從靜態 MSSP 到動態 MSSP：讓架構角色成為可觀察狀態

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

https://thisoneisneok.com/html/papers/ois-11.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：11 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

MSSP 原始方法論的核心價值，在於把大型軟體系統拆分為具有不同責任與治理方式的結構，例如 FMS、SMS、TMS、DMS、SCL、Router 與 Runtime，並進一步透過 FPL／MSSP-Lang 使模組角色、依賴、權限與架構規則變成可讀、可驗證、可生成的工程規格。

然而，本系列前十篇已逐步證明一個靜態分類模型無法完全描述長期演化軟體：宣告架構與有效架構可能分離；workaround 可能凝固為 architecture；technical debt、architecture erosion 與 historical residue 具有不同時間動力；重構可能只轉移複雜度；AI 又能大幅提高 candidate changes 的生成速度。因此，一個模組在設計文件中被標為 TMS，不代表它在今天的 runtime、data flow、incident recovery 或 external contract 中仍然只是 TMS。

本文提出「Dynamic MSSP／動態 MSSP」：將架構角色從固定標籤改為具有宣告、觀察、有效、時間與證據狀態的可觀察變量。對模組 M：

> R_d(M)
> =
> Declared Role

> R_o(M,t,c)
> =
> Observed Role

> R_e(M,t,c)
> =
> Effective Role

其中 t 表示時間，c 表示 context／operating condition。三者可以相等，也可以長期分離：

> R_d(M)
> ≠
> R_o(M,t,c)
> ≠
> R_e(M,t,c)

本文綜合 ArchUnit 類 architecture tests、2024 ICSA 的 continuous conformance、Software Reflexion Models、architecture recovery、models@run.time、MAPE-K／autonomic computing，以及 2026 年 LLM architectural-decision violation detection 等研究，提出 Dynamic MSSP 的五層判斷結構：

- Declared Model：FPL/MSSP 規格與 ADR；
- Deterministic Evidence：AST、bytecode、dependency、schema、policy；
- Runtime Evidence：trace、event、traffic、data flow、incident；
- Contextual Evidence：history、operations、human workflow、compatibility；
- Reasoned Effective Model：由規則、統計與 AI 共同形成的 evidence-backed architecture hypothesis。

  本文特別主張：Dynamic MSSP 不應把 AI 當成 architecture oracle。2026 年針對 980 份 ADR、109 個 repositories 的研究顯示，LLM 對 explicit、code-inferable architectural decisions 的判斷較強，但在 implicit、deployment-oriented 與 organizational knowledge 上明顯較弱。因此最合理的架構不是「AI 取

> 【Deterministic Rules
> +
> Runtime Observation
> +
> AI Sparse Reasoning
> +
> Evidence
> +
> Human/Policy Authority】

Dynamic MSSP 的核心產物不再只是一張 architecture diagram，而是一個隨時間更新、有來源、有信心值、有偏離狀態、有治理事件的 Architecture State Model。

關鍵詞：Dynamic MSSP、FPL、runtime architecture、architecture conformance、models@run.time、MAPE-K、architecture recovery、LLM architecture reasoning、effective architecture、continuous governance

SECTION

1. 靜態 MSSP 的成功，也暴露了靜態 MSSP 的邊界

MSSP 最初最重要的工作，是把「大型專案裡所有東西都混在一起」拆開。

例如：

- FMS：描述系統是什麼、為何存在；
- SMS：不可缺少的核心能力；
- TMS：可選、可替換、按需能力；
- DMS：觀測、診斷、證據；
- SCL：誰能改什麼、怎麼生效、如何回滾；
- Router：依條件選擇工具／模組；

- Runtime：執行允許的 plan、權限與隔離。

  這個方向本身沒有錯。

  問題出現在：

> role 被當成永久本體屬性。

  例如：

```
module:
 id: recommendation
 role: TMS
```

  這句話其實偷偷假設：

> R(M,t□)
> =
> R(M,t□)
> =
> R(M,t□)

