ChatGPT

Analyse des Sheratan Self-Loop-Systems und der LLM-Loops

Kontext und Vorgehensweise

Der Auftrag lautete, ein im Projekt hinterlegtes ZIP-Archiv zu entpacken, dessen Inhalte zu analysieren und die darin beschriebenen „LLM-Calls“ sowie die resultierenden **Loops** auf Konsistenz und **evolutionäre Effizienz** zu prüfen. Bei der Recherche in den bereitgestellten GitHub-Repositories sowie im lokalen Projektverzeichnis war kein ZIP-Archiv auffindbar. Stattdessen wurden die Dateien des Projekts direkt analysiert. Insbesondere die Dokumentation zum Self-Loop-System (Markdown-Datei) und die Python-Module `worker_loop.py` und `loop_runner.py` geben Einblick in die Abläufe der Loops und die Art der LLM-Aufrufe. Die folgenden Abschnitte fassen die gefundenen Informationen zusammen, analysieren die aktuelle Logik und geben konkrete Verbesserungsvorschläge.

Self-Loop-System vs. LCP-System

Die Dokumentation beschreibt zwei unterschiedliche Ausführungsmodi:

Merkmal	LCP-System	Self-Loop-System
Datenformat	Striktes JSON mit Feldern `decision`, `actions` und `explanation`	Lesbares Markdown mit vier Sektionen (A: Standortanalyse, B: Nächster Schritt, C: Umsetzung, D: Vorschlag für nächsten Loop)
Philosophie	Der LLM agiert als „Tool-Executor“. Jeder Job ist isoliert und führt lediglich konkrete Aktionen (Datei lesen, schreiben, Aufruf etc.) aus.	Der LLM wird als **kooperativer Ko-Denker** verstanden. In jeder Iteration wird ein kompletter Planungszyklus durchlaufen: Es findet eine Standortanalyse statt, der Agent wählt selbstständig den nächsten sinnvollen Schritt, führt diesen aus und schlägt anschließend einen nächsten Loop vor [1].
State Management	Kein persistenter Zustand; jeder Job ist isoliert.	Es existiert ein `loop_state` mit Zähler für die Iteration, History-Summary, offenen Fragen und Constraints. Die Loops können so auf vorherige Schritte aufbauen [2].
Use Cases	Geeignet für klar definierte, kurze Tasks wie das Lesen oder Schreiben von Dateien, API-Aufrufe usw.	Geeignet für iteratives Planen, strategische Problemstellungen und Multi-Step-Reasoning über mehrere Iterationen [3].

Der Self-Loop-System-Prompt gibt dem Modell klare Anweisungen: Es soll **in jeder Iteration nur einen sinnvollen Fortschritts-Schritt ausführen**, eine Standortanalyse liefern, den nächsten Schritt wählen,

diesen konkret umsetzen und einen Vorschlag für den nächsten Loop machen [4]. Dadurch wird ein iterativer Lern- und Verbesserungsprozess ermöglicht.

LLM-Aufruf im Worker (Planungsebene)

`call_llm_for_plan`

In `worker/worker_loop.py` wird über die Funktion `call_llm_for_plan` ein Plan für eine Folge von Jobs erstellt. Falls keine LLM-Endpunkt-URL konfiguriert ist, liefert die Funktion einen statischen Plan: Sie listet alle Python-Dateien auf und liest anschließend `main.py` [5]. Wird jedoch ein LLM-Endpunkt konfiguriert, wird ein System-Prompt generiert, der das LLM auffordert, auf Basis der Benutzeranfrage (`user_prompt`) und des Projektverzeichnisses (`project_root`) einen Plan aus 2-4 Folge-Jobs zu erstellen. Der Prompt definiert strikte Regeln: das Modell darf nur gültiges JSON ausgeben, nie Rückfragen stellen und muss ein bestimmtes Format verwenden [6]. Der Plan besteht aus einer Liste von Jobs mit Feldern `name`, `kind`, `params` und optional `auto_dispatch`.

