Das Terminierungs-Orakel: Konvergenz agentischer Softwareentwicklung gegen eine externalisierte, getypte Spezifikation

Cong Chanh Vinzenz Nguyen

19. Juni 2026

Abstract (English)

Agentic software development is increasingly organized as a loop: a generating agent proposes, a checking step accepts or rejects, and the result feeds the next iteration. We argue that the load-bearing problem of this loop is not the generator and not the iteration, but its *termination oracle* — the instance deciding when a proposal is „done“. A loop without a forgery-resistant termination oracle does not converge towards correctness but towards whatever its checking step happens to wave through. Our answer is not better generation but *convergence against an externalized, typed specification* that carries the termination oracle. We instantiate this in a method (`cong-driven-development`, a typed F# discriminated-union graph `SPOT` with the convergence states `Aligned`/`Pending`/`Diverged`/`Orphaned`) and two case studies: `runenruf` (existence proof, 46/46 tests, ~ 125 LoC, the honestly friendliest domain where the spec *is* the world) and `ledger-casestudy` (counterpart with real IO, a mutable requirement, 6/6 tests, and two Lean-proved, `sorry`-free invariants — value preservation and non-negativity). We separate the generator/oracle independence into two axes: a *mechanical* one (tool allowlist without generator write access, real and machine-checkable) and an *epistemic* one (decorrelation of spec author and code author, required only when the spec stems from a separate source). Every load-bearing claim is declared either **A** (machine- and externally verifiable: commit hash, arXiv id, CI run, `sorry` audit) or **B** (a human setting). We make no priority claim and prove no impossibility about machines: a second, sufficiently independent agent could set the ground invariant. We close with a gap, not a promise — two small case studies, one proved invariant, no unbounded autonomy: one screw turned further.

Inhaltsverzeichnis

1 Einleitung

Agentische Softwareentwicklung organisiert sich zunehmend als Schleife: ein generierender Agent erzeugt einen Vorschlag, ein Prüfschritt akzeptiert oder verwirft ihn, und das Ergebnis fließt in die nächste Iteration ein. Boris Cherny bringt diese Verschiebung auf den Punkt, wenn er die Tätigkeit des Entwicklers als „my job is to write loops“ beschreibt [8]. Wir argumentieren, dass das eigentliche Problem dieser Schleife nicht der Generator und nicht die Iteration selbst ist, sondern ihr *Terminierungs-Orakel*: jene Instanz, die entscheidet, wann ein Vorschlag „fertig“ ist. Eine Schleife ohne fälschungssicheres Terminierungs-Orakel konvergiert nicht gegen Korrektheit, sondern gegen das, was ihr Prüfschritt zufällig durchwinkt. Unser Lösungsansatz ist entsprechend nicht eine bessere Generierung, sondern die *Konvergenz gegen eine externalisierte, getypte Spezifikation*, die das Terminierungs-Orakel trägt.

Cognitive Debt und Intent Debt. Die Notwendigkeit einer externalisierten Spezifikation ist nicht bloß ergonomisch. Storey beschreibt einen Schuldbegriff jenseits der klassischen technischen Schuld: *Intent Debt* als „the absence of externalized rationale that developers and AI agents need to work safely with code“ [2]. Fehlt die externalisierte Absicht, so verlagert sich das Wissen darüber, *warum* ein Code korrekt sein soll, in nicht inspizierbare Zustände — in den menschlichen Kopf oder in den verworfenen Kontext eines Agenten. Lahiri formuliert die korrespondierende Aufgabe als benannte offene *Grand Challenge*: die Formalisierung von Intent als Voraussetzung verlässlicher KI-gestützter Programmierung [1]. Wir behandeln diese Externalisierung als Konstruktionsprinzip, nicht als Nebenprodukt.

Ehrlicher Stand der Technik. Drei Befunde rahmen die Ausgangslage und zugleich ihre Grenzen. Erstens zur Reichweite heutiger Agenten: METR misst einen „50%-task-completion time horizon“ und berichtet für Anfang 2025 eine Aufgabenlänge von etwa 50 bis 110 Minuten bei einer Verdopplung von rund 7 Monaten [6]; wir übernehmen ausschließlich diese Zahlen und leiten daraus keine Extrapolation ab. Zweitens zur prinzipiellen Schranke: nach dem Satz von Rice ist jede nicht-triviale semantische Eigenschaft von Programmen unentscheidbar [7]. Die Voraussetzung des Satzes — eine nicht-triviale semantische Eigenschaft — ist für „erfüllt die Spezifikation“ gegeben; daraus folgt aber nur eine unentscheidbare *obere Schranke* über alle Programme. Sie motiviert die entscheidbare, *pro Spezifikation* approximierende Prüfung, die wir verwenden, und ist insofern *Motivation*, nicht Existenzbeweis unserer Konstruktion. Wir betonen ausdrücklich: aus dem Satz von Rice folgt *nicht*, dass ein Mensch im Prüfvorgang unersetzlich wäre. Drittens zur Vertrauenswürdigkeit von Spezifikationen selbst: Misu, Ma

und Lopes zeigen mit VeriAct, dass *verifizierer-akzeptiert* nicht *korrekt* bedeutet — eine vom Verifizierer angenommene formale Spezifikation kann die Absicht verfehlen [4]. Dieser Befund ist die nächste Nachbarschaft unserer Arbeit und zugleich die Quelle ihres ehrlichsten Vorbehalts.

Leitfrage. Daraus ergibt sich die Frage, die dieses Papier strukturiert: *Lässt sich die agentische Schleife so konstruieren, dass ihr Terminierungs-Orakel an einer externalisierten, getypten Spezifikation hängt — und welcher Rest bleibt nach maximaler Mechanisierung notwendig menschlich?* Wir beantworten den ersten Teil konstruktiv anhand einer Methode und zweier Fallstudien; den zweiten Teil beantworten wir, indem wir den verbleibenden Rest exakt lokalisieren, statt ihn wegzuargumentieren.

