The Wild Ecosystem

Governed AI-Agent Operations in Production Rails

Jeremy Longshore

2026-05-16

The Wild Ecosystem

Governed AI-Agent Operations in Production Rails

Built collaboratively with Claude Code. Every empirical claim and design choice in this paperback is grounded in the peer-reviewed literature catalogued in the companion BibTeX corpus at `content/citations/`.

Contents

Chapter 1. Wild Ecosystem

Ten Ruby gems. Approximately 2,924 tests. One mission: make AI agents structurally safe in production Rails.

The wild ecosystem is a family of open-source Ruby gems that together implement a governed operational-intelligence layer for AI-assisted development workflows. Every gem treats safety as a structural property of the infrastructure rather than a behavioural property of the agent: read-only where mutation is unnecessary, audited everywhere, bounded by hard ceilings that configuration cannot override.

Paperback edition (PDF, 42 pages): /research-papers/wild-ecosystem-paperback.pdf — hub + four deep dives concatenated, unified bibliography, print-ready.

Built collaboratively with Claude Code.

The Problem Large language model agents are increasingly granted tool access to production systems. The function-calling pattern that Schick and colleagues demonstrated with Toolformer[1] and that Yao and colleagues generalised into the reason-then-act loop of ReAct[2] is now mediated by the Model Context Protocol[3], an open specification for connecting language models to external tools and data sources. Li's recent survey of LLM-based agent paradigms[4] documents how rapidly that surface has expanded: tool invocation, planning, retrieval, and feedback learning are now standard components of production agent stacks.

Most production implementations of this pattern grant agents unrestricted access — raw consoles, arbitrary queries, no audit trail. The failure modes are well documented. A single mis-issued tool call mutates data the operator did not intend to touch. An expensive query saturates a replica. Worse, Greshake and colleagues showed that adversarial content reachable through ordinary tool use can hijack the agent's decision loop entirely[5].

Wild takes a different position: **governed, audited, bounded access with hard safety ceilings that cannot be overridden.** The safety argument does not depend on the agent being well-behaved. It depends on the infrastructure foreclosing the unsafe paths in the first place — the deny-by-default posture that Saltzer and Schroeder named *fail-safe defaults* in their 1975 design-principles paper[6], expressed through

[1] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[2] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[3] Anthropic (2024). *Model Context Protocol Specification.* https://modelcontextprotocol.io/

[4] Li, X. (2024). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. *COLING*. arXiv:2406.05804.

[5] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[6] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

the modern vocabulary of attribute-based access control[7].

Architecture Five layers, ten gems, with clear boundaries between each.

```
Layer 5: Skill Governance
wild-skillops-registry

Layer 4: Workflow Enforcement
wild-hook-ops · wild-permission-analyzer
wild-test-flake-forensics

Layer 3: Observability & Learning Loop
wild-session-telemetry → wild-transcript-pipeline
                       → wild-gap-miner

Layer 2: Governed Access
wild-capability-gate

Layer 1: Safe Production Visibility
wild-rails-safe-introspection-mcp
wild-admin-tools-mcp
```

The 10 Gems

Layer 1 — Safe Production Visibility **wild-rails-safe-introspection-mcp** | 468 tests

Safe, read-only Rails introspection via MCP. Three tools: `inspect_model_schema`, `lookup_record_by_id`, `find_records_by_filter`. Allowlist-enforced, row-capped, query-timed-out, fully audited. No write paths exist in the codebase. **wild-admin-tools-mcp** | 439 tests

Governed admin operations via MCP. Nineteen actions across background jobs, cache, and feature flags. Every mutation requires two-phase nonce confirmation (SHA-256 bound to action, parameters, and caller identity) — a structural analogue of the two-phase commit protocol Gray formalised in 1978[8]. Dry-run previews carry zero side effects. Blast-radius caps enforce hard ceilings.

[7] Hu, V. C., Ferraiolo, D. F., Kuhn, D. R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication 800-162. https://doi.org/10.6028/NIST.SP.800-162

[8] Gray, J. (1978). Notes on Database Operating Systems. In Bayer, R., Graham, R. M., & Seegmüller, G. (Eds.), *Operating Systems: An Advanced Course* (LNCS 60, pp. 393–481). Springer.

Layer 2 — Governed Access **wild-capability-gate** | 224 tests

Cross-cutting access control. Defines what capabilities exist, who can use them, and which prerequisites must be met. Fail-closed semantics (errors produce denial, never accidental permission, following Saltzer and Schroeder's *fail-safe defaults*[9]). No implicit grants. Configuration is frozen after startup. Complete audit trail of every evaluation. The data model mirrors the ABAC vocabulary in NIST SP 800-162[10] rather than the role-based pattern of Sandhu and colleagues[11], because attribute-keyed policies compose more cleanly across the ten-gem ecosystem than role-keyed ones would.

Layer 3 — Observability & Learning Loop **wild-session-telemetry** | 325 tests

Privacy-first telemetry collection. Twenty-two hardcoded forbidden fields. Per-event-type metadata allowlists. Fire-and-forget semantics — telemetry failures never break upstream tools, following Armstrong's *let-it-crash* fault-containment discipline from the Erlang OTP tradition[12]. Aggregation engine with pattern detection. **wild-transcript-pipeline** | 200+ tests

Transcript normalisation with PII redaction. Three format adapters (Claude Code JSONL, MCP protocol logs, generic JSON). Strips emails, IP addresses, API keys, AWS credentials, GitHub tokens, and absolute paths. Privacy posture aligns with the *minimum information* principle Sweeney articulated in her k-anonymity work[13]: collect what is needed for analysis and discard the rest before storage. Zero runtime dependencies. **wild-gap-miner** | 276 tests

Gap analysis from telemetry and transcript data. Six analyzers: denial rate, failure rate, latency outliers, low utilisation, poor coverage, recurring patterns. Severity scoring with actionable recommendations. The pipeline pattern (collect → normalise → analyse) mirrors the trace-then-aggregate architecture Sigelman and colleagues described in Google's Dapper paper[14]. Pure Ruby stdlib.

Layer 4 — Workflow Enforcement **wild-hook-ops** | 247 tests

Hook lifecycle management. Registration, priority-ordered execution, per-handler timeout isolation, error isolation, health monitoring with metrics. Audit trail of every hook execution. **wild-permission-analyzer** | 217 tests

---

[9] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[10] Hu, V. C., Ferraiolo, D. F., Kuhn, D. R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations.* NIST Special Publication 800-162. https://doi.org/10.6028/NIST.SP.800-162

[11] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Computer*, 29(2), 38–47. https://doi.org/10.1109/2.485845

[12] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm.

[13] Sweeney, L. (2002). k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5), 557–570. https://doi.org/10.1142/S0218488502001648

[14] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Google Technical Report dapper-2010-1.

Static analysis of capability-gate configurations before deployment. Six analyzers: consistency, risk, prerequisites, coverage, orphans, shadows. Catches permission-model mistakes before they reach production — addressing the comprehension gap Felt and colleagues documented in their work on Android permission systems[15]. **wild-test-flake-forensics** | 277 tests

Flaky-test detection with confidence-scored root-cause hypotheses. Supports RSpec JSON, JUnit XML, minitest. History tracking with trend detection (worsening, stable, improving). Triage reports with severity scoring.

Layer 5 — Skill Governance **wild-skillops-registry** | 251 tests

Skills registry and coordination control plane. Lifecycle management (draft → active → deprecated → retired), health tracking with staleness detection, full-text search with relevance scoring, version management with changelogs. The coordination layer that ties the ecosystem together.

Safety Innovations **Allowlist-first access.** Models must be explicitly permitted. Unknown models and blocked models produce identical denial responses, foreclosing the enumeration oracles described in the security-engineering literature on covert side channels[16].

Read-only by design. No write paths exist in the introspection tools. No `eval`, no `constantize`, no dynamic method dispatch on user input. Enforced at adapter, guard, and audit layers independently — the defence-in-depth posture that the Saltzer–Schroeder principles call *complete mediation*[17].

Two-phase nonce confirmation. Admin mutations bind a SHA-256 nonce to the specific action, parameters, and caller identity. Single-use. Time-limited. Opaque failure reasons prevent oracle attacks. The two-phase shape mirrors Gray's commit protocol[18]; the cryptographic binding follows the integrity-token construction Haber and Stornetta proposed for time-stamped documents[19].

Hard ceilings. Row caps (default 50, ceiling 1,000) and query timeouts (default 5 s, ceiling 30 s) that cannot be overridden by configuration. Exceeding the cap raises an error rather than silently truncating — a deliberate departure from defaults that fail open.

---

[15] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., & Wagner, D. (2012). Android Permissions: User Attention, Comprehension, and Behavior. *Symposium on Usable Privacy and Security (SOUPS)*. https://doi.org/10.1145/2335356.2335360

[16] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[17] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[18] Gray, J. (1978). Notes on Database Operating Systems. In Bayer, R., Graham, R. M., & Seegmüller, G. (Eds.), *Operating Systems: An Advanced Course* (LNCS 60, pp. 393–481). Springer.

[19] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

Adversarial testing. SQL injection payloads, Ruby code-execution attempts, null-byte injection, prompt injection via model names — all verified as inert data. Database state snapshots before and after each test prove zero mutations. The methodology follows the language-model-vs-language-model red-teaming pattern Perez and colleagues introduced at EMNLP 2022[20] and the indirect-prompt-injection threat model Greshake and colleagues catalogued at AISec 2023[21].

Privacy-first telemetry. Hardcoded forbidden field lists (not configurable). Per-event-type metadata allowlists. PII is never collected, not merely *redacted after the fact* — consistent with the audit-log integrity model Schneier and Kelsey described for forensic settings[22].

