

TECHNISCHES WHITEPAPER / VERSION 1.1 / 26. AUGUST 2026

Level 3.5 Autonomie für kontrollierte agentische Softwareentwicklung

Warum nicht das Modell allein, sondern das Zusammenspiel aus Harness, Prüfung und Freigabe über den praktischen Einsatz entscheidet.

LEITTHESE

Nicht das Modell ist die Fabrik.

Praktische Fähigkeit = Modell × Harness × Prüfung × Freigabe.

CHRISTIAN BINDER

Senior Software Engineer und Applied AI Engineer
Für regulierte und IP-sensible Entwicklungsumgebungen



KURZFASSUNG

Die kontrollierte Fabrik beginnt dort, wo ein Chatfenster nicht mehr reicht.

HEPHAUSTOS richtet sich an Teams, die KI-gestützte Entwicklung einsetzen wollen, deren Quellcode aber nicht beliebig in Cloud-Werkzeuge übertragen werden darf. Dazu kommen gewachsene Systeme, uneinheitliche Dokumentation und Freigaberegeln, die sich nicht durch einen besseren Prompt ersetzen lassen.

Level 3.5 Autonomie bezeichnet einen klar abgegrenzten Betriebsmodus: Der Agent führt einen zusammenhängenden Arbeitslauf selbstständig aus, die Prüfung bleibt vom Arbeitsbereich getrennt, und ein Mensch entscheidet über die Übernahme. Wie beim teilautonomen Fahren arbeitet das System innerhalb definierter Grenzen, ohne die letzte Verantwortung zu übernehmen.

LEITTHESE

Nicht das Modell ist die Fabrik. Praktische Fähigkeit = Modell × Harness × Prüfung × Freigabe.

Modellunabhängig Lokale, selbst gehostete und externe Endpunkte werden über ein zentrales Gateway angebunden.	Nachvollziehbar Tasks, Reparaturversuche sowie Build- und Ausführungsstatus werden strukturiert protokolliert.
Kontrolliert Automatische Eskalation bleibt innerhalb vorab freigegebener Modellstufen und Vertrauensgrenzen.	Menschlich freigegeben Kein automatischer Commit in ein Kundenrepository und keine Übernahme ohne bewusstes Gate.

INHALT

01 / Das Problem	02-03 / These & Forschungsstand
04 / Architektur	05 / PoC 01
06 / Ausblick	07 / Über und Kontakt

01 / DAS PROBLEM

Coding Agents scheitern häufig nicht am Code, sondern am Einsatzkontext.

Ein Modell kann syntaktisch korrekten Code erzeugen und trotzdem an der Realität eines Unternehmens vorbeiarbeiten. Informationen liegen in Tickets, Schnittstellenbeschreibungen, Buildskripten, Datenbanken und den Köpfen weniger erfahrener Entwickler.

Gleichzeitig begrenzen Datenschutz, Geheimhaltung und regulatorische Vorgaben, welche Daten welche Umgebung verlassen dürfen. Ein belastbarer Agentenprozess muss deshalb nicht nur Code erzeugen, sondern Kontext, Berechtigungen, Prüfung und Verantwortung organisieren.

01 / Verteilter Kontext Gewachsene Systeme haben keinen einzelnen, vollständigen Projektkontext. Auswahl und Herkunft müssen reproduzierbar sein.	02 / Vertrauensgrenzen Quellcode, Konfiguration und Systemwissen dürfen oft nicht an frei gewählte Cloud-Endpunkte gesendet werden.	03 / Prüfung und Verantwortung Ein Agent darf nicht zugleich Lösung und unveränderliche Erfolgskriterien kontrollieren.
---	---	---

DIE AUTONOMIESTUFEN ALS ARBEITSMODELL

Stufe	Arbeitsweise	Verantwortung im Loop
1	Autocomplete für einzelne Fragmente	Durchgehend beim Entwickler
2	Chat und Copy-Paste ohne echten Projektzustand	Übertragung und Prüfung bleiben Handarbeit
3	Zusammenhängender autonomer Arbeitslauf	Getrennte Prüfebene fehlt
3.5	Autonome Ausführung mit geschützter Prüfung	Menschliches Gate vor Übernahme
4 bis 5	Vollautonome produktive Übernahme	Niemand; nicht Zielbereich von HEPHAUSTOS.

