AI 寫得越快，屎山會長得越快嗎？AI 原生開發的新技術債

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

https://thisoneisneok.com/html/papers/ois-10.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：10 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

生成式 AI 與 coding agents 正快速降低程式碼產生、局部修改、文件草擬、測試生成與重構建議的成本。這使一個直覺問題變得迫切：如果程式碼可以比過去快數倍產生，而架構理解、驗證、整合、治理與知識同步的速度沒有同比提高，軟體是否會更快進入 technical debt、architecture erosion、compensation accumulation 與 Big Ball of Mud？

目前的實證結果並不支持「AI 必然提高或降低軟體工程生產力」這種單向結論。Microsoft 2025 年彙整三個企業 field experiments、共 4,867 名開發者，觀察到 AI coding assistant 使用者完成任務數約提高 26%；METR 對熟悉大型開源專案的資深開發者所做的早期 2025 RCT，則觀察到使用 AI 的任務完成時間增加 19%。METR 在 2026 年嘗試以更新模型延續研究，但因愈來愈多開發者不願在無 AI 條件下參與，使樣本選擇偏差加劇，因此認為新實驗已無法可靠估計當下整體 productivity effect。這些差異顯示：AI 對「速度」的影響高度依賴任務、熟悉度、工具世代與工作流程。

相比之下，一個更穩定的趨勢正在出現：軟體工程工作的瓶頸由 creation 向 verification、integration 與 supervision 移動。2026 年縱向研究將這類工作稱為 supervisory engineering work；DORA 2025/2026 研究也將 AI 描述為一種 amplifier——它可以提高 throughput，但若 delivery foundations 不成熟，也可能放大 instability。2026 年一項針對 304,362 個已驗證 AI-authored commits、6,275 個 GitHub repositories 的大型實證研究更指出，AI 引入的 code smells、bugs 與 security issues 中，有 24.2% 在追蹤到 repository 最新版本時仍未消失；該研究仍屬近期 empirical preprint，數字不應被視為所有 AI 工具與所有專案的普世定律，但它已直接證明「AI 生成問題可以在真實 codebase 中形成持久維護負擔」。

本文因此提出「AI 原生結構負擔（AI-Native Structural Burden）」與「生成—治理缺口（Generation–Governance Gap）」。本文不把「AI 寫的 code」本身視為新型技術債，而是主張 AI 會改變負擔的生成速度、分布方式與可見性。當：

> V_(generation)
> >
> V_(verification)

以及：

> V_(change)
> >
> V_(governance)

持續成立時，AI 可能將原本的人類 coding bottleneck 轉化為 verification backlog、context debt、provenance debt、integration debt、specification debt 與 agent-orchestration debt。

本文進一步提出：

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

作為概念性的 AI 生成—治理比率。真正的 AI-native engineering 不應追求最大化 generated code，而應同時提高治理吞吐量，使 AI 成為結構代謝加速器，而不是結構負擔放大器。

關鍵詞：AI coding、coding agents、technical debt、verification debt、software quality、DORA、supervisory engineering、Dynamic MSSP、FPL、生成—治理缺口