`call_llm_generic`

Diese Funktion erzeugt einen generischen LLM-Aufruf für LCP-basierte Tasks. Wenn kein `SHERATAN_LLM_BASE_URL` definiert ist, gibt sie eine Fehlermeldung zurück [7]. Andernfalls extrahiert sie aus dem Job-Payload Kontextinformationen (Mission, Task und Parameter) und erzeugt einen strikten Prompt. Wichtig ist die klare Vorgabe, dass das Modell **ausschließlich JSON im LCP-Format** ausgeben darf. Dadurch wird verhindert, dass das Modell plausibel klingenden Fließtext zurückschickt, der nicht parsebar ist [6]. Nach dem Aufruf wird die Antwort auf Gültigkeit geprüft und ggf. ein Fallback oder Fehler ausgegeben.

Bewertung

Die strengen Vorgaben sorgen dafür, dass das Modell deterministisch arbeitet und auch ohne LLM-Anbindung funktionsfähig bleibt (statischer Plan). Allerdings haben beide Funktionen aktuell keine adaptive Lernlogik: Jeder Plan wird isoliert erstellt, es gibt keine Berücksichtigung früherer Ergebnisse. Für evolutive Effizienz wäre es sinnvoll, die Qualität vergangener Pläne zu bewerten und die Prompt-Parameter (z. B. Temperatur) oder die Zahl der Schritte dynamisch anzupassen.

Ausführung von Self-Loops im Loop-Runner

Der Loop-Runner (`loop_runner.py`) steuert den eigentlichen Self-Loop. Wichtige Aspekte:

1. **Initialisierung der Konfiguration:** Die Klasse `LoopRunner` liest ein `job_config` und erstellt daraus `LoopConfig` (u. a. `max_iterations`, `max_consecutive_errors`, `profile`) sowie `ModelConfig` und `MissionState`. Der `context_packet` sammelt Mission-ID, Ziel, Progress-Log und State [8].

2. **Iterativer Ablauf:** In der Methode `run` wird in jeder Iteration

3. der Iterationszähler erhöht,

4. das `context_packet` mit dem aktuellen Zustand synchronisiert,
5. aus dem Kontext ein Prompt via `build_selfloop_prompt` erstellt,
6. das LLM über den `LLMClient` aufgerufen (der momentan unimplementiert ist),
7. die LCP-Antwort validiert,
8. die resultierende Entscheidung verarbeitet und entsprechende Aktionen ausgeführt [9].
9. **Error-Handling und Safe Mode:** Wenn der LLM-Call fehlschlägt oder die Antwort nicht den Erwartungen entspricht, erhöht `_on_error` den Fehlerzähler und schaltet nach einer definierten Zahl von Fehlern in den **Safe-Mode**. Im Safe-Mode wird der Loop in einen Debug-Modus versetzt, in dem riskante Aktionen unterdrückt werden [10]. Dadurch wird verhindert, dass sich Fehler unkontrolliert aufschaukeln.
10. **Abbruchbedingungen:** Der Loop endet, wenn
11. das `finished`- oder `aborted`-Flag im Mission-State gesetzt ist,
12. die Maximalzahl an Iterationen erreicht ist (Schutz vor unendlichen Schleifen),
13. ein Safe-Mode-Ereignis das Programm stoppt [9].

Bewertung

Der Loop-Runner enthält robuste Mechanismen zur Fehlerbegrenzung und Stopplogik. Er überprüft jede LLM-Antwort auf Format-Konsistenz und bricht bei ungültigen Antworten ab. Allerdings fehlen auch hier Elemente, die echte „Evolution“ ermöglichen. Der `context_packet` enthält zwar einen State und ein Progress-Log, aber es gibt keine definierte **Lernstrategie**, die aus früheren Iterationen Schlüsse zieht. Eine LLM-Antwort in späteren Iterationen erhält nur indirekt über die `history_summary` Informationen über das bisherige Vorgehen; wie diese Zusammenfassung erstellt wird, ist im Code nicht zu finden. Das System nutzt keine Metriken, um den Fortschritt zu messen oder Parameter dynamisch zu verändern.

