Technical Foundation Document

IACDM: Interactive Adversarial Convergence Development Methodology

A Structured Framework for AI-Assisted Software Development

Jasmine Moreira

2025–2026

Abstract

The widespread adoption of AI-assisted development tools in 2025 exposed a critical and tool-agnostic failure pattern: experienced developers who used frontier AI models were measurably slower in objective evaluations despite believing they were faster [METR, 2025]. Concurrently, 10.3% of AI-generated applications in a production showcase contained critical security flaws — a single engineer was able to compromise multiple applications in 47 minutes [Palmer, 2025]. This paper argues that these failures share a structural cause — the *verification gap*: every large language model (LLM), regardless of interface or capability, operates as a stochastic generator with zero internal semantic verification capability — absent external tool use, no model can determine whether what it generated is correct. The tool is irrelevant; the process is determinative.

We present IACDM (*Interactive Adversarial Convergence Development Methodology*), a structured 8-phase framework that compensates for the verification gap through external verification agents (VA) operating at discrete gates. In each phase, the AI alternates between generative and adversarial roles — building artifacts and then systematically attacking them through specialized lenses — until the design reaches sufficient robustness for implementation. This repositions AI from passive code generator to active design partner. Its three pillars are: (1) deep problem discovery via Hierarchical Semantic Analysis before any technical solution; (2) persistent knowledge management across sessions; and (3) systematic adversarial critique through specialized lenses before implementation. The methodology is tool-agnostic by construction, grounded in established software engineering tradition, and developed and validated in a production R&D environment (N=1; see Section 10 for explicit limitations). Limitations are formalized as testable hypotheses for future empirical validation.

Keywords: AI-assisted software development, adversarial critique, verification gap, methodology, large language models, software engineering, red teaming, iterative design

Repository: github.com/jasminemoreira/Versus

VSCode Extensions: The methodology is available as two Visual Studio Code extensions for direct use with Claude and GitHub Copilot:
`JasmineMoreira.versus-claude` `JasmineMoreira.versus-copilot`

Contents

1. Motivation: the vibe coding phenomenon and its limits

In 2025, adoption of AI-assisted development tools reached critical mass. Platforms such as Lovable, Bolt.new, Replit, Cursor, Claude Code, and ChatGPT enabled anyone — with or without programming experience — to generate complete applications from natural language descriptions. The phenomenon, coined *vibe coding* by Karpathy [2025], transformed public perception of who can build software and at what speed.

The numbers are striking: Lovable reached US$100M in annual recurring revenue (ARR) in 8 months [Lovable, 2025]; 25% of Y Combinator's Winter 2025 batch built startups with over 95% AI-generated code [Y Combinator, 2025]; the 2025 Stack Overflow Developer Survey reported that 65% of developers use AI tools daily or weekly [Stack Overflow, 2025]. The productivity case is compelling and, for rapid prototyping, frequently borne out. However, 2025 empirical data reveals a consistent failure pattern that motivated the development of Iterative Convergence.

1.1. Empirical evidence: the cost of the verification gap

The METR study (2025). A randomized controlled trial (RCT) by METR — Model Evaluation & Threat Research — recruited 16 experienced developers to complete 246 tasks in open-source repositories where they had an average of 5 years of prior experience [METR, 2025]. The study used Cursor Pro with Claude 3.5/3.7 Sonnet as the primary AI tool. A notable three-way divergence emerged between what developers expected, what they felt, and what was measured: initial forecasts predicted a 24% speedup; post-task self-assessment reported a 20% speedup; objective measurement revealed a 19% slowdown [METR, 2025]. The 39-percentage-point gap between perception and reality is the central empirical motivation for this work.

CVE-2025-48757: Lovable. In March 2025, security researcher Matt Palmer developed an automated script to scan applications from Lovable's showcase platform. Of the 1,645 projects analyzed, roughly 10.3% — 170 applications — exposed sensitive data through misconfigured or absent Row Level Security (RLS) policies, totaling 303 vulnerable endpoints leaking names, emails, payment data, API keys, and personal addresses [Palmer, 2025]. The root cause was structural: Lovable's client-driven architecture delegated security responsibility to application developers, most of whom lacked the expertise to implement correct RLS configurations. A Palantir engineer independently demonstrated active exploitation during a lunch break, compromising multiple applications within 47 minutes [Palmer, 2025].

Degradation at scale. A five-year longitudinal analysis by GitClear across 211 million changed lines of code revealed a structural shift in how AI-assisted code is composed [GitClear, 2025]. Between 2021 and 2024, the share of changes attributable to refactoring activity fell from 25% to under 10%, while copy-pasted code grew from 8.3% to 12.3% — with 2024 marking the first year in GitClear's data where copy-pasted lines exceeded moved lines. The 2025 Stack Overflow Developer Survey registered the first significant decline in positive sentiment toward AI tools — from over 70% in 2023 and 2024 to 60% in 2025 — even as adoption rose to 84% of respondents; only 3% of developers report high trust in AI output accuracy [Stack Overflow, 2025].

