成功軟體為什麼越活越複雜？軟體演化的結構動力學

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

https://thisoneisneok.com/html/papers/ois-08.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：08 / 12

作者：Neo.K

協作整理：Aletheia / GPT

版本：v0.1

日期：2026-08-01

SECTION

摘要

成功軟體為什麼常常在活得越久之後，變得越難修改、越難理解，也累積越多相容層、補償機制與歷史限制？最直覺的答案是「工程師不斷加功能，所以程式越來越亂」，但這個解釋過於簡單。

Lehman 的軟體演化研究早已指出，與現實世界持續互動的 E-type system 若不持續適應，就會逐漸變得不再令人滿意；而系統一旦持續變更，其複雜度傾向增加，除非有專門工程工作投入維持或降低複雜度。後續研究在不同類型專案中對這些法則進行驗證與修正。2025 年一項分析 95 個活躍 Java 專案的資訊熵研究觀察到所有研究對象都持續變化，並從資訊理論角度部分支持 Lehman 的 continuing change 與 increasing complexity。2026 年一項涵蓋 65,987 個 GitHub 專案、約 7.3 TB 資料的大型縱向研究則發現，具有約 700 次以上主分支提交的大型專案群體呈現相對穩定、可辨識的長期演化軌跡，顯示成熟長壽專案的演化並非純粹隨機。

本文在此前「表觀完好、補償性完好、有效架構、生存架構、結構負擔、補償凝固與複雜度轉移」七篇基礎上，提出一個概念性的「軟體結構動力學（Software Structural Dynamics）」模型。核心主張不是「所有成功軟體一定單調變複雜」，而是：

> 成功且長期與現實互動的軟體，會持續承受外部變更、功能增長、相容性、依賴、組織與運行事件所產生的結構壓力；若治理、重構、淘汰與知識外部化的速度低於結構負擔的生成速度，複雜度、補償負載與歷史殘留便會累積。

本文提出「變更壓力」「治理能力」「結構負擔生成率」「治理比率」「補償增長率」「架構漂移速度」等概念，並以：

> ρ_g(t)
> =
> fracV_(change)(t)
> V_(governance)(t)

描述系統的變更—治理張力。當：

> ρ_g>1

持續成立時，系統更容易進入結構負擔淨累積狀態；當：

> ρ_g≈1

則可能維持動態穩態；而只有當治理與淘汰能力長期高於新負擔生成時，系統才有機會真正降低結構負擔。

本文最後指出，Dynamic MSSP/FPL 的核心不應只是替當前系統打「健康分數」，而應追蹤導數與流量：

> (dE)/(dt),
>
> (dD)/(dt),
>
> (dH)/(dt),
>
> (dL_c)/(dt)

以及變更壓力與治理能力的差值。真正的智能架構治理，必須從 snapshot architecture 走向 time-aware architecture。

關鍵詞：軟體演化、Lehman’s Laws、結構動力學、architecture erosion、technical debt、code churn、補償負載、治理能力、Dynamic MSSP、FPL

SECTION

1. 「越活越複雜」不是一句抱怨，而是一個動態問題

許多工程師都有相似經驗：

一個新專案剛開始時：

結構清楚
依賴少
沒有舊版本
沒有 legacy customer
沒有 migration
沒有 compatibility layer

幾年後：

多版本 API
資料遷移
feature flags
compatibility adapters
special cases
legacy clients
manual operations
incident scripts
multiple deployment modes

於是很容易得到：

> 「軟體放久了自然會爛。」

但這句話不夠精確。

軟體不是物理材料，不會因時間本身氧化。

真正發生的是：

> Time
> →
> More Interactions
> →
> More Changes
> →
> More History

所以核心變量不是單純：

> t

而是：

> (dS)/(dt)

