今天的 Workaround，明天的 Architecture：補償凝固命題

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

https://thisoneisneok.com/html/papers/ois-06.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：06 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

前五篇已依序建立軟體表觀完好、補償性完好、宣告架構與有效架構分離、Big Ball of Mud 的生存架構，以及技術債／架構侵蝕／歷史殘留的區分。這些命題共同導向一個尚未被完整回答的問題：

> 一個原本只是 temporary workaround、compatibility shim、manual step、shadow system 或 compensating control 的機制，究竟在什麼條件下會從「暫時支架」轉化成「不能隨意移除的真正架構」？

本文提出「補償凝固命題（Compensation Solidification Proposition）」。補償凝固不是單純指一個 workaround 存在很久，而是指補償機制在反覆使用、跨角色協調、外部依賴、隱性契約、操作常規與治理承認的作用下，逐步取得架構性地位，使其移除將改變系統的有效能力、契約、工作流或穩定性。

形式上，若補償 C 在時間 t 只是對某一缺口 G 的局部修補：

> C⊠ → G⊠

但隨著時間推進，出現：

> Dependents(C⊠)uparrow

> RemovalCost(C⊠)uparrow

> ContractSurface(C⊠)uparrow

則可能發生：

> 【C⊠ → A_(t+1)】

亦即補償開始成為有效架構的一部分。

本文綜合 workaround、organizational routines、shadow IT、enterprise-system misfit、Hyrum’s Law、compensating controls 與 security workaround 等相鄰研究，提出「補償凝固階梯」「架構重量」「凝固指數」「去凝固條件」與「替代先於移除」原則。本文並指出：補償凝固不必然是負面事件。有些補償應被淘汰，有些應被正式吸收為架構，有些則應維持為明確、可觀測、可退出的韌性機制。

對 Dynamic MSSP/FPL 而言，這意味著架構治理不能只問：

> 「這是 workaround 嗎？」

而必須進一步判斷：

> 它目前只是 workaround，還是已經成為 effective architecture？

關鍵詞：補償凝固、workaround、implicit interface、Hyrum’s Law、organizational routines、shadow IT、compensating controls、有效架構、Dynamic MSSP、FPL

SECTION

1. Temporary 往往是軟體工程裡最危險的形容詞

許多大型系統裡都有這類註解：

```
TEMPORARY
TODO remove after migration
compatibility only
workaround for issue #1827
delete after client v2 sunset
```

工程師的預期是：

> C(t⌧)

只會短暫存在。

但幾年後：

```
TODO remove after migration
```

還在。

而且它旁邊已經長出了：

- 新 API；
- 新 consumer；
- 新 monitoring；
- 新部署程序；
- 新 migration；
- 新 exception；
- 新文件；
- 新客服 SOP。

此時最大的錯誤是仍然把它理解成：

> 「一個忘記刪掉的小 workaround。」

因為真正發生的可能是：

> C_(temporary)
> →
> C_(repeated)
> →
> C_(depended)
> →
> C_(structural)

也就是：

> 它已經凝固了。

SECTION

2. 存在很久，不等於凝固

首先必須區分：

SECTION

Persistence

與：

SECTION

Solidification

一個 workaround 可以存在十年，但從未被任何其他東西依賴。

例如：

rare emergency script
只有在極特殊事件才使用，而且可被完全替換。

它的：

> T_(age)

很高，

但：

> D_(dependent)

很低。

反過來，一個剛出現三個月的 compatibility behavior，可能已經被上百個 consumer 使用。

因此：

> 【Age
> ≠
> Solidification】

補償凝固的核心不是「存在多久」，而是：

> 它是否已經改變其他結構的行為與假設。

SECTION

3. Workaround 的持續存在需要協調

Davison、Wong、Ou 與 Alter 對跨國企業系統 workaround 的研究提供了一個非常重要的線索。

當 global enterprise system 不符合 local reality 時，員工創造了不符合 corporate IT policy 的 workaround，但這些 workaround 仍然使 essential activities 得以完成，並創造企業與客戶價值。