A/B-Methodik dieses Papers. Wir unterwerfen jede tragende Behauptung einer expliziten Disziplin: sie deklariert sich entweder als **A** — maschinell und extern verifizierbar (Commit-Hash, `arXiv`-ID, CI-Lauf, `sorry`-Audit) — oder als **B** — eine menschliche Setzung (normative Wahl, Spezifikation-gegen-Welt-Bindung, Neuheit als Literaturaussage, Generalisierung über Domänen). Keine Aussage bleibt unmarkiert. Diese Markierung ist selbst der zentrale Hebel der Arbeit: Sie trennt, was Logik leistet, von dem, was der Mensch einbringt, und macht den „Rest“ prüfbar statt rhetorisch. Abschnitt 2 legt die Methode offen; die A/B-Tabelle in Abschnitt 6.5 bindet jede Marke an ihren Anker. Wir kündigen damit zugleich an, dass dieses Papier *mit* einer Lücke argumentiert: zwei kleine Fallstudien, zwei Lean-bewiesene Invarianten, keine unbeschränkte Autonomie — eine Schraube weiter, nicht das gelöste Problem.

2 Methode

Begriffe für Leser ohne F#/Lean-Hintergrund. Diese Arbeit nutzt Werkzeuge aus der funktionalen und formalen Programmierung; zur Einordnung jeweils eine grobe Analogie aus der OOP-/Java-Welt.

F# Funktionale Sprache auf .NET (gleiche Laufzeit wie C#), unveränderlich per Voreinstellung, mit Pattern Matching und algebraischen Datentypen — grob: Java mit `record`, `sealed` und Pattern-`switch`, durchgängig.

Discriminated Union (DU) Ein Typ, dessen Wert genau einer von mehreren Fällen ist (Summentyp); der SPOT-Graph ist so getypt. Analog: `sealed interface` mit Records und erschöpfendem `switch`.

Lean 4 Ein Theorembeweiser — man formuliert einen Satz, der Compiler prüft den Beweis, der dann für *alle* Fälle gilt (nicht stichprobenhaft wie ein Test). Am nächsten in der OOP-Welt: Dafny bzw. JML/KeY.

sorry-frei `sorry` ist Leans Platzhalter „Beweis ausgelassen“, eine Lücke, die dennoch kompiliert. *sorry-frei* bedeutet: keine solche Lücke; `#print axioms` weist die benutzten Grundannahmen aus (hier nur `propext`, `Quot.sound`).

Property-based Testing / FsCheck Statt fester Beispiel-Eingaben werden hunderte Zufallseingaben generiert und eine *Eigenschaft* geprüft; bei Fehlschlag wird auf das kleinste Gegenbeispiel „geshrinkt“. Analog: jqwik bzw. QuickCheck.

Orakel Der Schritt, der „fertig/korrekt?“ entscheidet — hier: ein Knoten wird nur `Aligned`, wenn ein echter Test grün ist, nicht weil „der Agent fertig sagt“.

2.1 Konvergenz gegen eine externalisierte Spezifikation

Den Kern bildet eine Umkehrung des Akzeptanzkriteriums. Statt „der Agent hält an, wenn er zufrieden ist“ gilt „der Agent hält an, wenn ein externes, getyptes Orakel die Konvergenz gegen die Spezifikation bestätigt“. Die Methode und ihre IDE-Realisierung liegen in `cong-driven-development` [9], ausgewertet am Commit `v0.7.0`. Das tragende Artefakt ist `SPOT` (*Single Point of Truth*): ein typisierter F#-Discriminated-Union-Graph, in dem jeder Knoten als ein `JSON` pro Knoten unter `.spot/` materialisiert ist. Der Konvergenzzustand jedes Knotens ist auf vier Werte getypt: `Aligned`, `Pending`, `Diverged`, `Orphaned`. Das Orakel `SetzeSpecAligned` setzt einen Test-Knoten genau dann auf `Aligned`, wenn seine Id als Marker im Testquellcode vorkommt (`covered`); die Funktion prüft Marker-Präsenz, keinen Testlauf. Dass diese Marker zu bestandenen Tests gehören, erzwingt die grüne CI-Suite (rot blockt den Merge) zusammen mit dem reflexiven Selbst-Test — daraus folgt `Aligned` $\Rightarrow$ Test-PASS, nicht bloßer Exit-Code 0. Das Selbst-Modell der Methode umfasst 66 Knoten, davon 62 `Aligned` und 4 `Pending`, bei 42/42 grünen Tests (autoritativer Runner-Count @ `v0.7.0`); der reflexive Selbst-Gate trägt den Knoten `spec-gate-selbst-hart`. Dass das Gate auf sich selbst angewandt grün ist, ist eine A-Aussage; dass es das *Richtige* prüft, ist eine B-Aussage (siehe Bemerkung 2.1).

2.2 Generator-/Orakel-Trennung — zweiachsig

Die Trennung von Erzeuger und Prüfer ist nicht eine einzige Eigenschaft, sondern zerfällt in zwei voneinander unabhängige Achsen.

Mechanische Achse (Generator $\perp$ Prüf-Orakel). Der Generator besitzt kein Schreibrecht auf das Prüf-Orakel. Technisch erzwungen wird dies durch eine Werkzeug-Allowlist ohne Schreibzugriff des Generators auf den Gate-Pfad; der Generator kann das Urteil über seine eigene Ausgabe nicht

fälschen. Diese Achse ist real und maschinell prüfbar — sie ist eine A-Aussage, verankert im Code @ `v0.7.0`.

Epistemische Achse (Generator ⊥ Spezifikation-Autor). Die zweite Achse betrifft die *Herkunft* der Spezifikation. Sie gilt nur, sofern die Spezifikation aus einer von der Codegenerierung *separaten* Quelle stammt. Erzeugt dieselbe Modell-Instanz sowohl Spezifikation als auch Code, so korrelieren ihre Fehler — ein *common-mode*-Versagen, genau der von VeriAct beschriebene Fall, in dem die verifizierer-akzeptierte Spezifikation mit dem Code gemeinsam die Absicht verfehlt [4]. Auf dieser Achse ist der Mensch der *Dekorrelator*: nicht weil eine Maschine die Rolle prinzipiell nicht einnehmen könnte — ein zweiter, unabhängiger Agent könnte die Spezifikation ebenso stellen —, sondern weil eine unabhängige Quelle *erforderlich* ist und in unseren Fallstudien der Mensch diese Quelle war. Konkret wurde der in `ledger-casestudy` gefangene Fehlschluss — ein Modell, das sich selbst als „Falsifiable, after 4 tests“ auswies — dadurch gefangen, dass ein Mensch eine stärkere Property und ein geändertes Requirement als Gegen-Orakel einbrachte; nicht die Schleife aus sich selbst heraus. Die Notwendigkeit der Dekorrelation ist strukturell (A: die Korrelation ist beobachtbar); *wer* dekorreliert, ist eine Setzung (B).

