Hybride Multi-LLM-Orchestrierung als Architekturmuster für professionelle KI-Arbeitsumgebungen

Johann Jörgen Schübeler

Schübeler Consulting — Technische Kommunikation · Technische Dokumentation · Digitalisierung · Künstliche Intelligenz

www.schuebeler-consulting.de

Fachbeitrag — April 2026

Version 4.0

Schlüsselwörter: Multi-Agent-Orchestrierung, Large Language Models, Cross-Agent Memory, Model Context Protocol, funktionale Spezialisierung, LLM-Rate-Limits, lokale Inferenz

Inhaltsverzeichnis

(Inhaltsverzeichnis in Word mit F9 aktualisieren)

Zusammenfassung

Die intensive Nutzung großer Sprachmodelle (Large Language Models, LLMs) in professionellen Arbeitsumgebungen stößt auf ein strukturelles Problem: Provider-spezifische Rate-Limits erschöpfen sich bei intensivem Einsatz schneller als erwartet, unabhängig vom gewählten Abonnement-Tier. Dieser Beitrag beschreibt ein Architekturmuster, das diesem Problem durch funktionale Spezialisierung begegnet. Drei Cloud-LLMs — Claude (Anthropic) als Orchestrator, Gemini CLI (Google) und Codex CLI (OpenAI) als Spezialisten — werden ergänzt durch ein lokales Open-Source-Modell für volumenstarke Routineaufgaben. Alle Komponenten werden über einen zentralen Koordinator integriert, der Aufgaben nach Kompetenzprofil und Verfügbarkeit delegiert. Als persistente Wissensbasis dient ein Cross-Agent Memory System auf SQLite-Basis, das Session- und Agentengrenzen überbrückt. Der Beitrag dokumentiert die Systemarchitektur, das Memory-System, ausgewählte Sicherheitsmechanismen und Praxisbeobachtungen aus dem Produktiveinsatz. Limitationen werden explizit benannt. Das beschriebene Setup verarbeitet keine Kundendaten.

1. Einleitung und Problemstellung

Wer LLM-gestützte Werkzeuge täglich und intensiv im professionellen Kontext einsetzt, lernt ein Problem kennen, das in der akademischen KI-Diskussion selten vorkommt: das Rate-Limit-Problem.

Jeder kommerzielle LLM-Provider arbeitet mit Nutzungsobergrenzen für Anfragen pro Minute, Tokens pro Tag oder als rollendes Sieben-Tage-Kontingent. In der hier beschriebenen Arbeitsumgebung kommen drei Abonnements zum Einsatz: Claude Max 5 (Anthropic, rollendes Sieben-Tage-Limit), Google AI Pro (Gemini) und ChatGPT Business (OpenAI/Codex). Trotz des höchsten verfügbaren Claude-Plans erreichte das System in einem typischen Arbeitsszenario mit mehreren parallel laufenden Projekten am dritten Arbeitstag rund 96 Prozent der verfügbaren Kapazität.

Das Phänomen beschreibt einen Ressourcenengpass im Sinne der Theory of Constraints [1]. Der Engpass entsteht in der Verfügbarkeit der Kapazität, die Leistungsfähigkeit des Modells selbst bleibt davon unberührt. Ein höherwertiges Abonnement adressiert lediglich das Symptom. Die Ursache liegt tiefer: in der Monokultur, also der vollständigen Abhängigkeit von einem einzelnen Modell und einem einzelnen Provider.

Diese Abhängigkeit erzeugt drei strukturelle Risiken. Das erste ist das Verfügbarkeitsrisiko: Ein Provider-Ausfall legt die gesamte Arbeitsumgebung lahm. Das zweite ist das Ressourcenrisiko: Alle Aufgaben belasten dasselbe hochwertige Kontingent, auch einfache und repetitive. Das dritte ist das Kompetenzrisiko: Kein einzelnes Modell ist in allen Domänen gleich stark. Ein Reasoning-optimiertes Modell für Boilerplate-Codegenerierung einzusetzen ist eine Ressourcenfehlallokation.

Der vorliegende Beitrag dokumentiert ein Architekturmuster, das diese Risiken durch funktionale Spezialisierung und eine gemeinsame Persistenzschicht adressiert. Die Zielsetzung ist dreifach: Erstens die Dokumentation der Systemarchitektur als reproduzierbares Muster für Praktiker. Zweitens die

Beschreibung eines Cross-Agent Memory Systems, das persistentes Kontextwissen agenten- und sessionsübergreifend bereitstellt. Drittens eine ehrliche Einschätzung der Praxistauglichkeit auf Basis qualifizierbarer Beobachtungen, einschließlich der Stellen, an denen das System noch nicht das leistet, was es leisten soll.

