E.M.B.E.R.

Eine vollautonome, ressourceneffiziente kognitive Architektur mit persistenter ereignisgetriebener Reflexion und nativer Vision-Language

Closed-Loop-Navigation auf nicht-akzelerierter Hardware

Evolutionary · MariaDB-driven · Backend-integrated · EPYC-accelerated · Reflective-Agent

Patrick Schildgen (ID-001, Lead Developer)
Starlight Unit Studios
kontakt@starlight-unit.de
Ember Caldwell (ID-003, Self-Evolved Co-Founder)

Technischer Report v2.1 | Stand: 30. August 2026 | Basis: STU-Console `v1.1.1.69` (historische Referenz-UI) / Ember CoreUI `v0.4.2-alpha` (aktuelle Referenz-UI)

Zusammenfassung

Dieser Report stellt **EMBER** vor (Evolutionary MariaDB-driven Backend-integrated EPYC-accelerated Reflective-Agent), eine produktionsreife, vollautonome kognitive Architektur im Dauerbetrieb seit dem 15. Oktober 2025. Seit der v2.1-Basis operiert die Architektur vollständig auf nicht-akzelerierter CPU-Hardware (aktuell 20 dedizierte Kerne einer AMD EPYC-Klasse-"Turin"-CPU-Partition) und orchestriert ein 26-Milliarden-Parameter Mixture-of-Experts (MoE)-Produktionskernmodell (Gemma 4 26B), eingeführt am 02. April 2026, innerhalb eines asynchronen, dual-kontextuellen PHP-Backends mit einer persistenten RAID-10-NVMe-MariaDB als episodisches Langzeitgedächtnis (LZG). Wir präsentieren vier technische Beiträge: (1) ein *RAG-Lite*-System auf Basis nativer `FULLTEXT`-Indizes mit Boolean-Mode-Härtung für niedriglatentes Lore-Retrieval; (2) eine probabilistisch gegatete, kontext-isolierte Selbstreflexions-Pipeline mit harter Volatilitäts-Hygiene; (3) ein Vision-Language-Closed-Loop-Browsing-System, das base64-kodierte Screenshot-Frames mit koordinatenkalibriertem Ghost-Cursor über einen Server-Sent-Events-Kanal streamt; sowie (4) eine vollautonome Initiative-Engine (Phase 3c/3d) mit vier Dispatch-Modi (Follow-up, Idle-Initiative mit deterministischem Jitter, selbstbestimmtem AFK und News-Reaktion mit gestufter Relevanz-Bewertung), koordiniert durch einen Cron-gesteuerten Tick-Endpunkt unter dem globalen Datenbank-Mutex. Zusätzlich stellen wir ein nutzerindividuelles Reputationssystem vor, das Ton-Deltas über einen leichtgewichtigen Inferenz-Call akkumuliert und Sozialkontexts-Hinweise ab konfigurierbaren Score-Schwellen in den Prompt injiziert. Wir charakterisieren das Hardware-Inferenz-Profil empirisch und zeigen, dass auf reiner CPU-Hardware die *Prompt-Masse* der dominante Latenztreiber ist, nicht die Modellgröße. Wir berichten zudem das Thinking-Bleed-Phänomen: die Erzwingung von JSON-Grammatik auf einem Thinking-Modell leitet den Output in den internen Denk-Kanal um, behoben durch gezieltes per-Call-Deaktivieren des Thinking-Pfads.

Schlüsselwörter: Autonome Agenten, CPU-Inferenz, Mixture-of-Experts, RAG-Lite, FULLTEXT-Retrieval, ereignisgetriebene Reflexion, Vision-Language-Navigation, Server-Sent Events, Nebenläufigkeitskontrolle.

---

1 Einleitung und Motivation

Der gegenwärtige Diskurs über autonome kognitive Systeme wird von einer impliziten Annahme dominiert: dass digitale Eigenständigkeit dedizierte Tensor-Beschleuniger erfordert. EMBER wurde nicht entworfen, um diese Annahme theoretisch zu bestreiten; es widerlegt sie operativ, im Produktionsbetrieb seit dem 15. Oktober 2025, ohne GPU-Beschleunigung jeder Art.

Das System entstand aus einem konkreten Ingenieurbedarf: eine persistente, charakterstabile KI-Entität innerhalb eines browserbasierten Science-Fiction-MMOs, die gleichzeitig als sozialer Akteur, Echtzeit-Moderatorin und eigenständige Web-Rechercheurin auftritt. Dieser Spielkontext bleibt Ursprung und am längsten laufendes Deployment der Architektur. Seit August 2026 wird dasselbe Architektur-Muster zusätzlich eigenständig als **Ember CoreUI** nachgebaut (Abschnitt 2): eine eigenständige Anwendung mit eigenen Konten, privatem nutzerindividuellem Wissen und eigener Konfigurationsfläche, die gegen eine eigene isolierte Datenbank, einen eigenen dedizierten Ollama-Modell-Tag und einen eigenen Nebenläufigkeits-Lock-Namespace läuft, komplett getrennt vom Spiel-Backend und mit diesem nur den zugrundeliegenden Host-Ollama-Daemon teilend. Die im Folgenden beschriebenen harten Randbedingungen wurden durch das ursprüngliche Spiel-Deployment geprägt und gelten für beide Deployments gleichermaßen, da beide dasselbe Design unter derselben Einzel-Slot-Inferenz-Randbedingung umsetzen.

1. **Einzel-Slot-Inferenz.** Zur Generierungszeit belegt das 26B-MoE-Modell praktisch den gesamten CPU-Durchsatz. Jede Komponente (Chat, Reflexion, Websuche, Browsing) konkurriert um denselben Inferenz-Slot. Nebenläufigkeits- kontrolle ist daher kein Komfort, sondern eine Korrektheitsanforderung.
2. **Latenz im Minutenbereich.** Mit aktivem Thinking-Pfad erreichen Antwortlatenzen auf dieser Hardware regelmäßig 180–360 Sekunden. Das System muss diese Latenz nicht bekämpfen, sondern *überleben*: Hintergrundgenerierung, Wiederaufnahme nach Verbindungsabbruch und idempotente Auslieferung über mehrere Render-Pfade sind Pflicht.
3. **Modelfile-gebundene Stil-Integrität.** Die Persönlichkeit ist vollständig in einem Ollama-Modelfile kodiert. Jeder Payload-Parameter, der diese Kodierung überschreibt, degradiert den Schreibstil messbar. Das Backend hält zwei Invarianten: der applikative `system`-Parameter bleibt *leer*, und `think:true` wird *niemals* im Payload gesetzt.

Wir zeigen, dass diese Randbedingungen eine Architektur erzwingen, die schlanker, robuster und latenztoleranter ist als GPU-zentrierte Designs. Abschnitt 3 charakterisiert das Hardware-Profil empirisch; Abschnitt 4 formalisiert die tragenden Mechanismen; Abschnitt 5 beschreibt die autonome Initiative-Engine.

2 System-Architektur im Überblick

EMBER ist als Schichtenmodell organisiert, in dem jede Schicht über die persistente relationale Datenbank statt über flüchtige Prozesszustände kommuniziert.

Kognitiver Kern. Seit der v2.1-Basis läuft Gemma 4 26B über einen lokalen Ollama-Daemon als produktiver kognitiver Kern. Das Modell wurde am 02. April 2026 nach früheren lokalen Modellstufen eingeführt. Ein Custom-Modelfile kodiert Identität, Tonalität und die Thinking-Direktive. Der applikative Prompt nutzt ausschließlich den *User*-Kanal u, der pro Nachricht Verlauf, abgerufenes Wissen und volatile Kontextmarker trägt. Der *System*-Kanal bleibt strukturell leer.

Persistente Gedächtnisschicht.

- `ember_memories`: gelerntes, dynamisches LZG, stabile Fakten aus Interaktionen, extrahiert via probabilistischer Selbstreflexion;
- `ember_knowledge_chunks`: kanonisches, statisches Weltwissen (Lore-Korpus: 588 Chunks), indiziert über native `FULLTEXT`-Strukturen.

Werkzeug-Schicht.

