軟體結構動力學：補償負載、架構健康與智能治理

v1.0 / 2026-08-01 / Neo.K / 系列總結篇／Software Structural Dynamics v1.0

https://thisoneisneok.com/html/papers/ois-12.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：12 / 12

定位：系列總結篇／Software Structural Dynamics v1.0

作者：Neo.K

協作整理：Aletheia / GPT

版本：v1.0

日期：2026-08-01

SECTION

摘要

軟體工程長期擁有 technical debt、architecture erosion、software evolution、resilience、architecture conformance、runtime models、observability 與 refactoring 等大量成熟概念。然而，在真實長壽系統中，一個更難處理的問題始終存在：系統可以功能正常，卻依賴大量未被正式架構描述的補償、操作、相容性、人類知識與歷史沉積；而這些結構又會持續演化、凝固、轉移與重新分布。

本系列前十一篇依序建立：

- 表觀完整性不推出結構完整性；
- 不完整結構可由多層補償維持；

- 宣告架構與有效架構可能分離；
- Big Ball of Mud 可以具有高生存適應度；
- technical debt、architecture erosion、historical residue 必須區分；
- workaround 可以因依賴形成而凝固為 architecture；
- 重構不必然消除總複雜度，而可能轉移複雜度；
- 長期演化取決於變更速度與治理速度之間的張力；
- 成熟程式語言本身也受歷史與生態相容性塑形；
- AI coding 會改變生成、驗證與治理之間的速度比例；
- MSSP 應從靜態分類升級為具有 declared、observed、effective role 與 evidence 的動態架構狀態系統。

  本文將上述概念統合為「軟體結構動力學（Software Structural Dynamics, SSD）v1.0」。SSD 不把軟體架構視為固定拓撲，而視為一個隨時間、情境、負載、需求與治理行為持續更新的結構狀態：

> 【cal-S(t,c)
> =
> (
> A_d,
> A_e,
> R,
> B,
> N,
> C,
> L_c,
> F_C,
> ρ_g,
> E,
> Q
>)】

其中：

- A_d：Declared Architecture；
- A_e：Effective Architecture；
- R：Declared／Observed／Effective Roles；
- B：Technical Debt、Architecture Erosion、Historical Residue 等可治理結構負擔；
- N：Necessary / Essential Complexity；
- C：補償機制集合；
- L_c：Compensation Load；
- F_C：Complexity Flow；
- ρ_g：Change–Governance Ratio；
- E：Evidence；
- Q：Confidence / Uncertainty。

本文特別拒絕將「架構健康」壓縮成一個未經校準的普世分數。2022 年 architecture erosion systematic mapping study 顯示 erosion 同時涉及 technical 與 non-technical causes；2025 年 Architecture Technical Debt lifecycle 研究則發現 debt repayment 後 FAN-IN 平均增加 57.5%、FAN-OUT 增加 26.7%，說明某個局部指標改善可能伴隨其他結構維度惡化。Google SRE 對 toil 的治理同樣顯示，人類操作負擔必須被量化與限制，而不能因服務仍可運行就視為不存在。

因此 SSD v1.0 採用「狀態向量＋時間導數＋複雜度流＋證據包＋治理閾值」而非單一健康分數。Dynamic MSSP 負責建立動態架構狀態與治理方法；FPL 負責把規格、事件、證據、角色、依賴、補償與決策編譯成可執行工具鏈；AI 則只在 deterministic rule 與統計 evidence 無法充分判斷時進行 sparse reasoning，且其輸出保持為 evidence-backed hypothesis，而非直接成為 architecture truth。

本文最後提出 SSD v1.0 的最小工程閉環：

> 【Observe
> →
> Model
> →
> Compare

真正的目標不是得到「完美架構」，而是讓長期演化中的軟體能持續知道：自己現在靠什麼活著、正在往哪裡變、哪些負擔正在累積、哪些補償已成為承重結構，以及哪些改動可以安全地被接受、遷移或淘汰。

關鍵詞：Software Structural Dynamics、補償負載、架構健康、Dynamic MSSP、FPL、technical debt、architecture erosion、effective architecture、complexity flow、AI governance

SECTION

1. 從一個很普通的工程疑問開始

整個系列最初的問題其實很簡單：

> 一個看起來很亂、很多地方不完整的軟體，為什麼仍然可以正常工作？
>
> 如果只從 source code 看，答案常常令人困惑。
>
> 因為真實系統可能同時有：

- 醜陋 workaround；
- 手工流程；
- retry；
- compatibility layer；
- emergency script；
- undocumented knowledge；
- legacy API；
- shared spreadsheet；
- 人工 reconciliation；
- 外部 SaaS；
- 固定部署儀式。

> 然而使用者看到的仍然是：

系統正常

所以第一篇提出：

> I_onRightarrow I_s

即：

SECTION

Observable Integrity does not imply Structural Integrity.

