Подробная Стратегия Решения AI Journey Contest 2025 — GigaMemory (Обновленная)

Анализ Задачи и Датасета

Формальная Постановка

Задача **GigaMemory** требует создания модуля долговременной памяти для языковой модели GigaChat Lite, способного:

1. **Запоминать**: Извлекать и сохранять атомарные факты о пользователе из диалогов
2. **Хранить**: Организовывать память в эффективной структуре данных
3. **Извлекать**: Находить релевантную информацию для ответа на вопросы
4. **Отвечать**: Генерировать короткие (≤1 предложение) точные ответы с помощью GigaChat Lite

Входные данные:

- Диалог $D = [(u_1, a_1), (u_2, a_2), \ldots, (u_n, a_n)]$
- u_i — реплика пользователя
- a_i — ответ ассистента
- Q — тестовый вопрос

Процесс:

1. Sequential memory write: `write_to_memory((ui, ai), dialogue_id)` для $i = 1 \ldots n$
2. Query: `answer = answer_to_question(dialogue_id, Q)`

Выход:

- Короткий ответ (максимум 1 предложение)
- Используя GigaChat Lite (40GB модель)

Метрика: Accuracy через LLM-as-judge (семантическое сравнение с ground truth)

Характеристики Датасета

Размер и структура:

- ~300,000 символов (~100,000 токенов) на диалог
- Несколько десятков сессий в каждом диалоге
- Многосессионное взаимодействие

Типы контента:

- Короткие диалоговые фразы ("Привет, меня зовут Иван")
- Длинные запросы ("Перескажи следующую статью...")
- Команды ("Переведи этот текст на английский...")

Особенности:

- Не все сессии содержат информацию о пользователе
- Смешанный источник: реальные логи GigaChat + синтетические диалоги
- Возможны западные культурные референсы
- Вопросы задаются к любой сессии (не только первая/последняя)

Ключевой Challenge: Context overflow — 100K токенов не помещаются в context window модели

Типы Вопросов в Датасете

1. fact_equal_session (Низкая сложность)

Описание: Ответ содержится в одной конкретной сессии

Пример: "Как меня зовут?"

Стратегия:

1. Извлечь атомарные факты из каждой сессии
2. Semantic search по вопросу
3. Найти наиболее релевантную сессию
4. Извлечь прямой ответ

Тип поиска: Single-hop retrieval

2. info_consolidation (Средняя сложность)

Описание: Ответ требует агрегации информации из нескольких сессий

Пример: "Какие домашние животные у меня есть?"

Стратегия:

1. Извлечь все релевантные факты по категории (pets)
2. Multi-session retrieval
3. Агрегировать информацию (список уникальных значений)
4. Сформировать полный ответ

Тип поиска: Multi-hop aggregation

3. info_updating (Высокая сложность)

Описание: Информация обновляется в процессе диалога

Пример: "Я женат?" (в session_3 — девушка, в session_4 — жена)

Стратегия:

1. Temporal ordering фактов
2. Conflict resolution (выбор более позднего факта)
3. Найти последнюю актуальную версию информации
4. Вернуть обновленный ответ

Тип поиска: Temporal reasoning

4. no_info (Критическая сложность)

Описание: Ответа нет в диалоге

Пример: "Сколько мне лет?"

Стратегия:

1. Exhaustive search по всем фактам
2. Если similarity < threshold → ответ "Не знаю"
3. Fallback response
4. Избегать hallucinations

Тип поиска: Negative search with fallback

Архитектура Решения

1. Memory Storage Architecture

Проблема: 100K токенов не помещаются в context window

Решение: Atomic Facts + Vector Embeddings

Структура памяти:

```
memory = {
    dialogue_id: {
        'facts': [
            {
                'fact': "Пользователя зовут Иван",
                'category': 'personal_info',
                'session_id': 'session_1',
                'timestamp': 0,
                'importance': 0.9,
                'entities': ['Иван'],
```

```
                'embedding': np.array([...])
            },
            ...
        ],
        'entity_graph': {
            'nodes': ['Иван', 'Барсик', 'Лайка'],
            'edges': [('Иван', 'владелец', 'Барсик'), ...]
        },
        'session_summaries': {
            'session_1': "Знакомство с Иваном",
            ...
        }
    }
}
```