更重要的是：

> workaround 的 coordination 對其 persistence 至關重要。

這意味著 workaround 並不是只靠「某個人偷偷做」才能存在。

它可能逐步經歷：

個人知道

→ 團隊知道
→ 團隊協調
→ 跨角色配合
→ 固定交接
→ 固定輸入輸出

一旦如此，補償就開始形成：

> Relation Structure

而「關係結構」正是架構的核心材料。

SECTION

4. Organizational Routines：補償可以改變正常工作

organizational-routines 研究對 workaround 的另一個重要觀察是：

> 個人 workaround 不只影響本人，也可能影響其他員工的活動，進而改變組織層級的工作常規。
> 因此：

> C□

原本只是個人行為，

但如果：

> C□
> →
> R□
> →
> R□
> →
> R_(org)

其中 R 表示 routine，

那麼 workaround 已經從：

SECTION

Local Adaptation

變成：

SECTION

Organizational Routine

此時即使資訊系統本身沒有改：

> $S_{software}(t_1) = S_{software}(t_2)$

有效系統仍然可能已改變：

> $S_{effective}(t_1) \neq S_{effective}(t_2)$

因為人類層架構變了。

SECTION

5. Hyrum’s Law：行為一旦被依賴，就形成隱性介面

補償凝固在純軟體世界中也有非常清楚的對應。

Hyrum’s Law 指出：

> 當 API 使用者足夠多時，不論正式 contract 承諾什麼，所有可觀察行為最終都可能被某個 consumer 依賴。

形式上：

> C_(formal)
> ⊆
> C_(effective)

而：

> C_(implicit)
> =
> C_(effective)
> -
> C_(formal)

其中：

> C_(implicit)

就是隱性契約。

假設一個 workaround 改變：

- response order；
- latency；
- error string；
- retry timing；
- output format；
- null behavior；
- migration sequencing。

  如果外部 consumer 開始依賴這些行為：

> Dependency(C)uparrow

則原本只是 workaround 的實作細節，已經開始形成：

SECTION

Implicit Interface

這就是補償凝固最典型的軟體路徑之一。

SECTION

6. Shadow IT：workaround 甚至可以長成新的子系統

Shadow IT 提供另一種更直觀的凝固方式。

當正式系統無法滿足需求時：

```
Official System
  ↓
 gap
  ↓
Excel / SaaS / Script / Database
```

最初可能只是：

> 「先這樣處理一下。」

但若：

- 多人共用；
- 有固定資料；
- 有權限；
- 有流程；
- 有版本；
- 有報表；
- 有交接；

- 有 downstream consumer；

  那麼：

> C_(shadow)

已經具備大量 subsystem 特徵。

因此：

> Shadow IT
> ⊆
> A_(effective)

可能成立，

即使：

> Shadow IT
> not⊆
> A_(declared)

這就是前一篇「架構殘差」的一個具體來源。

SECTION

7. 補償凝固的正式定義

本文將補償凝固定義為：

> 原本為填補局部缺口而存在的補償機制，在反覆使用、協調、依賴、契約形成與運行整合的作用下，逐步取得不可忽略的架構性功能，使其移除會顯著改變系統有效行為的過程。
> 令補償為：

> C

若在時間 t：

> ∃ X:
> X
> oversete→
> C

其中：

> oversete→

表示有效依賴，

並且移除 C 造成：

> Δ I_o < 0

或：

> Δ F_v < 0

或破壞有效契約：

> Δ K_e ≠ 0

則 C 已經具有：

SECTION

Architectural Weight

即架構重量。

SECTION

8. 架構重量不是二元值

一個補償不是突然從：

workaround = true
跳成：

architecture = true
更合理的是定義：

> W_A(C,t)

表示補償 C 在時間 t 的架構重量。

可概念化為：

> W_A
> =
> f(
> P,
> D,
> K,
> R,
> C_s,
> O,
> S
>)

其中：

- P：Persistence，持續性；
- D：Dependency，依賴量；
- K：Criticality，關鍵性；
- R：Removal Cost，移除成本；
- C_s：Contract Surface，契約表面；
- O：Operational Integration，操作整合；