也就是 Recommendation 永遠是 TMS。

但真實系統可能不是這樣。

SECTION

2. 一個 TMS 可以活成 SMS

假設一個推薦模組最初只是：

```
optional recommendation
```

所以：

> R_d(M)=TMS

幾年後：

- 首頁 90% 流量依賴它；

- 廣告收入依賴它；
- search ranking 也讀它；
- incident recovery 需要它；
- 沒有它 conversion 大幅崩落。

此時：

> R_o(M,t)

已經顯示：

runtime_required = high
business_criticality = high
dependency_centrality = high
那麼：

> R_e(M,t)

可能已接近 SMS。

如果 MSSP 仍回答：

> 「它在 YAML 裡寫 TMS，所以它就是 TMS。」

那 MSSP 只是另一份過時文件。

SECTION

3. 宣告角色不是錯，而是 normative role

第三篇已建立：

> A_d
> =
> Normative Architecture

> A_e
> =
> Empirical / Effective Architecture

同理，角色也應拆成：

SECTION

Declared Role

> R_d(M)

表示：

> 系統治理上希望這個模組扮演什麼角色。

這仍然非常重要。

因為沒有 declared role，

系統只剩：

> 「現在碰巧怎麼運作。」

那就失去架構治理能力。

所以 Dynamic MSSP 不是消滅宣告角色。

而是停止把：

> R_d

當成唯一真相。

SECTION

4. 觀察角色：它今天實際呈現什麼特徵？

定義：

SECTION

Observed Role

> R_o(M,t,c)

它不是單一標籤，而是一組可觀察特徵。

例如：

> R_o(M,t)
> =
> [
> q_(activation),
> q_(criticality),
> q_(centrality),
> q_(authority),
> q_(state),
> q_(recovery),
> q_(replacement)
>]

其中可包含：

- activation frequency；
- runtime required rate；
- dependency centrality；
- data authority；
- write authority；
- incident criticality；

- recovery dependence；
- substitutability；
- blast radius；
- external contract count。

也就是：

> 先看它實際做了什麼，不急著重新分類。

SECTION

5. 有效角色：最符合現實的 architecture hypothesis

再定義：

SECTION

Effective Role

> R_e(M,t,c)

表示：

> 綜合宣告規格、靜態結構、runtime、data flow、history、operations 與 external dependency 後，目前最能解釋模組實際架構地位的角色。

注意：

> R_e

不是 ground truth。

它是：

SECTION

Evidence-backed Architecture Hypothesis

所以應輸出：

> (R_e,E,q)

其中：

- R_e：角色候選；
- E：evidence set；
- q：confidence。

例如：

```
effective_role:
 candidate: SMS
 confidence: 0.87
evidence:
 - "99.2% checkout paths call this module"
 - "module required in 11/11 recovery incidents"
 - "6 downstream services consume its state"
```

SECTION

6. 三角色模型

因此對模組 M：

> 【cal-R(M,t,c)
> =
> [
> R_d,
> R_o,
> R_e
>]】

可能有：

SECTION

Case A：一致

> $R_d = R_o = R_e$

架構健康。

SECTION

Case B：觀察偏離，但未必需要改架構

> $R_d \neq R_o$

例如 rollout 期間 TMS 使用率突然 100%。

可能只是 temporary state。

SECTION

Case C：有效角色長期偏離

> $R_d \neq R_e$

且：

> $T_{(divergence)} \gg 0$

此時需要 architecture review。

SECTION

7. Role Divergence 不等於 Error

這一點非常重要。

如果：

> $R_d \neq R_e$

不能直接輸出：

ERROR

因為可能是：

- architecture drift；
- legitimate evolution；
- incident mode；
- migration；
- temporary rollout；
- newly discovered requirement；
- compensation solidification。