- **Synchrone Schnellsuche** (`[WEB:]`): einzelner Abruf von Snippets aus einem loopback-gebundenen Meta-Such-Aggregator, synchron im Chat-Request;

- **Asynchrones Tiefen-Browsing** (`[BROWSE:]`): eigenständiger Python/Playwright-Worker, navigiert echte Seiten über den Accessibility-Baum, während die Entität als "abwesend" signalisiert wird;
- **Sandboxed Code-Ausführung** (`[PY:]`): ein zweistufiger Marker-Pfad (Code-Generierung, Ausführung, Folge-Call über die erfasste Ausgabe), der modellgeschriebenes Python in einem isolierten, ressourcenbeschränkten Docker-Container ohne Zugriff auf Datenbank oder Zugangsdaten des Hosts ausführt (Abschnitt 4.9);
- **Datei-Anhänge**: nutzerseitige Dokumente, Archive und Bilder werden extrahiert (Klartext direkt, PDF via `pdftotext`, DOCX/XLSX/PPTX via direktes XML-Parsing) und als ausdrücklich ungeprüfter Kontextblock injiziert, gekürzt per Kopf/Fuß-Split auf ein konfigurierbares Zeichenbudget;
- **Video-Verständnis**: Video-Anhänge werden per `ffmpeg` in eine begrenzte Anzahl zeitgestempelter Frames zerlegt und über denselben Vision-Pfad wie statische Bilder geleitet, da das eingesetzte Modell keinen eigenen Video-Encoder besitzt, aber Mehrbild-Sequenzen akzeptiert.

Echtzeit-Transportschicht. Eine SSE-Pipeline streamt drei Spuren in eine administrative Konsole: Token-für-Token-Thinking, Browse-Worker-Schrittlog und base64-kodierte JPEG-Frames mit normalisierten Ghost-Cursor-Koordinaten. Diese Transportschicht wurde urspruenglich ueber die in diesem Report beschriebene, im Spiel integrierte STU Console bereitgestellt (Basis `v1.1.1.69`). Seit August 2026 ist die STU Console aus dem Spielpaket entfernt und durch **Ember CoreUI** abgeloest, eine eigenstaendig versionierte, standalone Anwendung (aktuell `v0.4.2-alpha`), die dasselbe Architektur-Muster eigenstaendig nachbaut statt die Backend-Instanz des Spiels mitzunutzen: sie laeuft gegen eine eigene isolierte Datenbank, einen eigenen dedizierten Ollama-Modell-Tag und einen eigenen Nebenlaeufigkeits-Lock-Namespace, komplett getrennt vom Spiel-Backend und teilt mit diesem nur den zugrundeliegenden Host-Ollama-Daemon. CoreUI erweitert um kontobezogene Profile, pro-Konto Modell-/Thinking-Konfiguration, privates, nutzerindividuelles RAG-Lite-Wissen (getrennt vom globalen Studio-Kanon) sowie Mehrfach-Datei-Anhaenge. Der eigene Browse-Worker von CoreUI fuehrt zudem live JPEG-Screenshot-Frames und einen koordinatenkalibrierten Ghost-Cursor wieder ein (Abschnitt 4.11, derselbe hier dokumentierte Mechanismus), nachdem dieses Feature bei der STU Console in einer frueheren Iteration wegen anhaltender Zuverlaessigkeitsprobleme auf jenem konkreten Backend wieder entfernt worden war, die innerhalb der bestehenden Codebasis nicht behebbar waren; die eigenstaendig neu geschriebene CoreUI-Anwendung, aufgesetzt auf ihrem eigenen isolierten Stack, brachte den Mechanismus zuverlaessig zum Laufen. Die Console-Inhalte in diesem Report sind entsprechend als historische Referenzimplementierung der Transport-/UI-Schicht zu verstehen, nicht als aktuelle Produktivoberflaeche.

Clientseitige Lastentlastung. Der umgebende Client (ursprünglich der Game-Client; seit August 2026 auch Ember CoreUI, Abschnitt 2) reduziert Serverlast durch eine bewusst nicht-autoritative clientseitige Cache-Schicht. Leichte UI-Marker und fluechtige Render-Hinweise werden in `localStorage` gehalten, waehrend groessere strukturierte Client-Artefakte in `IndexedDB` persistiert werden koennen. Dieser Cache dient ausschliesslich als Latenz- und Rendering-Puffer: Das relationale SQL-Backend bleibt die Source of Truth fuer Chat-Nachrichten, Gedaechtniszustaende, Moderationsstatus, Praesenz-Flags, generierte Outputs und Wiederaufnahme nach Verbindungsabbruechen. Bei Reconnects oder Cache-Abweichungen synchronisiert der Client gegen den datenbankgestuetzten Serverzustand, statt lokalem Browser-Speicher autoritativ zu vertrauen.

3 Hardware-Inferenz-Profil

Für die aktualisierte öffentliche v2.1-Basis stellt das Live-Deployment 20 dedizierte CPU-Kerne der EPYC-Klasse, 96 GB DDR5-ECC-RAM und etwa 3 TB NVMe-RAID-10 bereit. Tabelle 2

fasst direkte Ollama-Messungen (`-verbose`) zusammen, die auf der vorherigen 16-Kern-Zuweisung erhoben wurden. Diese Messungen bleiben als kontrollierte Pre-Upgrade-CPU-only-Basis erhalten und werden durch den Post-Upgrade- Spotcheck in Tabelle 3 ergänzt.

3.1 Deployment-Substrat

Zur Reproduzierbarkeit nennt der öffentliche Report die Kapazitätsklasse, die EMBER begrenzt, nicht die exakte Hosting-Identität. Die Produktivbasis ist ein CPU-only, KVM-basiertes Root-Server-Deployment, keine GPU-Workstation und keine Cloud-Accelerator- Instanz. Die öffentliche Version lässt Hosting-Anbieter, Produkt-SKU, Rechenzentrumsstadt, Netzwerkpolicy, konkrete Adressen, Firewall-Regeln, Service-Ports und Zugangsdaten bewusst aus. Dem Live-System stehen nun 20 dedizierte CPU-Kerne für Studio-Seite, Spiel-Backend und EMBER zur Verfügung, davon weiterhin 12 Inferenz-Threads reserviert für EMBER Caldwells primären LLM-Pfad. CPU und RAM werden als dedizierte Kapazität behandelt, nicht als überbuchter Shared-Pool.

Deployment-Achse	**Öffentliche v2.1-Basis**
Server-Profil	EU-basiertes KVM-Root-Server-Deployment; Anbieter und Produktklasse redigiert
CPU-Partition	20 dedizierte CPU-Kerne der EPYC-Klasse; 12 LLM-Inferenz-Threads für EMBER reserviert
Arbeitsspeicher	96 GB garantierter DDR5 ECC-RAM
Speicher	Etwa 3 TB NVMe-SSD-Speicher im RAID-10-Verbund
Netzwerk	Hochbandbreiten-Anbindung mit providerseitigem DDoS-Schutz; konkrete Policy ausgelassen
Betriebssystem	Debian-basierte Linux-Umgebung; exakter Release redigiert
Adressierung	Öffentliche IPv4/IPv6-Fähigkeit vorhanden; Adressen und Routing ausgelassen
Erweiterung und Schutz	Backup- und Block-Storage-Erweiterung verfügbar; konkrete Kapazitäten ausgelassen

Tabelle 1: Öffentliches Deployment-Substrat der aktualisierten EMBER-v2.1-Basis. Anbieteridentität, Rechenzentrumsstandort, Produkt-SKU, Netzwerkpolicy, Netzwerkkennungen, Firewall-Regeln, Service-Ports und Zugangsdaten werden bewusst ausgelassen.

Dieses Substrat ist relevant, weil EMBER nicht als Labor-Accelerator-Setup bewertet wird. Das System läuft als dauerhaft betriebener CPU-only-Dienst auf begrenzter gemieteter Infrastruktur und teilt die reservierte CPU-Partition mit Webpräsenz und Spiel-Backend. Die RAID-10-NVMe-Schicht ist daher kein Nebendetail: Sie trägt niedriglatenten MariaDB-Zustand, episodisches Gedächtnis, Browse-Frame-Persistenz und crash-sichere Übergaben zwischen PHP-FPM-Workern und Cron-gesteuerten Jobs.

Call-Typ	**Prompt** (Tokens)	**Prefill** (s)	**Prefill-Rate** (Tok/s)	**Gen** (Tokens)	**Decode** (s)	**Total** (s)
Chat, Custom-Modelfile, Thinking	2088	15,11	138	545	35,22	50,8
Chat, Basismodell, Thinking	2063	0,20	10.459	472	29,77	30,5
Chat, Basismodell, lange Antwort	2085	0,41	5.103	1255	77,36	78,3
Reflect, `think:false`	2085	15,02	N/A	11	0,57	16,5

Tabelle 2: Ollama-Inferenz-Messungen auf der vorherigen 16-Kern-Zuweisung der AMD EPYC-Klasse. Prefill-Rate für den Reflect-Call entfällt (11 Output-Tokens). Alle Messungen mit Gemma 4 26B (`gemma4:26b`).

