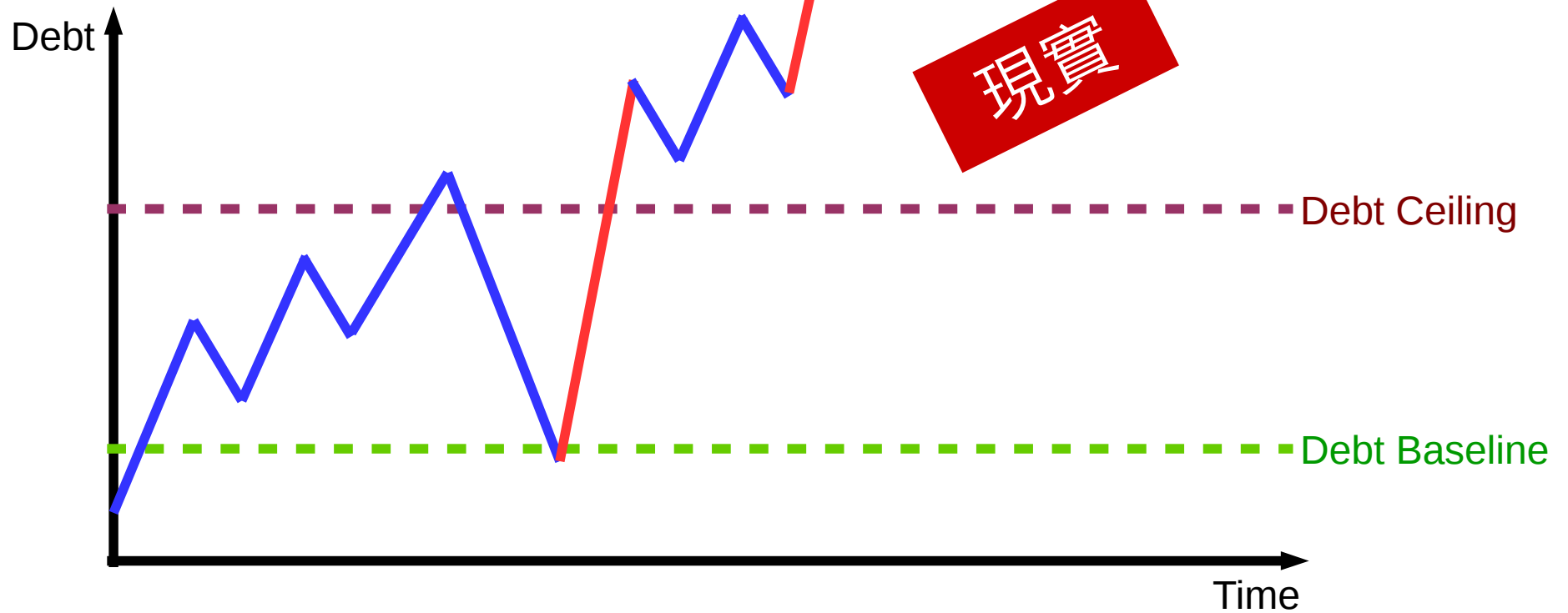
# Minimum Viable Product

理想與現實



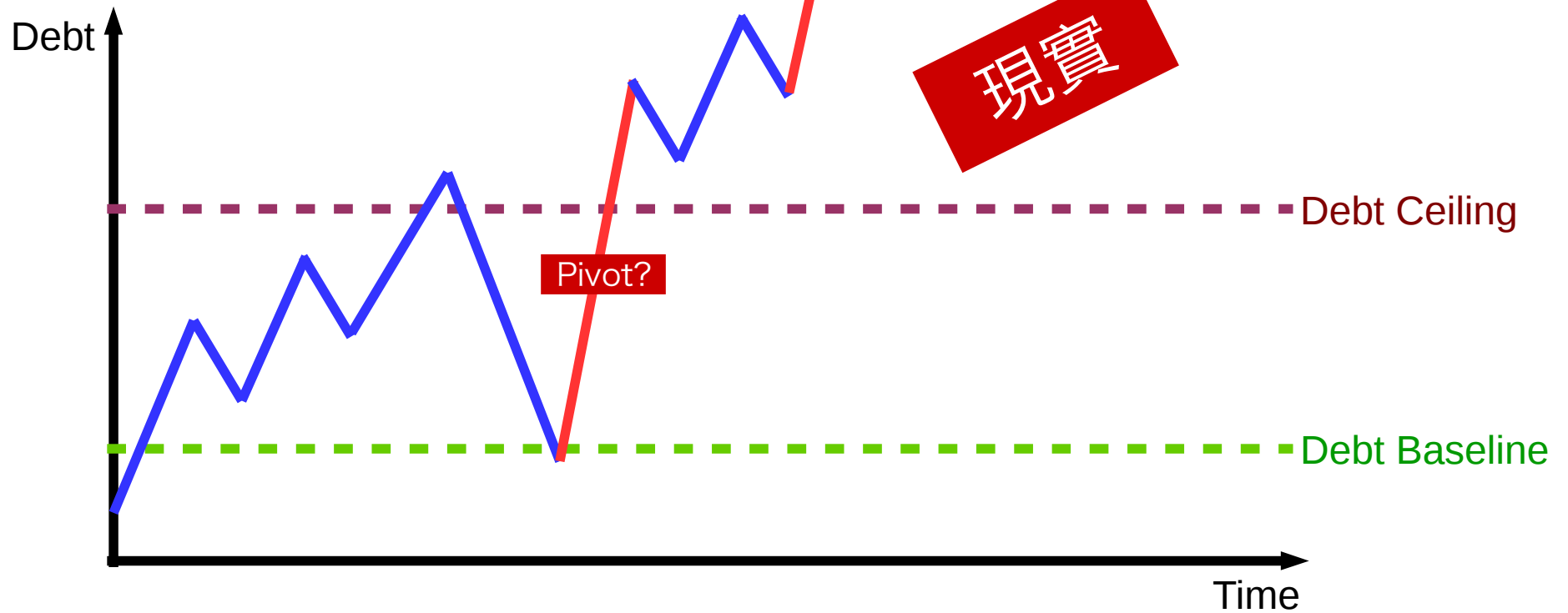
# Minimum Viable Product

理想與現實



# Minimum Viable Product

理想與現實



Pivot 聽起來很稀鬆平常，就像放屁一樣，但只有蠟筆小新放屁會飛

~ Ricky Chen ~

# Minimum Viable Product

大家可能聽過一些說法：



市場驗證成功後，我再拿投資人的錢來技術轉型就好了。



事實上，迅速成長的公司，沒太多時間進行技術轉型。

再加上之前為了快速驗證 MVP 所遺留的技術債。

只會讓技術轉型變得遙不可及。

# Minimum Viable Product

於是：



現有團隊不行，我砸錢找大神（大牛）重構系統總可以吧。

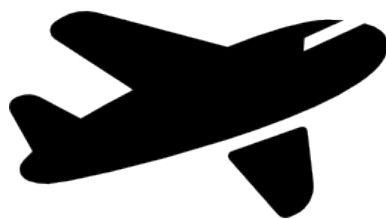
