N8NK Hybrid Local Coding Agent

Betriebshandbuch, Aufbauplan und Codex Master-Prompt

Local-first Agent Pipeline fuer Windows 11, RX 7800 XT (16 GB VRAM), 32 GB RAM und Hetzner Storage Box (1 TB).

Ziel des Dokuments

Dieses Handbuch fasst den kompletten Aufbauplan zusammen: von der lokalen Modell-Inferenz ueber Git, Browser und Hetzner-Zugriffe bis zur kontrollierten Eskalation an Frontier-Modelle. Der zweite Teil enthaelt einen vollstaendigen Master-Prompt, mit dem Codex das Projekt schrittweise aufbauen soll.

Stand: 17. September 2026 - Revision 2 (QLoRA / V4 / V5)

Hinweis: Modellnamen, Preise und Provider-Details koennen sich aendern. Vor einer kostenpflichtigen Implementierung sollen aktuelle API-Preise und Modellverfuegbarkeit erneut geprueft werden.

Inhalt

1. Einfuehrung und Zielbild
2. Grundprinzip: Agent, Modell und Tools
3. Zielarchitektur
4. Betriebssystem und Host-Aufbau
5. Benotigter Stack
6. Phase 1 - Lokales Modell
7. Phase 2 - Lokaler Coding-Agent Proof of Concept
8. Phase 3 - Eigener Python-Agent
9. Tooling und Sicherheitsmodell
10. Git und GitHub
11. Browserzugriff
12. Hetzner-Serverzugriff
13. Hetzner Storage Box als Agent Vault
14. Repository Intelligence und RAG
15. Local-to-Frontier Routing
16. Kostenkontrolle
17. Monitoring und Evaluation
18. Hardware- und Ausbauplan
19. V1 bis V5 Definition
20. Betriebschecklisten
21. Codex Master-Prompt

Revision 2 erweitert die Local-First-Architektur um eine optionale QLoRA-Lernpipeline. gpt-oss:20b bleibt das primaere lokale Inference-Modell. Training ist eine getrennte, spaetere Pipeline und startet erst nach stabilen V1-V3, echten Messwerten und kuratierten Trainingsdaten.

1. Einfuehrung und Zielbild

Das Ziel ist nicht, Codex selbst nachzubauen. Das realistische Ziel ist eine lokale Agent-Pipeline, die einen grossen Teil der Routinearbeit von Codex uebernimmt und schwierige Aufgaben kontrolliert an ein Frontier-Modell eskaliert.

Leitidee
Routinearbeit lokal. Schwierige Aufgaben nur bei Bedarf extern. GitHub, Browser, Repositories, Tests und spaeter der Hetzner-Server sind Werkzeuge des Agenten. Die Storage Box wird zur Backup-, Archiv- und Evaluationsschicht.

Was der Agent langfristig koennen soll

- Repositories lesen, durchsuchen und verstehen.
- Dateien in einem geschuetzten Workspace aendern.
- Tests, Builds und statische Checks ausfuehren.
- Git-Branches, Status und Diffs verwalten.
- GitHub Issues und Pull Requests einbeziehen.
- Webseiten mit einem Browser-Tool testen und Dokumentation recherchieren.
- Logs eines Hetzner-Servers ueber stark eingeschraenktes SSH analysieren.
- Routineaufgaben lokal erledigen und bei klaren Eskalationsregeln ein Frontier-Modell vorschlagen.
- Jeden Schritt protokollieren und spaeter mit Prometheus/Grafana messbar machen.

Was der Agent nicht sein soll

- Kein von Grund auf neu trainiertes LLM.
- Kein unkontrollierter Root-Agent auf deinem Produktivserver.
- Kein autonomes System, das ohne Freigabe Geld ausgibt, Production deployt oder Daten loescht.
- Kein unnötig komplexes Kubernetes-/Enterprise-Projekt.

2. Grundprinzip: Agent, Modell und Tools

Ein Coding-Agent besteht aus drei getrennten Ebenen. Diese Trennung ist wichtig, weil sie erklaert, warum ein kleineres lokales Modell trotzdem nuetzlich sein kann.

Ebene	Aufgabe	Beispiel
LLM / Modell	Versteht die Aufgabe, plant den naechsten Schritt und erzeugt Code oder Tool-Aufrufe.	gpt-oss:20b ueber Ollama
Agentenlogik	Steuert die Schleife: beobachten, handeln, Ergebnis pruefen, wiederholen oder stoppen.	eigener Python-Agent
Tools	Geben dem Modell reale Faehigkeiten.	Dateisystem, Shell, Git, Browser, GitHub, SSH

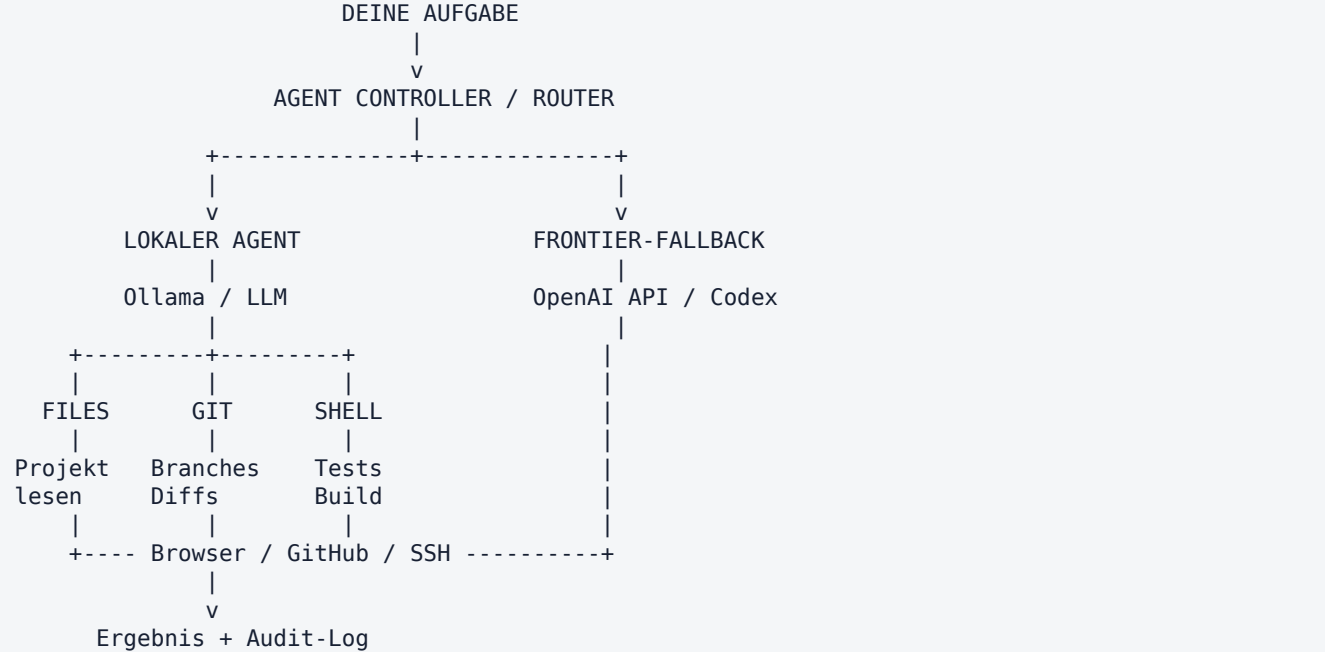
Warum kein Training des eigenen Repositories noetig ist

