修好真的比較簡單嗎？複雜度轉移與重構悖論

v0.1 / 2026-08-01 / Neo.K

https://thisoneisneok.com/html/papers/ois-07.html

系列：《表觀完好系統：從軟體屎山、補償性完整到動態架構治理》

篇次：07 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

軟體工程經常把重構、技術債償還、服務拆分與架構清理描述成「降低複雜度」的活動。然而，複雜度並不是一個只能存在於程式碼中的單一標量。當一個大型函式被拆成多個服務、當重複邏輯被集中成共同抽象、當人工流程被自動化、當 legacy adapter 被 framework 取代時，某些局部複雜度確實可能下降，但新的複雜度也可能出現在網路通訊、部署、故障模式、配置、資料一致性、共享基礎設施、組織協調、觀測、驗證與認知負荷中。

2025 年 ICSA 的 architecture technical debt 生命周期研究提供了一個重要實證例子：研究觀察到 ATD repayment 後 FAN-IN 平均增加 57.5%、FAN-OUT 增加 26.7%，顯示償債可以改善某些短期品質，同時使依賴更集中並增加另一種架構複雜度。Microservices 的既有工程經驗也長期指出，較強的模組邊界與獨立部署能力，是以 distribution、eventual consistency 與 operational complexity 等成本交換而來。

本文因此提出「複雜度轉移（Complexity Transfer）」與「重構悖論（Refactoring Paradox）」：一次成功重構可能顯著改善目標區域，卻不必然降低系統的總複雜度；它更可能改變複雜度的位置、型態、集中程度、可見性、承擔者與可自動化程度。

形式上：

> 【Repair
> nRightarrow
> Total Complexity Reduction】

更合理的模型為：

> 【C_(t+1)
> =
> T
> (
> C⊠,
> R,
> E
>)】

其中 C 是多維複雜度向量，R 是重構／架構變換，E 是環境，而 T 描述複雜度如何被重新分布。

本文建立程式碼、依賴、分布式、操作、資料、配置、認知、組織、驗證與 AI 監督等複雜度維度，並提出「複雜度守恆」不是嚴格物理定律，但「複雜度不應被假定會因抽象或重構而消失」是一項重要工程原則。真正好的重構，不一定讓所有複雜度變少，而是把不可治理、不可觀測、不可替換的複雜度，轉化成更集中、更顯式、更可驗證、更可自動化或更符合責任邊界的複雜度。

本文最後將此命題接回 Dynamic MSSP/FPL：架構治理不能只比較重構前後的 code smell 或 dependency count，而應追蹤「複雜度流」——複雜度從哪裡消失、又在哪裡重新出現，以及新的承擔主體是否真的更適合處理它。

關鍵詞：複雜度轉移、重構悖論、architecture technical debt、microservices、operational complexity、leaky abstraction、AI coding、Dynamic MSSP、FPL、軟體結構動力學

SECTION

1. 「修乾淨了」究竟代表什麼？

軟體重構之後，我們常會說：

> 這裡乾淨多了。

這句話通常有真實基礎。

例如：

- 5000 行函式拆成 20 個模組；
- 重複程式碼被集中；
- shared database 被拆開；
- legacy branch 被清除；
- package cycle 被消除；
- manual process 被自動化；
- 巨大 monolith 被拆成服務。

在局部視角：

> C_(local)(t+1)
> <
> C_(local)(t)

完全可能。

問題是：

> 這是否意味整個系統真的變簡單？

答案不是必然。

因為「局部簡單」可以透過把責任推到其他地方取得。

SECTION

2. 2025 ICSA：償還技術債後，依賴反而更集中

2025 年 ICSA 的研究 Tracing the Lifecycle of Architecture Technical Debt in Software Systems: A Dependency Approach 對 architecture technical debt 的生命周期與依賴變化進行分析。