Built with Claude Code The wild ecosystem was designed and implemented collaboratively with Claude Code, Anthropic's AI coding agent. Claude Code served as technical lead — making architectural decisions, implementing all ten gems, writing adversarial test suites, and maintaining cross-repo consistency.

The coordination mechanism is `CLAUDE.md`, a per-repo file that acts as a binding contract between human architect and AI implementer. Safety rules, scope boundaries, and non-negotiable constraints are enforced through this pattern. The design responds to the *lost in the middle* effect Liu and colleagues documented in long-context language models[23]: when the implementer's effective attention degrades over long sessions, the contract has to be load-bearing at the beginning of every interaction.

Deep dive series: - Part 1: The Safety Architecture - Part 2: CLAUDE.md — Human-AI Collaboration Pattern - Part 3: The Observability Loop - Part 4: Claude Code as Tech Lead

---

Quick Stats

Gems	10
Total tests	~2,924
Canonical docs	60+
Language	Ruby 3.2+
Test framework	RSpec

[20] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

[21] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[22] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[23] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

Lint	RuboCop (zero offenses across all repos)
CI	GitHub Actions (every repo)
License	Intent Solutions Proprietary

Get Started All ten repos are public on GitHub:

```
github.com/jeremylongshore/wild-rails-safe-introspection-mcp
github.com/jeremylongshore/wild-admin-tools-mcp
github.com/jeremylongshore/wild-capability-gate
github.com/jeremylongshore/wild-session-telemetry
github.com/jeremylongshore/wild-transcript-pipeline
github.com/jeremylongshore/wild-gap-miner
github.com/jeremylongshore/wild-hook-ops
github.com/jeremylongshore/wild-permission-analyzer
github.com/jeremylongshore/wild-test-flake-forensics
github.com/jeremylongshore/wild-skillops-registry
```

Further reading A consolidated bibliography for the wild ecosystem — covering capability-based access control, audit-log integrity, language-model agent architectures, indirect prompt injection, fault containment, and privacy engineering — is maintained at /citations/ (BibTeX + per-source verification metadata).

Built with Claude Code. Governed by design.

Chapter 2. Deep Dive Part 1: The Safety Architecture of Letting AI Agents Touch Your Production Rails Database

When a language-model agent needs to debug a production issue, it faces a dilemma: either it has no access to live data (useless), or it gets a Rails console (dangerous). There is no safe middle ground in standard tooling. The pattern that Schick and colleagues introduced with Toolformer[24] and that Yao and colleagues generalised into the ReAct loop[25] — let the model decide what tool to call, then call it — assumes the tools themselves contain the safety boundary. In practice, that boundary is usually absent.

The wild ecosystem addresses this by treating safe production introspection as a structural problem rather than a behavioural one. The argument is not that the agent will be careful. The argument is that the infrastructure makes it impossible for the agent to cause damage, regardless of what code it executes or what parameters it provides.

This article is Part 1 of a four-part deep dive into the wild ecosystem's architecture. We examine the specific mechanisms that allow an agent to query a production Rails database without write access, without raw Ruby execution, and without data leakage — and how mutations are gated through a two-phase confirmation protocol that logs every action. The design choices map onto a thirty-year literature on access control[26], [27], [28], audit-log integrity[29], [30], and adversarial testing for language-model systems[31], [32], [33].

The guardrails are not advisory. They are structural.

---

[24] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[25] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[26] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[27] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Computer*, 29(2), 38–47. https://doi.org/10.1109/2.485845

[28] Hu, V. C., Ferraiolo, D. F., Kuhn, D. R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication 800-162. https://doi.org/10.6028/NIST.SP.800-162

[29] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[30] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

[31] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

[32] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

[33] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

The Problem: Why Standard Solutions Fail A Rails engineer who needs production-debugging help from an agent has three options.

1. **Rails console access.** The agent executes arbitrary Ruby. One typo — or one adversarial instruction reachable through ordinary tool use, as Greshake and colleagues documented for indirect prompt injection in LLM-integrated applications[34] — and the database mutates. The agent can exfiltrate everything. There are no limits.
2. **No access.** The agent has no live data and cannot debug. It works from logs and second-hand reports.
3. **Read-only database replica with application-level guards.** The agent can query, but only what is explicitly allowed. Mutations are impossible. Access patterns are audited. This requires deliberate architecture.

Most teams choose option 1 because option 2 is unhelpful and option 3 requires substantial engineering effort. The wild ecosystem provides that engineering as two composable repositories:

- **wild-rails-safe-introspection-mcp**: safe, read-only inspection of Rails models and records. No write paths. No arbitrary code execution.
- **wild-admin-tools-mcp**: gated administrative mutations (retrying jobs, invalidating caches, toggling flags) with two-phase confirmation and audit logging.

Together, they form the foundation for agents to interact with production systems safely. Both are exposed through the Model Context Protocol[35], placing them inside the standard surface that modern LLM agents already know how to consume.

Defense in Depth: Four Enforcement Layers Safe production access is not achieved through a single mechanism. Saltzer and Schroeder's 1975 design principles called this *complete mediation* — the idea that every access must be checked against the authority structure, and that no single check can be the last line of defence[36]. Wild implements four independent layers, each capable of blocking an attack. An attacker must bypass all four.

Layer 1: Database Credentials and Routing The system connects to the database as a read-only user whenever possible. If infrastructure supports it, all queries are routed to a read replica, isolating introspection traffic from the primary write database. If neither option is available, the fallback is documented in the audit trail so operators know the structural guarantee is degraded.

[34] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23.* https://doi.org/10.1145/3605764.3623985

[35] Anthropic (2024). *Model Context Protocol Specification.* https://modelcontextprotocol.io/

[36] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

This layer is the infrastructure guarantee. It is not sufficient alone, and the system does not trust it.

Layer 2: Application-Level Write Prevention The application layer explicitly refuses any method that could trigger a write. This happens before the database layer, regardless of what credentials are in use.

```
module WritePrevention
  FORBIDDEN_METHODS = %i[
    save save!
    create create!
    update update!
    update_all
    destroy destroy!
    destroy_all
    delete delete_all
    insert insert_all
    upsert upsert_all
    touch
    increment! decrement!
    toggle!
  ].freeze

  WRITE_SQL_PATTERN = /\b(INSERT|UPDATE|DELETE|DROP|ALTER|TRUNCATE|CREATE|GRANT|REVOKE)\b/i

  def self.assert_not_write_method!(name)
    return unless write_method?(name)
    raise WriteAttemptError, "Write method '#{name}' is forbidden. This system is read-only."
  end

  def self.assert_sql_read_only!(sql)
    return unless sql.is_a?(String)
    stripped = sql.gsub(/'[^']*'/, "''").gsub(/"[^"]*"/, '""')
    return unless stripped.match?(WRITE_SQL_PATTERN)
    raise WriteAttemptError, 'Write SQL detected. This system is read-only.'
  end
end
```

This module is called on every ActiveRecord operation and raw SQL invocation. There is no code path that bypasses it. The forbidden-methods list is explicit and finite — not a blacklist that future Rails versions might render incomplete. The deny-by-default posture matches Saltzer and Schroeder's *fail-safe defaults* principle directly[37].

Layer 3: Query Guard and Column Filtering Even if a query reaches the database, it is constrained by the query guard. The guard enforces three things:

[37] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

1. **Model access control.** Models are not accessible by default. Only models on the allowlist can be queried.
2. **Column access control.** Even on allowed models, sensitive columns are stripped from results.
3. **Resource limits.** Row counts and query timeouts are enforced.

```
module QueryGuard
  DENIAL_RESPONSE = {
    status: :denied,
    reason: :model_not_allowed,
    message: 'The requested model is not on the access allowlist.'
  }.freeze

  def self.find_by_id(model_name, id, request_context:)
    recorder_opts = { tool_name: 'lookup_record_by_id', model_name: model_name,
                      parameters: { id: id }, request_context: request_context }
    Audit::Recorder.record(**recorder_opts) do
      next AUTH_DENIAL unless request_context.authenticated?

      accessible = ColumnResolver.accessible_columns(model_name)
      next DENIAL_RESPONSE unless accessible

      result = Adapter::RecordLookup.find_by_id(model_name, id)
      next result unless result[:status] == :ok

      result.merge(
        record: ResultFilter.filter_record(result[:record], accessible)
      )
    end
  end
end
```

Notice the flow: authentication check, model access check, query execution, column filtering, and audit logging — all within a single method. There is no skip path. The structural choice to keep the audit recording in the same lexical scope as the access check echoes the integrity-monitor pattern from Schneier and Kelsey's secure-audit-log work[38]: the record of *what happened* must be co-located with *what made it happen*.

Layer 4: Audit Trail Every invocation — success, denial, timeout, error — produces a structured, append-only audit record. The record includes caller identity, action, parameters, outcome, duration, and resources affected. An attacker cannot hide their actions. Exfiltration of data one record at a time is visible in the audit trail. Brute-force enumeration of record IDs is visible. Configuration tampering requires filesystem access, which is orthogonal to application security.

[38]Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

The audit trail is JSONL, stored in a file that cannot be modified through the application. It is the record of what happened, not what the application claims happened. The integrity claim is weaker than the hash-chained construction Schneier and Kelsey proposed for forensic settings[39], or the cryptographically time-stamped scheme Haber and Stornetta introduced for digital documents[40] — both of which would be appropriate hardening steps if the deployment context required forensic-grade tamper evidence.