這是整個系列的入口。

SECTION

2. 第一層：完整性不是單一維度

軟體完整性至少可以區分：

> I(t)
> =
> [
> I_f,
> I_s,
> I_e,
> I□,
> I_o
>]

其中：

- I_f：Functional Integrity；
- I_s：Structural Integrity；
- I_e：Evolutionary Integrity；

- I□：Knowledge / Understandability Integrity；
- I_o：Observable Integrity。

因此：

> I_f≈1

不代表：

> I_s≈1

更不代表：

> I_e≈1

或：

> I□≈1

一個產品今天能工作，

只證明：

> 在目前環境與補償條件下，它完成了目前被觀察的工作。

SECTION

3. 第二層：補償性完好

如果：

> I_o>I_s

差距從哪裡來？

第二篇提出：

I_o(t)
=
Φ
(
I_s(t),
C(t),
E(t)
)

其中：

C

代表補償機制。

補償可以存在於：

- technical；
- human；
- operational；
- compatibility；
- external ecosystem；
- governance。

NIST 的 compensating control 提供了一個既有正式對照：當 baseline control 無法使用時，可以採用管理、操作或技術上的替代控制以提供 equivalent／comparable protection。

SSD 將這個思想擴張到一般軟體結構：

> 原生結構沒有直接提供的穩定性，可以由其他結構提供。

SECTION

4. 補償不是缺陷的同義詞

這一點必須再次固定。

例如：

- redundancy；
- retry with bounded policy；
- graceful degradation；
- failover；
- backup；
- human approval；

  都可能是健康架構。

  因此：

> C>0

不能推出：

> Q_s<0

真正值得治理的是：

> 補償是什麼？
>
> 它在補什麼？
>
> 誰依賴它？
>
> 有沒有 owner？
>
> 有沒有 evidence？

> 能不能替換？

> 它失效時會發生什麼？

SECTION

5. 第三層：宣告架構與有效架構

第三篇提出：

> A_d
> ≠
> A_e

其中：

> A_d
> =
> Declared Architecture

是：

- architecture document；
- ADR；
- FPL；
- module rule；
- dependency policy；
- intended ownership。

而：

> A_e
> =
> Effective Architecture

是：

> 真正參與功能、狀態、資料、恢復、權限、相容與結果形成的關係結構。

因此：

$$
A_e = \Psi(A_C, A_R, A_D, A_O, A_H, A_X, A_K)
$$

其中包含 code、runtime、data、operations、human、external ecosystem 與 compatibility。

SECTION

6. 真正的系統不是 repository

所以：

$$
\text{System} \neq \text{Repository}
$$

$$
\text{System} \neq \text{Diagram}
$$

$$
\text{System} \neq \text{Runtime Trace Alone}
$$

更接近：

> 【System
> =
> Effective Relation Structure】

這個定義非常重要。

因為只要 system boundary 定錯，

後面的 health measurement 就全部會錯。

SECTION

7. 第四層：生存能力與結構品質分離

第四篇處理 Big Ball of Mud。

其核心不是：

> 屎山其實很好。

而是：

> 【Q_s
> ≠
> F_v】

其中：

- Q_s：Structural Quality；
- F_v：Viability / Survival Fitness。

一個系統可以：

> Q_sdownarrow

同時：

> F_vuparrow

因為它可能擁有：

- local adaptability；
- embedded operational knowledge；
- compatibility；
- compensation；
- familiarity capital；
- mature recovery routines。

所以：

> Messiness
> ≠
> Non-viability

SECTION

8. 生存不等於低成本

生存型泥巴通常支付的是：

> C_(Δ)
> +
> L_c
> +
> D_(hidden)
> +
> R_(rewrite)

即：

- change cost；
- compensation load；
- hidden dependency；
- rewrite risk。

因此它的問題不是：

> 完全不能運作。

而是：

> 越來越昂貴地維持運作。

SECTION

9. 第五層：結構負擔必須異質化

第五篇提出：

> D≠ E≠ H

分別為：

- Technical Debt；
- Architecture Erosion；
- Historical Residue。

並加入 Necessary / Essential Complexity：

> N

來避免把問題世界本身的必要複雜度誤判成缺陷。

在第五篇中，為了分類方便，曾將：

> [D,E,H,N]

一起稱作結構負擔向量。

SSD v1.0 在此做一個語義上的正式精煉：

> 【B(t)
> =
> [D(t),E(t),H_u(t)]】

其中：

> H_u

表示 unjustified／harmful historical residue。

而：

> N(t)
> =
> Necessary Complexity

獨立建模。

原因很簡單：

> 必要複雜度是需要治理的複雜度，但不應被語義上稱為「債務或負擔」。

SECTION

10. Technical Debt 是未來成本關係

SEI 將 technical debt 描述為：

> 短期便利的設計或建造方式，使未來完成相同工作變得更昂貴。

因此：

> D
> =
> Future-Cost Relation

而不是：

> D
> =
> Ugly Code

所以：

> Old
> nRightarrow
> D

> Complex
> nRightarrow
> D

> AI-generated
> nRightarrow
> D

SECTION

11. Architecture Erosion 是時間過程

Architecture erosion 更適合寫成：

> (dE)/(dt)

它關注：

- architecture violation；
- structural degradation；
- deviation；
- evolution difficulty。

