真正的系統在哪裡？宣告架構與有效架構的分離

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

https://thisoneisneok.com/html/papers/ois-03.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：03 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

前兩篇分別提出「軟體表觀完好命題」與「補償性完好」：一個軟體系統的外部功能正常，不足以推出其內部結構完整；而當內部結構不足時，技術機制、人類技能、操作流程、相容層、外部平台與治理制度可以共同補償系統，使其維持可觀察的正常運作。

本文進一步追問一個更根本的問題：真正的系統究竟在哪裡？

傳統架構文件、UML、C4、ADR、模組圖與 DSL 所描述的，是設計者宣告或希望存在的系統；來源碼與靜態依賴呈現的是實作的一部分；runtime trace、資料流、部署拓撲、人工 workaround、runbook、Excel、客服處理、歷史 API 相容行為與組織知識，又共同構成另一套實際運作條件。當這些層次不一致時，「架構圖」便不再等同於「真正運行的架構」。

本文提出「宣告架構（Declared Architecture）」與「有效架構（Effective Architecture）」的區分，並將有效架構定義為：在給定時間、環境與工作負載下，實際參與系統功能、約束、恢復、權限、資料轉換與結果產生的全部關係結構。

形式上：

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

並且：

$$
A_{(effective)}(t) = \Psi(A_{(code)}, A_{(runtime)}, A_{(data)}, A_{(ops)}, A_{(human)}, A_{(external)}, A_{(compat)})
$$

本文以 architectural drift／erosion、Software Reflexion Models、architecture recovery、workaround 與 enterprise-system misfit 等既有研究為外部地基，並進一步提出「架構可見域」「有效依賴」「架構殘差」與「宣告—有效差距」等概念。本文的目的不是否定正式架構，而是指出：正式架構若要成為治理工具，就必須與實際運行證據持續對照，而不能被當成一次性完成的真相。

這一命題也直接推進 Dynamic MSSP/FPL：未來的架構管理不應只驗證「程式碼是否符合規格」，而應持續比較「宣告角色」「觀察角色」與「有效角色」，並允許 AI 以多來源證據推斷未被正式建模的架構關係。

關鍵詞：宣告架構、有效架構、architectural drift、architecture recovery、Software Reflexion Models、workaround、socio-technical system、MSSP、FPL、動態架構治理

SECTION

1. 我們畫的系統，真的是那個正在運作的系統嗎？

軟體架構工作常從一張圖開始。

例如：

Client
↓
API Gateway
↓
Order Service
↓
Payment Service
↓
Database

這張圖可能完全合理。

它可以出現在：

- 架構文件；
- README；
- ADR；
- C4 diagram；
- UML；
- FPL/MSSP 規格；
- onboarding 文件；
- 投影片；
- audit 文件。

但假設真實流程其實是：

Client
↓
API Gateway
↓
Order Service
↓
Payment Service
↓

Database
↓
settlement mismatch
↓
nightly reconciliation job
↓
finance operator
↓
Excel correction
↓
manual approval
↓
support script
↓
Database

那麼第二張圖中的哪些部分算「架構」？

如果只把前半段稱為系統，而把後半段視為「營運細節」，我們可能會錯過一個事實：

> 沒有後半段，整個產品未必能長期維持對外承諾。

因此，架構的第一個問題不應只是：

> 系統被設計成什麼？

還應該問：

> 今天真正使它成立的是什麼？

SECTION

2. 宣告架構：系統被描述成什麼

本文將：

> A_d
> =
> A_(declared)

定義為「宣告架構」。

它包含被正式表述、承認或規範化的架構內容，例如：

- module boundary；
- service boundary；
- API contract；
- dependency rule；
- ownership；
- deployment topology；
- data ownership；
- security boundary；
- FMS／SMS／TMS；
- SCL／DMS；
- ADR；
- architecture principle；
- lifecycle rule。

宣告架構回答：

> 我們認為系統應該如何被理解與治理？

它非常重要。

沒有宣告架構，大型系統往往只能依靠局部記憶與逆向理解。

但宣告本身不保證現實服從它。

SECTION

3. 早在 1995 年：高階模型與來源碼之間就可能不一致

Murphy、Notkin 與 Sullivan 在 Software Reflexion Models 中就處理了這個問題。

其基本前提非常直接：

> 工程師用高階模型理解系統，但這些高階模型經常與來源碼不一致。

Reflexion Model 的方法因此不是直接假定高階架構正確，而是：

- 定義 high-level model；
- 將來源實體映射到 high-level model；
- 計算兩者的 correspondence；
- 顯示 convergence、divergence 與 absence。