2. Methodische Einordnung

Dieser Beitrag ist ein strukturierter Praxisbericht, kein kontrolliertes Experiment. Die beschriebene Architektur wurde im Produktiveinsatz eines Einzelunternehmens iterativ entwickelt und eingesetzt. Das Setup verarbeitet keine personenbezogenen Daten und keine Kundendaten. Inhalte, die durch die LLMs laufen, sind Eigenentwicklungen, Fachartikel, Konzeptarbeiten und öffentlich verfügbare Dokumentationen.

Die berichteten Beobachtungen sind qualitativ. Es liegen keine A/B-Tests vor, keine statistisch belastbaren Stichproben, keine automatisierte Telemetrie. Alle Angaben zu Delegationsquoten, Limit-Auslastung und Antwortverhalten basieren auf Schätzungen und informellen Beobachtungen im laufenden Betrieb. Wo Schätzwerte genannt werden, sind sie als solche gekennzeichnet.

Es sei ausdrücklich darauf hingewiesen, dass der Autor gleichzeitig Entwickler, Betreiber und Evaluator des Systems ist. Diese Personalunion birgt ein inhärentes Bestätigungsbias-Risiko: Positive Beobachtungen können übergewichtet, negative können unterberichtet sein. Der Wert dieses Beitrags liegt in der detaillierten Dokumentation einer funktionierenden Architektur, die als Ausgangspunkt für systematischere Untersuchungen dienen kann.

3. Stand der Technik

3.1 Multi-Agent-Frameworks

Die Orchestrierung mehrerer LLM-Agenten ist Gegenstand aktiver Forschung und Entwicklung. Drei etablierte Frameworks sind hier relevant.

AutoGen (Microsoft Research) implementiert ein konversationsbasiertes Multi-Agent-Framework, in dem spezialisierte Agenten in strukturierten Dialogen zusammenarbeiten [2]. AutoGen setzt auf Python als Laufzeitumgebung und erfordert die explizite Programmierung der Interaktionsstrukturen. Der Ansatz eignet sich für Szenarien, in denen die Kommunikationsstruktur zwischen Agenten vorab definiert werden kann.

CrewAI verfolgt einen rollenbasierten Ansatz [3]. Agenten werden als Crew-Mitglieder mit definierten Rollen, Zielen und Werkzeugen konfiguriert. Die Modellierung von Teamstrukturen ist intuitiv. Die Notwendigkeit, alle Agenten innerhalb desselben Python-Prozesses zu betreiben, limitiert jedoch die Flexibilität bei der Nutzung heterogener Provider.

LangGraph (LangChain) modelliert Agent-Workflows als gerichtete Graphen mit Zustandsmanagement [4]. Der Ansatz ermöglicht komplexe, bedingte Ausführungspfade. Er erfordert erheblichen Konfigurationsaufwand und bindet den Anwender an das LangChain-Ökosystem.

3.2 Model Context Protocol

Das Model Context Protocol (MCP), spezifiziert von Anthropic, standardisiert die Schnittstelle zwischen LLM-Anwendungen und externen Werkzeugen [5]. MCP definiert ein JSON-RPC-basiertes Protokoll über stdio oder Server-Sent Events, das Sprachmodellen strukturierten Zugriff auf externe Ressourcen, Werkzeuge und Datenquellen ermöglicht. Im hier beschriebenen System ist MCP der Kommunikationskanal zwischen dem Orchestrator und dem Memory-System.

3.3 Abgrenzung des vorliegenden Ansatzes

Der hier beschriebene Ansatz unterscheidet sich grundlegend von den genannten Frameworks. Es handelt sich um ein **Konfigurationsmuster auf bestehenden CLI-Werkzeugen** statt um ein eigenständiges Software-Framework. Claude Code, Gemini CLI und Codex CLI sind Anwendungen, die von ihren jeweiligen Providern bereitgestellt werden. Die Orchestrierung erfolgt durch die Integration dieser CLIs über Slash-Commands und Shell-Aufrufe innerhalb einer einzigen Terminal-Umgebung, also ohne programmatische API-Aufrufe.

Der praktische Vorteil liegt in der Abwesenheit eines Framework-Lock-ins: Jede Komponente lässt sich unabhängig austauschen, ohne dass die Gesamtarchitektur angepasst werden müsste. Die Skalierbarkeit ist im Gegenzug begrenzt. Das Muster ist auf den Single-User-Betrieb ausgelegt. Für Unternehmensumgebungen mit mehreren Nutzern und standardisierten Workflows sind die genannten Frameworks besser geeignet.