  因此：

> 【Role Divergence
> →
> Investigation】

而不是：

> Role Divergence
> →
> Immediate Rejection

SECTION

8. ArchUnit：靜態 architecture rule 仍然非常重要

Dynamic MSSP 不應因為「動態」就否定靜態驗證。

ArchUnit 這類工具已經證明：

- package dependency；
- class dependency；
- layer；
- cycle；

  可以被直接寫成 architecture tests。

  例如：

  domain must not depend on infrastructure
  這類規則如果可以 deterministic 判定，

  就應該：

> AI involvement=0

或者接近 0。

這是 Dynamic MSSP 的第一原則：

SECTION

Deterministic First

SECTION

9. Continuous Conformance：架構一致性可以不是 pass/fail

2024 ICSA 的 Continuous Conformance of Software Architectures 提出：

> 將 architecture conformance 表示為 distance function。

而不是只有：

PASS
FAIL

其方法支援：

- multi-level；
- incremental；
- non-blocking；
- partial architecture；

的持續檢查與 restoration。

這和 Dynamic MSSP 非常接近。

定義：

```
δ_A(t)
=
D(
A_d,
A_e(t)
)
```

表示：

SECTION

Architecture Divergence Distance

那麼：

```
δ_A=0
```

表示完全一致。

但：

$$\delta_A > 0$$

不必立刻失敗。

真正重要的是：

$$\frac{d\delta_A}{dt}$$

以及偏離發生在哪些高權重結構。

SECTION

10. 架構一致性也需要權重

例如：

README naming convention violation
和：

payment service bypasses authorization boundary
不能同權。

因此：

$$\delta_A = \Sigma \square w\square \, d\square$$

其中：

- d□：特定架構偏離；
- w□：criticality / authority / risk weight。

  這使 conformance 從：

> 格式正不正確

升級成：

> 結構偏離有多重要。

SECTION

11. Models@run.time：運行時模型不是新概念

models@run.time 研究早已建立一個重要思想：

> 軟體可以在 runtime 維持與自身／環境相關的模型，並利用該模型支援 self-adaptation、assurance 與 uncertainty handling。

這和 Dynamic MSSP 有共同點：

> Architecture Model

不是只在 design-time 存在。

它可以持續與 runtime evidence 對接。

但 Dynamic MSSP 與典型 self-adaptive systems 仍有區別。

SECTION

12. Dynamic MSSP 不等於「系統自動改自己」

典型 self-adaptive architecture 可能：

> Observe
> →
> Analyze
> →
> Plan
> →
> Execute

直接修改 runtime。

Dynamic MSSP 的預設更保守：

> Observe
> →
> Infer
> →
> Explain
> →
> Govern

也就是：

> 先更新對架構的理解，不必自動更新產品。

只有 policy 明確允許時，

才進一步：

> Govern
> →
> Execute

這對大型商業系統更安全。

SECTION

13. MAPE-K 是一個非常好的對照

Autonomic Computing 的 MAPE-K 模型：

- Monitor；
- Analyze；
- Plan；

- Execute；
- Knowledge；

  可以映射 Dynamic MSSP。

SECTION

Monitor

收集：

- AST；
- dependency；
- runtime trace；
- metrics；
- incident；
- data flow。

SECTION

Analyze

比较：

> A_d

與：

> A_e

以及：

> R_d

與：

> R_o

SECTION

Plan

提出：

- retain；
- reclassify；
- migrate；
- refactor；
- deprecate；
- investigate。

SECTION

Execute

由：

- CI；
- developer；
- agent；
- operator；

在 policy 允許下執行。

SECTION

Knowledge

就是：

- FPL IR；
- ADR；
- history；
- evidence；
- exceptions；
- prior decisions。

SECTION

14. Dynamic MSSP 的 Knowledge 不應是單一 snapshot

定義：

> K_A(t)