2022 systematic mapping study 分析 73 篇研究，並特別指出：

> architecture erosion 不只有 technical causes，non-technical causes 也需要同等重視。
> 因此：

> E

不能只從 AST 推斷。

SECTION

12. Historical Residue 是時間沉積

Historical Residue 描述：

在：

> t⊠

某結構：

> x

有合理存在理由：

> J(x,t⊠)>0

但在：

> t⊠

理由已弱化：

> J(x,t⊠)≈0

而：

> x∈ S(t⊠)

仍成立。

它可能是：

- harmless；
- still useful；
- active compatibility；
- removal candidate。

所以 SSD 不把：

> H

全部視為 harmful。

真正治理的是：

> H_u

即缺乏目前有效理由、但仍產生成本的歷史殘留。

SECTION

13. 第六層：補償可以凝固

第六篇提出：

> 【C⊠→ A_(t+1)】

補償不只是「存在很久」才算凝固。

真正關鍵是：

> Dependents(C)uparrow

> ContractSurface(C)uparrow

> RemovalCost(C)uparrow

因此：

> Age
> ≠
> Solidification

SECTION

14. 架構重量

補償 c⊠ 的架構重量可以概念化為：

> W_A(c⊠,t)
> =
> f(
> P⊠,
> D⊠,
> K⊠,
> R⊠,
> C⊠^(surface),
> O⊠,
> S⊠^(-1)

其中：

- P：persistence；
- D：dependency；
- K：criticality；
- R：removal cost；
- C^(surface)：contract surface；
- O：operational integration；
- S：substitutability。

  這不是已校準的 universal metric。

  它是 evidence schema。

SECTION

15. 第七層：複雜度會流動

第七篇提出：

> Repair
> nRightarrow
> Total Complexity Reduction

2025 ICSA 的 Architecture Technical Debt lifecycle 研究提供很好的實證警告：

ATD repayment 後：

> FAN-IN
> uparrow57.5%

> FAN-OUT
> uparrow26.7%

平均而言依賴變得更集中。

這說明：

> 一個 repair 可以是真的成功，同時把複雜度搬到其他地方。

SECTION

16. 複雜度向量

SSD 使用：

```
C_x(t)
=
[
C_c,
C_d,
C□,
C_o,
C_s,
C_f,
C_h,
C_g,
C_v,
C_a
]
```

來提醒治理者至少考慮：

- code；
- dependency；
- network/distribution；
- operations；

- state/data；
- configuration；
- human/cognitive；
- governance；
- verification；
- AI supervisory complexity。

  這不是最終 taxonomy。

  核心原則是：

> $$C_{(system)} \neq C_{(code\text{-}only)}$$

SECTION

17. 複雜度流

定義：

> $$F_{(ij)}$$

表示 complexity 從維度 i 移向 j。

例如：

> $$F_{(code \rightarrow operations)} > 0$$

> $$F_{(human \rightarrow metadata)} > 0$$

> $$F_{(local \rightarrow platform)} > 0$$

所以一次重構應該分析：

> ΔC_x

以及：

> F_C

而不是只看單一 code metric。

SECTION

18. 好的轉移，是把複雜度搬到更能治理的地方

因此 SSD 的目的不是：

> $\min |C_x|$

而是：

> 【min
> Ungovernable Complexity】

好的 complexity transfer 可能是：

> Tacit
> →
> Explicit

> Manual
> →
> Automatable

> Duplicated
> →
> Centralized

> Unowned
> →
> Owned

SECTION

19. 第八層：軟體結構是一個動態系統

第八篇把時間正式帶進來。

Lehman 的 E-type software evolution 研究表明：

- 系統需要 continuing change；
- 持續變更會帶來 increasing complexity pressure；
- evolution 是 multi-loop feedback system。

因此 SSD 不問：

> 軟體現在多亂？

而還要問：

> 它正在多快地變亂、變好、轉移或凝固？

SECTION

20. 結構速度與加速度

令：

> X(t)

為結構狀態。

則：

$$
V_s(t)
=
\frac{dX}{dt}
$$

稱為：

SECTION

Structural Velocity

而：

$$
A_s(t)
=
\frac{d^2X}{dt^2}
$$

稱為：

SECTION

Structural Acceleration

因此：

$$
E=0.2
$$

但：

$$
\frac{dE}{dt}\gg 0
$$

可能比：

$$
E=0.7
$$

但：

> (dE)/(dt)<0

更加值得立即治理。

SECTION

21. Change–Governance Ratio

定義：

> 【ρ_g(t)
> =
> (V_(change)(t))/(V_(governance)(t))】

其中：

> V_(change)

不是 LOC／commit 數，

而是加權後會改變：

- dependency；
- authority；
- state；
- compatibility；
- operations；
- architecture role；

的結構變更速度。

而：

> $V_{governance}$

包含：

- review；
- testing；
- refactoring；
- deprecation；
- migration；
- documentation；
- observability；
- architecture recovery；
- knowledge externalization。