# Minimum Viable Product



強烈的既視感？

更重要的是  
有考慮過大神（大牛）的感受嗎？

架構重構是個苦工，很多時候他們也是千百個不願意



技術難度高。

但迅速成長的公司，仍然有很多需求待開發。  
同時還要處理累積的技術債，很可能因債築債而嚴重惡化。

空中換引擎

---



技術難度低。

但商業模式是否能夠允許新功能推進緩慢或無進展？

落地換引擎

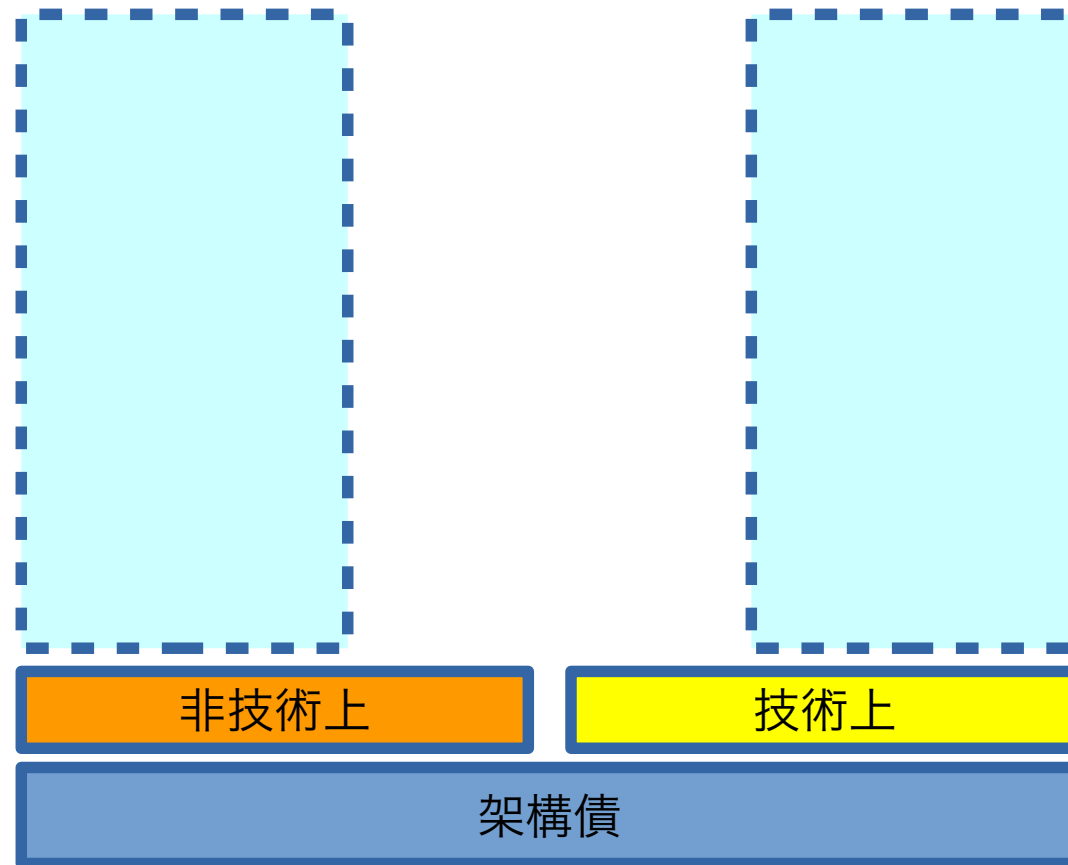
---

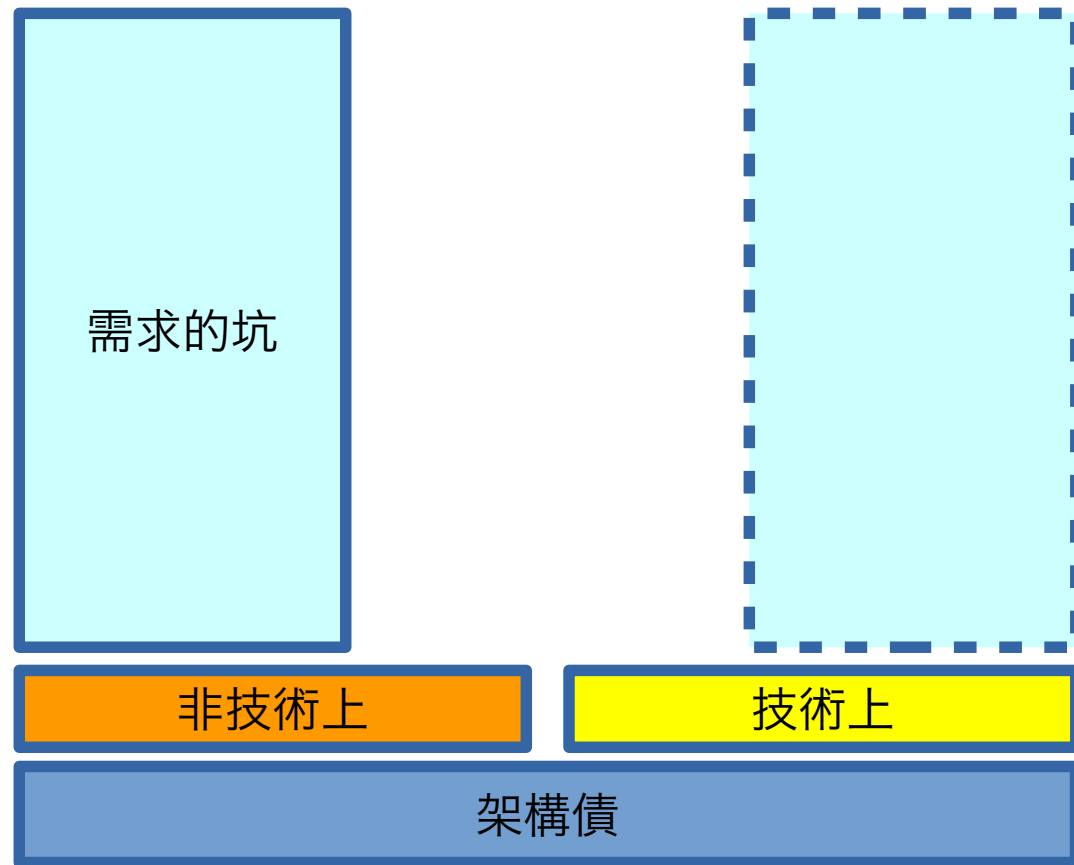


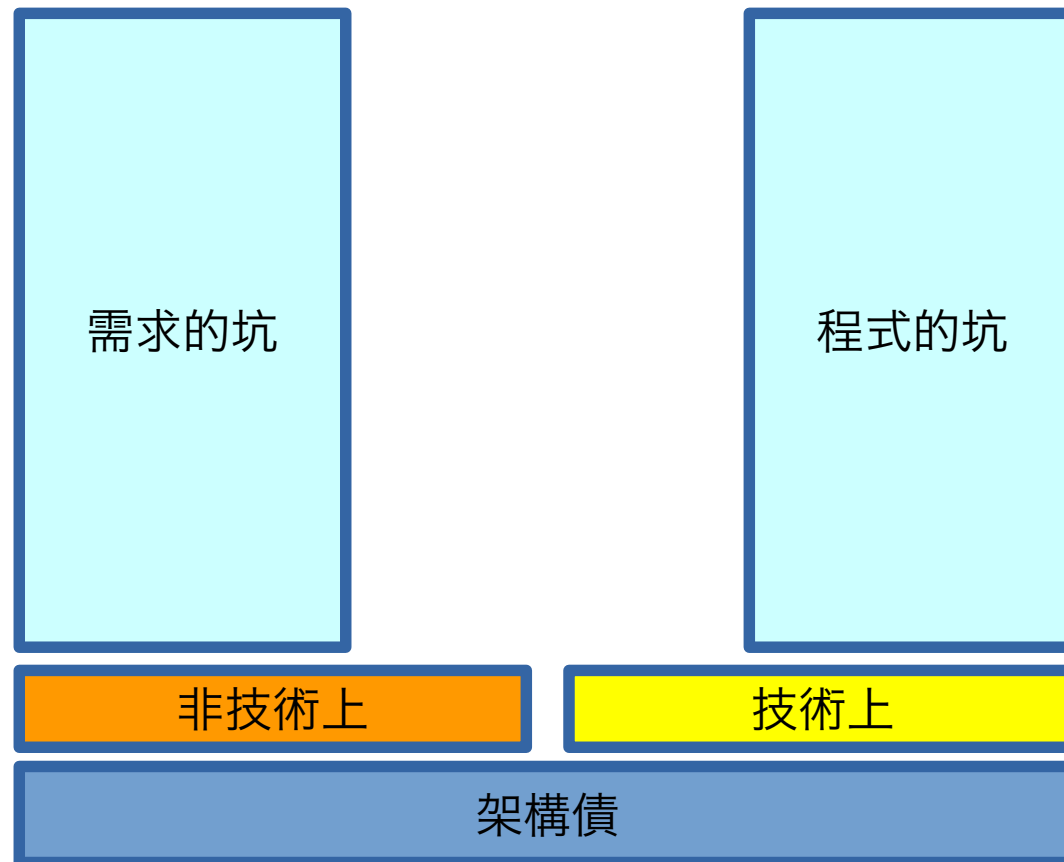
資源投入巨大。

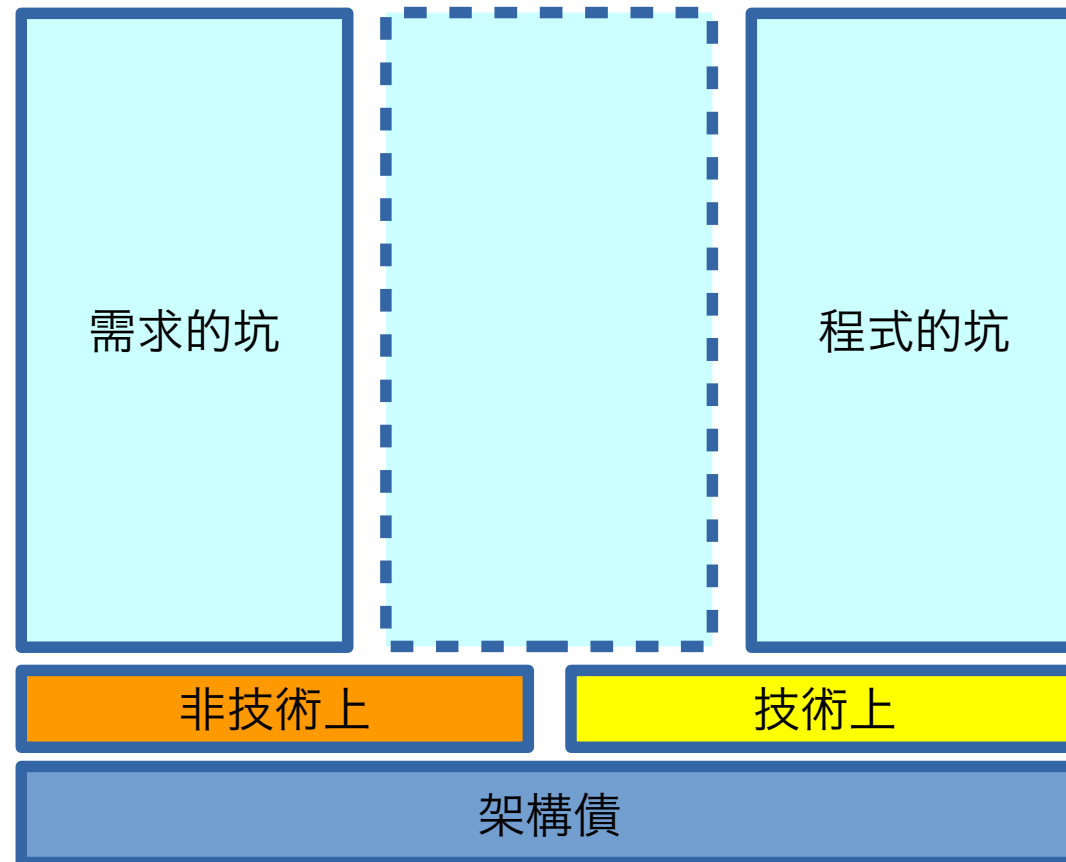
不是現行團隊拆分戰力，就是要另招一批團隊開發新引擎。  
但若問題本質不解決，新引擎未來也會遭遇一樣的問題。