Bemerkung 2.1 (Spezifikation gegen Welt). Die mechanische Achse garantiert, dass der Code die Spezifikation erfüllt; sie garantiert nicht, dass die Spezifikation die *Welt* trifft. Die Bindung „Spezifikation $\models$ Welt“ ist nach VeriAct nicht orakel-prüfbar [4] und damit durchgängig eine B-Aussage. Wir behaupten an keiner Stelle Spezifikation-$\leftrightarrow$-Code-Äquivalenz.

2.3 Verschiebung des Akzeptanzkriteriums

Die Methode verschiebt das Akzeptanzkriterium von einer impliziten, generator-internen Zufriedenheit zu einem expliziten, externen, getypten Prädikat. Diese Verschiebung ist die operationale Antwort auf das Terminierungs-Orakel-Problem aus Abschnitt 1: Die Schleife terminiert genau dann, wenn der `SPOT`-Graph keinen `Diverged`- und keinen `Orphaned`-Knoten mehr trägt und jeder relevante Knoten über `SetzeSpecAligned` auf `Aligned` gehoben ist. Der Loop selbst bleibt unverändert agentisch; verändert ist allein, woran sein Haltepunkt hängt. Im Sinne von LLM-Modulo [3] ist der LLM der *Generator*, das getypte Orakel der *Kritiker*; im Sinne property-orientierten Feedbacks [5] liefert der `Diverged`-Zustand eine strukturell minimale, property-bezogene Rückmeldung statt eines diffusen „versuche es erneut“.

2.4 Abgrenzung

Spezifikationsgetriebene Werkzeuge wie Kiro oder GitHub Spec-Kit externalisieren Absicht ebenfalls, jedoch typischerweise als *natürlichsprachliche* Spezifikationsdokumente, deren Erfüllung nicht über ein getyptes, ausführbares Orakel mit fälschungssicherer Generator-/Prüfer-Trennung erzwungen wird. Der Unterschied liegt nicht im Vorhandensein einer Spezifikation, sondern in deren Typisierung, ihrer maschinellen Prüfbarkeit und der architektonisch erzwungenen Trennung nach Abschnitt 2.2. Wir erheben hieraus *keinen* Prioritätsanspruch: Uns ist keine publizierte Implementierung mit genau dieser Kombination bekannt — getypte externalisierte Spezifikation, architektonisch getrennter Generator und Orakel, Werkzeug-Allowlist ohne Generator-Schreibrecht sowie ein property- und beweis-verifiziertes Gate — , doch diese Neuheit ist eine *Literaturaussage* (B), kein Beweis. Sie kann durch einen einzigen Gegenbeleg fallen.

Bemerkung 2.2 (Setzen der Grund-Invariante). Eine Konsequenz der Konstruktion verdient explizite Trennung in eine strukturelle und eine normative Aussage. *(i) Strukturell (Regress):* Die Grund-Invariante, gegen die geprüft wird, ist im System nicht selbst-fundierbar — jede Begründung einer Invariante setzt eine weitere Invariante voraus; der Regress terminiert nur durch einen Anker *außerhalb* des Prüfvorgangs. Dies ist ein architektonisches Argument, kein im Papier geführter formaler Beweis. *(ii) Normativ:* Dass der *Mensch* diesen Anker setzt, ist eine *Wahl* — begründet darin, dass die Absicht extern zum Generator und verantwortungsbehaftet ist —, und *keine* Unmöglichkeitsaussage über Maschinen. Ein zweiter Agent könnte die Invariante setzen; der Mensch ist hier *verschoben*, nicht durch ein Theorem als notwendig erwiesen. Wir vermeiden bewusst jede Formulierung im Sinne von „unmechanisierbar“ oder „nur ein Mensch kann“; das wäre der Penrose/Lucas-Fehlschluss, den wir ablehnen. Weder Rice (als Motivation, Abschnitt 1) noch die Gödel/Tarski-Analogie (Abschnitt 5.4) tragen diese Aussage: aus keinem von ihnen folgt die Unersetzlichkeit eines Menschen.

3 Fallstudie A: Runenruf

Runenruf ist der *Existenzbeweis* der Methode: die Frage, ob eine getypte, externalisierte Spezifikation als Terminierungs-Orakel eines agentischen Loops überhaupt einen sorry-freien, deterministisch reproduzierbaren Konvergenz-Zustand erzeugen kann, wird hier an einer minimalen Domäne positiv beantwortet. Wir markieren die Domänenwahl sofort als das, was sie ist: Runenruf ist die *ehrlich freundlichste* Domäne, die wir kennen — eine reine Simulationsdomäne, in der die Spezifikation mit der Welt zusammenfällt (Bemerkung 3.2). Das Artefakt belegt Anwendbarkeit, nicht Skalierung.

3.1 Aufbau

Der Code liegt in `github.com/Koschnag/runenruf`, ausgewertet am Commit `56376a5`. Die Simulationsdomäne umfasst rund 125 Zeilen F#/.NET-Code @ `56376a5`. Über dem Generator-Loop steht das in Abschnitt 2.1 beschriebene Orakel `SetzeSpecAligned`, das einen Test-Knoten auf `Aligned` setzt, sobald ein Test-Marker ihn im Quellcode abdeckt; die CI-erzwungene grüne Suite bindet dieses `Aligned` an einen bestandenen Test — gewertet wird das Test-PASS, nicht der Prozess-Exitcode 0.

3.2 Verifikation

Der autoritative Test-Runner meldet 46/46 grüne Tests @ `56376a5`; die Zahl stammt aus der Runner-Ausgabe, nicht aus einem `grep` über Testnamen. Tragend ist eine FsCheck-Property mit dem Spec-Schlüssel `spec-siegel-lager-nichtnegativ`: sie quantifiziert über *jeden* Seed und *jede* Folge von Domänenoperationen und behauptet die Nichtnegativität des Siegel-Lagerbestands. Die Auswertung ist deterministisch — gleicher Seed erzeugt gleichen Lauf —, sodass ein Gegenbeispiel reproduzierbar bleibt und nicht von einem nichtdeterministischen Scheduler abhängt.

Bemerkung 3.1 (Verifizierbarkeitsachse). Die Aussage „46/46 grün“ ist eine **A**-Behauptung im Sinne von Abschnitt 1: maschinell extern verifiziert, an `github.com/Koschnag/runenruf` @ `56376a5` und den dortigen CI-Run gebunden. Die FsCheck-Property ist ebenfalls **A**: sie wird vom Runner ausgeführt. Was *nicht* aus diesen Läufen folgt, ist die Bindung der Spezifikation an eine externe Welt — diese fällt in **B**.