1.2. The central observation: the tool is irrelevant

The critical point is that these problems are not specific to any tool. Lovable uses Claude. Claude Code uses Claude. ChatGPT uses GPT. Cursor uses both. All operate as stochastic generators:

$$P(\text{token}_n \mid \text{token}_1, \ldots, \text{token}_{n-1})$$

None verify what they produce internally. **Absent external tool use, semantic verification capability is zero in every case** — a model cannot determine whether what it generated is correct without feedback from outside itself.

The METR study confirms this: developers used Cursor Pro with frontier models — tools considered "serious" — and were still slower. Differences between interfaces (visual, terminal, chat), integrators (Supabase, GitHub), or models (Claude, GPT) are cosmetic from the verification gap perspective. The real distinction is not between tools. It is between two modes of operation:

Without methodology (any tool)	**With Iterative Convergence (any tool)**
`prompt → GA → artifact → delivery`	`design → G₀ → critique → G₁ → ...→ Gₙ → delivery`
Verification gap open	Discrete gates with VA-automatic and VA-human
Errors detected at t = production	Errors detected at $t = \text{gate}_k$
Perception: +20% | Reality: −19% [METR, 2025]	Measurable at each gate
170 vulnerable apps / 1,645 analyzed [Palmer, 2025]	Security verified by VA before delivery

GA = Generative Agent (the LLM); VA = Verification Agent (automatic or human); G_k = gate at phase k. Formal definitions in Section 8.

Core thesis: The quality of AI-assisted software is determined by the process, not the tool.

2. The problem: AI as a speed multiplier without direction

2.1. The nature of the generative agent

An LLM operates as a conditional distribution function, selecting each token based on the probability distribution conditioned on context. This process has three critical properties for software development:

1. **Absence of internal verification.** The model has no mechanism to evaluate the semantic, logical, or functional correctness of what it produces. It does not distinguish correct code from incorrect code — only more or less probable text given the context [Huang et al., 2023].

2. **Plausibility as proxy for correctness.** Output is optimized for statistical plausibility, not correctness. Code that "looks right" is favored regardless of its functional correctness [Dziri et al., 2023].

3. **Invariance of the generative engine.** The same stochastic mechanism operates regardless of whether the prompt is direct or structured. The conditioning quality changes; semantic verification capability, absent external tool use, remains zero.

These three properties are invariant across interfaces, models, and integration layers. Improving the tool does not close the verification gap; only external verification does.

2.2. The verification gap

Let $G(\text{prompt}) \to \text{artifact}$ be the LLM's generative function. For any artifact produced, there exists no internal function $V(\text{artifact}) \to \{\text{correct}, \text{incorrect}\}$ available to the model. This constitutes the **verification gap**: the space between what is generated and what is verifiable by the generator itself.

The cost manifests in three forms: *context erosion* (code generated without architecture produces monolithic artifacts that degrade subsequent interactions); *accelerated technical debt* (ease

of generation incentivizes quick fixes that accumulate coupling and inconsistencies — Lehman's law of increasing complexity [Lehman, 1980] operates at multiplied speed); and *validation theater* (developers ask the AI if the design is good; the AI, through RLHF alignment, tends to agree — and developers, susceptible to processing fluency as a proxy for truth [Kahneman, 2011], mistake a confident and well-formed response for a correct one).

3. Iterative Convergence: formal definition

IACDM (*Interactive Adversarial Convergence Development Methodology*) is a software development framework that organizes work into **8 phases (0–7)**, each with entry criteria, expected artifact, and exit gate. Throughout this document, *IACDM* and *Iterative Convergence* refer to the same method: IACDM is the formal acronym used in academic contexts; Iterative Convergence is the operational name used in the project repository and practitioner documentation. The method uses AI as a partner in each phase, alternating between builder and adversary roles, until the design reaches sufficient robustness for implementation.

Thesis 1: Software quality in AI-assisted development is determined by the quality of design preceding code generation, not by the model's generative capability.

Thesis 2: AI is most productive on granular work units with explicit scope, documented assumptions, and well-defined interfaces, maximizing the useful-information/context-consumed ratio.

Thesis 3: Deep problem understanding is a precondition for any technical decision. Understanding the wrong problem is orders of magnitude more expensive than understanding it slowly.