其中最值得本文注意的結果是：

- ATD repayment 後 FAN-IN 平均增加 57.5%；
- FAN-OUT 平均增加 26.7%。

  研究者指出，這代表償還 architecture technical debt 可以改善短期品質，同時使依賴向某些位置集中，增加另一種架構複雜度。

  這個結果非常重要，因為它反駁了一個過度簡化的直覺：

> Debt Repaid
> ⇒
> Complexity Lower

更可能是：

> Debt Repaid
> ⇒
> Dependency Topology Changed

甚至：

> Local Disorder Reduced
> →
> Central Dependency Increased

也就是：

> 修復不是把複雜度消掉，而可能是把分散複雜度重新集中。

SECTION

3. 重構悖論

本文因此定義：

SECTION

重構悖論

SECTION

Refactoring Paradox

> 一次重構可以成功改善其直接目標，同時使系統在其他維度產生新的複雜度或負擔。

形式上：

> $\Delta C\boxtimes < 0$

不代表：

> $\Sigma \boxtimes \Delta C\boxtimes < 0$

其中 $C\boxtimes$ 是某一類複雜度。

例如：

> C_(code)downarrow

但：

> C_(operations)uparrow

仍然可能。

這不是「重構失敗」。

相反地，它可能是完全合理的工程交換。

真正錯誤的是：

> 只測量下降的那一維，然後宣稱複雜度消失了。

SECTION

4. 複雜度不是一個數，而是一個向量

本文將系統複雜度表示為：

$$
C(t) = [C_c, C_d, C_\square, C_o, C_s, C_f, C_h, C_g, C_v, C_a]_\square
$$

其中：

- C_c：Code Complexity；
- C_d：Dependency Complexity；
- $C_\square$：Network / Distribution Complexity；
- C_o：Operational Complexity；
- C_s：State / Data Complexity；

- C_f：Configuration Complexity；
- C_h：Human / Cognitive Complexity；
- C_g：Governance / Organizational Complexity；
- C_v：Verification Complexity；
- C_a：AI / Agent Supervisory Complexity。

  此模型並不主張十個維度已經是最終分類。

  它的目的只是強調：

> 【C_(system)
> ≠
> C_(code-only)】

SECTION

5. 第一種轉移：Code → Dependency

假設原本有：

A
A'
A''
A'''
四份重複邏輯。

重構後：

A ─┐
A' ├→ SharedService
A'' ├→
A'''┘
重複下降：

> C_cdownarrow

但 SharedService 的：

> FAN-INuparrow

此時：

> C_d

可能上升。

這是一個非常典型的：

SECTION

Duplication-to-Centralization Transfer

也就是：

> Duplication Complexity
> →
> Central Dependency Complexity

新的架構可能更好。

但代價是：

- shared service 變成 blast-radius center；
- 修改需要更強 backward compatibility；
- outage 影響更多 consumer；
- ownership 變得更重要。

所以：

> 集中不是免費的簡化。

SECTION

6. 第二種轉移：Monolith → Distribution

Microservices 是複雜度轉移最清楚的案例之一。

Martin Fowler 對 microservice trade-offs 的整理指出，其優點包括：

- stronger module boundaries；
- independent deployment；
- technology diversity。

  但同時帶來：

- distribution complexity；
- eventual consistency；
- operational complexity。

  Microsoft 的 microservice architecture 指南同樣指出，分散應用增加：

- service communication；
- testing；
- exception handling；
- latency；
- deployment；
- distributed transaction；

  等複雜度。

  因此：

> C_(module-boundary)downarrow

可能同時：

> C_(distribution)uparrow

> C_(operations)uparrow

> C_(state consistency)uparrow

所以 microservices 不是：

> Complex System
> →
> Many Simple Systems

而更接近：

> In-process Complexity
> →
> Inter-service Complexity

SECTION

7. 「Big Ball of Microservice Mud」

這個問題甚至可能變得更糟。

如果 service boundary 選錯：

Service A
↔ Service B
↔ Service C
↔ Service D

而且需要大量同步 RPC、共享資料與協調部署，

那麼原本：

> C_(monolith)

沒有真正消失，

反而再加上：

> C_(network)
> +
> C_(deployment)
> +
> C_(distributed debugging)

得到：

> C_(distributed mud)
> >
> C_(original)

Fowler 也特別指出，microservices 可以形成「Big Ball of Microservice Mud」，而錯誤的服務邊界會讓 distribution 的成本更嚴重。

因此：

> Decomposition
> nRightarrow
> Decoupling

拆開部署單位，不代表真正解耦。

SECTION

8. 第三種轉移：Implementation → Framework

Framework 的核心價值就是隱藏重複機制。

例如：

routing

authentication
serialization
retry
dependency injection
ORM

如果每個專案都自己寫：

> C_(local implementation)

很高。

使用 framework 後：

> C_(local implementation)downarrow

但整體並不是沒有複雜度。

它被推進：

> C_(framework)

與：

> C_(framework understanding)