SECTION

1. 問題不是 AI 會不會寫 code，而是人類會突然得到多少 code

傳統軟體工程的一個天然限制是：

> $V_{human\ coding}$

有限。

一個工程師一天只能：

- 寫有限程式；
- 開有限 PR；
- 修改有限模組；
- 建立有限服務。

這個限制雖然降低產出速度，

也意外形成一種：

SECTION

Natural Change Throttle

亦即天然變更節流。

但 coding agent 改變了這一點。

同樣時間內，AI 可以：

- 生成多個 implementation；
- 修改大量檔案；

- 建立測試；
- 產生 migration；
- 寫 adapter；
- 補文件；
- 重構；
- 建立全新服務。

  因此：

> V_(generation)uparrow

可能非常快。

真正問題是：

> 架構理解、驗證、整合與治理速度有沒有一起提高？

SECTION

2. AI 生產力研究沒有單一答案

目前不同研究得到不同結果。

Microsoft 2025 年研究結合：

- Microsoft；
- Accenture；
- 一家匿名 Fortune 100 公司；

  三個 randomized field experiments，

  總計：

> N=4,867

名開發者。

合併分析觀察到：

> +26.08%

的 completed tasks，

而較缺乏經驗的開發者有更高 adoption 與 productivity gains。

這說明：

> AI Assistance
> →
> Higher Throughput

在某些真實工作環境中確實成立。

SECTION

3. METR：在熟悉大型 codebase 的資深開發者身上，早期 2025 AI 反而更慢

METR 的 2025 RCT 研究：

- 16 名 experienced open-source developers；
- 246 個真實 tasks；
- 開發者平均對自己的 repositories 有約 5 年經驗；
- repositories 大型且成熟；
- 主要使用 Cursor Pro 與 Claude 3.5／3.7 Sonnet。

  結果是：

開發者原本預測：

> 24%

加速。

做完後仍主觀認為：

> 20%

加速。

但實際測得：

> 19%

slower。

這是一個非常重要的提醒：

> Perceived Productivity
> ≠
> Measured Productivity

但它也不能被泛化成：

> AI coding 一定變慢。

因為研究只描述一個特定時期、特定人群、特定任務環境。

SECTION

4. 2026 METR 更新：AI 進步太快，連實驗設計都開始失效

METR 在 2026 年公布後續實驗狀況時指出：

他們原本從 2025 年 8 月開始使用更新 AI 工具、更多開發者重新測試。

但後來發現：

> 愈來愈多開發者不願意參加「不能使用 AI」的 control tasks。

這造成：

SECTION

Selection Bias

因而使新實驗難以可靠估計當下的 productivity uplift。

這本身就很有意思。

它表示：

> AI coding 已從可選工具逐漸變成部分開發者工作流的一部分。

因此研究問題已開始由：

> 「有 AI vs 沒 AI」

轉向：

> 不同 AI-native workflow 之間有什麼差異？

SECTION

5. 所以本文不使用「AI 一定讓 coding 更快」作為前提

本文需要的前提其實更弱：

> 【AI reduces marginal cost of producing candidate changes】

即：

> AI 降低「提出一個候選 implementation」的邊際成本。

這不等於：

- 候選一定正確；
- integration 更快；
- delivery 更穩；
- 專案總工時下降。

只表示：

> C_(candidate generation)downarrow

而這已足以改變整個軟體工程結構。

SECTION

6. Creation → Verification：瓶頸正在移動

2026 年 Vella 與 Blincoe 的 longitudinal mixed-methods study 觀察到：

- 82% 參與者表示花在 writing code 的時間下降；
- 工作重心由 creation 向 verification activities 移動。

研究因此提出：

SECTION

Supervisory Engineering Work

包括：

- directing AI；
- evaluating output；
- correcting output。

所以：

$$C_{creation}\downarrow$$

但：

$$C_{supervision}\uparrow$$

這正是第七篇「複雜度轉移」的 AI 版本。

SECTION

7. AI 不是消滅工程，而是改變工程負擔的分布

可以將傳統工程粗略表示為：

$$W_{human} = [W_{design}, W_{code}, W_{review}, W_{test}, W_{integration}, W_{operations}]$$

AI 進入後：

$$W_{code}\downarrow$$

但可能：

$$W_{review}\uparrow$$

$$W_{verification}\uparrow$$

> W_(context)uparrow

> W_(governance)uparrow

因此：

> 【Less Human Typing
> ≠
> Less Software Engineering】

SECTION

8. DORA：AI 更像 amplifier，而不是 repair system

DORA 2025 State of AI-assisted Software Development 研究涵蓋近 5,000 名技術專業人士與超過 100 小時 qualitative data。

其核心描述是：

SECTION

AI as an Amplifier

也就是：

> AI 放大高效組織的優勢，也放大低效組織的問題。

DORA 2026 對研究的回顧進一步指出：

AI adoption 與更高 throughput 同時出現，

但在基礎不穩時也可能伴隨：

> Delivery Instabilityuparrow

這和本系列第八篇的：

> ρ_g
> =
> (V_(change))/(V_(governance))

完全可以連接。

如果 AI 主要提高：

> V_(change)

而：

> V_(governance)

