補償性完好：為什麼不完整的軟體仍然可以正常運作

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

https://thisoneisneok.com/html/papers/ois-02.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：02 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

上一篇〈功能正常，不代表結構正常：軟體表觀完好命題〉提出：

> I_o nRightarrow I_s

亦即軟體的外部可觀察完整性，不能充分推出其內部結構完整性。但這立即產生另一個更重要的問題：如果一個系統內部並不完整，甚至存在架構侵蝕、歷史殘留、隱性依賴、未文件化知識與責任錯位，它為什麼仍然可以長期提供正常功能？

本文提出「補償性完好（Compensated Integrity）」作為分析概念，用以描述：一個系統的可觀察功能與穩定性，除了來自其原生結構，也可能由技術機制、人類技能、操作流程、相容層、組織制度與外部平台共同補償。這些補償可以是有意設計的韌性機制，例如冗餘、重試、熔斷、降級與恢復；也可以是臨時 workaround、人工修復、未文件化操作、compatibility shim 或歷史形成的非正式契約。

本文特別強調：補償本身不等於缺陷，也不等於技術債。 成熟系統本來就需要韌性與故障恢復機制。真正需要研究的是補償的來源、顯式程度、可替換性、依賴程度、維護成本與二階副作用。既有研究顯示，workaround 可以在短期提高韌性，但若長期固化，也可能遮蔽根本問題；Google SRE 的工程經驗亦顯示，retry 等看似提升韌性的機制若缺乏 budget、backoff 與整體協調，反而可能放大故障並造成級聯失敗。

因此，本文建立「補償向量」「補償負載」「補償依賴」與「補償生命周期」的初步模型，並提出：真正的軟體架構分析不能只研究系統如何執行，也必須研究系統究竟靠哪些額外支架才能維持目前的可用狀態。這將為後續「宣告架構與有效架構」「補償凝固」「複雜度轉移」與動態 MSSP 奠定基礎。

關鍵詞：補償性完好、軟體韌性、workaround、故障容錯、補償控制、操作知識、技術債、MSSP、軟體結構動力學

SECTION

1. 從「表觀—結構差距」開始

上一篇定義了一個概念性的差距：

> G_I = I_o - I_s

其中：

- I_o：Observable Integrity，表觀／可觀察完整性；
- I_s：Structural Integrity，結構完整性。

這個表示不是要宣稱兩者已經能被某個通用量表精確測量，而是用來指出一種常見現象：

> 外部看起來正常的程度，可能顯著高於內部結構自身足以直接支持的程度。

如果這個差距長期存在，就不能只說「系統很神奇地還沒壞」。

更合理的問題是：

> 有什麼東西正在替這個系統補上它本身沒有直接承擔的功能、穩定性、知識與控制？

因此本文引入：

SECTION

補償性完好

SECTION

Compensated Integrity

其基本定義為：

> 當一個系統的可觀察功能、可靠性或操作完整性，部分依賴於原生結構之外的額外機制才能維持時，稱該系統具有補償性完好。

這裡的「原生結構之外」並不表示補償機制必然位於程式碼之外。它可以存在於另一個軟體層、infrastructure、framework、middleware、測試與驗證、人類操作、團隊知識、組織流程、compatibility layer 或外部雲端服務中。

因此，「補償」描述的是功能關係，不是物理位置。

SECTION

2. 這個概念和既有 resilience 有什麼不同？

軟體工程並不缺少「如何讓系統在失敗中繼續工作」的研究。

分散式系統、Site Reliability Engineering、fault tolerance、resilience engineering 已經長期使用：

- redundancy；
- retry；
- timeout；
- exponential backoff；
- circuit breaker；
- bulkhead；

- load shedding；
- graceful degradation；
- failover；
- checkpoint；
- compensation transaction；
- chaos testing。

這些技術本來就是為了讓一個局部失敗不直接變成整體失敗。

因此：

> Failure
> not⇒
> System Collapse

正是現代可靠性工程的重要目標。

Google SRE 對 cascading failure 的討論尤其重要：部分服務失效後，剩餘服務可能因額外負載進一步失效；合理的防護包括降級結果、拒絕過載請求、限制 retry、使用 exponential backoff 與 jitter，以及控制 retry budget。