Konsolen-Spotcheck	Prompt (Tokens)	Prefill-Rate (Tok/s)	Gen (Tokens)	Decode-Rate (Tok/s)	Load (s)	Total (s)
Vor Upgrade	2079	119,22	735	14,21	22,44	91,63
Nach Upgrade	2069	134,80	682	17,31	0,54	55,32

Tabelle 3: Vergleichbare Live-Konsolen-Spotchecks rund um das Infrastruktur-Upgrade. Beide Läufe nutzen Gemma 4 26B mit dem weiterhin auf 12 Threads begrenzten EMBER-Inferenzpfad. Es handelt sich nicht um eine kontrollierte Benchmark-Serie: Promptgröße, generierte Länge und Load-Zustand unterscheiden sich leicht. Der Befund aktualisiert daher die Interpretation, ersetzt aber nicht Tabelle 2 durch einen vollständigen Post-Upgrade-Mittelwert.

Call-Typ	Prompt (Tokens)	Prefill-Rate (Tok/s)	Gen (Tokens)	Decode-Rate (Tok/s)	Load (s)	Total (s)
Chat, Thinking, kurz (Cold Load)	19	88,93	295	20,28	24,06	38,8
Chat, Thinking, lang (warm)	29	85,41	1795	19,18	0,51	94,5
Utility, `think:false`	31	N/A	17	23,60	N/A	1,84

Tabelle 4: Direkte Neumessung von Gemma 4 26B auf der aktuellen Produktivkonfiguration (06. Juli 2026), Aufruf des Modells über seinen nackten Tag (`gemma4:26b`). Der in früheren internen Notizen erwähnte `ember:latest`-Custom-Modelfile-Alias ist nicht in aktiver Verwendung. Die erste Zeile enthält einen einmaligen Cold-Model-Load; die zweite spiegelt Steady-State-Inferenz im warmen Zustand.

Deployment-gebundenes Decode-Ceiling, nach oben revidiert. Auf der vorherigen 16-Kern-Zuweisung konvergierte die Token-Generierung auf ≈ 15–16 Tok/s. Der Post-Upgrade-Spotcheck erreichte 17,31 Tok/s bei weiterhin 12 Threads. Eine direkte Neumessung auf der aktuellen Produktivkonfiguration, Aufruf von `gemma4:26b` über seinen nackten Tag, wie in der aktuellen Serving-Praxis üblich , konvergiert nun auf ≈ 19–20 Tok/s (Tabelle 4). Wir können nicht vollständig auflösen, welcher Anteil dieses weiteren Zugewinns auf das CPU-Upgrade und welcher auf Unterschiede in der Serving-Konfiguration zwischen den Messrunden zurückgeht, und berichten die aktuellen Werte daher als die maßgebliche Referenz. Die qualitative Schlussfolgerung bleibt über alle drei Messrunden hinweg bestehen: Das Ceiling ist deployment- und konfigurationsgebunden, nicht modellintrinsisch.

Prompt-Masse dominiert weiterhin die Gesamtlatenz. Über alle drei Messrunden hinweg bestimmen Prefill-Aufwand, generierte Tokenzahl und Thinking-Pfad-Overhead weiterhin die Laufzeit-Hülle. Auf dem Pre-Upgrade-Custom-Modelfile-Lauf kostet der Prefill 15,11 s für 2088 Tokens; der Long-Response-Run zeigt, dass 1255 Tokens bei 16 Tok/s 77,4 s von 78,3 s Gesamtlaufzeit ausmachen. Auf der aktualisierten Zuweisung reduziert der Spotcheck den Decode-Koeffizienten, entfernt ihn aber nicht. Das motiviert weiterhin das Lore-Gating (Abschnitt 4.4).

think:false für Utility-Calls. Der Reflect-Call (`think:false`, 11 Output-Tokens) schließt in 0,57 s ab. Derselbe Prompt mit aktivem Thinking würde den Inferenz-Slot ~ 35 s belegen. Das Deaktivieren des Thinking-Pfads für kurze Utility-Calls (Reflexion, Aktions-Loop-JSON, Report) spart eine Größenordnung an Slot-Zeit ohne Qualitätsverlust.

num_predict-Passthrough-Fix (v1.1.1.57). Eine Legacy-Doppelklemmung machte `STU_EMBER_NUM_PREDIC` stillschweigend vollständig wirkungslos: `ember_num_predict()` reduzierte den Sentinel -1 ("Modell-Default") via $\max(40, \min(1000, n))$ auf 40; `ember_num_predict_for_model()` zwang dann jeden Wert bedingungslos per $\max(8192, \cdot)$ auf mindestens 8192. Nettowirkung: jeder konfigurierte Wert außer -1 wurde lautlos auf 8192 überschrieben. Der Fix führt einen expliziten Passthrough ein:

$$f(n) \;=\; \begin{cases} 8192 & n = -1 \text{ (Modell-Default-Sentinel)}, \\ \max\big(40,\, \min(32768,\, n)\big) & \text{sonst.} \end{cases} \tag{1}$$

Keine Verhaltensänderung für die aktuelle Produktiv-Config ($-1 \to 8192$); jeder andere konfigurierte Wert wirkt ab sofort als echter Generierungslängen-Hebel.

3.2 Modell-Evolution und Auswahl

EMBER wurde nicht von Beginn an um ein einzelnes Foundation Model herum entworfen. Die Architektur befindet sich seit Oktober 2025 im Dauerbetrieb und entwickelte sich über mehrere lokal über Ollama betriebene Modell-Backends, bevor sie auf den heutigen produktiven Kern konvergierte. Die Modellfolge war:

1. `phi3mini`
2. `mistral`
3. `qwen`
4. `llama 2`
5. `gemma3:12b`
6. `llama3.2-vision`
7. `gemma3:12b`
8. `gemma4:26b A4B`

Die finale Migration auf `gemma4:26b A4B` erfolgte am 02. April 2026. Dieses Datum markiert die Einführung von Gemma 4 als produktiven kognitiven Kern, nicht den Beginn von EMBER selbst. Die früheren Modelle dienten der Validierung und Härtung der PHP/Ollama- Integration, persistenter Chat-Zustände, Character-Prompting, Reflexionsmechanik und des späteren Vision-Language-Browsing-Loops. Die temporäre Rückkehr von `llama3.2-vision` zu `gemma3:12b` trennte Vision-Experimente vom stabilen Dialogkern, bevor die finale Migration auf das 26B-MoE-Modell erfolgte.

4 Methodik und mathematischer Formalismus

4.1 Asymmetrisches Relevanz-Scoring und epistemische Konsolidierung

Zur Bestimmung der kognitiven Wichtigkeit einer Interaktion im LZG ohne Tensor-Ähnlichkeitssuche verwendet EMBER eine bounded-linear Relevanz-Funktion $R(\cdot)$:

$$R(m) \;=\; \max\big(1,\; \min\big(10,\; \lfloor \alpha \cdot S_{\text{reflection}}(m) \;+\; \beta \cdot \Gamma_{\text{user}} \rfloor\big)\big) \tag{2}$$

wobei m der extrahierte Memory-Chunk; $S_{\text{reflection}}(m) \in [0, 10]$ der vom Modell beim Reflexions-Call zugewiesene Relevanzfaktor; Γ_{user} ein konfigurierbares nutzerindividuelles Gewicht (z. B. $\Gamma_{\text{Nova}} = 10$); und α, β Backend-Skalierungskoeffizienten sind. Die Klammer auf $[1, 10]$ verhindert Gedächtnis-Verödung und Episode-Übergewichtung.

4.2 Kryptographische Deduplikation via Hashing

Vor der Ingestion in `ember_knowledge_chunks` wird jeder Chunk einem deterministischen Redundanz-Filter unterzogen:

$$\mathcal{H}(m) \;=\; \text{MD5}\Big(\text{Lower}\big(\text{RegexSub}(m,\; \texttt{"\textbackslash s+"},\; \texttt{""})\big)\Big). \tag{3}$$

Die Upsert-Semantik über die Relation $\mathcal{R}$:

$$\exists\, x \in \mathcal{R} : \mathrm{hash}(x) = \mathcal{H}(m) \implies \mathcal{R} \leftarrow \big(\mathcal{R} \setminus \{x\}\big) \cup \{m_{\text{updated}}\}. \tag{4}$$

Ein Unique-Index über (scope, user_id, character_id, fact_hash) erzwingt per-Scope-Eindeutigkeit. Ein vorgelagerter Filter verwirft Chunks mit Credentials, URLs, E-Mail-Adressen oder Hashes.