為 Architecture Knowledge State。

它應該至少包含：

```
K_A
=
[
A_d,
A_e,
R,
E,
H,
X,
```

其中：

- A_d：declared architecture；
- A_e：effective architecture；
- R：role states；
- E：evidence；
- H：history；
- X：exceptions；
- Q：confidence / uncertainty。

  所以：

> K_A(t□)
> ≠
> K_A(t□)

才是常態。

SECTION

15. Architecture Recovery：單一訊號不夠

2023 年 SARIF architecture recovery 研究指出，過往 technique 常只依賴一到兩種資訊。

SARIF 融合：

- dependency；
- code text；
- folder structure；

  並在其 evaluation 中平均比最佳 prior technique 提高 36.1% architecture recovery accuracy。

  這給 Dynamic MSSP 一個重要原則：

SECTION

Evidence Fusion

即：

> $E_{(architecture)}$
> $\neq$
> $E_{(dependency\text{-}only)}$

SECTION

16. Runtime Evidence 必須正式進入 MSSP

若只看 source：

module imported = yes
無法知道：

runtime actually used = ?
因此需要：

> E_R
> $=$
> Runtime Evidence

包括：

- trace；
- span；
- event；
- invocation count；
- traffic；

- latency；
- failure；
- failover；
- dynamic load；
- feature flag state。

  例如：

```
runtime_evidence:
  window: 30d
  invocations: 12_848_201
  critical_paths: 0.93
  failure_dependency: high
  fallback_used: 317
```

  這才讓：

> R_o(M,t)

有實證基礎。

SECTION

17. Data Flow 比 Import Graph 更接近某些真實依賴

假設：

Service A
完全沒有 import：

Service B
但兩者：

read/write same database table
則：

> D_(static)(A,B)=0

但：

> D_(effective)(A,B)>0

所以 Dynamic MSSP 至少需要 typed edges：

- compile；
- runtime；
- data；
- interface；
- event；
- permission；
- deployment；
- test；
- AI-context。

這與 FPL 走向 Architecture IR 的方向一致。

SECTION

18. Human / Operational Evidence 也必須進模型

第二、三篇已經建立：

> A_H

Human Architecture，

以及：

> A_O

Operational Architecture。

所以 evidence 不只來自 machine telemetry。

例如：

```
operational_evidence:
 manual_recovery:
  frequency_90d: 7
  actor: finance_ops
  required_for_settlement: true
```

或者：

```
human_dependency:
 knowledge_holder_count: 1
 documented: false
```

如果不把這些放進 MSSP，

則：

> A_e

永遠是不完整的。

SECTION

19. 2026 LLM architecture violation study：AI 的強項與弱點剛好畫出界線

2026 年 Su 等人研究：

- 980 個 ADR；
- 109 個 GitHub repositories；
- multi-model validation pipeline。

  結果顯示：

  LLM 對：

SECTION

Explicit / Code-Inferable Decisions

具有相當好的 agreement 與 accuracy。

但在：

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

  方面明顯較弱。

  這對 Dynamic MSSP 幾乎是直接的設計指引。

SECTION

20. AI Sparse Reasoning

因此本文提出：

SECTION

AI Sparse Reasoning

AI 不應該介入每一個 architecture check。

可以 deterministic 的：

> D

直接交給：

- parser；
- type system；

- graph algorithm；
- policy engine；
- architecture test。

只有：

> U
> =
> Uncertain / Cross-source Cases

才呼叫 AI。

因此：

> 【AI Cost
> ∝
> |U|】

而不是：

> AI Cost
> ∝
> |Entire Repository|

這同時降低：

- cost；
- latency；
- hallucination surface。

SECTION

21. 三層判斷器

Dynamic MSSP 可以形成：

SECTION

Layer 1：Deterministic Validator

負責：

- syntax；
- declared dependency；
- cycles；
- schema；
- permission；
- version；
- known policy。

輸出：

PASS / FAIL

SECTION

Layer 2：Evidence Analyzer

負責：

- runtime statistics；
- change history；
- dependency centrality；
- incident correlation；
- data-flow frequency。