中。

這就是 leaky abstraction 類型問題的核心直覺：

> abstraction 可以讓日常使用更簡單，但在邊界情況、效能問題與故障中，底層複雜度仍可能重新浮現。

因此：

> Hidden Complexity
> ≠
> Removed Complexity

更精確是：

> Complexity Visibility
> downarrow

而不是一定：

> Complexity Existence
> downarrow

SECTION

9. 抽象的真正價值：重新分配認知，而不是消滅世界

這並不降低 abstraction 的價值。

恰恰相反。

好的 abstraction 很重要，因為它讓大部分使用者在大部分時間不需要處理底層細節。

因此它降低：

> $C_h^{(common path)}$

即日常認知複雜度。

但在 failure／edge case：

> $C_h^{(exception path)}$

可能重新上升。

所以抽象的真正價值不是：

> 底層不複雜了。
>
> 而是：

> 把需要處理複雜度的人數、頻率與情境縮小。

這是非常重要的複雜度治理思想。

SECTION

10. 第四種轉移：Runtime → Configuration

許多平台工程會試圖把：

hardcoded behavior
改成：

configuration
於是：

> C_(code)downarrow

但：

> C_(config)uparrow

例如：

```
routing:
 rules:
  - ...
feature_flags:
 - ...
policy:
 - ...
workflow:
 - ...
```

當設定越來越強大：

> Configuration
> →
> Programming Language

幾乎是自然結果。

此時：

- config dependency；
- versioning；
- validation；
- rollback；
- environment drift；
- permission；

  都成為新問題。

  因此：

> Code Simplification
> →
> Configuration System Complexity

完全可能。

SECTION

11. 第五種轉移：Code → Data

某些複雜邏輯會被改成：

lookup table
rules database
metadata
schema
這可以減少大量 condition。

但代價是：

> C_(data semantics)uparrow

系統不再只需要理解：

> code 在做什麼。

還要理解：

> 目前資料裡有哪些規則。

如果 rule table 缺乏：

- version；
- provenance；
- validation；
- ownership；
- migration；

就可能形成：

SECTION

Data-Encoded Architecture

也就是架構邏輯從 code 移到 data。

它仍然是複雜度。

只是靜態程式掃描看不到。

SECTION

12. 第六種轉移：Developer → Operations

microservices、serverless、Kubernetes、managed platform 常降低開發者某些負擔。

例如：

> C_(deployment logic in app)downarrow

但 operations 必須處理：

- orchestration；
- observability；
- service discovery；
- autoscaling；
- rollout；
- secrets；
- incident response；
- distributed tracing。

所以：

> C_(developer)
> →
> C_(operations)

可能成立。

O’Reilly 曾直接用「microservices shift complexity」來描述：複雜度被從軟體設計／開發轉移到更容易自動化的 operations。

這其實提供一個更成熟的判斷：

> Complexity Transfer 不一定是壞事。

如果轉移目的地：

> D□

比原目的地：

> D⊠

更適合自動化、監控與專業化，

那麼：

> C(D⊠)→ C(D⊠)

即使總量沒有下降，也可能提高整體治理能力。

SECTION

13. 第七種轉移：Local → Platform

平台化是另一個經典例子。

每支產品團隊自己處理：

- deployment；
- authentication；
- logging；
- monitoring；
- secrets；
- CI/CD；

會形成大量：

> C_(duplicated)

建立 internal platform 後：

> C_(product teams)downarrow

但平台團隊承擔：

> C_(platform)uparrow

這是一種：

SECTION

Complexity Concentration

如果做好，這是好事。

因為：

> N_(teams handling complexity)downarrow

同時：

> Expertise Concentrationuparrow

但新風險是：

> Platform Criticalityuparrow

以及：

> Organizational Dependencyuparrow

SECTION

14. 第八種轉移：Implementation → Verification

AI coding 讓這個問題在 2025–2026 年變得更加明顯。

2026 年一項對專業軟體工程師的 longitudinal study 發現，使用 AI coding assistants 後，多數開發活動所需時間下降，其中 82% 受訪者表示寫程式碼花費的時間更少；但研究同時觀察到工作重心由 creation 轉向 verification，並提出：

SECTION

Supervisory Engineering Work

用來描述：

- directing AI；
- evaluating output；
- correcting output。

也就是：

> C_(code creation)downarrow

但：

> C_(verification)uparrow

以及：

> C_(supervision)uparrow

可能同時成立。

所以 AI 不一定只是：

> 把軟體工程變簡單。

它也可能是：

> 把工程複雜度從 syntax production 移到 specification、review、verification、architecture 與 governance。