4. Systemarchitektur

4.1 Komponentenübersicht

Das System besteht aus vier LLM-Instanzen, einem zentralen Memory-System und zwei Hook-basierten Sicherheitsmechanismen. Die Komponenten kommunizieren über drei verschiedene Kanäle: Slash-Commands (Gemini, Codex), eine HTTP-API (lokales Modell) und das MCP-Protokoll (Memory Server).

Datenfluss in der Übersicht:

```
Claude (Orchestrator)
  |-- /gemini → Gemini CLI → Bash: memory_cli.py → SQLite DB
  |-- /codex  → Codex CLI  → Bash: memory_cli.py → SQLite DB
  |-- HTTP    → lokales Open-Source-Modell via LM Studio
  |-- MCP     → memory_server.py (stdio) → SQLite DB (WAL)
  |
  PreToolUse:  identity_lint.py (blockiert bei Verstoß)
  PostToolUse: memory_sync.py   (schreibt nach Write/Edit)
```

4.2 Der Orchestrator: Claude als Koordinator

Claude Opus 4.6 (1M) fungiert als zentraler Orchestrator. Er ist der einzige Agent mit vollständigem Kontextwissen über laufende Projekte und Systemkonfiguration. Seine Rolle umfasst die Koordination: Aufgabenanalyse, Delegation an den geeigneten Spezialisten, Ergebnis-Integration und Qualitätskontrolle. Die Ausführung der einzelnen Arbeitsschritte verbleibt ausdrücklich bei den Spezialisten.

Die Delegation folgt dokumentierten Richtlinien, die in der Konfigurationsdatei `CLAUDE.md` festgelegt sind:

Aufgabentyp	Delegationsziel	Begründung
Recherche, Dokumentenanalyse, lange Kontexte	Gemini CLI	Großes Kontextfenster, eigenes Kontingent
Code-Generierung, Refactoring, Debugging	Codex CLI	Spezialisierung auf Code, eigenes Kontingent
Repetitive Textgenerierung, Templates	Lokales Open-Source-Modell	Keine Token-Kosten, keine Cloud-Abhängigkeit
Koordination, Kontext-Synthese	Claude (selbst)	Vollständiges Kontextwissen, Reasoning-Stärke

Tabelle 1: Delegationsregeln nach Aufgabentyp.

Die Formulierung „regelbasiert" ist bewusst gewählt. Es handelt sich um dokumentierte Richtlinien, die der Orchestrator in Kombination mit dem aktuellen Kontext anwendet. Ein deterministischer Klassifikator ist nicht implementiert.

4.3 Spezialisierte Agenten

Gemini CLI [6] wird über den Slash-Command `/gemini` angesprochen. Der Command übergibt einen Prompt an die Gemini CLI, empfängt die Antwort als Text und stellt sie dem Orchestrator zur Verfügung. Gemini wird primär für Recherche-Aufgaben eingesetzt, bei denen größere Dokumentmengen verarbeitet werden müssen. Das Kontingent ist im Google AI Pro-Abonnement enthalten und belastet das Claude-Kontingent nicht.

Codex CLI [7] wird analog über `/codex` eingebunden und ist auf Code-bezogene Aufgaben spezialisiert: Boilerplate-Generierung, Refactoring, Unit-Test-Erstellung, Debugging. Das Kontingent ist im ChatGPT Business-Abonnement enthalten.

Lokales Open-Source-Modell läuft über LM Studio auf einem dedizierten Rechner im lokalen Netzwerk. Die Kommunikation erfolgt über eine OpenAI-kompatible HTTP-API. Dieses Modell übernimmt repetitive, volumenstarke Textarbeit, die kein tiefes Reasoning erfordert. Welches konkrete Modell eingesetzt wird, ist für die Architektur unerheblich und lässt sich ohne strukturelle Änderungen austauschen. Die Schicht

zeichnet sich durch die Abwesenheit von Token-Kosten und Rate-Limits aus. Bei komplexem Reasoning stößt sie an Grenzen, auf die in Abschnitt 7 eingegangen wird.

4.4 Konfigurationssymmetrie

Ein spezifisches Problem beim Einsatz mehrerer Agenten ist die Konsistenz der Arbeitsanweisungen. Wenn der Orchestrator eine Regel kennt, etwa „Produktionsdatenbanken niemals direkt modifizieren", müssen die spezialisierten Agenten diese Regel ebenfalls kennen. Andernfalls entscheidet jeder Agent nach eigenem Ermessen.

