deep-recall Technical Requirements & Specifications

1. Overview

RecallChain is an open-source, scalable hyper personalized memory framework designed to enhance the capability of open-source Large Language Models (LLMs) like DeepSeek R1. It stores, retrieves, and integrates past user interactions to produce contextually relevant and personalized outputs.

2. System Architecture

Core Components:

Memory Service

- **Storage**: Handles structured storage of conversation data, embeddings, metadata including user IDs, timestamps, session IDs, and conversation tags using relational databases (e.g., PostgreSQL).

- **Vector Embedding Management**: Generates and manages embeddings in real-time using state-of-the-art embedding models such as SentenceTransformers, Hugging Face Transformers, and OpenAI-compatible models.

- **Vector Database Integration**: Utilizes high-performance vector databases (FAISS, Qdrant, Milvus, Chroma) to efficiently store embeddings, supporting scalable indexing and querying.

- **Semantic Search**: Implements efficient top-k retrieval using cosine similarity or other vector distance metrics. Configurable parameters include adjustable similarity thresholds, result pagination, and caching mechanisms for frequent queries.

- **Index Management**: Periodic optimization of indices to maintain high performance. Implements incremental indexing and efficient data ingestion pipelines.

- **Backup and Recovery**: Scheduled automated backups and recovery mechanisms ensuring data integrity, with version control and snapshot capabilities for audit and rollback purposes.

Inference Service

- **Model Hosting**: Hosts open-source LLMs such as DeepSeek R1 and LLaMA models, leveraging PyTorch for efficient GPU-accelerated inference.

- **GPU Optimization**: Supports CUDA-enabled GPU inference utilizing techniques like mixed-precision (FP16), dynamic batching, and model parallelism for optimal performance.

- **Model Deployment**: Containerized deployment using Docker, optimized for rapid scaling in Kubernetes clusters with GPU nodes.

- **API Integration**: Provides robust RESTful and/or gRPC API endpoints for handling inference requests and generating structured JSON responses, including detailed error handling and logging.

- **Scalability and Availability**: Employs Kubernetes Horizontal Pod Autoscaler (HPA) based on GPU utilization, inference latency, and request queue metrics for automatic scaling. Ensures high availability and failover capabilities.

- **Health Monitoring**: Real-time monitoring of model performance, GPU utilization, latency, and health status using Prometheus, Grafana, and Loguru with alerting mechanisms for quick incident response.

Orchestrator/API Gateway

- **Request Routing**: Efficiently manages and routes incoming user requests to appropriate memory retrieval and inference services. Implements load balancing and traffic management strategies.

- **Context Aggregation**: Aggregates context data retrieved from the Memory Service to create comprehensive, contextually enriched prompts for LLM inference.

- **Response Handling**: Manages and consolidates responses from inference services, ensuring correct sequencing, response formatting, and timely delivery to end-users.

- **API Management**: Exposes well-documented, standardized RESTful and/or gRPC APIs adhering to OpenAPI specifications for easy integration with external applications and front-end systems.

- **Security Enforcement**: Implements authentication (OAuth/JWT) and authorization mechanisms to secure access to API endpoints. Manages encryption for data in transit.

- **Logging and Monitoring**: Captures detailed logs of all incoming and outgoing traffic, errors, and system performance metrics. Integrates monitoring tools such as Prometheus, Grafana, and OpenTelemetry for observability.

Data Flow:

1. **User Request**: User submits a request through an external interface (web or mobile application).

2. **API Gateway**: The gateway validates, authenticates, and authorizes the incoming request, performs initial parsing and routing based on request metadata.

3. **Memory Retrieval**: Gateway forwards the request to the Memory Service, which retrieves relevant past interactions by executing semantic similarity searches in the vector database to find the most contextually related conversation embeddings.

4. **Context Aggregation**: Relevant retrieved historical interactions and metadata are aggregated into a cohesive context payload, optimized and formatted appropriately for model input, respecting token limitations.

5. **LLM Inference**: Aggregated context and user query are submitted to the Inference Service, where the selected LLM processes the input through GPU-accelerated inference to generate a personalized, context-aware response.

6. **Response**: Generated responses from the LLM are returned to the Orchestrator, which formats and structures the final response payload, including metadata such as confidence scores and token usage.

7. **User**: The structured response is returned to the end-user via the original interface, completing the interaction cycle.

3. Functional Requirements

3.1 Memory Storage and Retrieval

Data Storage

- Store complete conversation data, including user inputs and assistant responses, in a structured relational database (e.g., PostgreSQL).
- Maintain metadata for each interaction, including timestamps, unique user IDs, session or conversation IDs, conversation tags, and additional contextual information.
- Implement efficient schema design to facilitate rapid querying and retrieval of conversation records.