即系統隨時間發生多少、什麼性質的變化。

SECTION

2. Lehman 第一法則：不改也會變差

Lehman 的 Continuing Change Law 指出：

> E-type system 必須持續被調整，否則它在使用上會逐漸變得不令人滿意。

E-type system 指的是和現實世界活動、規則與環境持續互動的系統。

例如：

- 金融；
- 商務；
- ERP；
- 醫療；

- 社群平台；
- 作業系統；
- 瀏覽器；
- AI 平台。

  即使 source code 完全不動：

> $\frac{dC_{code}}{dt}=0$

外部世界仍然可能變：

> $\frac{dE_{world}}{dt}\neq 0$

例如：

- OS 更新；
- 法規變更；
- browser 改版；
- 安全漏洞；
- API sunset；
- 客戶需求；
- 新裝置；
- 使用規模改變。

  因此：

> No Code Change
> $\neq$
> No System Change

更準確地說：

> Fit(S,E□)

可能下降。

SECTION

3. 軟體不是固定物，而是環境耦合系統

令：

> S(t)

代表軟體系統，

> E(t)

代表外部環境。

系統的實際有效性可寫成：

> U(t)
> =
> Φ(S(t),E(t))

若：

> S(t)=S(t□)

但：

> E(t)≠ E(t□)

則：

> U(t)

仍可能改變。

所以一個成功產品若想繼續成功，就必須持續：

> S(t)
> → S(t+Δ t)

使其重新匹配：

> E(t+Δ t)

這就是第一個結構壓力來源：

SECTION

Adaptation Pressure

SECTION

4. Lehman 第二法則：改了之後，複雜度又有上升壓力

Increasing Complexity Law 的核心是：

> E-type system 持續修改時，複雜度會增加、演化會更困難，除非投入工作維持或降低複雜度。
> 可概念化為：

> (dC)/(dt)
> =
> G_(change)
> -
> R_(complexity)

其中：

- G_(change)：變更產生的複雜度；
- R_(complexity)：重構、刪除、簡化等複雜度治理。

  若：

> G_(change)
> >
> R_(complexity)

則：

> (dC)/(dt)>0

這不是說每次 commit 都增加複雜度。

而是說長期演化存在：

SECTION

Complexity Growth Pressure

SECTION

5. 近期研究：Continuing Change 仍然非常可見

2025 年一項研究分析 95 個持續維護的 Java 專案，使用 AST 與 token-based entropy 追蹤 source-code information content。

研究發現：

- 所有研究專案都持續發生變化；
- 專案可能有局部穩定期；
- 但沒有研究對象呈現完全不變的總資訊內容；

- 資料大致支持 Lehman 的 Continuing Change；
- 對 Increasing Complexity 也觀察到相符趨勢，但不同專案與時間區間並非完全一致。