Bemerkung 3.2 (Grenze: die Spec ist die Welt). In Runenruf existiert kein externer Referent jenseits der Simulation. Die Spezifikation *ist* die Welt, daher entfällt die Spec-gegen-Welt-Bindung („Spec $\models$ Welt“), die nach VeriAct [4] nicht orakel-prüfbar ist und in unserer Klassifikation zu **B** gehört. Genau deshalb ist Runenruf ein Existenzbeweis und kein Realismus-Beweis: er zeigt, dass die Typen-Property-Maschinerie geschlossen *durchläuft*, nicht, dass sie etwas außerhalb ihrer selbst trifft. Der härtere Fall mit echtem IO und veränderlichem Requirement folgt in Abschnitt 4.

4 Fallstudie B: Ledger

Ledger ist das *Gegenstück* zu Runenruf: dieselbe Methode in einer Domäne mit echtem IO, einem über die Zeit *veränderlichen* Requirement und einer geschlossenen Drei-Schichten-Kette Typen $\rightarrow$ FsCheck $\rightarrow$ Lean-Beweis (Abschnitt 4.3). Hier fällt die Spec nicht mit der Welt zusammen, und genau hier zeigt sich, wo das Loop-interne Orakel endet und der menschliche Dekorrelator beginnt (Abschnitt 4.2).

4.1 Aufbau

Der Code liegt in `github.com/Koschnag/ledger-casestudy`, ausgewertet am Commit `0f39ab0`; F#/.NET 9, Beträge als `int64`-Cent, um Gleitkomma-Fehler an der Wertgrenze auszuschließen. Anders als in Abschnitt 3 gibt es echtes IO: ein persistentes Journal und einen Replay-Pfad. Die Kerndomäne umfasst rund 41 Zeilen F#-Code @ `0f39ab0`. Der autoritative Runner meldet 6/6 grüne Tests @ `0f39ab0`.

4.2 Gefangener Mis-Spec — der Mensch als Gegen-Orakel

Im `.spot/`-Ledger findet sich der Eintrag „Falsifiable, after 4 tests“. Wir attribuieren diesen Fund präzise: Er wurde *nicht* aus dem Loop heraus erzeugt. Die ersten, schwächeren Properties akzeptierten eine fehlerhafte Buchungslogik; verifizierer-akzeptiert ist eben nicht korrekt [4]. Gefangen wurde der Fehler erst, als ein *Mensch* eine stärkere Property *und* das geänderte Gebühren-Requirement als Gegen-Orakel einbrachte — erst diese externe Setzung machte die bis dahin grüne Spezifikation falsifizierbar. Das ist der zweiachsige Befund aus Abschnitt 2.2: Generator $\perp$ Prüf-Orakel ist mechanisch erzwungen, aber Generator $\perp$ Spec-Autor gilt nur bei separater Quelle. Erzeugt dieselbe Modell-Instanz Spezifikation *und* Code, korrelieren die Fehler (common-mode); der Mensch ist hier der Dekorrelator, nicht der Loop aus sich selbst.

Der Befund ist als reproduzierbares Negativ-Artefakt konserviert: `demo-gate-at-failure.sh` @ `0f39ab0` führt das Gate in den Falsifikationszustand und macht den Mis-Spec wiederholbar sichtbar. Anschließend wurde die Domäne auf doppelte Buchführung (Doppik) umgestellt, sodass jede Buchung wertneutral bleibt.

Bemerkung 4.1 (Verifizierbarkeitsachse). „6/6 grün“, der sorry-freie Lean-Lauf (Abschnitt 4.3) und die Existenz des Negativ-Artefakts sind **A**-Behauptungen, gebunden an `github.com/Koschnag/ledger-casestudy` @ `0f39ab0`. Die Aussage, *wer* die stärkere Property setzt, und dass die Doppik-Spec die buchhalterische Welt trifft, sind **B**: normativ bzw. Spec-gegen-Welt, nicht aus dem Lauf ableitbar.

4.3 Drei Schichten: Typen, FsCheck, Lean

Ledger trägt die einzige in dieser Arbeit *geschlossene* Drei-Schichten-Kette:

1. **Typen.** `int64`-Cent und die F#-Datentypen schließen ganze Fehlerklassen (Gleitkomma, negative Repräsentationslücken) bereits statisch aus.

2. **FsCheck.** Properties quantifizieren über Buchungsfolgen und prüfen Nichtnegativität, Isolation und Replay-Treue über zufällig generierte Eingaben.

3. **Lean-Beweis.** `proofs/Werterhaltung.lean` @ `0f39ab0` beweist die Werterhaltung für *jede* Buchungsfolge — ein Allquantor, der über das stichprobenhafte FsCheck hinausgeht.

Entscheidend für die Aussagekraft des Lean-Beweises ist, dass das Lean-Modell den Deckungs-/Ablehnungspfad *enthält* und damit modellgleich zur F#-Funktion `buche` ist: eine Buchung, die die Deckung verletzen würde, wird im Modell abgelehnt, genau wie im Code. Ein Beweis über ein Modell ohne Ablehnungspfad wäre vakuum-stark und würde die kritische Eigenschaft am Code vorbeibeweisen.

```
theorem werterhaltung (hb : Hauptbuch) (bs : List Buchung) :
    total (buchen hb bs) = total hb := by
  unfold buchen
  induction bs generalizing hb with
  | nil => rfl
  | cons b bs ih =>
      simp only [List.foldl_cons]
      rw [ih (buche hb b), werterhaltung_einzel]
```

Listing 1: Kern-Theorem in `proofs/Werterhaltung.lean` @ `0f39ab0` (verbatim; Definitionen und das Lemma `werterhaltung_einzel` im Repo)

Der Beweis (Listing 1) ist *sorry-frei*: `#print axioms werterhaltung` liefert ausschließlich `propext` und `Quot.sound`, also keine zusätzlichen Axiome jenseits des Lean-Kerns. Der Beweis kommt ohne Mathlib aus (nur `Core` + `omega`); der CI-Job `lean-proof` ist grün @ `0f39ab0`.

