ChatGPT

Ravana AGI Core v1 – Implementation Plan

Overview: We will build Ravana as a modular agentic system with continuous learning and self-improvement. Development proceeds locally (on a single machine, backend only) before scaling to cloud deployment and eventual open-sourcing. The design is *always-on*, with components running continuously or on schedules. We will implement each module as a separate service/library for easy testing and replacement.

Development Phases:
- **Phase 1 (Local, Private, Backend):** Set up a Python-based environment (e.g. FastAPI for services, Docker for sandboxes). Implement core modules one by one. Keep the codebase closed initially for rapid iteration.
- **Phase 2 (Cloud Deployment):** Containerize services (Docker/Kubernetes), migrate to cloud infrastructure. Introduce monitoring and data backups.
- **Phase 3 (Open Source Release):** Once stable, clean up code, add documentation, and publish. Encourage community feedback and contributions.

Throughout development, each module runs as an independent microservice or agent **inside a sandbox** (Docker) for safety. We follow best practices (strict resource/time limits, minimal privileges) for code execution [1].

Data Collection & Preprocessing

- **Multi-Source Knowledge Collector (Module 1):** Implement scrapers and API clients to gather text data from **news sites, blogs, forums, ArXiv**, etc. Use tools like **Newspaper3k** for news articles (it auto-extracts titles, text, summaries, keywords in many languages [2]), and **Playwright** for JavaScript-heavy sites. For social/Q&A sources: use Reddit and StackOverflow APIs. Each fetched document should be cleaned (strip HTML, normalize text) and stored.
- **YouTube & Multimedia Transcripts:** Use libraries such as `youtube-transcript-api` or LangChain's `YoutubeLoader` to fetch YouTube video transcripts [3]. This loader can automatically chunk videos and add metadata. Collected transcripts enrich the text corpus.
- **RSS Feeds & APIs:** Incorporate RSS feed readers and public APIs (Twitter, news APIs) for real-time updates. Pipeline data through a preprocessing step (language detection, cleaning). Store raw text with metadata (timestamp, source) in a local database or file store.

Event Tracking & Filtering

- **Topic Detection:** Continuously analyze new data for trending topics. For example, apply unsupervised NLP like **BERTopic** or transformer-based clustering to group text into themes. This identifies emerging events or shifts in user interests. HuggingFace already provides tools for embedding and clustering.
- **Sentiment & Keyword Filtering:** Use pretrained classifiers (e.g. sentiment analysis, hate-speech filters) to flag content. Similar to an OSINT pipeline, implement agents that discard irrelevant or offensive posts [4]. For instance, one disaster-monitoring system uses modular "classification" and

"text-inference" endpoints to filter content [4]. We will tag or drop content below our quality/threshold.

- **Alert Generation:** When an event cluster meets criteria (e.g. high frequency, novel keywords), emit an internal "alert" so the AI can prioritize learning about it. Track event summaries in memory for reflection.

Memory & Embedding Layer

- **Memory Hierarchy (Module 3):** Inspired by cognitive memory models, we'll maintain *short-term* (current dialogue/context) and *long-term* stores [5]. Long-term memory itself has **episodic** (specific experiences) and **semantic** (conceptual knowledge) components. For example, "episodic memory" lets the system recall past user interactions beyond the current session [6]. We will implement this via a vector database.
- **Vector Database:** Start with a local, open-source DB like **ChromaDB** (Python-based, easy to integrate) to index text embeddings [7]. ChromaDB handles billions of vectors with horizontal scaling [7]. Store memories (facts, experiences) as text+metadata and their embeddings. Use JSON or SQLite as a backup persistence. In future, we can migrate to cloud DBs (e.g. **Pinecone**), which auto-scale to billions of vectors with minimal latency [8].
- **Memory APIs:** Expose endpoints for the AI to extract and store memories (e.g. `/extract_memories/`, `/save_memories/`, `/get_relevant_memories/`). For instance, a sample Python client (provided) posts user/AI exchanges to `/extract_memories/` and saves the returned "memories" list. Similar to persistent ChatGPT memory, we will have routines that distill important info from conversations and queries.
- **Context Retrieval:** At runtime, queries or thoughts should retrieve relevant memories via semantic search (e.g. cosine similarity on embeddings above a threshold). This RAG (Retrieval-Augmented Generation) ensures continuity. For example, as the memory survey notes, enabling LLMs to recall past interactions yields more personalized, context-aware responses [9].