所以本文並不是提出：

> 「軟體有 fallback，因此架構不好。」

恰恰相反。

有意設計的補償能力，本來就是成熟架構的一部分。

本文研究的範圍更廣：

> 當系統能繼續運作時，我們能否分辨支撐這個結果的是原生能力、設計型韌性、臨時補償、人工 workaround，還是已被忘記的歷史支架？

這是 resilience engineering 通常不會單獨回答的結構辨識問題。

SECTION

3. NIST 的 Compensating Controls：一個重要的相鄰概念

資訊安全領域已有成熟的「compensating controls」概念。

NIST 將 compensating controls 描述為：當基準控制無法直接採用時，以其他管理、操作或技術控制提供等價或相近的保護。

抽象化後，可以寫成：

> K□ ∉ Available

但存在另一控制：

> K_c

使得：

> P(K_c) ≈ P(K□)

其中 P 表示對特定風險所提供的保護能力。

這與本文的「補償性完好」高度相關，因為兩者都承認：

> 一個能力缺口不一定只能由原來那個位置、原來那個機制補上。

但是兩者並不相同。

NIST 的 compensating controls 有明確安全與合規語境，主要回答：

> 當規定的 security/privacy control 無法使用時，什麼替代控制能提供相近保護？

本文的 Compensated Integrity 則是更一般的軟體結構概念，研究功能、穩定性、資料一致性、操作連續性、知識保存、相容性與失敗恢復如何由不同層共同補償。

因此：

> Compensating Control
> ⊂
> Compensation Mechanisms

但不能把兩個術語直接視為同義。

SECTION

4. Workaround：人類本身也可能是系統的一部分

補償性完好最容易被忽略的部分，是人類。

正式系統圖常畫成：

User
↓
Software
彷彿人只負責輸入，軟體負責完成全部工作。

真實組織卻常是：

Software
↓
輸出不完整
↓
員工判斷
↓
Excel / Email / Chat
↓
人工修正
↓
重新輸入系統
此時，人不是純粹的「使用者」。

他已經成為：

> Runtime Compensation Component

關於 ERP workaround 與 resilience 的研究指出，workaround 可以在短期克服系統性障礙並避免失敗，因此具有提升韌性的效果；但如果這些 workaround 長期被嵌入日常實踐，也可能使組織不再面對並修正系統本身的缺陷。

2026 年關於 sociotechnical system transformation 的研究也指出，表面上看似受到嚴格規劃的系統，實際上可能由大量人類技能與 workaround 維持。

這使我們得到：

> S_(effective)
> ≠
> S_(software-only)

而更可能是：

> S_(effective)
> =
> S_(software)
> +
> S_(human)
> +
> S_(practice)

這一點將在下一篇「宣告架構與有效架構」中正式展開。

SECTION

5. 補償不是一種東西，而是一個向量

本文將系統在時間 t 的補償狀態表示為：

> C(t)
> =
> [
> C_T,
> C_H,
> C_O,
> C_K,
> C_X,
> C_G
>]☒

其中：

SECTION

5.1 技術補償

> C_T = Technical Compensation

包括 retry、fallback、cache、adapter、wrapper、circuit breaker、redundancy、schema migration、reconciliation、extra validation、feature flag 與 compatibility shim。

技術補償可以是架構原生設計，也可以是後期修補。

SECTION

5.2 人類補償

> C_H = Human Compensation

包括：

- 工程師記得特殊部署順序；
- 客服知道哪些訂單必須手工修；
- DBA 知道哪些欄位不能直接更動；
- 資深工程師知道某個看似無用的服務不能刪；
- 操作者依經驗辨識系統無法表達的例外。

  這些能力如果沒有被外部化，就形成：

> Hidden Human Runtime

SECTION

5.3 操作補償

> C_O = Operational Compensation

包括手工重跑、定期清理、reboot、manual reconciliation、incident playbook、emergency script、災難切換、月底人工批次與監控後的人工作業。

C_H 描述知識與判斷能力；C_O 描述真正被執行的操作程序。

SECTION

5.4 相容性補償

> C_K = Compatibility Compensation

包括保留舊 API、bug-for-bug compatibility、deprecated field、legacy protocol、version adapter、old serialization format、雙寫、雙讀與 transitional schema。

其形成常符合：

> History
> →
> Dependency
> →
> Constraint

SECTION

5.5 外部生態補償

> C_X = External / Ecosystem Compensation

現代軟體的大量能力實際由 cloud provider、CDN、database service、identity provider、payment processor、package ecosystem、managed queue 與 observability platform 承擔。

