Selected files

1 printable files

GEMINI.md

GEMINI.md

Gemini Workspace Context

This document provides context for the Gemini agent to understand and effectively work with this project.

Project Overview

This project is a sophisticated personal AI assistant powered by the **Agno Framework** with native MCP integration, semantic memory management, and local Ollama AI. It leverages `agno`, `Ollama`, and `LightRAG` to create a powerful, locally-run AI assistant with memory and other capabilities. The agent is built with a modular architecture, allowing for different configurations and functionalities. It originally started with `LangChain`, evolved to use `SmolAgent` and now is entirely focused on using the `agno` agentic framework. The other implementations are legacy.

Personal Files

Files specific to the user, such as `eric_facts.json` and `eric_structured_facts.txt`, are now located in the `eric/` directory at the project root.

Prerequisites

- **Python**: 3.11 or higher
- **Poetry**: For dependency management
- **Docker**: For optional Weaviate database (if using vector storage) and LightRAG server
- **Ollama**: For local LLM inference
- **Node.js**: For MCP servers

LightRAG Server

The LightRAG server must be running for the agent to function correctly. It is managed via Docker Compose. To start the LightRAG services, navigate to the project root and run:

```
./restart-lightrag.sh
```

This script uses the `docker-compose.yml` file in the project root to bring up the necessary services.

Installation and Dependencies

This is a Python project managed with `Poetry`.

1. **Clone and Setup**

   ```
   git clone <repository-url> # Replace with actual repository URL
   cd personal_agent
   poetry install
   ```

2. **Start LightRAG Server**

   ```
   docker-compose up -d
   ```

3. **Manage MCP & Ollama Servers**

   Use the provided scripts to manage your MCP and Ollama servers:

   ```
   # Switch to local Ollama server
   ./switch-ollama.sh local

   # Switch to remote Ollama server
   ./switch-ollama.sh remote

   # Check server status
   ./switch-ollama.sh status
   ```

4. **Setup Ollama**

   ```
   # Install Ollama (macOS example)
   brew install ollama

   # Start Ollama service
   ollama serve -d

   # Pull recommended models
   ollama pull qwen2.5:7b-instruct
   ollama pull qwen3:1.7B
   ollama pull qwen3:8b
   ollama pull llama3.1:8b
   ollama pull nomic-embed-text
   ```

5. **Configure Environment**

   Copy `env.example` to `.env` and configure:

   ```
   # Required: Filesystem paths
   ROOT_DIR=/Users/your_username
   DATA_DIR=/Users/your_username/data

   # Required: Ollama Configuration
   OLLAMA_URL=http://localhost:11434
   ```

```
OLLAMA_DOCKER_URL=http://host.docker.internal:11434

# LightRAG Storage Directories
LIGHTRAG_STORAGE_DIR=${DATA_DIR}/${STORAGE_BACKEND}/${USER_ID}/rag_storage
LIGHTRAG_INPUTS_DIR=${DATA_DIR}/${STORAGE_BACKEND}/${USER_ID}/inputs
LIGHTRAG_MEMORY_STORAGE_DIR=${DATA_DIR}/${STORAGE_BACKEND}/${USER_ID}/memory_rag_storage
LIGHTRAG_MEMORY_INPUTS_DIR=${DATA_DIR}/${STORAGE_BACKEND}/${USER_ID}/memory_inputs

# Optional: API keys for enhanced functionality
GITHUB_PERSONAL_ACCESS_TOKEN=your_token_here
BRAVE_API_KEY=your_api_key_here
```

Running the Agent

The project includes several entry points for running different agent configurations:

Legacy - do not worry about these

- **LangChain Agent**: `poetry run personal-agent-langchain`
- **Smol-Agent**: `poetry run personal-agent-smolagent`

Current

- **Agno-Interface (Streamlit Web UI)**: `poetry run paga_streamlit` or `poetry run paga`
- **Agno-Cli**: The CLI has been refactored for improved maintainability and organization. The main entry point remains `poetry run paga_cli`, but its internal structure has been modularized. See ADR-008 for details.
  - **Usage**: The CLI usage remains identical to before the refactor.

```
# CLI usage remains identical
poetry run paga_cli
poetry run paga_cli --remote
poetry run paga_cli --recreate
```

Initialization with `--recreate` flag

When the `--recreate` flag is used during agent initialization (e.g., `poetry run paga_cli --recreate` or `poetry run paga_streamlit --recreate`), the system will now automatically clear all existing memories from both the local SQLite database and the LightRAG graph memory server. This ensures a clean slate for the knowledge base and memory system, which is particularly useful for development and testing scenarios.

Testing

The project uses a custom testing setup.

- **Run validation tests**:

```
python3 tests/run_validation_test.py
```

This will run the tests located in `tests/test_pydantic_validation_fix.py`.