SECTION

15. AI 生成量越大，複雜度轉移速度越快

如果 AI 讓：

> V_(code generation)

提高十倍，

而：

> V_(verification)

只提高兩倍，

則：

> B_(verification)

即 verification backlog 可能上升。

這不是說 AI 必然造成品質下降。

而是：

> Production Bottleneck

可能從：

writing code

轉成：

understanding
reviewing
testing
integrating
governing

因此：

SECTION

Bottleneck Transfer

也是 Complexity Transfer 的一種。

SECTION

16. 第九種轉移：Human Memory → Machine Metadata

Dynamic MSSP/FPL 本身也會造成複雜度轉移。

原本很多架構知識存在：

> C_(human tacit knowledge)

例如：

> 「這個模組其實不能直接碰那張表。」

FPL 把它寫成：

```
authority:
  forbidden:
    - user_credentials
```

於是：

> C_(human memory)downarrow

但：

> C_(schema)
> +
> C_(validator)
> +
> C_(governance)
> uparrow

這是合理交換。

因為機器可驗證複雜度通常比：

> 只存在某個人腦中的複雜度

更容易治理。

因此 Dynamic MSSP 的目標不應是：

> min |C|

而是：

> min Ungovernable Complexity

SECTION

17. 不應叫「複雜度守恆定律」

這裡必須保持理論嚴謹。

直覺上很容易說：

> 複雜度不會消失，只會轉移。

但這不能當成嚴格定律。

因為優秀演算法、正確抽象、需求刪除、資料模型改善，確實可以真正降低總複雜度。

例如：

> $O(n^2)$
> →
> $O(n \log n)$

可能是實質改善。

刪除不再需要的 feature，也可能真正：

> $C_{total} \downarrow$

所以本文不提出：

SECTION

Complexity Conservation Law

而提出较弱、可防止誤判的原則：

SECTION

Complexity Non-Disappearance Assumption

即：

> 在重構或抽象後，不應預設被局部消除的複雜度已從整個系統消失；必須檢查它是否被刪除、壓縮、隱藏、集中或轉移。

SECTION

18. 五種複雜度結果

對一次重構 R，可以有：

SECTION

18.1 Elimination

真正刪除不必要複雜度：

> C_(total)downarrow

SECTION

18.2 Compression

用更強抽象壓縮多個重複結構。

總複雜度可能下降，但新抽象承擔更高密度語義。

SECTION

18.3 Concealment

底層複雜度仍在，只是大部分使用者看不到。

> C_(visible)downarrow

SECTION

18.4 Concentration

複雜度從多處集中到少數平台、服務或專家。

> Dispersion(C)downarrow

但：

> Criticality_(center)uparrow

SECTION

18.5 Transfer

複雜度從一個維度或主體搬到另一個：

> C⊠downarrow, C⊠uparrow

這五者必須區分。

SECTION

19. 複雜度流

為了描述這種變化，本文提出：

SECTION

Complexity Flow

令：

> F_(ij)

表示複雜度從維度 i 轉向維度 j 的流量。

例如：

> F_(code→operations)>0

> F_(human→metadata)>0

> F_(monolith→network)>0

因此重構不是只比較：

> C□

和：

> C_(t+1)

還要觀察：

> F_C(t,t+1)

即複雜度流矩陣。

SECTION

20. 好的複雜度轉移

複雜度轉移不應被污名化。

好的轉移通常具有以下特徵：

20.1 從不可見 → 可見

> C_(implicit)
> →
> C_(explicit)

20.2 從人工記憶 → 可驗證規格

> C_(tacit)
> →
> C_(machine-checkable)

20.3 從到處重複 → 集中平台

> C_(duplicated)
> →
> C_(centralized)

20.4 從難自動化 → 易自動化

> C_(manual)
> →
> C_(automatable)

20.5 從錯誤 owner → 正確 owner

> C_(misowned)
> →
> C_(properly owned)

所以真正目標不是：

> 複雜度零。

而是：

> 複雜度出現在最適合承擔它的位置。

SECTION

21. 壞的複雜度轉移

反過來，壞轉移可能是：

21.1 從顯式 → 隱式

clear business rules
→ magical framework behavior

21.2 從局部 → 全局

local helper
→ shared singleton

21.3 從技術 → 人工

automated validation removed
→ operator must remember

21.4 從 code → configuration without validation

if/else
→ huge YAML

21.5 從開發 → 使用者

system cannot resolve ambiguity
→ user manually cleans everything
此時局部工程數字可能改善，

但：

> C_(effective system)uparrow

SECTION

22. 重構評估需要「前後複雜度向量」

因此一次重要重構前，可以記錄：

> C_(before)

