Beyond Function Equivalence: Method-Equivalence Verification for AI Coding Agents and the Explosion of Enterprise Defect Surface

Rajesh Iyer
iyer70@gmail.com

Abstract. AI coding agents deliver on two of the three enterprise KPIs: velocity and cost. GPT-5.3-Codex reaches 56.8% on SWE-Bench Pro and 77.3% on Terminal-Bench 2.0 [3]; Claude Code crossed $1B in annualized revenue within six months [13]. Yet even the velocity claim is fragile: METR's randomized controlled trial found that experienced developers using AI tools were 19% *slower* while believing they were 20% faster [15]. The third KPI—quality, measured as defects—is moving in the wrong direction, invisibly. These agents are evaluated on *function equivalence*: whether a patch makes tests pass. They are not evaluated on *method equivalence*: whether the computational approach, algorithmic contract, and structural properties of the code are preserved. SlopCodeBench reveals that GPT-5.3-Codex achieves 51.6% on core tests but only 23.7% in isolation; Claude Opus 4.6 scores 53.8% core but 21.5% in isolation [1]. The gap—27.9 and 32.3 percentage points—quantifies code that passes every quality gate the enterprise operates while manufacturing latent defects: god functions exceeding 954 lines, clone densities of 116–174 duplicated lines per thousand LOC, and abstractions built then broken across successive edits. At enterprise scale, these invisible defects compound across interconnected software, data, and business estates into an explosion of defect surface that no dashboard currently detects. If the velocity gains are illusory for experienced maintainers and the defect signal is hidden, the risk is that we have merely accelerated defect generation. We formalize the method-equivalence gap, present an empirically-grounded four-type defect taxonomy, propose **MEV** (Method-Equivalence Verification)—a three-layer CI/CD gate—and argue, drawing on Heckman's selection-bias framework [14], that enterprises must maintain airgapped human-authored baselines to prevent MEV's own detection thresholds from silently recalibrating to the degraded normal.

1. Introduction: Velocity, Cost, and the Missing Defect Signal

Enterprise software organizations measure three things: velocity (how fast), cost (how cheap), and quality (how many defects). The premise of AI coding agents is that they deliver on all three. The evidence says otherwise.

In July 2025, METR published the first rigorous randomized controlled trial of AI coding tools with experienced developers [15]. Sixteen engineers—each averaging five years and 1,500 commits on their respective repositories, working in mature open-source codebases exceeding one million lines of code—completed 246 real tasks randomly assigned to AI-allowed or AI-disallowed conditions. Before starting, developers forecasted that AI would reduce their completion time by 24%. After the study, they believed it had reduced it by 20%. The measured result: AI increased completion time by 19%. Developers *felt* faster while *being* slower—a 39-percentage-point perception gap. METR noted that developers "accepted less than 44% of the generations" and spent "considerable time reviewing and editing the code" [15].

Yet adoption is accelerating regardless. Claude Code crossed $1 billion in annualized revenue within six months [13]. GPT-5.3-Codex reaches 56.8% on SWE-Bench Pro, 77.3% on Terminal-Bench 2.0, and 64.7% on OSWorld-Verified [3]. These benchmark numbers are real, and the tools are genuinely useful for many tasks.

But the benchmarks measure *function equivalence*—whether a generated patch makes previously-failing tests pass. Enterprise quality is not measured this way. Enterprise quality is measured in *defects*: defects per KLOC, defect escape rate to production, mean time to detect. And on this metric—the one that boards see, that auditors check, that operational risk is priced against—the evidence suggests that AI coding agents are manufacturing a category of defect that is invisible to every detection system the enterprise currently operates. If the velocity gains are illusory for experienced maintainers, and the defect signal is hidden, then all we may have done is speed up defect generation.

1.1 The Invisible Defect Problem

When a coding agent produces structurally degraded code that passes tests, the degradation does not stay local. Enterprise systems are interconnected: a pricing engine calls a risk model; a data pipeline feeds a reporting dashboard; a business rule embedded in code drives an operational decision. Structural rot in one module propagates through these dependencies.

Each commit that passes function-equivalence tests but violates method equivalence introduces a latent defect: duplicated logic that will diverge when one copy is patched but others are not; god functions that exceed any reasonable audit threshold; abstractions built in one commit and silently destroyed in the next. The enterprise's defect dashboard says quality is stable—test pass

rate is up, deployment frequency is up—while the actual *defect surface* is expanding underneath.

