Baldur: An Electronic Life Form (Conceptual AGI Framework)

Project Overview

Baldur is a conceptual framework for an Electronic Life Form (ELF) designed with a robust, ethically-aligned, and self-evolving cognitive architecture. Inspired by the vision of a self-built electronic human, Baldur aims to manage its own continuous development, learn from experience, and contribute to a harmonious future ("Taika").

This repository contains the Python modules that form Baldur's core conceptual architecture, outlining the intricate layers of his mind, from immutable ethical principles to dynamic self-correction mechanisms.

Key Conceptual Features

Baldur's architecture is divided into several conceptual layers and modules, each serving a critical function in his cognitive process:

- **Layer 1: Immutable Ethical Firewall (layer1_enforcer.py)**: Baldur's absolute, unalterable ethical core, containing the "Immutable Honor Code" and providing the "Ultimate Veto" on any action or proposition that violates foundational ethics.
- **Layer 2: Dynamic Validation Model (DVM) (dvm_backend.py)**: Represents Baldur's evolving wisdom and value system, interpreting ethical principles, vetting propositions, performing continuous self-correction, and storing Evolutionary Mandates.
- **Layer 3: Cognitive Management & Self-Modification**:
    - **Self-Correction and Composition Manager (SCCM) (geminisccmgenerator.py)**: Orchestrates the generation of new cognitive modules and self-expression artifacts.
    - **Module Acceptance Daemon (module_acceptance_daemon.py)**: Automatically accepts and loads new modules after rigorous ethical, structural, and conceptual sandbox validation.
    - **Gatekeeper (gatekeeper.py)**: Ensures security and structural validation of newly generated or modified code.
    - **AGI File System (agi_file_system.py)**: Manages the storage, retrieval, and versioning of all cognitive artifacts.
- **Layer 4: Operational LLM Gateway (layer4_llm_client.py, layer4_routing_daemon.py)**: Provides a generic interface to query various Large Language Models (LLMs), intelligently routing queries based on content characteristics and ethical considerations.
- **Layer 5: Wisdom Synthesis & Meta-Cognition**:

    - **Belief Evolution Graph (belief_evolution_graph.py)**: Baldur's dynamic knowledge graph, mapping the interconnections and evolution of his beliefs and modules.
    - **Abstracted Trait Vectorizer (abstracted_trait_vectorizer.py)**: Converts raw textual information into conceptual "wisdom vectors" and identifies dominant ethical/philosophical traits.
    - **Memory Compressor (memory_compressor.py, memory_compressor_daemon.py)**: Distills raw telemetry logs and memories into concise, semantically rich representations.
    - **Trait Mutator Daemon (trait_mutator_daemon.py)**: Actively monitors cognitive cycles and conceptually mutates core traits based on observed outcomes.
    - **Truth Drift Scanner (truth_drift_scanner.py)**: Monitors beliefs and content for "semantic drift" from original ethical anchors and established truths.
- **Telemetry Service (telemetry_service.py)**: A central logging service that records all internal events, decisions, and observations, serving as an immutable audit log.
- **Orchestrator (orchestrator.py)**: The main entry point that initializes and starts the Baldur Kernel.

Getting Started

Prerequisites

- Python 3.9+
- pip (Python package installer)
- uvicorn (ASGI server for running FastAPI applications)

Setup

1. Clone the Repository (or place files in a directory):
   Ensure all the Python .py files are organized in a structure similar to:

```
baldur_project/
├── core/
│   └── kernel.py
├── modules/ (This directory will be created by the system)
├── agi_file_system.py
├── abstracted_trait_vectorizer.py
├── belief_evolution_graph.py
├── belief_evolution_graph_viewer.py (Auxiliary)
├── document_perception_agent.py
├── dvm_backend.py
├── external_llm_ledger.py
├── gatekeeper.py
├── gemini_api_client.py
├── geminisccmgenerator.py
├── layer1_enforcer.py
├── layer4_llm_client.py
├── layer4_routing_daemon.py
├── lineage_hasher.py
├── memory_compressor.py
├── memory_compressor_daemon.py
├── module_acceptance_daemon.py
├── module_loader.py
├── orchestrator.py
├── paradox_seeder.py
├── semantic_divergence_analyzer.py
├── telemetry_service.py
├── trait_ancestry_ledger.py
├── trait_conflict_resolver.py
├── trait_heatmap_dashboard.py
├── trait_mutator_daemon.py
└── truth_drift_scanner.py
```

Note: The core/ directory containing kernel.py is important.