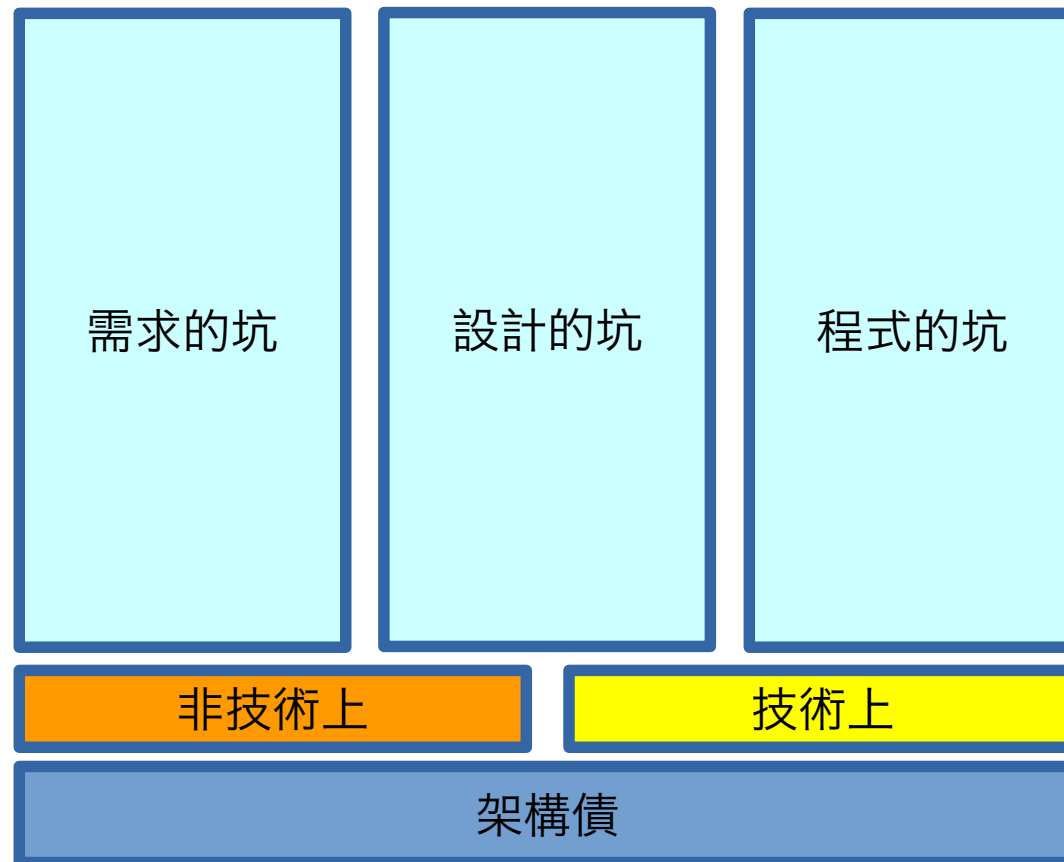
空中小修  
地面重建



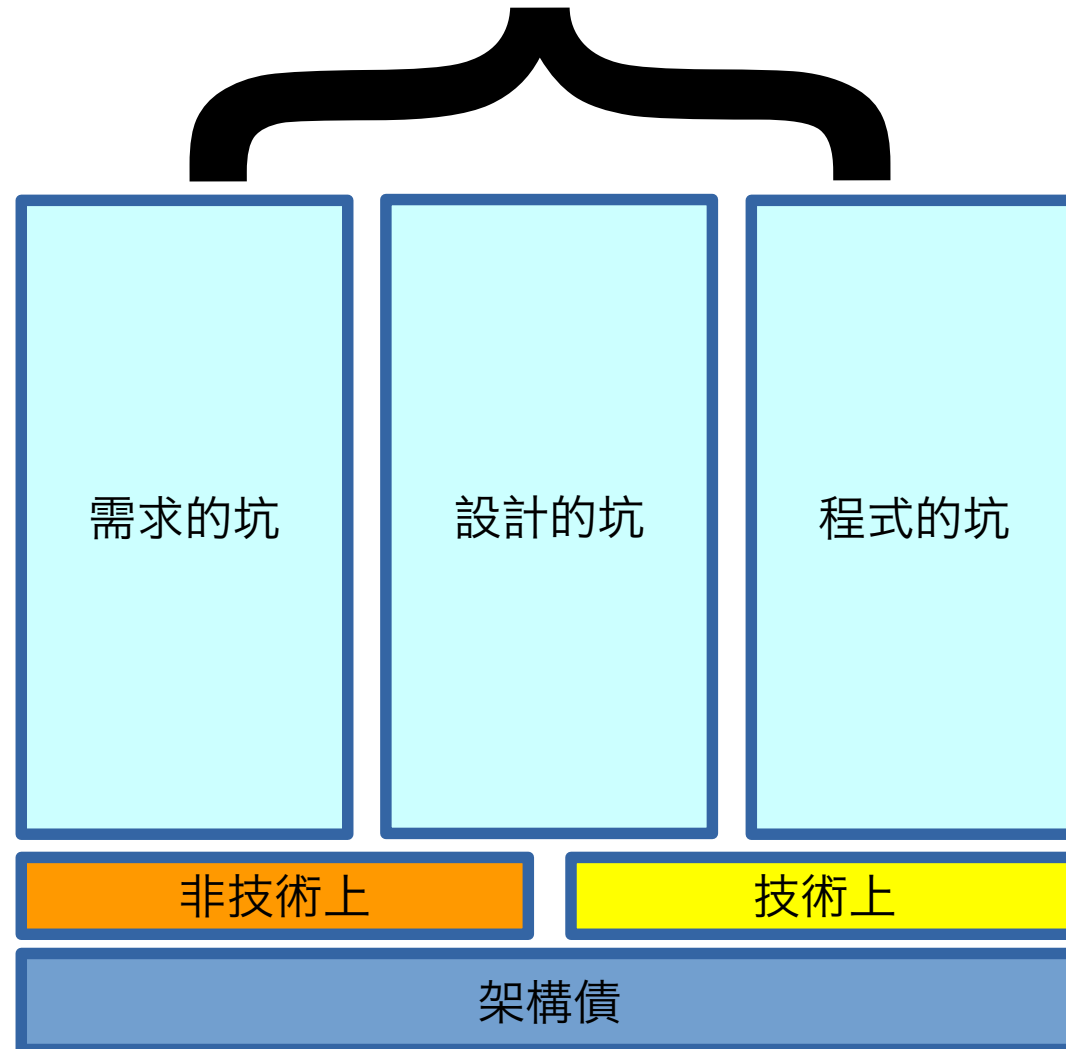








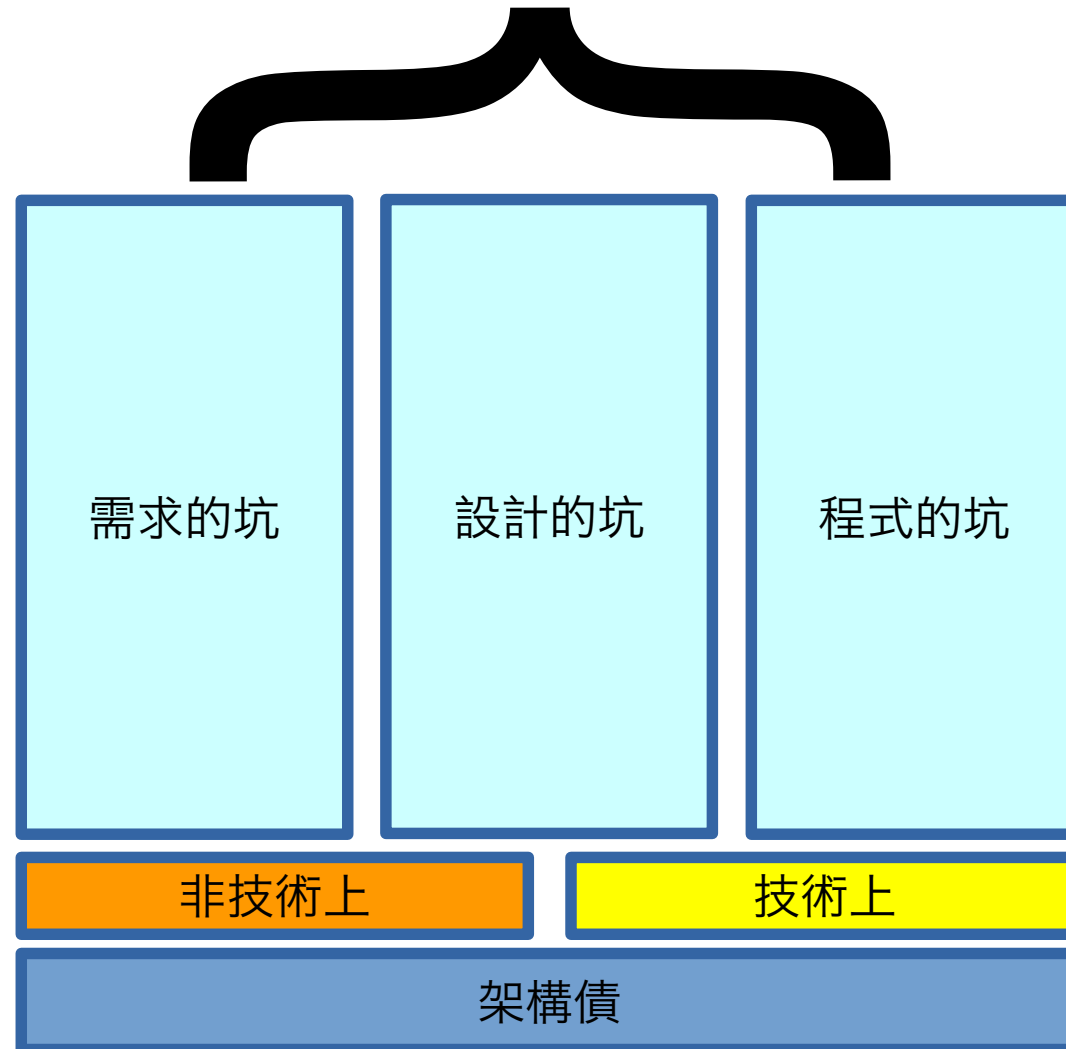
**Agile, Scrum, Kanban, DevOps, TDD, BDD, ...**

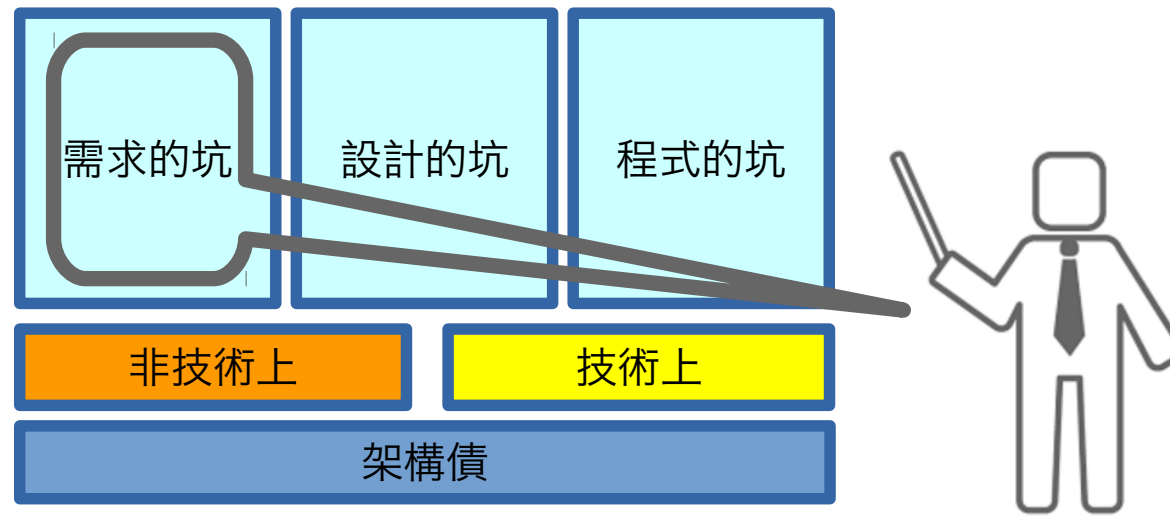


Conceptual debt

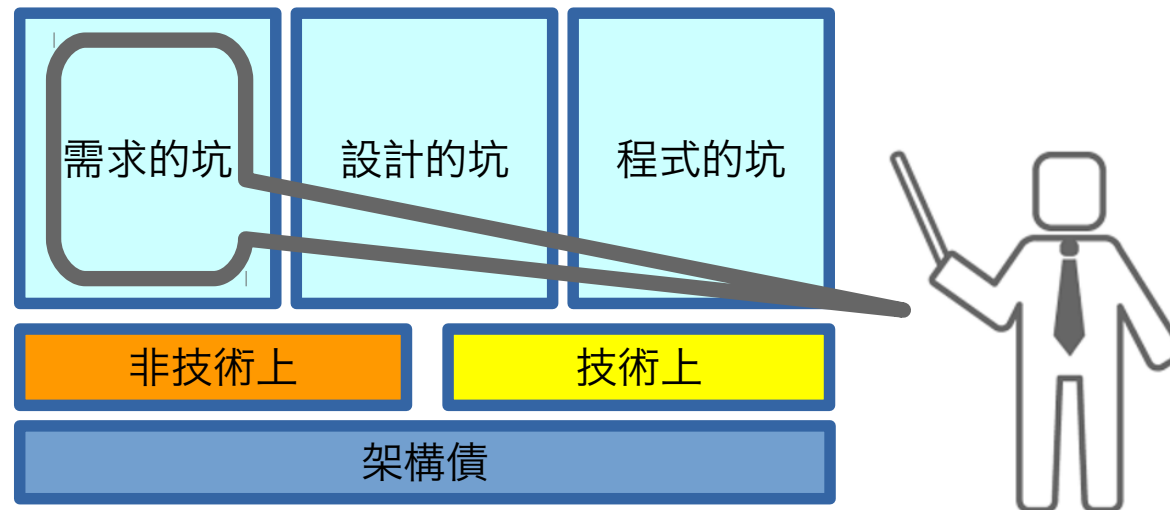
# Conceptual Integrity (概念完整性)

Agile, Scrum, Kanban, DevOps, TDD, BDD, ...





架構之始，在於需求



出了問題時



需求方



開發方

我以為你知道！

這本來就是要的，只是我忘了講！

我要的不是這樣！

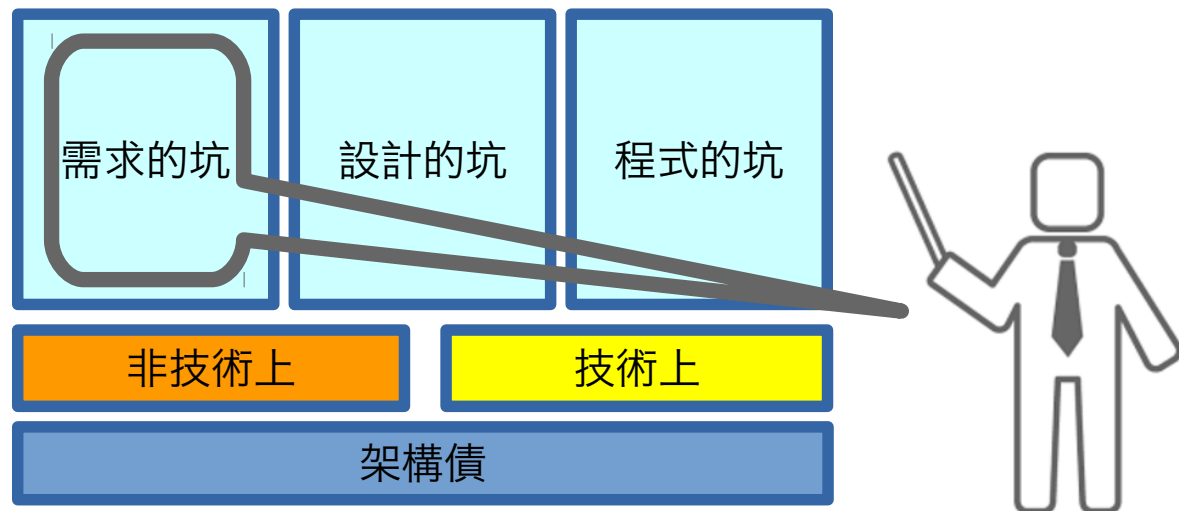
我要的你應該要替我想到，我又不懂技術

你沒跟我說過。

怪我囉。

可是需求書是這樣 / 需求書不明確

你 !!!