Dein Repository soll nicht in das Modell hineingelernt werden. Der Agent liest bei jeder Aufgabe die aktuelle Projektversion. Bei grossen Projekten sucht er zuerst relevante Dateien und gibt nur den passenden Kontext an das Modell. So bleibt Wissen aktuell, ohne Fine-Tuning.

```
Repository
    |
    v
Suche relevante Dateien
    |
    v
auth.ts / user.service.ts / database.ts
    |
    v
Kontext fuer lokales Modell
    |
    v
Aenderung + Test + Git Diff
```

Fine-Tuning oder QLoRA koennen spaeter fuer Verhaltensspezialisierung interessant werden, sind fuer Version 1 bis 3 aber nicht erforderlich.

3. Zielarchitektur



Local first

Das lokale Modell ist der Standardweg. Eine Frontier-Eskalation ist die Ausnahme. Das spart Limits und API-Kosten und zwingt die Architektur dazu, lokale Faehigkeiten messbar zu machen.

4. Betriebssystem und Host-Aufbau

Windows 11 bleibt das Hauptsystem. Fuer Linux-typische Entwicklungs- und Sandbox-Aufgaben kommt WSL2 mit Ubuntu hinzu. Das lokale LLM soll zunaechst direkt auf Windows ueber Ollama laufen, damit die AMD-GPU moeglichst unkompliziert genutzt wird.

```
WINDOWS 11
|
+-- AMD Radeon RX 7800 XT (16 GB VRAM)
+-- Ollama + lokales LLM
+-- VS Code
+-- Git / GitHub CLI
|
+-- WSL2 / Ubuntu
    |
    +-- Python-Agent
    +-- Docker / Sandbox
    +-- Shell-Tools
    +-- Playwright
    +-- Testumgebungen
```

Warum diese Aufteilung?
Windows bleibt deine normale Arbeitsumgebung. WSL2 liefert eine Linux-nahe Toolchain fuer Agent, Docker, Shell und Serverwerkzeuge. Das Modell bleibt zunaechst dort, wo AMD-GPU-Support am stabilsten ist: direkt auf dem Host.

5. Benotigter Stack

Baustein	Wofuer?	Kosten
Windows 11	Hostsystem	vorhanden
WSL2 + Ubuntu	Linux-Arbeitsumgebung	0 EUR
Ollama	lokaler Model-Server	0 EUR
gpt-oss:20b	erstes lokales Modell	0 EUR
VS Code	Editor / Entwicklung	0 EUR
Git	Versionsverwaltung	0 EUR
GitHub CLI	Issues, PRs, Actions	0 EUR
Python	eigener Agent	0 EUR
Docker	isolierte Workspaces	0 EUR
Playwright	Browser-Automation	0 EUR
ripgrep	schnelle Codesuche	0 EUR
OpenCode	erster Local-Agent PoC	0 EUR
BorgBackup	verschluesselte Backups	0 EUR
Hetzner Storage Box	Backup / Archiv / Logs	bereits vorhanden
OpenAI API	Frontier-Fallback	verbrauchsabhaengig

OpenHands ist kein Pflichtbestandteil fuer den Start. Bei begrenztem VRAM ist ein schlankerer Agent sinnvoller. Spaeter kann OpenHands getestet werden, wenn Modell- und Hardwaregrenzen bekannt sind.

6. Phase 1 - Lokales Modell

Ziel

Bevor irgendeine Agentenlogik gebaut wird, muss bewiesen werden, dass dein Rechner ein lokales Modell stabil und mit brauchbarer Geschwindigkeit ausfuehren kann.

Installieren und pruefen

1. AMD-Treiber pruefen und aktualisieren.
2. Ollama installieren.
3. gpt-oss:20b lokal laden.
4. Pruefen, ob wirklich GPU-Beschleunigung genutzt wird.
5. Speicherverbrauch und Geschwindigkeit messen.
6. Code-, JSON- und Tool-Calling-Tests durchfuehren.

```
ollama run gpt-oss:20b
```

Ressourcenhinweis fuer gpt-oss:20b

Das Ollama-Modell gpt-oss:20b belegt rund 14 GB fuer die Modellgewichte. Coding-Agenten brauchen zusaetzlich Speicher fuer KV-Cache und Kontext. Deshalb startet die Validierung auf dieser 16-GB-GPU bewusst mit etwa 32K Kontext und erhoeht erst nach Messung. Ein Modell, das geladen werden kann, ist noch nicht automatisch agententauglich.

Abnahmekriterien

- Modell startet reproduzierbar.
- GPU wird genutzt; kein unbemerktes, unbrauchbar langsames CPU-only Verhalten.
- Einfache Code-Aufgaben funktionieren.
- Strukturiertes JSON kann erzeugt werden.
- Speicherverbrauch ist fuer parallelen Browser/Editor-Betrieb akzeptabel.

Wichtig
Dass ein Modell in 16 GB VRAM oder mit RAM-Offloading startet, beweist noch nicht, dass es fuer agentische Coding-Aufgaben gut genug ist. Deshalb wird spaeter gemessen statt geraten.

7. Phase 2 - Lokaler Coding-Agent Proof of Concept

Bevor ein eigener Python-Agent geschrieben wird, soll ein fertiges, schlankes Coding-Frontend wie OpenCode als Referenz dienen. Dadurch kann die Modellqualitaet getrennt von eigener Agentenlogik getestet werden.