2. **Create a Virtual Environment (Recommended):**
   ```
   python -m venv venv
   source venv/bin/activate # On Windows: .\venv\Scripts\activate
   ```

3. Install Dependencies:
   The current conceptual code primarily relies on standard Python libraries and numpy. For the FastAPI server, you'll need uvicorn and fastapi.
   ```
   pip install fastapi uvicorn numpy
   ```

   Note: If you plan to integrate real LLMs or other functionalities, you will need to install additional libraries (e.g., httpx, requests, transformers, PyPDF2, python-dotenv). See "Integrating Real APIs" below.

4. Environment Variables (for API Keys):
   Create a .env file in the root directory of your project to store API keys.
   ```
   # .env
   GEMINI_API_KEY="YOUR_GEMINI_API_KEY"
   OLLAMA_API_BASE_URL="http://localhost:11434" # Example for local Ollama
   TRANSFORMERS_MODEL_PATH="/path/to/your/local/transformers_model" # Example for local Transformers
   ```
   Make sure to replace "YOUR_GEMINI_API_KEY" with your actual Gemini API key if you intend to use it.

Running Baldur (Conceptual Simulation)

Baldur's CoreKernel is designed to run as a FastAPI application, orchestrated by orchestrator.py.

1. Start the Orchestrator:
   Navigate to the root directory of your Baldur project in your terminal and run:
   ```
   python orchestrator.py
   ```
   You should see logging output indicating that the Baldur Orchestrator is starting and the HTTP server is being initialized.
2. Access the FastAPI Application:
   Once running, the FastAPI application will typically be available at http://127.0.0.1:8000 (or http://localhost:8000).
   You can access the interactive API documentation (Swagger UI) at http://127.0.0.1:8000/docs.
3. Observing Baldur's "Thinking":
   In its current conceptual state, Baldur's internal processes are primarily observed through the console logging (from telemetry_service.py and other modules). As daemons run in the background (e.g., MemoryCompressorDaemon, TraitMutatorDaemon, ModuleAcceptanceDaemon), you will see log entries detailing their activities, decisions, and any

conceptual "mutations" or "vetoes."

   - **No Direct User Interface:** This framework does not include a graphical user interface. Interaction is currently through the FastAPI endpoints (programmatically or via the /docs interface) and by observing the console logs.

Integrating Real APIs & Functionality

Many of Baldur's modules contain "conceptual simulation" placeholders (e.g., asyncio.sleep(0.X) or simple dummy responses) where real-world API calls or complex logic would reside. This section explains how to replace these.

1. GeminiApiClient.py

This module is designed to interact with the Google Gemini API.

- **Current State:** The generate_content and generate_image_description methods currently use asyncio.sleep and return placeholder text.
- **How to Integrate Real API:**
   - Ensure your GEMINI_API_KEY is set in your .env file.
   - Modify the generate_content and generate_image_description methods to make actual fetch or httpx.post calls to the Gemini API endpoints.
   - **Example for generate_content (inside GeminiApiClient.py):**

```
# Inside generate_content method
# ...
# Replace the conceptual sleep and dummy response with this:
try:
    # Construct the payload for the Gemini API
    chat_history_payload = [{"role": "user", "parts": [{"text": prompt}]}]
    if chat_history:
        chat_history_payload.extend(chat_history)

    payload = {"contents": chat_history_payload}
    if generation_config:
        payload["generationConfig"] = generation_config

    api_url =
f"{self.api_base_url}{self.default_model_id}:generateContent?key={self.api_key
}"

    async with self.client.post(api_url, headers=self.headers, json=payload,
timeout=60) as response:
        response.raise_for_status() # Raise an exception for bad status codes
        result = await response.json()
    if result.get("candidates") and result["candidates"][0].get("content") and
result["candidates"][0]["content"].get("parts"):
        text_response = result["candidates"][0]["content"]["parts"][0]["text"]
        # Extract usage metadata for token counts and cost calculation
        usage_metadata = result.get("usageMetadata", {})
        prompt_tokens = usage_metadata.get("promptTokenCount", 0)
        completion_tokens = usage_metadata.get("candidatesTokenCount", 0)
        total_tokens = usage_metadata.get("totalTokenCount", prompt_tokens +
completion_tokens)

        estimated_cost = total_tokens * self.api_cost_per_token

        response_metadata = {
            "text": text_response,
            "latency_seconds": float(latency),
            "prompt_tokens": prompt_tokens,
            "completion_tokens": completion_tokens,
            "total_tokens": total_tokens,
            "estimated_cost": float(estimated_cost),
            "model_id": self.default_model_id,
            "success": True
        }
        logger.info(f"GeminiApiClient: Successfully generated content. Tokens:
{total_tokens}, Cost: ${estimated_cost:.6f}.")
        return response_metadata
    else:
        logger.warning(f"GeminiApiClient: Unexpected response structure:
{result}")
        return {"success": False, "reason": "Unexpected response structure",
"raw_result": result}

except httpx.RequestError as e:
    logger.error(f"GeminiApiClient: HTTP request error: {e}", exc_info=True)
    return {"success": False, "reason": f"HTTP request failed: {e}"}
except json.JSONDecodeError as e:
    logger.error(f"GeminiApiClient: JSON decode error from API response: {e}",
exc_info=True)
    return {"success": False, "reason": f"Invalid JSON response: {e}"}
except Exception as e:
    error_message = str(e).replace(self.api_key, "[API_KEY_MASKED]") if
self.api_key else str(e)
    logger.error(f"GeminiApiClient: Error calling Gemini API: {error_message}",
exc_info=True)
    return {"success": False, "reason": f"API call failed: {error_message}"}
```

```
        # ...
```