出了問題時



需求方



開發方

我以為你知道！

這本來就是要的，只是我忘了講！

我要的不是這樣！

我要的你應該要替我想到，我又不懂技術

你沒跟我說過。

怪我囉。

可是需求

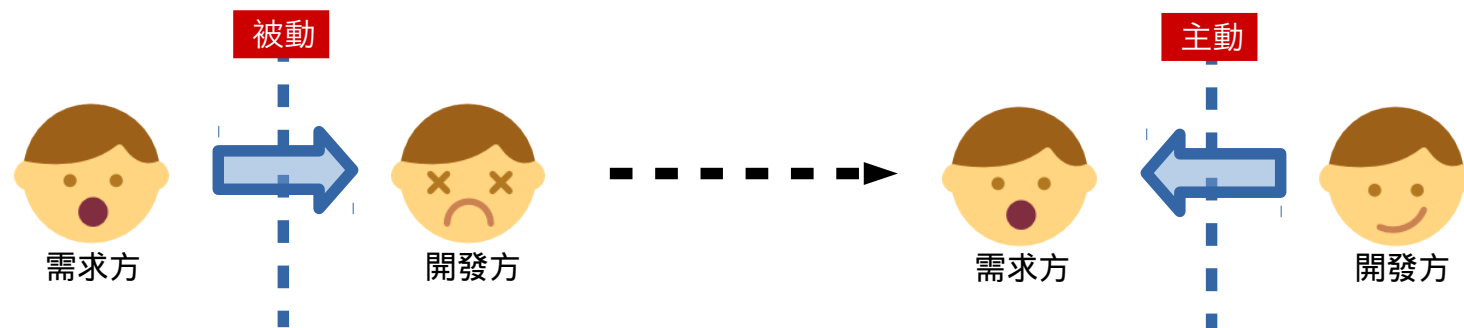
你 !!!

程式還是要趕，  
班還是照加。

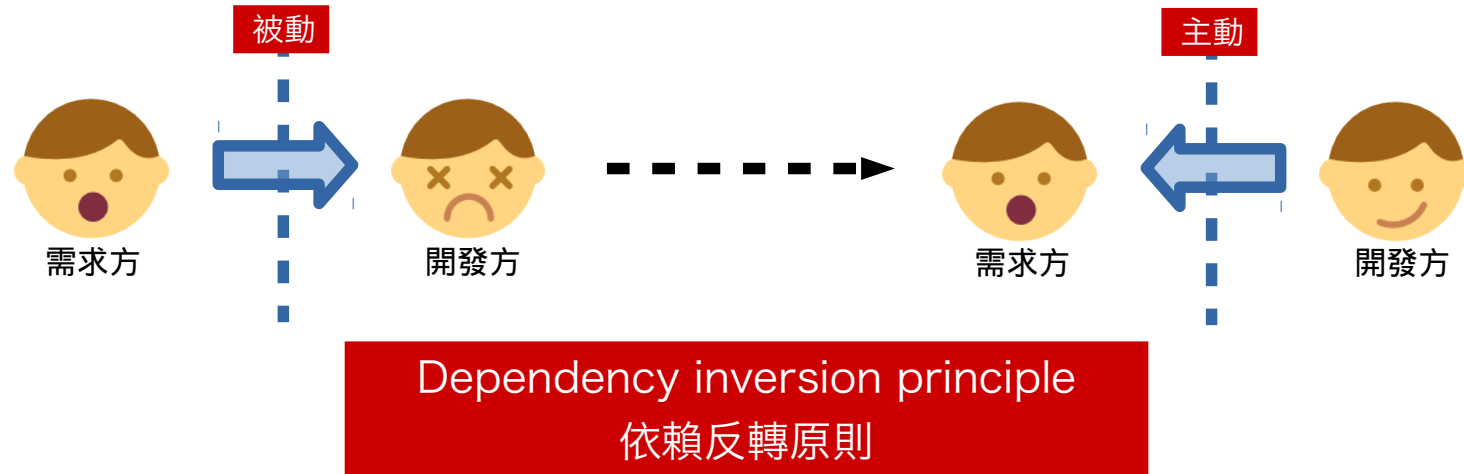
需求書不明確

需求永遠是不明確的  
但我們能做的是讓架構更具彈性

需求永遠是不明確的  
但我們能做的是讓架構更具彈性



需求永遠是不明確的  
但我們能做的是讓架構更具彈性



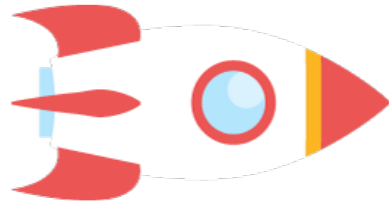
引用去年簡報

# 恰如其分的 MySQL 設計技巧

Ant

yftzeng@gmail.com

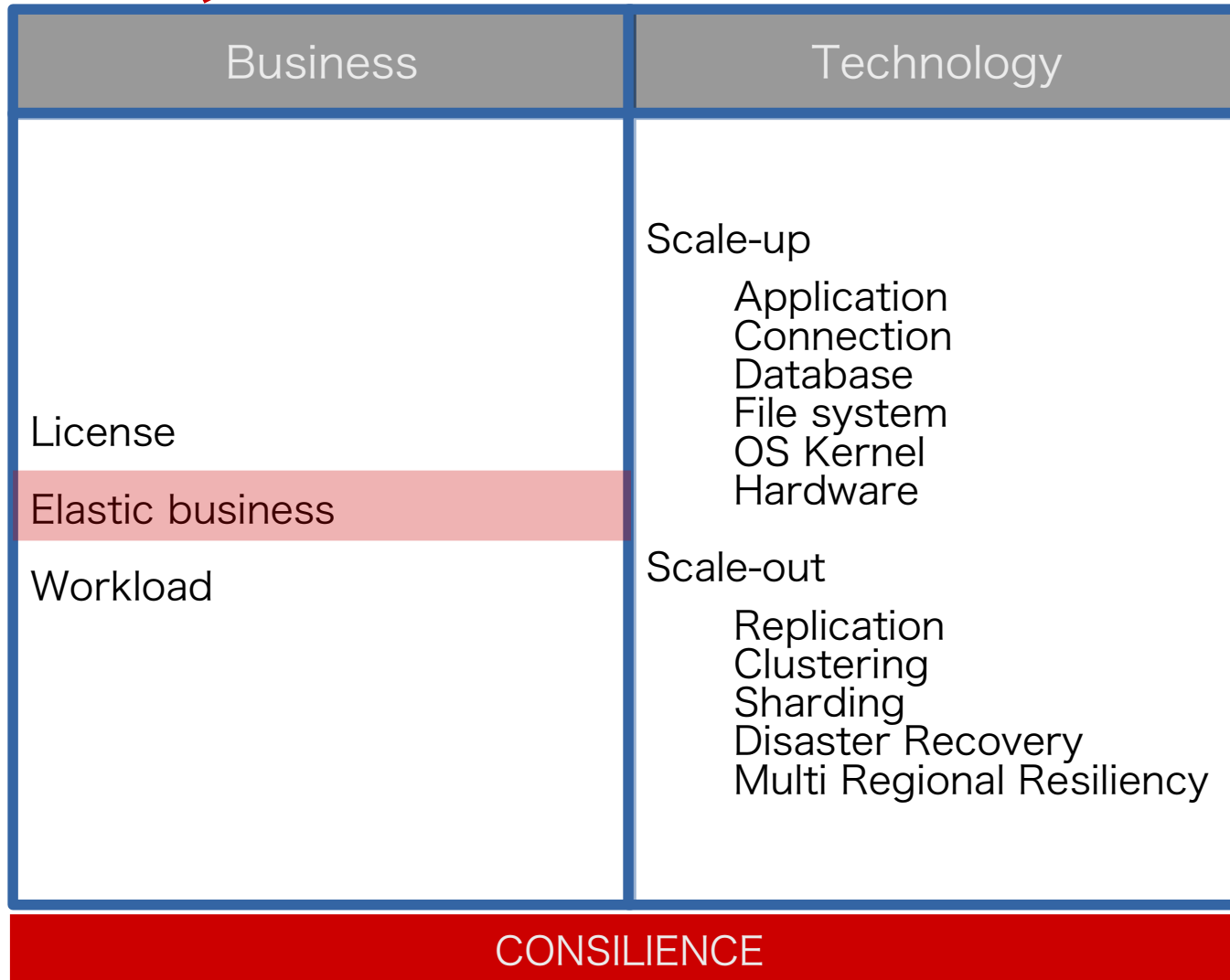
2016-08-24



Modern Web 2016

引用去年簡報

# Architecture



and more ...

引用去年簡報

## Elastic business

當業務需求變更程式設計時

預想但不過早調優

總工程師不該把每次新需求都認為獨立需求  
(多想二分鐘, 團隊可以不必自虐)

預想但不過早調優

Premature optimization is the root of all evil.  
過早最佳化是萬惡的根源

Premature optimization is the root of all evil.  
過早最佳化是萬惡的根源

手拿菜刀砍電線，一路火花帶閃電

Premature optimization is the root of all evil.

過早最佳化是萬惡的根源

手拿菜刀砍電線 一路火花帶閃電

We should forget about small efficiencies,  
say about 97% of the time:

Premature optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%.

We should forget about small efficiencies,  
say about 97% of the time:

Premature (micro) optimization is the root of all evil.

Yet we should not pass up our opportunities in that critical 3%.

Premature (micro) optimization is the root of all evil.

Do some { Architectural optimizations, Algorithms,  $\cdots$  } early.

Premature (micro) optimization is the root of all evil.

Do some { Architectural optimizations, Algorithms, ... } early.

程式與架構需懂得區分  
【架構】異動會牽扯組件邊界，影響巨大

Premature (micro) optimization is the root of all evil.

Do some { Architectural optimizations, Algorithms, ... } early.

程式與架構需懂得區分

【架構】異動會牽扯組件邊界，影響巨大

即未來需求變更時，屬程式異動，還是架構異動？

簡言之，如果需求發生異動，需花多久時間滿足？

引用去年簡報

# Elastic business

狀態

原表格設計

| id | name   | ... | is_deleted |
|----|--------|-----|------------|
| 1  | Apple  | ... | 0          |
| 2  | Banana | ... | 1          |

引用去年簡報

# Elastic business

狀態

新業務需要儲存「鎖定」狀態

| id | name   | ... | is_deleted |
|----|--------|-----|------------|
| 1  | Apple  | ... | 0          |
| 2  | Banana | ... | 1          |



| id | name   | ... | is_deleted | is_locked |
|----|--------|-----|------------|-----------|
| 1  | Apple  | ... | 0          | 1         |
| 2  | Banana | ... | 1          | 0         |

引用去年簡報

# Elastic business

狀態

其實若狀態是沒交集的，則可以合併

| id | name   | ... | is_deleted | is_locked |
|----|--------|-----|------------|-----------|
| 1  | Apple  | ... | 0          | 1         |
| 2  | Banana | ... | 1          | 0         |



| id | name   | ... | status |
|----|--------|-----|--------|
| 1  | Apple  | ... | 2      |
| 2  | Banana | ... | 0      |

{ 0: deleted, 1: enabled, 2: locked }

引用去年簡報

# Elastic business

標籤雲

原表格設計

| id | name   | tag1     |
|----|--------|----------|
| 1  | Apple  | admin    |
| 2  | Banana | reporter |
| 3  | Cherry | reporter |

```
SELECT * FROM {Table}  
WHERE tag1 = 'admin'
```