Konsistenz und evolutionsbezogene Effizienz

Die vorhandenen Loops sind konsistent, was das Einhalten der Formate und die Fehlerbehandlung betrifft. Sie verhindern Endlosschleifen durch eine **Maximalanzahl an Iterationen** und schalten bei zu vielen Fehlern in einen Safe-Mode, der die Ausführung stoppt. Allerdings zeigen sie folgende Schwächen im Hinblick auf evolutive Effizienz:

1. **Fehlende Erfolgsmetriken:** Es gibt keine Bewertung, ob ein Schritt zum Ziel beigetragen hat. Ohne „Fitness-Funktion“ kann nicht bestimmt werden, ob eine Änderung des Plans sinnvoll war.
2. **Starre Prompt-Parameter:** Temperatur, Token-Limit und Anzahl der Schritte sind fest vorgegeben. Ein adaptiver Ansatz, der sich bei Stagnation für explorativere Prompts entscheidet oder bei Erfolg fokussierter arbeitet, könnte die Evolution beschleunigen.
3. **Eingeschränkte Nutzung von `loop_state`:** Der `loop_state` speichert Iteration und eine History-Summary, aber im Code ist keine Logik vorhanden, um diese Summary zu aktualisieren oder daraus neue Constraints abzuleiten. LLM-Aufrufe könnten durch detailliertere Kontextdaten (z. B. Liste der bisherigen Aktionen, Ergebnisse, entdeckte Fehler) deutlich präziser arbeiten.

4. **Kein selbstkorrigierendes Lernen:** Bei Fehlern wird lediglich der Fehlerzähler erhöht, aber es werden keine alternativen Strategien ausprobiert. Ein evolutives System würde bei wiederholten Fehlschlägen den Prompt anpassen, die Nutzung anderer Tools prüfen oder zusätzliche Informationen sammeln.

5. **Fehlende Integration externer Ressourcen:** Der Auftrag deutete auf die Analyse von PDF-Inhalten hin („JSON-Versionen des PDF-Inhalts“). Im aktuellen Code sind keine Funktionen vorhanden, um PDFs zu extrahieren und in strukturierter Form bereitzustellen. Ein solches Feature könnte das System um Lernmaterial bereichern.

Vorschläge für Verbesserungen

Um die Loops nicht nur konsistent, sondern **evolutiv effizient** zu gestalten, bieten sich folgende Maßnahmen an:

1. **Evaluationsmetriken einführen:**

2. Definieren Sie für jede Mission eine Metrik, die den Fortschritt misst (z. B. Anzahl implementierter Features, Laufzeitverbesserung, Reduktion der Fehlermeldungen). Speichern Sie diese Kennzahlen nach jedem Loop und vergleichen Sie sie mit dem vorherigen Stand. So können Sie feststellen, ob der Loop zu einer Verbesserung geführt hat.

3. **Adaptive Steuerung der LLM-Parameter:**

4. Variieren Sie Temperatur und Token-Limit in Abhängigkeit von den bisherigen Ergebnissen. Falls mehrere Iterationen keine Verbesserung bringen, erhöhen Sie die Temperatur (exploratives Verhalten). Bei guten Ergebnissen senken Sie sie (konvergentes Verhalten).

5. Passen Sie die Anzahl der im Plan generierten Schritte dynamisch an. Bei komplexen Aufgaben können mehr Schritte sinnvoll sein; bei klaren Aufgaben genügen wenige.

6. **Verbesserte Nutzung von `loop_state` und Memory:**

7. Implementieren Sie eine Funktion, die nach jedem Loop eine **zusammenfassende History** aus den durchgeführten Aktionen erstellt und im `loop_state` speichert. Diese Zusammenfassung könnte z. B. die wichtigsten Erkenntnisse aus PDF-Analysen, entdeckte Probleme und gelöste Aufgaben enthalten.

8. Fügen Sie eine Liste von **offenen Fragen** und **Constraints** hinzu, die automatisch aus den Ergebnissen generiert werden. So kann der LLM im nächsten Loop gezielt darauf eingehen [2].