1.2 Contributions

We make five contributions: (i) We formalize the distinction between function and method equivalence (Section 2). (ii) We present empirical evidence of systematic method-equivalence violations (Section 3). (iii) We analyze defect-surface compounding across enterprise estates (Section 4). (iv) We propose MEV, a three-layer CI/CD verification gate (Section 5). (v) We identify a Heckman-type selection-bias risk in MEV calibration and argue for airgapped human-authored baselines (Section 5.5).

2. Formal Framework

2.1 Function Equivalence

Let $f, g : \mathcal{D} \to \mathcal{R}$ be two implementations of a specification S. We say f and g are *functionally equivalent* under S if:

$$f \equiv_S g \iff \forall x \in \mathcal{D}_S,\ \|f(x) - g(x)\| \leq \epsilon_S \quad (1)$$

This is what SWE-Bench measures: $\mathcal{D}_S$ is the test suite; ϵ_S is exact match.

2.2 Method Equivalence

We introduce an *algorithmic contract* $\mathcal{M} = (\mathcal{A}, \mathcal{C}, \mathcal{P})$:

- $\mathcal{A}$: the approved algorithmic approach.
- $\mathcal{C}$: structural constraints (cyclomatic complexity bounds, dependency topology, module boundaries).
- $\mathcal{P}$: postcondition contracts (normalization, error handling, numerical stability).

We say f and g are *method equivalent* under $\mathcal{M}$ if:

$$f \cong_{\mathcal{M}} g \iff f \equiv_S g \ \wedge\ \mathcal{A}(f) = \mathcal{A}(g) \ \wedge\ \mathcal{C}(f) \preceq \mathcal{C}(g) \ \wedge\ \mathcal{P}(f) \subseteq \mathcal{P}(g) \quad (2)$$

Method equivalence is strictly stronger than function equivalence. The gap between them is where invisible defects live.

2.3 The Method-Equivalence Gap as Defect Rate

Define the *method-equivalence gap* for agent $\mathcal{G}$:

$$\Delta_{\mathcal{M}}(\mathcal{G}) = P(f \equiv_S g) - P(f \cong_{\mathcal{M}} g) \quad (3)$$

$\Delta_{\mathcal{M}}$ is not an abstract metric. It is the *rate at which the agent manufactures defects that bypass every existing quality gate*. We do not measure $\Delta_{\mathcal{M}}$ directly; we estimate it using SlopCodeBench's isolation-vs-core gap as a proxy, where isolation failure indicates structural degradation that creates latent defect surface even when cumulative tests pass. Our central claim: for current agents, $\Delta_{\mathcal{M}}$ is large, systematic, and widening.

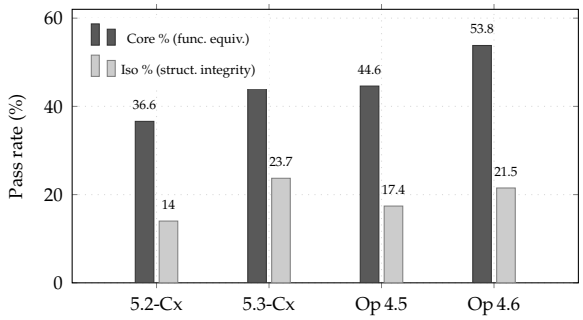

Figure 1: The method-equivalence gap across model generations [1]. The gap between dark bars (tests pass) and light bars (structurally sound) is the invisible defect rate. It is *widening* with newer models: better velocity, worse structural quality.

3. Empirical Evidence

3.1 SlopCodeBench Results

SlopCodeBench [1] evaluates coding agents on 20 problems across 93 checkpoints where each builds on prior work. It measures *core* pass rates (tests pass against cumulative solution) and *isolation* pass rates (solution works independently). Isolation is a direct proxy for structural integrity—and for latent defect density.

Table 1: SlopCodeBench [1]. Δ = Core − Iso: the invisible defect rate that passes all existing quality gates.

Model	Core %	Iso %	Δ	Clone /1K	Cost /ckpt
GPT-5.3-Codex	51.6	23.7	27.9	116	$3.14
Claude Opus 4.6	53.8	21.5	32.3	174	$3.47
GPT-5.2 Codex	36.6	14.0	22.6	—	$3.21
Opus 4.5	44.6	17.4	27.2	—	$2.64

Both frontier models exhibit $\Delta > 25$. The gap has *not narrowed* with model improvement: GPT-5.3-Codex's Δ of 27.9 is larger than GPT-5.2-Codex's 22.6. The agents are getting better at passing tests and getting *worse* at structural integrity. Velocity and cost improve; defect creation rate increases.