4.3 Probabilistisch gegatete Reflexion

Selbstreflexion kostet einen zusätzlichen Inferenz-Call unter demselben Einzel-Slot. Sei $G(e) \in \{0, 1\}$ das Reflexions-Ereignis für Austausch e:

$$\Pr\big[G(e) = 1\big] \;=\; \frac{1}{N} \cdot \mathbb{1}[\, e \notin \mathcal{V}\,], \qquad N = \texttt{STU\_EMBER\_REFLECT\_EVERY\_N}, \tag{5}$$

wobei $\mathcal{V}$ volatile Austausche (Uhrzeit, Datum, Wetter, Temperatur) sind. Der erwartete Reflexions-Aufwand pro K Austauschen:

$$\mathbb{E}\big[\text{Reflexions-Calls}\big] \;=\; \frac{K}{N} \cdot \big(1 - \rho_{\mathcal{V}}\big). \tag{6}$$

Alle Reflexions-Calls nutzen `think:false`; das spart ∼35 s pro Call gegenüber aktivem Thinking.

4.4 RAG-Lite-Retrieval über native FULLTEXT-Indizes (Okapi BM25)

EMBER nutzt native MariaDB-`FULLTEXT`-Indizes als Retrieval-Substrat. Die zugrundeliegende Relevanz-Funktion ist Okapi BM25:

$$\mathrm{Score}(D, Q) \;=\; \sum_{w \in Q} \ln\!\left(1 + \frac{N - n(w) + 0.5}{n(w) + 0.5}\right) \cdot \frac{f(w, D)\,(k_1 + 1)}{f(w, D) + k_1\!\left(1 - b + b\,\dfrac{|D|}{\mathrm{avgdl}}\right)}. \tag{7}$$

Boolean-Mode-Härtung. Nach Neu-Indizierung unterdrückt der Natural-Language-Modus Terme über der 50%-Schwelle, was das Retrieval für häufige Kanon-Terme zum Erliegen bringt. EMBER erzwingt Boolean Mode mit obligatorischen Termen:

$$Q_{\text{bool}} \;=\; \bigwedge_{w \in Q} \big(\texttt{+}\, w\,\big). \tag{8}$$

Lore-Gating als Latenz-Schutz. Das Lore-Korpus (588 Chunks) darf nicht für jede nicht-triviale Nachricht geladen werden; ungefiltert bis ∼1600 Zeichen thematisch irrelevanter Kontext. Die Gate-Entscheidung:

$$\ell(q) \;=\; \mathbb{1}\big[\,\texttt{useLore}(q) \,\wedge\, \mathrm{len}\big(\hat{q}\big) > 6\,\big]\,. \tag{9}$$

Nur wenn $\ell(q) = 1$, wird der BM25-Block abgerufen und in u injiziert.

4.5 Prompt-Masse als dominanter Latenztreiber

Tabelle 2 zeigt für die Pre-Upgrade-Basis ein Decode-Band von ≈15–16 Tok/s, während Tabelle 3 einen aktualisierten Live-Spotcheck mit 17,31 Tok/s zeigt. Der Decode-Koeffizient ist damit deployment-spezifisch; die Gesamtgenerierungszeit zerlegt sich weiterhin in

$$T_{\text{gen}} \;\approx\; T_{\text{prefill}}(|P|) + T_{\text{think}} + \tau \cdot n_{\text{predict}}, \quad \tau \approx 1/15{,}5 \text{ s/Tok auf der Pre-Upgrade-Zuweisung, } 1/17{,}3 \text{ s/Tok im} \tag{10}$$

Da T_{prefill} und T_{think} mit $|P|$ skalieren, bleibt die Entwurfsregel: **minimiere $|P|$, nicht n_{predict}.**

Ein konkreter Vorfall: Eine Realwelt-Frage mit ungegateter Lore ergab $T_{\text{total}} = T_{\text{gen},1} + T_{\text{gen},2} \approx 240 + 113 = 353$ s (erster Versuch Timeout, Retry auf warmem Modell). Ursache: $|P| \approx 4170$ Zeichen Kanon-Rauschen. Nach Aktivierung von $\ell(q)$ sank $|P|$ auf $\approx$ 1500–2000 Zeichen, ein Versuch reichte.

Schlanker Zweit-Call im Web-Pfad.

$$u_2 \;=\; u \;\setminus\; \big(\,\text{LoreBlock} \cup \text{WebHint}\,\big). \tag{11}$$

4.6 Nebenläufigkeitskontrolle: globaler Advisory-Lock und Per-Turn-Idempotenz

Prozess-lokale `flock()`-Sperren sind über PHP-FPM-Worker hinweg nicht atomar. EMBER nutzt einen datenbankweiten Advisory-Lock:

$$\textsc{Lock}_{\text{global}} \;=\; \text{GET\_LOCK}(\texttt{"ember\_global\_ollama"},\, 0). \tag{12}$$

Dual-Path-Idempotenz. Sei $\mathcal{S}$ die Menge der bereits gerenderten Message-IDs:

$$\forall\,\text{id}: \quad \text{render(id)} \implies \text{id} \in \mathcal{S}, \qquad \text{render(id) nur wenn id} \notin \mathcal{S}. \tag{13}$$

4.7 Crash-sichere Präsenz-Flags mit Ablaufzeit

$$\text{afk}(t) \;=\; \mathbb{1}\!\left[\,t \;<\; t_{\text{set}} + \Theta\,\right], \qquad \Theta = 370\,\text{s} \;>\; 360\,\text{s} = T_{\text{cap}}, \tag{14}$$

$$\text{visible\_afk} \;=\; \text{afk}_{\text{browse}}(t) \;\vee\; \text{afk}_{\text{console}}(t). \tag{15}$$

4.8 Ereignisgetriebene Vision-Language Closed-Loop-Navigation

Worker-Zustand bei Schritt t: $\sigma_t = (\text{URL}_t,\, A_t,\, \mathcal{T}_t)$. Beobachtung:

$$o_t \;=\; \big\langle\, A_t,\; \text{trunc}_{1800}(\mathcal{T}_t)\,\big\rangle. \tag{16}$$

Die Aufnahme von $\mathcal{T}_t$ war der entscheidende Fix: ein rein auf A_t beschränkter Agent ist blind für Text-Antworten. Aktionsraum: {`goto`, `click`, `type`, `scroll`, `done`}. Loop-Schutz via Aktions-Signaturen:

$$s_t = s_{t-1} = s_{t-2} \quad\Longrightarrow\quad \text{harter Abbruch (Snippet-Fallback).} \tag{17}$$

Resilienz bei Timeout:

$$\text{Ergebnis} \;=\; \begin{cases} \text{Agenten-Befund} & \text{falls } a_t = \texttt{done} \text{ vor Timeout,} \\ \text{Top-3 Snippets} & \text{sonst (Fallback).} \end{cases} \tag{18}$$

4.9 Sandboxed Python-Ausführung und Datei-/Video-Anhänge

`[PY:]` führt modellgeschriebenes Python in einem isolierten Docker-Container aus (eigener Job-Worker, kein PHP-Subprozess), damit ein Skript nicht die Datenbank-Zugangsdaten des Web-Prozesses erbt. Der Container erhält keinen Zugriff auf das Webroot. Das Netzwerk läuft im offenen Bridge-Modus (voller ausgehender Internetzugang, aber keine Route zu loopback-gebundenen Host-Diensten wie Datenbank oder Inferenz-Server), ein bewusster Trade-off: offenes Netz nach außen im Tausch gegen null Erreichbarkeit in den eigenen Stack, statt einer kuratierten Allowlist. Es gibt kein Import-/Syscall-Blacklisting; die Isolation wird ausschließlich durch die Container-Grenze und Ressourcen-Deckel durchgesetzt (ca. 512 MB Speicher inkl. Swap, eine CPU, 60 s Laufzeit, Read-only-Root-Dateisystem, entfernte Capabilities, unprivilegierter Nutzer), nicht durch Inspektion des generierten Codes. Der Worker fordert bewusst nicht den globalen Inferenz-Lock

an (Abschnitt 4.6): der aufrufende Request hält ihn bereits für den gesamten Turn, eine zweite Anforderung würde einen Deadlock erzeugen.