重構後：

> C_(after)

再計算概念上的：

> ΔC
> =
> C_(after)
> -
> C_(before)

例如：

```
complexity_delta:
  code: -0.40
  dependency: +0.15
  operations: +0.20
  data: +0.05
  cognitive: -0.30
  verification: +0.10
```

這不是要假裝數字可以精確比較不同維度。

而是建立：

> 多維變化帳本。

SECTION

23. 再加入「可治理性」

複雜度本身還不是唯一重要變量。

定義：

> G(C)
> =
> Governability of Complexity

考慮：

- observability；
- ownership；
- testability；
- determinism；
- documentation；
- tooling；
- reversibility；
- automation。

則一個重構即使：

> C_(total)

沒有明顯下降，

但：

> G(C)uparrow

仍然可能是非常成功的架構改善。

例如：

10 種 undocumented human exceptions
轉成：

1 個複雜但有 schema、tests、owner 的 policy engine
總語義不一定少很多。

但治理能力提高巨大。

SECTION

24. 重構效益的新模型

因此，重構價值不應只寫成：

> V_R
> =
> -Δ C

而可以概念化為：

> V_R
> =
> α
> Δ G
> +
> β
> Δ O
> +
> γ
> Δ R_v
> +
> δ
> Δ A

其中：

- Δ G：governability 改善；
- Δ O：observability 改善；
- Δ R_v：reversibility 改善；
- Δ A：automation potential；
- Δ C_(harmful)：有害複雜度變化。

意思是：

> 好的架構改善未必把系統變得「簡單」，但應把它變得更可理解、更可驗證、更可逆、更能被正確主體治理。

SECTION

25. Dynamic MSSP：從 Architecture Diff 升級成 Complexity Flow

因此未來 FPL/MSSP 的重構分析不應只有：

```
before graph
vs
after graph
```

還應記錄：

```
transformation:
 id: extract-payment-core
 removed:
  - duplicated_payment_rules
 introduced:
  - payment_core_service
  - service_api
  - retry_policy
  - tracing
  - deployment_unit
 complexity_flow:
  code_to_dependency: medium
  code_to_operations: high
```

```
  local_to_platform: medium
benefits:
  module_boundary: high
  testability: high
  independent_deployment: medium
new_risks:
  network_failure: medium
  centrality: high
evidence:
  - dependency_graph
  - runtime_trace
  - deployment_metrics
```

這樣 AI 才不會只看到：

> cycle 消失了，好耶。

而是繼續問：

> cycle 消失後，複雜度跑去哪裡？

SECTION

26. AI Agent 可以做複雜度流偵測

AI 特別適合分析：

> C_(before)
> →
> C_(after)

因為證據可能跨越：

- code diff；
- dependency graph；
- CI；
- infrastructure；

- config；
- runtime；
- ticket；
- incident；
- human workflow。

  可以讓 Agent 提出：

  Hypothesis:
  The refactor reduced local duplication,
  but increased central dependency and
  introduced two new operational failure modes.

  並附：

- evidence；
- confidence；
- counterevidence。

  這仍然是：

> AI Hypothesis
> ≠
> Truth

但比單一 lint metric 更接近架構現實。

SECTION

27. AI 本身也是複雜度轉移器

更有趣的是：

Dynamic MSSP 本身會把：

> C_(human architecture review)

部分轉移到：

> C_(AI inference)
> +
> C_(evidence pipeline)
> +
> C_(verification)

這不是問題。

只要：

> G(C_(new))
> >
> G(C_(old))

就可能值得。

也就是：

> AI 的真正價值不必是「讓複雜度消失」。
>
> 它可以是：
>
> 把人類難以持續處理的複雜度，轉成機器可以持續觀測、搜尋、比對與預警的形式。

SECTION

28. 命題 7：複雜度轉移命題

本文提出系列第七命題。

SECTION

複雜度轉移命題

對軟體系統 S 施加架構變換或重構 R：

> S□
> oversetR→
> S_(t+1)

即使某一複雜度維度下降：

> C□(S_(t+1))
> <
> C□(S□)

也不能推出：

> C_(total)(S_(t+1))
> <
> C_(total)(S□)