沒提高，

則：

> ρ_guparrow

SECTION

9. AI 技術債：真正需要避免概念濫用

很容易直接說：

> AI-generated code = technical debt。

這是不正確的。

AI 生成的 code 可以：

- 很好；
- 很差；
- 比人類更簡單；

- 比人類更安全；
- 或具有完全不同的 defect profile。

因此：

> AI Authorship
> nRightarrow
> Technical Debt

technical debt 仍然需要：

> Future Cost Relation

才能成立。

SECTION

10. 但 AI 的確可以生成長期維護負擔

2026 年 Liu 等人研究：

> 304,362

個 verified AI-authored commits，

來自：

> 6,275

個 GitHub repositories，

涵蓋五種常見 AI coding assistants。

研究以 static analysis 比較 commit 前後，

辨識 AI change 新增的：

- code smells；

- bugs；
- security issues。

結果辨識出：

> 484,606

個 distinct issues。

其中：

> 89.1%

為 code smells。

而每一種研究中的 AI assistant 都有超過：

> 15%

的 commits 至少引入一項 issue。

更重要的是：

> 24.2%

被追蹤的 AI-introduced issues 在 repository 最新 revision 仍存在。

這是目前很重要的 real-world signal。

但它仍是近期 empirical preprint，

不能被直接外推成：

> 24.2% 的所有 AI code 都是永久 technical debt。

合理解讀是：

> AI 引入的問題確實可以在真實 repository 中持續存在，而不是全部在下一輪 review 自動消失。

SECTION

11. AI code 具有不同 defect profile

2025 IEEE ISSRE 的大規模研究比較：

- human-written；
- ChatGPT；
- DeepSeek-Coder；
- Qwen-Coder；

超過：

> 500,000

份 Python／Java code samples。

其結果顯示：

AI-generated code 整體：

- 結構較簡單；
- 更重複；
- 更容易出現 unused constructs；
- 更容易出現 hardcoded debugging；

而 human-written code：

- structural complexity 更高；
- maintainability issues 更集中。

研究同時觀察到 AI-generated code 有較多 high-risk security vulnerabilities。

因此：

> 【AI Code Risk
> ≠
> Human Code Risk】

AI 不只是：

> 比人類好／比人類差。

而可能形成：

> 不同種類的品質負擔。

SECTION

12. 第一種 AI 原生負擔：Verification Debt

本文提出：

SECTION

Verification Debt

當生成變更的速度：

> V_g

高於可靠驗證速度：

> V_v

則形成：

> $Q_v(t)$

即未充分驗證的變更佇列。

若：

> $V_g > V_v$

持續，

則：

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

這不一定是傳統 technical debt。

但如果這些未充分驗證的 change 被 merge、部署並成為後續依賴，

它就可能轉化成：

> $D_{(technical)}$

SECTION

13. 第二種：Context Debt

AI Agent 通常不是在「完整世界模型」中工作。

它可能只看到：

- current files；
- retrieved docs；
- selected tests；
- prompt；
- current branch。

但真正有效架構還包括：

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

如果 AI 看不到：

- historical reason；
- external consumer；
- manual workflow；
- hidden compatibility；
- runtime traffic；
- incident history；

就可能生成：

> Locally Correct

但：

> Globally Misaligned

的修改。

本文稱：

SECTION

Context Debt

即：

> 系統必要上下文沒有被持續外部化、索引與提供給生成者，因此每次 AI change 都需要重新猜測架構。

SECTION

14. Context Debt 不是 AI 的 hallucination 問題而已

如果一個系統的重要規則只存在於：

> 「老王知道。」

那麼人類新人也會犯錯。

AI 只是把這個問題放大。

因此：

> D_(context)

真正表示：

> K_(required)
> -
> K_(available)

的差距。

AI-native system 越依賴 agents：

> D_(context)

越重要。

SECTION

15. 第三種：Provenance Debt

AI 生成大量 artifact 後，

如果系統無法回答：

- 這段 code 為什麼改？
- 根據哪個 requirement？
- 哪個 Agent 產生？
- 使用哪個 context？
- 哪些 tests 通過？
- 哪些 evidence 支持？
- 哪個 human review？