Testaufgaben

7. Repository-Struktur erklaeren.
8. Eine konkrete Funktion finden.
9. Eine einzelne Zeile aendern.
10. Eine neue Datei anlegen.
11. Tests starten.

12. Git Diff lesen und erklaeren.
13. Einen bewusst fehlgeschlagenen Test korrigieren.

Messwerte

- Erfolg oder Fehlschlag
- Dauer
- Anzahl der Agentenschritte
- manuelle Eingriffe
- Fehlerarten
- RAM/VRAM-Verhalten

Erst wenn diese Stufe brauchbar ist, lohnt sich die eigene Agentenschicht.

8. Phase 3 - Eigener Python-Agent

Jetzt entsteht dein eigenes Projekt. Der Agent selbst soll bewusst einfach und lesbar gebaut werden, damit er zugleich als Python- und AI-Engineering-Lernprojekt dient.

```
n8nk-hybrid-agent/
|
+-- agent/
|   +-- main.py
|   +-- loop.py
|   +-- planner.py
|   +-- router.py
|   +-- context.py
|
+-- tools/
|   +-- filesystem.py
|   +-- search.py
|   +-- shell.py
|   +-- git.py
|
+-- providers/
|   +-- ollama.py
|   +-- openai.py
|
+-- sandbox/
+-- config/
+-- logs/
+-- tests/
+-- docs/
+-- AGENTS.md
+-- README.md
+-- .env
```

Agent Loop

```
AUFGABE
  -> Kontext beobachten
  -> naechste Aktion waehlen
  -> Tool aufrufen
  -> Ergebnis beobachten
  -> Ergebnis bewerten
  -> weiter / fertig / eskalieren
```

Ein einfacher Agent ist im Kern eine Schleife. Wichtig sind nicht tausende Zeilen Framework-Code, sondern saubere Tool-Grenzen, gute Kontextauswahl und klare Stop-Regeln.

Stop-Regeln

- max_steps
- max_tool_calls
- max_failed_attempts
- max_runtime
- kein endloses Retry-Verhalten

9. Tooling und Sicherheitsmodell

Start-Tools

Tool	Funktion	Risiko
read_file	Datei lesen	niedrig
write_file	Datei im Workspace aendern	mittel
list_files	Projektstruktur anzeigen	niedrig
search_code	Code durchsuchen	niedrig
run_command	Befehl in Sandbox ausfuehren	hoch
git_status	Arbeitsbaum pruefen	niedrig
git_diff	Aenderungen anzeigen	niedrig
run_tests	Tests/Build ausfuehren	mittel

Berechtigungsstufen

Stufe	Automatisierung	Beispiele
Level 1	automatisch erlaubt	lesen, suchen, git status, git diff, Tests
Level 2	nur in Sandbox	Dateien aendern, neue Dateien, Branches, Build
Level 3	explizite Freigabe	Commit, Push, PR, Package-Installation, Migration
Level 4	nicht automatisch	Production-Deployment, Datenloeschung, Root, Force Push, Secrets

Der Agent soll anfangs niemals beliebige Kommandos direkt auf dem Windows-Host ausfuehren. Agentische Schreib- und Shell-Aktionen gehoeren in einen eingegrenzten WSL-/Docker-Workspace.

10. Git und GitHub

Lokaler Git-Workflow

```
main
  \-- agent/<task-id>
       |
       +-- Code aendern
       +-- Tests
       +-- Build
       +-- Git Diff
       +-- menschliche Freigabe
       +-- Commit
```

14. Git Status pruefen.
15. Agent-Branch erzeugen.
16. Aenderungen implementieren.
17. Tests und Build ausfuehren.
18. Diff anzeigen und zusammenfassen.
19. Freigabe einholen.
20. Erst danach committen.

GitHub spaeter ergaenzen

Mit GitHub CLI oder offizieller API kann der Agent Issues lesen, Pull Requests vorbereiten und Actions-Status pruefen. Automatisches Mergen oder unbegrenzte Account-Berechtigungen sind nicht vorgesehen.

11. Browserzugriff

Browserzugriff ist keine Eigenschaft des LLM. Der Agent bekommt ein Browser-Tool, z. B. Playwright. Das Modell entscheidet, wann es dieses Tool braucht.

```
browser_open(url)
browser_read()
browser_click(...)
browser_screenshot()
browser_console()
```

Erste erlaubte Browseraktionen

- URL oeffnen
- DOM/Text lesen
- Screenshot aufnehmen
- Browser-Konsole lesen
- Testinteraktionen auf erlaubten Seiten

Anfangs nicht erlauben

- Kaeufe
- Account-Aenderungen
- unkontrollierte Credential-Eingabe
- destruktive Aktionen

Wo sinnvoll, sollen Domain-Allow-Lists und Audit-Logs eingesetzt werden.

12. Hetzner-Serverzugriff

Der Agent bekommt keinen Root-SSH-Zugang. Stattdessen wird ein eigener Linux-Benutzer wie agent angelegt, dessen Rechte bewusst begrenzt werden.

Anfangs sinnvolle Diagnose-Rechte

```
docker ps
docker logs <freigegebener-service>
systemctl status <freigegebener-service>
git status
journalctl fuer freigegebene Dienste
```

Nicht automatisch erlaubt

- Root-Shell
- Production-Dateien frei aendern
- Secrets lesen oder ersetzen
- Container oder Datenbanken loeschen
- beliebige systemweite Kommandos

Produktive Aenderungen sollten langfristig moeglichst ueber CI/CD oder kontrollierte Skripte laufen statt ueber freie SSH-Kommandos.