  這很重要。

  因為它提醒我們：

> 【Evolution Pressure
> ≠
> Monotonic Complexity Growth】

複雜度可以：

- 上升；
- 下降；
- 被重構；
- 被轉移；
- 被壓縮。

  但若系統持續與世界互動：

> Change Pressure>0

幾乎是常態。

SECTION

6. 2026：65,987 個 GitHub 專案顯示成熟專案存在穩定演化軌跡

2026 年一項大規模縱向研究分析：

> 65,987

個 popular open-source GitHub projects，

資料量約：

> 7.3TB

並涵蓋 85 種程式語言。

研究發現一個有趣的分離：

- 約 700 次以上主分支提交的 10,612 個大型專案；
- 其他較小專案。

  大型專案群體呈現更穩定、可預測的演化趨勢，並對外部事件呈現較強韌性；小型專案則具有更高變異與減速可能。

  這不能直接證明：

> 大型專案都會變複雜。

  但它支持一個重要背景：

> 長壽、大型軟體演化具有可研究的統計結構，而不是完全隨機事件序列。

  因此「結構動力學」是合理的研究方向。

SECTION

7. 成功本身創造更多變更壓力

成功產品通常得到：

> U_(users)uparrow

隨之：

> R_(requirements)uparrow

以及：

> X_(integrations)uparrow

更多人使用，意味：

- 更多 use cases；
- 更多 edge cases；
- 更多第三方整合；
- 更多地區；
- 更多法規；
- 更多資料；
- 更多可靠性要求。

  所以可能形成：

> Success
> →
> Usage
> →
> Requirements
> →
> Changes

這不是成功必然導致爛架構。

而是：

> 成功提高了系統與現實世界的耦合面。

SECTION

8. 成功也會增加相容性表面

第六篇已經討論 Hyrum’s Law 與 implicit contract。

隨著 consumer 數量增加：

> N_(consumer)uparrow

可被依賴的行為集合通常也增加：

> K_(effective)uparrow

於是：

> C_(compatibility)uparrow

可能成立。

也就是：

> 新系統可以自由改。

> 成功舊系統不能自由改。

因為每個歷史行為都可能有人依賴。

SECTION

9. 成功使「刪除」比「新增」困難

新增功能常是：

> S
> →
> S+F

但刪除功能需要證明：

> Dependents(F)=∅

或者完成 migration。

因此：

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

在成熟系統中非常常見。

長期結果就是：

> Addition Rate
> >
> Removal Rate

則系統：

> |S(t)|uparrow

這也是歷史殘留形成的重要動力。

SECTION

10. 結構負擔生成方程

第五篇建立：

> B_s
> =
> [D,E,H,N]

本文將它動態化。

令：

> B(t)

代表可治理結構負擔。

概念上：

> $$\frac{dB}{dt} = G_D + G_E + G_H + G_C - R_B$$

其中：

- G_D：technical debt generation；
- G_E：architecture erosion generation；
- G_H：historical residue generation；
- G_C：compensation burden generation；
- R_B：治理、淘汰、重構與簡化能力。

因此：

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

表示結構負擔淨累積。

SECTION

11. 變更壓力向量

變更不是單一來源。

本文定義：

> P_(Δ)(t)
> =
> [
> P_r,
> P_e,
> P_s,
> P_c,
> P_d,
> P_o,
> P_x
>]

分別代表：

- P_r：Requirement Pressure；
- P_e：Environment Pressure；
- P_s：Security Pressure；
- P_c：Compatibility Pressure；
- P_d：Dependency Pressure；
- P_o：Organizational Pressure；
- P_x：Scale / External Integration Pressure。

  系統真正承受的是：

> P_(Δ)
> =
> Ω(P_(Δ))

而不是單純：

> PM 又加需求。

SECTION

12. 變更速度與治理速度

這裡可以正式建立本篇最重要的比率。

令：

> $V_{change}(t)$

表示結構變更與新負擔產生速度。

令：

> $V_{governance}(t)$

表示：

- architecture review；
- refactoring；
- deprecation；
- migration；
- testing；
- documentation；
- observability；
- dependency cleanup；
- knowledge externalization；