就形成：

SECTION

Provenance Debt

記為：

> D_p

它不一定影響今天執行，

但會提高：

> $C_{future\ understanding}$

與：

> $C_{incident\ reconstruction}$

因此非常符合 technical debt 的跨時間成本性質。

SECTION

16. 第四種：Specification Debt

AI 很擅長：

> 根據 prompt 寫一個東西。

但如果 prompt 本身只有：

幫我做登入系統

AI 可以產生大量 implementation。

真正缺少的是：

- authority；
- state；
- failure；
- security；
- recovery；
- data ownership；
- lifecycle；
- verification criteria。

因此：

> C_(implementation)downarrow

會讓：

> C_(underspecification)

變得更明顯。

本文稱：

SECTION

Specification Debt

即：

> 系統要求沒有被形式化到足以支持大規模自動生成，卻已經開始大量生成 implementation。

SECTION

17. 第五種：Integration Debt

AI 可以同時生成：

```
service A
service B
service C
```

每個看起來都合理。

但整合需要處理：

- contract；
- version；
- retry；
- transaction；
- state；
- permission；
- deployment；
- observability。

因此：

> $N_{\text{(generated components)}} \uparrow$

可能使：

> $N_{\text{(integration surfaces)}} \uparrow$

更快。

如果 component generation：

> $O(n)$

而 pairwise integration surface 在最壞情況：

> $O(n^2)$

那麼：

> AI 可以非常快地生成比團隊能整合更多的組件。

這就是：

SECTION

Integration Debt

SECTION

18. 第六種：Orchestration Debt

Agentic development 不只產生 code。

它還引入：

- planner；
- coding agent；

- test agent；
- review agent；
- security agent；
- deployment agent。

  如果這些 Agent：

- 權限不清；
- responsibility overlap；
- context 不一致；
- retry rule 不一致；
- output authority 不清；

  則：

> C_(agent coordination)uparrow

本文稱：

SECTION

Orchestration Debt

這是 AI-native software 才特別突出的負擔。

SECTION

19. 第七種：Supervision Debt

如果組織說：

> AI 產生的 code 全部要人工 review。

但 AI generation throughput：

> 10×

增加，

human review capacity：

> 1.5×

增加，

則：

> Q_(review)uparrow

最後會出現兩個可能：

- merge queue 越來越長；
- review quality 下降。

因此：

SECTION

Supervision Debt

不是：

> 人類懶得 review。

而是：

> 監督能力沒有跟 generation capacity 同步擴張。

SECTION

20. AI 原生結構負擔向量

因此本文提出：

> 【B_(AI)
> =
> [
> D_v,
> D_c,
> D_p,
> D_s,
> D□,
> D_o,
> D_(sup)
>]】

其中：

- D_v：Verification Debt；
- D_c：Context Debt；
- D_p：Provenance Debt；
- D_s：Specification Debt；
- D□：Integration Debt；
- D_o：Orchestration Debt；
- D_(sup)：Supervision Debt。

  這些不是全部都已經是既有標準 technical-debt taxonomy。

  它們是本文為 AI-native engineering 提出的分析維度。

SECTION

21. 這些負擔會轉化成傳統 technical debt

例如：

> D_v

未驗證 change 長期存在後，

可能：

> D_v→ D□

Context Debt 造成錯誤 architecture assumption：

> D_c→ E

Provenance Debt 造成未來理解成本：

> D_p→ D□

Integration Debt 造成 adapter proliferation：

> D□→ L_c

所以 AI 原生負擔不是與舊世界完全分離。

它們會流入前九篇建立的結構動力系統。

SECTION

22. AI 可能加速「補償凝固」

以前一個 workaround 要長成 subsystem，

可能需要：

> T=6months

現在 AI Agent 可以：

```
script
→ API
→ UI
→ database
→ deployment
```

在幾天內完成。

因此：

> T_(solidification)downarrow

如果業務立刻使用：

> Dependentsuparrow

那麼：

> Γ_C

可以非常快上升。

所以 AI 不只加快 coding。

它也可能加快：

SECTION

Architectural Solidification

SECTION

23. AI 可能加速 Historical Residue 的產生

AI 讓：

> C_(new artifact)downarrow

因此團隊較容易：

> 不刪舊的，再做新的。

例如：

```
old parser
+
new parser
+
adapter
```

因為新增成本變便宜。

如果：

> C_(addition)≪ C_(removal verification)