Bemerkung 4.2 (Was Lean hier *nicht* abdeckt). Zwei Invarianten — Werterhaltung und Nichtnegativität — sind Lean-bewiesen. Isolation und Replay-Treue sind ausschließlich durch FsCheck abgesichert. Die Drei-Schichten-Kette Typen → Property → Beweis ist derzeit für Werterhaltung und Nichtnegativität geschlossen; für die übrigen Invarianten fehlt die Beweis-Schicht. Wir rahmen das ausdrücklich als Lücke, eine Schraube weiter, nicht als vollständige Verifikation der Domäne.

4.4 Kompression

Über beide Fallstudien hinweg ist die verifizierte Invariante deutlich kürzer als ihre Implementierung: das Verhältnis liegt zwischen rund 40× und 125× (Abschnitt 3: ∼125 LoC Domäne; Abschnitt 4: ∼41 LoC Domäne, jeweils gegen die tragende Property bzw. das Kern-Theorem, jeweils im Kern eine Zeile). Wir formulieren die Lesart dieser Zahlen streng konditional und eng.

Bemerkung 4.3 (Kompression ist Diskrimination, keine Äquivalenz)*.* Die Invariante *fixiert* bzw. diskriminiert die kritische Korrektheitseigenschaft — Werterhaltung, Nichtnegativität —, sie *spezifiziert die Funktion nicht vollständig.* Unendlich viele Implementierungen erfüllen dieselbe Invariante; aus „Code $\models$ Invariante“ folgt keine Spec-Code-Äquivalenz. Die genannten Faktoren gelten konditional auf genau diese zwei Fallstudien ($N = 2$) und sind kein Beleg für eine domänenübergreifende Kompressionsrate. Die Größenangaben sind **A** (Code @ `56376a5` bzw. `0f39ab0`); der Schluss, dass die fixierte Eigenschaft „die richtige“ ist, ist **B** (Spec-gegen-Welt).

5 Reflexive Selbstanwendung

Eine Methode, die Korrektheit gegen eine externalisierte, getypte Spezifikation erzwingt, muss sich der Frage stellen, ob sie auf sich selbst anwendbar ist. Wir prüfen das nicht rhetorisch, sondern operativ: Die Methode `CDD` ist in ihrem eigenen `.spot/`-Graphen modelliert, und ihr Gate läuft über sich selbst.

5.1 Selbst-Gate

Das Selbstmodell von `CDD` ist im Repository `cong-driven-development` @ `v0.7.0` als typisierter F#-DU-Graph abgelegt: ein JSON-Knoten je Element unter `.spot/`, mit den Konvergenzzuständen `Aligned`, `Pending`, `Diverged` und `Orphaned`. Das Orakel `SetzeSpecAligned` markiert einen Test-Knoten als `Aligned`, sobald ein Test-Marker ihn im Quellcode abdeckt; die grüne CI-Suite (rot blockt den Merge) und der reflexive Selbst-Test binden dieses `Aligned` an einen bestandenen Test, sodass `Aligned` den Test-`PASS` impliziert, nicht den Exit-Code 0 des Prozesses. Diese Unterscheidung ist nicht kosmetisch: Ein Prozess kann mit Exit-0 terminieren, ohne dass eine einzige Property tatsächlich geprüft wurde.

Der reflexive Selbst-Gate `spec-gate-selbst-hart` (Repository `@main`) wendet exakt dieses Orakel auf das Selbstmodell an. Der autoritative Testlauf (Runner, nicht `grep`) meldet 42/42 Tests grün; das Selbstmodell umfasst 66 Knoten, davon 62 `Aligned` und 4 `Pending`. Wir behaupten hier keine Vollständigkeit: Die vier `Pending`-Knoten sind genau die Stellen, an denen die Methode ihre eigene Spezifikation noch nicht eingeholt hat. Das ist der ehrliche, nicht der geschönte Konvergenzzustand.

5.2 Ehrliche Konvergenz

Konvergenz im Sinne von `CDD` ist kein Fixpunkt-Versprechen, sondern ein beobachtbarer Zustand pro Knoten. Ein Graph mit 62 `Aligned` und 4 `Pending` ist nicht „fertig“, sondern *vermessen*: Die Methode macht ihre eigene Lücke

sichtbar, statt sie zu verbergen. Wir lesen das als Funktionsmerkmal, nicht als Mangel — ein Terminierungs-Orakel, das nur `Aligned` kennt, wäre kein Orakel, sondern eine Zusicherung ohne Diskriminationskraft.

5.3 Die reflexive Grenze

Wendet man das Gate auf das Gate an, stellt sich die Frage nach der *Grund-Invariante*: Welche Invariante rechtfertigt das Orakel, das alle übrigen Invarianten prüft? Hier sind zwei Aussagen strikt zu trennen.

(i) Strukturell — der Regress. Die Grund-Invariante ist im System nicht selbst-fundierbar. Jede Begründung der prüfenden Invariante I_0 müsste eine weitere Invariante I_1 heranziehen, die I_0 als korrekt ausweist; deren Begründung wiederum I_2, und so fort. Der Regress terminiert nicht durch ein internes Argument, sondern nur durch einen Anker *außerhalb* des Prüfvorgangs. Das ist eine strukturelle, architektonische Aussage über das Prüfen — ein Argument, kein im Papier geführter formaler Beweis, und kein Werturteil.

(ii) Normativ — die Wahl des Ankers. Dass im hier beschriebenen Aufbau der *Mensch* diesen Anker setzt, ist eine Wahl, keine Unmöglichkeitsaussage über Maschinen. Sie ruht auf zwei Gründen, die beide normativ und nicht berechnungstheoretisch sind: Intent ist extern zum Generator, und das Setzen des Ankers ist verantwortungsbehaftet. Ein zweiter, hinreichend unabhängiger Agent könnte die Grund-Invariante ebenso setzen; der Mensch ist dann *verschoben*, nicht durch ein Theorem als notwendig erwiesen. Wir behaupten ausdrücklich nicht, der Anker sei „unmechanisierbar“ oder „nur ein Mensch“ könne ihn setzen — das wäre der Penrose/Lucas-Fehlschluss, den wir ablehnen (vgl. Abschnitt 5.4).