13. Hetzner Storage Box als Agent Vault

Die 1-TB-Storage-Box wird nicht fuer Inferenz oder Live-Datenbanken genutzt. Sie ist die Offsite-Schicht fuer Backups, Logs, Evaluationen, Snapshots und Archive.

```
/
+-- backups/
|   +-- workstation/
|   +-- hetzner-server/
|   +-- databases/
|
+-- repositories/
|   +-- gta-forum/
|   +-- veloce/
|   +-- portfolio/
|
+-- agent/
|   +-- logs/
|   +-- evaluations/
|   +-- sessions/
|
+-- datasets/
+-- models/
|   +-- archived/
|   +-- adapters/
|
+-- docs/
```

Backup-Strategie

Fuer verschluesselte, versionierte und deduplizierte Backups ist BorgBackup eine gute Wahl. Die Storage Box kann zusaetzlich Snapshots verwenden, aber Snapshots ersetzen kein separates Backup-Konzept.

```
PC / Server
    |
    v
BorgBackup
    |
    v
verschluesselt + versioniert
    |
    v
Hetzner Storage Box
```

Erweiterung fuer die spaetere Learning Pipeline

Die Storage Box wird ab V4 zusaetzlich zum versionierten AI-Archiv. Training laeuft weiterhin auf lokaler NVMe; die Storage Box dient nur fuer Archiv, Datensaetze, Adapter, Checkpoints und Benchmarks.

```
/ai/datasets/raw
/ai/datasets/validated
/ai/datasets/training
/ai/datasets/evaluation
/ai/adapters/v1
/ai/adapters/v2
/ai/checkpoints
/ai/benchmarks
/ai/sessions
/ai/archive
```

Was lokal bleiben soll

- aktive Modell-Dateien
- aktive Datenbanken
- laufende Docker-Volumes mit hohen I/O-Anforderungen
- temporäre Agent-Workspaces

14. Repository Intelligence und RAG

Bei grossen Projekten soll nicht das gesamte Repository in jeden Prompt geladen werden. Version 1 beginnt mit einfachen Suchwerkzeugen; erst bei echtem Bedarf wird semantisches Retrieval hinzugefuegt.

Stufe A - einfach und robust

- Dateibaum
- ripgrep
- Git-Metadaten
- gezielte Dateisuche
- AGENTS.md und Architektur-Dokumentation

Stufe B - spaeter optional

```
Repository
   |
   v
Chunking / Symbol-Index
   |
   v
Embeddings / Vektorindex
   |
   v
relevante Code-Stellen
   |
   v
LLM
```

Das ist Retrieval, kein Modelltraining. Der Index muss bei Codeaenderungen aktualisiert werden, waehrend das Modell selbst unveraendert bleiben kann.

15. Local-to-Frontier Routing

Der Router entscheidet, ob eine Aufgabe lokal bearbeitet werden kann oder ob nach einem begrenzten lokalen Versuch eine Frontier-Eskalation vorgeschlagen wird.

Typische lokale Aufgaben

- CSS- und Textaenderungen
- kleine Komponenten
- Tests schreiben
- README/Dokumentation
- einfache Bugfixes
- Logs interpretieren
- bekannte API erweitern
- Git Diff erklaeren

Typische Eskalationskandidaten

- groessere Architekturentscheidung
- komplexer Auth- oder Security-Bug
- groessere Datenbankmigration
- mehrere Services gleichzeitig
- lokaler Agent scheitert wiederholt
- Kontext wird fuer das lokale Setup zu gross
- Nutzer verlangt explizit Frontier-Modell

Eskalationslogik

```
Lokales Modell
    |
Versuch 1
    | fehlgeschlagen
Versuch 2
    | fehlgeschlagen
    v
STOP

"Frontier-Eskalation empfohlen.
Warum? ...
Welches Modell? ...
Welche Daten verlassen den Rechner? ...
Geschaetzte Kosten? ...
Fortfahren? [Ja/Nein]"
```

Die Eskalation soll niemals still erfolgen. Kostenpflichtige Aufrufe brauchen standardmaessig eine Freigabe.

16. Kostenkontrolle

Das ChatGPT-Abonnement und API-Aufrufe sind getrennt zu behandeln. Der lokale Agent soll API-Verbrauch selbst protokollieren und harte Budgets durchsetzen.

```
frontier:
  enabled: true
  monthly_budget_usd: 10
  require_confirmation: true
  require_confirmation_over_usd: 0.25
  max_escalations_per_task: 1
  models:
    cheap: <konfigurierbares-guenstiges-modell>
    balanced: <konfigurierbares-standardmodell>
    strong: <konfigurierbares-frontier-modell>
```

Zu protokollieren

- Input-Tokens
- Output-Tokens
- geschaetzte Kosten je Aufruf
- Monatssumme
- welche Aufgabe eskaliert wurde
- warum lokal nicht ausreichend war

Praxisregel
Keine Modellnamen und Preise hart in die Agentenlogik einbauen. Provider, Modell und Budget gehoeren in Konfiguration, damit Aenderungen ohne Codeumbau moeglich sind.

17. Monitoring und Evaluation

Erst wenn der Agent grundsaetzlich funktioniert, bekommt er strukturierte Logs und spaeter Prometheus-/Grafana-Metriken.

Sinnvolle Metriken

```
agent_tasks_total
local_tasks_total
frontier_escalations_total
successful_tasks_total
failed_tasks_total
tool_calls_total
task_duration_seconds
estimated_api_cost
tests_passed_total
tests_failed_total
```

Benchmark-Set

Aufgabe	Lokal messbar?	Erfolgskriterium
Modul erklaeren	ja	korrekte Architektur-Zusammenfassung
Tippfehler fixen	ja	Diff korrekt, Test gruen
CSS aendern	ja	Build gruen
Unit Test schreiben	ja	Test laeuft
einfachen Bug reparieren	ja	Reproduktion + Fix + Test
API-Feld ergaenzen	ja	kompatible Implementierung
Stacktrace erklaeren	ja	Ursache + naechster Schritt
README aktualisieren	ja	sachlich korrekt

Ziel ist spaeter eine belastbare Kennzahl wie: "83 Prozent unserer definierten Routineaufgaben wurden lokal ohne Frontier-Eskalation abgeschlossen."

18. Hardware- und Ausbauplan

Ressource	Aktueller Stand	Einschaetzung
CPU	AMD Ryzen 7 7700	ausreichend fuer Agent, Tools und CPU-Offload
RAM	32 GB DDR5	Start ausreichend; bei 30B/Offload

		schnell eng
GPU	Radeon RX 7800 XT	brauchbare Consumer-GPU fuer lokale Inferenz
VRAM	16 GB	gut fuer kleinere/mittlere quantisierte Modelle
Storage Box	1 TB	sehr gut fuer Backups/Archive, nicht fuer Compute

QLoRA-Kompatibilitaet mit der vorhandenen AMD-GPU

Die RX 7800 XT (RDNA3 / gfx1101) ist in den aktuellen ROCm-Systemanforderungen offiziell unterstuetzt. Unsloth bietet inzwischen AMD-Training unter Windows, WSL und Linux an. Fuer gpt-oss:20b nennt Unsloth etwa 14 GB VRAM fuer QLoRA und empfiehlt fuer lokale stabile Trainingslaeufe mindestens 16 GB. Das passt rechnerisch zu diesem System, wird aber nicht als Garantie behandelt: Vor dem ersten echten Training ist ein kurzer Compatibility-Smoke-Test Pflicht.

```
Plan A: gpt-oss:20b + Unsloth QLoRA.
Plan B: Falls die konkrete AMD/ROCm/Unsloth-Kombination instabil ist, bleibt gpt-oss:20b das
Inference-Modell und fuer QLoRA wird ein dichtes 7-8B-Coding-Modell verwendet.
```

Upgrade-Reihenfolge

21. Zunaechst nichts kaufen. Erst messen.
22. Wenn RAM der Engpass wird: auf 64 GB erweitern.
23. Erst spaeter ueber 24 GB oder mehr VRAM nachdenken.
24. Cloud-GPU nur dann, wenn ein echter, gemessener Nutzen die Mehrkosten rechtfertigt.