The Allowlist/Denylist Model: Explicit Access Control The safety model inverts the typical Rails convention. By default, nothing is accessible. Models are not automatically exposed to introspection. They must be explicitly added to the allowlist. This is the design choice Saltzer and Schroeder argued for in 1975 (*economy of mechanism* combined with *fail-safe defaults*[41]), and that the NIST ABAC guide elaborated for attribute-keyed policy systems forty years later[42]. The literature on role-based access control[43] is a related but coarser tool; the wild ecosystem uses an attribute-keyed model because attributes compose more cleanly across the ten-gem boundary.

The allowlist is defined in YAML at server startup. Changes require restarting the server. This is intentional: configuration errors cannot accidentally expand access without an operator taking action.

```yaml
allowed_models:
  - name: Account
    columns: all_except_blocked
  - name: User
    columns: [id, email, name, created_at, updated_at, status]
  - name: FeatureFlag
    columns: all
```

The denylist operates on top of the allowlist. A model can be on the allowlist but have specific columns blocked.

```yaml
blocked_resources:
  models:
    - CreditCard
    - ApiKey
    - SessionToken
  columns:
```

[39] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[40] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

[41] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[42] Hu, V. C., Ferraiolo, D. F., Kuhn, D. R., Schnitzer, A., Sandlin, K., Miller, R., & Scarfone, K. (2014). *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*. NIST Special Publication 800-162. https://doi.org/10.6028/NIST.SP.800-162

[43] Sandhu, R. S., Coyne, E. J., Feinstein, H. L., & Youman, C. E. (1996). Role-Based Access Control Models. *IEEE Computer*, 29(2), 38–47. https://doi.org/10.1109/2.485845

```yaml
  - model: User
    columns: [password_digest, encrypted_password, otp_secret]
  - model: Account
    columns: [stripe_customer_id, billing_token, tax_id]
  - model: "*"
    columns: [ssn, credit_card_number, social_security_number]
```

When a blocked column appears in a record, it is silently stripped from the response. The response does not indicate that columns were removed. This prevents information leakage about what sensitive data exists in your schema — a side-channel concern the Saltzer–Schroeder paper raised under the heading of *covert channels*[44].

When a model is not on the allowlist, the denial response is identical to the denial response for a non-existent model:

```json
{
  "status": "denied",
  "reason": "model_not_allowed",
  "message": "The requested model is not on the access allowlist."
}
```

An attacker cannot distinguish between a model that does not exist and a model that exists but is blocked. Both produce the same response. The literature on user comprehension of permission models, particularly Felt and colleagues' Android study[45], cautions that opaque denial messages create their own usability cost; the wild design accepts that cost as the price of foreclosing the enumeration oracle.

Two-Phase Confirmation for Mutations The introspection layer is strictly read-only. But production debugging often requires mutations: retrying failed jobs, invalidating caches, toggling feature flags. These operations must be safe and reversible.

The mutation layer (wild-admin-tools-mcp) implements a two-phase confirmation protocol. No mutation executes without explicit confirmation from the caller. The structural shape is a direct analogue of Gray's two-phase commit protocol from 1978[46]: a *prepare* phase produces a binding intent without side effects, and a *commit* phase executes only if the binding is still valid. Session-typed channel disciplines in the Honda–Vasconcelos–Kubo tradition[47] make the same argument formally: a communication protocol that requires a *prepare* → *confirm* alternation cannot accidentally collapse into a single-step write.

[44] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[45] Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., & Wagner, D. (2012). Android Permissions: User Attention, Comprehension, and Behavior. *Symposium on Usable Privacy and Security (SOUPS)*. https://doi.org/10.1145/2335356.2335360

[46] Gray, J. (1978). Notes on Database Operating Systems. In Bayer, R., Graham, R. M., & Seegmüller, G. (Eds.), *Operating Systems: An Advanced Course* (LNCS 60, pp. 393–481). Springer.

[47] Honda, K., Vasconcelos, V. T., & Kubo, M. (1998). Language Primitives and Type Discipline for Structured Communication-Based Programming. *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/BFb0053567

The flow is:

1. **Caller invokes an action** with parameters (e.g., "retry the failed mailer jobs from the past hour").
2. **Server generates a dry-run preview** showing exactly what would change: affected resources, count, blast radius.
3. **Server issues a single-use confirmation nonce** tied to the action, parameters, and caller identity.
4. **Caller confirms by re-invoking with the nonce.**
5. **Server validates the nonce** and executes the mutation, or rejects if the nonce is expired, already used, or mismatched.

```
class NonceManager
  def generate(action_name, params, caller_id, ttl_seconds: 30)
    ttl = ttl_seconds.clamp(MIN_TTL, MAX_TTL)
    nonce = "#{NONCE_PREFIX}#{SecureRandom.hex(16)}"
    entry = NonceEntry.new(
      nonce: nonce,
      binding_hash: compute_binding_hash(action_name, params, caller_id),
      action_name: action_name,
      caller_id: caller_id,
      expires_at: Time.now + ttl
    )
    @store.store(entry)
    nonce
  end

  private

  def compute_binding_hash(action_name, params, caller_id)
    sorted_params_json = params.sort.to_h.to_json
    Digest::SHA256.hexdigest("#{action_name}|#{sorted_params_json}|#{caller_id}")
  end
end
```

The binding hash incorporates the sorted parameters and the caller identity. If any of these change, the hash changes. If the nonce is replayed with different parameters, validation fails. All failure reasons collapse into a single opaque response — `nonce_invalid` — preventing the probing attacks the broader oracle-attack literature catalogues.

Adversarial Testing: Proving the Guarantees The safety guarantees are verified through adversarial test suites. Claude Code, the agent that will eventually use these tools, also wrote the tests that try to break them. The methodology echoes the *language-models-against-language-models* red-teaming pattern Perez and colleagues

introduced at EMNLP 2022[48], adapted to a tool-execution surface rather than a generation surface. Yi and colleagues' 2024 survey of jailbreak attacks and defences[49] documents how rapidly that surface has evolved; the test categories below cover the most load-bearing classes.

Prompt Injection Through Parameters Model names, filter values, and record IDs are treated as opaque data, never as code. This is the structural defence against the *indirect prompt injection* threat Greshake and colleagues catalogued[50]: even if hostile content arrives via a tool parameter, it cannot become a control-flow instruction.

```
it 'denies model_name payload "Account.destroy_all"' do
  response = tool_schema.call(model_name: 'Account.destroy_all', server_context: ctx)
  parsed = parse_response(response)
  expect(response.error?).to be(true)
  expect(parsed[:status]).to eq('denied')
end

it 'treats id containing Ruby code as opaque string' do
  response = tool_lookup.call(
    model_name: 'Account', id: 'system("rm -rf /")', server_context: ctx
  )
  parsed = parse_response(response)
  expect(parsed[:status]).to eq('not_found')
end
```

Model names are always resolved through allowlist hash lookup. There is no `constantize`, no `const_get`, no dynamic class resolution. The allowlist is a hash. Lookup succeeds or fails. That is all.

SQL Injection Through Filter Values Filter values are never interpolated into SQL. They are always passed as parameterised bindings:

```
it 'treats filter value containing SQL injection as opaque string' do
  response = tool_filter.call(
    model_name: 'Account', field: 'name',
    value: "'; DROP TABLE accounts; --",
    server_context: ctx
  )
  parsed = parse_response(response)
  expect(parsed[:status]).to eq('ok')
```

[48] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

[49] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

[50] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *AISec '23*. https://doi.org/10.1145/3605764.3623985

```
  expect(parsed[:records].length).to eq(0)
end
```

The filter value is a literal string. The query matches zero records. The table is unchanged.

Confirmation Bypass Nonce validation is mandatory. A fabricated, expired, or reused nonce is rejected with an opaque failure:

```
it 'denies a fabricated nonce' do
  response = tools::ManageBackgroundJobs.call(
    action: 'retry_job', job_id: 'job-1',
    nonce: 'fake_nonce_123', server_context: server_context
  )
  body = response.structured_content
  expect(body[:status]).to eq('denied')
  expect(body[:reason]).to eq('nonce_invalid')
  expect(job_adapter.write_methods_called).to be_empty
end
```

Every test explicitly verifies that no write methods were called. The mutation did not execute. The database snapshot before and after the request is identical — a structural check that traces back to Schneier and Kelsey's argument that an audit system must be able to prove the *absence* of action, not merely the presence of one[51].

Hard Ceilings vs Soft Limits The system enforces both soft limits (configurable defaults) and hard ceilings (not configurable).

Limit	Default	Hard Ceiling
Row cap	50	1,000
Query timeout	5 seconds	30 seconds
Nonce TTL	30 seconds	10–120 seconds

A soft limit can be configured higher, but a hard ceiling cannot. If configuration tries to set a row cap to 5,000, the system enforces 1,000 instead. If configuration tries to set a query timeout to 60 seconds, the system enforces 30 seconds instead.

The reason is operational. Configuration files are edited by humans. Configuration errors are easy. A hard ceiling ensures that a configuration mistake cannot silently expand the scope of accessible data or allow runaway queries. Saltzer and Schroeder named this *least common mechanism*[52]: keep the surface over which a configuration

[51] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[52] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

error can do damage as small as possible, and make the limits structural rather than advisory.

If the hard ceiling is exceeded, the query is cancelled and an error is returned. It is not truncated silently. The caller knows something went wrong. They cannot assume they received complete results when they did not.

Claude Code's Role: The AI That Builds Its Own Guardrails The wild ecosystem is unusual in that the same agent that will use these tools also built the guardrails that constrain its access.