引用去年簡報

# Elastic business

標籤雲

新增標籤

| id | name   | tag1     | tag2       | tag3       |
|----|--------|----------|------------|------------|
| 1  | Apple  | admin    | reporter   | programmer |
| 2  | Banana | reporter | programmer | NULL       |
| 3  | Cherry | reporter | admin      | NULL       |

ALTER TABLE !!

```
SELECT * FROM {Table}
WHERE (tag1 = 'admin' OR tag2 = 'admin' OR tag3 = 'admin')
      AND (tag1 = 'reporter' OR tag2 = 'reporter' OR tag3 = 'reporter')
```

```
SELECT * FROM {Table}
WHERE 'admin' IN (tag1, tag2, tag3)
      AND 'reporter' IN (tag1, tag2, tag3)
```

引用去年簡報

# Elastic business

標籤雲

新增標籤 (另他法)

| id | name   | X | X | X |
|----|--------|---|---|---|
| 1  | Apple  | X | X | X |
| 2  | Banana | X | X | X |

Tag

| id  | tag        |
|-----|------------|
| 1   | admin      |
| 1   | reporter   |
| 1   | programmer |
| 2   | reporter   |
| ... | ...        |

```
SELECT * FROM {Table}
  INNER JOIN 'Tag' AS t1 USING (id)
  INNER JOIN 'Tag' AS t2 USING (id)
WHERE t1.tag = 'admin'
      AND t2.tag = 'reporter'
```

引用去年簡報

# Elastic business

標籤雲

或是 M:N

| id | name   |
|----|--------|
| 1  | Apple  |
| 2  | Banana |

| id | tag_id |
|----|--------|
| 1  | 1      |
| 1  | 2      |
| 1  | 3      |
| 2  | 2      |
| 2  | 3      |

| tag_id | name       |
|--------|------------|
| 1      | admin      |
| 2      | reporter   |
| 3      | programmer |

引用去年簡報

# Elastic business

廣告需求

第一天



營運有新的需求，受眾在  
**一天內**不要看到相同廣告。

瞭解，預計一個工作天。



引用去年簡報

# Elastic business

廣告需求

第二天



營運有新的需求，受眾在  
小時內不要看到相同廣告。

時間粒度不同

瞭解，預計二個工作天。



引用去年簡報

# Elastic business

廣告需求

第三天



營運有新的需求，受眾在  
一天內不要看到同類廣告。

目標粒度不同

瞭解，預計八個工作天。



不能一次講清楚嗎？

引用去年簡報

# Elastic business

廣告需求

預想



受眾在 **M** 時間內不要看到 **N** 廣告



該需求的延伸會是什麼？

M → 年 / 季 / 月 / 週 / 時 / 分 / 秒

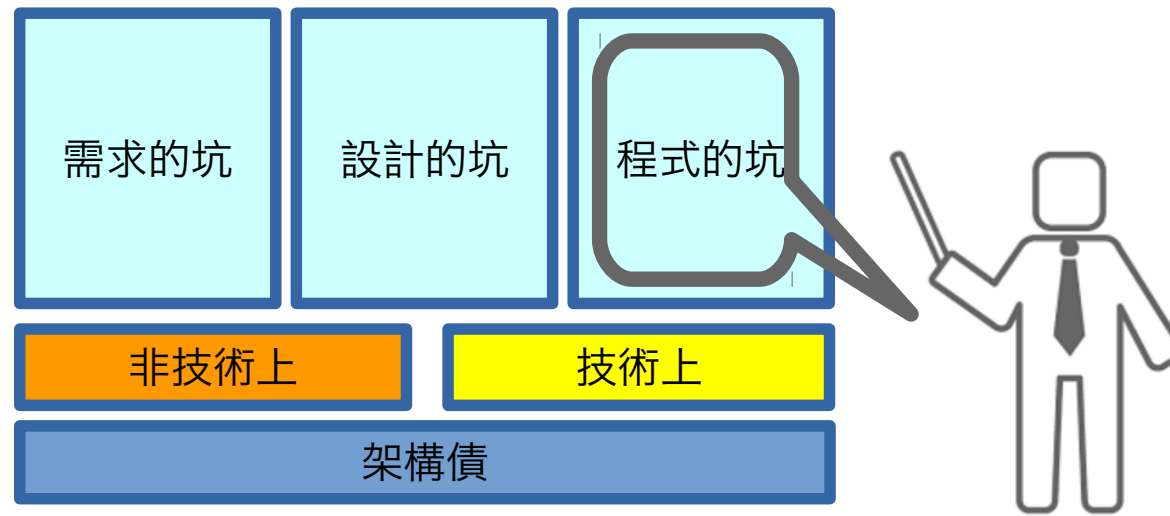
N → 相同 / 同類

看到的次數？ 1/2/3...

裝置有別？

區域？

廣告主屬性？



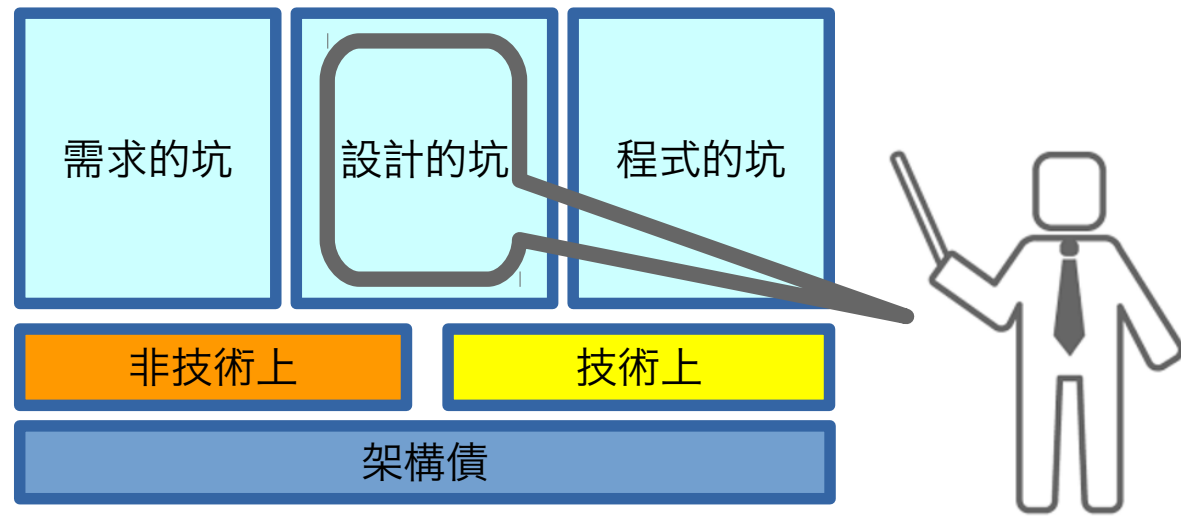
KISS(Keep it simple, stupid!)

DRY(Don't repeat yourself)

SOLID Principles

...