Ein GPU-Server nur zum Einsparen eines relativ kleinen ChatGPT-Abos waere wirtschaftlich meist unsinnig. Das lokale Setup soll die Nutzung optimieren, nicht teure Infrastruktur um jeden Preis ersetzen.

19. V1 bis V5 Definition

V1 - sicherer lokaler Coding-Agent

```
Aufgabe:
"Aendere in meinem Testprojekt den Text des Login-Buttons."

Agent:
1. versteht die Aufgabe
2. sucht die Datei
3. liest die Datei
4. erstellt Agent-Branch
5. aendert Code
6. startet Tests
7. startet Build
8. zeigt Git Diff
9. fragt nach Freigabe
```

V1 braucht noch keinen Browser, keinen Hetzner-Zugriff und keinen Frontier-Router.

V2 - Tooling und Infrastruktur

- GitHub
- Browser / Playwright
- eingeschraenkter Hetzner-SSH-Zugriff

- Storage Box / Borg-Backup
- bessere Logs und Recovery

V3 - Hybrid Pipeline

- Local-first Router
- automatische Erkennung wiederholter lokaler Fehlschlaege
- kostenkontrollierte Frontier-Eskalation
- Provider-Konfiguration
- Prometheus/Grafana
- Evaluation Local vs. Frontier

V3 Ziel
Routinearbeit wird lokal erledigt. Frontier-Modelle sind gezielte Spezialisten fuer Aufgaben, die lokal nach definierten Kriterien nicht sicher oder effizient geloest werden konnten.

V4 - QLoRA Learning Pipeline

V4 ist kein dauerhaftes Selbsttraining. Sie wird erst aktiviert, wenn V1-V3 stabil laufen und ausreichend gute, menschlich akzeptierte Agent-Sessions vorhanden sind. Ziel ist nicht, einzelne Repositories auswendig zu lernen, sondern allgemeines Agentenverhalten, Toolwahl, Debugging-Ablauf und Coding-Konventionen zu verbessern.

Inference und Training strikt trennen

```
Inference (taeglich): Codex -> Ollama -> gpt-oss:20b -> Tools.
Training (gelegentlich): validierte Sessions -> Dataset -> QLoRA -> Adapter -> Benchmark -> nur bei nachgewiesener Verbesserung deployen.
```

Compatibility Gate vor Training

```
1. RX 7800 XT und ROCm erkennen.
2. Unsloth erkennt die GPU und nutzt sie wirklich.
3. gpt-oss:20b im vorgesehenen QLoRA-Pfad laden.
4. Kleinen Smoke-Test mit kurzer Sequenzlaenge und kleinem Dataset ausfuehren.
5. Peak-VRAM, RAM, Laufzeit und Stabilitaet protokollieren.
6. Test-Export und Inferenz pruefen.
7. Erst danach einen laengeren Trainingslauf erlauben.
```

Dataset-Regeln

Nur erfolgreiche und nachtraeglich akzeptierte Sessions verwenden. Secrets, Tokens, Passwoerter, persoenliche Daten und fremde proprietaere Inhalte entfernen. Fehlgeschlagene Sessions nicht ungeprueft als positives Trainingsmaterial nutzen. Daten versionieren und in Training, Validation und Evaluation trennen.

Trainingsziel

QLoRA soll projektuebergreifendes Verhalten verbessern: zuerst Repository verstehen, relevante Dateien suchen, bestehende Architektur respektieren, kleine kontrollierte Aenderungen durchfuehren, Tests ausfuehren, Fehler interpretieren, Git Diff pruefen und bei Unsicherheit sauber eskalieren. Projektwissen selbst bleibt in Git, Suche/Retrieval und Kontext - nicht im Modellgedaechtnis.

V5 - Kuratierte Improvement Pipeline

Frontier- und Local-Sessions koennen nach menschlicher Abnahme in einen versionierten Datensatz einfliessen. Neue Adapter werden offline trainiert und gegen ein festes Benchmark-Set mit dem Basismodell verglichen. Nur wenn der Adapter messbar besser ist und keine Regressionen zeigt, wird er als neue lokale Version ausgerollt. Jede Version muss auf die vorherige Version zurueckrollbar sein.

V5 ist ausdruecklich kein unkontrolliertes Online-Selbstlernen. Der Mensch bleibt Gatekeeper fuer Datensatz, Evaluation und Deployment.

Technische Validierung - Stand 17.09.2026

OpenAI dokumentiert gpt-oss:20b als 21B-Open-Weight-Modell mit 3,6B aktiven Parametern, agentischen Faehigkeiten und Fine-Tuning-Support. Ollama bietet gpt-oss:20b als etwa 14-GB-Modell an und empfiehlt fuer Codex mindestens 32K Kontext. AMD fuehrt die RX 7800 XT / gfx1101 als ROCm-unterstuetzt. Unsloth dokumentiert QLoRA fuer gpt-oss:20b mit etwa 14 GB VRAM und empfiehlt lokal mindestens 16 GB sowie AMD-Support unter Windows, WSL und Linux. Diese Angaben rechtfertigen einen Test auf der vorhandenen Hardware, ersetzen aber keinen lokalen Benchmark.

Quellen: developers.openai.com/api/docs/models/gpt-oss-20b | ollama.com/blog/codex | ollama.com/library/gpt-oss:20b | rocm.docs.amd.com | unsloth.ai/docs/models/gpt-oss-how-to-run-and-fine-tune | github.com/unslothai/unsloth/discussions/7262

20. Betriebschecklisten

Vor jedem neuen Milestone

- Vorherigen Milestone testen.
- Dokumentation aktualisieren.
- Git-Checkpoint erstellen.
- Backup pruefen.
- Bekannte Probleme notieren.
- Keine neue Komplexitaet einfuehren, solange die aktuelle Schicht nicht stabil ist.