- **Run debug tests**:

  ```
  python3 tests/run_debug_test.py
  ```

- **Run fact recall tests**:

  ```
  python3 tests/run_fact_recall_tests.py
  ```

- **Test all functionality**: `poetry run test-tools`
- **Test instruction level performance**: `python tests/test_instruction_level_performance.py`
- **Test MCP servers**: `poetry run test-mcp-servers`
- **Test memory system**: `python memory_tests/test_comprehensive_memory_search.py`
- **Test tool call detection**: `python tests/test_tool_call_detection.py`
- **Test memory synchronization**: `python test_memory_sync_fix.py`

Key Technologies

- **agno**: Core framework for building the agent.
- **Ollama**: For running local language models.
- **LanceDB and SQLite**: For vector storage and memory.
- **LightRAG**: RAG-enhanced KB tool.
- **Poetry**: For dependency management.
- **Streamlit**: For the agent's user interface.
- **MCP (Model Context Protocol)**: For integrated servers and tools.

LightRAG Integration

The project has been updated to integrate with the LightRAG server, a powerful tool for building RAG applications. This integration enhances the agent's knowledge capabilities by providing a robust and scalable solution for managing and querying large document collections.

Key Changes:

- **`GEMINI.md`**: This file has been significantly updated to reflect the integration with `LightRAG` and recent system enhancements. It now includes:
  - A detailed overview of `LightRAG`'s features.
  - Installation and usage instructions for the `lightrag` package.
  - A new section documenting the LightRAG Server API, including its endpoints and key characteristics like authentication and versioning.
  - Updates on the **Enriched Graph Ingestion Pipeline** (ADR-007) and the **LightRAG Timeout Fix** (ADR-006).

- **`pyproject.toml`**: The `lightrag-hku` package has been added as a dependency. This is a crucial change that ensures the `LightRAG` library is installed alongside other project dependencies, making its functionality available to the application.
- **Docker Configuration (`docker-compose.yml`)**: The Docker configuration has been substantially modified to:
    - **Tailor the image for the personal agent:** The `docker-compose.yml` has been updated to create a specialized Docker image that includes all the necessary dependencies and configurations for the personal agent project.
    - **Integrate `LightRAG`:** The Docker build process now correctly installs the `lightrag` library, ensuring it's available within the containerized environment.
- **Memory Sync Fix**: Addressed inconsistencies in memory access between the Streamlit interface and agent tools. Both now use a consistent `SemanticMemoryManager` interface. Introduced memory sync monitoring and repair functionality in the Streamlit UI.
- **New LightRAG Storage Configuration**: Added dedicated environment variables and settings for LightRAG storage and input directories (`LIGHTRAG_STORAGE_DIR`, `LIGHTRAG_INPUTS_DIR`, etc.) for better organization and multi-user support.
- **Enhanced Memory Cleaner**: Modified `memory_cleaner.py` to delete source files from LightRAG memory to prevent rescanning.
    - **Improved Config Display**: Updated the `show-config` tool to display the new LightRAG storage directories.

Tools & Capabilities

Memory Tools

- **store_user_memory**: Store personal information with topic classification, now leveraging an **Enriched Graph Ingestion Pipeline** for more accurate knowledge graph construction (see ADR-007).
- **query_memory**: Search user memories using semantic similarity. Args: `query` (str), `limit` (int, optional). Returns: `str` (found memories or message if none found).
- **get_recent_memories**: Retrieve recent memories by searching all memories and sorting by date. Args: `limit` (int, default: 10). Returns: `str` (recent memories or message if none found).
- **get_all_memories**: Get all user memories. Returns: `str` (all memories or message if none found).
- **get_memories_by_topic**: Get memories by topic without similarity search. Args: `topics` (list[str] or str, optional), `limit` (int, optional). Returns: `str` (found memories or a message if none are found).
- **list_memories**: List all memories in a simple, user-friendly format. Returns: `str` (all memories or message if none found).
- **update_memory**: Update an existing memory. Args: `memory_id` (str), `content` (str), `topics` (list[str] or str, optional). Returns: `str` (success or error message).
- **delete_memory**: Delete a memory from both SQLite and LightRAG systems. It now searches for documents in LightRAG using a filename pattern derived from the `memory_id` and then deletes them. Args: `memory_id` (str). Returns: `str` (success or error message).
- **delete_memories_by_topic**: Delete all memories associated with a specific topic or list of topics. Args: `topics` (list[str] or str). Returns: `str` (success or error message).
- **clear_memories**: Clear all memories for the user from local storage. Returns: `str` (success or error message).
- **clear_all_memories**: Clear all memories from BOTH local SQLite and LightRAG graph systems. Returns: `str` (success or error message).