可抽象成：

> A_(high)
> oversetcomparelongleftrightarrow
> A_(source)

這是一個重要思想：

> 架構模型不是天然真相，而是一個需要被實作證據反覆檢驗的假設。

後續 architectural drift 研究更明確指出，實作會偏離 intended architecture。

因此：

> A_(intended)(t)
> ≠
> A_(implemented)(t)

並不是罕見例外，而是軟體演化中必須持續處理的現象。

SECTION

4. 但「來源碼架構」也不等於完整的有效架構

如果問題只停在：

> A_(declared)
> ≠
> A_(code)

那麼我們只需要 architecture recovery。

但真實系統比這更複雜。

因為來源碼本身仍然看不到：

- runtime feature flag；
- dynamic routing；
- actual traffic；
- runtime-loaded plugin；
- configuration；
- data-dependent behavior；
- queue backlog；
- retry topology；
- external SaaS；
- API caller behavior；
- operator intervention；
- spreadsheet；
- emergency script；
- manual approval；
- undocumented deployment order；

- historical compatibility expectations。

  所以：

> A_(code)
> ≠
> A_(effective)

同樣成立。

這也是為什麼現代 architecture recovery 已經不滿足於單一靜態來源。

2023 年 SARIF architecture-recovery 研究便同時融合 dependencies、code text 與 folder structure，而不是只依賴一種資訊；2026 年的 ArchAgent 又進一步面對大型 legacy、cross-repository、business-aligned architecture recovery。

這代表研究方向本身正在承認：

> 架構訊息分散在多種證據中。

SECTION

5. 有效架構：真正參與結果形成的結構

本文因此定義：

SECTION

有效架構

SECTION

Effective Architecture

令：

> $A_e(t,E,W)$

表示系統在時間 t、環境 E、工作負載 W 下的有效架構。

定義為：

> 所有實際參與系統功能實現、狀態轉移、資料保存、權限控制、錯誤補償、恢復、相容與結果形成的必要關係集合。

概念上：

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

其中：

- A_C：Code Architecture；
- A_R：Runtime Architecture；
- A_D：Data Architecture；
- A_O：Operational Architecture；
- A_H：Human Architecture；
- A_X：External Ecosystem Architecture；

- A_K：Compatibility Architecture。

  它不是簡單集合相加，而是多層關係共同形成的運作圖。

SECTION

6. 七個有效架構層

SECTION

6.1 Code Architecture

> A_C

包括：

- package；
- module；
- class；
- function；
- import；
- interface；
- static dependency；
- generated code。

  這是傳統靜態分析最容易觀察的部分。

SECTION

6.2 Runtime Architecture

> A_R

包括：

- process；
- thread；
- service instance；
- runtime call；
- event；
- message；
- dynamic dispatch；
- feature flag；
- retry；
- failover；
- autoscaling；
- traffic route。

  某些系統在靜態上看起來完全合理，但 runtime 上實際形成熱點、環路或隱性中心。

SECTION

6.3 Data Architecture

> A_D

包括：

- database；

- table；
- schema；
- data ownership；
- read/write path；
- cache；
- replication；
- ETL；
- reconciliation；
- event log；
- derived state。

  模組沒有 import 某服務，不代表它沒有透過資料表直接依賴該服務。

  因此：

> No Code Dependency
> nRightarrow
> No Effective Dependency

SECTION

6.4 Operational Architecture

> A_O

包括：

- runbook；
- cron；

- deployment step；
- rollback；
- manual repair；
- incident procedure；
- reconciliation；
- backup／restore；
- maintenance window。

  如果系統必須依靠這些流程才能長期保持可用，它們就是有效架構的一部分。

SECTION

6.5 Human Architecture

> A_H

包括：

- 誰擁有判斷權；
- 誰知道例外；
- 誰可以批准；
- 誰能手動修資料；
- 哪些知識只存在於某個人；
- 哪些決策依賴 tacit knowledge。

  如果：

> Person P unavailable
> ⇒
> system function fails

那麼 P 目前就是一個系統依賴。

這不是價值判斷，而是架構事實。

SECTION

6.6 External Ecosystem Architecture

> A_X

包括：

- cloud；
- CDN；
- identity provider；
- payment provider；
- third-party API；
- package registry；
- OS；
- browser behavior；
- managed database；
- external agent。

  現代軟體大量「簡潔」是因為把複雜度搬到了外部服務。

  因此局部 repository 的簡潔：

> C_(repo)downarrow

並不一定代表：

> C_(system)downarrow

SECTION

6.7 Compatibility Architecture

> A_K

包含所有由歷史與外部依賴形成的事實契約：

- legacy API；
- old schema；
- undocumented output；
- timing expectation；
- deprecated field；
- bug-compatible behavior；
- version shim。