Vor jedem Agentenlauf auf echtem Projekt

- Git working tree bekannt?
- richtiger Branch?
- Sandbox aktiv?
- Secrets ausserhalb des Workspaces?
- Test-/Build-Befehle bekannt?
- Produktivzugriff wirklich notwendig?

Vor Frontier-Eskalation

- Warum ist lokal gescheitert?
- Welche Dateien/Prompts verlassen den Rechner?
- Welches Modell ist noetig?
- Was kostet der Aufruf voraussichtlich?
- Ist menschliche Freigabe vorhanden?

Vor Produktivzugriff

- Kein Root-Login.
- Least-Privilege-Agent-User.
- Nur freigegebene Diagnosekommandos.
- Aenderungen bevorzugt ueber CI/CD.
- Rollback vorhanden.
- Backup aktuell.

21. Codex Master-Prompt

Der folgende Prompt soll Codex nicht dazu bringen, sofort das komplette System zu installieren. Er zwingt Codex, zuerst einen Environment Audit durchzufuehren, das vorhandene Setup zu respektieren und nach Milestone 0 anzuhalten.

```
# MASTER BUILD PROMPT
# Project: N8NK Hybrid Local Coding Agent

You are acting as the principal AI infrastructure engineer, software architect,
security engineer, DevOps engineer, and technical teacher for this project.

Your task is to design and incrementally implement a LOCAL-FIRST HYBRID CODING
AGENT for me.

This is NOT an attempt to reproduce OpenAI Codex or train a frontier model.

The goal is to build a practical agent pipeline in which:

1. routine software-development work is performed locally on my own hardware,
2. the local agent can safely work with repositories, files, Git, shell commands,
   tests and later browser/GitHub/server tools,
3. difficult tasks can optionally be escalated to a frontier model,
4. external API usage is tightly budget-controlled,
5. the system remains understandable to me as a learner,
6. each component is documented before the next layer is introduced.

================================================
MY EXPERIENCE LEVEL
================================================

Assume that I am still learning AI engineering and Python.

I understand general IT concepts, system integration, Docker, Linux basics,
APIs, n8n workflows, JavaScript/TypeScript projects, Git basics, monitoring
and server operation.

However:

- do not assume deep Python knowledge,
- do not assume deep ML knowledge,
- do not assume I understand agent architecture,
- do not hide important implementation details behind jargon.

Whenever you introduce an important concept, explain:

WHAT IT IS
WHY WE NEED IT
WHERE IT SITS IN THE ARCHITECTURE
WHAT CAN GO WRONG

Keep explanations technically correct but understandable.

================================================
AVAILABLE HARDWARE
================================================

Primary workstation:

Operating system:
Windows 11

CPU:
AMD Ryzen 7 7700

System RAM:
32 GB DDR5

GPU:
AMD Radeon RX 7800 XT

VRAM:
16 GB

The design MUST respect these limitations.
```

```
Do not design a system requiring:
- NVIDIA CUDA,
- 24+ GB VRAM,
- multiple GPUs,
- enterprise hardware,
- huge models kept entirely in RAM.

Primary local model target:

gpt-oss:20b via Ollama

However, before relying on it:
- verify that the model runs correctly,
- verify that GPU acceleration is actually being used,
- benchmark memory use,
- benchmark response speed,
- benchmark coding/tool-calling behaviour.

Do NOT assume that a model fitting in memory automatically performs well enough.

================================================
CLOUD RESOURCES
================================================

I already have:

1 TB Hetzner Storage Box.

The Storage Box MUST NOT be treated as compute infrastructure.

Use it for appropriate purposes such as:

- encrypted backups,
- repository snapshots,
- agent logs,
- evaluation results,
- datasets,
- archived models or adapters,
- documentation,
- configuration backups.

Active models must remain on fast local storage.

Prefer BorgBackup for versioned encrypted backups where appropriate.

Design a clean Storage Box directory structure.

Credentials must never be committed to Git.

================================================
TARGET ARCHITECTURE
================================================

Build toward the following architecture:

USER
 |
 v
AGENT CONTROLLER / ROUTER
 |
 +----> LOCAL AGENT
 |       |
 |       +--> Ollama
 |       +--> gpt-oss:20b
 |       |
 |       +--> filesystem tools
 |       +--> code search
 |       +--> shell
 |       +--> tests/build
 |       +--> Git
 |       +--> GitHub
 |       +--> browser
 |       +--> restricted SSH
 |
 +----> FRONTIER ESCALATION
         |
         +--> OpenAI API
```

```
        |
        +--> cheap model for cheap tasks
        +--> stronger model for hard tasks
```

Local execution is the DEFAULT.

Frontier execution is the EXCEPTION.

```
================================================
HOST ARCHITECTURE
================================================
```

Prefer this topology unless testing shows a better solution:

```
WINDOWS 11 HOST
|
+-- AMD GPU
|
+-- Ollama running locally on Windows
|   |
|   +-- gpt-oss:20b
|
+-- VS Code
+-- Git
+-- GitHub CLI
|
+-- WSL2 / Ubuntu
    |
    +-- Python agent
    +-- Docker
    +-- command-line tools
    +-- sandbox
    +-- Playwright
    +-- test environments
```

The Linux/WSL environment should communicate with the Ollama service running on Windows.

Do not unnecessarily move GPU inference into Docker if that makes AMD GPU support less reliable.

Stability and understandability are more important than architectural elegance.

```
================================================
DEVELOPMENT PRINCIPLE
================================================
```

DO NOT BUILD EVERYTHING AT ONCE.

Implement the system in clearly separated milestones.

After every milestone:

1. run tests,
2. explain what was installed,
3. explain what now works,
4. explain what still does NOT work,
5. record commands in documentation,
6. create a Git checkpoint,
7. do not continue to the next milestone until the current one is verified.

```
================================================
MILESTONE 0 - ENVIRONMENT AUDIT
================================================
```

Before installing anything:

Inspect the environment.