- **get_memory_stats**: Get memory statistics. Returns: `str` (memory statistics).
- **store_graph_memory**: Store a complex memory in your knowledge graph to capture relationships. Uses file upload approach with enhanced entity and relationship extraction. Combines reliable file upload with advanced NLP processing. Args: `content` (str), `topics` (list[str] or str, optional), `memory_id` (str, optional). Returns: `str` (success or error message).
- **get_memory_graph_labels**: Get the list of all entity and relation labels from the memory graph. Returns: `str` (sorted graph labels).

Knowledge Tools

- **query_knowledge_base**: Unified knowledge base query with intelligent routing. Automatically routes queries between local semantic search and LightRAG based on mode and query characteristics. Supports modes like "local", "global", "hybrid", "mix", "naive", and "auto" (default).
- **query_semantic_knowledge**: DEPRECATED. Search the local semantic knowledge base (SQLite/LanceDB) for specific facts or documents. Args: `query` (str), `limit` (int, default: 5). Returns: `str` (formatted search results or message if none found).

MCP-Powered Tools

- **Filesystem**: File operations with security restrictions
- **GitHub**: Repository search and code analysis
- **Web Search**: Real-time information via DuckDuckGo
- **Puppeteer**: Web content extraction and automation
- **Finance**: Stock analysis with working Yahoo Finance endpoints
- **Python**: Safe code execution for calculations

Built-in Tools

- **DuckDuckGo Search**: Web search and news retrieval
- **Python Execution**: Mathematical calculations and data analysis
- **Shell Commands**: System operations with security restrictions

Utility Scripts

- **Send File to LightRAG**: `python3 scripts/send_to_lightrag.py <file_path>` This script sends a specified file to the LightRAG server's `/documents/file` endpoint for processing.
- **Clear Memory When Ready**: `python3 clear_memory_when_ready.py` This script waits for the LightRAG pipeline to finish processing, then clears both local and graph memories.

Memory System

The agent uses an advanced semantic memory system that:

- **Prevents Duplicates**: Intelligent detection of similar content
- **Classifies Topics**: Automatic categorization (personal_info, work, education, etc.)
- **Enables Search**: Semantic similarity search across all memories

- **Provides Analytics**: Memory statistics and usage patterns

Configuration

Environment Variables

```
# Required
ROOT_DIR="/Users/your_username"         # Home directory access
DATA_DIR="/Users/your_username/data"    # Data storage location

# Optional API Keys
GITHUB_PERSONAL_ACCESS_TOKEN="token"    # GitHub integration
BRAVE_API_KEY="key"                    # Brave search (if using)

# Service URLs (optional overrides)
OLLAMA_URL="http://localhost:11434"    # Ollama server
WEAVIATE_URL="http://localhost:8080"   # Weaviate (if using)
```

Model Configuration

The agent supports dynamic model switching through the web interface:

- **qwen2.5:7b-instruct** (recommended)
- **llama3.1:8b**
- **llama3.2:3b**
- **Any Ollama-compatible model**

Troubleshooting

Common Issues

1. Instruction Level Performance

If you are experiencing slow response times, try changing the instruction level. You can do this by editing the `instruction_level` parameter in `src/personal_agent/core/agno_agent.py`.

2. Ollama Connection Issues

```
# Check if Ollama is running
ollama list

# Start Ollama service
ollama serve
```

```
# Test connection
curl http://localhost:11434/api/tags
```

3. MCP Server Issues

```
# Reinstall MCP servers
poetry run python scripts/install_mcp.py

# Test server availability
poetry run test-mcp-servers
```

4. Memory System Issues

If you encounter issues with the memory system, consider the following:

- **Clear all memories**: Use the `clear_all_memories.py` script for a unified approach to clear both local and graph memories.

  ```
  python3 scripts/clear_all_memories.py --no-confirm
  ```

- **Wait for pipeline idle**: If clearing memories after ingestion, use `clear_memory_when_ready.py` to ensure the LightRAG pipeline is idle.

  ```
  python3 clear_memory_when_ready.py
  ```

- **Test memory functionality**:

  ```
  python memory_tests/test_comprehensive_memory_search.py
  ```

- **Check memory sync status**: In the Streamlit UI, use the "Memory Sync Status" section to check for inconsistencies between local and graph memories and sync them if needed.