5.4 Theorie-Disclaimer

Drei metamathematische Resultate werden in dieser Arbeit in unterschiedlichen Rollen genutzt, und die Rollen dürfen nicht vermischt werden. Der Satz von Rice [7] ist ein *Beweis*: Für jede nicht-triviale semantische Eigenschaft partieller rekursiver Funktionen ist die zugehörige Entscheidungsfrage unentscheidbar; die Voraussetzung (nicht-triviale semantische Eigenschaft) ist im hier betrachteten Fall erfüllt. Rice liefert damit ausschließlich die unentscheidbare obere Schranke und motiviert die entscheidbare *pro-Spec-Approximation* — die Verschiebung von „korrekt schlechthin“ auf „erfüllt diese eine getypte Spec“. Gödels Unvollständigkeitssätze und Tarskis Undefinierbarkeit dienen demgegenüber nur als *Motivation* und Analogie. Ihre Voraussetzungen — rekursive Axiomatisierung, Subsumtion der Robinson-Arithmetik Q, Konsistenz, Syntax-Kodierung — beanspruchen wir für den

`.spot/`-Graphen *nicht*, und sie gelten dort nicht. Aus keinem dieser drei Resultate folgt, dass ein Mensch unersetzlich ist; der Regress (i) ist architektonisch, die Anker-Setzung (ii) ist normativ.

6 Grenzen und Geltungsbereich

Wir halten die Grenzen dieser Arbeit explizit, weil ein Terminierungs-Orakel, das seine eigene Reichweite überschätzt, denselben Fehler begeht, den es verhindern soll.

6.1 Empirische Basis

Die Evidenz beruht auf *zwei* kleinen Fallstudien. `runenruf` @ `56376a5` (46/46 Tests, Runner) ist mit $\sim$ 125 LoC die ehrlich freundlichste Domäne, weil die Spezifikation dort mit der Welt zusammenfällt — die Simulation *ist* ihre eigene Realität. `ledger-casestudy` @ `0f39ab0` (6/6 Tests, Runner; F#/.NET 9, int64-Cent, $\sim$ 41 LoC Domänenkern) ist das Gegenstück mit echtem IO (Journal und Replay) und veränderlichem Requirement. Zwei Fallstudien zeigen *Anwendbarkeit*; sie beweisen keine Skalierung über Domänen ($N = 2$). Es gibt keinen Issue-Tracker im großen Stil und keine unbeschränkte Autonomie des Loops.

6.2 Reichweite der Verifikation

Im Repository `ledger-casestudy` @ `0f39ab0` sind *zwei* Invarianten maschinell bewiesen: `proofs/Werterhaltung.lean` zeigt Werterhaltung und Nichtnegativität für jede Buchungsfolge. Das Modell enthält den Deckungs- und Ablehnungspfad und ist modellgleich zur F#-Funktion `buche`; der Beweis ist `sorry`-frei (`#print axioms` weist nur `propext` und `Quot.sound` aus), kommt ohne Mathlib aus (nur `Core` und `omega`), und der CI-Job `lean-proof` ist grün (`@0f39ab0`). Die übrigen Eigenschaften — Isolation und Replay-Treue — sind *nur* per FsCheck geprüft, etwa `spec-siegel-lager-nichtnegativ` in `runenruf` @ `56376a5` über jeden Seed und jede Folge, deterministisch. Damit ist die Drei-Schichten-Kette (Typen $\rightarrow$ Property $\rightarrow$ Beweis) derzeit für Werterhaltung und Nichtnegativität geschlossen. Wir behaupten keine durchgehende Beweisabdeckung.

6.3 Common-Mode-Fehler

Die mechanisch erzwungene Trennung Generator $\perp$ Prüf-Orakel (Werkzeug-Allowlist ohne Schreibrecht des Generators, fälschungssicher; Code `@main`) schützt nicht gegen jede Fehlerklasse. Wenn dieselbe Modell-Instanz sowohl die Spec als auch den Code erzeugt, korrelieren die Fehler (*common-mode*) — genau der von VeriAct [4] beschriebene Fall, dass

verifizierer-akzeptiert nicht korrekt heißt. Der gefangene Mis-Spec in `ledger-casestudy` @ `0f39ab0` („Falsifiable, after 4 tests“; reproduzierbares Negativ-Artefakt `demo-gate-at-failure.sh`) wurde nicht durch den Loop aus sich selbst entdeckt, sondern weil ein Mensch eine stärkere Property und ein geändertes Requirement als Gegen-Orakel einbrachte. Der Mensch ist hier der Dekorrelator; die Trennung Generator $\perp$ Spec-Autor gilt nur, wenn die Spec aus separater Quelle stammt.

6.4 Was in der Spalte B bleibt

In Anlehnung an die A/B-Verifizierbarkeitstabelle (Abschnitt 6.5) markieren wir die folgenden Behauptungen ehrlich als **B**, also als menschliche Setzung ohne maschinelle Außenprüfung:

- die Bindung der Spec an die Welt (Spec $\models$ Welt) — nach VeriAct [4] nicht orakel-prüfbar;
- das Setzen der Grund-Invariante (normativ, nicht berechnungstheoretisch; Abschnitt 5.3);
- Neuheit und Abgrenzung — eine Literatur-Aussage, kein Beweis (Abschnitt 7);
- die Skalierung der Methode über Domänen — $N = 2$ zeigt Anwendbarkeit, nicht Skalierung; der lokale Lean-Allquantor über Buchungsfolgen ist **A**, der Sprung auf „für jedes System“ ist **B**.

6.5 A/B-Verifizierbarkeitstabelle

Jede tragende Behauptung dieser Arbeit deklariert sich als **A** (maschinell und extern verifiziert: Commit-Hash, arXiv-ID, CI-Run, `sorry`-Audit) oder als **B** (menschliche Setzung). Tabelle 1 listet die tragenden Behauptungen mit ihrer Marke und ihrem Anker auf. Keine tragende Assertion bleibt unmarkiert; das ist der zentrale Hebel von Logik gegenüber bloßer LLM-Plausibilität.

7 Verwandte Arbeiten

7.1 Intent als offene Herausforderung

Lahiri [1] formuliert die Formalisierung von Intent als benannte, *offene* Grand Challenge zuverlässigen Programmierens im Zeitalter von KI-Agenten. Wir beanspruchen keine Lösung dieser Challenge, sondern eine partielle, getypte Externalisierung des Intents pro Spec — und markieren die verbleibende Spec-gegen-Welt-Bindung als **B** (Abschnitt 6.4). Storey [2]

Tabelle 1: A/B-Verifizierbarkeit jeder tragenden Behauptung. A = maschinell und extern verifiziert (Commit-Hash, arXiv-ID, CI-Run, `sorry`-Audit); B = menschliche Setzung.