它們可能沒有出現在「理想架構」中，卻決定什麼可以改、什麼不能改。

SECTION

7. Workaround 證明：正式系統之外可以存在必要工作系統

企業資訊系統研究提供了一個很重要的對照。

當 global enterprise system 與 local reality 不吻合時，員工可能建立並協調 workaround。研究顯示，這些 workaround 即使不符合正式企業 IT policy，也可能讓 essential activities 得以完成，並為企業與客戶創造價值。

因此存在：

$$W_{(formal)} \neq W_{(actual)}$$

其中：

- $W_{(formal)}$：正式工作流；
- $W_{(actual)}$：真正完成工作的工作流。

  如果 workaround 是持續完成業務不可缺少的一環，那麼：

$$W_{(actual)} \subseteq A_e$$

即便：

$$W_{(actual)} \not\subseteq A_d$$

這正是宣告架構與有效架構分離的典型案例。

SECTION

8. 「影子系統」不是外部雜訊，而可能是架構殘差

workaround literature 中還常出現：

- feral systems；
- shadow systems；
- shadow IT。

  這些東西通常是正式資訊系統無法完整支援工作時，由使用者建立的替代工具、流程或資訊管道。

因此本文引入：

SECTION

架構殘差

SECTION

Architectural Residual

令：

> R_A
> =
> A_e - A_d

概念上表示：

> 實際運作中存在，但沒有被宣告架構充分表達的必要結構。

注意這不是嚴格集合運算，而是概念表示。

R_A 可以包含：

- undocumented dependency；
- manual step；
- shadow database；
- spreadsheet；
- private script；
- external API；
- tacit knowledge；
- hidden compatibility constraint；

- runtime-only route。

  如果：

> |R_A|

持續增加，就代表宣告架構對真實系統的解釋力正在下降。

SECTION

9. 另一方向：宣告了但實際不存在

同樣重要的是：

> R_D
> =
> A_d - A_e

它代表：

> 文件或設計聲稱存在，但實際上已經沒有重要效果的架構元素。

例如：

- 文件寫著某服務負責驗證，但實際已被旁路；
- architecture diagram 保留一個已停用 service；
- ADR 宣稱所有資料必須通過某層，但實際新模組直接讀 DB；
- module 被標成 core，實際 runtime 從不載入；
- TMS 被標成 optional，但沒有它系統根本無法完成主要工作。

  因此宣告—有效差距至少有兩個方向：

> Δ_A
> =
> (
> R_A,

也就是：

- Missing-from-declaration
- Missing-from-reality

SECTION

10. 宣告錯了，還是系統演化了？

這裡必須避免一個危險錯誤：

> 只要 A_d ≠ A_e，就認為實作錯了。

不一定。

可能有三種情況。

SECTION

10.1 Implementation Drift

> A_d is still intended

但實作不小心偏離。

這是典型 architectural drift。

SECTION

10.2 Architecture Evolution

現實條件改變後：

> A_e

其實比舊的：

> A_d

更合理。

此時問題不是「把程式碼修回文件」，而是：

> A_d(t)
> →
> A_d(t+1)

也就是更新正式架構。

SECTION

10.3 Emergent Architecture

沒有任何單一設計者明確決定，但系統逐步形成新的有效組織。

此時需要先理解：

> A_e

再判斷是否：

- 正式承認；
- 重構；
- 替換；
- 限制；
- 逐步退出。

因此：

> A_d ≠ A_e

不是 verdict。

它是 investigation trigger。

SECTION

11. 從「Single Source of Truth」改成「Authority + Evidence」

這對 FPL/MSSP 很重要。

最簡單的架構工具很想建立：

> YAML 是唯一真相來源。

這在生成型 greenfield 專案很有效。

但對長期運行系統，如果強制：

> A_d
> =
> absolute truth

就會出現問題。

因為真實 runtime 可能已經變了。

反過來，如果說：

> code is the only truth，

也不完整。

因為 code 看不到全部運行、外部與人類關係。

所以更合理的模型是：

SECTION

Authority + Evidence

宣告架構仍然具有：

> Normative Authority

即：

> 我們希望系統遵守什麼。
>
> 有效架構則由：

> Operational Evidence

建立：

> 系統現在事實上怎麼運作。
>
> 因此：

> A_d
> =
> Normative Model

> A_e
> =
> Empirical Model

架構治理就是持續處理：

> A_d
> ↔
> A_e

的差異。

SECTION

12. Dynamic MSSP：三種角色，而不是一個標籤

這也直接修改 MSSP 的早期思路。

靜態 MSSP 容易寫成：

```
module:
 name: Analytics
 role: TMS
```

未來更合理的模型可能是：

```
module:
 name: Analytics
 declared_role: TMS
 observed_role:
  dependency_centrality: high
  runtime_required: true
  data_authority: partial
  activation_rate: 0.98
 effective_role:
  candidate: SMS
  confidence: 0.86
 evidence:
  - runtime_trace
  - dependency_graph
  - incident_history
  - data_flow
```