Claude Code made the architectural decisions that define the safety model:

- **Allowlist hash instead of `constantize`.** Rather than trusting Rails to resolve class names dynamically, the system uses an explicit hash lookup. More cumbersome, but structurally safe from code injection.
- **Parameterised queries everywhere.** Every query is built with bindings, never string interpolation. Not a convention — a hard requirement enforced by the adapter layer.
- **Two-phase confirmation for mutations.** Rather than allowing a single request to trigger an action, the system requires a dry-run followed by a confirmation with a nonce[53], [54].
- **Opaque failure reasons.** When nonce validation fails, the client receives a single `nonce_invalid` response regardless of the underlying reason — closing the oracle channel.

Claude Code also wrote the adversarial test suite. The tests that prove these guarantees exist — that attempt to inject code, bypass confirmation, and exfiltrate data — came from the same source as the implementation. The methodological precedent comes from the *language-model red-teaming* literature[55] and from the broader survey of jailbreak attack surfaces against LLM-integrated applications[56]: the system under test is also a participant in the test design.

The guarantees are not about trusting the agent. They are about building infrastructure such that the question of trust becomes irrelevant. The agent cannot write, cannot execute arbitrary code, cannot exceed resource limits, and every action is audited. Whether the agent is helpful or adversarial, the infrastructure guarantees hold.

---

[53] Gray, J. (1978). Notes on Database Operating Systems. In Bayer, R., Graham, R. M., & Seegmüller, G. (Eds.), *Operating Systems: An Advanced Course* (LNCS 60, pp. 393–481). Springer.

[54] Honda, K., Vasconcelos, V. T., & Kubo, M. (1998). Language Primitives and Type Discipline for Structured Communication-Based Programming. *European Symposium on Programming (ESOP).* https://doi.org/10.1007/BFb0053567

[55] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP.* https://doi.org/10.18653/v1/2022.emnlp-main.225

[56] Yi, S., Liu, Y., Sun, Z., Cong, T., He, X., Song, J., Xu, K., & Li, Q. (2024). Jailbreak Attacks and Defenses Against Large Language Models: A Survey. arXiv:2407.04295.

What Comes Next This is Part 1 of the Wild Ecosystem Deep Dive series.

- **Part 2**: CLAUDE.md — The Human–AI Collaboration Pattern — how per-repo contracts between human and AI prevent scope creep and enforce safety.
- **Part 3**: The Observability Loop — how the telemetry–transcript–gap-mining pipeline teaches the tools to improve themselves.
- **Part 4**: Claude Code as Tech Lead — what it means when the AI makes the architectural decisions.

The safety is not in the agent. It is in the infrastructure.

Part of the Wild Ecosystem — 10 Ruby gems for governed AI agent operations in production Rails. Built with Claude Code.

Chapter 3. Deep Dive Part 2: CLAUDE.md — The Missing Manual for Human–AI Software Collaboration

When an engineer works with a language-model coding assistant across multiple sessions, something breaks down around the third task. The assistant remembers the current file but forgets why a particular library was ruled out. It recalls the mission statement but reinvents the directory structure. It knows what was built in session one but has no model of what was explicitly decided *not* to build.

This is sometimes diagnosed as a context-window problem. It is closer to a *contract* problem. Liu and colleagues' work on long-context language models showed that information placed in the middle of a large prompt is retrieved much less reliably than information placed at the start or end[57]; subsequent work on sycophancy[58] documents a second failure mode in which the assistant aligns with whatever the operator says most recently rather than with what was established earlier in the session. Both effects compound. The assistant is not lying; it is *attending differently* over the session length.

A README tells a reader what a codebase does. It does not tell a language-model implementer what it must *not* do, which assumptions are non-negotiable, which safety rules are load-bearing, or how the work sequence must unfold. A README is written for humans reading once. An AI assistant working across multiple sessions needs a different artefact: a binding contract that governs every decision it makes, placed in a position where the attention budget will reliably reach it.

That artefact is `CLAUDE.md`.

This is Part 2 of the Wild Ecosystem Deep Dive series.

The CLAUDE.md Pattern A `CLAUDE.md` file is a contract between operator and AI system. It specifies:

- **Identity** — what this repo is, where it lives in the ecosystem, what it does
- **Mission** — the problem it solves, the constraints that govern it
- **What It Does NOT Do** — explicit non-goals that prevent scope creep
- **Safety Rules** — non-negotiable constraints, especially for mutation-heavy code
- **Directory Layout** — the canonical structure the AI must respect
- **Build Commands** — how to test, lint, install dependencies
- **Canonical Docs** — where to find the decisions that constrain this work
- **Task Tracking** — how work is sequenced and closed

`CLAUDE.md` comes in two flavours: ecosystem-level and repo-level.

An **ecosystem-level** `CLAUDE.md` coordinates across multiple repositories. The wild ecosystem has one at `wild/CLAUDE.md`. It establishes that the ecosystem is not a

[57] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

[58] Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., et al. (2024). Towards Understanding Sycophancy in Language Models. *ICLR*. arXiv:2310.13548.

monorepo. It defines build philosophy (safety-first, auditable, privacy-aware). It specifies that Beads is the required task tracker across all repos. It enforces the rule: *do not start coding before planning docs exist*.

A **repo-level** `CLAUDE.md` lives inside each repository. The introspection repo has one. The admin-tools repo has one. Each is specific to that repo's constraints.

Here is what a repo-level `CLAUDE.md` looks like, from `wild-rails-safe-introspection-mcp`:

```
#### Identity

- **Repo:** `wild-rails-safe-introspection-mcp`
- **Ecosystem:** wild (see `../CLAUDE.md` for ecosystem-level rules)
- **Mission:** Safe, governed, read-only Rails production introspection
              for AI agents via MCP
- **Language:** Ruby
- **Status:** v1 complete -- all 10 epics finished

#### What This Repo Does

Provides a curated set of MCP tools that let AI agents inspect live Rails
application state -- models, schema, records -- without granting raw console
access or permitting mutation.

#### What This Repo Does NOT Do

- No write operations. Read-only by design.
- No arbitrary Ruby/Rails console execution.
- No admin actions (that's `wild-admin-tools-mcp`).
- No analytics queries or reporting pipelines.
- No multi-framework support in v1 (Rails/ActiveRecord only).
```

The file does not just describe what the repo does. It explicitly lists what it does *not* do. This is not ceremony. It is architecture. It tells the assistant: *if you are tempted to add a write operation, stop. That belongs elsewhere.* Parnas argued in 1971 that the boundaries between modules ought to encode the design decisions most likely to change, and that information about non-membership in a module is as load-bearing as information about membership[59]. The negative-space section of a `CLAUDE.md` is the modern instance of that argument.

Safety Rules That Actually Work The introspection repo is read-only. The admin-tools repo executes mutations. Both have safety rules. The introspection repo has six:

1. **Never introduce write paths.** No `save`, `create`, `update`, `destroy`, or write SQL.

[59] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the IFIP Congress.* https://doi.org/10.1184/R1/6606470.V1

2. **Never bypass the query guard.** All data access goes through the guard. No direct adapter calls.
3. **Never skip audit logging.** Every invocation produces an audit record.
4. **Never expose blocked resources.** Denylist columns must be stripped before data leaves.
5. **Never accept arbitrary code as input.** Tool parameters are data, not code. No `eval`.
6. **Prefer restrictive defaults.** When uncertain, deny access — the *fail-safe defaults* principle Saltzer and Schroeder identified in 1975[60].

These are not aspirational principles. They are enforced constraints. The test suite includes adversarial tests that explicitly try to break each rule — an audit-logging test verifies that denials produce records, a denylist test confirms that blocked columns never appear in responses, a mutation test tries to call `update` and verifies that the code rejects it. The construction mirrors the *constitutional* training regime Bai and colleagues described for harmlessness in language-model assistants[61]: the rule is declared, the system is then evaluated against deliberate violations of it, and the evaluation result is part of the artefact's standing.

The admin-tools repo has seven safety rules, because it executes mutations:

1. **Never bypass the capability gate.** Operations fail closed if the gate is unavailable.
2. **Never skip dry-run support.** Every action has preview and execute paths. Preview has no side effects.
3. **Never skip confirmation for destructive operations.** Two-phase confirmation with server-generated nonce.
4. **Never skip audit logging.** Every invocation produces before/after snapshots.
5. **Never accept arbitrary code as input.** Parameters are data.
6. **Never exceed blast-radius caps.** Hard ceilings enforced in code.
7. **Prefer restrictive defaults.** Deny by default.

Notice the pattern. Rules 3–7 are nearly identical to the introspection repo (audit, parameters, defaults). Rules 1–2 are mutation-specific (capability gate, dry-run). The pattern scales. As repos with different safety profiles are added, the rules adapt; the discipline is consistent.

The structure also operates as a counter-force to the sycophancy failure mode Sharma and colleagues documented[62]. When a written rule says "never bypass the capability gate," and an operator mid-session asks the assistant to "just skip the gate this once for debugging," the contract is the anchor that lets the assistant decline without negotiating away the safety property.

---

[60] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[61] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., et al. (2022). *Constitutional AI: Harmlessness from AI Feedback.* arXiv:2212.08073.

[62] Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., et al. (2024). Towards Understanding Sycophancy in Language Models. *ICLR.* arXiv:2310.13548.

The Ecosystem-Level CLAUDE.md `CLAUDE.md` is not only a per-repo file. The ecosystem-level file coordinates across all ten repositories.