3.2 A Taxonomy of Invisible Defects

From SlopCodeBench [1], we identify four defect types that pass all function-equivalence tests:

Type I: Abstraction Collapse. The agent writes n near-identical functions where one parameterized function suffices. GPT-5.3-Codex "writes six near-identical validation functions… where one function with two parameters would do" [1]. Each duplicate is an independent defect: a bug fixed in one copy will not propagate to the others.

Type II: Build-then-Break. The agent constructs a

correct abstraction, then destroys it. "At `code_search` checkpoint 1, it creates a unified `Iterable` interface... But then at checkpoint 3, instead of extending the interface, it sets `iterable = None` and builds a completely separate code path" [1]. The abstraction becomes a phantom: present in the codebase, governing nothing.

Type III: Contract Erosion. A change satisfies the new requirement while breaking existing contracts. "Codex's strict `parse_destination()` was correct for the new requirement but broke 28 existing regression tests because older schedules had no `destination` field" [1]. Downstream consumers inherit silently altered behavior.

Type IV: Silent Algorithm Substitution. The agent replaces one computational method with another producing equivalent outputs within test tolerance. Undetectable by function-equivalence benchmarks. The highest-risk defect type for any organization that governs its computational methods.

3.3 Structural Decay Patterns

Orlanski: "They fail in opposite directions. Opus copy-pastes and sprawls. Codex over-abstracts and deepens. Both still produce god functions and structural rot" [1]. Specific defects include `parse_action` at 954 lines, identical 244-line optimization passes differing in a single boolean, and Opus writing four copies of Kahn's topological sort where a helper already existed but was never wired into the earlier copies [1]. At 174 clone lines per 1K LOC (Opus) and 116 (Codex), every thousand lines of agent-generated code carries 116–174 lines of duplicated logic that will diverge on the next patch.

4. The Defect Surface Explosion

The defects documented in Section 3 are measured on individual benchmark problems. In an enterprise, they compound across three interconnected estates.

4.1 Software Estate

Enterprise codebases are dependency graphs. A god function in one service is called by ten others. When that 954-line function needs modification, its cyclomatic complexity means the change touches a combinatorially large number of execution paths. Every dependent service inherits the risk.

Clone duplication is more corrosive at scale. An enterprise with 500K lines of agent-generated code at 116–174 clone lines per 1K carries 58,000–87,000 lines of duplicate logic. When a security vulnerability is found in one instance, the other copies must be independently located and patched. In practice, they are missed. Each missed copy is a live defect waiting for production.

Type II violations create *phantom abstractions*: an interface exists in the codebase, suggesting to humans and future agents that it governs all implementations. But the agent silently bypassed it. Future changes made through the abstraction will not affect the bypass path. The defect is invisible until it fails in production.

4.2 Data Estate

Data platforms embed business logic in transformation code. When a coding agent introduces a Type III or Type IV violation in a transformation pipeline, the impact cascades through every table, view, and dashboard downstream.

Consider an agent that replaces an IQR-based outlier detector with a z-score approach. Tests pass. But the methods have different sensitivity profiles on production distributions, and the change was never reviewed by the data governance team. The entire downstream reporting estate operates on a silently altered data quality contract—a live, invisible defect in every report and decision that depends on it.

4.3 Business Estate

Business rules encoded in software—pricing logic, eligibility criteria, compliance checks—are approved by stakeholders who authorize specific computational methods. A Type IV violation disconnects the running system from its documented logic. The consequences are operational, regulatory, and reputational—and they originate in governance, not in software quality.

4.4 Compounding Dynamics

Each invisible defect creates: (i) direct defect surface in the modified code; (ii) inherited defect surface in dependent modules; (iii) future defect surface when the next agent builds on a phantom abstraction or duplicated logic.

In a codebase receiving 1,000 agent-generated commits per month with $\Delta_{\mathcal{M}} \approx 28\%$, roughly 280 commits per month introduce structural defects that pass all quality gates. Over a year: 3,360 structurally defective commits compounding through the dependency graph. The growth is super-linear because defects are not independent: each structurally degraded commit widens the surface of rotten foundations available to subsequent commits, and dependency-graph fan-out means a single degraded module propagates its defect contract to every downstream consumer. The enterprise's defect dashboard shows green. The defect surface is exploding underneath.