Behauptung	**A/B**	**Anker**
Runenruf: 46/46 Tests grün	**A**	`runenruf` @ `56376a5` (Runner)
Ledger: 6/6 Tests grün	**A**	`ledger-casestudy` @ `0f39ab0` (Runner)
CDD-Selbst-Gate: 42/42 Tests, 62/66 `Aligned`	**A**	`cong-driven-development` @ `v0.7.0` (Runner)
Werterhaltung + Nichtnegativität `sorry`-frei	**A**	`#print axioms` (nur `propext`, `Quot.sound`) @ `0f39ab0`
Lean ohne Mathlib, CI `lean-proof` grün	**A**	CI-Job `lean-proof` @ `0f39ab0`
Allowlist-Trennung Generator ⊥ Prüf-Orakel	**A**	Code @ `v0.7.0`
Negativ-Artefakt `demo-gate-at-failure.sh`	**A**	Skript @ `0f39ab0`
FsCheck `spec-siegel-lager-nichtnegativ` (jeder Seed/Folge)	**A**	`runenruf` @ `56376a5` (Runner)
Kompressionsfaktoren ($\sim$ 40–125×) als LoC-Zahlen	**A**	Code @ `56376a5`/`0f39ab0`
Zitate (vollständige Bib-Daten)	**A**	arXiv-ID / Journalband
Spec trifft die Welt (Spec $\models$ Welt)	**B**	per VeriAct nicht orakelprüfbar
Wer die Grund-Invariante setzt (Mensch)	**B**	normative Wahl, Abschnitt 5.3
Stärkere Ledger-Property als Gegen-Orakel	**B**	menschliche Setzung, Abschnitt 4.2
Neuheit/Abgrenzung der Kombination	**B**	Literatur-Aussage, Abschnitt 7.6
Methoden-Skalierung über Domänen	**B**	$N = 2$; Sprung auf „jedes System“
„Fixierte Eigenschaft ist die richtige“	**B**	Spec-gegen-Welt, Bemerkung 4.3

prägt den komplementären Begriff der *Intent Debt*, „the absence of externalized rationale that developers and AI agents need to work safely with code“; der externalisierte `.spot/`-Graph ist genau ein Versuch, diese Schuld nicht aufzubauen.

7.2 LLM im Modulo-Rahmen

Kambhampati und andere [3] argumentieren, dass LLMs nicht selbst planen, wohl aber im *LLM-Modulo*-Rahmen mit externen Verifikatoren nützlich sind. Unsere Architektur ist eine Instanz dieses Musters: Der Generator schlägt vor, ein externer, mechanisch getrennter Prüfer entscheidet (Abschnitt 6.3).

7.3 Nächste Nachbarschaft

VeriAct [4] ist die nächste Nachbarschaft: Die Arbeit zeigt, dass verifiziererakzeptierte Spezifikationen nicht korrekt sein müssen, und benennt damit präzise die Grenze, an der unser menschlicher Dekorrelator ansetzt. He und andere [5] verfolgen mit *Property-Oriented and Structurally Minimal Feedback* ein verwandtes Ziel auf der Feedback-Seite: minimale, eigenschaftsorientierte Rückmeldung zur Code-Verfeinerung. Beide Linien teilen unsere Prämisse, dass das Orakel die kritische Stelle ist, nicht der Loop.

7.4 Trajektorie und Werkzeuge

METR [6] misst den „50%-task-completion time horizon“: Zu Beginn 2025 liegt er bei etwa 50 bis 110 Minuten, mit einer Verdopplung etwa alle 7 Monate. Wir referenzieren ausschließlich diese Zahlen; sie motivieren, warum das Terminierungs-Orakel mit wachsender Aufgabenlänge wichtiger wird, nicht weniger. Werkzeuge wie Kiro und Spec-Kit verfolgen eine spezifikationsgetriebene Arbeitsweise; sie verankern die Spec jedoch nicht in einem typisierten Graphen mit architektonisch getrenntem Generator/Orakel und property- bzw. beweisverifiziertem Gate.

7.5 Das breitere Feld

Spezifikationsgetriebene Entwicklung ist 2026 kein Randphänomen, sondern die dominante Antwort auf „Vibe-Coding“: neben Kiro und Spec-Kit etwa Cursor, OpenSpec, BMAD, Google Antigravity sowie die kommerzielle Plattform Tessl [10], die die Spec als primäres Artefakt führt und mit Tests als Leitplanken paart — ihr Problem-Framing (Agenten, die „fertig“ melden, ohne es zu sein) deckt sich mit unserer Diagnose. Parallel hat sich ein eigenes Subfeld, *vericoding*, etabliert: aus formalen Spezifikationen verifizierten Code synthetisieren und mit Lean, Dafny oder Verus prüfen — etwa CLEVER [11], ein Benchmark für end-to-end in Lean verifizierte

Codegenerierung, oder VeriGuard [12], das Agenten-Sicherheit über verifizierte Codegenerierung mit getrenntem Generator und Prüfer absichert. Unser Beitrag liegt damit weder im Gedanken „Spec als Wahrheit“ (heute Commodity) noch in „Code gegen formale Spec verifizieren“ (vericoding), sondern in deren *Integration* zu einer laufenden, getypten Entwicklungsmethode: dem Konvergenz-Zustandsgraphen (`SPOT`), der architektonischen Generator/Orakel-Trennung per Allowlist und der reflexiven Selbstanwendung. Das ist eine Architektur- und Integrationsaussage, kein Ideen-Primat.

7.6 Neuheit, konditional

Wir erheben keinen Prioritätsanspruch. Uns ist keine publizierte Implementierung mit *genau dieser Kombination* bekannt: getypte, externalisierte Spec; architektonisch getrennter Generator und Prüf-Orakel; Werkzeug-Allowlist ohne Schreibrecht des Generators; sowie ein property- und beweisverifiziertes Gate. Diese Aussage ist eine Literatur-Aussage und damit **B** (Abschnitt 6.4) — kein Beweis.

8 Fazit

Das eigentliche Problem agentischer Entwicklung ist nicht der Loop — den zu schreiben ist, wie Cherny [8] es formuliert, die Routinearbeit — sondern sein Terminierungs-Orakel. Unsere Antwort ist Konvergenz gegen eine externalisierte, getypte Spec, geprüft durch ein vom Generator architektonisch getrenntes Orakel. Wir belegen das mit zwei kleinen Fallstudien (`runenruf` @ `56376a5`, 46/46; `ledger-casestudy` @ `0f39ab0`, 6/6), einem reflexiven Selbst-Gate (`cong-driven-development` @ `v0.7.0`, 42/42, 62/66 `Aligned`) und genau zwei maschinell bewiesenen Invarianten (`Werterhaltung.lean`: `werterhaltung` + `nichtnegativ`, `sorry`-frei).