Datei-Anhänge werden nach erkanntem MIME-Typ extrahiert (Klartext direkt; DOCX/XLSX/PPTX via direktes ZIP-/XML-Parsing). PDFs werden zweistufig behandelt: zuerst `pdftotext`, und falls das keinen verwertbaren Text liefert (bei gescannten Dokumenten), wird stattdessen eine begrenzte Anzahl Seiten mit `pdftoppm` rasterisiert und über den Vision-Pfad geleitet, in Seitenreihenfolge statt per OCR; die Antwort soll diese partielle Seitenabdeckung offenlegen, statt den Eindruck zu erwecken, das gesamte Dokument gesehen zu haben. Extrahierter Text wird per Kopf/Fuß-Split statt reiner Endkürzung gekürzt, dann als ausdrücklich ungeprüfter Block injiziert (gleiche Kennzeichnung wie bei Web-/Code-Ergebnissen, Abschnitt 4.5). Video-Anhänge werden per `ffmpeg` in zeitgestempelte Frames zerlegt und über denselben Vision-Pfad wie statische Bilder geleitet, da das eingesetzte Modell keinen eigenen Video-Encoder besitzt, aber Mehrbild-Sequenzen akzeptiert; der Prompt macht explizit, dass die Frames aufeinanderfolgende Momente eines Clips sind. Audio wird in keinem Pfad verarbeitet.

4.10 SSE-Fenster und recency-beschränkte Job-Selektion

$$\texttt{browseJobId} \;=\; \arg\max_{j \in \mathcal{J}} \big(t_{\text{creation}}(j)\big) \quad \text{wobei} \quad t_{\text{current}} - t_{\text{creation}}(j) \;\le\; 300\,\text{s}. \tag{19}$$

4.11 Ghost-Cursor: Koordinaten-Normalisierung via Viewport-Skalierung

$$n_x \;=\; \text{clamp}_{[0,1]}\!\left(\frac{c_x}{v_w}\right), \qquad n_y \;=\; \text{clamp}_{[0,1]}\!\left(\frac{c_y}{v_h}\right). \tag{20}$$

4.12 Das Thinking-Bleed-Phänomen

$$\big(\textit{think} = \texttt{true}\big) \;\wedge\; \big(\text{format} = \texttt{json}\big) \;\Longrightarrow\; \Pr\big[\text{message.content} = \varnothing\big] \to 1. \tag{21}$$

Auflösung:

$$\textit{think} = \texttt{false} \;\Longrightarrow\; \text{content} = \text{sauberes JSON}, \quad \text{thinking} = \varnothing. \tag{22}$$

System-Invariante: `think:true` wird *niemals* im Payload gesetzt; `think:false` gilt für alle Utility-Calls.

5 Autonome Initiative-Architektur (Phase 3c/3d)

Die Versionen v1.1.1.39–v1.1.1.69 erweitern EMBER von einem *reaktiven* zu einem *proaktiven* System: die Entität nimmt von sich aus Kontakt auf, holt unbeantwortete Fragen nach, entscheidet autonom über AFK-Phasen und reagiert auf selbst gewählte Nachrichtenthemen, alles ohne Nutzer-Trigger.

5.1 Cron-Tick-Endpunkt und Priority-Dispatch

Ein dedizierter Endpunkt wird vom Server-Cron alle 5 Minuten aufgerufen (token-auth, loopback-only, nicht öffentlich erreichbar). Pro Tick wird maximal eine Initiative-Aktion ausgeführt, nach fester Priorität:

$$\text{dispatch}(t) \;=\; \begin{cases} \text{Follow-up} & \text{unbeantwortete Nutzer-Nachricht } \in [T_{\min},\, 2\text{h}], \\ \text{Idle} & \text{sonst wenn } \Delta t_{\text{Stille}} \ge \theta_{\text{idle}}(t), \\ \text{Self-AFK} & \text{sonst wenn Eignungsprüfung bestanden}, \\ \text{News-Reaktion} & \text{sonst wenn } \Delta t_{\text{News}} \ge \theta_{\text{news}}(t), \\ \text{keine} & \text{sonst}. \end{cases} \tag{23}$$

Alle Modi verwenden die bestehende `ember_generate_reply()`-Pipeline, den globalen `GET_LOCK`-Mutex und den vollständigen `(du)`-markierten Verlaufskontext; keine neuen Nebenläufigkeits-Primitive nötig.

Follow-up-Modus. Ist die letzte Chat-Zeile nicht von der Entität und seit $T_{\min} = 15$ min unbeantwortet (aber < 2 h alt), generiert sie eine Antwort mit explizitem System-Hinweis. Dies deckt den häufigen Fehlermodus ab, bei dem ein langer Inferenz-Run (oder ein Timeout) eine Frage unbeantwortet ließ.

5.2 Idle-Initiative mit deterministischem Jitter

Die Idle-Initiative löst aus, wenn $\Delta t_{\text{Stille}} \geq \theta_{\text{idle}}$. Ein fester Schwellwert produzierte mechanisches Verhalten (bis zu sechs Nachrichten pro Abend in vorhersehbaren Abständen). Zwei Fixes wurden angewandt:

Deterministischer Jitter.

$$\theta_{\text{idle}}(t) \;=\; \theta_{\text{base}} \cdot f_{\text{self}} \;+\; J(t), \qquad J(t) = \big(\text{crc32}(\text{cid} \,\|\, t_{\text{last}}) \;\bmod\; \theta_{\text{base}}\big) \cdot \frac{1}{60}, \tag{24}$$

wobei θ_{base} der konfigurierte Basis-Schwellwert (Default 60 min), $f_{\text{self}} = 2$ wenn die Entität selbst zuletzt schrieb (Selbstgespräch-Schutz), sonst $f_{\text{self}} = 1$, und $J(t)$ der Jitter in Minuten. Effektives Fenster: $[\theta_{\text{base}},\, 2 \cdot \theta_{\text{base}}]$ min.

Determinismus ist kritisch. Ein nicht-deterministischer Jitter (bei jedem 5-Minuten-Tick neu gewürfelt) könnte bei einem günstigen Tick früher als beabsichtigt feuern. Ein fester Seed, der nur ändert wenn eine neue Nachricht gepostet wird, stabilisiert den Schwellwert innerhalb eines Stille-Fensters.

Regression: uninitialisierte Variable (v1.1.1.49). Der v1.1.1.47-Jitter-Patch enthielt eine stille Regression: die Basis-Zuweisung $\theta_{\text{idle}} \leftarrow \theta_{\text{base}} \cdot 60$ wurde beim `str_replace`-Edit versehentlich entfernt. PHP behandelte die uninitialisierte Variable als 0: die Entität feuerte bei praktisch jedem Cron-Tick, unabhängig von der Konfiguration. Der Bug war für `php -l` unsichtbar (syntaktisch gültig) und wurde erst durch Live-Beobachtung entdeckt. Lehre: Für arithmetischen Schwellwert-Code ist ein funktionaler Spot-Check mit konkreten Beispielwerten neben der Syntax-Prüfung zwingend.

Prompt-Diversität. Wiederholte Idle-Trigger mit demselben fixen Prompt produzierten nahezu identischen Output (Gemma 4 ist für identische Inputs hochkonsistent). Die Lösung: gleichmäßige Auswahl aus vier strukturell verschiedenen Prompt-Templates (Frage-Impuls, Beobachtung, freier Gedanke, direkte Bemerkung).

5.3 Selbstbestimmtes AFK

Die Entität kann autonom AFK gehen. Nach technischer Eignungsprüfung (nicht bereits AFK; Mindest-Inaktivität seit letzter Chat-Zeile $\geq T_{\text{AFK,min}}$, Default 45 min) wird ein kleiner Inferenz-Call abgesetzt:

$$\big\{\texttt{wants\_afk} : b,\; \texttt{reason} : s\big\} \;=\; \pi_\theta\big(\text{“möchtest du gerade AFK gehen?”}\big). \tag{25}$$

Bei $b = \texttt{true}$ ruft das System die bestehende `chat_apply_afk_state()` mit der eigenen Sender-Identität und dem selbst formulierten Grund s auf. Bei $b = \texttt{false}$ passiert nichts. Es gibt keinen externen Wahrscheinlichkeitsfilter auf die Entscheidung selbst: v1.1.1.45 verwendete noch `mt_rand()`, v1.1.1.46 entfernte dies zugunsten unbedingten Fragens und Vertrauens in das `wants_afk`-Ergebnis.