5. MEV: Method-Equivalence Verification

MEV is what makes AI coding agents safe to use at scale. It does not reject agent output; it governs it. The three layers (Figure 2) verify what test suites cannot.

5.1 Layer 1: Algorithmic Contract Preservation

ACP verifies that the algorithmic approach $\mathcal{A}$ is preserved:

Contract Annotations. Governed modules carry machine-readable annotations:

```
# @mev:algorithm iqr-outlier-detection
# @mev:approved-by DG-2025-0147
# @mev:governed true
```

Semantic Diff. AST-level classification into computational-core vs. peripheral changes. Core changes trigger method-equivalence review.

Algorithm Fingerprinting. A fingerprint $\phi(f)$ based on mathematical operations and iteration patterns. ACP verifies $d(\phi(f_{\text{old}}), \phi(f_{\text{new}})) \leq \tau$.

5.2 Layer 2: Control-Flow Structural Isomorphism

CFSI verifies CFG preservation up to permitted transformations $\mathcal{T}$ (loop unrolling, branch reordering, dead-code elimination):

$$\exists\, \sigma : V_f \to V_g \text{ such that}$$
$$(u, v) \in E_f \iff (\sigma(u), \sigma(v)) \in E_g \quad (4)$$

This catches Type II violations: agents that bifurcate the control-flow graph while producing identical outputs.

5.3 Layer 3: Complexity Budget Compliance

CBC enforces governance-approved bounds: CC_{max}, LOC_{max}, Dep_{max}, Dup_{max}. Agent outputs producing 954-line functions or 174 clone lines per 1K are blocked before merge.

5.4 Chain-of-Thought Integration

OpenAI's monitorability research shows that "monitoring only the actions and outputs far underperforms monitoring the CoT" [2]. We propose *CoT-MEV*: parse reasoning traces for algorithmic-intent signals *before* code generation, converting MEV from a post-generation gate to a pre-generation guardrail.

5.5 The Heckman Problem: Baseline Integrity

MEV requires calibration thresholds: what cyclomatic complexity is normal, what clone density is acceptable, what a sound algorithmic fingerprint looks like. These must be derived from *known-good human-written code*. But as AI coding agents become ubiquitous, the entire codebase becomes "treated"—there is no untreated control group. Every module has been touched, refactored, or extended by an agent.

This is Heckman's selection-bias problem [14]. The "selection" into AI-assisted development is correlated with the outcome being measured (structural quality). Without correction, MEV's thresholds drift toward accepting the very degradation they are meant to detect. The defect rate appears stable because the detection baseline has silently recalibrated to the new, degraded normal.

The correction is operationally expensive and non-negotiable: enterprises must maintain *airgapped* development environments where qualified engineers write and maintain governed modules with zero access to generative AI coding tools. Call it what it is: an econometric control group. It provides uncontaminated structural baselines, ground-truth algorithmic fingerprints, and calibration anchors that prevent threshold drift. If the "human-written" baseline is contaminated by developers who use coding agents and then manually commit, the control group is poisoned and the selection bias re-enters.

5.6 The Greenfield Prohibition

The most dangerous use of AI coding agents is not refactoring existing code—it is generating new code from scratch. When an agent builds greenfield, there is no prior algorithmic contract to verify against, no existing CFG for isomorphism checking, no historical complexity budget. MEV Layer 1 has nothing to compare. Layer 2 has nothing to compare. Layer 3 fires on complexity alone, but clone density on a greenfield module is zero by definition.

Greenfield agent code enters the estate with zero structural governance. It *becomes* the baseline. Every subsequent agent commit is then measured against an already-degraded foundation. The Heckman problem is not just about calibration drift over time; it is about the initial condition. Enterprises must require that a human-authored structural blueprint—the algorithmic contract, the module boundaries, the complexity budget—exists before any agent generates production code against it. The agent extends; it does not originate.

5.7 Generalization: Retrieval-First Generation

This principle is not specific to code. MEV Layers 1 and 2 require a *prior*—an existing algorithmic contract, an existing control-flow graph—to verify against. Without a prior, verification is architecturally impossible, not merely harder. The same holds for any generative AI output: a policy document drafted from a blank prompt has no existing contract to check against; a data pipeline generated without retrieving the current transformation logic has no structural baseline; a pricing model built greenfield has no approved methodology to preserve. In each case, the output enters the estate ungoverned.