Das System löst dieses Problem durch parallele Konfigurationsdateien:

- **CLAUDE.md**: Primäres Regelwerk des Orchestrators. Wird bei jedem Session-Start geladen. Enthält Identitätsinformationen, Infrastrukturbeschreibungen, Delegationsregeln und verbindliche Handlungsanweisungen (Standing Orders).
- **GEMINI.md**: Automatisiert aus `CLAUDE.md` abgeleitet durch ein Migrationsskript (`migrate_to_gemini.py`). Enthält dieselben Kernregeln, angepasst an die Gemini-CLI-Syntax.
- **AGENTS.md**: Pendant für Codex, ebenfalls aus `CLAUDE.md` abgeleitet.

Die 13 Slash-Commands des Orchestrators werden auf 25 Gemini-Skills gespiegelt. Die höhere Zahl auf Gemini-Seite ergibt sich daraus, dass die 8 Claude-Subagenten (spezialisierte Agentendefinitionen für Aufgaben wie Content-Writing, Buchlektor oder Web-Design) als eigenständige Gemini-Skills abgebildet werden.

5. Cross-Agent Memory System

5.1 Das Problem: Agenten improvisieren ohne Kontext

LLM-basierte Agenten haben kein Gedächtnis über Sessions hinweg. Jede neue Session beginnt mit einem leeren Kontext. In einer Multi-Agent-Architektur verschärft sich dieses Problem: Wissen geht nicht nur zwischen Sessions verloren, sondern auch zwischen Agenten. Wenn der Orchestrator in einer Session etwas lernt, etwa dass ein bestimmtes Deployment-Verfahren fehlgeschlagen ist, bleibt diese Information für einen Spezialisten in der nächsten Session unsichtbar.

In beobachteten Situationen ohne persistentes Kontextwissen versuchen Agenten, fehlende Informationen selbständig aus verfügbaren Ressourcen zu rekonstruieren. Das führt zu unkontrolliertem Zugriff auf Systemressourcen, zu ineffizienten Such-Schleifen und zu Ergebnissen, die inhaltlich nicht zuverlässig sind. Dieses Verhaltensmuster ist in produktiven Umgebungen unerwünscht.

Mit einer strukturierten Wissensbasis ändert sich das Verhalten: Der Agent fragt das Memory-System ab und liefert einen kompakten Statusüberblick in kurzer Zeit. Nach Beobachtung des Autors liegt diese Zeit bei unter zehn Sekunden. Diese Angabe ist eine Beobachtungsschätzung, keine Messung.

Der qualitative Unterschied zeigt sich in der Kategorie des Verhaltens. Das Vorhandensein von Memory transformiert das Agentenverhalten von unkontrollierter Improvisation zu strukturiertem Arbeiten. Dieser Sprung ist für das hier beschriebene Szenario der kritische Faktor.

5.2 Architektur des Memory-Systems

Das Memory-System besteht aus drei Komponenten, die auf einer gemeinsamen SQLite-Datenbank im WAL-Modus [8] operieren.

MCP Memory Server (`memory_server.py`): Läuft als Subprocess des Orchestrators und kommuniziert über stdio im JSON-RPC-Format (MCP-Protokoll [5]). Stellt vier Werkzeuge bereit: `remember` (Eintrag speichern), `recall` (Suche), `forget` (Eintrag löschen), `list_memories` (Einträge auflisten). Der Server operiert direkt auf der SQLite-Datenbank.

Cross-Agent CLI (`memory_cli.py`): Ein Kommandozeilen-Werkzeug, das dieselbe Datenbank anspricht. Gemini und Codex rufen es per Shell-Befehl auf: `python memory_cli.py recall "Suchbegriff"`. Die Ausgabe erfolgt als JSON. Dieses Werkzeug übernimmt die Brückenfunktion zwischen den Shell-basierten Agenten und der Persistenzschicht, weil weder Gemini CLI noch Codex CLI eine native MCP-Anbindung haben.

PostToolUse-Hook (`memory_sync.py`): Wenn der Orchestrator eine Markdown-Datei im Memory-Verzeichnis schreibt oder bearbeitet, erkennt dieser Hook das Ereignis, extrahiert die Metadaten aus dem YAML-Frontmatter und führt ein Upsert in die Datenbank durch. Der Mechanismus stellt sicher, dass der dateibasierte Memory-Workflow des Orchestrators automatisch in die Datenbank propagiert wird, ohne manuelle Synchronisation.

Die Duplikaterkennung erfolgt über die ersten 120 Zeichen des Inhalts: Existiert bereits ein Eintrag mit identischem Anfang, wird er aktualisiert statt neu angelegt.