輸出：

score / trend / anomaly

SECTION

Layer 3：AI Architecture Reasoner

只處理：

- ambiguity；
- historical reason；
- conflicting evidence；
- implicit responsibility；
- likely compensation；
- role reclassification。

輸出：

hypothesis + evidence + confidence

SECTION

22. AI 不能直接把 hypothesis 寫回 truth

這是最重要的治理規則之一。

若：

> $H_{(AI)}$

是 AI hypothesis，

則：

> $H_{(AI)} \neq A_d$

除非經：

> Authority

批准。

因此：

> AI Suggestion
> →
> Review
> →
> Architecture Decision

而不是：

> AI Suggestion
> →
> Architecture Truth

SECTION

23. Evidence Packet

每個 Dynamic MSSP 判斷都應帶：

SECTION

Evidence Packet

例如：

```
evidence_packet:
 subject: recommendation-service
 claim:
  "effective role may have shifted from TMS to SMS"
 evidence:
  static:
```

```
  - downstream_dependents: 18
 runtime:
  - activation_rate_30d: 0.992
 incident:
  - required_in_recovery: 5/5
 data:
  - authoritative_write: true
counterevidence:
 - "system has degraded mode without recommendation"
confidence: 0.84
generated_at: 2026-08-01T10:30:00+08:00
```

架構判斷因此變成：

> Claim
>
> +
>
> Evidence
>
> +
>
> Counterevidence
>
> +
>
> Confidence

SECTION

24. Confidence 不能等於 Probability of Truth

這裡需要避免過度數學化。

如果：

```
confidence: 0.84
```

它不應自動被解讀成：

> 84% 機率一定正確。
>
> 它可以只是：

SECTION

Decision Confidence

根據：

- evidence coverage；
- source reliability；
- consistency；
- freshness；

  形成的治理指標。

  除非未來建立真正校準過的 probabilistic model。

SECTION

25. 時間窗口：角色不能被一次尖峰改寫

如果一個 TMS 因促銷活動：

7 days activation = 100%
不能立刻改成 SMS。

因此：

> R_o(M,t,c)

必須具有：

SECTION

Observation Window

例如：

- 24h；
- 30d；
- 180d；

- incident mode；
- seasonal mode。

  並比較：

> R_o^(30d)

與：

> R_o^(180d)

避免短期事件誤導架構。

SECTION

26. Contextual Role

這比時間更進一步。

同一模組可能：

在正常模式：

> R_e(M,normal)=TMS

在 disaster recovery：

> R_e(M,DR)=SMS

因此角色應該允許：

> 【R_e(M,t,c)】

而不是：

> R_e(M)

這是 Dynamic MSSP 真正的「Dynamic」。

SECTION

27. 防止 Role Flapping

如果：

今天 TMS
明天 SMS
後天 TMS
架構治理會失去意義。

所以需要：

SECTION

Role Hysteresis

只有當：

$$q>q_{min}$$

且：

$$T_{persistence}>T_{min}$$

才觸發：

ROLE_REVIEW_REQUIRED
而不是每次 telemetry 變化都重新分類。

SECTION

28. Role Lifecycle

因此可以建立：

DECLARED

→ OBSERVED_DIVERGENCE
→ CANDIDATE_SHIFT
→ REVIEW
→ ACCEPTED
→ MIGRATING
→ STABLE
或者：

CANDIDATE_SHIFT
→ REJECTED
→ EXCEPTION_RECORDED
也就是角色本身具有 lifecycle。

SECTION

29. Dynamic MSSP 不只是監控，而是 Architecture State Machine

由此，MSSP 可以從：

SECTION

Classification Scheme

升級為：

SECTION

Architecture State Machine

對每個 entity：

> M□

維持：

> S□(t)
> =
> [

其中：

- R：role；
- B：structural burden；
- C：compensation；
- E：evidence；
- q：confidence。

SECTION

30. 全系統狀態

整個軟體則可寫成：

> 【cal-S(t)
> =
> (
> A_d,
> A_e,
> R,
> B,
> C,
> D,
> E,
> Q
>)】

其中：

- A_d：declared architecture；
- A_e：effective architecture；
- R：role states；

- B：structural burdens；
- C：compensation states；
- D：dependency graph；
- E：evidence；
- Q：confidence / uncertainty。