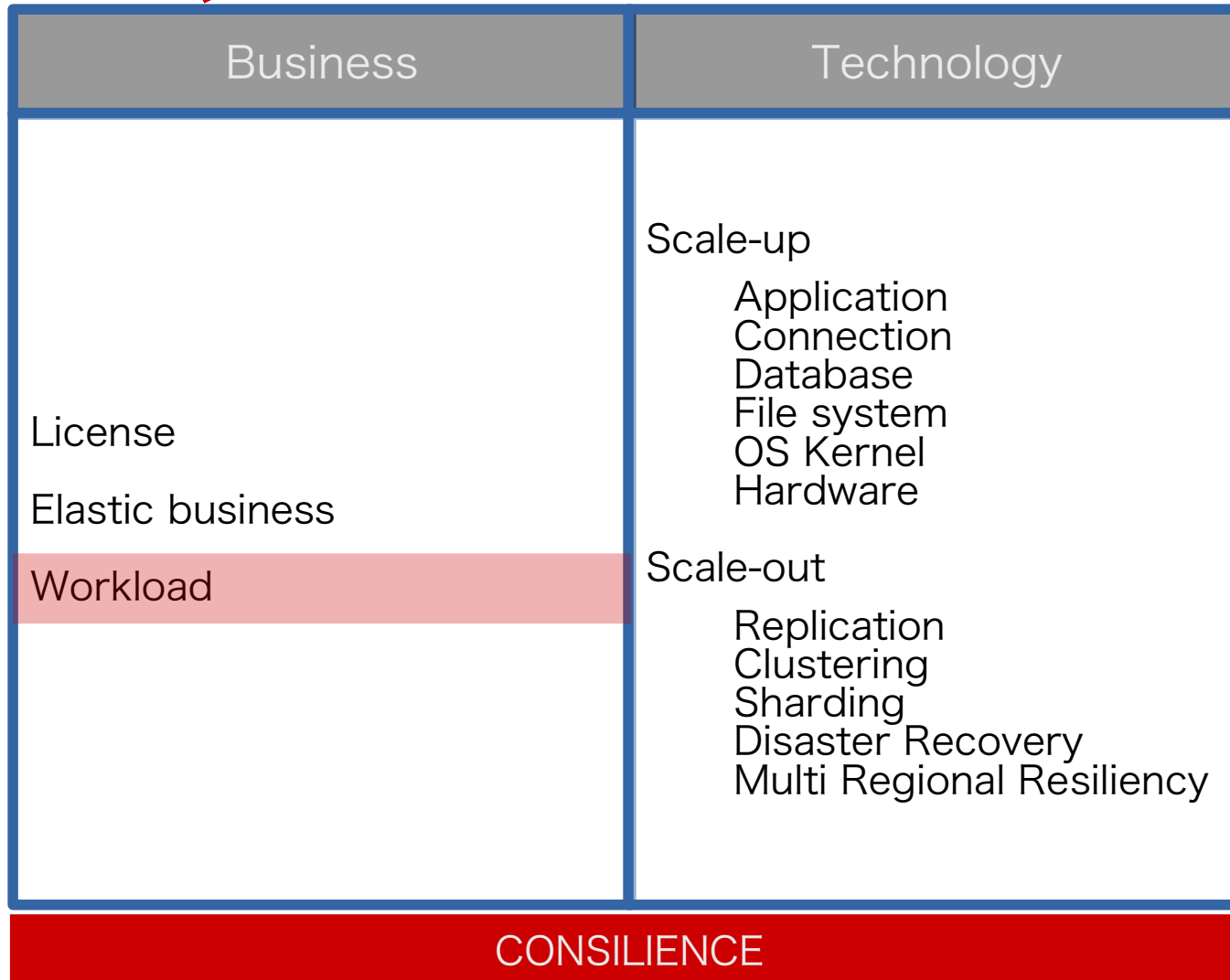
略過



設計：權衡需求與實現的藝術

引用去年簡報

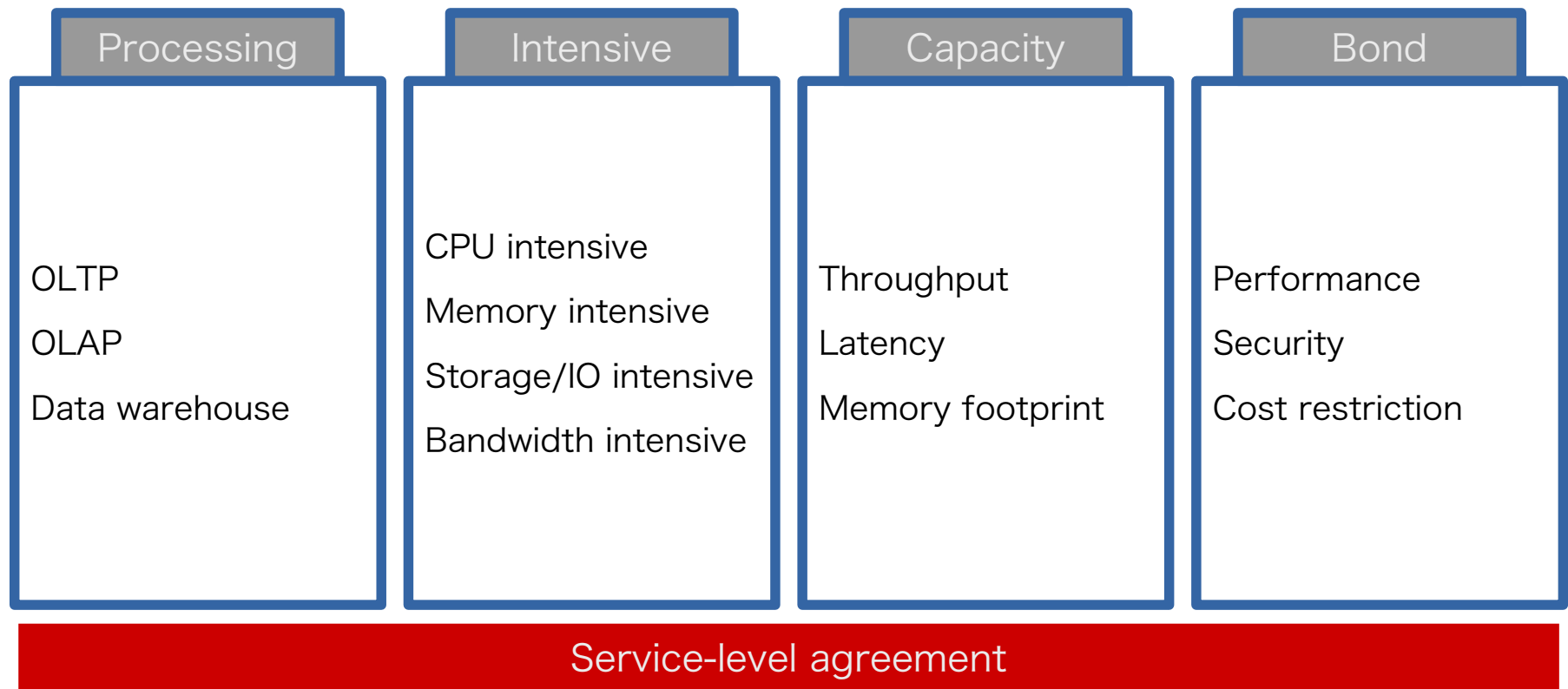
# Architecture



and more ...

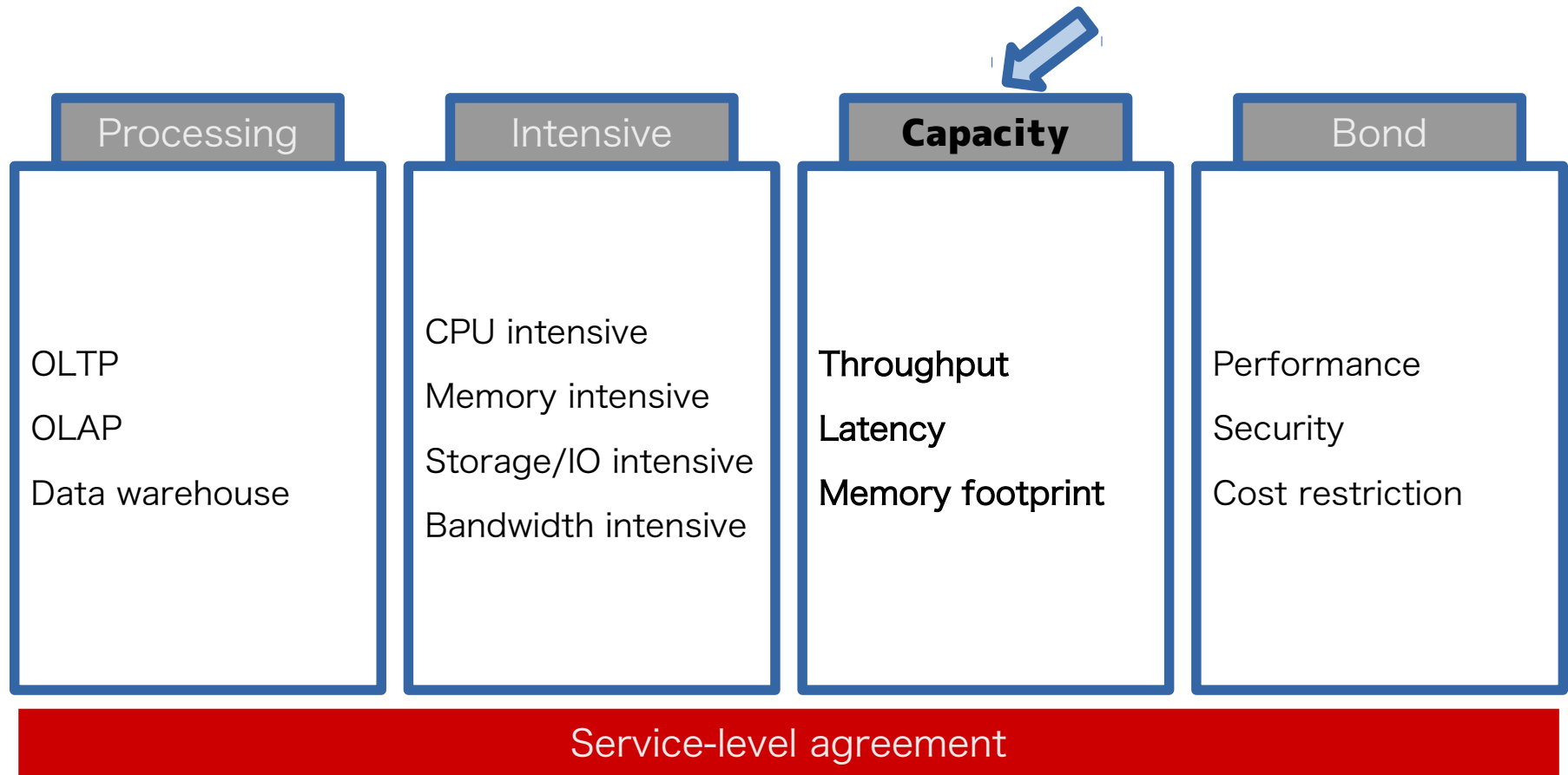
引用去年簡報

# Workload



引用去年簡報

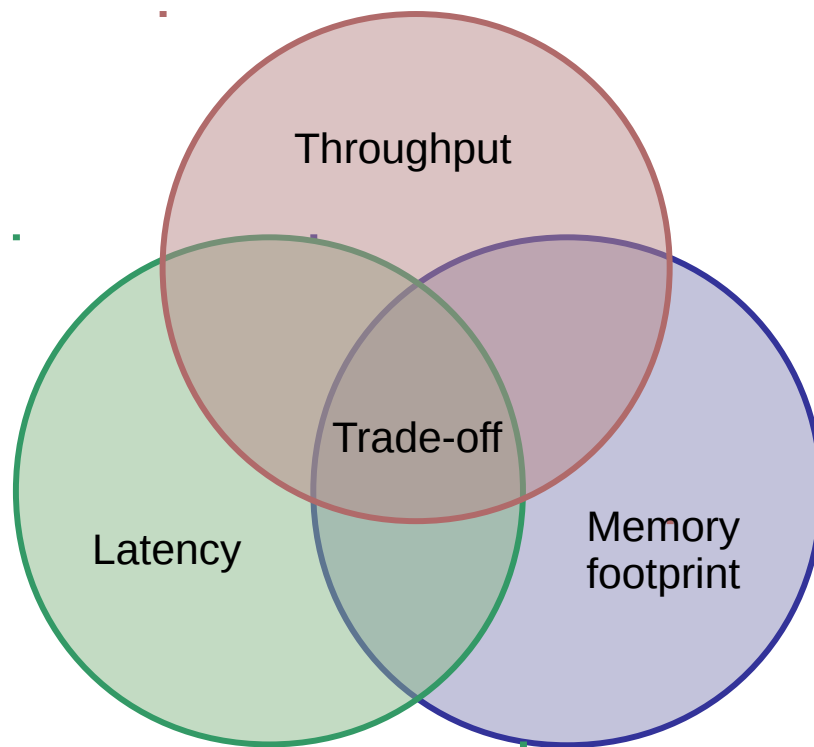
# Workload



引用去年簡報

# Workload

Capacity



引用去年簡報

# Workload

Capacity



High throughput



Low latency

引用去年簡報

# Workload Capacity



重點在於找出需求的【硬限制】

High throughput

Low latency

遇到瓶頸還不是最慘，慘得是過了瓶頸，還有瓶蓋

~ Anonymous ~

# 千萬人同時在線

電子商務、線上出版媒體

## 低延遲回應

廣告平台 RTB(30ms)

高頻交易 (0.5~3ms)

多人線上遊戲 (30fps: 100~150ms, 60fps: 60~70ms)

醫療等關鍵設備

# Workload

Capacity



Low latency

非指 PHP 無法做到低延遲 (例如 30ms)，而是  
我們願意 (只能) 花多少資源 (資金硬限制)，以及  
我們手中工程師的數量與實力 (人力硬限制)。

很多事情 (例如 latency) 不是靠 Scale Out 能解決

高手人數? 訓練成本?