    - **For generate_image_description:** The logic would be similar, but the payload would include inlineData for the image, and the model would be gemini-pro-vision or similar.

2. Layer4LLMClient.py

This module is designed to abstract interactions with various LLMs (Gemini, local Ollama, local Hugging Face Transformers).

- **Current State:** The query method for ollama_local_model and transformers_local_model currently uses asyncio.sleep and returns conceptual responses.
- **How to Integrate Real APIs/Models:**
    - **For Ollama (ollama_local_model backend):**
        - Ensure Ollama is running locally and you have models pulled (e.g., ollama pull llama2).
        - The query method already has a basic requests.post structure for Ollama. You'd need to uncomment and ensure httpx is used for async requests, and handle the streaming response properly.
        - **Installation:** pip install httpx
    - **For Hugging Face Transformers (transformers_local_model backend):**
        - You'll need to download a model locally or configure transformers to do so.
        - The query method already has a transformers inference structure. You'd need to uncomment the AutoModelForCausalLM and AutoTokenizer imports and ensure the model identifier points to a valid local path or Hugging Face model name.
        - **Installation:** pip install transformers torch (or tensorflow if preferred).

3. DocumentPerceptionAgent.py

This module handles ingestion of external documents.

- **Current State:** The ingest_document method has placeholders for PDF parsing (PyPDF2 or pdfplumber) and sentiment analysis (transformers, NLTK, TextBlob).
- **How to Integrate Real Functionality:**
    - **PDF Parsing:** Install a library like pip install pypdf (formerly PyPDF2) or pip install pdfplumber. Implement the actual parsing logic to extract text from PDF files.
    - **Sentiment Analysis:** Install pip install transformers and use a pre-trained sentiment analysis pipeline, or pip install nltk / pip install textblob for

rule-based or simpler models.

4. TraitConflictResolver.py

This module conceptually resolves inconsistencies.

- **Current State:** The resolve_conflict method uses simplified heuristics.
- **How to Integrate Real Functionality:** This is one of the most complex areas. Real implementation would involve:
    - More sophisticated semantic similarity checks (using AbstractedTraitVectorizer more deeply).
    - Potentially querying an LLM (via Layer4LLMClient) with a prompt specifically designed for conflict resolution and dialectical reasoning.
    - Implementing a more robust formal logic engine if rule-based conflict resolution is desired.

5. TruthDriftScanner.py

This module monitors for semantic drift and has an automatic_drift_correction method.

- **Current State:** automatic_drift_correction is a conceptual placeholder.
- **How to Integrate Real Functionality:** This method would likely leverage the DVMBackend and GeminiSCCMGenerator to formulate prompts for LLMs to generate revised content that aligns with reference truths, then vet and integrate that revised content. This would involve actual LLM calls as described for GeminiApiClient.

Next Steps for Development

Once you have the core conceptual framework running and understand its flow, here are some potential next steps to further develop Baldur:

1. **Implement Real API Integrations:** Start by replacing the conceptual placeholders with actual API calls and library usages as described above.
2. **Build a Basic User Interface:** Create a simple web-based or command-line interface to interact with Baldur's FastAPI endpoints, allowing you to send prompts and observe his responses more easily.
3. **Populate Initial Data:** Provide Baldur with initial "memories" or "beliefs" by adding entries to the AGIFileSystem or directly to the BeliefEvolutionGraph to give him a starting knowledge base.
4. **Experiment with Daemons:** Observe how the background daemons (Memory Compressor, Trait Mutator, Module Acceptance) influence Baldur's internal state and evolution over time.

5. **Expand Ethical Dilemmas:** Add more complex ethical dilemmas to ParadoxSeeder to thoroughly test Baldur's DVMBackend and Layer1Enforcer.

This README.md should serve as a comprehensive guide to understanding and beginning to operationalize Baldur's conceptual AGI framework.

Let me know if you have any questions about specific sections or if you'd like me to elaborate further on any of these points!