這就是 Dynamic MSSP 的最小狀態模型。

SECTION

31. 事件驅動更新

不需要每秒重新分析整個 repository。

更新可以由：

SECTION

Architecture Events

觸發。

例如：

- PR merged；
- new dependency；
- schema changed；
- incident；
- ownership changed；
- service retired；
- traffic pattern changed；

- feature flag permanent；
- manual workaround repeated；
- external API deprecated。

因此：

> Event
> →
> Δ cal-S

SECTION

32. 這正好符合 AI Sparse Reasoning

只有：

> Δcal-S

涉及模糊區域時，

才呼叫 AI。

因此：

known rule
→ deterministic
runtime threshold
→ statistical
ambiguous structural meaning
→ AI reasoning
這比：

> 每次讓大模型讀完整 repository 再問「架構健康嗎？」

有效得多。

SECTION

33. Dynamic MSSP 與 Continuous Conformance 的差異

Continuous Conformance 主要問：

> 當前 architecture 與 reference architecture 差多少？

Dynamic MSSP 還要再問：

> 這個差距代表 violation、evolution、compensation、context shift，還是 declared model 已經過時？

因此：

> $\delta_A > 0$

只是一個 signal。

後續分類函數：

> $\kappa(\delta_A, E, H, C)$

才決定治理語義。

SECTION

34. Dynamic MSSP 與 Models@run.time 的差異

Models@run.time 強調：

> runtime model 被系統本身用來適應。

Dynamic MSSP 更偏：

> runtime evidence 被架構治理系統用來維持對有效架構的理解。

因此：

> Runtime Model
> →
> Self-Adaptation

不是必須。

可以先：

> Runtime Evidence
> →
> Architecture Governance

這使它更容易導入既有大型軟體。

SECTION

35. Dynamic MSSP 與 observability 的差異

Observability 回答：

> 系統現在發生什麼？

Dynamic MSSP 要回答：

> 這些運行事實對架構意味著什麼？

例如 trace 告訴我們：

A calls B 99.9% of requests

Dynamic MSSP 再推：

> B 是否已從 optional dependency 變成 effective core dependency？

所以：

> Telemetry
> ≠
> Architecture Meaning

需要中間推理層。

SECTION

36. 多觀察者原則

2026 年 edge inference observability 研究提供一個有趣提醒：

單一 software-reported timing source 可能顯示正常，

外部 observer 卻能看到不同 failure mode。

這支持：

SECTION

Multi-Observer Principle

也就是：

> 【No Single Evidence Source
> =
> Architecture Truth】

所以 Dynamic MSSP 應允許：

- source；
- runtime；
- external monitoring；
- human report；

彼此驗證。

SECTION

37. Runtime Conformance：AI Agent 時代更重要

2025 年 runtime monitor 研究已開始利用：

- LLM instrumentation；
- execution logs；
- conformance checking；

比較 design-time model 與 runtime control flow。

2026 年 AI-agent runtime compliance 研究也進一步將 policy predicate 套在 agent execution traces 上，並在 runtime 攔截違規 tool invocation。

這說明：

> Design-time rule + Runtime trace + Enforcement

正在成為 AI 系統治理的重要方向。

Dynamic MSSP 可以把這種思想擴張到 architecture role 與 effect boundary。

SECTION

38. Authority 仍然必須獨立建模

即使：

> $R_e(M)=SMS$

也不代表：

> AI 可以自動改 declared role。

真正決定：

> A_d

的是：

SECTION

Architecture Authority

例如：

- module owner；
- architecture board；
- policy；
- CI gate；
- designated AI governor；
- human reviewer。