因為可能存在：

> ∃ j:
> C□(S_(t+1))
> >
> C□(S□)

因此：

> 【Repair
> nRightarrow
> Total Complexity Reduction】

而更合理的分析對象是：

> 【ΔC
> +
> F_C
> +

即：

- 複雜度向量變化；
- 複雜度流；
- 可治理性變化。

SECTION

29. 重構悖論的強版本

由此可以得到：

> 最成功的重構，有時不是讓系統擁有更少複雜度，而是讓複雜度終於出現在正確的地方。

例如：

> Hidden Human Knowledge
> →
> Explicit Rules

> Duplicated Logic
> →
> Central Service

> Implicit Deployment Ritual
> →
> CI/CD Pipeline

> Local Failure Guessing
> →
> Observability Platform

這些改造甚至可能增加：

- code；
- config；
- infrastructure；

但整體：

> G(C)

大幅提高。

所以：

> 【More Artifacts
> not⇒
> Worse Architecture】

同樣：

> 【Less Code
> not⇒
> Less System Complexity】

SECTION

30. 結論：複雜度最重要的不是多少，而是在哪裡、誰承擔、能不能治理

本文從一個常見工程直覺開始：

> 「我們把它重構了，所以它更簡單。」

這句話可能正確。

但必須先回答：

> 哪一部分變簡單？

因為軟體複雜度可以在：

- code；
- dependency；
- network；
- operations；
- data；
- configuration；
- cognition；
- organization；
- verification；
- AI supervision；

之間轉移。

因此：

> 【Local Simplicity
> nRightarrow
> Global Simplicity】

而：

> 【Abstraction
> nRightarrow
> Complexity Elimination】

同樣不能直接成立。

但本文也不主張「複雜度永遠守恆」。

真正的工程原則是：

> 不要假設複雜度已經消失。先追蹤它。

因此一次架構變換後應問：

- 哪種複雜度真的被消除了？
- 哪種只是被隱藏？
- 哪種被集中？
- 哪種被轉移？
- 新位置是否更容易觀測？
- 是否有明確 owner？
- 是否更容易驗證與自動化？
- 是否增加新的單點或組織依賴？

最終，好的架構不是：

> C→0

而更接近：

> 【Complexity
> →
> Right Place
> +
> Right Owner
> +
> Right Evidence
> +
> Right Tooling】

這也把系列推向下一個問題。

如果軟體只要持續成功，就會持續改變；如果每次改變又都可能重新分配複雜度，那麼：

> 成功軟體為什麼幾乎必然越活越複雜？

下一篇：

SECTION

〈成功軟體為什麼越活越複雜？軟體演化的結構動力學〉

將正式把時間、變化率、需求壓力、治理速度與結構負擔放進同一個動態模型。

SECTION

參考文獻

- Sutoyo, E., Avgeriou, P., & Capiluppi, A. (2025). Tracing the Lifecycle of Architecture Technical Debt in Software Systems: A Dependency Approach. Proceedings of the 2025 IEEE 22nd International Conference on Software Architecture (ICSA 2025), 199–209. DOI: 10.1109/ICSA65012.2025.00028.
- Fowler, M. (2015). Microservice Trade-Offs. MartinFowler.com. https://martinfowler.com/articles/microservice-trade-offs.html
- Fowler, M. et al. Microservices Guide. MartinFowler.com. https://martinfowler.com/microservices/
- Microsoft. Designing a Microservice-Oriented Application. Microsoft Learn. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/microservice-application-design
- O’ Reilly. (2015). Microservices Shift Complexity to Where It Belongs. https://www.oreilly.com/content/microservices-shift-complexity-to-where-it-belongs/
- Fritzsch, J., Bogner, J., Wagner, S., & Zimmermann, A. et al. / related literature. Decomposition of Monolith Applications Into Microservices Architectures: A Systematic Review. IEEE Transactions on Software Engineering, 49(8), 4213–4242, 2023.

- Kluck, T., & Vermeer, L. (2017). Leaky Abstraction in Online Experimentation Platforms: A Conceptual Framework to Categorize Common Challenges. arXiv:1710.00397.
- Vella, A., & Blincoe, K. (2026). The Impact of AI Coding Assistants on Software Engineering: A Longitudinal Study. arXiv:2605.23135.
- Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity. 2025 IEEE 36th International Symposium on Software Reliability Engineering (ISSRE). DOI: 10.1109/ISSRE66568.2025.00035.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–6 篇，以及 MSSP / FPL 既有架構研究。