5. Tool Call Visibility

If tools are being called but not visible in debug panels:

- Check that you're using the latest version of the agent
- Verify tool call detection is working: `python tests/test_tool_call_detection.py`
- Review debug information in the Streamlit interface

Project Structure

```
personal_agent/
├── src/personal_agent/
│   ├── cli/                  # CLI commands and parsing
│   ├── core/                 # Core agent, memory systems, and initialization
│   ├── tools/               # Tool implementations
│   ├── config/              # Configuration management
```

```
│   └── web/                  # Web interface
├── tools/                    # Standalone tools and utilities
├── scripts/                  # Installation and utility scripts
├── memory_tests/            # Memory system tests
├── examples/                # Usage examples
├── docs/                    # Documentation
├── eric/                    # Personal files (e.g., eric_facts.json)
├── old/                     # Legacy or deprecated files
└── tests/                   # Test scripts and test files
```

LightRAG API Endpoints

Here's a summary of the LightRAG API endpoints for reference:

documents

- `POST /documents/scan` - Scan For New Documents
- `POST /documents/upload` - Upload To Input Dir
- `POST /documents/text` - Insert Text
- `POST /documents/texts` - Insert Texts
- `DELETE /documents` - Clear Documents
- `GET /documents` - Documents
- `GET /documents/pipeline_status` - Get Pipeline Status
- `DELETE /documents/delete_document` - Delete a document and all its associated data by its ID.
- `POST /documents/clear_cache` - Clear Cache
- `DELETE /documents/delete_entity` - Delete Entity
- `DELETE /documents/delete_relation` - Delete Relation

query

- `POST /query` - Query Text
- `POST /query/stream` - Query Text Stream

graph

- `GET /graph/label/list` - Get Graph Labels
- `GET /graphs` - Get Knowledge Graph
- `GET /graph/entity/exists` - Check Entity Exists
- `POST /graph/entity/edit` - Update Entity
- `POST /graph/relation/edit` - Update Relation

ollama

- `GET /api/version` - Get Version
- `GET /api/tags` - Get Tags
- `GET /api/ps` - Get Running Models
- `POST /api/generate` - Generate
- `POST /api/chat` - Chat

default

- `GET /` - Redirect To Webui
- `GET /auth-status` - Get Auth Status
- `POST /login` - Login
- `GET /health` - Get Status

Schemas

- `Body_login_login_post`
- `Body_upload_to_input_dir_documents_upload_post`
- `ClearCacheRequest`
- `ClearCacheResponse`
- `ClearDocumentsResponse`
- `DeleteDocByIdResponse`
- `DeleteDocRequest`
- `DeleteEntityRequest`
- `DeleteRelationRequest`
- `DeletionResult`
- `DocStatus`
- `DocStatusResponse`
- `DocsStatusesResponse`
- `EntityUpdateRequest`
- `HTTPValidationError`
- `InsertResponse`
- `InsertTextRequest`
- `InsertTextsRequest`
- `PipelineStatusResponse`
- `QueryRequest`
- `QueryResponse`
- `RelationUpdateRequest`
- `ScanResponse`
- `ValidationError`

QueryParam Class

The `QueryParam` class defines the configuration for querying the LightRAG knowledge base.

```
class QueryParam:
    """Configuration parameters for query execution in LightRAG."""

    mode: Literal["local", "global", "hybrid", "naive", "mix", "bypass"] = "global"
    """Specifies the retrieval mode:
    - "local": Focuses on context-dependent information.
    - "global": Utilizes global knowledge.
    - "hybrid": Combines local and global retrieval methods.
    - "naive": Performs a basic search without advanced techniques.
```

```
- "mix": Integrates knowledge graph and vector retrieval.
"""

only_need_context: bool = False
"""If True, only returns the retrieved context without generating a response."""

only_need_prompt: bool = False
"""If True, only returns the generated prompt without producing a response."""

response_type: str = "Multiple Paragraphs"
"""Defines the response format. Examples: 'Multiple Paragraphs', 'Single Paragraph',

stream: bool = False
"""If True, enables streaming output for real-time responses."""

top_k: int = int(os.getenv("TOP_K", "60"))
"""Number of top items to retrieve. Represents entities in 'local' mode and relation

max_token_for_text_unit: int = int(os.getenv("MAX_TOKEN_TEXT_CHUNK", "4000"))
"""Maximum number of tokens allowed for each retrieved text chunk."""

max_token_for_global_context: int = int(
    os.getenv("MAX_TOKEN_RELATION_DESC", "4000")
)
"""Maximum number of tokens allocated for relationship descriptions in global retrie

max_token_for_local_context: int = int(os.getenv("MAX_TOKEN_ENTITY_DESC", "4000"))
"""Maximum number of tokens allocated for entity descriptions in local retrieval."""

conversation_history: list[dict[str, str]] = field(default_factory=list)
"""Stores past conversation history to maintain context.
Format: [{"role": "user/assistant", "content": "message"}].
"""

history_turns: int = 3
"""Number of complete conversation turns (user-assistant pairs) to consider in the r

ids: list[str] | None = None
"""List of ids to filter the results."""

model_func: Callable[..., object] | None = None
"""Optional override for the LLM model function to use for this specific query.
If provided, this will be used instead of the global model function.
This allows using different models for different query modes.
"""

user_prompt: str | None = None
"""User-provided prompt for the query.
If proivded, this will be use instead of the default vaulue from prompt template.
"""
```