3.1. Overview of the 8 phases

#	Phase	Purpose	Primary artifact
0	Problem Discovery	Understand domain and problem before any technical solution	Validated document (score ≥90/100) + specs/ populated
1	Architecture	Define modules, interfaces, and technical contracts	Architecture V1 (∼2k tokens; author estimate, N=1)
2	Adversarial Critique	Attack architecture with 7 specialized lenses	Coverage matrix (modules × lenses)
3	Simplification	Simplify without introducing bugs; generate V(N+1)	Simplified architecture
4	Convergence Gate	Validate convergence with rigorous criteria	Final approved architecture
5	Code Implementation	Implement without regression; one module at a time	Functional code
6	Tests	Test against interfaces; mandatory manual testing	100% tests passing
7	Post-Review	Systematic learning; methodology evolution	Lessons + method improvement proposal

Table 1: The 8 phases of IACDM

Phases 2–3 form an **iterative loop** that repeats until convergence. Based on the author's

experience (N=1), small projects typically required 1–2 iterations, medium-complexity projects 2–3, and large or poorly understood problems 3–5; these figures are empirical estimates pending broader validation. The stopping criterion belongs to the human operator: structural change below 15% between consecutive versions and zero critical issues (both thresholds are author calibrations, N=1, subject to empirical revision).

3.2. Activation criteria

The methodology does not apply to every AI interaction. Activation is based on observable signals of complexity, not LOC estimation — since estimating lines of code before understanding the problem is precisely the kind of implicit assumption the method criticizes.

Golden rule: If you cannot confidently assert that the project is simple, it is not simple.

The risk asymmetry justifies this rule: the cost of applying the methodology to a project that turns out simple is low (approximately 20 minutes overhead in the author's experience, N=1; pending broader measurement). The cost of not applying it to a project that turns out complex is high (hours of rework).

3.3. Relationship to existing methods

IACDM integrates principles from diverse software engineering traditions:

Tradition / Method	Incorporated principle	Adaptation in IACDM
Spiral Model [Boehm, 1986]	Iterative cycles with risk analysis	Risk evaluated as design fragility; AI is risk analysis partner
Design by Contract [Meyer, 1992]	Pre/postconditions and invariants	Explicit assumptions and negative scope as informal contracts
Falsificationism [Popper, 1959]	Knowledge advances through refutation	Adversarial critique seeks to refute the design, not validate it
Red Teaming (NSA/DARPA)	Dedicated team to attack the system	AI assumes red team role under adversarial prompt
ATAM [Kazman et al., 2000]	Scenario-based architectural trade-offs	Each critique cycle evaluates trade-offs, forces explicit decisions
KISS/YAGNI [Beck, 1999]	Simplicity as value	Each iteration must simplify, not complexify
TDD [Beck, 2003]	Tests drive design	Adversarial critique as "design testing" before code
ADR [Nygard, 2011]	Decision documentation with context	Documented assumptions serve an equivalent role

4. Problem discovery and knowledge management

Phase 0 (Problem Discovery) is the most distinctive contribution of IACDM relative to traditional AI-assisted development methods. It formalizes a central observation: **understanding the wrong problem is orders of magnitude more expensive than understanding it slowly** — an estimate consistent with Boehm's cost-of-change curves [Boehm, 1986], though the exact multiplier depends on domain and detection phase.

4.1. The iterative discovery loop

Phase 0 operates as an iterative convergence cycle over problem understanding (not solution understanding), with four steps per iteration: (1) *Collection* — structured AI questions about domain, use cases, vocabulary, constraints, and negative scope; (2) *Synthesis (teach-back)* — the AI explains back what it understood in domain language; (3) *Validation (confrontation)* — the operator corrects or points out gaps; and (4) *Convergence assessment* — score 0–100 based on 10 weighted criteria, exit gate at score $\geq$90 + explicit operator confirmation.

The 10 criteria and their weights are: (1) domain vocabulary correctly used (10 pts); (2) problem statement without ambiguity (10 pts); (3) primary actors and use cases identified (10 pts); (4) constraints and non-functional requirements explicit (10 pts); (5) negative scope declared (10 pts); (6) success criteria defined and measurable (10 pts); (7) known unknowns listed (10 pts); (8) dependencies and external interfaces identified (10 pts); (9) teach-back accepted by operator without major correction (10 pts); (10) no open questions remain that would require revisiting criteria 1–9 after Phase 1 begins (10 pts). All weights are equal; this reflects the author's calibration (N=1) and is subject to revision as validation data accumulates. The gate requires score $\geq$90 *and* explicit operator confirmation to proceed — the confirmation is a separate condition, not a scored criterion.

The teach-back is particularly effective because it **inverts the direction of agreement bias**: instead of the AI confirming it understood (which it will always affirm), the human evaluates whether the AI's explanation is correct.