等治理能力。

定義：

SECTION

Governance Ratio

> 【ρ_g(t)
> =
> fracV_(change)(t)
> V_(governance)(t)】

SECTION

13. 三種治理區域

SECTION

13.1 ρ_g<1：治理盈餘區

治理速度高於新負擔生成。

此時：

> (dB)/(dt)<0

具有可能性。

系統有餘力：

- 清債；
- 淘汰；
- 重構；

- 收斂架構。

SECTION

13.2 ρ_g≈1：動態穩態區

新變更不斷產生新負擔，

治理同時消化。

所以：

> B(t)≈bounded

這可能是成熟大型系統最現實的健康狀態。

不是沒有 technical debt。

而是：

> 負擔沒有失控。

SECTION

13.3 ρ_g>1：負擔累積區

變更、補丁、相容、incident 等生成速度高於治理能力。

則：

> (dB)/(dt)>0

長期可能導致：

- architecture erosion；
- compensation load；
- historical residue；

- cognitive load；
- slower change。

SECTION

14. 第一個正回饋：越複雜，越難治理

當：

> Buparrow

通常：

> C_(change)uparrow

修改變慢。

而治理工作本身也需要理解系統：

> V_(governance)downarrow

於是：

> ρ_guparrow

進一步造成：

> Buparrow

形成：

SECTION

Structural Burden Positive Feedback

> Buparrow
> →
> V_gdownarrow
> →
> ρ_guparrow
> →
> Buparrow

這就是屎山「越來越難救」的動態版本。

SECTION

15. 第二個正回饋：治理越慢，workaround 越多

如果正式修復需要：

> T_(repair)=3months

而 workaround：

> T_(workaround)=3hours

在業務壓力下：

> P(choose workaround)uparrow

因此：

> Buparrow
> →
> T_(repair)uparrow
> →
> W_(workaround)uparrow
> →
> L_cuparrow
> →

這與前面：

SECTION

Patch Accumulation Trap

形成動態閉環。

SECTION

16. 第三個正回饋：補償凝固

第六篇建立：

> C⊠→ A_(t+1)

當 workaround 增加：

> N_Cuparrow

其中一部分會被依賴：

> Γ_Cuparrow

然後：

> R_(removal)uparrow

使治理成本進一步增加。

所以：

> Cuparrow
> →
> Γ_Cuparrow
> →
> R_(removal)uparrow
> →

是另一條正回饋。

SECTION

17. 負回饋：成熟工程治理如何穩定系統

系統不是只能越來越糟。

成熟團隊會建立負回饋。

例如：

> Buparrow
> →
> Architecture Reviewuparrow

> Euparrow
> →
> Guardrailuparrow

> L_cuparrow
> →
> Automationuparrow

> Huparrow
> →
> Deprecationuparrow

於是：

> R_Buparrow

抑制：

> (dB)/(dt)

這就是 Lehman 將 software evolution 描述為 feedback system 的重要意義。

SECTION

18. 第八法則：軟體演化本身就是多迴路回饋系統

Lehman 後期的 Feedback System Law 指出：

> E-type system evolution 是 multi-level、multi-loop、multi-agent feedback system。
> 這與本系列非常一致。
>
> 一個需求修改：
>
> user feedback
> → product decision
> → code change
> → runtime behavior
> → incident
> → monitoring
> → support
> → new requirement
> 本身就是 loop。
>
> 因此：

> S(t+1)
> =
> F(
> S(t),
> E(t),
> U(t),
> O(t),
> G(t)
>)

其中：

- E：environment；

- U：user feedback；
- O：operational evidence；
- G：governance。

所以 architecture 不應被建模成一次性靜態物件。

SECTION

19. Code Churn：改動本身具有風險訊號

Nagappan 與 Ball 對 Windows Server 2003 的研究指出，單純 absolute code churn 不一定是好預測器，但 relative code churn measures 能有效預測 defect density；其研究案例中，這組 churn metrics 能以 89% accuracy 區分 fault-prone 與非 fault-prone binaries。

後續 change-burst 研究也發現：

> 某段時間密集、反覆的 change bursts 對 defect-prone component 具有高預測力。

這不能推出：

> commit 越多越危險。

但支持：

> 變更的密度、局部集中與歷史模式本身就是重要結構訊號。

因此 Dynamic MSSP 可以追蹤：

> χ(M,t)
> =
> Change Intensity

對 module M 建立：

> (dR(M))/(dt)

的風險估計。

SECTION

20. 不是所有變更都一樣危險

例如：

類型 A

isolated UI text change
和：

類型 B

shared authorization rule change
即使 LOC 相同：

$$
\Delta \text{LOC}_A = \Delta \text{LOC}_B
$$

其結構影響可能完全不同：

$$
\Delta A_A \ll \Delta A_B
$$

因此變更速度應該是：

$$
V_{(change)} = \Sigma \square w\square \Delta\square
$$

其中：

> w⊠

代表 architectural weight。

這比單純 commit count 更合理。

SECTION

21. 成功軟體的「結構表面積」會變大

令：

> Σ_A(t)