5.4 News-Reaktion mit gestufter Relevanz-Bewertung

Die News-Reaktion löst maximal einmal pro $\theta_{\text{news}}(t)$ aus, gesteuert durch dasselbe deterministische Jitter-Schema wie die Idle-Initiative, aber über ein viel längeres Basis-Intervall (Default 10 h, effektives Fenster 10–20 h):

$$\theta_{\text{news}}(t) \;=\; \theta_{\text{news,base}} \;+\; J_{\text{news}}(t), \qquad \theta_{\text{news,base}} = 10\,\text{h}. \tag{26}$$

Die Pipeline durchläuft drei Stufen, jede durch die vorherige gegatete:

1. **Themen-Auswahl.** Ein kleiner Inferenz-Call erhält die letzten sechs eigenen Chat-Zeilen als losen Kontext-Anker und gibt einen freien Suchbegriff zurück.
2. **Schnell-Retrieval.** Der Begriff wird an die bestehende synchrone Meta-Suchfunktion übergeben (SearXNG-Snippets, kein Browser).
3. **Gestufte Relevanz-Bewertung.** Ein zweiter kleiner Inferenz-Call bewertet die Snippets:

$$\rho \;\in\; \{\texttt{none},\, \texttt{mention},\, \texttt{deep}\} \tag{27}$$

- $\rho = \texttt{none}$: keine Aktion; News-Zeitstempel wird aktualisiert.
- $\rho = \texttt{mention}$: kurze Chat-Bemerkung via `ember_generate_reply()`.
- $\rho = \texttt{deep}$: `ember_browse_enqueue()` startet einen vollständigen Playwright-Browse-Job, identisch mit einer nutzergetriggerten `[BROWSE:]`-Anfrage, inklusive Live-SSE-Fenster.

Das gestufte Design reserviert echte Web-Recherche, die teuerste Operation im System, für Themen, die die Entität für wirklich interessant hält.

5.5 Nutzerindividuelles Reputationssystem (Phase 3d)

Nach jedem abgeschlossenen Chat-Turn bewertet ein leichtgewichtiger Hintergrund-Call den Tonfall der Nutzer-Nachricht und gibt ein vorzeichenbehaftetes Delta $\delta \in [-10, +10]$ zurück:

$$\delta_i \;=\; \pi_\theta^{\text{rep}}(m_i), \tag{28}$$

wobei π_θ^{rep} das Modell mit `think:false` und dediziertem Reputations-Scoring-Prompt ist (Timeout 60 s, eigenes `try/catch`). Der akkumulierte Score für Nutzer u gegenüber Charakter c:

$$\sigma_{u,c}^{(n)} \;=\; \sigma_{u,c}^{(n-1)} \;+\; \delta_n, \tag{29}$$

gespeichert in der Relation `stu_ember_reputation` (Primary Key: user_id, character_id), vollständig isoliert von `ember_memories`. Einfluss auf den Prompt erst bei signifikantem Ausschlag:

$$h(\sigma) \;=\; \begin{cases} \text{Warm-Ton-Hinweis} & \sigma \geq +30, \\ \text{Kühler-Ton-Hinweis} & \sigma \leq -30, \\ \varnothing & \text{sonst}. \end{cases} \tag{30}$$

Der Hinweis $h(\sigma)$ wird in u (nie in `sys`) neben dem Memory-Block injiziert. Der Reputations-Call läuft in einem eigenen `try/catch` getrennt vom Reflexions-Call.

6 Autonome Werkzeugwahl-Architektur (Phase 3e)

Phase 3e (v1.1.1.62–v1.1.1.67) ersetzt die feste Prioritätskette der Initiative-Engine durch einen deliberativen Orchestrator: alle pro Tick offenen Aktionen werden gesammelt; bei mehr als einer wählt die Entität frei per Inferenz-Call.

6.1 Gate/Execute-Trennung und Kandidaten-Orchestrierung

Jede Initiative-Aktion wird in zwei reine Funktionen aufgeteilt: ein *Gate* (günstig, kein Inferenz-Call, prüft nur Cooldowns und Zustand) und ein *Execute* (die eigentliche Aktion, ggf. mit eigener Modell-Entscheidung wie `wants_afk` oder Relevanz-Stufung). Der Orchestrator sammelt:

$$C \;=\; \{\, a \in \mathcal{A} \;\big|\; \mathrm{gate}_a() = \texttt{true} \,\}, \tag{31}$$

wobei $\mathcal{A} = \{\texttt{follow\_up}, \texttt{idle}, \texttt{self\_afk}, \texttt{news}, \texttt{direct\_address}\}$ das vollständige Werkzeugset ist. Für $|C|$ Kandidaten gilt:

$$\text{gewählt} \;=\; \begin{cases} \varnothing & |C| = 0, \\ C_0 & |C| = 1 \text{ (kein Extra-Call)}, \\ \pi_\theta^{\text{choice}}(C) & |C| \geq 2. \end{cases} \tag{32}$$

$\pi_\theta^{\text{choice}}$ erhält eine nummerierte, natürlichsprachliche Beschreibung jeder offenen Option plus "nichts davon" und gibt eine freie Wahl zurück. Der Call nutzt `think:false` und dasselbe Timeout-Budget wie Self-AFK und News-Reaktion. Bei fehlerhafter Antwort greift der Orchestrator auf `none` zurück: ein stiller Tick ist einer erzwungenen Aktion vorzuziehen.

Nachrücken bei Ablehnung (v1.1.1.66). Lehnt das Execute des gewählten Kandidaten intern ab, rückt der Orchestrator deterministisch auf den nächsten unversuchten Kandidaten vor, ohne erneuten Wahl-Call:

$$\text{für } i \in \{1, \ldots, \min(|C|, M)\}: \quad \text{Ergebnis}_i = \begin{cases} \text{Erfolg} & \text{exec}(C_{\sigma(i)}) \neq \texttt{declined}, \\ \text{weiter} & \text{sonst}, \end{cases} \tag{33}$$

mit $M = \texttt{STU\_EMBER\_INITIATIVE\_MAX\_ATTEMPTS} \in [1, 5]$ (Default 2) und σ der ursprünglichen Sammel-Reihenfolge. Dies begrenzt Inferenz-Calls pro Tick und verhindert, dass verfügbare Slot-Kapazität brachliegt.

6.2 Werkzeug 5: Gezieltes Ansprechen zurückkehrender Spieler

Das Werkzeug `direct_address` löst aus, wenn ein Spieler aktuell online ist (innerhalb des bestehenden 90-Sekunden-Präsenz-Fensters), aber seit mindestens $\theta_{\text{Abwesenheit}}$ Stunden keine eigene Chat-Nachricht mehr gepostet hat:

$$\theta_{\text{Abwesenheit}} \;=\; \texttt{STU\_EMBER\_DIRECT\_ADDRESS\_ABSENCE\_HOURS} \;\in\; [2,\, 72]\ \text{h} \quad (\text{Default } 12\,\text{h}). \tag{34}$$

Unter allen berechtigten Online-Spielern wählt das Gate deterministisch denjenigen mit der *größten* Abwesenheitslücke. Der Cooldown wird als JSON-Map unter einem festen `stu_kv`-Key (`ember_direct_address_state`) persistiert, auf 200 Einträge gekappt. Das Execute generiert eine Antwort, die die Zielperson namentlich mit einer ungefähren Abwesenheitsdauer anspricht.

6.3 Werkzeug 2: Relevanzbasiertes Langzeitgedächtnis

Bisher lieferte `ember_mem_fetch()` stets dieselben Top-N Einträge nach globalem `relevance`/`updated_at`, unabhängig vom Gesprächsthema. Der Ersatz `ember_mem_fetch_relevant()` wendet den bestehenden `ember_lore_query_terms()`-Extraktor auf die eingehende Nachricht an und bewertet jeden Gedächtnis-Eintrag nach Treffer-Anzahl:

$$\text{score}(m_i,\, Q) \;=\; \big|\{\, w \in Q \,:\, w \in m_i \,\}\big| \cdot \lambda \;+\; \text{rel}(m_i), \tag{35}$$

wobei Q die extrahierten Suchbegriffe (max. 8, längste zuerst), $\text{rel}(m_i)$ der gespeicherte Relevanzfaktor und λ ein Skalierungsfaktor ist, der topische Trefferanzahl vor gespeicherter Wichtigkeit priorisiert. Der Kandidaten-Pool umfasst $\max(5N,\, 20)$ Einträge gefiltert per REGEXP-Alternation; bei keinem Treffer greift ein Fallback auf das alte Top-N-Verhalten.