9. **Simulation und Training ohne echtes LLM:**

10. Entwickeln Sie einfache **Simulationsfunktionen**, die eine LLM-Antwort approximieren. So können Sie den Loop-Mechanismus testen, ohne auf ein LLM zuzugreifen. Die Simulation kann deterministische Pläne zurückgeben (z. B. „Liste Dateien“, „Lies README“, „Extrahiere PDF-Text“) und dabei zufällig kleine Variationen einbauen, um die Lernlogik zu testen.

11. **Robustes Parsing von LLM-Rückgaben:**

12. Auch wenn der Prompt strikte Vorgaben macht, kann eine echte LLM-Antwort fehlerhaft sein. Implementieren Sie Parser, die JSON auch aus unstrukturiertem Text extrahieren (RegEx oder heuristisches Filtern), bevor ein Loop abgebrochen wird. Damit lassen sich mehr Antworten retten und die Evolution bleibt nicht wegen eines Parsefehlers stehen.

13. **Integration eines PDF-Extractors:**

14. Fügen Sie einen **PDF-to-JSON-Konverter** als Tool hinzu. Damit können PDF-Dokumente in strukturierte Daten verwandelt werden, die der Loop analysieren kann. In den Schritten: (1) `list_files` nach `**/*.pdf`, (2) `read_pdf` oder `convert_pdf_to_json`, (3) Extraktion und Speicherung der Inhalte als JSON. Diese JSON-Daten können dann in das `core_data` des nächsten Loops aufgenommen werden, wodurch das LLM auf neu gewonnene Informationen zugreifen kann.

15. Alternativ können PDF-Texte in ein Vektor-Store geschrieben und beim nächsten LLM-Aufruf zur Kontextanreicherung verwendet werden.

16. **Lernschritte persistieren:**

17. Erfassen Sie nach jedem Loop, welche **Lernschritte** abgeschlossen wurden und speichern Sie diese in einer Versionshistorie (z. B. als JSON-Datei). Die Versionen könnten das Datum, die Iterationsnummer, wichtige Erkenntnisse und den aktuellen Planungsstand enthalten. Ein Viewer könnte diese Historie visualisieren, was wiederum das menschliche Feedback erleichtert.

18. **Optimierung des Safe-Mode:**

19. Anstatt im Safe-Mode lediglich Aktionen zu blockieren, könnten Sie automatisierte Diagnose-Schritte durchführen (z. B. Log-Analyse oder Code-Linting), um die Fehlerursache zu identifizieren. Außerdem könnte der Safe-Mode nutzen, um das Prompting anzupassen, statt nur abzusichern.

Fazit

Die untersuchten Dateien geben einen guten Überblick über das Sheratan-Self-Loop-System. Der derzeitige Stand sorgt dafür, dass Loops formale Konsistenz wahren und Fehler kontrollieren. Allerdings fehlt es an Mechanismen, die zu einer echten **evolutiven Verbesserung** führen. Eine Erweiterung um Erfolgsmessungen, adaptive Parametersteuerung, bessere Nutzung des `loop_state` sowie die Integration von PDF-Extraktion würde das System deutlich leistungsfähiger machen. Zudem sollten strenge LLM-Prompts mit robustem Parsing ergänzt und Simulationsmöglichkeiten genutzt werden, um die Loops auch ohne externes LLM zu testen. Mit diesen Verbesserungen können die Self-Loops nicht nur konsistent, sondern auch effizient und lernfähig arbeiten.

---

1 2 3 4 raw.githubusercontent.com
https://raw.githubusercontent.com/J3r3C0/2_sheratan_core/5defaa6916544056ade27e667fe2c9ab2a79c444/SELFLOOP_SYSTEM.md

5 6 7 raw.githubusercontent.com
https://raw.githubusercontent.com/J3r3C0/2_sheratan_core/5defaa6916544056ade27e667fe2c9ab2a79c444/worker/worker_loop.py

8 9 10 raw.githubusercontent.com
https://raw.githubusercontent.com/J3r3C0/2_sheratan_core/5defaa6916544056ade27e667fe2c9ab2a79c444/core/sheratan_core_v2/loop_runner.py