所以：

> Evidence
> →
> Decision

中間必須存在：

> Authority

SECTION

39. FPL 的角色：把 Dynamic MSSP 編譯成工具鏈

Dynamic MSSP 是方法論與狀態模型。

FPL 則負責把它轉成：

- spec；
- IR；
- validator；
- graph；
- diff；
- evidence schema；
- runtime adapter；
- Agent task。

所以：

> 【Dynamic MSSP
> →
> FPL IR
> →
> Tooling / Runtime / Agents】

SECTION

40. 一個 Dynamic FPL 示例

```
module:
 id: recommendation
 declared:
```

```
  role: TMS
  authority:
   reads:
    - user_profile
   writes:
    - recommendation_cache
 observations:
  window: 30d
  runtime:
   activation_rate: 0.992
   critical_path_rate: 0.941
  dependency:
   downstream_dependents: 18
  incidents:
   recovery_required: 5
 effective:
  role_candidate: SMS
  confidence: high
  reasons:
   - "near-universal activation"
   - "high downstream dependence"
   - "required during incident recovery"
 governance:
  status: ROLE_REVIEW_REQUIRED
  auto_reclassify: false
```

這就是從：

```
role: TMS
```

進化成：

> 有宣告、有觀察、有證據、有推論、有治理狀態的 architecture entity。

SECTION

41. Dynamic MSSP 的核心輸出不是分數，而是可行動差異

最有價值的 output 不應是：

```
Architecture Health = 74
```

而是：

```
Declared TMS behaves as effective core.
Evidence persisted for 90 days.
```

Two new external contracts depend on it.
Removal cost increased.
Architecture review recommended.
因為這能直接支援：

- redesign；
- documentation；
- migration；
- ownership；
- deprecation；
- capacity planning。

SECTION

42. 命題 11：架構角色狀態命題

本文提出系列第十一命題。

SECTION

架構角色狀態命題

軟體模組角色不應被視為永恆固定的離散本體屬性，而應視為具有宣告、觀察、有效、時間、上下文與證據狀態的架構變量：

> 【cal-R(M,t,c)
> =
> [
> R_d,
> R_o,
> R_e,
> E,
> q
>]】

其中：

> R_d

提供治理意圖，

> R_o

提供運行觀察，

> R_e

提供 evidence-backed architecture hypothesis。

因此：

> 【R_d≠ R_e】

本身不是錯誤，

而是：

> 【Governance Event】

SECTION

43. 強版本：架構本身應從名詞變成狀態

傳統架構：

> 「這是我們的 architecture。」

Dynamic MSSP：

> 「這是目前 declared architecture、observed architecture、effective architecture 與它們之間的動態關係。」

因此：

> A

不再只是：

SECTION

Noun

而是：

SECTION

State

甚至：

SECTION

Trajectory

> A(t□)
> →
> A(t□)
> →
> A(t□)

SECTION

44. 從「架構規格」到「架構控制系統」

所以 MSSP 的下一階段不是：

> 寫更多分類規則。
>
> 而是：
>
> 【Architecture Specification
> →
> Architecture Observation
> →
> Architecture Inference
> →
> Architecture Governance】