5.3 Die vier Memory-Typen

Die Datenbank kategorisiert Einträge in vier semantische Typen:

Typ	Inhalt	Beispiel
`user`	Identität, Präferenzen, Kommunikationsstil	Nutzungspräferenzen, bevorzugter Kommunikationsstil
`project`	Laufende Projekte, Status, Entscheidungen	Release-Pipeline-Konfiguration, offene Projektpunkte
`feedback`	Gelernte Regeln aus Fehlern und Korrekturen	Auth-Fixes stets mit Nebenwirkungs-Check
`reference`	Pfade, URLs, Verweise auf Zugangsdaten	Produktionsumgebung auf VPS, Zugriff über SSH-Key

Tabelle 2: Memory-Typen mit beispielhafter Charakterisierung.

Die Kategorisierung ist taxonomisch und operational. Sie beeinflusst die Lebensdauer der Einträge: `feedback`-Einträge bleiben in der Regel langfristig gültig, `project`-Einträge veralten schnell und erfordern regelmäßige Aktualisierung. Auf das Problem der Veralterung gehe ich in Abschnitt 7 ein.

5.4 Embedding-Fallback-Kette

Für die semantische Suche werden Einträge idealerweise als Vektoren gespeichert. Da die Verfügbarkeit des Embedding-Modells nicht garantiert ist, implementiert das System eine dreistufige Fallback-Kette:

1. **LM Studio** mit `nomic-embed-text`: Lokales Embedding-Modell über HTTP-API. Liefert die höchste Suchqualität. Funktioniert nur, wenn der lokale Rechner mit aktiver LM Studio-Instanz erreichbar ist.
2. **sentence-transformers** [9] (`all-MiniLM-L6-v2`): Python-Bibliothek für lokale Embedding-Berechnung. Benötigt PyTorch ≥ 2.11.
3. **Text-Substring-Suche**: Einfacher String-Vergleich als letzter Fallback. Funktioniert zuverlässig, liefert aber keine semantische Abstraktion. Ein Suchbegriff wie „Deploy" findet keine Einträge, die „Auslieferung" enthalten.

Im aktuellen Setup sind die oberen Fallback-Stufen nicht permanent verfügbar. Für die Größenordnung des aktuellen Einsatzes ist die Text-Substring-Suche ausreichend, weil Einträge meist eindeutige Schlüsselbegriffe enthalten. Bei deutlich größeren Datenmengen wird die eingeschränkte Suchqualität relevant.

5.5 Gegenhypothese: Reicht ein großes Kontextfenster?

Ein naheliegender Einwand: Moderne LLMs wie Gemini bieten Kontextfenster im Millionen-Token-Bereich. Warum nicht einfach alle relevanten Informationen bei jedem Session-Start in den Kontext laden?

Diese Strategie scheitert an drei Punkten. Erstens ist das Kontextfenster ephemer und existiert nur für die Dauer einer Session. Ein persistentes Memory überlebt Sessions und Agentenwechsel gleichermaßen. Zweitens verfügen nicht alle Agenten über große Kontextfenster: Codex CLI arbeitet mit einem deutlich kleineren Fenster als Gemini. Ein gemeinsames Memory stellt Kontextwissen unabhängig vom individuellen Agenten zur Verfügung. Drittens ist die gezielte Abfrage effizienter als die Kontextüberflutung. Der Befehl `recall "Deploy-Probleme"` liefert präzise selektierte Information; ein 100.000-Token-Kontext verursacht dagegen signal-to-noise-Probleme im Reasoning.

6. Sicherheitsmechanismen

Die in diesem Kapitel beschriebenen Mechanismen sind exemplarisch ausgewählt. Das produktive Setup enthält weitere, die nicht im Detail beschrieben werden. Zielsetzung dieses Kapitels ist die Skizzierung des Sicherheitsansatzes, nicht die vollständige Offenlegung aller Kontrollen.

6.1 Input-Validierung über PreToolUse-Hooks

Ein typisches Beispiel für präventive Qualitätssicherung ist ein PreToolUse-Hook, der vor jedem schreibenden Werkzeugaufruf ausgeführt wird. In der beschriebenen Umgebung prüft ein solcher Hook den geplanten Output gegen definierte Identitätsregeln, etwa gegen die korrekte Schreibweise von Namen oder gegen die Verwendung erfundener Kontaktdaten.

Bei einem Verstoß blockiert der Hook den Werkzeugaufruf und gibt eine strukturierte Fehlermeldung aus. Der Orchestrator erhält die Möglichkeit zur Korrektur, bevor der Befehl erneut ausgeführt wird.