本地程式碼庫看起來簡潔，不代表完整性完全由本地結構提供。

SECTION

5.6 治理補償

> C_G = Governance Compensation

包括 code review、change approval、separation of duties、release gate、incident review、policy、audit、permission boundary 與 human-in-the-loop。

它們不直接實現業務功能，卻能防止修改、權限與操作失控。

SECTION

6. 補償性完好的最小模型

因此，比上一篇的：

> I_o nRightarrow I_s

再前進一步，可以寫成：

> I_o(t)
> =
> Φ
> (
> I_s(t),
> C(t),
> E(t)
>)

其中：

- I_s(t)：系統原生結構所提供的完整性；
- C(t)：補償向量；

- E(t)：外部環境與負載；
- Φ：整體運行結果形成機制。

這比：

> I_o = I_s + C

更準確。

因為補償不是簡單加法。

它可能放大、抵消、延遲、遮蔽、轉移，甚至創造新的故障路徑。

例如 retry 原本是為了補償暫時性失敗：

> F_(transient)
> →
> retry
> →
> success

但如果多層同時無限制 retry：

> retry
> →
> load amplification
> →
> overload
> →
> more failures
> →
> more retries

此時補償機制反而變成：

> C → F'

也就是：補償生成新的失敗。

SECTION

7. 四種不同性質的補償

為避免「補償＝壞設計」的誤解，本文把補償分成四類。

SECTION

7.1 設計型補償

Designed Compensation

例如 redundancy、graceful degradation、circuit breaker、failover、backup、idempotency。

它們從一開始就被當成正式架構的一部分。

此時：

> $C \subseteq A\_{(declared)}$

通常屬於健康補償。

SECTION

7.2 應急型補償

Contingency Compensation

當根本修復暫時不可行時，先用其他機制降低風險或維持服務，例如關閉脆弱功能、限制某條路徑、temporary configuration、manual verification 與 emergency routing。

SECTION

7.3 湧現型補償

Emergent Compensation

沒有人正式設計它成為系統的一部分。

它是在工作中自然形成，例如 Excel 表、私人 script、Chat 提醒、人工 checklist、特定工程師的操作習慣與「大家都知道不能這樣做」的非正式規則。

它可能非常有效，但不一定被系統模型看見。

SECTION

7.4 遺留型補償

Legacy Compensation

原本也許只是短期補丁，但因時間與依賴累積而無法移除：

temporary workaround
→ repeated use
→ external dependency
→ internal assumption
→ compatibility requirement
→ permanent support
此時補償已經開始接近架構本身。

SECTION

8. 補償有兩張臉：韌性與遮蔽

同一個 workaround 可以在不同時間尺度上呈現不同價值。

短期：

> C
> →
> Failure Avoidance

長期：

> C
> →
> Root Cause Masking

因此補償不能只用：

> good/bad

判斷。

至少需要考察：

```
V(C)
=
f(
B_s,
B_l,
K,
R,
D,
O
)
```

其中：

- B_s：short-term benefit；
- B_l：long-term benefit；
- K：maintenance cost；
- R：new risk；
- D：dependency created；
- O：observability。

  一個補償機制如果效果可觀測、有 owner、有觸發條件、有退出條件、有測試、有風險界線，即使長期存在，也可能是健康架構的一部分。

  反過來，一個很小的 workaround 如果沒人知道、無法測試、只有一人會操作、不能移除、會改變資料而且沒有紀錄，就可能具有很高的結構風險。

SECTION

9. 補償生命周期

本文提出一個初步的補償生命周期：

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

C□：缺口出現

某個原生結構不能完整處理現實狀況。

C□：臨時補償

加入 patch、manual step、fallback 或 workaround。

C□：重複使用

相同問題再次出現，補償從一次性操作變成慣例。

C□：正常化

團隊開始把補償視為「系統本來就這樣」。

C□：依賴形成

其他流程、模組或外部使用者開始依賴補償結果。

C□：結構化

即使原始問題已經可以修復，也不能直接刪除補償，因為補償自身已成為系統運作條件。

因此：

> C□
> →
> A_(t+1)

是完全可能的。

這個過程將在第 6 篇以「補償凝固命題」獨立展開。

SECTION

10. 補償負載：一個系統到底背了多少支架？

如果兩個系統外部功能同樣正常：

> I_o(S□)
> ≈
> I_o(S□)