SECTION

22. 三種治理區域

SECTION

治理盈餘

> $\rho_g<1$

有機會：

> $\frac{dB}{dt}<0$

SECTION

動態穩態

> $\rho_g \approx 1$

新負擔被持續代謝：

> $B(t) \approx \text{bounded}$

SECTION

負擔累積

> $\rho_g > 1$

若長期成立：

> $\frac{dB}{dt} > 0$

的風險增加。

SECTION

23. 結構代謝

因此健康大型軟體真正需要的是：

SECTION

Structural Metabolism

> 【New Structure
> →
> Observe

成熟系統不是：

> 永遠不產生 technical debt。

而是：

> 能持續辨認、處理與淘汰自己產生的結構負擔。

SECTION

24. 第九層：程式語言本身也在這套動力學中

第九篇將模型提升到 programming-language layer。

成熟語言可寫成：

> L(t)
> =
> L_(core)
> +
> H(t)
> +
> K(t)
> +
> E(t)
> +
> M(t)

其中：

- history；
- compatibility；
- ecosystem；
- migration；

共同限制語言的演化自由度。

因此：

> Language Success
> →
> Compatibility Pressure

同樣是 SSD 的一個跨尺度例子。

SECTION

25. 第十層：AI 改變的是速度比

AI coding 最重要的結構影響，不是：

> AI code 一定比較好或比較差。

而是：

> $C_{\text{(candidate generation)}}\downarrow$

所以：

> $V_{\text{(change)}}$

可能迅速提高。

如果：

> $V_{\text{(verification)}}$
> +
> $V_{\text{(architecture)}}$
> +
> $V_{\text{(governance)}}$

沒有同比增加，

就形成：

SECTION

Generation–Governance Gap

SECTION

26. AI 生成—治理比率

> 【ρ_(AI)
> =
> (V_(AI-assisted change))/(V_(verification)
> +
> V_(architecture)
> +
> V_(governance))】

DORA 2025 的研究以 amplifier 描述 AI：AI 可以提高 throughput，但薄弱的基礎也可能使 instability 被放大。

因此：

> AI

在 SSD 中不是：

SECTION

Repair Tool

也不是：

SECTION

Debt Generator

而更接近：

SECTION

Structural Rate Multiplier

它可以同時放大：

> G_(burden)

或：

> R_(governance)

SECTION

27. 第十一層：Dynamic MSSP

第十一篇把所有概念工程化。

對模組：

> M

不再只存：

role = TMS

而是：

```
cal-R(M,t,c)
=
[
R_d,
R_o,
```

其中：

- R_d：Declared Role；
- R_o：Observed Role；
- R_e：Effective Role；
- E：Evidence；
- q：Confidence。

SECTION

28. Dynamic MSSP 的總狀態

現在可以正式將它嵌入 SSD。

SSD v1.0 的最小狀態為：

> 【cal-S(t,c)
> =
> (
> A_d,
> A_e,
> R,
> B,
> N,
> C,
> L_c,
> F_C,
> ρ_g,
> E,
> Q
>)】

這不是要求所有維度都必須數值化。

它更像：

SECTION

Typed Architecture State

其中一些是：

- graph；
- set；
- event；
- categorical state；
- trend；
- numerical metric；
- qualitative evidence。

SECTION

29. Compensation Load：SSD v1.0 的正式定義

前文多次使用：

> L_c
> =
> Compensation Load

現在可以正式收斂。

設系統目前有補償集合：

> cal-C
> =
> {c⊠,c⊠,…,c⊠}

對每個補償 c□ 定義一個負載函數：

> ℓ□(t)
> =
> Φ
> (
> d□,
> k□,
> m□,
> r□,
> o□^(-1),
> s□^(-1),
> h□
>)

其中：

- d□：dependence，依賴程度；
- k□：criticality，關鍵性；
- m□：maintenance / cognitive cost；
- r□：failure / removal risk；
- o□：observability；
- s□：substitutability；
- h□：human knowledge concentration。

則：

> 【L_c(t)
> =
> Σ□
> w□ℓ□(t)】

其中：

> w□

為依 domain 調整的權重。

SECTION

30. 為什麼不用「補丁數量」？

因為：

100 個低風險、完全自動化 fallback
可能比：

1 個只有某位員工知道、每天手改核心資料的流程
安全很多。

所以：

> |mathcal C|

不是好的 compensation load 指標。

真正重要的是：

> Dependence
> +
> Criticality
> +
> Hiddenness
> +
> Removal Risk

SECTION

31. Google SRE 的 Toil 是很好的營運類比

Google SRE 將 toil 定義為維持服務所需的重複、可預測工作，並指出若不加以限制，這些維運工作可以快速吞噬團隊時間。

這對 SSD 的重要性在於：

> 系統能正常運行，不代表支撐它的人工維運成本可以忽略。
>
> 因此 operational compensation 應直接進：

> L_c

而不是被排除在 software architecture 之外。

SECTION

32. Compensation Load 與 Structural Burden 不是同一件事

> L_c
> $\neq$
> B

例如：

一個非常健康的高可靠系統可能有：

- redundancy；
- backup；
- failover；
- incident process；

所以：

> $L_c>0$

但：

> B

很低。

真正危險的是：

> L_cuparrow

同時：

- observability 低；
- substitutability 低；
- human concentration 高；
- declared architecture 不知道它存在。