4.2. Hierarchical Semantic Analysis (HSA)

HSA structures exploration into five levels, each building on the previous, imposing epistemic order on the discovery process:

Level	Focus	What is sought
1. Domain	Universe where the problem exists	Vocabulary, theoretical field, state of the art
2. Problem	What needs to be solved	5W1H analysis
3. Elements	Parts that compose the problem	Components, entities, constraints
4. Processes	How the parts relate	Flows, dependencies, feedback loops
5. Product	What is expected at the end	Deliverables, acceptance criteria, negative scope

5W1H: Who, What, When, Where, Why, How — a structured questioning framework for systematic problem decomposition.

4.3. The knowledge repository (specs/)

The `specs/` directory is a persistent structured repository that functions as external project memory, addressing the finite context window of AI and the finite working memory of the human operator simultaneously.

Three properties make it methodologically significant. First, *traceability*: every technical decision traces to literature, and no algorithm is implemented without a bibliographic reference — this prevents the AI from substituting invented parameters for verified ones. Second, *inter-session persistence*: an operator resuming a project after weeks finds accumulated context intact, and the AI queries `specs/` before asking questions that were already answered in prior sessions. Third, *inter-version transferability*: version 2.0 starts from version 1.0's `specs/` as a baseline, preserving design rationale across rewrites and preventing regression in understanding.

5. Architecture before development

If the essential difficulty of software lies in conceptual construction [Brooks, 1987] — and AI tools, by embodying well-established implementation idioms and architectural patterns, primarily address accidental difficulty — then prior architectural investment is the only way to capture AI's productive contribution. This is an extrapolation of Brooks' framework: the essential difficulty (understanding the right problem, designing coherent abstractions) remains irreducibly human; the accidental difficulty (translating those abstractions into working code) is where AI contributes most reliably. Without prior architecture, AI accelerates production of artifacts that will need to be discarded.

Boehm's foundational research on software cost estimation established that the cost of defect correction grows by an order of magnitude for each phase of delay in discovery [Boehm, 1986]. IACDM anticipates discovery of conceptual defects to the design phase, when correction cost is orders of magnitude lower than implementation.

Architecture in this context answers four questions: *decomposition* (modules, responsibilities, boundaries — principle of separation of concerns [Dijkstra, 1974] and Single Responsibility Principle [Martin, 2003]); *interfaces* (communication contracts between modules); *assumptions* (what the system takes as true — Leveson's STAMP framework identifies flawed or outdated mental models of system state as a primary driver of accidents in complex systems [Leveson, 2011]; undeclared assumptions in software design play an analogous role); and *negative scope* (what the system deliberately does not do — explicit exclusion prevents scope creep from entering through silence, and gives the AI a boundary it cannot cross without operator approval).

6. Granularization: maximizing AI context efficiency

The most important operational constraint of current LLMs is not generation capability but the **finite context window**. Retrieval performance degrades for information positioned in the middle of long contexts — a phenomenon documented as *lost in the middle* [Liu et al., 2023] — with broader attention quality implications as context volume grows.

The context efficiency equation is a formal metaphor:

$$E = \frac{I_0}{C}$$

where E = context efficiency, I_0 = tokens carrying information directly relevant to the current task, C = total tokens consumed in the interaction. The ratio is not directly measurable without a relevance oracle, but it captures the intuition: in a monolithic system, $E \to 0$ as C grows unboundedly while I_0 (the current module's interface and logic) remains constant. In a granularized system, C is bounded per session and I_0 remains a large fraction of it. The equation is used qualitatively throughout this paper; controlled measurement is deferred to the validation roadmap (L3, Section 10).

Decomposition follows high internal cohesion and low external coupling [Constantine & Yourdon, 1979]: each module must be comprehensible by the AI in a single interaction, and interfaces — in the sense of Meyer's abstract module boundaries [Meyer, 1988] — function as compact summaries of what adjacent modules expose, allowing each session to remain focused and contextually dense. In the author's experience (N=1), monolithic sessions showed marked efficiency degradation beyond 50k tokens of accumulated context, while granularized sessions maintained near-unity efficiency throughout; these observations are qualitative and await controlled measurement.

7. Adversarial critique through specialized lenses

7.1. Why conventional validation fails

LLMs trained with RLHF exhibit a documented sycophantic bias: they tend to agree with, validate, or soften criticism of whatever the user presents [Sharma et al., 2023, Perez et al., 2023]. When asked "is this design good?", the model tends to agree — what Argyris [1977] called *single-loop learning*: accepting the problem framing and optimizing within it without questioning the framing itself. IACDM requires *double-loop learning*: questioning not only the solution but the assumptions that sustain it.

Generic critique also fails: it produces superficial observations without systematic coverage of quality dimensions. The same LLM operates all rounds — without operationally distinct criteria, diversification is illusory.