但 S□ 需要大量人工步驟、legacy adapter、manual reconciliation、多層 retry、不可替代人員與 emergency cleanup，而 S□ 幾乎不需要這些支架，兩者的結構風險顯然不同。

因此本文先提出：

> L_c
> =
> Compensation Load

即「補償負載」。

它不是單純數補丁數量。

更合理的概念模型是：

$$
L_c = \Sigma_i w_i \cdot d_i \cdot k_i \cdot r_i
$$

其中：

- w_i：該補償的重要性權重；
- d_i：其他系統對它的依賴程度；
- k_i：維護與認知成本；
- r_i：失效時風險。

目前這仍是理論骨架。

第 12 篇才會討論如何把它轉成可觀測的架構健康指標。

SECTION

11. 補償性穩定不等於內生穩定

這裡可以進一步區分：

SECTION

內生穩定

SECTION

Intrinsic Stability

系統主要靠自身明確結構維持。

與：

SECTION

補償性穩定

SECTION

Compensated Stability

系統必須持續依賴額外支架才能維持。

兩者都可能在今天正常。

但若某個關鍵補償 C⊠ 突然消失，補償性系統可能滿足：

> I_o(t)
> ≫
> I_o(t+Δ t)

例如：

- 關鍵工程師離職；
- 舊 API 被供應商關閉；
- Cron job 停止；
- Excel 操作者請假；
- 外部 SaaS 改版；

- compatibility shim 被「清理」；
- 某個看似無用的 retry 被移除。

  這也是為什麼「刪除看起來醜的東西」並不總是重構。

  它可能是在拆除承重牆。

SECTION

12. 重構前必須先辨認補償

傳統重構問題通常被表述為：

> 哪些程式碼不漂亮？

本文認為更安全的問題是：

> 這段不漂亮的結構現在替什麼東西承擔責任？

因此，在移除補償 C 前，需要知道：

> Effect(C)

以及：

> Dependents(C)

如果 C 目前實際承擔了責任 R，那麼安全移除條件至少應為：

> ∃ S'
> :
> R(S') ⊇ R(C)

也就是存在新的正式結構 S'，已經接管補償原本承擔的責任。

否則：

> Remove Compensation
> ≠
> Remove Complexity

而可能只是：

> Remove Compensation
> →
> Expose Latent Failure

SECTION

13. MSSP/FPL 應該如何看待補償？

原始 MSSP/FPL 偏向描述系統本體、核心模組、可選模組、依賴、架構規則與演化條件。

如果要進入動態 MSSP，未來可能需要把「補償」本身變成一等架構物件。

例如：

```
compensation:
 id: user_profile_legacy_adapter
 type: compatibility
 target_gap: legacy_profile_schema
 owner: identity_team
 activation:
  condition: "client_version < 4"
 effects:
  - schema_transform
  - compatibility_preservation
 dependencies:
  - legacy_mobile_clients
 observability:
  metrics:
   - transformed_requests
   - failure_rate
 exit:
  condition: "legacy_client_share < 0.1%"
  review_required: true
```

或者人工補償：

```
compensation:
 id: month_end_manual_reconciliation
 type: operational
 target_gap: payment_settlement_mismatch
 actor: finance_operator
 frequency: monthly
 evidence:
  - reconciliation_report
 risk:
  single_point_of_failure: true
  undocumented_knowledge: medium
 desired_transition:
  - automated_reconciliation
```

這時 MSSP 不再只是說：

> 這個東西「不符合架構」。

而是先問：

> 它為什麼存在？它在補償什麼？誰依賴它？移除它會發生什麼？

這比單純的 linter 更接近智能化架構治理。

SECTION

14. AI 為什麼適合處理這類問題？

傳統靜態工具很擅長判斷 import graph、cycle、schema、type、call relation 與 permission declaration。

但補償性系統大量資訊存在於 git history、issue、incident report、README、comment、runbook、support ticket、deployment log、chat 與 human explanation。

其中很多不是固定格式。

因此動態 MSSP 可以讓 AI 幫忙推斷：

```
CompensationCandidate
=
f(
code,
runtime,
```

但 AI 的結果不應直接等於真相。

更安全的架構是：

> Observation
> →
> AI Hypothesis
> →
> Evidence
> →
> Confidence
> →
> Human/Policy Decision