因此需要：

> L_c
> +
> A_d/A_e
> +
> Evidence

一起看。

SECTION

33. 架構健康不能只用一個分數

SSD v1.0 不提出：

> Health=83.7

這種 universal architecture health score。

原因是：

不同軟體具有不同：

- domain；
- reliability requirement；
- regulation；
- scale；
- lifecycle；
- safety level。

  而且多個指標之間可能 trade off。

  因此定義：

SECTION

Structural Health Profile

> 【H_S(t)
> =
> [
> I_s,
> I_e,
> I⊠,
> δ_A,
> B,
> L_c,
> ρ_g,
> G_c
>]】

其中：

- I_s：structural integrity；
- I_e：evolvability；
- $I\boxtimes$：knowledge integrity；
- δ_A：declared-effective divergence；
- B：structural burden；
- L_c：compensation load；
- ρ_g：change-governance ratio；
- G_c：governability of complexity。

SECTION

34. Health Profile 還需要導數

靜態值不夠。

因此實際 dashboard 應顯示：

> $H_S(t)$

與：

> $\frac{dH_S}{dt}$

例如：

```
architecture_health:
 erosion:
  level: medium
  trend: falling
 compensation_load:
  level: low
  trend: rising_fast
 governance_ratio:
  level: 1.34
  trend: worsening
```

```
knowledge_integrity:
  level: high
  trend: stable
```

這比：

```
Health = 72
```

更有行動價值。

SECTION

35. 架構健康是「能不能持續治理」，不只是「今天乾不乾淨」

因此 SSD 的健康概念更接近：

> 系統能否在維持功能與現實適配的同時，持續知道自己的結構、負擔、補償與變化，並有能力安全調整它們。

所以健康軟體可以有：

> $D>0$

> $H>0$

> $L_c>0$

只要它們：

- 可見；
- 有 owner；
- 可驗證；
- 有理由；
- 有治理；
- 沒有失控增長。

SECTION

36. 四種結構治理狀態

SSD v1.0 可以先使用四種高階狀態，而不是單分數。

SECTION

S0：可控穩態

特徵：

> $\rho_{glesssim1}$

> $\frac{dB}{dt} \leq 0$

> $\frac{dL_c}{dt} \leq 0$

主要補償可觀測。

SECTION

S1：受控演化

負擔或補償上升，

但：

- 有 owner；
- 有 migration；
- 有 deadline；
- 有 evidence。

  例如大型版本遷移。

SECTION

S2：結構漂移

出現：

> δ_Auparrow

> Buparrow

或：

> L_cuparrow

但尚未造成明顯 operational failure。

SECTION

S3：治理失配

長期：

> ρ_g>1

且：

> (dB)/(dt)>0

> (dL_c)/(dt)>0

再加上：

- hidden critical compensation；
- low observability；
- unclear authority。

這才接近需要高優先處理的狀態。

SECTION

37. 不要把 S3 直接叫「系統要死了」

因為第四篇已經證明：

> $Q_s \neq F_v$

一個 S3 系統仍然可能：

> $I_o \approx 1$

而且：

> F_v

短期很高。

真正意思是：

> 目前可觀察成功愈來愈依賴難以治理的結構。

這就是「表觀完好系統」系列真正要抓住的風險。

SECTION

38. Controlled Decompensation：安全地測試支架

如果懷疑某補償：

> c⊠

已經成為隱性承重結構，

可以在：

- staging；
- simulation；
- shadow traffic；
- chaos environment；

  中進行受控移除：

> c⊠downarrow

並觀察：

> Δ I_o

> Δ F_v

> Δ A_e

本文稱這類方法：

SECTION

Controlled Decompensation

它不是直接在 production 拔掉 workaround。

而是：

> 在安全條件下測量系統對某個補償的真實依賴。

SECTION

39. Replacement-Before-Removal

若反事實測試證明：

> R(c⊠)

仍然存在，

則遵守第六篇：

> 【Retire(c⊠)
> only after
> Transfer(R(c⊠))】

即：

> 先轉移責任，再移除支架。

這比「clean code therefore delete」安全得多。

SECTION

40. Architecture Governance Loop

SSD v1.0 最終閉環：

> 【Observe
> →
> Model
> →
> Compare
> →
> Infer
> →
> Govern
> →
> Act
> →
> Verify
> →

具體對應：

Observe

收集 source、dependency、runtime、data、operations、history。

Model

更新：

> mathcal S(t)

Compare

計算：

- divergence；
- trend；
- burden；
- compensation；
- complexity flow。

Infer

在不確定區域建立 architecture hypothesis。

Govern

根據 authority 決定：

- retain；
- accept；
- refactor；

- migrate；
- deprecate；
- investigate。

Act

人類或 agent 執行批准的 change。

Verify

重新量測 runtime 與 effective architecture。

Update

產生：

> mathcal S(t+1)