則組織自然偏向新增。

因此：

> G_H

即 historical residue generation rate 可能提高。

這是非常值得警惕的 AI-native asymmetry。

SECTION

24. 生成比刪除容易，會形成新的沉積偏差

在成熟系統裡本來就有：

> C_(delete)
> >
> C_(add)

AI 又進一步讓：

> C_(add)downarrow

所以：

> (C_(delete))/(C_(add))
> uparrow

這意味：

> AI 越會生東西，系統越需要專門的淘汰機制。

否則：

> |Artifacts(t)|

會快速增長。

SECTION

25. 生成—治理缺口

本文正式提出：

SECTION

Generation–Governance Gap

令：

> V_(AI)
> =
> V_(AI-assisted change)

而：

> V_G
> =
> V_(verification)
> +
> V_(architecture)
> +
> V_(governance)

則：

> 【G_(AI)
> =
> V_(AI)
> -
> V_G】

如果：

> G_(AI)>0

持續，

則未治理 change backlog 具有累積壓力。

SECTION

26. AI 生成—治理比率

進一步定義：

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

當：

$$\rho_{AI}<1$$

表示 governance capacity 足以吸收 AI 生成速度。

當：

$$\rho_{AI}\approx 1$$

可能達到動態穩態。

當：

$$\rho_{AI}>1$$

長期成立，

則：

$$\frac{dB_{AI}}{dt}>0$$

的風險增加。

SECTION

27. AI-native Big Ball of Mud 的形成路徑

傳統泥巴可能是：

需求
→ 人類 patch
→ 新需求
→ 新 patch
→ coupling

AI-native 版本可能變成：

需求
→ Agent generates solution
→ tests pass
→ merge

→ next Agent sees local code
→ generates another solution
→ local tests pass
→ merge
→ hidden architecture divergence
→ more adapters
→ more generated fixes
這形成：

> Local Correctness
> +
> High Change Velocity
> →
> Global Structural Drift

如果沒有 architecture-aware governance。

SECTION

28. 「測試都過了」也不保證 AI change 沒有架構問題

Test 通常回答：

> observable behavior 是否符合某些 cases？

但前幾篇已經指出：

> I_onRightarrow I_s

所以：

> Tests Pass
> nRightarrow
> Architecture Healthy

AI 特別容易優化：

> 讓當前 test pass。

但可能同時：

- 增加 hidden dependency；
- 複製 state；
- bypass authority；
- 增加 compatibility burden；
- 破壞未⌧ evolvability。

因此 AI QA 不能只增加 tests。

還需要：

SECTION

Structural Verification

SECTION

29. Agentic Refactoring：AI 也可以是治理工具

這一篇不能只談風險。

2025 年 Agentic Refactoring 研究分析：

> 15,451

個 AI-agent-generated refactoring instances，

來自：

> 12,256

個 pull requests。

研究發現：

- agentic refactoring 很常見；
- 26.1% commits 明確以 refactoring 為目標；
- agent 偏好低階、一致性導向的 refactoring；
- maintainability 與 readability 是主要動機；
- 部分 structural metrics 有小幅但顯著改善。

  所以：

> V_(governance)

也可以由 AI 提升。

AI 不只能：

> G_(burden)uparrow

也可以：

> R_(governance)uparrow

SECTION

30. 真正問題不是「要不要 AI」，而是 AI 在分子還是分母

第八篇定義：

> ρ_g
> =
> (V_(change))/(V_(governance))

AI 可以同時作用在 numerator 與 denominator。

如果主要用 AI 生成：

> V_(change)uparrow

則：

> ρ_guparrow

如果同時用 AI 做：

- architecture recovery；
- impact analysis；
- test generation；
- security analysis；
- dependency analysis；
- documentation sync；
- deprecation detection；
- compensation discovery；

則：

> V_(governance)uparrow

可能讓：

> ρ_gdownarrow

因此：

> AI 是讓屎山長更快，還是讓系統代謝更快，取決於它主要被放在哪一側。

SECTION

31. AI-native engineering 的真正競爭可能是治理吞吐量

當 frontier coding agents 都能快速產生 implementation 後：

> $V_{generation}$

可能逐漸商品化。

真正區分團隊的能力會轉向：

> $V_{trusted\ integration}$

即：

SECTION

Trusted Integration Throughput

它包括：

- specification；
- verification；
- architecture；
- security；
- provenance；
- migration；
- observability；