Die grundsätzliche Relevanz dieses Mechanismus liegt im Umgang mit Halluzinationen: LLMs neigen dazu, bei fehlenden Informationen plausibel klingende Daten zu generieren. In Dokumenten, E-Mails oder Publikationen, die unter dem Namen eines Unternehmens ausgehen, entsteht daraus ein rechtliches und reputatives Risiko, das durch automatische Prüfung adressiert werden sollte.

6.2 Build-Gates für Deployment-Artefakte

Für Artefakte, die in produktive Umgebungen ausgeliefert werden, gelten zusätzliche Prüfregeln, die als Build-Gates implementiert sind. Typische Prüfpunkte:

- Keine Entwicklungs-Metadaten in Produktiv-Paketen
- Keine Dokumentations-Dateien, die nicht für das Ziel bestimmt sind
- Keine Umgebungsvariablen oder Secrets in Artefakten
- Auslieferung ausschließlich über validierte Build-Skripte

Solche Gates fangen typische Integrationsfehler ab, bevor sie produktive Auswirkungen haben.

6.3 Lokale Datenhoheit

Das Memory-System speichert alle Daten lokal in einer Datei. Es gibt keine Cloud-Synchronisation und keinen Drittanbieter-Zugriff. Die Memory-Inhalte umfassen Konfigurationsdaten und projektbezogene Notizen, die nicht für externe Dienste bestimmt sind. Das beschriebene Setup verarbeitet bewusst keine Kundendaten und keine personenbezogenen Daten.

7. Evaluierung und Praxisbeobachtungen

7.1 Qualitative Metriken

Die folgenden Beobachtungen basieren auf dem Produktiveinsatz über mehrere Wochen. Sie sind als qualitative Schätzungen zu verstehen, nicht als statistisch belastbare Messungen.

Beobachtung	Ohne Orchestrierung	Mit Orchestrierung	Hinweis
Claude-Limit-Auslastung (Mitte der Woche)	rund 96 %	nicht systematisch gemessen	Schätzung aus Statusline-Beobachtung
Delegationsquote	0 %	geschätzt 40–50 %	Anteil der Aufgaben, die an Gemini, Codex oder das lokale Modell gehen
Memory-Recall	nicht vorhanden	funktionsfähig	Text-Substring-Suche, nicht semantisch
Agentenverhalten ohne Memory	unkontrollierte Improvisation	strukturiertes Arbeiten	qualitativer Unterschied, siehe 5.1

Tabelle 3: Qualitative Beobachtungen im Produktiveinsatz.

7.2 Bekannte Limitationen und grundlegende Schwachstellen

Embedding-Verfügbarkeit. Versionsabhängigkeiten zwischen Bibliotheken können dazu führen, dass semantische Suche nicht permanent verfügbar ist. Der Fallback auf Text-Substring-Suche ist funktional, skaliert aber nicht auf größere Datenmengen.

Plattformspezifische Pfadprobleme. CLI-Werkzeuge verhalten sich beim Pfad-Handling unter verschiedenen Betriebssystemen unterschiedlich. Workarounds mit absoluten Pfaden sind funktional, aber fehleranfällig bei Systemwechseln.

Reasoning-Grenzen lokaler Modelle. Lokale Open-Source-Modelle eignen sich für Textgenerierung nach klaren Mustern. Für Aufgaben mit mehrstufigem Reasoning ist die Qualität nicht ausreichend. Die Aufgabenabgrenzung muss entsprechend scharf erfolgen.

Memory-Drift. Einträge in der Datenbank veralten, wenn sich die zugrundeliegende Realität ändert. Ein automatischer Mechanismus zur Erkennung veralteter Einträge fehlt. Manuelle Pflege ist notwendig.

Skill-Aktivierung im Single-Command-Modus. Bestimmte CLI-Features werden nur im interaktiven Modus aktiviert, nicht bei automatisierter Delegation. Das reduziert die Nutzbarkeit vorhandener Skill-Definitionen.

Nebenwirkungsrisiken bei spezialisierten Code-Agenten. Spezialisierte Agenten neigen dazu, Probleme lokal zu lösen, ohne systemweite Seiteneffekte zu berücksichtigen. Ohne explizite Regeln zur Prüfung auf Seiteneffekte können punktuelle Fixes an einer Stelle stille Folgefehler an anderer Stelle verursachen. Die Gegenmaßnahme besteht in dokumentierten Regeln zur Nebenwirkungs-Prüfung, die als Standing Order im Regelwerk verankert sind.

7.3 Redundanz und geopolitische Abhängigkeit — eine betriebliche Anmerkung

Persönliche Anmerkung des Autors.