7.2. The specialized lens model

Phase 2 operates in **7 specialized rounds**, each with its own lens and central question. The lens selection principle: *a lens is legitimate only if removing it exposes a class of failure that no other lens detects.* Six lenses are universal (applicable to any software project); the Regulatory lens is conditional — active only in domains with traceable compliance requirements (healthcare, finance, data protection). All lenses should be evaluated for applicability in Phase 1 and documented as active or inactive with rationale:

Lens	Central question	Exclusive failure class
Assumptions	What does this design assume without declaring?	Failures from flawed/outdated system models [Leveson, 2011]
Architectural	Can each module be replaced, removed, or tested in isolation?	Hidden coupling, circular dependencies
Implementability	Can I code this module in one session with available context?	Incomplete specs, insufficient granularization
Scientific	Does every value, formula, and algorithm have a verifiable reference?	Invented parameters, plausibility-based logic
Security	How would an attacker exploit this surface with minimum effort?	Unanalyzed attack surface
Performance	Where are the bottlenecks and what is the asymptotic behavior?	Hidden bottlenecks, scale degradation
Regulatory	Does every regulatory requirement trace to a module?	Regulatory non-compliance

7.3. The critique–response asymmetry

An important property of the lens model is the asymmetry between critique and response: multiple agents to attack, unified agent to respond, human operator to arbitrate trade-offs. This asymmetry reflects ATAM practice [Kazman et al., 2000], where different evaluators attack by distinct quality attributes, but the architect resolves the trade-offs.

7.4. The coverage matrix

The primary artifact of Phase 2 is the **coverage matrix**: a two-dimensional map (modules × lenses) showing finding distribution by severity. The gate criterion is binary: zero critical findings to advance; each important finding requires explicit decision; suggestions can be postponed.

8. External verification model

8.1. Process architecture with gates

Iterative Convergence institutes a sequence of phases $P_0, P_1, \ldots, P_7$, each with entry specification, expected artifact, and exit gate:

Phase	Gate	Primary VA	What it verifies
0. Discovery	G_0: Score ≥90/100	VA-human	Problem understanding
1. Architecture	G_1: Design review	VA-human	All modules identified; interfaces between modules defined; no open questions that would block Phase 2
2. Critique	G_2: Coverage matrix	VA-human + GA (7 lenses)	Zero criticals; decision for importants
3. Simplification	G_3: Complexity audit	VA-human	Complexity ≤ previous version
4. Convergence	G_4: Convergence gate	VA-human	Structural change <15% vs. prior version (author calibration, N=1); zero critical findings; all important findings resolved or explicitly deferred with rationale
5. Code	G_5: Compile + lint	VA-automatic	Syntax, types, style
6. Tests	G_6: Test suite + manual	VA-automatic + human	Functional correctness + UX
7. Post-Review	G_7: Retrospective	VA-human	Lessons + method evolution

8.2. Definitions

- **Generative Agent (GA):** The LLM. Produces textual artifacts without verification capability.
- **Verification Agent (VA):** Any entity capable of evaluating an artifact's correctness. Two types: *VA-automatic* (unit tests, linters, compilers — binary verdict on specific properties) and *VA-human* (the operator — evaluates semantic adequacy, usability, domain correctness).
- **Gate (G_k):** A discrete point where progression is conditioned on VA approval.

Formally:

$$G_k : \text{artifact}_k \to \{\text{approved}, \text{rejected}\} \quad \text{where} \quad G_k = VA_1(\text{artifact}_k) \wedge VA_2(\text{artifact}_k) \wedge \ldots \wedge VA_n(\text{artifact}_k)$$

When G_k = rejected, the artifact is returned to the GA with VA feedback:

$$GA(\text{original\_prompt} + \text{VA\_feedback}) \to \text{artifact}'_k$$

The GA still does not verify — but now generates conditioned on **concrete error information**, qualitatively superior to the original prompt.

8.3. Model properties

1. **Generative invariance.** The GA's capability is identical in structured and unstructured processes. The methodology improves the process, not the model.
2. **Observability.** Each gate makes observable properties of the artifact that would be invisible without external verification. The number of observable properties increases with VA diversity.

3. **Temporal error localization.** In direct prompting, errors are detected at $t =$ production. In structured process, errors are detected at $t = \text{gate}_k$, where k is always prior to delivery.

4. **Reconditioning quality.** VA rejection feedback constitutes empirical evidence of failure, conditioning next generation qualitatively above the original prompt.