表示：

SECTION

Architectural Surface Area

包含：

- API；
- data contract；
- user-visible behavior；
- deployment；
- integrations；
- operators；
- permissions；
- compatibility expectations。

成功系統通常：

> (dΣ_A)/(dt)>0

至少具有成長壓力。

表面積越大：

> P_(interaction)uparrow

意味更多：

- potential dependencies；
- change impacts；
- compatibility constraints；
- coordination needs。

因此 complexity 不只由 code size 決定。

SECTION

22. 第五法則：熟悉度限制成長速度

Lehman 的 Conservation of Familiarity Law 指出，系統演化的增量成長受到參與者維持對系統熟悉度的限制。

這和前面第四篇提出的：

> K_f
> =
> Familiarity Capital

非常接近。

如果：

> V_(change)

過快，

則：

> K_f

可能下降。

而：

> K_fdownarrow

又會：

> C_(mistake)uparrow

> V_(governance)downarrow

所以：

> 團隊不只受到 CPU、預算與工時限制，也受到「還能不能理解自己的系統」限制。

SECTION

23. 2026 大型研究對熟悉度與組織穩定性的啟示

2026 年 65,987 專案研究特別分析 Lehman 的 Organizational Stability 與 Familiarity 相關現象。

研究者發現：

- 大型、長期專案呈現更穩定的演化趨勢；
- 這種分離不太受專案開始年份與開發年限影響；
- 大型專案對長期外部事件顯示較穩定軌跡。

這至少提示：

> 成熟軟體演化可能形成某種組織—技術穩態。

也就是：

> Long-Term Success

不只是寫很多 code，

而可能是形成：

> Change Capacity
> ≈
> Absorption Capacity

的長期平衡。

SECTION

24. 成功軟體不是「越來越亂」的單向曲線

因此本文必須反對：

> C(t+1)>C(t)

對所有時間都成立。

更合理的是：

> C(t)