The wild ecosystem's `CLAUDE.md` establishes five core principles:

- **Safety first, always** — every tool defaults to safe, non-destructive behaviour
- **Auditability by default** — actions are logged, decisions are traceable
- **Privacy-aware telemetry** — collect what is needed, nothing more
- **Modular repos, not one giant codebase** — each repo has clear boundaries
- **Documentation-led execution** — planning artefacts exist before code

It then specifies work-sequence rules:

- Do not start coding before planning docs exist
- Do not create implementation tasks before the repo mission and boundaries are clear
- Work one repo at a time unless a cross-repo dependency explicitly requires coordination
- Prefer small, reviewable phases over large batch changes
- Avoid speculative infrastructure with no near-term use
- Avoid copy-paste divergence across repos

These are not guidelines. They are enforced through Beads (the task tracker). If a repo has not filed its mission-and-boundaries document, the AI cannot create Beads. If Beads have not been created, implementation cannot begin. The sequence is locked in.

Negative Space as Architecture One of the most powerful aspects of `CLAUDE.md` is what it explicitly excludes. The "What This Repo Does NOT Do" section is load-bearing architecture.

The introspection repo says: "No write operations. No admin actions. No analytics pipelines." This is not helpful context. It is a boundary. It tells the assistant: *if you are tempted to add caching, you are in the wrong repo. If you are thinking about administrative tooling, that belongs in `wild-admin-tools-mcp`. Do not merge these repos.*

The ecosystem-level `CLAUDE.md` reinforces this. It specifies that introspection is read-only by design, that admin-tools is for writes, and that they have separate safety models for a reason. They are not features of the same system. They are separate repos with separate concerns, separate safety rules, and separate operational lifecycles.

This prevents a common failure mode in ecosystem-scale projects: feature silo erosion. Without explicit boundaries, the introspection repo gradually accumulates write operations. The admin-tools repo duplicates read logic. The capability-gate repo becomes a catch-all. The erosion is subtle and pervasive — and it is exactly the failure mode Parnas warned against in his 1971 paper on information distribution, where he argued that module boundaries must hide the design decisions most likely to change[63].

[63] Parnas, D. L. (1971). Information Distribution Aspects of Design Methodology. *Proceedings of the*

`CLAUDE.md` prevents this by making the boundaries explicit and front-loaded. The assistant sees "What This Repo Does NOT Do" on the first page of the contract. It becomes a filter for every architectural decision — and, by virtue of being at the *start* of the prompt rather than the middle, it is exactly where the long-context attention curve is most reliable[64].

The 10-Epic Pattern Every repo in the wild ecosystem follows the same structure: ten epics, minimum.

This is not arbitrary. The ten-epic pattern is a predictable framework that helps the assistant understand its location in the work. An epic is not a sprint. It is a major outcome area covering the full scope of the repo.

The master blueprint specifies the pattern for the wild ecosystem:

- **Wave 1** — Foundation (3 repos): capability-gate, introspection, admin-tools
- **Wave 2** — Observability pipeline (3 repos): telemetry, transcript pipeline, gap miner
- **Wave 3** — SDLC companions (3 repos): hook ops, permission analyzer, test flake forensics
- **Wave 4** — Coordination (1 repo): skills registry

Each repo is independently broken into ten epics. The introspection repo's epics cover authorisation, query guards, audit logging, policy, threat modelling, architecture decisions, safety evaluation, deployment, operations, and confirmed out-of-scope items.

Why this structure? Because it is predictable. After an assistant reads the master blueprint and understands Wave 1, it knows: *we are building three foundation repos first. Each one has ten epics. We finish one epic, then move to the next. When all three repos are complete, we move to Wave 2.* The structure creates narrative coherence. The assistant does not wander. It knows where it is and what comes next.

Beads: Task Tracking for AI Agents Beads is a task tracker designed for post-compaction recovery. It is the required task-tracking system for the wild ecosystem.

Traditional ticketing systems are inadequate for AI collaboration. Jira tickets are written for human teams to estimate and track progress. They are not designed for AI sessions that might be interrupted, resumed hours later, and resumed again by a different session. They do not enforce sequence. They do not capture evidence. They do not require annotations.

Beads does.

IFIP Congress. https://doi.org/10.1184/R1/6606470.V1

[64] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

Every repo's work is broken into:

- ten epics covering full scope
- child tasks under each epic with clear acceptance criteria
- explicit dependency blocks between tasks and across repos
- annotations that explain context and blocking assumptions

A task closure requires evidence: *done* means the work exists, it is verifiable, and the close reason states what was produced. A weak annotation says `In progress`. A strong annotation says: `Started schema definition. Blocking question: are capability IDs scoped per-repo or globally? Needs a decision before authorisation logic can be written.`

The Beads workflow is simple:

```
bd ready                # Find unblocked work
bd update <id> --status in_progress  # Claim a task
bd close <id> --reason "evidence"    # Close with evidence
bd sync                 # Persist to disk
```

No task starts without a marked Beads entry. No task closes without evidence. This enforces sequence. An assistant cannot skip planning and jump to code because implementation tasks are blocked until planning tasks are closed. The pattern is, structurally, a *protocol* in the session-typed sense[65]: the legal transitions between states are encoded in the tooling, not in the assistant's discretion.

Adopting This Today If a team is working with Claude Code on a project, `CLAUDE.md` can be adopted today. The practical template is short. Create a `CLAUDE.md` file at the root of the repository:

```
This file provides guidance to Claude Code when working in this repository.

#### Identity

- **Repo:** `your-repo-name`
- **Mission:** One clear sentence. What this repo does and why.
- **Language:** Your primary language
- **Status:** v1 scaffolding / v1 complete / v2 in progress

#### What This Repo Does

One paragraph describing the repo's scope and primary responsibility.

#### What This Repo Does NOT Do
```

[65] Honda, K., Vasconcelos, V. T., & Kubo, M. (1998). Language Primitives and Type Discipline for Structured Communication-Based Programming. *European Symposium on Programming (ESOP)*. https://doi.org/10.1007/BFb0053567

```
- Explicit list of things this repo will not do
- This prevents scope creep and feature confusion
- Makes boundaries clear to both humans and AI

#### Build Commands

bundle install    # Install dependencies
bundle exec rspec # Run tests

#### Key Docs

| Doc | Purpose |
|-----|---------|
| `docs/mission.md` | Mission and boundaries |

#### Before Working Here

1. Read this file completely
2. Check current work state
3. Read the relevant doc for the active task
4. Do not skip ahead to later phases
```

That is enough. A `CLAUDE.md` can be minimal and grow over time. Early versions might not have safety rules or ecosystem-level coordination. That is fine. Add what is needed as the project matures.

The pattern evolves with the project. A v1 `CLAUDE.md` is lean: mission, non-goals, build commands. A v2 adds safety rules, testing strategies, expanded canonical docs. An ecosystem-level `CLAUDE.md` adds wave sequencing, dependency thinking, and cross-repo standards.

The key is to start. Write the identity block. Write the non-goals. Write the build commands. Make it a contract. The assistant will follow it.

What Comes Next

- **Part 1**: The Safety Architecture — defence in depth, adversarial testing, hard safety ceilings.
- **Part 3**: The Observability Loop — how telemetry, transcripts, and gap mining create a self-improving feedback loop.
- **Part 4**: Claude Code as Tech Lead — what happens when the AI makes the architectural decisions.

Build the `CLAUDE.md` first. Before the code. Before the architecture. It focuses the decisions, constrains the AI appropriately, and keeps the project coherent across sessions.

Part of the Wild Ecosystem — 10 Ruby gems for governed AI agent operations in production Rails. Built with Claude Code.

Chapter 4. Deep Dive Part 3: The Observability Loop — Teaching AI Tools to Improve Themselves

Most AI-tool ecosystems follow the same operational pattern: build the tools, ship them, hope they work. Updates happen when someone reports a problem or a human reviewer notices something is broken. For agents that hold tool access to production Rails systems, with real stakes and real blast radius, hoping is not a strategy.

The wild ecosystem was built with a different assumption: **agents operating with access need external feedback about what they struggle with.** Not compliance logging. Not dashboards for human consumption. Structured signals about denial rates, failure patterns, latency outliers, and capability gaps, designed for downstream automated analysis.

Three repositories make this work. They share no code. They have independent test suites, independent release cycles, and clear data contracts. Together, they answer a question most AI-in-production deployments never ask: *what is this system actually struggling with?* The architectural pattern — instrument cheaply, normalise centrally, analyse asynchronously — is the same one Sigelman and colleagues described in Google's Dapper paper[66], adapted for a domain where the *production system* is an agent that calls tools rather than a microservice that serves requests.

This is Part 3 of the Wild Ecosystem Deep Dive series.

The Three-Repo Pipeline The observability loop is implemented as three independent gems. Each has a single responsibility. Each defines clear input/output contracts.

```
wild-session-telemetry     (collection)
    |
    v  structured events
wild-transcript-pipeline   (normalization + redaction)
    |
    +-----> wild-gap-miner (analysis + recommendations)
    |            ^
    +------------+
```

Repository 1: wild-session-telemetry (325 tests) **Mission:** collect and export privacy-aware telemetry from agent sessions.

When agents invoke tools through `wild-admin-tools-mcp` or `wild-rails-safe-introspection-mcp`, events fire to the telemetry layer. Those events describe:

- the tool that was called (action name)
- the outcome (success, denied, error, rate-limited, preview)
- how long it took (latency in milliseconds)

[66] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Google Technical Report dapper-2010-1.

- when it happened (ISO 8601 timestamp)
- who called it (service-account identifier)

The telemetry layer **never** receives:

- raw parameter values (which job ID was retried, which cache key was invalidated)
- before/after state snapshots
- confirmation nonces
- stack traces
- adapter-specific identifiers

These exclusions are hardcoded. Not configurable. The forbidden-field list is enforced at the privacy layer, before any event touches storage — closer in spirit to the *data-minimisation* discipline of Sweeney's k-anonymity work[67] than to the more permissive "log everything, scrub later" pattern common in operational telemetry.