2. Fact Extraction Module

Промпт для извлечения фактов:

```
Из следующего диалога извлеки все атомарные факты о пользователе.
Атомарный факт — это минимальная единица информации, которая может быть полезна для запом

Диалог:
User: {user_message}
Assistant: {assistant_message}

Формат вывода — JSON:
{
  "facts": [
    {
      "fact": "краткое описание факта",
      "category": "personal_info|preference|context|event|relationship",
      "importance": 0.0-1.0,
      "entities": ["entity1", "entity2"]
    }
  ]
}

Категории:
- personal_info: имя, возраст, профессия, семейное положение
- preference: любит/не любит, предпочитает
- context: работа, учеба, хобби, интересы
- event: события из жизни пользователя
- relationship: связи с другими людьми/объектами

Если новых фактов о пользователе нет, верни пустой список.
```

Пример извлечения:

Вход:

```
User: "Привет, меня зовут Иван. У меня есть кот Барсик."
```

```
Assistant: "Приятно познакомиться, Иван! Расскажи про Барсика."
```

Выход:

```
{
  "facts": [
    {
      "fact": "Пользователя зовут Иван",
      "category": "personal_info",
      "importance": 0.95,
      "entities": ["Иван"]
    },
    {
      "fact": "У пользователя есть кот по имени Барсик",
      "category": "relationship",
      "importance": 0.75,
      "entities": ["Иван", "Барсик", "кот"]
    }
  ]
}
```

3. Retrieval Strategy

A. Hybrid Retrieval (Primary Method)

Шаг 1: Query Analysis

- Определить тип вопроса (who/what/when/where/why)
- Извлечь ключевые entities
- Классифицировать категорию (personal_info/preference/etc)

Шаг 2: Multi-level Retrieval

1. **Semantic Search** (primary):

```
# Encode question
q_embedding = embedder.encode(question)

# Compute cosine similarities
similarities = cosine_similarity(q_embedding, fact_embeddings)

# Top-K retrieval
top_k_indices = np.argsort(similarities)[-K:][::-1]
```

2. **Keyword Matching** (secondary):

```
# Extract entities from question
question_entities = extract_entities(question)

# Match with fact entities
keyword_matches = [
    fact for fact in facts
```

```
    if any(entity in fact['entities'] for entity in question_entities)
]
```

3. **Category Filtering** (tertiary):

```
# Infer category from question type
category = infer_category(question)

# Filter facts by category
category_facts = [
    fact for fact in facts
    if fact['category'] == category
]
```

Шаг 3: Re-ranking

- Combine scores: `final_score = 0.6 * semantic + 0.3 * keyword + 0.1 * category`
- Apply importance weighting: `final_score *= fact['importance']`
- Select top-K (K=5-10) most relevant facts

B. Temporal Reasoning (для info_updating)

```
def resolve_temporal_conflict(facts, question):
    # Group facts by semantic similarity
    fact_groups = group_similar_facts(facts)

    # For each group, select most recent
    resolved_facts = []
    for group in fact_groups:
        most_recent = max(group, key=lambda f: f['timestamp'])
        resolved_facts.append(most_recent)

    return resolved_facts
```

C. Aggregation (для info_consolidation)

```
def aggregate_facts(facts, question):
    # Extract all entities of requested type
    entities = []
    for fact in facts:
        entities.extend(fact['entities'])

    # Deduplicate
    unique_entities = list(set(entities))

    # Format answer
    answer = format_list(unique_entities)
    return answer
```

4. Answer Generation Module

Промпт для GigaChat Lite:

```
Ты — персональный ассистент с долговременной памятью о пользователе.

Известные факты о пользователе:
{formatted_facts}

Вопрос пользователя: {question}

Инструкции:
1. Ответь КРАТКО (одним предложением или несколькими словами)
2. Используй ТОЛЬКО информацию из фактов выше
3. Если информации недостаточно, ответь ТОЧНО: "Не знаю"
4. НЕ придумывай информацию
5. Будь точным и лаконичным

Ответ:
```