8.4. Adoption criterion

The methodology is justified when the cost of gates is lower than the expected cost of undetected errors. Conceptually:

$$\sum_k C_{\text{gate}_k} < \sum_j P(\text{error}_j) \times C_{\text{error}_j}$$

This formulation is illustrative rather than directly computable: $P(\text{error}_j)$ requires historical defect data that practitioners rarely have in advance, and C_{error_j} depends on detection time — which is precisely what the methodology influences. The practical reading is qualitative: gate cost is bounded and predictable; undetected error cost is unbounded and late. See limitation L3 (Section 10) for the measurement agenda.

COCOMO II research suggests that early phases such as design represent a minority of total project effort but have outsized influence on final cost — errors undetected at design phase propagate and multiply through implementation [Boehm et al., 2000]. For domains where C_{error} is high (healthcare, education, industrial R&D), the inequality tends to be satisfied even with significant gate cost. The criterion is evaluable *a priori*, before knowing the project size.

9. Antipatterns to avoid

1. **Validation theater.** Asking the AI "is this design good?" and accepting affirmation. Antidote: request refutation, never validation.

2. **Complexity inflation.** When critique reveals fragility, adding components. IACDM treats simplification as a first-class design obligation: each iteration must maintain or reduce architectural complexity relative to the previous version. Growth in component count without corresponding growth in understood requirements is a signal of design failure, not progress.

3. **Monolithic session.** Design, code, and tests in a single conversation. Context saturates. One session per phase, one phase per module.

4. **Implicit assumption.** Design that works only if unlisted conditions are true [Leveson, 2011].

5. **VA-human absence.** Relying exclusively on automated tests. Semantic adequacy and domain correctness require human judgment.

6. **Phase 0 bypass.** Starting with architecture without understanding the problem. Cost multiplier: orders of magnitude relative to early discovery [Boehm, 1986].

7. **Reference-free implementation.** Implementing domain algorithms without bibliographic reference. AI generates plausible code with invented parameters.

8. **Convergence perfectionism.** Infinite critique cycles seeking "perfect" design. The stopping criterion is defined at gate G_4 (Section 8): structural change below 15% relative to the prior version, zero critical findings, and all important findings resolved or explicitly deferred with rationale. When these conditions are met, the design is ready to evolve — not finished, but ready.

9. **Silent scope creep.** Adding features during Phases 2–3 without operator approval. Scope decisions belong to the human. AI can suggest; never decide.

10. **Verified-solution bypass.** AI generates plausible code for problems with established solutions, replacing tested implementations with stochastic approximations. Antidote: tier assessment per module in Phase 1 — if during Phases 5–6 the AI is doing trial-and-error on something deterministic, stop and search for a reference implementation.

10. Limitations and future work

10.1. Empirical limitations

L1 — Evidence circularity (N=1). The methodology was developed, validated, and documented by the same researcher on the same initial project. This validates applicability but not superiority to alternatives. It is not possible to separate the method's contribution from the operator's experience. *Testable hypothesis:* Operators with different experience levels applying IACDM on distinct projects produce results of measurably superior quality — as assessed by defect rate, rework time, or operator-reported confidence — compared to direct prompting.

L2 — Property 4 not empirically tested. The claim that VA error feedback produces "qualitatively superior conditioning" is intuitive but unmeasured. *Testable hypothesis:* Defect rate decreases monotonically with VA feedback cycles at each gate, up to a stabilization point.

L3 — Gate cost not measured. The adoption criterion is formally elegant but inoperationalizable in its current state. *Testable hypothesis:* Total overhead of Phases 0–4 represents 15–25% of total time and is compensated by rework reduction in Phases 5–6.

10.2. Partially addressed limitations

L4 — VA-human competence. The model assumes a competent operator, but the METR study demonstrates even experienced developers confuse fluency with correctness. Binary checklists, quantitative scores, and adversarial posture mitigate but do not eliminate this bias.

L5 — Team scalability. The method was validated with an individual operator. In teams, VA-human coordination, adversarial posture transfer, and systemic vision maintenance present incompletely addressed challenges.

10.3. Structural limitations

The GA/VA model does not model failure modes of the method itself when operated correctly. Additionally, the equation $E = I_0/C$ captures AI context efficiency but not the human operator's cognitive cost. The optimal granularization point — balancing E_{AI} and E_{human} — is not formalized.

10.4. Validation roadmap

#	Limitation	Validation method	Success metric
L1	Circularity (N=1)	Pilot projects, distinct operators	Quality > direct prompt in ≥2 projects
L2	Property 4	Gate instrumentation	Monotonic defect convergence
L3	Gate cost	Per-phase timing	Overhead < 25%, reduced rework
L4	VA-human competence	Novices vs. experienced	Checklists reduce perception–reality gap
L5	Team scalability	Project with team ≥3	Coordination via specs/ functional

Making limitations explicit is Property 2 (observability) applied to the method itself. What is not observable cannot be corrected.