SECTION

41. Dynamic MSSP 是 SSD 的治理語義層

可以將關係寫成：

> 【SSD
> →
> Dynamic MSSP
> →
> FPL】

其中：

SECTION

SSD

回答：

> 系統結構如何演化？

SECTION

Dynamic MSSP

回答：

> 這些演化在架構角色、責任、補償與治理上代表什麼？

SECTION

FPL

回答：

> 如何把這些模型變成可編譯、可檢查、可觀測的工程工具鏈？

SECTION

42. FPL IR：最小 SSD 擴充

未來 FPL IR 至少可以新增：

```
architecture_state:
 observed_at: 2026-08-01T12:00:00+08:00
 context: production
 declared_architecture:
  version: "3.4"
 effective_architecture:
  evidence_window: 30d
 structural_burden:
  technical_debt:
   trend: stable
  erosion:
   trend: rising
  historical_residue:
```

```
    trend: falling
  compensation:
   load:
    trend: rising
  governance:
   change_rate: high
   governance_rate: medium
   ratio:
    status: overloaded
  evidence:
   coverage: medium
  uncertainty:
   organizational_context: high
```

SECTION

43. 每個 Entity 都需要 Structural Passport

對 module／service／workflow／agent：

```
entity:
 id: payment-reconcile
 role:
  declared: TMS
  effective:
   candidate: SMS
   confidence: medium
 authority:
  writes:
   - settlement_state
 compensation:
  type: operational
  solidification: high
 burden:
  historical_residue: medium
 dynamics:
  change_velocity: low
  dependency_growth: high
 evidence:
  - runtime
  - incidents
  - git
  - runbook
 governance:
  action: ROLE_REVIEW_REQUIRED
```

它不是「身份證」式僵化 metadata，

而是：

> 可隨 evidence 更新的結構履歷。

SECTION

44. Deterministic First

SSD 的 AI 原則仍然是：

> 【Deterministic First】

能由：

- parser；
- compiler；
- graph；
- schema；
- policy engine；

確定的事情，

不要交給 AI 猜。

例如：

undeclared dependency exists
可以直接算。

SECTION

45. Evidence Before Inference

只有：

- historical reason；
- implicit contract；
- likely compensation；
- role shift；
- business criticality；
- human workflow；

這類跨來源、語義模糊問題，

才進：

SECTION

AI Sparse Reasoning

SECTION

46. 2026 的 LLM architecture study 支持 Hybrid Route

2026 年一項研究分析：

- 980 個 ADR；
- 109 個 GitHub repositories；

發現 LLM 對：

- explicit；
- code-inferable；

architectural decisions 的 violation detection 較強，

但在：

- implicit；
- deployment-oriented；
- organizational knowledge；

上較弱。

這代表：

> AI 已足以成為架構治理的重要輔助層，但尚不足以作為唯一架構真相來源。

因此：

> 【AI Hypothesis
> +
> Evidence
> +
> Authority】

才是合理組合。

SECTION

47. Evidence Packet v1.0

每個非確定性判斷輸出：

```
claim:
 subject: payment-reconcile
 proposition:
  "effective role may have shifted from TMS to SMS"
evidence:
 runtime:
  critical_path_rate: 0.98
 incident:
  recovery_required: 8/8
 dependency:
  downstream_dependents: 14
```

```
counterevidence:
 - "degraded operation is possible without module"
confidence:
 level: medium_high
authority:
 required:
  - architecture_owner
action:
 suggested:
  - role_review
```

核心是：

> 【Claim
> +
> Evidence
> +
> Counterevidence
> +
> Confidence
> +
> Authority】

SECTION

48. Governance Event，而不是自動改真相

當：

> $R_d \neq R_e$

或：

> $\delta_A > \delta_{(threshold)}$

SSD 不自動寫回 declared architecture。

而產生：

SECTION

Governance Event

例如：

ROLE_REVIEW_REQUIRED
COMPENSATION_REVIEW_REQUIRED
DEPRECATION_CANDIDATE
ARCHITECTURE_DRIFT_ALERT
MIGRATION_REQUIRED
因此：

> 【Inference
> ≠
> Authority】

SECTION

49. SSD v1.0 的 MVP 可以非常小

第一個工程 MVP 不需要理解整個人類組織。

只需：

SECTION

Input

- FPL declared architecture；
- repository；
- git history；
- dependency graph；
- basic runtime traces；

- incident metadata。

SECTION

Engine

- deterministic rules；
- dependency trends；
- declared/effective diff；
- compensation registry；
- AI sparse reasoning。

SECTION

Output

- architecture residual；
- role divergence；
- burden trend；
- compensation candidates；
- evidence packets；
- governance events。

SECTION

50. MVP 不需要一開始就算 Universal Health Score

第一版甚至應該拒絕：

Architecture Health = 82%

而輸出：

3 critical role divergences
2 hidden compensations
erosion trend rising
governance ratio overloaded
historical residue falling
這已經足夠有工程價值。

SECTION

51. SSD v1.0 的驗證方法

如果未來要把本文從理論框架推到研究驗證，至少需要四類研究。

SECTION

51.1 Retrospective Validation

使用具有：

- git；
- ADR；
- issue；
- incident；

  的成熟 repository，

  回放：

> mathcal S(t)