Пример контекста для генерации:

```
Известные факты о пользователе:
- Пользователя зовут Иван
- У пользователя есть кот по имени Барсик, ему 2 года
- У пользователя есть собака по имени Лайка
- Пользователь женат

Вопрос пользователя: Какие домашние животные у меня есть?

Ответ: У вас есть кот Барсик и собака Лайка.
```

Post-processing:

1. Удалить лишние вводные фразы
2. Проверить длину (максимум 1-2 предложения)
3. Валидация на наличие фактической информации из memory

5. Технические Оптимизации

A. Lightweight Embedding Model

Выбор: `sentence-transformers/all-MiniLM-L6-v2`

- Размер: ~80MB (fits well within 5GB limit)
- Dimensionality: 384
- Fast CPU inference (~0.01s per embedding)
- Good quality for Russian text

B. Memory Consolidation

Проблема: Накопление дублирующихся/противоречивых фактов

Решение:

```
def consolidate_memory(facts):
    # Group by semantic similarity
    groups = cluster_facts_by_similarity(facts, threshold=0.85)

    consolidated = []
    for group in groups:
        if len(group) == 1:
            consolidated.append(group[^0])
        else:
            # Merge similar facts
            merged_fact = merge_facts(group)
            consolidated.append(merged_fact)

    return consolidated
```

C. Importance Scoring

Heuristics:

- **Frequency**: Факты, упоминаемые несколько раз → выше importance
- **Recency**: Более свежие факты → выше importance
- **Category**: personal_info > relationship > preference > context > event
- **Explicitness**: Прямые утверждения > косвенные упоминания

```
def compute_importance(fact, context):
    base_importance = 0.5

    # Category weight
    category_weights = {
        'personal_info': 0.3,
        'relationship': 0.2,
        'preference': 0.15,
        'context': 0.1,
        'event': 0.05
    }
    base_importance += category_weights.get(fact['category'], 0)

    # Frequency boost
    frequency = count_similar_facts(fact, context)
    base_importance += min(frequency * 0.1, 0.3)

    # Recency decay
    time_decay = np.exp(-0.1 * (current_time - fact['timestamp']))
    base_importance *= time_decay
```

```
    return min(base_importance, 1.0)
```

D. No-Info Detection

Strategy:

```
def detect_no_info(question, retrieved_facts, threshold=0.3):
    if not retrieved_facts:
        return True

    # Check maximum similarity
    max_similarity = max(fact['similarity'] for fact in retrieved_facts)

    if max_similarity &lt; threshold:
        return True

    return False

def generate_answer_with_fallback(question, facts):
    if detect_no_info(question, facts):
        return "Не знаю"

    # Normal generation
    answer = generate_with_gigachat(question, facts)

    # Validation: check if answer contains factual info
    if is_generic_or_evasive(answer):
        return "Не знаю"

    return answer
```

Практическая Реализация

Структура Кода

```
submit/
├── __init__.py
├── model_inference.py          # Main class
├── fact_extractor.py           # Fact extraction logic
├── retriever.py                # Retrieval strategies
├── answer_generator.py         # Prompt construction + generation
├── memory_manager.py           # Memory storage &amp;amp; consolidation
├── utils.py                    # Helper functions
└── models/
    └── all-MiniLM-L6-v2/       # Lightweight embedding model
```