DEFINITION	Level 3.5 ist kein allgemeiner Industriestandard und keine Versionsnummer. Es ist das HEPHAUSTOS-Arbeitsmodell für autonome Ausführung, getrennte Prüfung und bewusste menschliche Freigabe.
-------------------	--

Nicht das Modell ist die Fabrik.

Ein Modell liefert Sprach- und Codefähigkeit. Erst der Harness macht daraus einen belastbaren Arbeitsprozess. Er entscheidet, welchen Kontext das Modell sieht, welche Werkzeuge es verwenden darf, wie Fehler zurückgeführt werden und wann ein Lauf endet.

$$\text{PRAKTISCHE FÄHIGKEIT} = \text{MODELL} \times \text{HARNESS} \times \text{PRÜFUNG} \times \text{FREIGABE}$$

Dabei gilt: Harness = Kontext, Werkzeuge und Feedbackloop.

Austauschbare Modellendpunkte schaffen vergleichbare Rahmenbedingungen, aber keine identischen Fähigkeiten. Modelle unterscheiden sich in Codeverständnis, Werkzeugnutzung, Kontexttreue, Kosten und Betriebsanforderungen.

AGENT LIGHTNING: DER HARNESS BESITZT DIE INTERAKTIONSSCHLEIFE

Agent Lightning v1.0 bezeichnet den Ansatz als harnessed agentic RL. Der im Betrieb verwendete Harness besitzt die Interaktion mit Werkzeugen und Umgebung; das Trainingssystem beobachtet Modellanfragen über eine getrennte Schnittstelle. Im veröffentlichten Coding-Experiment verbessert RL Qwen3.5-9B auf SWE-bench Verified von 41,8 % auf 56,4 % (+14,6 Prozentpunkte) bei rund 6.000 Trainingsbeispielen. [3]

EINORDNUNG

Diese Werte gehören zum Agent-Lightning-Experiment. Sie sind weder ein HEPHAUSTOS-Ergebnis noch ein Produktivitätsversprechen und lassen sich nicht auf beliebige Aufgaben übertragen.

SELBSTVERBESSERUNG UND TRAJEKTORIEN-AUDITS

Agentic Harness Engineering untersucht die iterative Weiterentwicklung von Werkzeugen, Middleware und Langzeitgedächtnis bei unverändertem Basismodell. [4] HarnessAudit verschiebt die Sicherheitsbewertung vom Endergebnis auf die gesamte Trajektorie. Die Arbeit umfasst 210 Aufgaben in acht Domänen und zeigt, dass ein korrektes Ergebnis keine regelkonforme Ausführung garantiert. [5]

ABGRENZUNG HEPHAUSTOS

HEPHAUSTOS baut kein RL-Trainingsframework nach. Das Projekt betreibt angewandtes Harness Engineering für gewachsene Codebasen, lokale oder streng kontrollierte Betriebsmodelle, verdeckte Prüfschranken und menschliche Freigabe.

Fünf Prinzipien für eine kontrollierte Software-Fabrik.

Die Prinzipien trennen Modellfähigkeit, Wissensbereitstellung, Ausführung, Prüfung und Governance. So lassen sich Komponenten austauschen, ohne die Regeln des Gesamtsystems aufzugeben.

01 / LLM-AGNOSTIK

Warum: Datenschutz, Verfügbarkeit, Modellqualität und Kosten unterscheiden sich je Projekt. Eine Architektur, die nur mit einem Anbieter funktioniert, bindet die Vertrauensgrenze an dessen Produktpolitik.

Umsetzung: LiteLLM bildet das zentrale Modell-Gateway. Ollama bindet selbst betriebene Open-Weight-Modelle an; externe Endpunkte können separat freigegeben werden. Routing und Modellstufe bleiben Konfiguration.

Grenze: Ein austauschbarer Endpunkt ist nicht automatisch gleich leistungsfähig. Jeder Modellwechsel braucht Prüfung unter denselben Factory-Bedingungen.

02 / HARNESS ENGINEERING

Warum: Das Modell kennt Repository-Zustand, Buildkonventionen und Freigaberegeln nicht von selbst. Unstrukturierter Vollkontext erzeugt Kosten und lenkt vom relevanten Problem ab.

Umsetzung: HEPHAUSTOS wählt Projektkontext, startet die Codebearbeitung über Aider, führt definierte Build- und Ausführungsschritte aus und gibt Fehlermeldungen in eine begrenzte Reparaturschleife zurück. Strukturierte Events dokumentieren den Ablauf.