看模型能否在重大 architecture change 前發現：

- divergence；
- compensation；

- erosion。

SECTION

51.2 Expert Ground Truth

讓 maintainer／architect 標註：

- effective core；
- workaround；
- hidden dependency；
- technical debt；
- residue。

比較 Dynamic MSSP 推論。

SECTION

51.3 Ablation

比較：

static-only
runtime-only
AI-only
hybrid
哪種 architecture inference 最可靠。

SECTION

51.4 Longitudinal Validation

真正持續：

> 6~24

個月觀察：

> $\frac{dB}{dt}$

> $\frac{dL_c}{dt}$

> ρ_g

是否與：

- incident；
- change lead time；
- maintainability；
- architecture review；

存在穩定關係。

SECTION

52. False Positive 是核心工程風險

Architecture tool 最容易死亡的原因之一是：

> 每個 PR 都叫你修一堆沒人在乎的東西。

所以 Dynamic MSSP 的第一工程目標不應是：

> $\text{Recall}=1$

而是：

> 高價值事件的低噪音偵測。

因此需要：

- severity；
- evidence threshold；
- hysteresis；
- time window；
- suppression；
- exception；
- owner routing。

SECTION

53. SSD 不是「用 AI 自動重構所有 legacy」

這套理論最容易被誤讀成：

> 讓 AI 自己掃 repository，然後把舊 code 清乾淨。

完全不是。

前十一篇反而一直證明：

> Ugly
> nRightarrow
> Useless

以及：

> Violation
> nRightarrow
> Immediate Repair

AI 的第一任務應該是：

> 理解承重關係。

而不是：

> 自動拆承重牆。

SECTION

54. SSD 也不是「所有軟體都會越來越爛」

同樣：

> tuparrow
> nRightarrow
> Buparrow

因為：

- refactoring；
- deprecation；
- migration；
- tooling；
- governance；

可以讓：

> Bdownarrow

SSD 研究的是：

SECTION

Dynamics

而不是宣稱：

SECTION

Fate

SECTION

55. SSD 不是複雜度守恆定律

第七篇已限制：

> complexity 可以真的被消除。

所以 SSD 不主張：

> C_(before)
> =
> C_(after)

真正主張的是：

> 不要在沒有追蹤的情況下假定複雜度已經消失。

SECTION

56. SSD 不把所有人類操作當成缺陷

NIST 本身就承認 management、operational、technical controls 可以共同提供保護。

Google SRE 也將某些 operational work 視為不可避免，只是需要限制 toil 的累積。

所以：

> Human-in-the-loop
> nRightarrow
> Bad Architecture

真正問題是：

- 是否明確；
- 是否可替代；
- 是否集中；
- 是否可驗證；
- 是否失控。

SECTION

57. SSD v1.0 的十二個命題

至此，可以把十二篇壓縮成十二個命題。

SECTION

P1 表觀完好命題

> I_onRightarrow I_s

SECTION

P2 補償性完好命題

> I_o
> =
> Φ(I_s,mathbf C,E)

SECTION

P3 宣告—有效架構分離命題

> A_dnRightarrow A_e

SECTION

P4 生存架構命題

> Q_snRightarrow F_v

SECTION

P5 結構負擔非同質命題

> D≠ E≠ H≠ N

SECTION

P6 補償凝固命題

> C⊠→ A_(t+1)

SECTION

P7 複雜度轉移命題

> Repair
> nRightarrow
> Total Complexity Reduction

SECTION

P8 軟體結構動力命題

$$
\frac{dB}{dt} = G_{burden} - R_{governance}
$$

SECTION

P9 語言歷史妥協命題

$$
L(t) = L_{core} + H + K + E + M
$$

SECTION

P10 AI 生成—治理不對稱命題

$$
V_{AI\ change} > V_{trusted\ governance} \Rightarrow \frac{dB_{AI}}{dt} > 0
$$

具有更高風險。

SECTION

P11 架構角色狀態命題

$$
R_d \neq R_o \neq
$$

可以是合法、可治理的狀態。

SECTION

P12 軟體結構動力學命題

軟體架構健康不能僅由靜態結構或單一品質分數描述，而應由：

> 【cal-S(t,c),
>
> (dmathcal S)/(dt),
>
> mathbf F_C,
>
> mathbf E,
>
> Governance】

共同判斷。

SECTION

58. 統一命題

因此整個系列最終可以收斂成：

> 現代複雜軟體未必由一組局部最優、結構完備的部件構成；它更可能是不同年代、不同約束、不同品質的結構，在技術補償、人類知識、操作慣例、相容性與治理機制下形成的動態穩定體。其功能完整性是一種系統層結果，而不必對應於每一局部結構的完整性。

進一步：

> 當環境、需求與使用者持續改變時，軟體結構本身也持續生成、沉積、凝固、轉移與被治理。因此架構不應只被理解為某一時刻的拓撲，而應被理解為具有狀態、速度、證據與治理能力的演化系統。