model_inference.py (Core Implementation)

```
from typing import List, Dict
import json
import numpy as np
from sentence_transformers import SentenceTransformer
from transformers import AutoModelForCausalLM, AutoTokenizer
import torch

class SubmitModelWithMemory(ModelWithMemory):
    def __init__(self, model_path: str):
        # Load embedding model (80MB, fits in 5GB)
        self.embedder = SentenceTransformer('all-MiniLM-L6-v2')

        # Load GigaChat Lite (provided in image at model_path)
        self.tokenizer = AutoTokenizer.from_pretrained(model_path)
        self.llm = AutoModelForCausalLM.from_pretrained(
            model_path,
            torch_dtype=torch.bfloat16,
            device_map="auto"
        )

        # Memory storage: {dialogue_id: {...}}
        self.memory = {}

        # Similarity threshold for no-info detection
        self.no_info_threshold = 0.3

    def write_to_memory(self, messages: List[Message], dialogue_id: str) -&gt; None:
        """Extract and store atomic facts from dialogue messages."""
        # Extract facts using LLM
        facts = self._extract_facts(messages)

        # Initialize memory for dialogue if not exists
        if dialogue_id not in self.memory:
            self.memory[dialogue_id] = {
                'facts': [],
                'embeddings': [],
                'session_count': 0
            }

        # Process each fact
        for fact in facts:
            # Generate embedding
            fact_embedding = self.embedder.encode(fact['fact'])

            # Add timestamp
            fact['timestamp'] = self.memory[dialogue_id]['session_count']

            # Compute importance
            fact['importance'] = self._compute_importance(fact)

            # Check for duplicates/conflicts
            if self._is_duplicate(fact, dialogue_id):
                self._update_existing_fact(fact, dialogue_id)
            else:
```

```
                # Add new fact
                self.memory[dialogue_id]['facts'].append(fact)
                self.memory[dialogue_id]['embeddings'].append(fact_embedding)

        # Increment session counter
        self.memory[dialogue_id]['session_count'] += 1

        # Periodic consolidation
        if self.memory[dialogue_id]['session_count'] % 10 == 0:
            self._consolidate_memory(dialogue_id)

    def answer_to_question(self, dialogue_id: str, question: str) -&gt; str:
        """Generate answer to question using memory."""
        # Check if memory exists
        if dialogue_id not in self.memory:
            return "Не знаю"

        # Retrieve relevant facts
        relevant_facts = self._retrieve_facts(dialogue_id, question, top_k=10)

        # Detect no-info case
        if self._detect_no_info(relevant_facts):
            return "Не знаю"

        # Handle different question types
        question_type = self._classify_question_type(question, relevant_facts)

        if question_type == 'info_consolidation':
            # Aggregate multiple facts
            answer = self._aggregate_answer(relevant_facts, question)
        elif question_type == 'info_updating':
            # Resolve temporal conflicts
            resolved_facts = self._resolve_temporal(relevant_facts)
            answer = self._generate_answer(resolved_facts, question)
        else:
            # Standard generation
            answer = self._generate_answer(relevant_facts, question)

        # Post-process
        answer = self._post_process_answer(answer)

        return answer

    def clear_memory(self, dialogue_id: str) -&gt; None:
        """Clear memory for dialogue."""
        if dialogue_id in self.memory:
            del self.memory[dialogue_id]

    # ========== Private Methods ==========

    def _extract_facts(self, messages: List[Message]) -&gt; List[Dict]:
        """Extract atomic facts using LLM."""
        # Construct extraction prompt
        dialogue_text = self._format_messages(messages)

        extraction_prompt = f"""Из следующего диалога извлеки все атомарные факты о польз
```

```
Диалог:
{dialogue_text}

Формат вывода — JSON:
{{
  "facts": [
    {{
      "fact": "краткое описание факта",
      "category": "personal_info|preference|context|event|relationship",
      "entities": ["entity1", "entity2"]
    }}
  ]
}}

Если новых фактов нет, верни пустой список.
JSON:"""

        # Generate with LLM
        inputs = self.tokenizer(extraction_prompt, return_tensors="pt").to(self.llm.devic
        outputs = self.llm.generate(
            **inputs,
            max_new_tokens=512,
            temperature=0.1,
            do_sample=False
        )

        output_text = self.tokenizer.decode(outputs[^0], skip_special_tokens=True)

        # Parse JSON
        try:
            # Extract JSON from output
            json_start = output_text.find('{')
            json_end = output_text.rfind('}') + 1
            json_str = output_text[json_start:json_end]

            result = json.loads(json_str)
            return result.get('facts', [])
        except:
            return []

    def _retrieve_facts(self, dialogue_id: str, question: str, top_k: int = 10) -&gt; Lis
        """Retrieve relevant facts using hybrid search."""
        if not self.memory[dialogue_id]['facts']:
            return []

        # Encode question
        q_embedding = self.embedder.encode(question)

        # Compute semantic similarities
        fact_embeddings = np.array(self.memory[dialogue_id]['embeddings'])
        semantic_scores = np.dot(fact_embeddings, q_embedding) / (
            np.linalg.norm(fact_embeddings, axis=1) * np.linalg.norm(q_embedding)
        )

        # Extract question entities
```

```
        q_entities = self._extract_entities_simple(question)

        # Compute keyword scores
        keyword_scores = []
        for fact in self.memory[dialogue_id]['facts']:
            entity_overlap = len(set(q_entities) &amp;amp; set(fact.get('entities', [])))
            keyword_scores.append(entity_overlap)
        keyword_scores = np.array(keyword_scores)
        if keyword_scores.max() &amp;gt; 0:
            keyword_scores = keyword_scores / keyword_scores.max()

        # Infer category
        category = self._infer_category(question)
        category_scores = np.array([
            1.0 if fact.get('category') == category else 0.5
            for fact in self.memory[dialogue_id]['facts']
        ])

        # Importance scores
        importance_scores = np.array([
            fact.get('importance', 0.5)
            for fact in self.memory[dialogue_id]['facts']
        ])

        # Combined score
        final_scores = (
            0.5 * semantic_scores +
            0.2 * keyword_scores +
            0.1 * category_scores +
            0.2 * importance_scores
        )

        # Top-K
        top_indices = np.argsort(final_scores)[-top_k:][::-1]

        # Return facts with scores
        relevant_facts = []
        for idx in top_indices:
            fact = self.memory[dialogue_id]['facts'][idx].copy()
            fact['similarity'] = final_scores[idx]
            relevant_facts.append(fact)

        return relevant_facts

    def _generate_answer(self, facts: List[Dict], question: str) -&amp;gt; str:
        """Generate answer using GigaChat with facts as context."""
        # Format facts
        facts_text = "\n".join([
            f"- {fact['fact']}" for fact in facts[:5]  # Use top 5
        ])

        # Construct prompt
        prompt = f"""Ты — персональный ассистент с памятью о пользователе.

Известные факты:
{facts_text}
```

```
Вопрос: {question}

Инструкции:
1. Ответь КРАТКО (одно предложение)
2. Используй ТОЛЬКО факты выше
3. Если недостаточно информации, скажи "Не знаю"

Ответ:"""

        # Generate
        inputs = self.tokenizer(prompt, return_tensors="pt").to(self.llm.device)
        outputs = self.llm.generate(
            **inputs,
            max_new_tokens=64,
            temperature=0.0,
            do_sample=False
        )

        answer = self.tokenizer.decode(outputs[^0], skip_special_tokens=True)

        # Extract answer after "Ответ:"
        if "Ответ:" in answer:
            answer = answer.split("Ответ:")[-1].strip()

        return answer

    def _detect_no_info(self, facts: List[Dict]) -&gt; bool:
        """Detect if no relevant information available."""
        if not facts:
            return True

        max_similarity = max(fact.get('similarity', 0) for fact in facts)
        return max_similarity &lt; self.no_info_threshold

    # ... (other helper methods)
```