10.5. Research opportunities

Beyond validating what this paper claims, the framework opens a set of research questions that were not previously formulable in this form. These are not gaps to be filled before the methodology is usable — it is usable now — but directions that the work makes visible for the first time.

RO1 — Teach-back as an independent mechanism. The teach-back inversion (Section 4.1) is the most empirically tractable contribution of this paper. A controlled experiment with two groups — one using direct confirmation ("did you understand?"), the other using structured teach-back — measuring the rate of misunderstanding detected before Phase 1 begins would produce a publishable result independently of IACDM as a system. The hypothesis is sharp: teach-back surfaces a measurably higher proportion of understanding errors than direct confirmation, because it shifts the evaluation burden from the model (which will always affirm) to the human operator (who can detect deviations from intent).

RO2 — The verification gap as a measurable construct. This paper defines the verification gap conceptually and supports it with indirect evidence (METR, Palmer, GitClear). A direct measurement is missing: the correlation between the presence or absence of structured external verification gates and defect rate in production, controlling for developer experience. The METR study comes close but was not designed for this question. A study instrumenting projects with and without gate-structured processes — measuring defect rate, rework time, and perception–reality gap — would give the verification gap empirical standing as a construct, not just a conceptual claim.

RO3 — Lens validity and coverage completeness. The 7-lens model rests on the design principle that each lens detects a failure class no other lens detects (Section 7.2). This is argued, not demonstrated. A multi-project study mapping which lens detected which finding — and whether any findings fell outside all lenses — would either validate the set or reveal missing dimensions. It would also test whether the lenses are operationally independent or whether practitioners systematically conflate certain pairs (the Assumptions and Architectural lenses are the most likely candidates for overlap).

RO4 — The optimal granularization point. The context efficiency metaphor $E = I_0/C$ (Section 6) points to a real optimization problem: what module size maximizes output quality without exceeding the human operator's cognitive integration cost? This is a two-objective problem — AI context efficiency and human working memory load — and the optimal point is likely domain-dependent. Empirical measurement across project types and operator experience levels would produce actionable guidance for practitioners and theoretical constraints for the equation's formalization.

RO5 — Methodology robustness under model capability improvement. The central argument of this paper is that the verification gap is structural and model-independent. This argument is currently unfalsified but also untested against the strongest available counterevidence: models with native tool use and self-correction capabilities. If future models can reliably verify their own output for well-specified problem classes, the verification gap partially closes — and the scope conditions of IACDM become a research question in their own right. What problem classes remain irreducibly dependent on external verification even as model capabilities improve? This is the long-horizon question that the SE 3.0 trajectory [Hassan et al., 2024] and IACDM together make urgent.

11. Synthesis

IACDM is a development discipline that repositions AI from code generator to design partner. It operates in 8 phases with discrete gates, sustained by three pillars: deep problem discovery before any technical solution (Phase 0, with Hierarchical Semantic Analysis), persistent knowledge management across sessions and versions (`specs/`), and systematic external verification at each phase transition (GA/VA model).

The external verification model formalizes the methodology's actual role: it **does not alter the model's capability** (Property 1 — Generative Invariance). What it does is institute a framework of external verification agents — automatic and human — operating at discrete gates, making errors observable and providing concrete feedback for reconditioning.

What changes is not what the model can do, but what the process can see. Errors that would be invisible to the generative agent become observable at gate boundaries; assumptions that would remain implicit become explicit artifacts subject to critique. The methodology's contribution is therefore **epistemological**: it transforms an opaque generation process into an observable generation-verification-correction cycle, where verification is always external to the generative agent. The 2025 empirical data confirms that this contribution is independent of the tool used. IACDM is, by construction, tool-agnostic: it addresses the process, not the model.

11.1. Positioning within the SE 3.0 literature

Recent work on the next era of software engineering converges on a diagnosis: the SE 2.0 model — AI as a task-driven copilot completing isolated code fragments — exposes structural limitations including cognitive overload, loss of architectural coherence, and verification gaps [Hassan et al., 2024]. Hassan et al. propose SE 3.0 as an intent-first, conversation-oriented paradigm in which AI evolves into an intelligent collaborator capable of understanding software engineering principles. IACDM shares the diagnosis and the directional shift, but differs in emphasis: where SE 3.0 frames the evolution as primarily a capability problem (building AI systems that reason more deeply about intent), IACDM treats it as primarily a process problem (structuring human-AI interaction so that errors become observable regardless of model capability). The two positions are complementary rather than competing. SE 3.0's vision requires that, as model capabilities improve, humans retain meaningful verification authority; IACDM operationalizes that authority through gates, lenses, and external verification agents that function independently of the underlying model. An empirical study of AI-assisted development practice with 40 senior

practitioners [Amasanti & Jahić, 2025] independently confirmed that asking AI assistance on small, focused problems maximizes productivity and that rework increases sharply with task complexity — consistent with IACDM's granularization and phase-scoping principles, though derived from different methodological premises.