因此對模組 M：

> $$R_d(M) = \text{Declared Role}$$

> $$R_o(M,t) = \text{Observed Role}$$

> $$R_e(M,t) = \text{Effective Role}$$

可能存在：

> $$R_d(M) \neq R_e(M,t)$$

這不是立即自動修改的理由。

而是：

> $$\text{Role Divergence} \rightarrow \text{Review}$$

SECTION

13. 有效依賴：比 import graph 更大的依賴概念

Dynamic MSSP 另一個必要擴張，是將 dependency 從靜態引用擴張為「有效依賴」。

對兩個系統元素 X,Y，若移除 Y 會使 X 的有效行為、恢復能力、資料完整性或承諾無法維持，則可定義：

> $$X \text{ oversete} \rightarrow Y$$

表示：

SECTION

Effective Dependency

它可能來自：

- import；
- API；
- DB；

- event；
- config；
- human；
- schedule；
- external service；
- compatibility；
- recovery procedure。

  因此：

> D_e
> ⊇
> D_(static)

通常才是更合理的假設。

SECTION

14. 架構恢復應該從「找模組」升級成「恢復有效系統」

傳統 architecture recovery 常聚焦：

> 從 source code 重建 module／layer／component。

這仍然很重要。

但若本文命題成立，未來更完整的 Architecture Recovery 應該變成：

> Effective Architecture Recovery

資料來源至少包含：

> cal-E
> =
> {

對應：

- code evidence；
- runtime evidence；
- data evidence；
- operational evidence；
- human evidence；
- external evidence；
- compatibility evidence。

2023 年 SARIF 已經展示多來源資訊融合能提高 architecture recovery 準確性；2026 年 ArchAgent 則進一步把 LLM／Agent 引入 legacy architecture recovery。

本文認為下一步還可以再向外擴：

> 不只恢復「程式碼長成什麼」，而是恢復「真正維持產品運作的社會—技術結構」。

SECTION

15. AI 在這裡的作用：不是替架構師猜，而是建立假設

AI 很適合處理這種問題，原因不是 AI 「懂架構」這麼簡單。

而是有效架構證據往往：

- 分散；
- 不同格式；
- 部分衝突；
- 存在自然語言；
- 有時間性；

- 有置信度；
- 不一定能用單一 static rule 表達。

因此可以建立：

> H_A
> =
> f(cal-E)

其中：

> H_A

是 AI 產生的 Architecture Hypothesis。

例如：

> 「PaymentReconcile 看似是 TMS，但在過去 180 天所有結算成功路徑中均被使用，而且三次 incident recovery 都依賴它，因此可能已成為 effective core。」

但系統必須保留：

- evidence；
- confidence；
- contradiction；
- timestamp；
- authority；
- review status。

所以：

> AI Inference
> ≠
> Architecture Truth

更合理的是：

> AI Inference
> =
> Evidence-backed Candidate

SECTION

16. 動態有效架構

有效架構不是固定的。

令：

> $A_e(t)$

代表時間 t 的有效架構。

則：

> $A_e(t_1) \neq A_e(t_2)$

完全可能。

例如：

- 白天與夜間批次不同；
- 正常狀態與 disaster mode 不同；
- feature flag rollout 前後不同；
- Black Friday 與日常負載不同；
- legacy client 尚存與完全淘汰後不同；
- AI Agent 啟用與停用時不同。

因此更完整應寫：

> A_e
> =
> A_e(t,s,w)

其中：

- t：time；
- s：system state；
- w：workload/context。

這意味著：

> 真正的架構不是單張靜態圖，而可能是一族條件化架構。

SECTION

17. 「真實架構」也不是所有東西都塞進圖裡

這裡需要另一個限制。

如果有效架構把所有 OS call、所有人、所有 library、所有網路路由都放進去，架構將失去抽象能力。

所以：

> A_e
> ≠
> Everything

有效架構仍然需要尺度與觀察目的。

可以定義：

> A_e^((q))