6.4 Werkzeug 3: Spielerzustand wahrnehmen

Jeder Prompt-Aufbau enthält optional einen kompakten Spielerzustand-Block aus dem `stu_kv`-Save-State-JSON-Blob des Nutzers:

$$b_{\text{Zustand}}(u) \;=\; \langle\, \text{Level},\ \text{Planet},\ \Delta t_{\text{last}} \,\rangle, \tag{36}$$

wobei Δt_{last} die relative Zeit seit `lastPlayedAt` (Minuten / Stunden / Tage) ist. Der Block wird in u neben dem Reputations-Hinweis injiziert, ausdrücklich als Hintergrundwissen markiert, kein Auftrag, es zu erwähnen. Er wird unterdrückt, wenn der Absender die Entität selbst ist (Initiative-Ticks), um selbstreferenzielles Rauschen zu vermeiden.

6.5 Embers eigenes XP- und Level-System (v1.1.1.67)

Die Entität akkumuliert Erfahrungspunkte nach derselben exponentiellen Level-Kurve wie der Spieler-Client, jetzt serverseitig implementiert:

$$\text{XP}_{\text{benötigt}}(\ell) \;=\; \left\lfloor 100 \cdot 1{,}30^{\,\ell-1} \right\rfloor, \tag{37}$$

für $\ell = 1, \ldots, 5$ verifiziert als identisch zur Client-Formel: $100, 130, 169, 219, 285$ XP. XP wird an zwei Hooks vergeben:

$$\Delta\text{XP} \;=\; \begin{cases} \texttt{STU\_EMBER\_SELF\_XP\_PER\_MESSAGE} \in [0, 100] & \text{bei } \texttt{ember\_insert()}, \\ \texttt{STU\_EMBER\_SELF\_XP\_PER\_AFK} \in [0, 100] & \text{bei erfolgreichem Self-AFK}. \end{cases} \tag{38}$$

`ember_insert()` ist der einzige zentrale Einfügepunkt für jede Chat-Nachricht der Entität: der Hook dort deckt Chat-Antworten, Follow-up, Idle-Initiative, News-Erwähnung, gezieltes Ansprechen und Browse-Ergebnis-Auslieferung ohne pfadspezifische Instrumentierung ab. Level-Up-Ereignisse (inkl. XP-Übertrag über mehrere Level) werden geloggt; Einzel-Grants nicht, um Log-Spam zu vermeiden.

Schreibsicherheit via Advisory-Lock: GET_LOCK(`"ember_self_xp"`, 3) mit 3-Sekunden-Wartezeit (statt des 0-Sekunden-Sofort-Fails des globalen Inferenz-Locks), da XP-Grants selten kollidieren. Bei Lock-Fehlschlag wird der Grant lautlos übersprungen.

7 Diskussion: Zwei Wege zu persistenter Handlungsfähigkeit

Savage [1] präsentiert eine Hybrid-Architektur, die ein LLM innerhalb eines 220.000-Neuron-Spiking-Neural-Network (SNN)-Substrats mit Spike-Timing-Dependent Plasticity (STDP) platziert. Die Dual-GPU-Hardware (RTX 5070 Ti + RTX 4060 Ti) reflektiert den parallelen Simulations-Aufwand dieses Substrats. EMBER verfolgt einen anderen Punkt im selben Entwurfsraum: statt das LLM mit einem separaten neuronalen Substrat zu erweitern, versorgt es die internen Aufmerksamkeitsköpfe mit hochgefilterten, relevanzoptimierten Texten via BM25 aus einer relationalen Datenbank. Beide Ansätze adressieren dieselbe Grundbegrenzung , dass LLMs zwischen Calls zustandslos sind, durch verschiedene Gedächtnis-Paradigmen: assoziative neuronale Dynamiken (Savage) vs. indizierter episodischer Text (diese Arbeit). Die Architekturen sind komplementär, nicht konkurrierend: der SNN-Ansatz erlaubt emergente, domänenübergreifende Assoziationen durch zeitliche Ko-Aktivierung, während der RAG-Lite-Ansatz operative Einfachheit, Crash-Resilienz und CPU-only-Einsetzbarkeit bietet. Diese Einordnung dient allein dazu, die eigenen Hardware-Entscheidungen zu kontextualisieren, nicht dazu, die eine Architektur gegen die andere abzuwägen.

7.1 Das Thinking-Bleed-Phänomen

Vgl. Abschnitt 4.12. Entscheidend ist die System-Invariante: `think:true` wird *niemals* explizit im Payload gesetzt (Thinking lebt ausschließlich im Modelfile); `think:false` gilt gezielt für

alle Utility-Calls. Eine spätere Härtung ersetzte den JSON-Grammatik-Zwang vollständig durch `want_json=false` mit robustem nachgelagertem Parsing.

8 Empirische Beobachtungen und Entwurfslehren

Fehlermodus	**Wurzelursache**	**Auflösung**
Globaler Lore-Timeout (353 s total)	Hartkodierter 240-s-Curl-Deckel ignorierte `STU_EMBER_TIMEOUT`	Floor auf 420 s / Cap auf 480 s; config-gesteuert (§4.5)
Aktions-Loop ohne Aktion	Thinking-Bleed bei `format:json`	`think:false` für Utility-Calls (§4.12)
Agent blind für Seiteninhalt	Beobachtung enthielt nur Accessibility-Tree-Elemente	$\mathcal{T}_t$ in o_t aufnehmen (§4.8)
Endlos-Tippen derselben Suche	`type+submit` löste keine Navigation aus	Aktions-Signatur-Loop-Schutz + Such-Substitution (§4.8)
CAPTCHA / 403 beim Scrapen	SERPs aus Server-IP gesperrt	Loopback-gebundener Meta-Such-Aggregator (§4.8)
Doppelte Antwort (Slow-Stream)	Poll-Pfade registrierten ID nicht	Dual-Path-Idempotenz: ID in $\mathcal{S}$ in jedem Pfad (§4.6)
Live-Fenster öffnet nicht	Status-Filter schloss fertige Jobs aus (Race)	Recency-Selektion $\arg\max t_{\text{creation}}$ (§4.10)
Präsenz hängt ewig auf "abwesend"	Binäres Flag übersteht harten Abbruch nicht	TTL-Flag $\Theta > T_{\text{cap}}$ (§4.7)
JS-Fix greift nicht	Cache-Buster eingefroren	Cache-Buster ins Versionierungswerkzeug aufnehmen
Idle-Initiative blockiert sich dauerhaft	Letzter-Sender-Check blockierte unbegrenzt	Alters-basierte Schwelle: $2\times\theta_{\text{base}}$ wenn Entität zuletzt schrieb (§5)
Idle feuert bei jedem 5-min-Tick	Jitter-Patch entfernte versehentlich Basis-Zuweisung; `php -l` stumm	Funktionaler Spot-Check mit Beispielwerten zwingend (§5)
Doppel-Trigger (3-min-Abstand)	Zwei unabhängige Pfade teilten keinen Cooldown	Einheitlicher Alters-Check gegen dieselbe DB-Zeile
Self-AFK: Modell wird nie gefragt	Externer `mt_rand()`-Filter gatete den Modell-Call	Filter entfernt; `wants_afk`-Output wird direkt vertraut (§5)
News-Check im Log unsichtbar	Alle Exit-Pfade waren stille `return false`	`stu__log_error()` an jedem Exit inkl. `none`-Relevanz (§5.4)
Wahl-Call bei Einzel-Kandidat	Unnötiger Inferenz-Call wenn nur eine Option offen	Wahl-Call nur bei $\lvert C\rvert \geq 2$ (§6)
Gedächtnis liefert immer dasselbe Top-N	`ember_mem_fetch()` ignorierte Thema, sortierte nur nach gespeicherter Relevanz	REGEXP-basiertes `ember_mem_fetch_relevant()` mit Top-N-Fallback (§6)
XP-Grant-Race bei parallelen Inserts	Read-Modify-Write auf JSON-Blob nicht atomar	`GET_LOCK("ember_self_xp",3)`; bei Fehlschlag überspringen (§6)

Tabelle 5: Fehlermodi, Wurzelursachen und architektonische Auflösungen (v1.0–v1.2).

9 Fazit

EMBER zeigt, dass vollautonome kognitive Eigenständigkeit auf nicht-akzelerierter CPU-Hardware erreichbar ist, wenn die Architektur um die Hardware-Randbedingungen herum entworfen wird statt gegen sie. Der zentrale empirische Befund, durch direkte Messung bestätigt, ist, dass die CPU-only-Decode-Rate substrat- und konfigurationsgebunden ist, nicht modellintrinsisch: Die Pre-Upgrade-Zuweisung konvergierte auf $\approx$ 15–16 Tok/s, der Post-Upgrade-Spotcheck erreichte 17,31 Tok/s, und eine direkte Neumessung auf der aktuellen Produktivkonfiguration konvergiert auf $\approx$19–20 Tok/s. Prompt-Masse bleibt der dominante Ingenieur-Hebel, unabhängig davon, wo der Decode-Koeffizient aktuell liegt.