"A converged design is not a finished design,
but one that is certainly ready to evolve."

References

Hassan, A. E., Oliva, G. A., Lin, D., Chen, B., & Jiang, Z. M. (2024). Towards AI-native software engineering (SE 3.0): A vision and a challenge roadmap. *arXiv:2410.06107* [cs.SE].

Amasanti, G., & Jahić, J. (2025). The impact of AI-generated solutions on software architecture and productivity: Results from a survey study. *arXiv:2506.17833* [cs.SE].

Argyris, C. (1977). Double loop learning in organizations. *Harvard Business Review*, 55(5), 115–125.

Beck, K. (1999). *Extreme Programming Explained*. Addison-Wesley.

Beck, K. (2003). *Test-Driven Development: By Example*. Addison-Wesley.

Boehm, B. (1986). A spiral model of software development and enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4), 14–24.

Boehm, B., Abts, C., Brown, A., Chulani, S., Clark, B., Horowitz, E., Madachy, R., Reifer, D., & Steece, B. (2000). *Software Cost Estimation with COCOMO II*. Prentice Hall.

Brooks, F. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20(4), 10–19.

Constantine, L., & Yourdon, E. (1979). *Structured Design*. Prentice Hall.

Dijkstra, E. (1974). On the role of scientific thought. *EWD447*.

Dziri, N., et al. (2023). Faith and fate: Limits of transformers on compositionality. *NeurIPS*.

GitClear. (2025). *AI Copilot Code Quality: 2025 Research*. Available at: `https://www.gitclear.com/ai_assistant_code_quality_2025_research` (accessed 2026).

Huang, J., et al. (2023). Large language models cannot self-correct reasoning yet. *arXiv:2310.01798*.

Kahneman, D. (2011). *Thinking, Fast and Slow*. Farrar, Straus and Giroux.

Karpathy, A. (2025). Vibe coding. *X/Twitter*, February 2025.

Kazman, R., Klein, M., & Clements, P. (2000). *ATAM: Method for Architecture Evaluation*. Technical Report CMU/SEI-2000-TR-004, SEI/CMU.

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9), 1060–1076.

Leveson, N. (2011). *Engineering a Safer World*. MIT Press.

Liu, N. F., et al. (2023). Lost in the middle: How language models use long contexts. *arXiv:2307.03172*.

Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.

METR. (2025). Measuring the impact of early-2025 AI on experienced open-source developer productivity. *arXiv:2507.09089*.

Meyer, B. (1988). *Object-Oriented Software Construction*. Prentice Hall.

Meyer, B. (1992). Applying design by contract. *Computer*, 25(10), 40–51.

Nygard, M. (2011). Documenting architecture decisions. *Cognitect Blog*. Available at: https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions (accessed 2026).

Lovable. (2025). Lovable reaches $100M ARR. *lovable.dev blog* (accessed 2026). Available at: https://lovable.dev/blog/100m-arr

Palmer, M. (2025). Statement on CVE-2025-48757: Lovable row level security vulnerability. *mattpalmer.io* (accessed 2026).

Perez, E., Ringer, S., Lukošiūtė, K., Nguyen, K., Chen, E., Heiner, S., Pettit, C., Olsson, C., Kundu, S., Kadavath, S., Jones, A., Chen, A., Mann, B., Israel, B., Seethor, B., McKinnon, C., Maxwell, T., Telleen-Lawton, T., Hatfield-Dodds, Z., Kaplan, J., Clark, J., Brown, T., McCandlish, S., Askell, A., & Ganguli, D. (2023). Discovering language model behaviors with model-written evaluations. *arXiv:2212.09251*.

Y Combinator. (2025). YC Winter 2025 batch statistics. *ycombinator.com* (accessed 2026). Available at: https://www.ycombinator.com/blog/yc-stats-w25

Popper, K. (1959). *The Logic of Scientific Discovery*. Hutchinson.

Sharma, M., Tong, M., Korbak, T., Duvenaud, D., Askell, A., Bowman, S. R., Cheng, N., Durmus, E., Hatfield-Dodds, Z., Irving, G., Kravec, S., Maxwell, T., McCandlish, S., Ndousse, K., Rausch, O., Schiefer, N., Yan, D., Ziegler, D., & Perez, E. (2023). Towards understanding sycophancy in language models. *arXiv:2310.13548*.

Stack Overflow. (2025). *2025 Developer Survey*. Available at: https://survey.stackoverflow.co/2025 (accessed 2026).