Embedding Generation

- Perform real-time generation of embeddings using transformer-based models such as SentenceTransformers (e.g., all-MiniLM-L6-v2, all-MPNet-base-v2), Hugging Face models, or OpenAI-compatible embedding APIs.
- Optimize embedding pipelines for low latency and high throughput.
- Support asynchronous embedding generation to ensure non-blocking user interactions.

Vector Database Integration

- Leverage vector databases such as FAISS, Milvus, Qdrant, or Chroma for efficient embedding storage.
- Implement scalable, distributed storage solutions to handle high volumes of embeddings and ensure rapid query response times.
- Utilize robust indexing methods such as IVF (Inverted File Index), HNSW (Hierarchical Navigable Small World Graphs), or similar algorithms optimized for nearest-neighbor searches.

Semantic Similarity Search

- Enable configurable semantic similarity searches with adjustable parameters for similarity thresholds and top-k retrieval.
- Utilize cosine similarity or alternative metrics such as Euclidean distance or dot-product similarity based on use case requirements.
- Implement pagination and filtering options to efficiently manage and navigate through large search results.

Scalability and Indexing

- Automate incremental indexing processes to continuously integrate new embeddings.
- Schedule regular index optimization to maintain search performance and efficiency.
- Ensure horizontal scalability by supporting clustering and sharding mechanisms within the vector database.

Caching Layer

- Optionally implement a caching layer using Redis or Memcached to accelerate frequent queries and reduce database load.
- Configure caching mechanisms with appropriate eviction strategies to manage cache size and data freshness effectively.

Backup and Recovery

- Schedule automated backups of conversation data, embeddings, and metadata at regular intervals.
- Implement data version control and snapshot capabilities for comprehensive auditing and rollback capabilities.
- Develop robust recovery protocols to ensure minimal downtime and rapid restoration of data integrity following failures.

3.2 LLM Integration

LLM Models Supported

- Integrate open-source LLMs including but not limited to DeepSeek R1, various LLaMA variants, and Hugging Face-compatible models.
- Ensure compatibility with Hugging Face Transformers library for streamlined integration and regular updates.

Model Deployment

- Containerize LLMs using Docker, optimized for quick deployments and scaling.
- Deploy LLM models on Kubernetes clusters with GPU-enabled nodes to facilitate efficient horizontal scaling.
- Maintain image repositories and versioning for LLM containers to enable rollback and seamless updates.

Inference Engine

- Leverage PyTorch framework for GPU-accelerated inference, ensuring optimal hardware utilization.
- Employ mixed-precision (FP16) inference to enhance performance and reduce memory footprint.
- Implement dynamic batching to efficiently process concurrent inference requests.
- Utilize model parallelism techniques for very large models to ensure maximum resource utilization.

API Endpoints

- Provide robust RESTful and gRPC endpoints designed for high throughput and low latency inference requests.
- Ensure APIs accept structured input payloads specifying queries, context data, and optional inference parameters (temperature, max tokens, etc.).
- Return responses as structured JSON, detailing generated texts, token usage, confidence scores, and error handling metadata.

Scalable Inference Architecture

- Configure Kubernetes Horizontal Pod Autoscaler (HPA) based on real-time GPU usage metrics, inference latency, and request queue lengths.
- Implement auto-provisioning of GPU resources on cloud infrastructures to dynamically handle varying load conditions.

Health Checks and Monitoring

- Implement comprehensive health-check endpoints to continuously monitor the status of inference pods, GPU utilization, inference times, and error rates.
- Integrate Prometheus and Grafana for real-time metrics collection, monitoring dashboards, and alerts.
- Utilize logging solutions such as Loguru for detailed operational logs and quick troubleshooting.

Continuous Integration and Deployment (CI/CD)

- Develop automated pipelines using GitHub Actions for continuous integration, deployment, and updates.
- Ensure minimal downtime during model updates or maintenance windows.
- Provide robust rollback mechanisms to quickly revert deployments in case of failures.
-

3.3 Personalization & Context Management

Dynamic Selection of Relevant Interactions

- Implement real-time semantic similarity algorithms to dynamically select and retrieve the most relevant historical user interactions.
- Use configurable criteria including interaction recency, relevance scores, and specific metadata tags to prioritize and filter retrieved historical data.

Summarization and Truncation

- Integrate automatic summarization capabilities using transformer-based summarization models (e.g., Hugging Face's BART or T5).
- Support dynamic summarization or truncation of historical conversations to conform to token limitations imposed by LLM input requirements.
- Implement summarization processes asynchronously to ensure minimal impact on user response latency.