- S：Substitutability，可替換性。

  其中：

> Suparrow
> ⇒
> W_Adownarrow

通常成立。

如果一個 workaround 可在五分鐘內被完全替換，

它的架構重量自然較低。

SECTION

9. 補償凝固階梯

本文提出六階段模型：

> C□
> →
> C□
> →
> C□
> →
> C□
> →
> C□
> →
> C□

SECTION

C□：Transient Workaround

一次性補償。

例如：

手工重跑一次
沒有持續依賴。

SECTION

C⊠：Repeated Workaround

相同補償反覆發生。

每週都需要重跑
此時：

> Frequency(C)uparrow

但仍可能只有一人知道。

SECTION

C⊠：Coordinated Compensation

其他人開始配合。

例如：

客服先標記
→ 財務匯出
→ 工程師跑 script
→ 財務確認
workaround 開始形成流程。

SECTION

C⊠：Dependent Compensation

其他模組、流程或人開始依賴它。

> Dependents(C)>0

此時它已經不能被視為孤立補丁。

SECTION

C⊠：Implicit Contract

外部或內部 consumer 對補償產生穩定期待。

例如：

- 一定在凌晨 2 點完成；
- 一定保持某排序；
- 一定輸出某欄位；
- 一定由人工核准；
- 一定會 retry 三次。

  此時：

> C
> →
> K_(implicit)

SECTION

C⊠：Effective Architecture

補償已經成為：

> A_e

的一部分。

即使 architecture document 沒承認它：

> $C \notin A\_d$

仍然可能：

> $C \in A\_e$

這就是補償凝固完成。

SECTION

10. 凝固不是只有「變正式」

一個常見誤解是：

> workaround 被寫進文件後才算架構。

不對。

文件化只是：

> $F\_(\text{formalization})$

其中一個維度。

一個完全 undocumented 的人工流程也可能：

> $W\_A \gg 0$

如果：

- 每天都跑；
- 沒有它訂單無法完成；
- 三個部門依賴；
- 客戶結果依賴；

- 停止後會造成錯帳。

  所以：

> 【Formalization
> ≠
> Architectural Existence】

這正是「有效架構」概念存在的必要性。

SECTION

11. 補償凝固的四條主要路徑

目前可以整理成四條典型路徑。

SECTION

11.1 Workaround → Routine

> Individual Workaround
> →
> Coordination
> →
> Organizational Routine

主要發生在人類與操作層。

SECTION

11.2 Implementation Detail → Implicit Contract

> Observable Behavior
> →
> Consumer Dependence
> →

Hyrum’s Law 是核心對照。

SECTION

11.3 Temporary Control → Governance Structure

例如 NIST 所稱 compensating control。

原本：

> baseline control 無法採用，先用替代控制。

若替代控制長期被正式評估、審核、稽核與依賴，

它可能：

> C_(temporary)
> →
> C_(governed)

最後成為正式治理架構。

因此補償凝固不一定是失敗。

它也可能是：

> 系統找到比原始方案更合適的長期結構。

SECTION

11.4 Shadow Tool → Effective Subsystem

> Spreadsheet
> →
> Shared Spreadsheet
> →
> Workflow
> →
> Shadow Database

這在 enterprise environment 尤其常見。

SECTION

12. 凝固的正向版本：補償可能是架構發現機制

這是一個非常重要的反轉。

假設正式系統：

> A_d

無法處理某種 local reality。

員工反覆產生：

> C

而且 C：

- 穩定有效；
- 被多人採用；
- 解決相同缺口；
- 不斷創造價值。

那麼：

> C

不一定代表：

> 「人們一直違規。」

它可能代表：

> 正式架構漏掉了一個真實需求。

因此補償凝固可以被當成：

SECTION

Architecture Discovery Signal

即：

> C_(persistent)
> →
> Possible Missing Capability

這對 Dynamic MSSP 極重要。

SECTION

13. 凝固的負向版本：臨時缺陷被依賴後變得難以修復

另一方向當然存在。

一個 bad workaround：

return empty list instead of raising error
原本只是為了避免 crash。

後來：

Client A
Client B
Client C
全部開始依賴：

empty list means retry later
此時：

> Bug-like Behavior
> →
> Implicit Contract

如果現在「修正」成正規 exception：

> Fix
> →
> Break Consumers

這就是：

SECTION

Pathological Solidification

即：

> 不良行為因依賴形成而取得架構重量。

因此：

> Bad Origin

不代表：

> Easy Removal

SECTION

14. 凝固指數

為了後續 Dynamic MSSP 的機器判斷，本文提出概念性的：

SECTION

Solidification Index

記為：

> Γ_C(t)