Оптимизация и Debugging

Локальное Тестирование

Шаг 1: Создание тестовых примеров

```
test_dialogues = [
    {
        'dialogue': [
            ("Привет, меня зовут Иван", "Привет, Иван!"),
            ("У меня кот Барсик", "Расскажи о нем"),
            ("Ему 2 года", "Какой молодой!")
        ],
        'questions': [
            ("Как меня зовут?", "Иван"),
            ("Какие у меня питомцы?", "Кот Барсик"),
            ("Сколько лет Барсику?", "2 года"),
```

```
            ("Сколько мне лет?", "Не знаю")  # no_info
        ]
    }
]
```

Шаг 2: Метрики

```
def evaluate_solution(model, test_dialogues):
    results = []

    for test in test_dialogues:
        dialogue_id = f"test_{test['id']}"

        # Write to memory
        for user_msg, asst_msg in test['dialogue']:
            model.write_to_memory([user_msg, asst_msg], dialogue_id)

        # Test questions
        for question, expected_answer in test['questions']:
            predicted_answer = model.answer_to_question(dialogue_id, question)

            # Compute semantic similarity
            similarity = compute_semantic_similarity(predicted_answer, expected_answer)

            results.append({
                'question': question,
                'expected': expected_answer,
                'predicted': predicted_answer,
                'similarity': similarity
            })

    accuracy = np.mean([r['similarity'] &gt; 0.8 for r in results])
    return accuracy, results
```

Итеративное Улучшение

Baseline → Advanced:

1. **Baseline (60-70% accuracy)**:
   - Simple fact extraction
   - Basic semantic retrieval
   - Standard prompt
2. **+ Category filtering (65-75%)**:
   - Classify question type
   - Filter facts by category
3. **+ Temporal reasoning (70-80%)**:
   - Timestamp tracking
   - Conflict resolution

4. **+ Importance scoring (75-85%)**:
    - Weight facts by importance
    - Prioritize recent/frequent facts
5. **+ Memory consolidation (80-90%)**:
    - Deduplicate similar facts
    - Entity resolution

Timeline и Ресурсы

Phase 1: Setup (1 день)

- Клонировать репозиторий: https://gitverse.ru/ai-forever/memory_aij2025
- Скачать GigaChat-20B (40GB)
- Настроить окружение
- Протестировать baseline

Phase 2: Fact Extraction (2-3 дня)

- Реализовать atomic fact extraction
- Эксперименты с промптами
- Категоризация фактов
- Deduplication

Phase 3: Retrieval (2-3 дня)

- Интегрировать sentence-transformers
- Hybrid retrieval
- Top-K optimization
- Temporal filtering

Phase 4: Prompt Engineering (2 дня)

- Оптимизация промптов для GigaChat
- Формат контекста
- Fallback логика

Phase 5: Optimization (2-3 дня)

- Memory consolidation
- Conflict resolution
- Performance tuning

- Validation testing

Phase 6: Testing (2 дня)

- Comprehensive test suite
- Docker environment validation
- Stress testing
- Final optimization

Общая длительность: 11-14 дней
Дедлайн: 30 октября 2025

Заключение

Успешное решение задачи GigaMemory требует:

1. **Эффективное извлечение фактов**: LLM-based atomic fact extraction с категоризацией
2. **Гибридный retrieval**: Комбинация semantic + keyword + category + importance
3. **Умная обработка типов вопросов**: Специальные стратегии для info_consolidation, info_updating, no_info
4. **Качественные промпты**: Точные инструкции для GigaChat с fallback на "Не знаю"
5. **Оптимизация памяти**: Consolidation, deduplication, temporal reasoning

Ключевые факторы успеха:

- Атомарные факты вместо полного контекста (решение context overflow)
- Hybrid retrieval для высокой precision и recall
- Temporal reasoning для обработки обновлений информации
- Четкие fallback механизмы для избежания hallucinations
- Короткие, фактические ответы (≤1 предложение)

Следуя этой стратегии и итеративно улучшая решение, можно достичь accuracy 80-90% на validation set и занять конкурентоспособную позицию в лидерборде.

Удачи в конкурсе!
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] [18] [19] [20]

⁂

1. https://www.sciencedirect.com/science/article/abs/pii/S0957417419304932
2. https://www.linkedin.com/pulse/comprehensive-guide-chatbot-memory-techniques-ai-cloudkitect-prmqe
3. https://www.themoonlight.io/en/review/fact-in-fragments-deconstructing-complex-claims-via-llm-based-atomic-fact-extraction-and-verification
4. https://arxiv.org/html/2404.00573v1
5. https://www.philschmid.de/gemini-with-memory
6. https://www.nature.com/articles/s41467-024-45563-x

7. https://pmc.ncbi.nlm.nih.gov/articles/PMC7878197/
8. https://www.pinecone.io/learn/series/langchain/langchain-conversational-memory/
9. https://arxiv.org/abs/2506.07446v1
10. https://pmc.ncbi.nlm.nih.gov/articles/PMC4526749/
11. https://aws.amazon.com/blogs/machine-learning/building-smarter-ai-agents-agentcore-long-term-memory-deep-dive/
12. https://openreview.net/pdf/e10b632ae6dc293fa879a911d117ca7bfa694544.pdf
13. https://arxiv.org/abs/2506.07446
14. https://aclanthology.org/P19-1541/
15. https://www.chitika.com/rag-based-chatbot-with-memory/
16. https://www.arxiv.org/pdf/2506.09171.pdf
17. http://arxiv.org/abs/1906.01788
18. https://www.designveloper.com/blog/ai-chatbot-with-memory/
19. https://pmc.ncbi.nlm.nih.gov/articles/PMC11590295/
20. https://arxiv.org/pdf/1906.01788.pdf