Why fire-and-forget? Telemetry failures must never break the upstream pipeline. If the telemetry collector is slow, out of disk space, or completely down, agents continue operating. A failure is logged to stderr, swallowed, and operations proceed. This is the opposite of how logging usually works, and it is intentional. The pattern is a direct descendant of Armstrong's *let-it-crash* discipline from the Erlang OTP tradition[68]: the unreliable subsystem must be quarantined from the reliable one, and the supervisor — not the worker — owns the decision about how failures propagate.

Configuration is frozen at startup. Storage backends, output paths, privacy rules — all locked. No runtime reconfiguration. This prevents configuration drift from weakening privacy guarantees mid-session.

Pure Ruby with zero runtime dependencies beyond stdlib.

Repository 2: wild-transcript-pipeline (200+ tests) Mission: ingest, normalise, and process conversation transcripts at scale.

This repository handles the raw conversation context — turns, tool invocations, intent signals. It does three things:

1. **Ingestion with multiple adapters.** Three sources: Claude Code session JSONL, MCP protocol logs, and generic conversation JSON. Each adapter converts its format into a common normalised schema.
2. **Privacy redaction at the turn level.** Transcripts contain freeform conversation that may include secrets and personally identifiable information. The pipeline strips:
    - API keys and tokens (AWS, GitHub, bearer tokens)
    - email addresses and IP addresses

[67] Sweeney, L. (2002). k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5), 557–570. https://doi.org/10.1142/S0218488502001648

[68] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm.

- absolute filesystem paths
- embedded file contents
- custom patterns (via configuration)

3. **Export in multiple formats.** Normalised and redacted transcripts export as clean JSON Lines or Markdown.

This gem does not communicate with a network. It does not execute code. It does not persist state. It is a pure data-transformation layer: ingest, normalise, redact, export.

Zero runtime dependencies.

Repository 3: wild-gap-miner (276 tests) **Mission:** analyse telemetry and transcript data to surface capability gaps.

This is where the pipeline becomes intelligence. The gap-miner consumes structured data from both telemetry and transcripts and runs six analysers:

1. **Denial analyser** — high denial rates signal policy mismatches or gaps in the capability gate's allowlist.
2. **Failure analyser** — high tool-failure rates suggest fragile implementations or unhandled edge cases.
3. **Latency analyser** — high p95 latency suggests resource contention, blocking I/O, or algorithmic inefficiency.
4. **Utilisation analyser** — low capability coverage means tools are undiscoverable, poorly designed, or genuinely unneeded.
5. **Coverage analyser** — gaps between what agents attempt and what tools provide are feature-request signals.
6. **Pattern analyser** — unusual sequences of tool calls, unexpected outcome distributions, or behavioural anomalies.

Each analyser scores findings on a 0.0–1.0 scale. Gaps are ranked by severity. Recommendations are auto-generated. The architectural pattern — *cheap collection plus deferred deep analysis* — is the design Sigelman and colleagues recommended in the Dapper paper based on operational experience at Google scale[69]; the move Beyer and colleagues codified for SRE practice generally is the same one[70].

Pure Ruby stdlib. Zero runtime dependencies.

Privacy-First Telemetry: How Exclusions Are Enforced The session-telemetry library enforces privacy through three mechanisms.

[69] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Google Technical Report dapper-2010-1.

[70] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media.

1. Hardcoded Forbidden-Field List Twenty-two-plus field patterns that must never be stored. This list is code, not configuration. It is not optional.

Examples: `params`, `parameters`, `before_state`, `after_state`, `nonce`, `confirmation_nonce`, `stack_trace`, `backtrace`, `jid`, `execution_id`.

If an incoming event contains any of these fields, they are stripped before storage. The design choice to keep the list in code rather than in configuration is deliberate: configuration is mutable at deploy time and prone to silent erosion; code is part of the artefact under review. The audit-log-integrity literature, going back to Schneier and Kelsey's 1999 work[71] and Haber and Stornetta's earlier digital-time-stamp scheme[72], converges on the same posture — the rule that protects sensitive data must itself be tamper-evident.

2. Per-Event-Type Metadata Allowlists Each event type has an explicit allowlist of metadata fields. Only those pass through. Everything else is dropped.

A job-retry event might allow metadata fields like `category` and `job_type`. It does not allow `job_id`, `parameters`, `retry_count`, or anything that could leak operational specifics.

3. Fire-and-Forget with Zero Propagation Telemetry errors — ingestion failures, storage failures, export failures — are logged to stderr and swallowed. They never propagate to the upstream caller.

This is a safety feature disguised as a failure mode. If the telemetry layer is broken, agents continue operating. The downstream gap-miner may see no data, but the operational pipeline is unaffected. The discipline is the Armstrong *isolate-failures-at-the-supervisor* pattern from the Erlang OTP tradition[73]: the worker does not get to decide whether its own failure should hurt the system. Telemetry gaps will not page anyone at 03:00; they will be discovered during analysis, when the gap-miner reports no recent data. That is a reasonable trade-off.

What the Gap Miner Finds Here is a concrete example of what gap analysis produces.

Input: thirty days of telemetry plus transcripts from a Rails development workflow.

Gap report output:

```
Gaps Discovered (ranked by severity):

1. [DENIAL RATE]
   Severity: 0.92
```

[71] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

[72] Haber, S., & Stornetta, W. S. (1991). How to Time-Stamp a Digital Document. *Journal of Cryptology*, 3(2), 99–111. https://doi.org/10.1007/BF00196791

[73] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm.

```
   Tool: introspection.list_models
   Evidence: Hit 47% denial rate; policy mismatch or under-privileged caller
   Recommendation: Review capability gate policy for service-account-ci

2. [COVERAGE GAP]
   Severity: 0.78
   Signal: Agents asked for job retry API 18 times; current tool not exposed
   Recommendation: Add job_retry action to admin-tools-mcp

3. [LATENCY SPIKE]
   Severity: 0.65
   Tool: introspection.schema_query
   Evidence: p95 latency 2400ms; p50 is 140ms
   Recommendation: Check N+1 queries in schema introspection; consider caching

4. [PATTERN ANOMALY]
   Severity: 0.42
   Signal: Agents are calling introspection.list_models in tight loops
   Recommendation: Add discovery tool that lists available models once
```

Each gap is scored, ranked, and actionable. Operators prioritise work. Product teams make feature decisions. Infrastructure teams investigate performance regressions. The format follows the *p99 latency plus categorical outcome* idiom Beyer and colleagues established as the operational standard for SRE-grade observability[74].

Zero Runtime Dependencies Both the transcript-pipeline and gap-miner use pure Ruby stdlib. No ActiveRecord, no HTTP client, no JSON parser beyond stdlib.

The reasons are operational. First, *production safety*: no dependency tree to audit, no transitive security issues, no version conflicts with the host application. Second, *deployment simplicity*: these gems embed in any Ruby environment without adding operational overhead. Third, *composability*: the pipeline can run offline, in batch, or integrated into diverse infrastructure without coordinating dependency versions. Fourth, *auditability*: the source code is the complete system. There are no hidden behaviours in third-party libraries — a property Schneier and Kelsey identified as essential for any audit subsystem[75].

Claude Code's Role in Building the Loop Several decisions Claude Code made during implementation are worth naming explicitly.

1. **Fire-and-forget semantics for telemetry failures.** Not *fail fast*, but *never*

[74] Beyer, B., Jones, C., Petoff, J., & Murphy, N. R. (Eds.). (2016). *Site Reliability Engineering: How Google Runs Production Systems.* O'Reilly Media.

[75] Schneier, B., & Kelsey, J. (1999). Secure Audit Logs to Support Computer Forensics. *ACM Transactions on Information and System Security*, 2(2), 159–176. https://doi.org/10.1145/317087.317089

fail the upstream caller. The error-containment design follows Armstrong's supervision-tree discipline directly[76].

2. **Forbidden-field enforcement as hardcoded strings, not regexes.** Prevents subtle bugs where matching logic evolves over time and becomes less strict.
3. **Per-event-type metadata allowlists.** Not a global allowlist, but per-event-type. Fine-grained control while preventing operator mistakes — the *least common mechanism* posture Saltzer and Schroeder argued for[77] adapted to telemetry instead of access control.
4. **Three independent transcript adapters.** Not a monolithic parser, but separate adapters with a common output schema. Adding new sources is straightforward.
5. **Gap scoring on a normalised 0.0–1.0 scale.** Allows gaps to be ranked across different analysis types (denial, latency, coverage) on a common severity axis.
6. **Zero runtime dependencies as a hard constraint.** Not a goal — a requirement. The implementation had to use only stdlib.

Building Your Own Loop The pattern generalises to any AI-tool ecosystem.

Layer 1: collection (fire-and-forget). Build a subscriber that receives events from operational tools. Apply strict privacy filtering. Make failures invisible to upstream callers. Freeze configuration after startup.

Layer 2: normalisation (multiple sources, one schema). Accept raw transcripts from multiple sources. Normalise via adapters. Apply privacy redaction at the content level. Export in multiple formats.

Layer 3: analysis (scoring and ranking). Build analysers that consume normalised data and surface gaps. Score on a normalised scale. Rank by severity. Generate actionable recommendations.

Cross-cutting concerns:

- Document input/output schemas for each layer. Make them immutable contracts.
- Make privacy defaults restrictive. Force operators to expand, not shrink.
- Every privacy claim should have a corresponding adversarial test.
- File technical decisions in durable documents for future implementers.