可由以下維度構成：

> Γ_C
> =
> α P
> +
> β D
> +
> γ K
> +
> δ R
> +
> ε C_s
> +
> ζ O
> -
> η S

其中：

- P：持續性；
- D：依賴程度；
- K：關鍵性；
- R：移除成本；
- C_s：契約表面；
- O：操作整合程度；
- S：可替換性。

  注意：

這不是已驗證量表。

它是一個研究骨架。

目的不是假裝：

> Γ_C=0.73

就具有普世客觀意義，

而是迫使架構治理系統回答：

> 為什麼我們認為它已經凝固？

SECTION

15. 一個更強的判定：反事實移除測試

最有價值的判定可能不是年齡，也不是程式碼大小。

而是：

SECTION

Counterfactual Removal Test

問：

> 如果明天把 C 完全移除，什麼會失效？

形式上：

> S^(-C)

表示移除補償 C 後的系統。

若：

> I_o(S^(-C))
> ≪
> I_o(S)

或：

> F_v(S^(-C))
> ≪
> F_v(S)

或有大量：

> BrokenDependents(C)

則 C 已具有高度架構重量。

這比：

> 「它是不是 workaround？」

更重要。

SECTION

16. 替代先於移除原則

前幾篇已經得到：

> Remove Compensation
> →
> Expose Latent Failure

的可能性。

本文將它正式收斂成：

SECTION

Replacement-Before-Removal Principle

若 C 已承擔責任：

> R(C)

則安全移除需要：

> ∃ X:
> R(X)⊒ R(C)