03 / FABRIK STATT AGENT

Warum: Eine Chat-Sitzung ist flüchtig. Wiederholbare Entwicklung braucht versionierbare Quellen, definierte Fertigungsaufträge und einen nachvollziehbaren Projektzustand.

Umsetzung: Der PoC erzeugt aus strukturierten Quellen ein LLM-Wiki aus Markdown-Dateien mit YAML-Frontmatter. Diese Grundform entspricht dem von Google Cloud veröffentlichten Open Knowledge Format (OKF). OKF v0.2 macht Provenienz, Vertrauen und Lebenszyklus zu ausdrücklich modellierten Metadatenfamilien. [1][2]

EHRLICHE GRENZE

HEPHAUSTOS behandelt vollständige OKF-Kompatibilität als überprüfbaren Vertrag. Die gemeinsame Grundform allein wird nicht als pauschale Konformitätsbehauptung verwendet.

Prüfschranke, Eskalation und menschliches Gate.

04 / HOLDOUT-PRÜFUNG

Warum: Wenn ein Agent die unveränderlichen Prüfkriterien bearbeiten kann, kann er einen Test passend machen, statt das Problem zu lösen. Erfolgreiche Selbstprüfung ist dann kein unabhängiger Nachweis.

Umsetzung: Prüfmaterial und Freigaberegeln liegen außerhalb des normalen Arbeitsbereichs. Der Harness führt die Prüfung aus und meldet nur das für die Reparatur notwendige Feedback zurück. Die genaue Schutzmechanik bleibt bewusst außerhalb dieses Whitepapers.

Grenze: Ein Holdout beweist nur die darin formulierten Eigenschaften. Relevanz, Trennung und Pflege der Kriterien müssen im Projekt evaluiert werden.

05 / ESKALATIONSSTUFEN

Warum: Lokal betriebene Modelle können aus Datenschutzgründen bevorzugt sein, aber nicht jede Aufgabe zuverlässig lösen. Souveränität bedeutet Kontrolle darüber, wann welche Daten welche Grenze passieren.

Umsetzung: Löst ein Modell eine Aufgabe nicht autark, kann die Fabrik automatisch auf stärkere, vorab freigegebene Modellstufen eskalieren. Jede Eskalation bleibt innerhalb der festgelegten Vertrauensgrenze, wird protokolliert und erhält nur den erlaubten Kontext.

MENSCHLICHES GATE	Die Freigabe der Modellstufen erfolgt vor dem Lauf. Das menschliche Gate vor der Übernahme bleibt davon unberührt.
-------------------	--

VERTRAUENSMODELL

A / VOLLSTÄNDIG LOKAL	B / FREIGELEGENER ENDPUNKT	MENSCH / FREIGABE
Quellcode, Wissen, Modell und Logs bleiben in der Kundenumgebung.	Nur definierter Kontext darf eine kontrollierte externe Modellstufe erreichen.	Ergebnis und Nachweis werden geprüft, bevor eine Übernahme möglich ist.

VEREINFACHTER ABLAUF

01 / Quellen und OKF-Wissen	02 / Task und Harness	03 / Modell und Build	04 / Getrennte Prüfung	05 / Menschliche Freigabe
-----------------------------	-----------------------	-----------------------	------------------------	---------------------------

Strukturierte Quellen werden in reproduzierbaren Projektkontext überführt. Auftrag, Werkzeuge, Dateibereiche, Stopbedingungen und Feedbackloop sind explizit. Das freigegebene Modell bearbeitet Code, während der Harness baut und ausführt. Holdouts bleiben außerhalb des Arbeitsbereichs; ein Mensch entscheidet bewusst über die Übernahme.

Vom API-Vertrag zum laufenden C#-Client.

Der Proof of Concept untersuchte nicht, wie schnell ein Frontier-Modell C# schreibt. Die engere Frage lautete: Kann ein austauschbares, selbst gehostetes Open-Weight-Modell innerhalb eines kontrollierten Harness einen fachlichen Task übernehmen, Fehler auswerten, seine Lösung reparieren und Wissen zurückschreiben?

FORSCHUNGSFRAGE

Wie weit kommt ein Open-Weight-Modell, wenn Kontext, Werkzeuge, Prüfschritte und Feedbackloop als eigenständige Fabrik organisiert werden?

AUSANGSMATERIAL UND LAUFZEIT

Der Lauf erhielt eine Postman-Collection, Projektkonfiguration, getrennt bereitgestellte Zugangsdaten und einen fachlichen Markdown-Task. HEPHAUSTOS wählte relevante Wissensseiten, ließ einen C#-Client bearbeiten und führte Build sowie Zielsystemprüfung aus.