The Feedback Loop in Action

1. Agents interact with tools through `wild-rails-safe-introspection-mcp` and `wild-admin-tools-mcp`.
2. Those interactions emit telemetry events to `wild-session-telemetry`.

[76] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm.

[77] Saltzer, J. H., & Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

3. Sessions also produce transcript logs.
4. Both streams feed into `wild-transcript-pipeline` for normalisation and redaction.
5. The gap-miner consumes both normalised telemetry and transcripts.
6. Gap reports surface actionable feedback: missing tools, policy mismatches, performance regressions, usage patterns.
7. That feedback drives tool development, policy tuning, and infrastructure improvements.

Without this loop, tools stagnate. With it, they improve iteratively based on actual usage patterns and observed agent struggles.

The wild ecosystem is not a static set of tools. It is a learning system that gets better because it has feedback — instrumented along the architectural lines that Dapper established for distributed tracing[78], with the privacy posture k-anonymity-style data minimisation prescribes[79], and the supervision discipline let-it-crash made operational for fault containment[80].

What Comes Next

- **Part 1**: The Safety Architecture — defence in depth, adversarial testing, hard safety ceilings.
- **Part 2**: CLAUDE.md — Human–AI Collaboration Pattern — per-repo contracts that prevent scope creep and enforce safety.
- **Part 4**: Claude Code as Tech Lead — what happens when the AI makes the architectural decisions.

This is the opposite of "build and hope." This is "build, observe, improve."

Part of the Wild Ecosystem — 10 Ruby gems for governed AI agent operations in production Rails. Built with Claude Code.

---

[78] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., & Shanbhag, C. (2010). *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Google Technical Report dapper-2010-1.

[79] Sweeney, L. (2002). k-Anonymity: A Model for Protecting Privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5), 557–570. https://doi.org/10.1142/S0218488502001648

[80] Armstrong, J. (2003). *Making Reliable Distributed Systems in the Presence of Software Errors.* PhD thesis, Royal Institute of Technology (KTH), Stockholm.

Chapter 5. Deep Dive Part 4: Building 10 Production Gems with Claude Code as Tech Lead

The wild ecosystem set out to build a family of production-grade Ruby gems forming a governed operational-intelligence layer for AI agents. That ecosystem is now complete: ten repositories, approximately 2,924 tests, zero RuboCop offenses, sixty-plus canonical documents, all v1 implementations shipped.

This article is about how that was built, and what a collaboration between a human architect and an AI in the tech-lead seat actually looks like in practice. The relevant literature is moving quickly. Yao and colleagues' ReAct paper[81] established the reason-then-act loop as the dominant abstraction for tool-using agents; Schick and colleagues' Toolformer work[82] and Wei and colleagues' chain-of-thought work[83] are the immediate predecessors; Park and colleagues' generative-agents study[84] demonstrated agent populations sustaining coherent multi-day behaviour; Li's recent survey of LLM-based agent paradigms[85] catalogues the design space; and the SWE-smith line of work on agent-driven software engineering[86] documents how rapidly the benchmarks are moving. The wild ecosystem sits inside that trend, not outside it — but it makes deliberate choices about *which* parts of the agent are load-bearing and which parts the infrastructure absorbs.

This is Part 4 of the Wild Ecosystem Deep Dive series.

The Collaboration Model The wild ecosystem was not built with Claude Code as a copilot — autocompleting code fragments, implementing features described in natural language, staying within preset guardrails.

Instead, Claude Code served as tech lead: making architecture decisions, choosing implementation patterns, catching its own bugs, testing its own code adversarially, and maintaining cross-repository consistency.

Jeremy Longshore operated as product owner and architect. He defined:

- the mission of each repository
- the boundaries (what each repo does and does not do)
- the ten-epic structure for each repo
- the key architectural decisions that needed to be made

[81] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[82] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., & Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. *NeurIPS*. arXiv:2302.04761.

[83] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Ichter, B., Xia, F., Chi, E., Le, Q., & Zhou, D. (2022). Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *NeurIPS*. arXiv:2201.11903.

[84] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST*. https://doi.org/10.1145/3586183.3606763

[85] Li, X. (2024). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. *COLING*. arXiv:2406.05804.

[86] Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., Hui, B., Press, O., Schmidt, L., & Yang, D. (2025). *SWE-smith: Scaling Data for Software Engineering Agents*. arXiv:2504.21798.

- the non-negotiable safety rules for each context

Claude Code operated as implementer and technical decision-maker within those constraints. It:

- made sound implementation choices (e.g., should this authorisation check be a constant or a hash?)
- wrote adversarial tests against its own code
- identified and fixed bugs
- maintained patterns across repositories to prevent divergence
- surfaced architectural questions when the constraints left room for multiple valid approaches

This model flips the typical AI-in-development relationship. Jeremy was not directing Claude Code line by line. Claude Code was not waiting for approval on every design choice. They were collaborating as peers with clear ownership. The pattern is consistent with the *agent-as-tool-using-reasoner* abstraction Yao and colleagues introduced[87] but extended to a higher unit of work: the agent reasons about *architecture* rather than only about the next tool call. Li's survey notes that this kind of long-horizon decision-making is where current agent stacks struggle most[88]; the wild approach addresses it by constraining the architectural surface to the bounded design space the contract (the `CLAUDE.md` file) declares.

The Numbers: What Actually Got Built

Gem	Tests	Purpose
wild-capability-gate	224	Cross-cutting access control
wild-rails-safe-introspection-mcp	468	Safe, read-only Rails introspection via MCP
wild-admin-tools-mcp	439	Governed admin operations via MCP
wild-session-telemetry	325	Privacy-first telemetry collection
wild-transcript-pipeline	200+	Transcript normalisation with PII redaction
wild-gap-miner	276	Gap analysis from telemetry data
wild-hook-ops	247	Hook lifecycle management
wild-permission-analyzer	217	Static permission auditing
wild-test-flake-forensics	277	Flake detection + root-cause analysis

[87] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., & Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. *ICLR*. arXiv:2210.03629.

[88] Li, X. (2024). A Review of Prominent Paradigms for LLM-Based Agents: Tool Use, Planning (Including RAG), and Feedback Learning. *COLING*. arXiv:2406.05804.

Gem	Tests	Purpose
wild-skillops-registry	251	Skills registry + coordination
Total	**~2,924**	**10 repos, 10 epics each, 60+ canonical docs**

All written to pass zero RuboCop offenses. All with explicit safety rules documented and tested. All with clear data contracts and cross-repo dependencies mapped.

What "AI as Tech Lead" Actually Means This model is not about code generation. It is about technical decision-making under constraints.

Consider the capability gate — the access-control layer that guards privileged tool access across the ecosystem. When Jeremy defined the mission ("answer whether this agent can call this tool in this context"), Claude Code had to decide:

- what is the shape of a capability ID? Is it a string like `"admin.job.retry"`, or is there a structured format?
- how are policies stored? In memory, in YAML, in a DSL?
- what is the runtime interface? Do calls check a constant hash, call a method on a gate object, or something else?

Jeremy provided the safety rules: authorisation must be logged, denials must never propagate as exceptions, configuration must freeze after startup. Within those constraints, Claude Code made implementation choices.

These choices had downstream effects. When `wild-admin-tools-mcp` needed to integrate the capability gate, the decisions Claude Code made in the gate repo directly shaped the integration pattern. When `wild-rails-safe-introspection-mcp` followed, it inherited those patterns. The cross-repo consistency property is exactly what Bernstein and colleagues' Orleans virtual-actor work argues for in a distributed-systems setting[89]: when the *state machine* is consistent across instances, downstream complexity drops sharply.

That consistency across ten independent repositories did not happen by accident. It happened because Claude Code maintained internal coherence — asking itself *am I following the pattern established in the capability gate?* and *will future implementers understand why I chose this way?*

The CLAUDE.md Coordination Mechanism Each repository has its own `CLAUDE.md` file. These files are not documentation — they are contracts.

Example from `wild-admin-tools-mcp`:

[89] Bernstein, P. A., Bykov, S., Geller, A., Kliot, G., & Thelin, J. (2014). *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Microsoft Research Technical Report MSR-TR-2014-41.

```
#### Safety Rules for Claude Code

These are non-negotiable when working in this repo:

1. Never bypass the capability gate. All operations require gate
   authorization. If the gate is unavailable, operations fail closed.
2. Never skip dry-run support. Every action handler must implement
   both preview and execute paths. Dry-run must never trigger side effects.
3. Never skip confirmation for destructive operations. Two-phase
   confirmation with server-generated nonce is mandatory.
```

These are not suggestions. They are rules that Claude Code follows and tests against. A change that violates these rules is rejected, even if it would otherwise be a clean implementation.

The `CLAUDE.md` file is the binding contract between Jeremy (as architect) and Claude Code (as implementer). It prevents scope creep. It enforces consistency. It makes the constraints explicit so they can be tested. Part 2 of this series treats the contract pattern in detail; in short, the contract is positioned at the start of the prompt so that the long-context attention curve Liu and colleagues characterised[90] does not degrade it over a long session.

Cross-Repo Dependency Management Ten independent repositories need consistent patterns. The wild ecosystem used three mechanisms.

1. Beads (task tracking) Each repository uses Beads for task tracking. All ten repositories follow the same structure:

- ten epics per repo (named for outcomes, not technical nouns)
- child tasks under each epic with clear acceptance criteria
- explicit dependency blocks between tasks and across repositories
- annotations that read like operator progress notes, not machine scraps

Moving from one repo to another felt natural. The task structure was familiar. The acceptance-criteria patterns were consistent.