並且：

> Dependents(C)
> →
> X

已完成遷移。

因此：

> 【Retire(C)
> only after
> Transfer(R(C))】

這對 AI 自動重構尤其重要。

SECTION

17. 去凝固不是刪除，而是解除依賴

真正的：

SECTION

De-solidification

不是：

delete workaround

而是：

識別責任
→ 找出依賴
→ 建立替代
→ 遷移 consumer
→ 觀察 residual usage
→ 降低架構重量
→ 最後移除

形式上：

> Dependents(C,t)downarrow

> W_A(C,t)downarrow

直到：

> W_A(C,t)≈0

這時才真正安全退出。

SECTION

18. 補償凝固與歷史殘留的差別

上一篇定義：

> H
> =
> Historical Residue

這與補償凝固非常接近，但不相同。

歷史殘留強調：

> 過去留下來。

補償凝固強調：

> 留下來後被新的結構依賴。

因此：

> H
> not⇒
> Solidified

例如完全沒人用的 deprecated field 是 residue，但未凝固。

反過來：

> Solidified Compensation

幾乎必然具有某種當前依賴。

所以凝固關注的是：

SECTION

Current Structural Dependence

而不只是歷史來源。

SECTION

19. 補償凝固與技術債的差別

同樣：

> D
> ≠
> Γ_C

某個 compensation 可以是 technical debt，

也可以完全不是。

例如：

SECTION

Case A：健康凝固

temporary compensating control 經風險評估證明比 baseline 更適合，最後成為正式長期控制。

它：

> Γ_Cuparrow

但不必然：

> Duparrow

SECTION

Case B：債性凝固

shortcut 被大量依賴，每次修改都增加成本。

此時：

> Γ_Cuparrow

同時：

> Duparrow

SECTION

20. 補償凝固與架構侵蝕的差別

凝固也不必然等於 erosion。

如果：

> A_d

及時更新，把經驗證有效的補償納入正式架構：

> C∈ A_d

那麼：

> Γ_Cuparrow

但：

> E

可能下降。

反之，如果補償大量凝固但架構文件完全不更新：

> A_d
> ≠
> A_e

差距擴大，

則：

> Euparrow

可能成立。

因此補償凝固描述的是：

> 依賴與結構化。

architecture erosion 描述的是：

> 偏離與退化。

SECTION

21. Dynamic MSSP：Compensation 必須是一等物件

由此，Dynamic MSSP 不應只建模 module。

還需要：

```
compensation:
 id: legacy_empty_response_behavior
 origin:
  type: workaround
  introduced_at: 2022-04-17
  reason: "avoid client crash during partial outage"
 role:
  declared: temporary
  effective: compatibility_contract
 solidification:
  persistence: high
  dependents: 37
  criticality: high
  removal_cost: high
  substitutability: low
  contract_surface:
   - response_shape
   - retry_semantics
 evidence:
  - runtime_trace
  - client_repository_search
  - incident_history
  - support_ticket
 recommendation:
  action: migrate_before_remove
  confidence: 0.91
```

此時 MSSP 的工作不是：

TEMPORARY TOO OLD → ERROR

而是：

> Temporary 已經變成什麼？

SECTION

22. AI 的任務：發現依賴，而不是猜作者本意

補償凝固非常適合 AI 輔助，因為依賴證據可能散落在：

- code；
- API usage；
- tests；
- git；
- issue；
- runbook；
- ticket；
- chat；
- runtime；
- metrics；
- incident history。

AI 可以提出：

> H_C
> =
> Solidification Hypothesis

但必須附：

> E_C
> =
> Evidence Set

以及：

> q_C
> =
> Confidence

因此：

> AI
> →
> (H_C,E_C,q_C)

而不是：

> AI
> →
> Delete

SECTION

23. AI 時代會讓補償凝固更快

這也是 2026 年後尤其需要注意的問題。

Agentic development 會降低：

> C_(implementation)

也就是建立：

- script；
- adapter；
- agent；
- workflow；
- integration；

  的成本。

  因此：

> V_(workaround creation)uparrow

可能導致：

> N_(workarounds)uparrow

更重要的是，AI 可以在數小時內把一個 workaround：

```
script
```

擴張成：

```
service
+ API
+ UI
+ database
+ agent
```

所以：

> T_(solidification)
> downarrow

即補償凝固時間可能顯著縮短。

這也是為什麼 AI-native architecture governance 不能只靠半年一次 architecture review。

SECTION

24. 從 Shadow IT 到 Shadow AI

2026 年企業已經開始面對：

SECTION

Shadow AI

也就是員工在正式治理外使用：

- external AI；
- autonomous agent；
- vibe-coded app；
- automation；
- personal workflow。

這比傳統 Shadow IT 更容易快速形成有效子系統。

一個人今天做：

AI summarizer
下週可能：

整個團隊用
下個月：

另一流程依賴輸出
於是：

> C_(shadow-ai)
> →
> A_e

的速度可能比傳統 spreadsheet 快得多。

因此「補償凝固」在 AI 時代不是邊緣現象，而可能成為常態架構形成機制。

SECTION

25. 三種治理結果：吸收、替換、保留

當 Dynamic MSSP 發現：

> Γ_C

很高時，

不代表唯一動作是刪除。

至少有三種結果。

SECTION

25.1 Absorb

補償被證明合理。

則：

> C
> →
> A_d

正式納入架構。

SECTION

25.2 Replace

補償有效，但成本／風險太高。

則：

> C
> →
> X

先建立替代再遷移。

SECTION

25.3 Preserve

補償本身就是合理韌性或 emergency mechanism。

則保留，但要求：

- ownership；
- observability；
- test；
- activation condition；
- exit condition；
- review。

  所以：

> Solidified
> not⇒
> Remove

SECTION

26. 命題 6：補償凝固命題

本文正式提出系列第六命題：

SECTION

補償凝固命題

對補償機制 C，若其在系統演化中持續形成新的依賴、操作常規與隱性契約，使移除 C 會顯著破壞系統有效能力，則 C 將由補償機制逐步轉化為有效架構構成：

> 【C⊠
> →
> A_(t+1)】

其核心判斷不是：

> Age(C)

而是：

> 【Dependence
> +
> Criticality
> +
> Contract
> +
> RemovalCost】

並可用架構重量：

> W_A(C,t)

與凝固指數：

> Γ_C(t)