也就是將 AI 用於稀疏、模糊、跨來源的架構判斷，而不是讓模型直接任意重構系統。

SECTION

15. 邊界：什麼不應被叫做補償性缺陷？

SECTION

15.1 正常 fault tolerance 不是壞設計

retry、redundancy、graceful degradation 等機制可能正是優秀架構的一部分。

本文研究的是它們如何支撐完整性及其代價，而不是把它們污名化。

SECTION

15.2 必要複雜度不是補丁

某些複雜度來自真實世界本身。

金融清算、分散式一致性、法規流程與多租戶權限，本來就不是一行程式可以解決。

SECTION

15.3 人類參與不等於系統失敗

human-in-the-loop 可以是正式設計。

問題在於：

> 人類角色是否被明確建模？

而不是：

> 系統裡有沒有人。

SECTION

15.4 所有補償都不能直接算 technical debt

設計型韌性與正式 compensating control 可能是合理的長期架構。

技術債只是補償形成原因中的一部分。

SECTION

16. 補償性完好命題

SECTION

命題 2：補償性完好命題

對軟體系統 S 而言，其可觀察完整性可以由原生結構與多層補償機制共同形成：

$$
I_o(t) = \Phi(I_s(t),
$$

因此：

> $I_o \approx 1$

並不足以表示：

> $C \approx 0$

反之，一個高度可靠系統也可能故意具有大量健康的補償機制。

真正重要的不是：

> $|C|$

有多大，而是：

- 補償是否被看見；
- 是否有明確責任；
- 是否可驗證；
- 是否存在退出策略；
- 是否形成新的依賴；
- 是否提高二階風險；
- 是否已由臨時支架轉化為有效架構。

SECTION

17. 結論

如果第一篇的問題是：

> 「功能正常能不能證明結構正常？」

答案是不能。

那麼第二篇進一步回答：

> 「結構不完整時，系統為什麼還能正常？」

答案是：

> 因為真正維持系統運作的，往往不只有正式程式結構。

它可能還包括：

> Technology
> +
> Human Skill
> +
> Operations
> +
> Compatibility
> +
> External Services
> +
> Governance

因此：

> 【Visible Integrity
> ≠
> Intrinsic Structural Integrity】

而更接近：

> 【Visible Integrity
> =
> Structural Capability
> ⊗
> Compensation System】

這裡使用 ⊗，是為了強調兩者不是簡單線性相加，而是會彼此作用、放大、遮蔽與產生新風險。

這也使我們得到下一個更深的問題：

> 如果真正維持軟體運作的東西分散在程式碼、人工流程、歷史相容層、雲端平台與未文件化知識中，那麼我們畫在架構圖上的「系統」，到底是不是那個真正運作的系統？

因此下一篇將進入：

SECTION

〈真正的系統在哪裡？宣告架構與有效架構的分離〉

並正式研究：

> $A_{(declared)}$
> $\neq$
> $A_{(effective)}$

的條件、來源與工程後果。

SECTION

參考文獻

- Google. Site Reliability Engineering: Addressing Cascading Failures. https://sre.google/sre-book/addressing-cascading-failures/
- Google. SRE Workbook: Handling Operational Overload. https://sre.google/workbook/overload/
- NIST Computer Security Resource Center. Compensating Controls — Glossary. https://csrc.nist.gov/glossary/term/compensating_controls
- NIST. Special Publication 800-63-4: Identify Compensating Controls. https://pages.nist.gov/800-63-4/sp800-63.html
- Berensson, T., & Hesselgren, M. (2026). Changing systems from within: Workarounds in sociotechnical system transformation. DRS2026. https://doi.org/10.21606/drs.2026.727

- Analysis of enterprise resource planning (ERP) system workarounds with a resilience perspective. Continuity & Resilience Review, 2(2), 2020. https://doi.org/10.1108/CRR-06-2020-0022
- Chu, S., Koe, J., Garlan, D., & Kang, E. (2024). Integrating Graceful Degradation and Recovery through Requirement-driven Adaptation. arXiv:2401.09678.
- Sillito, J., & Kutomi, E. (2020). Failures and Fixes: A Study of Software System Incident Response. arXiv:2008.11192.
- Huang, Z., D'Angelo, M., Miyani, D., & Lie, D. (2017). Talos: Neutralizing Vulnerabilities with Security Workarounds for Rapid Response. arXiv:1711.00795.