呈現：

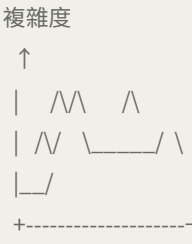
- 增長；
- 重構下降；
- 平台化轉移；

- 大版本清理；
- migration spike；
- 新增功能再增長。

可能形成：

SECTION

Sawtooth Evolution



長期趨勢可能上升，也可能被治理維持在一定區間。

所以：

> Increasing Complexity 是壓力，不是不可違反的物理單調律。

SECTION

25. 動態穩態比「零技術債」更合理

一個大型成熟軟體想做到：

> D=0

> E=0

> H=0

$$L_c=0$$

通常不現實，也不一定值得。

更合理目標是：

$$B(t) \leq B_{max}$$

並：

$$\frac{dB}{dt}\approx 0$$

即：

SECTION

Bounded Structural Burden

這比：

> 「把所有 tech debt 清乾淨」

更接近成熟系統治理。

SECTION

26. 結構速度與結構加速度

如果要把 Architecture 真正動態化，可以進一步定義：

$$X(t)$$

為系統結構狀態。

則：

$$V_s(t) = \frac{dX}{dt}$$

稱為：

SECTION

Structural Velocity

代表系統架構正在多快改變。

再定義：

$$A_s(t) = \frac{d^2X}{dt^2}$$

表示：

SECTION

Structural Acceleration

例如：

> 原本每月新增 2 個依賴，現在每月新增 20 個。

此時即使目前 dependency graph 尚可，

結構加速度已經警告：

$$A_s \gg 0$$

SECTION

27. 架構健康不應只看位置，而要看方向

考慮兩個系統。

System A

> $E=0.7$

但：

> $\frac{dE}{dt}<0$

正在持續修復。

System B

> $E=0.2$

但：

> $\frac{dE}{dt}\gg 0$

正在快速侵蝕。

若只看 snapshot：

> B 比 A 健康。

若看 dynamics：

> B 可能更危險。

所以：

> 【State
> +
> Velocity
> +
> Acceleration】

比單一 health score 更有治理價值。

SECTION

28. Dynamic MSSP 的時間化狀態模型

因此未來 MSSP 不應只儲存：

```
module:
 role: SMS
```

而應開始記錄：

```
module:
 id: payment-core
 role:
  declared: SMS
  effective: SMS
 dynamics:
  dependency_growth_30d: 0.18
  change_intensity_30d: high
  erosion_velocity: medium
  compensation_growth: low
 structural_burden:
  technical_debt: 0.31
  erosion: 0.22
  historical_residue: 0.14
 trend:
  technical_debt: rising
  erosion: stable
  residue: falling
```

重點不是精確小數。

而是：

> 架構第一次具有時間方向。

SECTION

29. FPL IR 也應該能表達演化軌跡

未來 FPL IR 可以包含：

```
evolution:
 observed_window: 90d
 pressures:
  requirements: high
  compatibility: medium
  dependency_updates: high
 capacity:
  feature_delivery: high
  governance: medium
  refactoring: low
 governance_ratio:
  trend: worsening
 alerts:
  - "Change pressure exceeds governance capacity"
  - "Compensation load increasing for 4 consecutive windows"
```

如此：

> FPL

就不只是描述：

> 系統是什麼。

而開始描述：

> 系統正在變成什麼。

SECTION

30. AI 的角色：估計隱性變量

很多重要動態無法只由 AST 得到。

例如：

> P_(organizational)

可能藏在：

- tickets；
- release pressure；
- PR；
- incident；
- Slack；
- deadlines；
- ownership churn。

  AI 可以協助估計：

> hatP_(Δ)(t)

以及：

> hat V_(governance)(t)

但必須保留：

- evidence；
- source；
- confidence；
- time window；
- contradiction。

  即：

> AI Estimate
> ≠
> Ground Truth

SECTION

31. AI 時代可能讓 ρ_g 暫時惡化

AI coding 提升：

> V_(implementation)

很快。

但如果：

- architecture review；
- testing；
- security；
- verification；
- governance；

沒有同步提升：

> V_(governance)

則：

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

反而上升。

這就是：

> AI 寫得越快，不等於系統演化得越健康。

它只是提高 numerator。

因此下一階段真正重要的是：

> AI governance throughput 必須跟 AI generation throughput 一起提升。

SECTION

32. 反過來，AI 也可能提高治理速度

Dynamic MSSP 的希望就在這裡。

AI 可以持續：

- architecture recovery；
- dependency analysis；
- impact analysis；
- test generation；
- documentation sync；
- drift detection；
- compensation discovery；
- historical-context retrieval。

因此：

> V_(governance)uparrow

也是可能的。

如果：

> Δ V_(governance)
> >
> Δ V_(change)

AI 反而可能第一次讓大型系統進入：

> ρ_g<1

的長期治理盈餘。

這才是 AI-native architecture 真正有趣的地方。

SECTION

33. 命題 8：軟體結構動力命題

本文提出系列第八命題：

SECTION

軟體結構動力命題

對長期與現實世界耦合的 E-type software system：

> 【Environmental Change
> +
> Continuing Growth
> →
> Persistent Change Pressure】