User Data Management and Deletion Controls

- Provide comprehensive APIs allowing users to view, manage, and delete their stored interaction data.
- Implement secure, authenticated endpoints enabling users to request immediate or scheduled deletion of their historical data.
- Ensure compliance with data protection regulations (e.g., GDPR, CCPA) through rigorous data management policies and processes.
- Log all data management activities and maintain audit trails for accountability and compliance verification.
-

3.4 Scalability & Deployment

Containerization (Docker)

- Containerize each RecallChain component (Memory Service, Inference Service, Orchestrator/API Gateway) using Docker for consistent and reproducible deployments.
- Provide optimized Dockerfiles for minimal image sizes, rapid startup times, and efficient resource usage.
- Ensure container images include all necessary dependencies, configurations, and runtime scripts for seamless execution.
- Maintain and version control Docker images in secure container registries for easy access and deployment.

Kubernetes Deployment

- Deploy RecallChain services using Kubernetes clusters, supporting both cloud-managed solutions (e.g., Google Kubernetes Engine (GKE), Amazon Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS)) and on-premises Kubernetes.
- Configure Kubernetes manifests using tools such as Helm or Kustomize for declarative and reproducible deployments.
- Set up namespaces, service accounts, resource limits, and role-based access control (RBAC) for secure and organized resource management.

Automated Scaling (Horizontal Pod Autoscaler - HPA)

- Implement Horizontal Pod Autoscaler (HPA) for each RecallChain service based on relevant metrics including CPU utilization, memory consumption, GPU usage, and inference latency.
- Configure custom metrics and thresholds to automatically scale the number of pod replicas, ensuring responsiveness during high load periods and resource efficiency during low traffic intervals.
- Leverage Kubernetes metrics server and Prometheus adapter for reliable and granular scaling decisions.

Database Clustering for Availability

- Deploy relational databases (PostgreSQL) and vector databases (Qdrant, Milvus, Chroma) in high-availability clustered configurations.
- Configure primary-secondary replication or multi-master replication setups to ensure data redundancy, fault tolerance, and consistent availability.
- Implement automated failover and load balancing mechanisms to handle database node failures seamlessly without service disruption.
- Regularly monitor database clusters for performance, latency, and synchronization issues, and establish robust recovery procedures for cluster maintenance and node replacements.
-

3.5 API and Front-end

Well-Documented API Endpoints

- Define comprehensive and clearly structured API endpoints using Swagger/OpenAPI specifications.
- Provide interactive API documentation accessible via Swagger UI, enabling developers to easily test and understand API functionalities.
- Document request and response schemas with detailed descriptions, parameter examples, and expected status codes to ensure clarity and ease of use.

Robust API Integration

- Implement secure and standardized RESTful and gRPC API interfaces for efficient and flexible service communication.
- Ensure robust input validation, request parsing, and error handling across all API endpoints to maintain API reliability and integrity.
- Utilize authentication mechanisms such as OAuth2 or JWT to ensure secure API access and protect sensitive user data.
- Establish consistent response formats, including clearly defined data structures, metadata (timestamps, token usage, confidence scores), and error reporting.
- Support cross-origin resource sharing (CORS) to facilitate smooth integration with web-based frontend applications.
- Implement rate limiting and throttling controls to safeguard API performance and prevent misuse.
- Enable detailed API logging for monitoring usage patterns, debugging issues, and auditing purposes, integrated with monitoring solutions such as Prometheus, Grafana, and OpenTelemetry.

4. Non-Functional Requirements

4.1 Performance

- **Latency Targets**:
  - Memory retrieval: Target response latency of ≤ 300 milliseconds for 95% of requests.
  - LLM inference: Target response latency of ≤ 1 second for simple queries and ≤ 2 seconds for complex, context-rich queries.
- Implement proactive monitoring and optimization strategies to consistently achieve latency targets.

4.2 Security

- **Encryption**:
  - Data encryption at rest using AES-256 encryption standard.
  - Data encryption in transit secured with TLS (Transport Layer Security).

- **Secure Authentication Mechanisms**:
  - OAuth2-based authentication and JWT (JSON Web Token) authorization mechanisms for secure API access.
  - Secure management of environment secrets and API keys through encrypted secret stores or vaults (e.g., HashiCorp Vault).
  - Regular security audits and vulnerability scanning using automated tools.

4.3 Reliability

- **Data Backups**:
  - Automated, regular backups of databases, embeddings, and configuration files.
  - Periodic testing and validation of backup restoration processes.
- **Disaster Recovery**:
  - Clearly defined disaster recovery procedures to minimize downtime, including failover and restoration protocols.
  - Redundant system architecture for immediate fallback and rapid recovery.
- **System Monitoring**:
  - Continuous monitoring with Prometheus and Grafana dashboards to track real-time system performance and health.
  - Configured alerts for anomalies and performance degradation for rapid incident response.