Komponente	Konfiguration
Modell	Qwen3-Coder 30B-A3B Q8_0
Serving	Ollama über LiteLLM
GPU	1 x NVIDIA A40, nominell 48 GB
Kontext	32.768 Tokens
Belegung	rund 35 GB VRAM
Infrastruktur	RunPod Cloud

TRANSPARENZ

Das Modell wurde selbst gehostet, die GPU lief jedoch auf RunPod und damit auf Cloud-Infrastruktur. Dieser PoC war nicht air-gapped und wird nicht als vollständig lokaler Lauf dargestellt.

PIPELINE

01 / Collection einlesen	02 / Kontext auswählen	03 / Task ausführen
04 / Bauen und prüfen	05 / Autonom reparieren	06 / Wissen zurückschreiben

Der deterministische Importer erzeugt Wissensseiten mit Quellenbezug. Der Harness wählt relevante Einheiten, lässt Aider den C#-Client bearbeiten und liefert maschinenlesbares Build- und Ausführungsfeedback zurück. Begrenztes Feedback startet den nächsten Versuch; nach erfolgreicher Ausführung synchronisiert die Factory das Wiki.

Vier Reparaturiterationen, bereinigt aus strukturierten Events.

Aus mehreren PoC-Läufen wurde für die veröffentlichte Aussage der erfolgreiche Lauf vom 26. Juli 2026 mit vier protokollierten Aider-Versuchen ausgewertet. Der Rohlog enthält reale Dokumentnamen und personenbezogene Dateibezeichnungen und wird nicht veröffentlicht. Der Auszug verwendet nur Ereignistyp, Status und Zeitfolge.

Zeit	Schritt	Status	Bereinigter Befund
21:22:52	Versuch 1	Fehlgeschlagen	Compilerfehler werden an den Reparaturloop zurückgegeben.
21:23:15	Versuch 2	Fehlgeschlagen	Build erfolgreich; Anmeldung antwortet mit HTTP 401.
21:23:44	Versuch 3	Fehlgeschlagen	Build erfolgreich; Authentifizierung weiterhin nicht akzeptiert.
21:24:25	Versuch 4	Erfolgreich	Build und Zielsystemprüfung bestehen; Reparaturloop endet.
21:25:30	Abschluss	Erfolgreich	Wiki synchronisiert und Task strukturiert abgeschlossen.

Run-ID, Pfade, Zielsystem und Nachrichteninhalte wurden entfernt. Zeitangaben in CEST. Der Holdout-Status blieb in allen vier Versuchen unverändert.

C#-Client autonom bearbeitet Kontext und Codebearbeitung liefen innerhalb des protokollierten Runs ohne manuellen Eingriff.	Buildfehler behoben Der erste Compilerfehler wurde zurückgeführt und im folgenden Versuch korrigiert.
Authentifizierungsfehler behoben Zwei fehlgeschlagene Prüfungen dienten als Feedback für weitere Reparaturen.	Ausführung und Wiki-Update Der vierte Versuch bestand; anschließend wurde das Wissen synchronisiert.

WAS DER POC BELEGT	Der Ablauf von Kontextauswahl über Codebearbeitung und vier Reparaturiterationen bis zu erfolgreicher Ausführung und Wiki-Update funktioniert unter den beschriebenen Bedingungen.
WAS DER POC NICHT BELEGT	Dies ist ein technischer Proof of Concept, kein allgemeiner Produktivitätsbenchmark. Er belegt weder einen prozentualen Produktivitätsgewinn noch beliebige Brownfield-Übertragbarkeit.

Prototypumfang: Markdown-Tasks, automatisch erzeugtes Wiki, deterministischer Postman-Importer, Aider, LiteLLM, Ollama, Build- und Repair-Loops, CLI, minimale Windows-GUI, Wiki-Browser, Dry-Run, Status, Logs, strukturierte Events und getrennte Schutzbereiche.

Der nächste Nachweis beginnt in einer gewachsenen Codebasis.

PoC 01 war ein bewusst abgegrenzter Greenfield-Lauf. Die nächste Erprobung muss zeigen, wie sich dieselben Prinzipien unter verrauschtem Kontext, bestehenden Buildstrecken, historischen Abhängigkeiten und realen Freigabeprozessen verhalten.