# Workload

Capacity



Low latency

此時， PHP 相較 C/C++/Go 通常會更辛苦。

伺服器規格需求較高？ 較多？

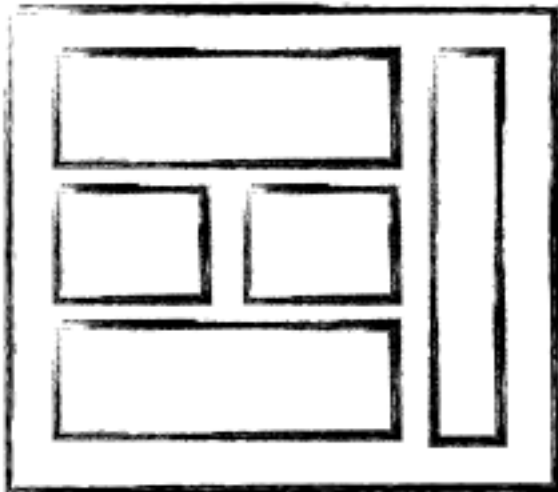
案主：不好意思，我只能給你兩台一般的伺服器

工程師技能要求較高？

PHP Extension？ Swoole？ 奇技淫巧？ 未來徵人的訓練成本？

徵聘更強的工程師？

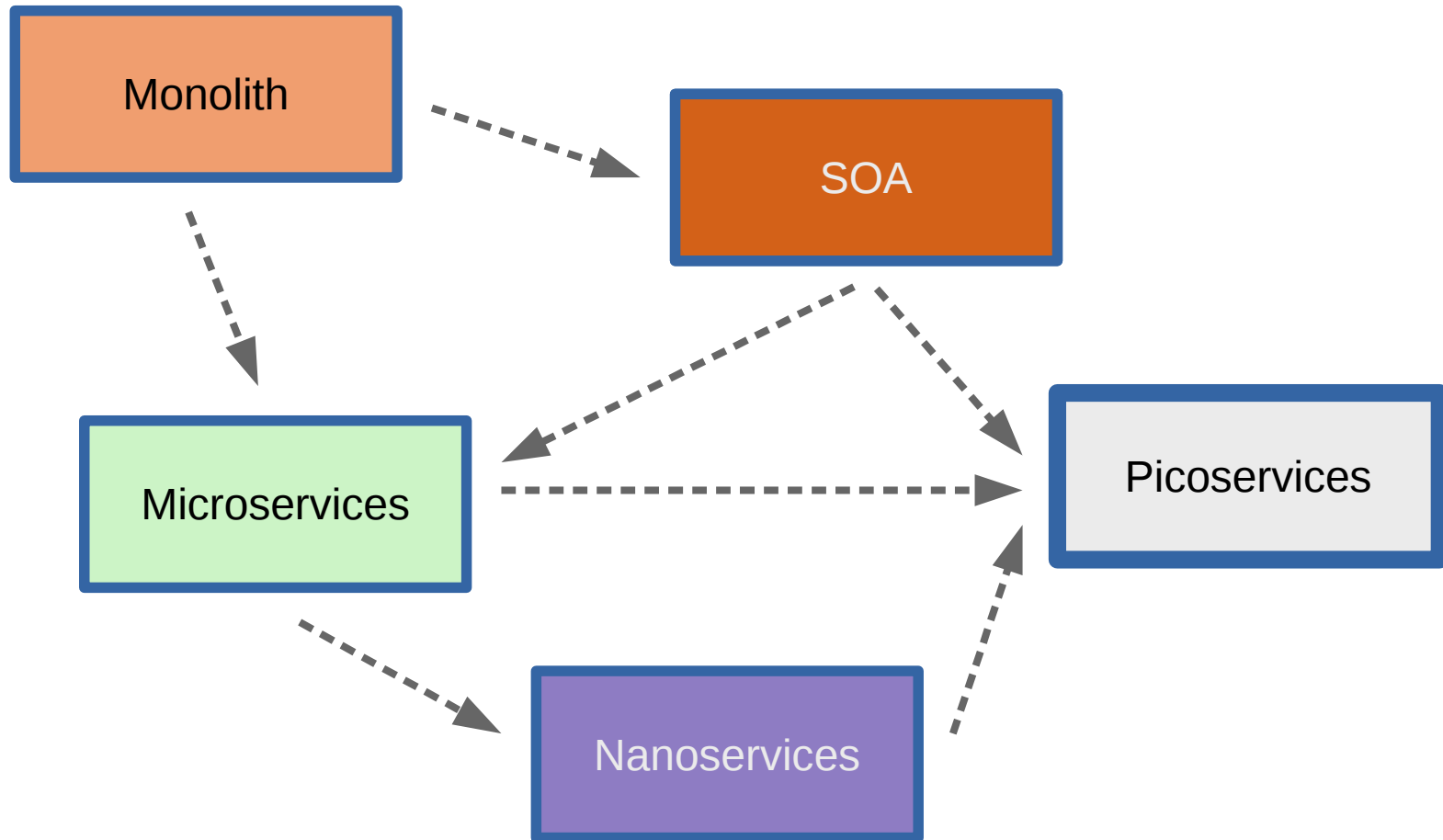
高手為什麼要來？ 有無足夠的產品吸引力？ 足夠的給薪條件？

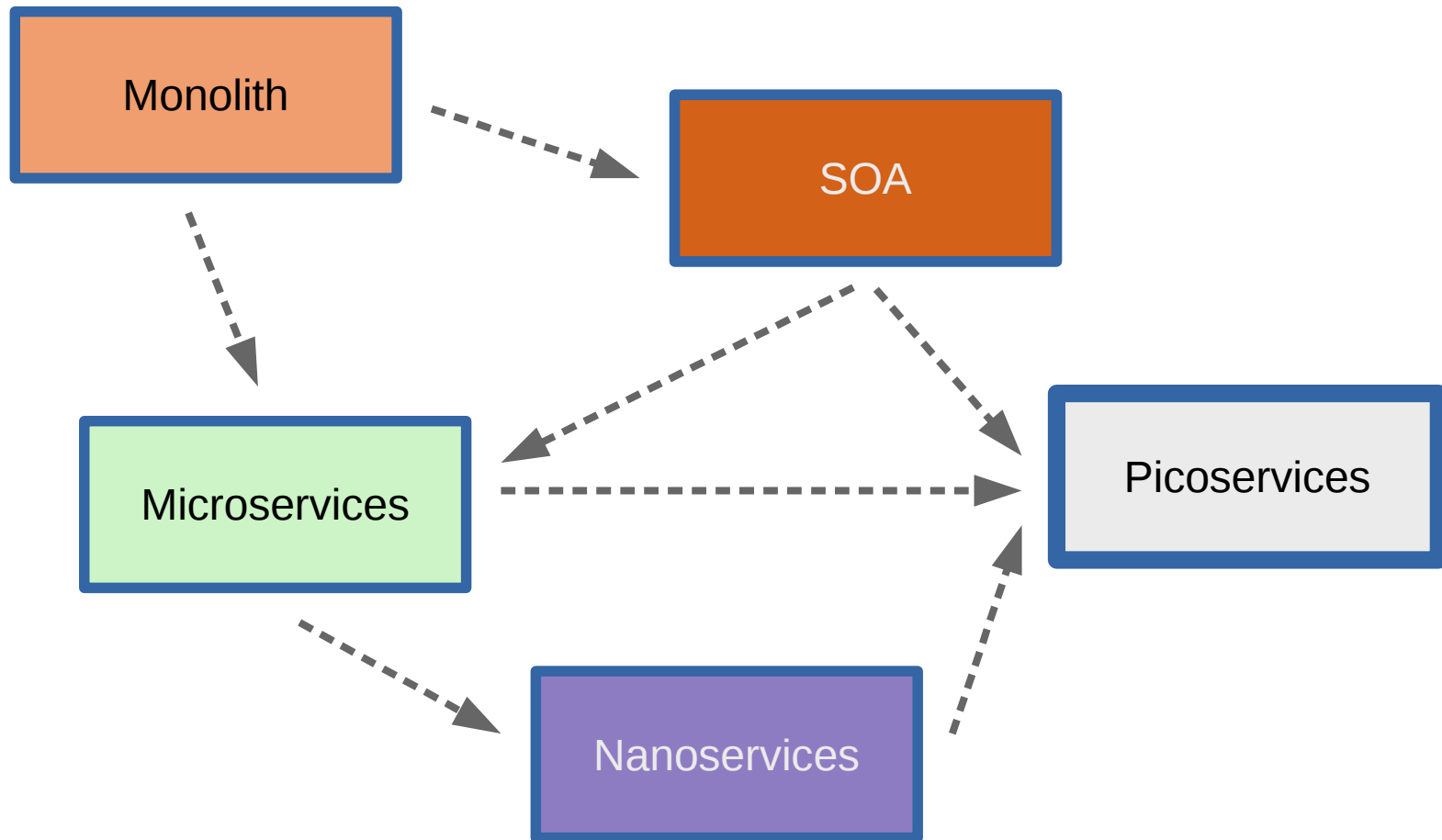


MONOLITHIC/LAYERED



MICRO SERVICES





有沒有能力駕馭微服務、奈米服務、微微服務？

# MonolithFirst

## 《單體》優先的技術選型

~ Martin Fowler (ThoughtWorks 的首席科學家) ~

對於《微服務》，他注意到了這兩種問題，

- ❶ 幾乎所有成功的微服務案例，都是從過大的《單體》拆分的。
- ❷ 幾乎所有他見過一開始就投入《微服務》的案例，最後都遭遇嚴重麻煩。

# MonolithFirst

網友提出不同見解

~ krisdol ~

Fowler 遇到的案例都是出問題的公司。ThoughtWorks 是一家顧問公司，也就是說，他們會在某個公司出現問題時提供幫助。

簡言之，Fowler 所見都是不對的組織採用了錯誤的架構，也就是他會見到一開始採用《微服務》而失敗的公司，但不會見到成功的。

# MonolithFirst

網友提出不同見解

~ room271 ~

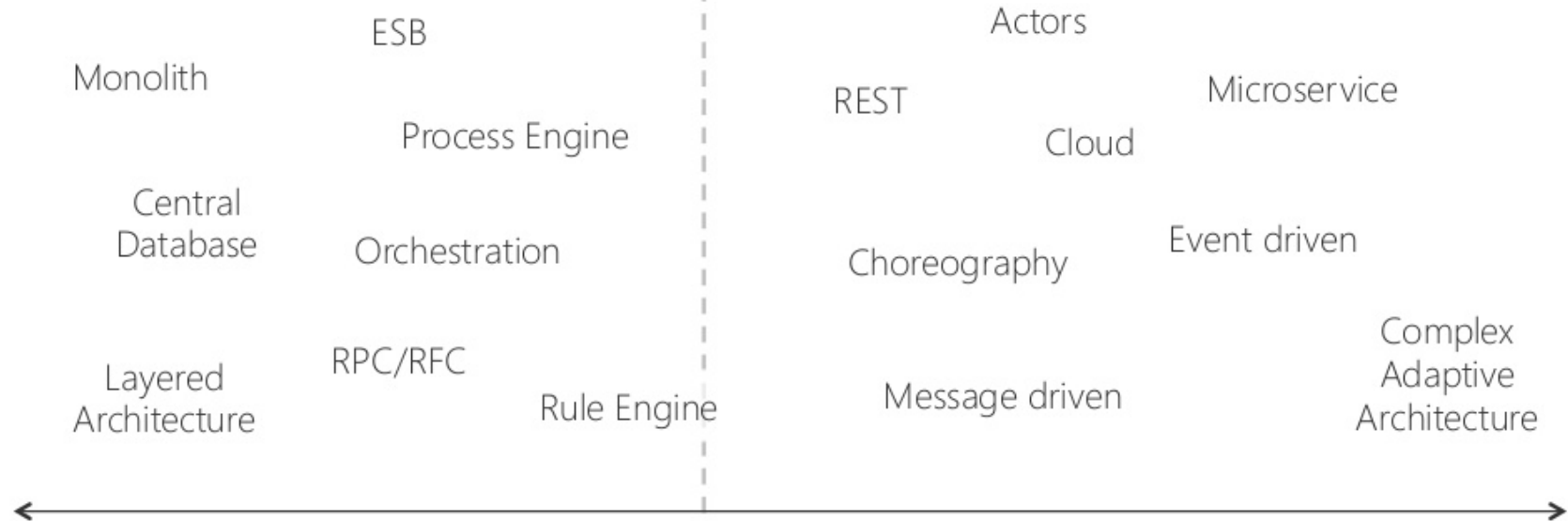
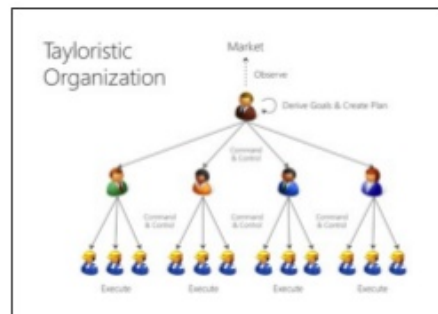
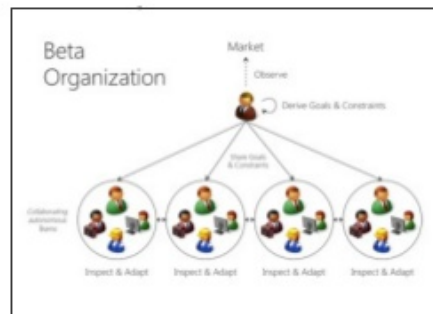
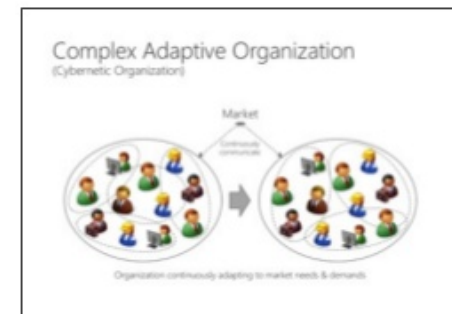
當你是顧問時，一開始很容易提倡採用《單體》，因為你會在《單體》出現問題前，就已經完成專案離開了。