SECTION

59. Software Structural Dynamics v1.0

最後正式給出：

SECTION

Software Structural Dynamics v1.0

研究對象：

> 【cal-S(t,c)
> =
> (
> A_d,
> A_e,
> R,
> B,
> N,
> C,
> L_c,
> F_C,
> ρ_g,
> E,
> Q
>)】

核心觀測：

> 【mathbf H_S(t)
> =
> [
> I_s,
> I_e,
> I⊠,

核心動態：

> 【(dB)/(dt)
> =
> G_(burden)
> -
> R_(governance)】

核心治理閉環：

> 【Observe
> →
> Model
> →
> Compare
> →
> Infer
> →
> Govern
> →
> Act
> →
> Verify
> →
> Update】

核心工程原則：

> 【Deterministic First
> +
> Evidence Before Inference
> +
> Authority Before Commit
> +
> Replacement Before Removal】

SECTION

60. 最後結論：真正要治理的不是「屎山」，而是失去理解與控制的速度

系列最初從「屎山」開始。

到了最後，真正值得研究的已經不是：

> 這段 code 到底醜不醜？

因為一段醜 code 可能是：

- debt；
- residue；
- compatibility；
- compensation；
- necessary complexity；
- effective contract。

真正危險的是：

> 系統產生新結構的速度，長期高於團隊理解、驗證、治理與淘汰這些結構的速度。

也就是：

$$\rho_g > 1$$

長期成立。

此時即使：

$$I_o \approx 1$$

系統仍可能逐步走向：

> A_d-A_euparrow

> Buparrow

> L_cuparrow

> G_cdownarrow

反過來，如果系統能：

- 持續觀察；
- 保存 evidence；
- 辨認 effective architecture；
- 追蹤 compensation；
- 追蹤 complexity flow；
- 管理 migration；
- 讓 AI 只處理真正模糊的部分；
- 保持 authority 與 rollback；

那麼即使它是一個活了二十年的巨大系統，

它仍然可以：

> 【Complex
> ∧
> Governable】

這可能才是長壽大型軟體真正合理的目標。

不是：

> Perfect Architecture

而是：

> 【Continuously Knowable
> +
> Continuously Governable
> +
> Continuously Evolvable】

至此，《表觀完好系統：從軟體屎山、補償性完整到動態架構治理》12 篇完成。

SECTION

參考文獻

- Lehman, M. M., & Ramil, J. F. (2003). Software Evolution—Background, Theory, Practice. Information Processing Letters, 88(1–2), 33–44. DOI: 10.1016/S0020-0190(03)00382-X.
- Li, R., Liang, P., Soliman, M., & Avgeriou, P. (2022). Understanding Software Architecture Erosion: A Systematic Mapping Study. Journal of Software: Evolution and Process, 34(3), e2423. DOI: 10.1002/smr.2423.
- Software Engineering Institute, Carnegie Mellon University. (2016). Managing Technical Debt in Complex Software Systems. https://www.sei.cmu.edu/library/managing-technical-debt-in-complex-software-systems/
- Sutoyo, E., Avgeriou, P., & Capiluppi, A. (2025). Tracing the Lifecycle of Architecture Technical Debt in Software Systems: A Dependency Approach. Proceedings of ICSA 2025, 199–209. DOI: 10.1109/ICSA65012.2025.00028.
- Sutoyo, E., Avgeriou, P., & Capiluppi, A. (2026). The Dangers of Non-Self-Fixed Architecture Technical Debt and Its Impact on Time-to-Fix. arXiv:2605.16133.
- Bucaioni, A., Di Salle, A., Iovino, L., Mariani, L., & Pelliccione, P. (2024). Continuous Conformance of Software Architectures. Proceedings of ICSA 2024, 112–122. DOI: 10.1109/ICSA59870.2024.00019.

- Blair, G., Bencomo, N., & France, R. B. (2009). Models@run.time. Computer, 42(10), 22–27. DOI: 10.1109/MC.2009.326.
- Bencomo, N., France, R. B., Cheng, B. H. C., & Aßmann, U. (eds.). (2014). Models@run.time: Foundations, Applications, and Roadmaps. Springer.
- Google. Site Reliability Engineering Workbook — Eliminating Toil. https://sre.google/workbook/eliminating-toil/
- NIST Computer Security Resource Center. Compensating Controls / Compensating Security Control. https://csrc.nist.gov/glossary/
- DORA. (2025). State of AI-assisted Software Development. Google / DORA.
- DORA. (2026). DORA 2025: Year in Review. https://dora.dev/insights/dora-2025-year-in-review/
- Su, R., Bakhtin, A., Ahmad, N., Esposito, M., Lenarduzzi, V., & Taibi, D. (2026). Evaluating Large Language Models for Detecting Architectural Decision Violations. arXiv:2602.07609.
- Zhang, Y., Xu, Z., Liu, C., Chen, H., Sun, J., Qiu, D., & Liu, Y. (2023). Software Architecture Recovery with Information Fusion. arXiv:2311.04643.
- Foote, B., & Yoder, J. (1997/1999). Big Ball of Mud. Pattern Languages of Program Design 4.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–11 篇，以及 MSSP / FPL / Program Ontology 既有研究。