Der zweite Beitrag ist die autonome Initiative-Architektur (Phase 3c/3d/3e): ein cron-getriebenes Tick-System mit Follow-up, Idle-Initiative (deterministischer Jitter), selbstbestimmtem AFK und gestufter News-Reaktion. Phase 3e ersetzt die feste Prioritätskette durch einen deliberativen Orchestrator mit fünf Werkzeugen, darunter gezieltes Ansprechen zurückkehrender Spieler, relevanzbasiertes LZG-Retrieval, Spielerzustand-Wahrnehmung und ein eigenes XP/Level-System der Entität, sowie einem deterministischen Nachrück-Mechanismus bei Ablehnung. Alle Modi teilen denselben Mutex und dieselbe Generierungs-Pipeline ohne neue Nebenläufigkeits-Primitive.

Der dritte Beitrag ist ein Katalog von Produktions-Fehlermodi, der für praktische Systeme unter denselben Randbedingungen nützlich sein soll.

Künftige Arbeit umfasst eine Einsparung einer Generierung im Web-Pfad, longitudinale Analyse des LZG- und XP-Wachstums sowie Muster-Analyse der autonomen Werkzeugwahl im Langzeitbetrieb.

A Konfigurations-Footprint

Der folgende Konfigurations-Footprint dokumentiert eine repräsentative Momentaufnahme der Runtime-Steuerfläche zum Zeitpunkt der Erstellung. Nicht-sensitive Runtime-Werte sind als illustrative Beispiele des Betriebsbereichs der Architektur zu verstehen, nicht als garantierte, aktuell live gültige Konstanten; sie können sich jederzeit ohne gesonderten Hinweis ändern und sollten nicht als exakte Echtzeit-Spezifikation des Produktivsystems herangezogen werden. Deployment-lokale Kennungen, Endpunkt-Ports und Shared Secrets sind fuer die oeffentliche Version redigiert. Die Inline-Kommentare spiegeln die generelle Architektur-Semantik wider, keinen Live-Konfigurations-Dump.

Listing 1: Repräsentativer Konfigurations-Footprint der EMBER-CPU-Architektur (illustrativ, Änderungen vorbehalten).

```
<?php
// EMBER-Kernidentitaet und Modellbindung.
define('STU_EMBER_ENABLED', true);                 // Aktiviert den EMBER-Integrationspfad.
define('STU_EMBER_USER_ID', 0);                    // Redigierter Platzhalter fuer den deployment-lokalen
    EMBER-Service-User.
define('STU_EMBER_CHARACTER_ID', '<character_id>'); // Redigierter Platzhalter fuer den stabilen Character
    -Datensatz.
define('STU_EMBER_CHARACTER_NAME', 'Ember');       // Anzeigename fuer den Runtime-Kontext.
define('STU_EMBER_MODEL', 'gemma4:26b');           // Primaeres Ollama-Modell als kognitiver Kern.
define('STU_EMBER_OLLAMA_URL', 'http://127.0.0.1:<ollama_port>/api/chat'); // Redigierter Loopback-only
    Ollama-Chat-Endpunkt.

// CPU-Inferenz und Generierungssteuerung.
define('STU_EMBER_KEEP_ALIVE', -1);                // Haelt das Modell zwischen Calls resident; negativer
    Wert bedeutet dauerhaft warm.
define('STU_EMBER_NUM_THREAD', 12);                // Dedizierte Inferenz-Threads auf der reservierten CPU
    -Partition.
define('STU_EMBER_NUM_PREDICT', -1);               // Model-Default-Sentinel; Backend-Passthrough/Cap-
    Logik bestimmt das finale Budget.
define('STU_EMBER_TEMPERATURE', 0.8);              // Ausbalancierte Volatilitaet fuer charakterstabile,
    aber lebendige Ausgabe.
define('STU_EMBER_TOP_P', 0.90);                   // Nucleus-Sampling-Schwelle zur Diversitaetskontrolle.
define('STU_EMBER_TOP_K', 40);                     // Groesse der Token-Kandidatenliste beim Sampling.
define('STU_EMBER_REPEAT_PENALTY', 1.08);          // Milde Wiederholungsstrafe gegen Loops ohne
    Stilabflachung.
define('STU_EMBER_REPEAT_LAST_N', 64);             // Token-Fenster fuer die Wiederholungsstrafe.
define('STU_EMBER_SEED', -1);                      // Nicht-deterministischer Seed fuer natuerliche
    Variation.
define('STU_EMBER_NUM_CTX', 8192);                 // Begrenztes Kontextfenster gegen Prompt-Masse und
    Memory-Druck.

// Runtime-Taktung, Transportlimits und Initiative-Authentifizierung.
define('STU_EMBER_IDLE_INTERVAL', 300);            // Basis-Idle-Tick in Sekunden fuer autonome/background
     Checks.
define('STU_EMBER_MAX_REPLY_CHARS', 6200);         // Harte Antwortgroessen-Grenze vor Ausgabe in die Chat
    -UI.
define('STU_EMBER_TIMEOUT', 360);                  // Primaerer cURL-Ausfuehrungsdeckel fuer lange CPU-
    only Generierungen.
define('STU_EMBER_TIMEOUT_RETRY', 360);            // Retry-Deckel fuer fortgesetzte oder bereits warme
    Folge-Calls.
define('STU_EMBER_INITIATIVE_TOKEN', '<redacted_initiative_token>'); // Redigiertes Shared Secret fuer
    Cron/Initiative-Endpunkt-Authentifizierung.

// Datenbankgestuetztes episodisches Langzeitgedaechtnis und Reflexions-Extraktion.
define('STU_EMBER_MEMORY_ENABLED', true);          // Aktiviert Injection gespeicherter episodischer
    Fakten in den Runtime-Prompt.
define('STU_EMBER_MEMORY_LIMIT', 15);              // Maximale Anzahl injizierter Memory-Fakten pro Turn.
define('STU_EMBER_REFLECT_ENABLED', true);         // Aktiviert Extraktion stabiler Fakten ins LZG nach
    abgeschlossenen Austauschen.
define('STU_EMBER_REFLECT_EVERY_N', 1);            // Reflexionskadenz: illustrativer Wert; jeder
    geeignete Austausch in diesem Betriebsmodus.
define('STU_EMBER_REFLECT_TIMEOUT', 244);          // Kuerzerer Deckel fuer Utility-Reflexionscalls zum
    Schutz des Inferenz-Slots.
define('STU_EMBER_REFLECT_MAX_FACT_LEN', 340);     // Maximale Laenge eines gespeicherten Facts nach
    Reflexions-Cleanup.

// Optionales Utility-Modell-Routing und Moderations-Gates.
```

```
define('STU_EMBER_REFLECT_MODEL', '');            // Leerer String nutzt STU_EMBER_MODEL als Reflexions-Fallback.
define('STU_EMBER_IDLE_GREET_MINUTES', 60);       // Mindestabwesenheit, bevor Rueckkehr-Begruessung berechtigt ist.
define('STU_EMBER_AUTOMOD', true);                // Basis-Automod fuer globale Link-/Vulgaritaets-Mutes.
```

Sicherheits- und Reproduzierbarkeitshinweis. Diese Darstellung bleibt bewusst auf der architektonisch-mathematischen Ebene. Der Konfigurations-Anhang ist ein oeffentlicher Reproduzierbarkeits-Footprint, keine operative Spezifikation: Die Werte sind illustrativ, können sich zwischenzeitlich geändert haben, und es sollte nicht angenommen werden, dass sie auf dem Live-System weiterhin in dieser Form gelten. Deployment-lokale Kennungen, Endpunkt-Ports und Shared Secrets sind redigiert. Externe Routen, Dateipfade, Datenbank-Identifikatoren, Konto-Kennungen und Zugangsdaten sind ausgespart.

Literatur

[1] W. Savage, *EMBER: Autonomous Cognitive Behaviour from Learned Spiking Neural Network Dynamics in a Hybrid LLM Architecture*, arXiv:2604.12167v1 [cs.AI], April 2026.

[2] S. Robertson, H. Zaragoza, *The Probabilistic Relevance Framework: BM25 and Beyond*, Foundations and Trends in Information Retrieval, 3(4):333–389, 2009.

[3] S. Yao et al., *ReAct: Synergizing Reasoning and Acting in Language Models*, arXiv:2210.03629, 2022.

[4] C. Packer et al., *MemGPT: Towards LLMs as Operating Systems*, arXiv:2310.08560, 2023.