Wir schließen mit einer Lücke, nicht mit einem Versprechen. Die Spec-gegen-Welt-Bindung bleibt **B**; die Grund-Invariante terminiert den Regress nur durch einen externen Anker, dessen Setzung eine Wahl ist und keine berechnungstheoretische Notwendigkeit; die Beweiskette ist erst für zwei Eigenschaften (Werterhaltung, Nichtnegativität) geschlossen; $N = 2$ zeigt Anwendbarkeit, nicht Skalierung. Aus keinem Theorem folgt, dass ein Mensch hier unersetzlich wäre — ein zweiter, hinreichend unabhängiger Agent könnte den Dekorrelator stellen. Was wir zeigen, ist bescheidener und überprüfbarer: dass das Orakel die richtige Stelle ist, um anzusetzen, und dass eine Schraube weiter gedreht ist — mit Lücke, ehrlich markiert.

A Reproduzierbarkeit

Alle in Tabelle 1 als **A** markierten Behauptungen sind an genau drei Repositories zu festen Commits gebunden. Tabelle 2 fasst die Anker zusammen; die nachfolgenden Befehle reproduzieren die autoritativen Runner-Counts, den `sorry`-freien Lean-Lauf und das Negativ-Artefakt.

Tabelle 2: Verifizierte Artefakte und ihre Anker (Repository @ Commit).

Repository	**Commit**	**Rolle**
`Koschnag/cong-driven-development`	`v0.7.0`	Methode/IDE, reflexives Selbst-Gate
`Koschnag/runenruf`	`56376a5`	Existenzbeweis (Simulationsdomäne)
`Koschnag/ledger-casestudy`	`0f39ab0`	Gegenstück (echtes IO, Lean-Beweis)

A.1 Methode und Selbst-Gate

```
git clone https://github.com/Koschnag/cong-driven-development.git
cd cong-driven-development
git checkout main
dotnet test tests/Cdd.Tests # autoritativer Runner-Count: 42/42
```

Listing 2: CDD-Selbst-Gate: 42/42 Tests, 62/66 `Aligned`.

A.2 Fallstudie A: Runenruf

```
git clone https://github.com/Koschnag/runenruf.git
cd runenruf
git checkout 56376a5
dotnet test tests/Runenruf.Tests # autoritativer Runner-Count: 46/46
```

Listing 3: Runenruf: 46/46 Tests, FsCheck `spec-siegel-lager-nichtnegativ`.

A.3 Fallstudie B: Ledger (Tests, Lean, Negativ-Artefakt)

```
git clone https://github.com/Koschnag/ledger-casestudy.git
cd ledger-casestudy
git checkout 0f39ab0

# 1) F#-Tests (autoritativer Runner-Count: 6/6)
dotnet test tests/Ledger.Tests

# 2) Lean-Beweis (sorry-frei; #print axioms: nur propext, Quot.sound
    )
lean proofs/Werterhaltung.lean
```

```
10
11 # 3) reproduzierbares NEGATIV-Artefakt: Gate im
       Falsifikationszustand
12 # ("Falsifiable, after 4 tests")
13 ./demo-gate-at-failure.sh
```

Listing 4: Ledger: 6/6 Tests, `sorry`-freier Lean-Beweis, reproduzierbares Negativ-Artefakt.

Der Befehl `lean proofs/Werterhaltung.lean` aus Listing 4 schließt den Beweis ohne Mathlib (nur `Core` + `omega`) ab; `#print axioms werterhaltung` weist ausschließlich `propext` und `Quot.sound` aus. Das Skript `demo-gate-at-failure.sh` ist kein Positiv-, sondern ein *Negativ*-Artefakt: Es führt das Gate reproduzierbar in den Zustand „Falsifiable, after 4 tests“ und macht damit den in Abschnitt 4.2 beschriebenen, vom Menschen gefangenen Mis-Spec wiederholbar sichtbar.

Literatur

[1] Lahiri, Shuvendu K.: Intent Formalization: A Grand Challenge for Reliable Coding in the Age of AI Agents. `arXiv:2603.17150`, 2026.

[2] Storey, Margaret-Anne: From Technical Debt to Cognitive and Intent Debt: Rethinking Software Health in the Age of AI. `arXiv:2603.22106`, 2026.

[3] Kambhampati, Subbarao and others: LLMs Can't Plan, But Can Help Planning in LLM-Modulo Frameworks. `arXiv:2402.01817`, 2024.

[4] Misu, Md Rakib Hossain; Ma, Iris; Lopes, Cristina V.: VeriAct: Beyond Verifiability — Agentic Synthesis of Correct and Complete Formal Specifications. `arXiv:2604.00280`, 2026.

[5] He, Lehan; Chen, Zeren; Zhang, Zhe; Gao, Xiang; Sheng, Lu: Effective LLM Code Refinement via Property-Oriented and Structurally Minimal Feedback. `arXiv:2506.18315`, 2025.

[6] METR: Measuring AI Ability to Complete Long Software Tasks. `arXiv:2503.14499`, 2025.

[7] Rice, Henry Gordon: Classes of Recursively Enumerable Sets and Their Decision Problems. Transactions of the American Mathematical Society 74(2):358–366, 1953.

[8] Cherny, Boris: zitiert in „Loop Engineering“, The New Stack, 2026 — „my job is to write loops“.

[9] Nguyen, Cong Chanh Vinzenz: cong-driven-development: A Spec-Convergent Method and IDE for Agentic Software Development. Software-Repository, `github.com/Koschnag/cong-driven-development` @ `v0.7.0`, 2026.

[10] Tessl: Spec-Driven Development with Tessl (Framework und Spec-Registry). Agent-Enablement-Plattform, `https://tessl.io`, 2026.

[11] Thakur, Amitayush u. a.: CLEVER: A Curated Benchmark for Formally Verified Code Generation. arXiv:2505.13938, 2025.

[12] Miculicich, Lesly u. a.: VeriGuard: Enhancing LLM Agent Safety via Verified Code Generation. arXiv:2510.05156, 2025.