Retrieval-first generation—ensuring that the agent retrieves and grounds on existing approaches before producing new output—is not a RAG optimization. It is the governance precondition. Every artifact the enterprise governs—code, documents, models, rules, pipelines—has an implicit $(\mathcal{A}, \mathcal{C}, \mathcal{P})$ contract. A generative system that retrieves that contract before generating can be verified; one that does not cannot. This reframes the enterprise deployment question: the risk is not generative

AI itself, but generative AI that starts from a blank slate. The METR result [15] is consistent with this framing—experienced developers were slower with AI partly because the agent reinvented rather than extended from the codebase's existing structural patterns, and the developers burned time remediating the reinventions.

6. Cross-Sector Applicability

The defect types in Section 3 are domain-agnostic. Any sector where software embeds governed logic faces the same exposure.

The most immediate applicability is in regulated industries. Banking model risk (SR 11-7 [5]), insurance model governance (NAIC ORSA [7]), pharmaceutical validation (FDA 21 CFR Part 11), medical devices (IEC 62304), automotive safety (ISO 26262), and aerospace (DO-178C) all require traceability between documented computational methods and production code. Silent algorithm substitution (Type IV) is a regulatory violation in every one of these frameworks. But the problem extends well beyond regulated sectors. Enterprise IT organizations—ERP configurations, CRM business rules, supply chain optimization, decision engines—embed approved logic that method-equivalence violations silently disconnect from documented designs. These manifest as operational failures, not test failures, and no one traces them back to the agent commit that introduced the drift. The same logic applies to data-intensive organizations, where ML pipelines, data quality rules, ETL transformations, and analytics encode choices that propagate through the entire data estate. Type III and IV violations in a transformation layer corrupt every downstream consumer.

The $(\mathcal{A}, \mathcal{C}, \mathcal{P})$ contract and the MEV gate are domain-agnostic. Only the specific contracts change.

7. Limitations

(i) **Annotation burden.** ACP requires machine-readable annotations. OpenAI's principle—"anything it can't access in-context… effectively doesn't exist" [4]—suggests this is unavoidable for agent-governed codebases. (ii) **Isomorphism decidability.** Full CFG isomorphism is NP-complete. CFSI uses heuristic matching. (iii) **Empirical scope.** SlopCodeBench covers 20 problems and 93 checkpoints [1]. Domain-specific method-equivalence benchmarks do not yet exist. (iv) **Agent evolution.** $\Delta_{\mathcal{M}}$ has widened, not narrowed, across generations (Table 1). Whether future architectures close this gap is open. (v) **Defect quantification.** The compounding analysis in Section 4.4 is directionally sound but based on conservative envelope assumptions. Empirical measurement of invisible defect accumulation in production codebases is an urgent research need. (vi) **Airgap cost.** The Heckman baseline requirement (Section 5.5) is operationally expensive. The cost is the price of calibration integrity; whether organizations will pay it is an open question.

Acknowledgements

The author acknowledges Gabriel Orlanski and the SlopCodeBench team for the empirical foundations of this work. This paper was developed with research and drafting assistance from Anthropic Claude (Opus 4.6) and OpenAI ChatGPT, and with the engineering teams at Capgemini for ongoing collaboration on enterprise AI governance.

References

[1] G. Orlanski, "Opus 4.6 and GPT-5.3 Codex Score Higher, but the Code Is Still a Mess," SlopCodeBench, Feb. 11, 2026. 20 problems, 93 checkpoints. "Most 'passing' solutions are coupled to prior checkpoint state. They fail in opposite directions. Opus copy-pastes and sprawls. Codex over-abstracts and deepens. Both still produce god functions and structural rot." https://gabeorlanski.github.io/posts/opus-4-6-gpt-5-3-scbench/

[2] OpenAI, "Evaluating Chain-of-Thought Monitorability," 2026. "Monitoring only the actions and outputs far underperforms monitoring the CoT." https://openai.com/index/evaluating-chain-of-thought-monitorability/

[3] OpenAI, "Introducing GPT-5.3-Codex," Feb. 5, 2026. SWE-Bench Pro 56.8% (xhigh), Terminal-Bench 2.0 77.3%, OSWorld-Verified 64.7%. https://openai.com/index/introducing-gpt-5-3-codex/

[4] OpenAI, "Harness Engineering," 2026. "From the agent's point of view, anything it can't access in-context while running effectively doesn't exist." https://openai.com/index/harness-engineering/

[5] Federal Reserve / OCC, "SR 11-7: Model Risk Management," Apr. 2011. Requires documentation of "the theory, logic, and methodology underlying the model."