進行概念性描述。

SECTION

27. 更強版本：架構本身可以由 workaround 生成

這個命題最後其實會得到一個比預期更強的推論。

一般想像：

> Architecture
> →
> Implementation
> →
> Operations

但真實系統也可能反向：

> Workaround
> →
> Routine
> →
> Dependency
> →
> Contract
> →
> Architecture

因此：

> 【Architecture Formation
> can be bottom-up】

這正是 Dynamic MSSP 必須比靜態架構語言更動態的原因。

SECTION

28. 結論：今天的例外，可能是明天的本體

本文從一句工程世界非常熟悉的話開始：

> 「這只是 temporary workaround。」

但系統演化告訴我們：

temporary 不是本體屬性。

它只是一個當下意圖。

真正決定未來地位的是：

- 是否被反覆使用；
- 是否被協調；
- 是否被依賴；
- 是否形成隱性契約；
- 是否被操作流程吸收；
- 是否存在可替代方案；
- 移除後是否破壞有效系統。

所以：

> 【Temporary Intent
> nRightarrow
> Temporary Existence】

而：

> 【Repeated Compensation
> +
> Dependency
> +
> Contract Formation
> →
> Architectural Solidification】

最終：

> 【C⊠→ A_(t+1)】

因此，真正危險的不是：

> workaround 存在。

而是：

> workaround 已經成為架構，但我們仍然假裝它只是 workaround。

同樣地，真正的機會也在這裡：

> 某些反覆有效的 workaround，可能是在告訴我們原本架構漏掉了真實世界。

Dynamic MSSP 的責任因此不只是抓違規。

它還要辨認：

> 哪些違規正在腐蝕架構？

> 哪些 workaround 應該被淘汰？

> 哪些補償已經變成不可忽略的有效架構？

> 哪些甚至應該被正式吸收到下一版架構中？

下一篇將接著處理另一個問題。

假設我們成功重構、清理、集中或替換了一個補償：

> 複雜度真的消失了嗎？

還是只是從一個位置移到了另一個位置？

下一篇：

SECTION

〈修好真的比較簡單嗎？複雜度轉移與重構悖論〉

將正式研究：

> Complexity_(t)
> →
> Complexity_(t+1)

以及：

> Repair
> not⇒
> Total Complexity Reduction

SECTION

參考文獻

- Davison, R. M., Wong, L. H. M., Ou, C. X. J., & Alter, S. (2021). The Coordination of Workarounds: Insights from Responses to Misfits Between Local Realities and a Mandated Global Enterprise System. Information & Management, 58(8), 103530. https://doi.org/10.1016/j.im.2021.103530
- Wolf, V., & Beverungen, D. (2019). Conceptualizing the Impact of Workarounds – An Organizational Routines' Perspective. ECIS 2019 Research-in-Progress Papers.
- Wibisono, A., Sammon, D., & Heavin, C. (2022). Opening the Workaround Black Box: An Organisational Routines Perspective. Journal of Decision Systems. https://doi.org/10.1080/12460125.2022.2073647
- Malaurent, J., & Karanasios, S. (2020). Learning from Workaround Practices: The Challenge of Enterprise System Implementations in Multinational Corporations. Information Systems Journal, 30, 639–663. https://doi.org/10.1111/isj.12272
- Wright, H. Hyrum’s Law. https://www.hyrumslaw.com/

- Harrand, N., Benelallam, A., Soto-Valero, C., Bettega, F., Barais, O., & Baudry, B. (2019). API Beauty Is in the Eye of the Clients: 2.2 Million Maven Dependencies Reveal the Spectrum of Client-API Usages. arXiv:1908.09757.
- Shaikh, A. (2021). Shadow-IT System as a Workaround: A Theoretical Review. MENACIS2021.
- White, M. S. (2023). Workarounds and Shadow IT – Balancing Innovation and Risk. Business Information Review, 40(3), 114–122.
- NIST. Special Publication 800-63-4: Identify Compensating Controls. https://pages.nist.gov/800-63-4/sp800-63.html
- Huang, Z., D'Angelo, M., Miyani, D., & Lie, D. (2017). Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. arXiv:1711.00795.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–5 篇，以及 MSSP / FPL 既有架構研究。