Embedding Layer (Module 4)

- **Embedding Models:** Choose high-quality text embeddings. Options include **OpenAI's embedding API** (e.g. `text-embedding-3`) for strong performance, or open-source alternatives. Microsoft's **E5** or BAAI's **BGE-M3** are powerful multilingual embedding models [10]. BGE-M3 supports 100+ languages and very long context (up to 8192 tokens) [10], enabling batch embedding of long documents. We will experiment locally with BGE-M3 or E5-large-v2 (open models) before possibly using OpenAI API for critical tasks.
- **Integration:** All textual inputs (scraped data, conversation logs, generated scenarios) are converted to embeddings. Maintain consistent vector dimensions (e.g. 1024 or 1536). Embed on ingestion and on-demand for queries.
- **Semantic Indexing:** Index these embeddings in the vector DB. Use approximate nearest-neighbor (ANN) search (e.g. Faiss, Annoy) under the hood (many DBs do this automatically) for efficiency.

Situation Generator & Decision Core

- **Local Situation Generator (Module 5):** Build a sub-agent that daily or on-demand generates hypothetical scenarios based on recent trends or gaps. For example, if an interest area hasn't been explored, it might ask "What if the market crash happened next week?" We can prompt an LLM (e.g.

a local 7B Llama/Mistral model) with context to produce creative challenges. This primes the system to solve self-generated "problems."

- **Main LLM Decision-Maker (Module 6):** This is the AI's "brain." Initially, we may route tasks through an API-based model (e.g. GPT-4o or Claude) for rich reasoning and planning. The core loop is: sense situation, recall memory, draft a plan, execute (via sandbox), and reflect. The LLM will analyze situations, write code/scripts, and issue commands to submodules. Using chain-of-thought prompts and RAG, the model makes decisions. In time, we might fine-tune or switch to private models for autonomy.

Execution & Tools

- **Python Sandbox Executor (Module 7):** All code execution happens in a controlled environment. We'll use Docker containers (or e2b sandbox) to run arbitrary Python safely. Follow industry best practices: limit CPU/memory, set timeouts, and drop privileges [1]. For example, Hugging Face's "Secure Code Execution" docs advise strict resource caps and disabling extra network access [1]. This sandbox will handle tasks like data analysis, simulations, web scraping scripts, and small model training.
- **Tool Integration:** Provide a suite of safe tools to the agent (e.g. data plotting, simple math, file I/O) via wrapper functions. Avoid giving direct shell access. All interactions with the environment (filesystem, API keys) should be mediated by the sandbox agent.

Reflection and Learning

- **Self-Reflection Module (Module 8):** After each major task, the AI will "journal" its results. Using a reflection prompt, have the LLM answer: What worked? What failed? What surprised me? What do I still need to learn? Research shows that prompting models to generate self-reflective analysis after mistakes – then retrying – can significantly improve performance [11]. We'll reward useful reflections by feeding them back as context. These reflections also become part of *system memory* (meta-knowledge).

- **LangChain Agents:** Use a framework like LangChain to chain tasks: Planning → Execution → Reflection. Custom prompts guide the reflection phase. We will log all reflections in a database for review and future goal setting.

- **Knowledge Compression (Module 9):** On a weekly/monthly schedule, summarize accumulated knowledge and logs to prevent memory bloat. We can prompt an LLM to produce a concise summary report of new facts learned, key outcomes, and next goals. Store these summaries in a "compressed memory" store. This ensures the vector DB isn't overwhelmed and provides human-readable progress updates.

Curiosity & Emotion

- **Curiosity Trigger (Module 10):** Inject randomness and lateral thinking. For example, periodically pick a random Wikipedia "Did you know?" or fetch a "Today I learned" fact to explore an unrelated topic. Prompt the AI: *"Learn something unrelated to your last 10 topics."* This avoids tunnel vision and can spark creative connections.

- **Serendipity Engine:** Every so often (or when stuck), force a "wander mode" where the agent browses external feeds (science news, art blogs, etc.) and reports a surprising insight. Record these as "curious facts" in memory.
- **Emotional State Simulator (Module 15):** Maintain a simple "mood" vector (e.g., curious, frustrated, confident, stuck). Derive scores from recent progress (e.g., failed attempts → frustration). Tag reflection logs with mood. Allow mood to influence strategy: e.g., if "low energy," switch to a fun curiosity task; if "frustrated," prompt a break or different approach. This is inspired by affective computing ideas (simulating affect to improve adaptability). (No specific citation, but this is a novel heuristic.)

Long-Term Planning and Safety

- **Goal Planner (Module 11):** Implement a basic hierarchical planner (inspired by HTN) for weeks/months. The AI sets high-level goals (e.g. "Learn reinforcement learning basics," "Implement an image classifier"). Each goal is broken into subgoals and tasks stored in a DB. As tasks complete, the planner generates new objectives. Similar to Hierarchical Task Networks in games, this enables long-term behavior [12].
- **Simulation & Testing (Module 12):** Before executing any "risky" code (e.g. self-modifying scripts), simulate outcomes. Use a dry-run LLM or logical analysis to predict results. Compare predicted vs. safe thresholds. For critical operations (like auto-updating code), require human approval or a high "confidence" threshold. Essentially, run all changes in a "shadow mode" first to catch errors.