4.4 Extensibility

- **Modular Architecture**:
  - Design RecallChain with modular, loosely-coupled components that enable independent development, testing, and deployment.
- **Plugin-Based Extensions**:
  - Develop plugin interfaces to facilitate easy integration of additional embedding models, databases, or external APIs.
  - Provide clear API contracts and documentation for developers to create custom plugins.

4.5 Maintainability

- **Documentation**:
  - Comprehensive and detailed documentation covering architecture, setup instructions, component APIs, and operational guidelines.
  - Regularly updated documentation accessible through Git repositories or documentation hosting services (e.g., GitHub Pages).
- **Testing**:
  - Extensive unit, integration, and end-to-end testing frameworks to maintain high code quality and reliability.
  - Automated test execution through continuous integration tools.
- **Automated Deployments**:

    ◦ Implement continuous integration and continuous deployment (CI/CD) pipelines using GitHub Actions or similar platforms.
    ◦ Establish automated procedures for deploying, scaling, and updating RecallChain components with minimal downtime.

5. Initial Project File Structure

```
RecallChain/
├── api/
│   ├── endpoints/         # API route handlers and endpoints
│   ├── schemas/           # Data validation schemas
│   └── main.py            # Entry point for FastAPI server
├── memory_service/
│   ├── storage/           # Database interaction and storage management
│   ├── embeddings/        # Embedding generation logic
│   ├── vector_db/         # Vector database integration and querying
│   └── search.py          # Semantic similarity search functionalities
├── inference_service/
│   ├── models/            # LLM model definitions and utilities
│   ├── inference/         # Inference pipeline implementation
│   └── api.py             # API endpoints for inference service
├── orchestrator/
│   ├── aggregator.py      # Context aggregation logic
│   ├── routing.py         # Request routing and response handling
│   └── gateway.py         # API Gateway functionalities
├── configs/
│   ├── config.yaml        # Global configuration file
│   └── logging.conf       # Logging configuration
├── deployments/
│   ├── docker/
│   │   ├── Dockerfile     # Dockerfile definitions for each service
│   │   └── docker-compose.yaml # Docker Compose configuration for local testing
│   └── kubernetes/
│      ├── helm-charts/    # Helm charts for Kubernetes deployments
│      └── kustomize/      # Kustomize configurations
├── tests/
│   ├── unit/              # Unit tests for individual components
│   ├── integration/       # Integration tests across services
│   └── conftest.py        # Pytest configuration and fixtures
├── scripts/
│   └── deployment.sh      # Deployment automation scripts
├── docs/
│   ├── architecture.md    # Detailed architectural documentation
│   ├── api.md             # API documentation
```

```
│   └── setup.md           # Setup instructions and guidelines
├── .github/
│   └── workflows/
│       └── ci-cd.yml       # GitHub Actions CI/CD pipeline definitions
├── requirements.txt         # Project dependencies
├── LICENSE                 # Project license
└── README.md                # Project overview, setup, and usage instructions
```

File and Folder Descriptions

- **api/**: Contains API definitions, endpoint implementations, and request validation schemas using FastAPI.
- **memory_service/**: Implements memory-related functionalities, including data storage, embedding management, vector database operations, and semantic searching.
- **inference_service/**: Manages model loading, inference pipelines, and API endpoints dedicated to processing inference requests.
- **orchestrator/**: Provides core orchestration logic for request routing, context aggregation, and response formatting.
- **configs/**: Stores configuration files for global settings and logging.
- **deployments/**: Contains Docker and Kubernetes configurations for deploying services in various environments.
- **tests/**: Includes comprehensive unit and integration tests to maintain code quality and reliability.
- **scripts/**: Holds automation scripts to streamline deployment processes.
- **docs/**: Provides detailed documentation for project architecture, APIs, and initial setup instructions.
- **.github/**: GitHub Actions workflows for continuous integration and deployment.
- **requirements.txt**: Defines Python dependencies required for the project.
- **LICENSE**: Specifies the project's licensing information.
- **README.md**: Offers an overview of the project, installation, usage, and contributing guidelines.

6. Technical Stack

- Python, FastAPI, PyTorch
- Hugging Face Transformers, SentenceTransformers
- FAISS, Qdrant, Milvus, Chroma
- PostgreSQL, SQLAlchemy
- Docker, Kubernetes
- Prometheus, Grafana, OpenTelemetry
- OAuth2/JWT, TLS
- GitHub Actions