已有兩年《微服務》經驗的他，提出了兩個見解：

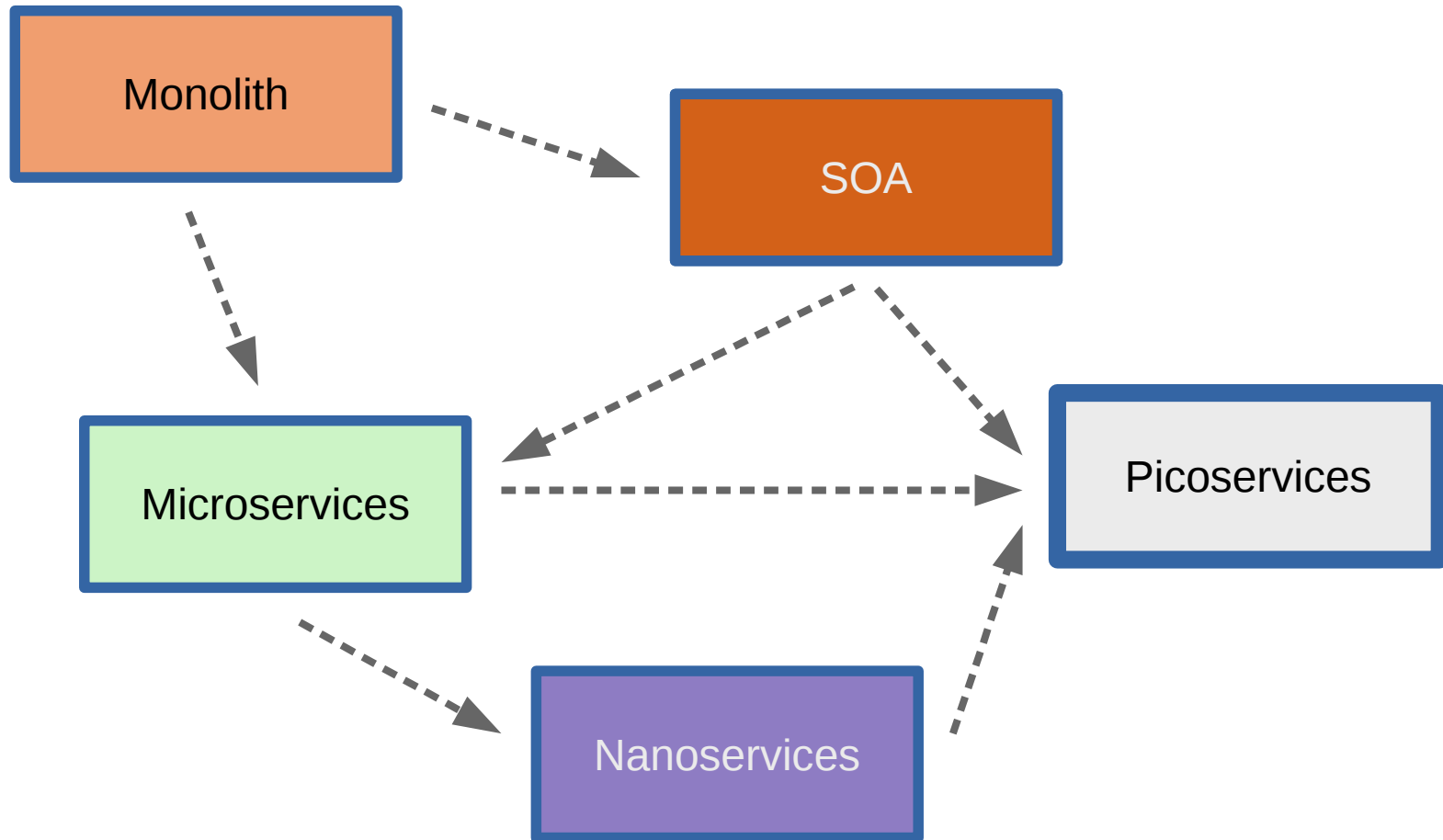
- ❶ 當公司或團隊很小時，除非你們技藝超群，否則不建議採用《微服務》。
- ❷ 當公司或團隊很大時，採用《微服務》有助於解耦組織並促進敏捷。

# Industrial

# Post-Industrial



組織會影響架構



**救世主？**

The Lambda logo is a stylized Greek letter λ (lambda) composed of several horizontal segments in different colors: green at the top, followed by orange, dark blue, light blue, yellow, red, and purple at the bottom. It is positioned behind the text "Serverless Architecture".

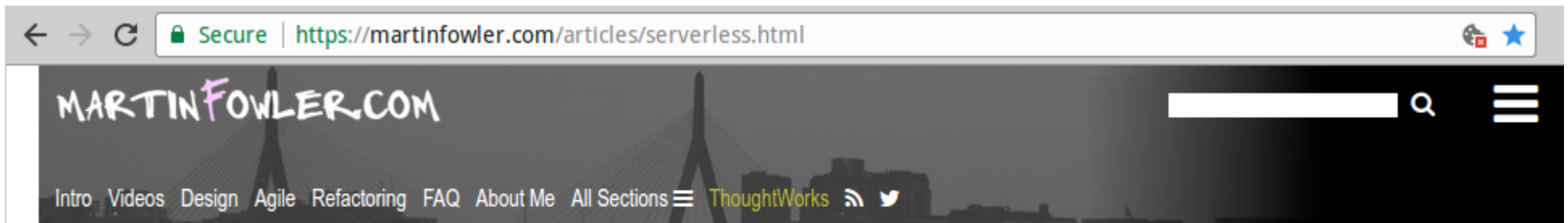
# Serverless Architecture

# Serverless

Serverless  $\neq$  Architectureless

# Serverless

Serverless  $\neq$  Architectureless

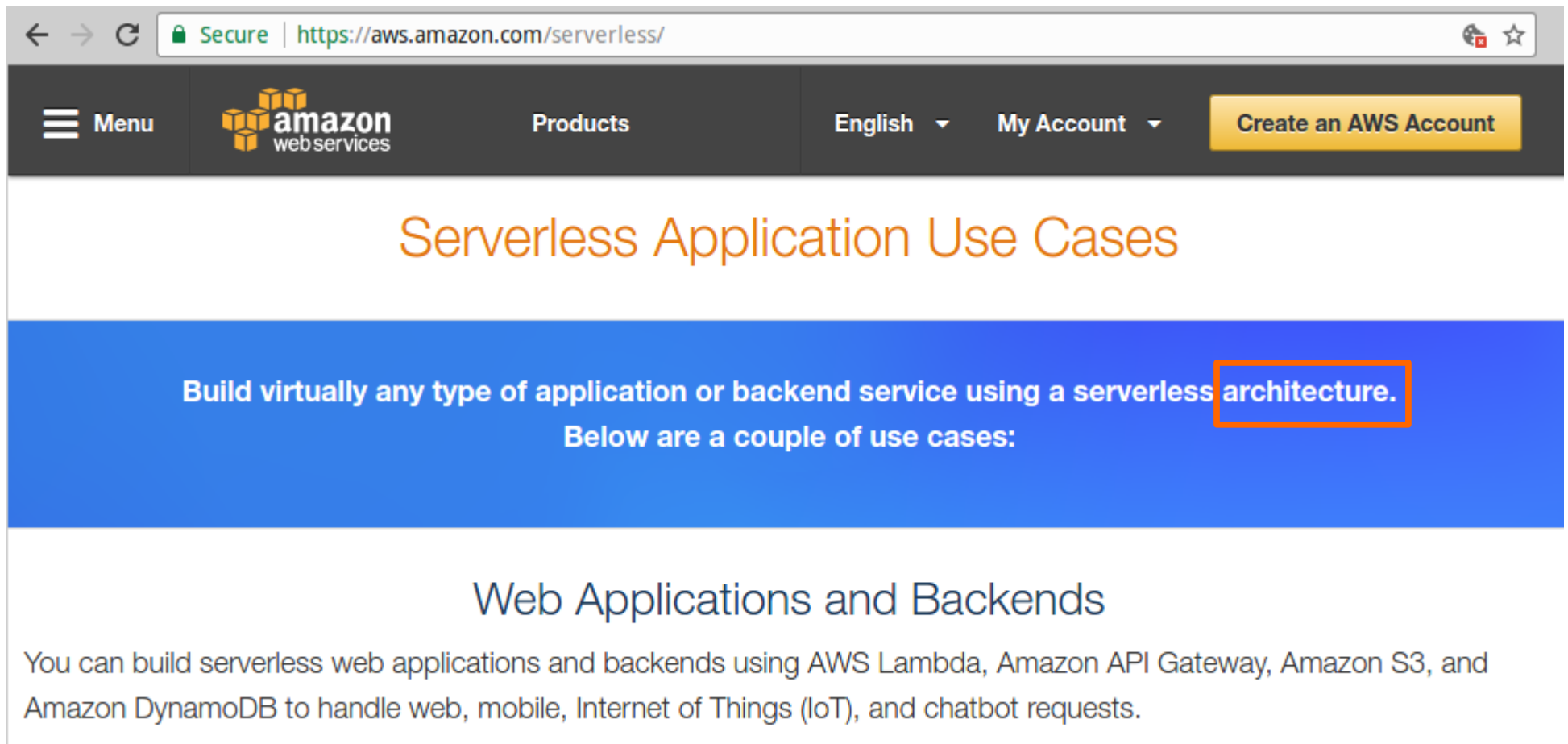


## Serverless Architectures

*Serverless architectures refer to applications that significantly depend on third-party services (known as Backend as a Service or "BaaS") or on custom code that's run in ephemeral containers (Function as a Service or "FaaS"), the best known vendor host of which currently is AWS Lambda. By using these ideas, and by moving much behavior to the front end, such architectures remove the need for the traditional 'always on' server system sitting behind an application. Depending on the circumstances, such systems can significantly reduce operational cost and complexity at a cost of vendor dependencies and (at the moment) immaturity of supporting services.*

# Serverless

Serverless  $\neq$  Architectureless



The screenshot shows the AWS Serverless landing page. At the top is a navigation bar with a menu icon, the Amazon Web Services logo, 'Products', language and account dropdowns, and a 'Create an AWS Account' button. The main heading is 'Serverless Application Use Cases'. Below it is a blue banner with the text 'Build virtually any type of application or backend service using a serverless architecture.' where 'architecture.' is highlighted with an orange box, followed by 'Below are a couple of use cases:'. The section 'Web Applications and Backends' follows, with a paragraph explaining that serverless web applications and backends can be built using AWS Lambda, Amazon API Gateway, Amazon S3, and Amazon DynamoDB to handle various requests.

Secure | <https://aws.amazon.com/serverless/>

Menu  Products English ▾ My Account ▾ [Create an AWS Account](#)

## Serverless Application Use Cases

**Build virtually any type of application or backend service using a serverless architecture.**

Below are a couple of use cases:

### Web Applications and Backends

You can build serverless web applications and backends using AWS Lambda, Amazon API Gateway, Amazon S3, and Amazon DynamoDB to handle web, mobile, Internet of Things (IoT), and chatbot requests.

- ① 架構先決
- ② 沒有完美的架構，只有最適的架構
- ③ 架構是演進的，預想但不過早調優

# 犧牲

Sacrificial Architecture (犧牲的架構)

~ Martin Fowler ~

即使捨棄現有架構，重構組件新架構，也未必是一件失敗的事

重點是【為什麼】、【新架構怎麼執行】及【何時執行】

簡言之，如果清楚明白打掉重練比較好，就勇敢執行吧！

# 犧牲

Sacrificial Architecture (犧牲的架構)

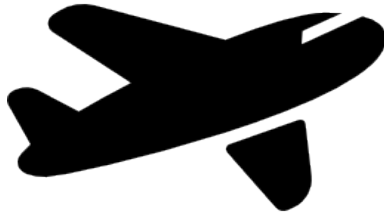
~ Martin Fowler ~

即使捨棄現有架構，重構組件新架構，也未必是一件失敗的事

重點是【為什麼】、【新架構怎麼執行】及【何時執行】

簡言之，如果清楚明白打掉重練比較好，就勇敢執行吧！

但免不了的，還是要面臨這些問題



技術難度高。

但迅速成長的公司，仍然有很多需求待開發。  
同時還要處理累積的技術債，很可能因債築債而嚴重惡化。

空中換引擎

---



技術難度低。

但商業模式是否能夠允許新功能推進緩慢或無進展？

落地換引擎

---



資源投入巨大。

不是現行團隊拆分戰力，就是要另招一批團隊開發新引擎。  
但若問題本質不解決，新引擎未來也會遭遇一樣的問題。

空中小修  
地面重建

共勉之



yftzeng@gmail.com



<https://www.facebook.com/yftzeng.tw>