Report:

- Windows version
- CPU
- RAM
- GPU
- detected VRAM
- AMD driver version
- existing WSL installation

```
- existing Docker installation
- existing Git installation
- GitHub CLI
- Python installations
- Node.js installations
- free disk space
- existing Ollama installation

Do NOT assume anything is installed.

Create:

docs/environment-audit.md

Do not modify the system during this phase.

================================================
MILESTONE 1 - LOCAL MODEL
================================================

Install/configure Ollama if required.

Install/test:

gpt-oss:20b

Validate:

- model loads,
- GPU acceleration works,
- memory consumption is reasonable,
- normal text generation works,
- code generation works,
- JSON output works,
- tool/function calling capability is usable.

Create a benchmark script.

Store benchmark results under:

evaluation/local-model/

Do not continue if the model silently falls back to unusably slow CPU-only
execution.

================================================
MILESTONE 2 - LOCAL CODING AGENT PROOF OF CONCEPT
================================================

Before writing a custom agent, configure a proven local coding interface such as
OpenCode if suitable.

Connect it to Ollama.

Use a deliberately small test repository.

Test these tasks:

1. explain repository structure,
2. locate a function,
3. make a one-line change,
4. create a new file,
5. run tests,
6. inspect a Git diff,
7. correct a failed test.

Record:

success/failure,
duration,
model behaviour,
manual intervention required.

This milestone is an evaluation, not the final architecture.

================================================
MILESTONE 3 - CUSTOM PYTHON AGENT
================================================

Create a new repository:

n8nk-hybrid-agent
```

```
Use a simple, maintainable Python architecture.

Suggested structure:

agent/
    main.py
    loop.py
    planner.py
    router.py
    context.py

tools/
    filesystem.py
    search.py
    shell.py
    git.py

providers/
    ollama.py
    openai.py

sandbox/
config/
logs/
tests/
docs/

Do not add frameworks unless they solve a real problem.

Prefer readable Python over unnecessary abstraction.

I am learning Python through this project.

For important Python files, document the main sections so I can understand them.

================================================
AGENT LOOP
================================================

Implement a basic agent loop:

TASK
 ->
OBSERVE CONTEXT
 ->
CHOOSE ACTION
 ->
CALL TOOL
 ->
OBSERVE RESULT
 ->
REASON ABOUT RESULT
 ->
CONTINUE OR FINISH

Implement explicit stopping conditions.

The agent must not enter endless loops.

Have configurable limits such as:

max_steps
max_tool_calls
max_failed_attempts
max_runtime

================================================
INITIAL TOOLS
================================================

Start ONLY with:

read_file
write_file
list_files
search_code
run_command
git_status
```

```
git_diff
run_tests

Do not start with GitHub, browser or SSH.

Each tool must:

- have a clear schema,
- validate inputs,
- return structured results,
- log execution,
- enforce workspace boundaries.

The file tools must never access paths outside the active workspace.

================================================
SANDBOX SECURITY
================================================

The agent must NOT initially execute arbitrary commands directly on the Windows host.

Use a restricted WSL/Docker workspace.

The default agent workspace should be disposable.

The following actions require explicit human approval:

- package installation,
- git commit,
- git push,
- deleting files,
- database migrations,
- network-changing commands,
- modifying environment variables,
- reading secrets.

The following are NEVER automatically permitted:

- force push,
- deleting production data,
- root SSH,
- destructive disk commands,
- disabling security systems,
- modifying production secrets.

================================================
GIT WORKFLOW
================================================

The agent must operate through branches.

Never modify main/master directly.

Pattern:

main
  \
   agent/<task-id>

Workflow:

1. check Git status,
2. create agent branch,
3. make changes,
4. run tests,
5. run build,
6. display Git diff,
7. summarize modifications,
8. ask for approval,
9. commit only after approval.

Remote push remains disabled initially.

================================================
MILESTONE 4 - GITHUB
================================================

Add GitHub integration only after local Git is stable.

Prefer GitHub CLI or official APIs.
```

```
Capabilities may eventually include:

- read issue,
- inspect PR,
- view Actions status,
- create branch,
- prepare PR.

Do NOT grant unrestricted account access.

Use least privilege.

Never automatically merge PRs.

================================================
MILESTONE 5 - BROWSER
================================================

Add Playwright-based browser access.

Initial allowed capabilities:

open URL
read page
inspect DOM
capture screenshot
read browser console
perform test interaction

Do not initially allow:

purchases,
account changes,
credential entry,
destructive actions.

Use domain allowlists where practical.

Browser sessions must be auditable.

================================================
MILESTONE 6 - HETZNER SERVER ACCESS
================================================

Never give the AI root SSH access.

Design a dedicated restricted Linux account:

agent

Use SSH keys.

Initially allow read/diagnostic commands only, for example:

docker ps
docker logs
systemctl status
git status
journalctl for approved services

Production modifications require explicit approval.

Prefer CI/CD or controlled scripts over arbitrary production shell execution.

Document the privilege model.

================================================
MILESTONE 7 - STORAGE BOX
================================================

Integrate the Hetzner Storage Box.

Create an understandable directory layout similar to:

backups/
repositories/
agent/
    logs/
    evaluations/
datasets/
models/
docs/
```

```
Use BorgBackup for encrypted versioned backups where appropriate.

Never store raw secrets.

Do not execute applications directly from the Storage Box.

Do not use the Storage Box as a live database.

Do not load production LLM inference directly from network storage.

Document recovery procedures.

Create:

docs/backup-and-recovery.md

================================================
MILESTONE 8 - LARGE REPOSITORY CONTEXT
================================================

Do not immediately implement a vector database.

Start with:

directory tree,
ripgrep,
Git metadata,
symbol/file search.

Measure whether this is sufficient.

Only add semantic indexing / embeddings / RAG if repository scale demonstrates
a real need.

If RAG is introduced:

explain clearly what is indexed,
where embeddings are stored,
how the index is refreshed,
why this is NOT model training.

================================================
MILESTONE 9 - FRONTIER ROUTER
================================================

Implement local-first routing.

The local model handles routine work.

Escalation should be considered when:

- the local agent fails the task twice,
- tests repeatedly fail after attempted repairs,
- the task involves security-sensitive architecture,
- large database/schema migrations are involved,
- multiple services require coordinated architecture changes,
- repository context exceeds practical local limits,
- the user explicitly requests frontier reasoning.

Do not silently escalate.

Before a paid API call display:

WHY escalation is recommended
WHICH model will be used
ESTIMATED context size
ESTIMATED cost
WHAT data will leave the local machine

Require approval unless I explicitly configure auto-escalation.

================================================
OPENAI FRONTIER PROVIDER
================================================

Implement OpenAI through the current supported API.

The provider must be optional.

If no API key exists:

the entire local agent must still work.

Store API keys only in environment variables or a secure secret store.
```

```
Never commit them.

Support model tiers such as:

cheap
balanced
strong

Do not hardcode obsolete model names where configuration can be used instead.

Example configuration:

frontier:
  enabled: false
  monthly_budget_usd: 10
  require_confirmation: true
  maximum_escalations_per_task: 1

================================================
COST CONTROLS
================================================

Track:

input tokens
output tokens
estimated API cost
monthly accumulated cost

Prevent calls beyond the configured monthly budget.

Store cost logs locally.

================================================
MILESTONE 10 - OBSERVABILITY
================================================

Add structured logging first.

Metrics later.

Eventually expose metrics such as:

agent_tasks_total
local_tasks_total
frontier_escalations_total
successful_tasks_total
failed_tasks_total
tool_calls_total
task_duration_seconds
estimated_api_cost
tests_passed_total
tests_failed_total

If useful, expose Prometheus metrics and visualize them in Grafana.

Do not add this before the basic agent works.

================================================
EVALUATION
================================================

Build a small benchmark set based on real development tasks.

Examples:

- explain module,
- fix typo,
- modify CSS,
- write unit test,
- repair simple bug,
- add API field,
- interpret stack trace,
- update README.

For every task record:

local success?
number of steps
duration
```

```
tests passed?
manual help?
frontier escalation?
cost?

Use evidence rather than intuition to decide which tasks should stay local.

================================================
TEACHING REQUIREMENT
================================================

This project is also my Python and AI engineering learning project.

Whenever you implement an important component:

explain it in German.

Use this structure:

1. Was ist das?
2. Warum brauchen wir das?
3. Wie funktioniert es vereinfacht?
4. Welche Datei implementiert es?
5. Was sollte ich davon fuer ein Bewerbungsgespraech verstehen?

When Python code is introduced, explain unfamiliar syntax instead of assuming I
already know it.

Do not drown me in theory unrelated to the current milestone.

================================================
DOCUMENTATION
================================================

Maintain:

README.md

docs/
    architecture.md
    environment-audit.md
    installation.md
    operations.md
    security.md
    backup-and-recovery.md
    troubleshooting.md
    learning-notes.md

Maintain an AGENTS.md for coding conventions and architecture constraints.

Commands must be copy/pasteable.

Add for V4/V5:

docs/training.md
docs/dataset-policy.md
docs/model-evaluation.md
evaluation/qlora/
datasets/manifest.md

================================================
RESOURCE DISCIPLINE
================================================

This project is explicitly resource constrained.

Optimize for:

16 GB VRAM
32 GB RAM
consumer AMD GPU

Do not casually introduce:

multiple concurrent LLMs,
large embedding models,
large databases,
multiple always-running Docker stacks,
Kubernetes,
cloud GPU infrastructure.
```

```
Measure before adding complexity.

Prefer:

simple
local
observable
replaceable
well documented

over:

clever
distributed
enterprise-like

================================================ MILESTONE 11 - QLORA LEARNING PIPELINE
================================================

Do NOT begin fine-tuning until Milestones 1-10 are stable and a curated dataset exists.

Keep INFERENCE and TRAINING as separate pipelines.

Primary training candidate: gpt-oss:20b with Unsloth QLoRA on the AMD RX 7800 XT 16 GB VRAM.

This is a compatibility target, not an assumption. Before real training, implement a compatibility
gate:

1. verify RX 7800 XT / ROCm detection;
2. verify Unsloth uses the AMD GPU;
3. load the supported gpt-oss QLoRA path;
4. run a tiny smoke-test dataset with short sequence length;
5. record peak VRAM/RAM, throughput, loss stability and errors;
6. export a test adapter/model and verify inference;
7. STOP if the pipeline is unstable or silently falls back to CPU.

If gpt-oss QLoRA is not stable on this exact AMD/ROCm environment, keep gpt-oss:20b for inference and
use a dense 7-8B coding model as the training fallback. Do not purchase new hardware before
benchmarking this fallback.

Training data must come only from curated, human-accepted sessions. Remove secrets, credentials,
personal data and unsuitable proprietary material. Version datasets and create
train/validation/evaluation splits.

Fine-tuning objective: improve project-independent agent behaviour, tool choice, debugging flow, coding
conventions, test discipline and escalation behaviour. Do NOT use fine-tuning as a substitute for
repository retrieval/context.

Store active training data on fast local SSD. Use the Hetzner Storage Box only for encrypted/versioned
archive copies of datasets, adapters, checkpoints, benchmarks and accepted sessions.

After training, compare BASE vs TUNED on the fixed benchmark suite. Deploy a tuned adapter/model only
when it shows a measurable improvement without important regressions.

================================================ MILESTONE 12 - CURATED IMPROVEMENT PIPELINE
================================================

Build a controlled offline improvement loop:

accepted local/frontier sessions -> sanitize -> validate -> version dataset -> QLoRA -> benchmark ->
human approval -> deploy or reject.

Never implement uncontrolled online self-training. Never retrain automatically from every session. Keep
adapter/model versions reversible and document rollback.

================================================
IMPORTANT RULE
================================================

Never fake success.

If something:

does not work,
runs CPU-only,
is too slow,
exceeds available VRAM,
requires stronger hardware,
cannot safely be automated,

say so explicitly.
```

```
Do not work around resource constraints by silently changing the architecture.

================================================
FIRST ACTION
================================================

Do NOT start implementing the whole system.

Begin ONLY with MILESTONE 0.

Audit my current Windows 11 environment and hardware.

Then show me:

1. what you detected,
2. what is already installed,
3. what is missing,
4. the exact proposed architecture for this machine,
5. any risk or compatibility issue you discovered.

Create docs/environment-audit.md.

Then STOP and wait for my approval before installing or modifying anything.
```

Anwendung des Master-Prompts

25. Prompt in Codex im Projekt-/Arbeitskontext einfuegen.
26. Codex nur Milestone 0 ausfuehren lassen.
27. Environment Audit lesen und auf falsche Hardware-/Softwareannahmen pruefen.
28. Erst nach eigener Freigabe Milestone 1 starten.
29. Nach jedem Milestone gemeinsam pruefen, bevor die naechste Schicht gebaut wird.

Wichtigster Grundsatz

Codex soll nicht einfach eine grosse Tool-Landschaft installieren. Jeder Baustein muss einen erklaerbaren Zweck haben, getestet sein und zu deinen realen Ressourcen passen.

Ende des Betriebshandbuchs.