這使 MSSP 從：

SECTION

Static Methodology

轉成：

SECTION

Architecture Control System

但又保持一個重要限制：

> 控制系統不等於無限制自動控制。
>
> 預設仍應是 evidence-first、authority-aware、reversible。

SECTION

45. 結論：Dynamic MSSP 的真正核心是「承認架構會被現實改寫」

靜態 MSSP 問：

> 這個模組是 SMS 還是 TMS？

Dynamic MSSP 問：

> 我們原本宣告它是什麼？今天觀察到它在做什麼？它實際上已經變成什麼？根據什麼證據？誰有權決定是否正式承認這個變化？

因此：

> 【Declared
> ≠
> Observed
> ≠
> Effective】

是被允許、被追蹤、被治理的狀態，

而不是架構模型必須假裝不存在的例外。

最終，Dynamic MSSP 的價值不是：

> 永遠保持 architecture diagram 不變。

而是：

> 讓架構的變化第一次可以被持續看見、解釋、驗證與治理。

它把前十篇的所有命題真正收回同一個工程框架：

- 表觀完好 → 不只看功能；
- 補償性完好 → 找出支架；
- 有效架構 → 找出真正系統；
- 生存架構 → 理解屎山為何能活；
- 結構負擔 → 分辨 debt／erosion／residue；

- 補償凝固 → 找出 workaround 何時變成 architecture；
- 複雜度轉移 → 追蹤 complexity flow；
- 結構動力學 → 追蹤速度與趨勢；
- 語言歷史妥協 → 承認相容性會塑造結構；
- AI 生成—治理缺口 → 讓治理 throughput 跟上 generation throughput。

  因此 Dynamic MSSP 的最終基本式可以寫成：

> 【cal-S(t)
> =
> (
> A_d,
> A_e,
> R,
> B,
> C,
> D,
> E,
> Q
>)】

而更新規則是：

> 【Event⊠
> →
> Observation
> →
> Evidence
> →
> Inference
> →
> Governance
> →
> Δcal-S】

下一篇，也是本系列最後一篇，將把所有概念統合成完整框架：

SECTION

〈軟體結構動力學：補償負載、架構健康與智能治理〉

第十二篇將正式收斂：

- Structural Burden；
- Compensation Load；
- Effective Architecture；
- Complexity Flow；
- Governance Ratio；
- Architecture State；
- Dynamic MSSP；
- FPL；
- AI-assisted governance；

  並嘗試給出一個可被工程化的 Software Structural Dynamics v1.0。

SECTION

參考文獻

- Bucaioni, A., Di Salle, A., Iovino, L., Mariani, L., & Pelliccione, P. (2024). Continuous Conformance of Software Architectures. IEEE International Conference on Software Architecture (ICSA 2024), 112–122. DOI: 10.1109/ICSA59870.2024.00019.
- ArchUnit. ArchUnit User Guide. https://www.archunit.org/userguide/html/000_Index.html
- Ford, N., Parsons, R., Kua, P., & Sadalage, P. Building Evolutionary Architectures / Architectural Fitness Functions. https://evolutionaryarchitecture.com/

- Blair, G., Bencomo, N., & France, R. B. et al. (eds.). (2014). Models@run.time: Foundations, Applications, and Roadmaps. Springer.
- White, S. R., Hanson, J. E., Whalley, I., Chess, D. M., Segal, A., & Kephart, J. O. (2006). Autonomic Computing: Architectural Approach and Prototype. Integrated Computer-Aided Engineering.
- Zhang, Y., Xu, Z., Liu, C., Chen, H., Sun, J., Qiu, D., & Liu, Y. (2023). Software Architecture Recovery with Information Fusion. arXiv:2311.04643.
- 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.
- Vitale, F., Flammini, F., Caporuscio, M., & Mazzocca, N. (2025). Architecting Software Monitors for Control-Flow Anomaly Detection through Large Language Models and Conformance Checking. arXiv:2511.10876.
- Kahani, N., Barati, M., & Addae, D. (2026). Runtime Compliance Verification for AI Agents. arXiv:2606.19242.
- UK Department for Science, Innovation and Technology / Government Digital Service. (2025). Architectural Decision Record Framework. GOV.UK.
- Murphy, G. C., Notkin, D., & Sullivan, K. J. (1995). Software Reflexion Models: Bridging the Gap Between Source and High-Level Models. ACM SIGSOFT FSE.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–10 篇，以及 MSSP / FPL / Program Ontology 既有研究。