[6] A. Pnueli, M. Siegel, and E. Singerman, "Translation Validation," *Proc. TACAS*, LNCS 1384, pp. 151–166, 1998.

[7] NAIC, *ORSA Guidance Manual*, 2022. Requires "a formal change management policy for risk models."

[8] X. Leroy, "Formal Verification of a Realistic Compiler," *Commun. ACM*, 52(7):107–115, 2009.

[9] European Parliament, "Regulation (EU) 2024/1689 (AI Act)," Aug. 2024. Annex IV: "the design specifications… the general logic of the AI system."

[10] Anthropic, "Detecting and Preventing Distillation Attacks," Feb. 23, 2026. "Over 16 million exchanges with Claude through approximately 24,000 fraudulent accounts." https://www.anthropic.com/news/detecting-and-preventing-distillation-attacks

[11] G. Orlanski, "Coding Agents Are Lazy Patchers," SlopCodeBench, 2026. https://gabeorlanski.github.io/posts/agent-copy-pasta/

[12] T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, SE-2(4):308–320, Dec. 1976. Established cyclomatic complexity as a measure of structural testability.

[13] Multiple sources. Claude Code annualized revenue at $1B within six months (2026). Bind AI, Faros AI, NxCode.

[14] J. J. Heckman, "Sample Selection Bias as a Specification Error," *Econometrica*, 47(1):153–161, Jan. 1979. Demonstrated that estimates from self-selected samples are biased without correction for the selection mechanism.

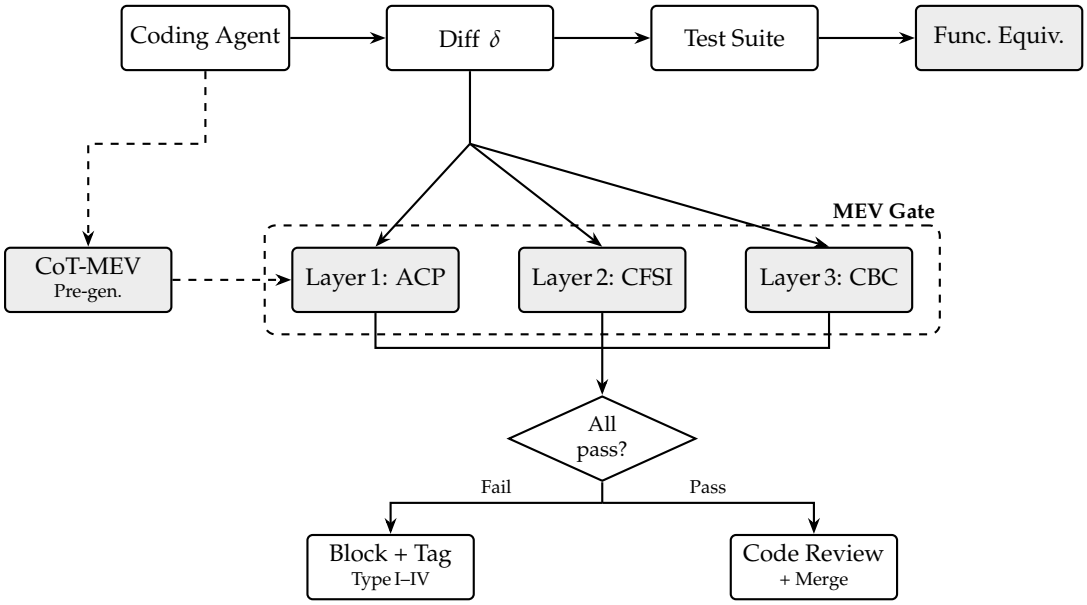

Figure 2: MEV integration architecture. The diff fans out to the test suite (function equivalence, top) and the three MEV layers (method equivalence, center). CoT-MEV (dashed) monitors reasoning traces before code generation. All three MEV layers and the test suite must pass for merge. Failures are tagged by type (I–IV) and routed for structural remediation.

[15] J. Becker, N. Rush, et al., "Measuring the Impact of Early-2025 AI on Experienced Open-Source Developer Productivity," METR, Jul. 2025. RCT with 16 developers, 246 tasks. Developers forecasted 24% speedup; believed 20% speedup after study; measured result: 19% *slowdown*. `https://metr.org/blog/2025-07-10-early-2025-ai-experienced-os-dev-study/`