F1 / Legacy-Kontext Welche Auswahl liefert genug Systemwissen, ohne widersprüchliche oder veraltete Quellen zu überladen?	F2 / Lokale Modellqualität Welche Aufgabenklassen lassen sich lokal lösen, und wann ist eine freigegebene Eskalation sinnvoll?
F3 / Holdout-Wirksamkeit Wie lässt sich messen, ob die getrennte Prüfung Reward-Hacking und Überanpassung reduziert?	F4 / Reproduzierbarkeit Welche Varianz bleibt bei identischem Task, Kontext, Modell und Harness über Wiederholungen?

VORGESCHLAGENER BROWNFIELD-PILOT

01	Aufgabenklasse begrenzen Ein relevantes, aber reversibles Arbeitspaket mit klarer Eigentümerschaft wählen.
02	Vertrauensgrenze festlegen Datenflüsse, erlaubte Modellstufen, Werkzeuge und Dateibereiche vorab dokumentieren.
03	Akzeptanzkriterien trennen Build, Tests, Holdouts und menschliche Prüfpunkte außerhalb des Agentenbereichs definieren.
04	Runs vergleichen Erfolg, Reparaturverlauf, Kontextnutzung, Eskalationen und Prüfaufwand protokollieren.
05	Bewusst entscheiden Erst nach technischer und organisatorischer Auswertung über eine Ausweitung befinden.

AKTUELLER STATUS	Die Brownfield-Erprobung ist geplant, aber noch kein abgeschlossener Nachweis. Modellendpunkte sind austauschbar, daraus folgt keine Gleichwertigkeit ihrer Fähigkeiten.
-------------------------	--

Engineering-Erfahrung trifft angewandte Agentenforschung.

Christian Binder ist Senior Software Engineer und Applied AI Engineer mit rund 20 Jahren Erfahrung in produktiven Softwaresystemen. Er arbeitet seit 2019 selbstständig und verbindet .NET- und Enterprise-Entwicklung, Legacy-Integration, datenintensive Systeme, CI/CD und Applied AI. Sein MSc in Computational Science stammt von der Universität Wien.

Der Name HEPHAUSTOS verbindet Hephaistos, den Schmied und Erbauer technischer Helfer, mit Faust, der nicht aufhören konnte zu fragen.

hi@hephaustos.com
hephaustos.com
christianbinder.com

PRIMÄRQUELLEN UND SPEZIFIKATIONEN

- [1] Sam McVeety, Amir Hormati: [Introducing the Open Knowledge Format](https://cloud.google.com/blog/products/data-analytics/.../how-the-open-knowledge-format-can-improve-data-sharing/). Google Cloud Blog, 12. Juni 2026. Abgerufen am 26. August 2026.
- [2] GoogleCloudPlatform: [Open Knowledge Format Specification, Version 0.2](https://github.com/GoogleCloudPlatform/knowledge-catalog/.../okf/SPEC.md). GitHub, 2026. Abgerufen am 26. August 2026.
- [3] Zhiyuan He et al.: [Agent Lightning v1.0: Towards Harnessed Agentic RL](https://arxiv.org/abs/2608.17528). arXiv:2608.17528, 18. August 2026.
- [4] Jiahang Lin et al.: [Agentic Harness Engineering: Observability-Driven Automatic Evolution of Coding-Agent Harnesses](https://arxiv.org/abs/2604.25850). arXiv:2604.25850, 28. April 2026.
- [5] Chengzhi Liu et al.: [Auditing Agent Harness Safety](https://arxiv.org/abs/2605.14271). arXiv:2605.14271, Mai 2026.

METHODISCHER HINWEIS

PoC-Aussagen basieren zusätzlich auf lokalen, strukturierten HEPHAUSTOS-Events des erfolgreichen Laufs vom 26. Juli 2026. Der öffentliche Auszug wurde zum Schutz realer Dokument- und Personendaten bereinigt. Rohlogs, Zugangsdaten, Kundendaten und Details des Holdout-Schutzes sind nicht Bestandteil dieses Dokuments.

© 2026 Christian Binder. [Impressum](#) · [Datenschutz](#)

Zitieren als: Binder, C. (2026): Level 3.5 Autonomie für kontrollierte agentische Softwareentwicklung. HEPHAUSTOS Whitepaper 01, v1.1.

TECHNISCHES
ERSTGESPRÄCH

30 Minuten technische Einordnung statt Produktdemo. Kontakt: hi@hephaustos.com