其中 q 是 query／viewpoint。

例如：

> $A_e^{(security)}$

強調 authority、data flow、external trust。

> $A_e^{(reliability)}$

強調 fallback、retry、failover、operator。

> $A_e^{(business)}$

強調核心 capability 與人工流程。

因此 Dynamic MSSP 不應試圖建立一張「宇宙總圖」。

它應建立：

> 同一有效系統本體的多重受控投影。

SECTION

18. 宣告—有效差距的四種狀態

本文提出一個簡單治理分類。

SECTION

State A：一致

> $A_d \approx A_e$

宣告與實際基本一致。

SECTION

State B：可接受偏離

> $A_d \neq A_e$

但差異已知、有原因、有 owner、有期限或正式 exception。

SECTION

State C：未知偏離

存在：

> R_A

但團隊尚未知道。

這是最危險的狀態之一。

SECTION

State D：凝固偏離

偏離已長期存在並被其他部分依賴。

此時：

> R_A
> $\rightarrow$
> $A_{(dependency)}$

直接「修回原架構」可能造成破壞。

這將在第 6 篇「補償凝固」正式處理。

SECTION

19. 命題 3：宣告架構—有效架構分離命題

本文提出系列第三個命題。

對長期演化的軟體系統 S：

> 【A_(declared)(S,t)
> nRightarrow
> A_(effective)(S,t)】

且：

> 【A_(code)(S,t)
> nRightarrow
> A_(effective)(S,t)】

有效架構需要透過多來源運行證據重建：

> 【A_e
> =
> Ψ
> (
> A_C,
> A_R,
> A_D,
> A_O,
> A_H,
> A_X,
> A_K
>)】

而架構治理的主要工作不應只是「保護宣告架構」，而是：

> 【Observe
> →
> Compare
> →
> Explain
> →
> Decide
> →
> Update】

SECTION

20. 結論：真正的系統存在於「使結果成立的關係」之中

本文從一個看似簡單的問題開始：

> 真正的系統在哪裡？

答案不是：

> 在 architecture diagram。

也不是：

> 在 source code。

甚至不是：

> 在 runtime trace。

更合理的回答是：

> 真正的系統存在於所有使其承諾、狀態、資料、失敗恢復與外部結果得以成立的有效關係之中。

因此：

> System
> ≠
> Repository

> System
> ≠
> Diagram

> System
> ≠
> Deployment

而是：

> 【System
> =
> Effective Relation Structure】

宣告架構仍然重要，因為它提供規範、方向與治理權威。

但它不能再被理解為：

> 「我們寫下來了，所以系統就是這樣。」

更成熟的架構觀應該是：

> A_d
> ↔
> A_e

宣告模型與有效模型持續互相校正。

這也為下一篇建立了新的問題。

如果一個實際系統的有效架構可以包含大量歷史殘留、workaround、隱性依賴與人工流程，但它仍然能活、能交付、甚至能高速演化，那麼：

> 這些看起來混亂的結構，到底為什麼具有如此強的生存能力？

下一篇將正式進入：

SECTION

〈屎山為什麼沒有死？Big Ball of Mud 作為生存架構〉

並研究：

> Messiness
> ≠
> Non-viability

以及混亂、適應性、局部最佳化與長期存活之間的關係。

SECTION

參考文獻

- Murphy, G. C., Notkin, D., & Sullivan, K. J. (1995). Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering.
- Rosik, J., Le Gear, A., Buckley, J., Babar, M. A., & Connolly, D. (2011). Assessing architectural drift in commercial software development: A case study. Software: Practice and Experience.
- 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.
- Li, R., Soliman, M., Liang, P., & Avgeriou, P. (2022). Symptoms of Architecture Erosion in Code Reviews: A Study of Two OpenStack Projects. Empirical Software Engineering / preprint arXiv:2201.01184.
- Zhang, Y., Xu, Z., Liu, C., Chen, H., Sun, J., Qiu, D., & Liu, Y. (2023). Software Architecture Recovery with Information Fusion. arXiv:2311.04643.

- Correia, F. F., Ferreira, R., Queiroz, P. G. G., Nunes, H., Barra, M., & Figueiredo, D. (2024). Towards Living Software Architecture Diagrams. arXiv:2407.17990.
- 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.
- Wong, L. H. M., Hurbean, L., Davison, R. M., Ou, C. X. J., & Muntean, M. (2022). Working around inadequate information systems in the workplace: An empirical study in Romania. International Journal of Information Management, 64, 102471.
- Pan, R., Mao, B., Ma, T., & Ling, Z. (2026). ArchAgent: Scalable Legacy Software Architecture Recovery with LLMs. arXiv:2601.13007.
- Neo.K / EveMissLab. MSSP / FPL 系列既有架構文件與本系列前兩篇。