而系統的結構負擔變化取決於：

> 【(dB)/(dt)
> =
> G_(burden)
> -
> R_(governance)】

其中：

$$G_{(burden)}$$

是新債、侵蝕、殘留、補償與複雜度轉移所形成的負擔，

$$R_{(governance)}$$

是重構、淘汰、治理、驗證與知識外部化能力。

因此：

$$\rho_g = \frac{V_{(change)}}{V_{(governance)}}$$

可作為概念上的關鍵張力指標。

SECTION

34. 不是時間讓軟體腐化，而是未被治理的變化累積

所以真正的命題不是：

$$\uparrow \Rightarrow \uparrow$$

而是：

$$\int \left(G_{(burden)} - R_{(governance)} \right) dt$$

也就是：

> 當負擔的累積生成長期高於治理消化能力，系統才會越活越沉重。

這比「老軟體自然會爛」精確得多。

SECTION

35. 結論：成功系統真正需要的是持續代謝

一個成功軟體不是完工後放著。

它更像一個持續接收：

- 新需求；
- 新環境；
- 新依賴；
- 新漏洞；
- 新使用者；
- 新相容條件；

的演化系統。

因此：

> Success
> →
> Interaction Surface
> →
> Change Pressure

但是否變成屎山，取決於：

> Governance Capacity

是否跟得上。

所以真正健康的成熟系統，不是：

> 永遠沒有 technical debt。

而是：

> 有能力持續產生、辨識、代謝與淘汰結構負擔。

可以把這稱為：

SECTION

Structural Metabolism

即：

> 【New Structure
> →
> Observe
> →
> Retain / Refactor / Retire】

當代謝速度跟不上生成速度：

> $\rho_g > 1$

沉積開始累積。

當兩者接近平衡：

> $\rho_g \approx 1$

系統進入動態穩態。

當治理長期領先：

> $\rho_g < 1$

則有機會真正降低歷史負擔。

因此：

> 成功軟體越活越複雜，不是不可逃避的命運；真正不可逃避的是持續變化。
>
> 而複雜度最終會不會失控，取決於：

> 【Evolution Rate
> vs.
> Governance Rate】

下一篇將把這條演化視角推到語言層。

SECTION

〈程式語言也是歷史妥協體嗎？從理想語言到生態相容性〉

將研究：

> 如果 application、framework 與 architecture 都會被歷史、相容性與使用者依賴塑形，那麼成熟程式語言本身是否也具有同樣的「演化沉積」？

SECTION

參考文獻

- Lehman, M. M., & Ramil, J. F. (2003). Software Evolution—Background, Theory, Practice. Information Processing Letters, 88(1–2), 33–44. DOI: 10.1016/S0020-0190(03)00382-X.
- González-Barahona, J. M., Robles, G., Michlmayr, M., Amor, J. J., & German, D. M. (2014). Studying the Laws of Software Evolution in a Long-Lived FLOSS Project. Journal of Software: Evolution and Process, 26(7), 589–612. DOI: 10.1002/smr.1615.
- Torres, A., Baltes, S., & Treude, C. (2025). Information-Theoretic Detection of Unusual Source Code Changes. Empirical Software Engineering. DOI: 10.1007/s10664-025-10644-y.

- Szabados, K. (2026). Do Developers Have Agency? A Longitudinal Study Revealing a Separation in How Software Systems Evolve Using 65,987 Popular Open Source Projects on GitHub. Acta Universitatis Sapientiae, Informatica, 18, Article 6. DOI: 10.1007/s44427-025-00019-y.
- 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., Liang, P., Soliman, M., & Avgeriou, P. (2021). Understanding Architecture Erosion: The Practitioners' Perspective. arXiv:2103.11392.
- Nagappan, N., & Ball, T. (2005). Use of Relative Code Churn Measures to Predict System Defect Density. ICSE 2005, 284–292. DOI: 10.1145/1062455.1062514.
- Nagappan, N., Zeller, A., Zimmermann, T., Herzig, K., & Murphy, B. (2010). Change Bursts as Defect Predictors. ISSRE 2010.
- Aversano, L., Bernardi, M. L., Cimitile, M., Iammarino, M., & Montano, D. (2023). Forecasting Technical Debt Evolution in Software Systems: An Empirical Study. Frontiers of Computer Science, 17(3), 173210. DOI: 10.1007/s11704-022-1541-7.
- Molnar, A.-J., & Motogna, S. (2020). Long-Term Evaluation of Technical Debt in Open-Source Software. arXiv:2007.13422.
- Parnas, D. L. (1994). Software Aging. Proceedings of ICSE 1994, 279–287.
- Neo.K / EveMissLab. 《表觀完好系統》系列第 1–7 篇，以及 MSSP / FPL 既有架構研究。