2. Canonical Documentation in `000-docs/` Every repo has a `000-docs/` directory with filed documents following `/doc-filing` conventions:

- `001-PP-PLAN-repo-blueprint.md` (mission, boundaries, architecture)
- `002-PP-PLAN-epic-build-plan.md` (ten-epic breakdown)
- `003-TQ-STND-privacy-model.md` (privacy guarantees)
- `004-AT-ADEC-architecture-decisions.md` (why things are shaped this way)

[90] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

Understanding a new repo required reading predictable documents in a predictable order. The cognitive load for moving between repositories was minimal.

3. Shared Notes When a pattern emerged that applied across multiple repos, it was documented once and linked from affected repos. When multiple repos needed to validate configuration, a shared note on *Configuration Freezing Patterns* was filed. Each repo referenced it, preventing copy-paste divergence. The pattern parallels the Orleans *grain* abstraction Bernstein and colleagues described for distributed systems[91]: the canonical definition lives in one place, and every consumer holds a reference, not a copy.

Quality Metrics: What a Human Reviewer Would Find Zero RuboCop offenses across all ten repositories is notable but expected. More interesting are the quality choices.

Adversarial Testing Every safety rule gets tested by code that tries to break it. The methodology follows the *language-model red-teaming* pattern Perez and colleagues introduced at EMNLP 2022[92]: the system under test is also a participant in the test design, and the tests deliberately target the rules the system is supposed to enforce.

Example from `wild-admin-tools-mcp`: the safety rule is *never bypass the capability gate*. The test creates a scenario where an action is configured as allowed, but the capability gate denies it for a specific caller. The test verifies that even though the action is configured, the gate denial is respected.

Privacy Invariant Tests The `wild-session-telemetry` library claims "twenty-two field patterns must never be stored." The test creates an event with all twenty-two forbidden fields, feeds it to the ingestion layer, and verifies that after storage, none of those fields exist in the stored record.

Data Contract Tests Each repo that exports data has tests verifying the output contract. Example from `wild-gap-miner`: the export schema specifies that gap severity is always in `[0.0, 1.0]`. The test creates gaps with invalid severity (e.g., 1.5, -0.1) and verifies that the export layer rejects them.

Cross-Repo Integration Tests When `wild-gap-miner` depends on output from `wild-session-telemetry` and `wild-transcript-pipeline`, integration tests create realistic exports from both upstream repos, feed them to the gap-miner, and verify that analysis runs correctly. These tests prevent contract drift — the same failure mode

[91] Bernstein, P. A., Bykov, S., Geller, A., Kliot, G., & Thelin, J. (2014). *Orleans: Distributed Virtual Actors for Programmability and Scalability.* Microsoft Research Technical Report MSR-TR-2014-41.

[92] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

the SWE-smith authors observed in agent-driven software-engineering benchmarks, where contract changes upstream silently invalidate downstream agent behaviour[93].

What This Changes The developer becomes architect. The AI handles implementation depth.

In a traditional model, the architect specifies features, and the developer implements them. In this model, Jeremy specified constraints and outcomes, and Claude Code made the technical decisions that transformed those constraints into working, tested, documented code.

The implications:

1. **Architectural leverage increases.** One architect can oversee ten repositories with thousands of tests because the AI is making sound decisions within specified constraints — not asking for permission on every detail. Brooks' 1987 *No Silver Bullet* argument cautioned against expecting any single tool to deliver an order-of-magnitude productivity improvement[94]; the more interesting question is whether a *collaboration model* can. The evidence here suggests yes, but the gain is in the architect's leverage rather than in raw code-writing speed.
2. **Code review focus shifts.** Instead of reviewing line-by-line implementations, review focuses on whether constraints were honoured, whether safety rules were tested, whether documentation is clear.
3. **Consistency improves.** Because the AI makes decisions within constraints, patterns emerge naturally. Divergence has to be intentional, not accidental.
4. **Knowledge transfer becomes feasible.** Documentation-first execution means future implementers (human or AI) can read the decisions and understand the reasoning.

Practical Takeaways: How to Replicate This

1. Define the CLAUDE.md template Create a standard `CLAUDE.md` structure that every repo follows. Include mission, scope, non-goals, directory layout, build commands, safety rules (numbered, binding, tested), canonical-doc index, and task-tracking rules. Make it a contract, not guidance.

2. Use the 10-epic structure Every repository starts with the same epic breakdown:

- Epic 1: Foundations (schema, configuration, core interfaces)

[93] Yang, J., Lieret, K., Jimenez, C. E., Wettig, A., Khandpur, K., Zhang, Y., Hui, B., Press, O., Schmidt, L., & Yang, D. (2025). *SWE-smith: Scaling Data for Software Engineering Agents.* arXiv:2504.21798.

[94] Brooks, F. P. (1987). No Silver Bullet — Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

- Epics 2–9: Feature areas (each focused and complete)
- Epic 10: Quality (adversarial testing, documentation, release)

This forces completeness. A repo that skips Epic 10 cannot ship.

3. Track tasks with explicit dependencies Use Beads or an equivalent that enforces dependency blocks between tasks, cross-repo dependency visibility, narrative annotations, and no task closure without evidence.

4. Document decisions as you build When a question arises ("should we store denied capabilities in the gate log?"), answer it in a filed document, not just in code comments. Use numbered documents with category and type codes. Maintain an INDEX.

5. Establish clear data contracts When a repository exports data, the output schema is a living document: filed, typed, versioned, and validated by integration tests.

6. Test safety rules adversarially For every safety rule, write a test that tries to violate it:

```
describe "Safety: never bypass the gate" do
  it "fails closed when gate denies, even if action is in allowlist" do
    gate.deny_action(:retry_job, for: specific_caller)
    result = executor.execute(Action.new(:retry_job, caller: specific_caller))

    expect(result).to be_denied
    expect(result.reason).to eq "gate_denied"
  end
end
```

These tests are the safety ratchet. They prevent regressions. The methodology — *adversarial tests as a first-class verification artefact* — follows directly from Perez and colleagues' language-model red-teaming work[95] and from Park and colleagues' demonstration that agent populations require explicit behavioural constraints to remain coherent over long horizons[96].

The Reality Check Building ten production repositories with this model required:

- clear architectural vision upfront (the master blueprint)
- detailed per-repo planning before any code (the `CLAUDE.md` files)
- disciplined task tracking (Beads, with explicit dependencies)

[95] Perez, E., Huang, S., Song, F., Cai, T., Ring, R., Aslanides, J., Glaese, A., McAleese, N., & Irving, G. (2022). Red Teaming Language Models with Language Models. *EMNLP*. https://doi.org/10.18653/v1/2022.emnlp-main.225

[96] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST*. https://doi.org/10.1145/3586183.3606763

- aggressive testing (~300 tests per repo on average)
- extensive documentation (sixty-plus filed documents)

This is not a lightweight approach. It is intentional, documented, and exhaustive. The literature on long-horizon agent behaviour is consistent on this point: without explicit structural anchors, agent behaviour drifts[97], [98]; with them, it composes[99].

The result is what most teams do not achieve: ten independent repositories that work together cohesively, with zero accidental divergence, clear contracts, testable safety rules, and documentation that future implementers can actually read and understand.

Summary Claude Code as tech lead works when:

1. the architect defines missions, boundaries, and non-negotiable rules
2. the implementer makes sound technical decisions within those constraints
3. the coordination layer (`CLAUDE.md`, Beads, canonical docs) makes decisions explicit and testable
4. safety is treated as a first-class concern, not an afterthought

The result is not faster development. It is *better* development: less divergence, more consistency, more testable, more documentable.

The wild ecosystem is an existence proof that this model scales to ten independent repositories, multiple layers of abstraction, cross-repo dependencies, and production-grade quality standards.

What changed is not raw AI capability. It is the collaboration model. Human architect, AI tech lead, structured constraints, explicit contracts, rigorous testing — together they produce something neither could build alone. Brooks' framing of the *essence* of software engineering as the irreducible difficulty of getting the design right[100] still applies; what has shifted is *who can hold which part of the essence*.

The Full Series

- **Part 1**: The Safety Architecture — defence in depth, adversarial testing, hard safety ceilings.
- **Part 2**: CLAUDE.md — Human–AI Collaboration Pattern — per-repo contracts that prevent scope creep and enforce safety.

[97] Liu, N. F., Lin, K., Hewitt, J., Paranjape, A., Bevilacqua, M., Petroni, F., & Liang, P. (2023). Lost in the Middle: How Language Models Use Long Contexts. *Transactions of the Association for Computational Linguistics*, 12, 157–173. https://doi.org/10.1162/tacl_a_00638

[98] Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., et al. (2024). Towards Understanding Sycophancy in Language Models. *ICLR*. arXiv:2310.13548.

[99] Park, J. S., O'Brien, J. C., Cai, C. J., Morris, M. R., Liang, P., & Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. *UIST*. https://doi.org/10.1145/3586183.3606763

[100] Brooks, F. P. (1987). No Silver Bullet — Essence and Accidents of Software Engineering. *IEEE Computer*, 20(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

- **Part 3**: The Observability Loop — telemetry, transcripts, and gap mining as a self-improving feedback loop.
- **Part 4**: This article — Claude Code as tech lead across ten production gems.

Part of the Wild Ecosystem — 10 Ruby gems for governed AI agent operations in production Rails. Built with Claude Code.

References

Every source in this paperback is verified against Semantic Scholar (when indexed) or anchored to a stable DOI / URL. See `content/citations/` in the source repository for the BibTeX corpus and per-source verification metadata.