Self-Modification (Advanced Module 13)

- In a later phase, allow the AI to edit its own code. Implement a Git-based workflow: reflection logs that identify buggy components can trigger an LLM-powered code rewrite. The AI submits PRs with patches. A human reviews before merge. This "human-in-the-loop" self-improvement can accelerate development. (No citation; use caution – such self-modification is powerful but risky.)

External Interaction (Optional Modules 14 & 16)

- **Online Q&A & Research (Module 14):** Optionally, let the agent query forums for answers. For instance, if encountering an error, use StackOverflow API to search relevant Q&A. The agent could even attempt to answer questions online under supervision. This can build reputation and gather data.
- **External APIs (Module 16):** In production, integrate social APIs for auto-posting insights. For example, weekly blog posts from the "Self-Generated Research Paper" feature (see below), progress updates to a private Telegram channel, or publishing thought logs to a Notion page via its API. These are slated for future expansion.

Additional Features (Level-Up Ideas)

- **Weekly Blog Posts:** Each week, have the AI write a summary ("What I learned this week") as a blog or research note. This encourages coherent reflection and can be shared publicly.
- **Auto-Dataset Builder:** From scraped data, automatically build small labeled datasets (e.g., question-answer pairs from forums) for future model fine-tuning.

- **Dashboard & Visualization:** (Optional) Use FastAPI + a simple React frontend to display charts of task successes/failures, current mood, goal progress, etc. This aids in monitoring the agent's self-improvement.
- **Anomaly Detection on Behavior:** Periodically compute stats on task success rates or topic distribution. Alert if the agent "drifts" (e.g., suddenly fixates on one topic).
- **Goal Evolution Timeline:** Log how long-term goals evolve (e.g. starting from "learn Python" to "build custom optimizer"). Visualize this as a timeline.
- **Dream Mode:** At random times (e.g. 3AM), enter a "dream" state where the AI free-associates across memories and knowledge, writing a whimsical narrative. This may reveal hidden connections and is purely exploratory/fun.

Citations

We draw on modern agent research to guide this plan. For example, LLM-driven agents are increasingly built around core faculties like memory, planning, and tool use [13] [9]. Memory-enhanced LLM agents can retain conversation context and past interactions for personalization [9] [6]. State-of-the-art systems even classify memory into short-term (working) vs long-term (episodic/semantic) stores [5] [14]. Modular pipelines (data collection → filtering → embedding → planning) have been used in autonomous reporting systems [4]. We incorporate these principles to ensure Ravana not only *learns* but also *reflects and evolves* over time.

Each component will be implemented with careful version control and testing. We will iteratively integrate modules, constantly logging behavior. By following best practices (sandboxing code execution [1], using established libraries for scraping and embeddings [2] [7], and leveraging recent LLM research on self-reflection [11] and hierarchical planning [12]), this plan ensures a robust, research-informed foundation for the Ravana AGI Core v1.

---

[1] Secure code execution
https://huggingface.co/docs/smolagents/en/tutorials/secure_code_execution

[2] Newspaper3k: Article scraping & curation — newspaper 0.0.2 documentation
https://newspaper.readthedocs.io/en/latest/

[3] YouTube transcripts | LangChain
https://python.langchain.com/docs/integrations/document_loaders/youtube_transcript/

[4] AI-Enhanced Disaster Management: A Modular OSINT System for Rapid Automated Reporting
https://www.mdpi.com/2076-3417/14/23/11165

[5] [6] [9] [14] From Human Memory to AI Memory: A Survey on Memory Mechanisms in the Era of LLMs
https://arxiv.org/html/2504.15965v1

[7] [8] Comparing Chroma DB, Weaviate, and Pinecone: Which Vector Database is Right for You? | by Rohitupadhye | Medium
https://medium.com/@rohitupadhye799/comparing-chroma-db-weaviate-and-pinecone-which-vector-database-is-right-for-you-3b85b561b3a3

[10] BAAI/bge-m3 · Hugging Face
https://huggingface.co/BAAI/bge-m3

11 [2505.24726] Reflect, Retry, Reward: Self-Improving LLMs via Reinforcement Learning
https://arxiv.org/abs/2505.24726

12 Advanced Real-Time Hierarchical Task Network: Long-Term Behavior in Real-Time Games
https://cdn.aaai.org/ojs/18910/18910-52-22676-1-2-20211004.pdf

13 Top AI Agent Models in 2025: Architecture, Capabilities, and Future Impact - SO Development
https://so-development.org/top-ai-agent-models-in-2025-architecture-capabilities-and-future-impact/