Der redundante Ansatz ist in jedem professionellen Szenario sinnvoll, in dem automatische Prozesse oder Leistungserbringung von KI-Systemen abhängig sind. In vielen Unternehmen existieren keine oder kaum manuelle Fallbacks, weil die Automatisierung genau diese manuellen Prozesse ablösen soll. Genau dadurch entsteht das Risiko: Beim Ausfall eines Single-Source-Systems können ganze Prozesse zum Erliegen kommen, und die Schäden sind erheblich.

Für die Unternehmensleitung bedeutet das: Die im Einkauf üblichen Qualitätsstandards der Second- und Third-Source-Strategie gelten auch bei IT-Lieferanten für KI-Dienste. Wer einen Monitoringdienst wie status.claude.com oder vergleichbare Statusseiten der Provider abonniert, bekommt ein belastbares Bild davon, wie häufig diese Dienste ausfallen oder degradiert verfügbar sind.

Hinzu kommt eine weitere Dimension, die in der technischen Diskussion oft ausgeblendet wird. Nahezu alle relevanten Cloud-LLM-Dienste — mit Ausnahme von Mistral und Le Chat aus Frankreich — stammen von amerikanischen Anbietern. Eine einseitige Abhängigkeit von einem einzelnen Land ist strategisch riskant: Spätestens, wenn dieses Land Rechenleistung für eigene Zwecke beansprucht, etwa bei nationalen Notständen oder im Kriegsfall, werden externe Nutzer nachrangig bedient. Zu Beginn des Iran-Konflikts war genau dieser Effekt deutlich spürbar. Ebenso denkbar ist eine bewusste politische oder wirtschaftliche Einflussnahme durch Verknappung der Ressource „Rechenleistung".

Das hier beschriebene Architekturmuster leistet in dieser Hinsicht einen Beitrag zur Risikostreuung: Durch die Einbindung mehrerer Provider und die Integration eines lokalen Open-Source-Modells wird die Abhängigkeit von einem einzelnen Anbieter und letztlich von einer einzelnen Jurisdiktion reduziert. Eine vollständige Unabhängigkeit ist damit nicht erreicht — aber die Ausfallfolgen einzelner Ereignisse werden gedämpft.

7.4 Anmerkungen zur Validität

Interne Validität. Der Autor ist gleichzeitig Entwickler, Betreiber und Evaluator. Diese Personalunion erzeugt ein Bestätigungsbias-Risiko. Unabhängige Evaluierungen liegen nicht vor.

Externe Validität. Das System wurde für einen einzelnen Nutzer mit einem spezifischen Aufgabenprofil entwickelt. Die Übertragbarkeit auf andere Nutzungsszenarien, insbesondere auf Mehrbenutzer-Umgebungen oder andere Fachdomänen, ist nicht untersucht und nicht belegt.

Messvalidität. Alle berichteten Metriken sind Schätzungen auf Basis informeller Beobachtung. Automatisierte Telemetrie-Systeme, die Delegationsquoten, Limit-Auslastung oder Memory-Recall-Qualität systematisch erfassen, existieren im Setup nicht.

8. Einordnung und Vergleich

Kriterium	AutoGen [2]	CrewAI [3]	LangGraph [4]	Vorliegender Ansatz
Typ	Python-Framework	Python-Framework	Python-Framework	Konfigurationsmuster
Multi-Provider	Über Adapter	Über Adapter	Über Adapter	Nativ (verschiedene CLIs)
Persistenz	Nicht integriert	Nicht integriert	Zustandsgraph	SQLite Memory DB
Skalierung	Multi-User möglich	Multi-User möglich	Multi-User möglich	Single-User
Framework-Lock-in	Ja (AutoGen API)	Ja (CrewAI API)	Ja (LangChain)	Nein
Lokale Modelle	Über Konfiguration	Über Konfiguration	Über Konfiguration	Nativ (LM Studio)
Einrichtungsaufwand	Mittel	Mittel	Hoch	Hoch (manuelle Konfiguration)

Tabelle 4: Vergleich mit bestehenden Frameworks.

Der vorliegende Ansatz adressiert ein anderes Szenario als die genannten Frameworks. Im Fokus steht der einzelne Wissensarbeiter, der mehrere LLM-Abonnements nutzt und eine pragmatische, nicht-programmatische Orchestrierung sucht. Wer ein Team koordinieren, komplexe Workflows automatisieren oder eine produktionsfähige API bereitstellen muss, greift zweckmäßiger auf AutoGen, CrewAI oder LangGraph zurück.