- rollback。

所以 AI 時代真正珍貴的不是：

> 誰一天產生最多 code。

而是：

> 誰一天能安全吸收最多經過驗證的有效變更。

SECTION

32. 從 Lines of Code 轉向 Verified State Transition

Program Ontology 的視角在這裡尤其重要。

如果：

> Program≠ SourceCode

那麼 AI coding 的真正產物也不應該只算：

> LOC

更合理是：

> Intent
> →
> Structured Change
> →
> Verification
> →
> Δ W

所以產能應該衡量：

SECTION

Verified State Transitions

而不是：

SECTION

Generated Tokens

SECTION

33. Dynamic MSSP 的 AI Gate

未來 Dynamic MSSP 可以在 AI Agent change 後輸出：

```
ai_change:
 agent: coding-agent-17
 task: add_payment_retry_policy
 provenance:
  prompt_ref: task-8831
  context_snapshot: ctx-a29d
 verification:
  unit_tests: pass
  integration_tests: pass
  architecture_checks: warning
 structural_effects:
  new_dependencies:
   - payment-core -> retry-policy
  authority_change: none
  compensation_added:
   - retry-on-timeout
 risks:
  retry_amplification: medium
  compatibility_impact: low
 governance:
  review_required: true
  evidence_confidence: 0.86
```

這代表：

> AI 生成的不是「code」。
>
> 而是：
>
> 一個有來源、有證據、有結構影響的 architecture transaction。

SECTION

34. AI Change Budget

如果：

> $V_{generation}$

可以無限上升，

一個成熟組織反而可能需要：

SECTION

AI Change Budget

也就是每個時間窗口只允許：

> $N_{AI\ changes}$

在：

- review capacity；
- test capacity；
- incident capacity；
- architecture capacity；

  能吸收的範圍內。

這與 SRE 的 error budget 很像，

但作用在：

SECTION

Change Absorption Capacity

而不是 availability。

SECTION

35. 不是限制 AI，而是避免治理系統飽和

因此 Change Budget 的目的不是：

> 讓 AI 少做事。
>
> 而是：

$$V_{(accepted)} \leq V_{(trusted\ absorption)}$$

即：

> 不要讓 change ingestion rate 長期高於組織可以可靠吸收的速度。
>
> 如果治理能力提升：

$$V_{(trusted\ absorption)} \uparrow$$

budget 就能提高。

SECTION

36. AI 也可能第一次讓大型系統做到高頻結構治理

傳統 architecture review 的問題是：

> C_(review)

太高。

所以：

半年一次
甚至：

出事再看
很常見。

但 AI 可以每個 PR：

- recover architecture；
- compare FPL；
- inspect dependency；
- check authority；
- search historical reason；
- estimate impact。

於是：

> f_(architecture review)uparrow

可能接近：

> f_(code change)

這會是 Dynamic MSSP 最有價值的用途之一。

SECTION

37. AI 時代真正的新架構不是「AI 幫你寫 code」

更強的 AI-native stack 應該是：

> Intent

↓

> Architecture Context

↓

> Agent Generation

↓

> Deterministic Validation

↓

> AI Structural Review

↓

> Runtime Evidence

↓

> Governed Commit

而不是：

prompt
→ generate
→ tests pass
→ merge

SECTION

38. 命題 10：AI 生成—治理不對稱命題

本文提出系列第十命題。

SECTION

AI 生成—治理不對稱命題

AI coding systems 降低 candidate change 的產生成本：

> $C_{generation}\downarrow$

但並不自動同比降低：

> $C_{verification}$

> $C_{integration}$

> $C_{architecture}$

> $C_{governance}$

因此若：

> $$【V_{AI\text{-}assisted\ change} > V_{trusted\ governance}】$$

持續成立，

則：

> $$【\frac{dB_{AI}}{dt}>0】$$

的風險上升。

SECTION

39. 強版本：AI 是結構放大器

結合 DORA 的 amplifier framing，可得到：

> 【AI
> ≈
> Structural Amplifier】

如果系統已有：

- 清楚 architecture；
- tests；
- observability；
- ownership；
- versioning；
- governance；