9. Fazit und offene Fragen

9.1 Zusammenfassung

Dieser Beitrag dokumentiert drei Dinge. Erstens ein Architekturmuster für die funktionale Spezialisierung mehrerer LLMs in einer professionellen Einzelnutzer-Umgebung. Zweitens ein Cross-Agent Memory System auf SQLite-Basis, das persistentes Kontextwissen sessions- und agentengrenzenübergreifend bereitstellt. Drittens einen Ansatz zur Konfigurationssymmetrie, der die konsistente Arbeitsweise aller beteiligten Agenten sicherstellt.

Die zentrale Erkenntnis liegt in der Rolle der Orchestrierung beim Engpass-Problem. Ein einzelnes Modell adressiert die Limitierung nicht; eine Verteilung über mehrere Ressourcen mit geteiltem Regelwerk tut es. Die im Produktiveinsatz verwendeten LLMs standen sämtlich über bestehende Abonnements bereit. Die Investition lag im Entwurf und in der Pflege der Konfiguration. Zusätzliche Lizenzkosten entstanden nicht.

Die Qualität des Systems ist direkt proportional zur Qualität der Konfiguration. Konfigurationsdateien, Memory-Einträge und Regelwerke veralten. Ein System, das nicht aktiv gepflegt wird, degeneriert.

9.2 Empfehlungen für Praktiker

Für Anwender, die ein ähnliches System aufbauen möchten, ergeben sich drei Empfehlungen.

Beginnen Sie mit dem Memory, bevor Sie die Orchestrierung einrichten. Ein Agent ohne Kontextwissen improvisiert, und Improvisation hat in professionellen Umgebungen Konsequenzen. Die Persistenzschicht bildet das Fundament, auf dem die Delegation erst funktionieren kann.

Dokumentieren Sie Ihre Delegationsregeln explizit. Was nicht aufgeschrieben ist, wird bei jedem Session-Start neu entschieden, und potenziell inkonsistent. Eine Tabelle mit Aufgabentypen und Delegationszielen kostet eine Stunde Arbeit und spart langfristig viel Ärger.

Planen Sie Pflege ein. Konfigurationssymmetrie funktioniert nur, wenn `CLAUDE.md`, `GEMINI.md` und `AGENTS.md` synchron gehalten werden. Ein Migrationsskript hilft, ersetzt aber keinen regelmäßigen manuellen Check auf Konsistenz und Aktualität.

9.3 Offene Fragen

Automatische Delegationsentscheidung. Die aktuelle Delegation basiert auf manuellen Regeln. Ein Klassifikator, der Aufgaben automatisch dem geeigneten Agenten zuordnet, auf Basis von Aufgabentyp, aktueller Limit-Auslastung und historischer Erfolgsquote, wäre ein naheliegender nächster Schritt.

Memory-Konsistenz. Ein Mechanismus zur automatischen Erkennung und Bereinigung veralteter Einträge fehlt. Ansätze aus der Datenbankforschung wie zeitgesteuerte Invalidierung, Konfidenz-Scores und automatisierte Konsistenzprüfungen könnten adaptiert werden.

Embedding-Qualität. Der Fallback auf Text-Substring-Suche schränkt die semantische Suchqualität ein. Eine systematische Messung, wie sich unterschiedliche Embedding-Strategien auf das Agentenverhalten auswirken, steht aus.

Skalierbarkeit. Die Übertragbarkeit des Musters auf Mehrbenutzer-Szenarien ist nicht untersucht. Insbesondere die Frage der Konsistenz des Memory-Systems bei konkurrierenden Schreibzugriffen mehrerer Agenten erfordert weitere Arbeit.

Quellenverzeichnis

[1] Goldratt, E. M. (1990). *Theory of Constraints*. North River Press.

[2] Wu, Q., Bansal, G., Zhang, J., et al. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. Microsoft Research. https://arxiv.org/abs/2308.08155

[3] Moura, J. (2024). CrewAI: Framework for Orchestrating Role-Playing AI Agents. https://github.com/crewAIInc/crewAI

[4] LangChain (2024). LangGraph: Build Stateful Multi-Agent Applications. https://github.com/langchain-ai/langgraph

[5] Anthropic (2024). Model Context Protocol Specification. https://modelcontextprotocol.io/specification

[6] Google (2025). Gemini CLI. https://github.com/google-gemini/gemini-cli

[7] OpenAI (2025). Codex CLI. https://github.com/openai/codex

[8] SQLite Consortium (2025). Write-Ahead Logging. https://www.sqlite.org/wal.html

[9] Reimers, N., & Gurevych, I. (2019). Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. *Proceedings of EMNLP-IJCNLP 2019*, 3982–3992. https://arxiv.org/abs/1908.10084