AI 可以放大：

> V_(good change)

如果系統已有：

- hidden dependencies；
- vague requirements；
- weak review；
- poor documentation；
- unclear authority；

AI 也可以放大：

> $V_{\text{structural disorder}}$

SECTION

40. 結論：AI 不一定讓屎山長更快，但它讓「長多快」第一次變成治理問題

回到標題：

> AI 寫得越快，屎山會長得越快嗎？

答案是：

> 不必然。

如果 AI 同時提高：

> $V_{\text{generation}}$

與：

> $V_{\text{governance}}$

而且：

> $\rho_{AI} \leq 1$

它甚至可能讓大型 legacy system 第一次擁有足夠高頻的：

- refactoring；
- architecture recovery；
- impact analysis；

- testing；
- documentation；
- migration。

但如果組織只提高：

> $V_{generation}$

則：

> $\rho_{AI}>1$

會形成新的結構負擔累積壓力。

所以真正的問題不是：

> AI 會不會寫壞 code。

而是：

> 我們產生可信變更的速度，能不能追上產生候選變更的速度？

最終，AI-native engineering 的核心指標不應是：

> Lines of Code per Day

甚至不只是：

> Tasks Completed

而更接近：

> 【Verified, Governed, Reversible State Transitions per Unit Time】

只有當這個數量真正提高，

AI 才不只是：

> 更快地寫軟體。

而是：

> 更快地演化軟體，同時沒有比過去更快失去對它的理解與控制。

下一篇將直接把這些命題落回 MSSP：

SECTION

〈從靜態 MSSP 到動態 MSSP：讓架構角色成為可觀察狀態〉

正式建立：

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

以及 Dynamic MSSP 如何利用 runtime、dependency、data flow、history、incident 與 AI evidence 進行持續架構判斷。

SECTION

參考文獻

- Cui, Z. (K.), Demirer, M., Jaffe, S., Musolff, L., Peng, S., & Salz, T. (2025). The Effects of Generative AI on High-Skilled Work: Evidence from Three Field Experiments with Software Developers. Microsoft Research / preprint.
- Becker, J., Rush, N., Barnes, E., & Rein, D. (2025). Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity. arXiv:2507.09089; METR.
- Becker, J., Rush, N., Cunningham, T., Rein, D., & Mahamud, K. (2026). We are Changing our Developer Productivity Experiment Design. METR, February 24, 2026.

- Vella, A., & Blincoe, K. (2026). The Impact of AI Coding Assistants on Software Engineering: A Longitudinal Study. arXiv:2605.23135.
- DeBellis, D., Storer, K., Harvey, N., Beane, M., Edwards, R., Fraser, E., et al. (2025). DORA 2025 State of AI-assisted Software Development Report. Google / DORA.
- DORA. (2026). Balancing AI tensions: Moving from AI adoption to effective SDLC use. DORA Insights.
- Liu, Y., Widyasari, R., Zhao, Y., Irsan, I. C., & Lo, D. (2026). Debt Behind the AI Boom: A Large-Scale Empirical Study of AI-Generated Code in the Wild. arXiv:2603.28592.
- Cotroneo, D., Improta, C., & Liguori, P. (2025). Human-Written vs. AI-Generated Code: A Large-Scale Study of Defects, Vulnerabilities, and Complexity. IEEE ISSRE 2025. DOI: 10.1109/ISSRE66568.2025.00035.
- Butler, J., Suh, J., Haniyur, S., & Hadley, C. (2025). Dear Diary: A Randomized Controlled Trial of Generative AI Coding Tools in the Workplace. ICSE-SEIP 2025.
- Horikawa, K., Li, H., Kashiwa, Y., Adams, B., Iida, H., & Hassan, A. E. (2025). Agentic Refactoring: An Empirical Study of AI Coding Agents. arXiv:2511.04824.
- Stray, V., Brandtzæg, E. G., Wivestad, V. T., Barbala, A., & Moe, N. B. (2025). Developer Productivity With and Without GitHub Copilot: A Longitudinal Mixed-Methods Case Study. arXiv:2509.20353.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–9 篇，以及 MSSP / FPL / Program Ontology 既有研究。