ТЕХНИЧЕСКОЕ ЗАДАНИЕ

LitFinder MVP: Полное описание функций и технологий

Версия: 1.0 MVP
Дата: 11 февраля 2026
Статус: Product Requirements Document
Стек: FastAPI, PostgreSQL, Next.js, OpenAlex API, Claude AI
Целевой рынок: Беларусь, СНГ, международные исследователи

---

1. ОБЗОР ПРОДУКТА

1.1. Описание платформы

LitFinder MVP — ИИ-платформа для поиска и управления научной литературой, предназначенная для аспирантов, исследователей и научных руководителей в СНГ[1][2].

Ключевые отличия от конкурентов:

- Фокус на требованиях ВАК РБ и ГОСТ Р 7.0.100-2018
- Интеграция поиска, анализа и форматирования в единой платформе
- Адаптация лучших практик Elicit и Scite AI под СНГ-рынок
- Нативная поддержка русского и белорусского языков

1.2. Технологический стек

Backend:

- FastAPI 0.110+ (Python 3.11+)
- PostgreSQL 15+ с pgvector расширением
- Redis 7+ для кеширования
- Anthropic Claude API (Haiku 4.5, Sonnet 4)
- OpenAI Embeddings API

Frontend:

- Next.js 14+ (App Router)
- React 18+
- TailwindCSS 3+
- shadcn/ui компоненты
- React Query для state management

Инфраструктура:

- Docker + Docker Compose
- Yandex Cloud / DigitalOcean
- Nginx reverse proxy
- GitHub Actions CI/CD

Внешние API:

- OpenAlex Works API
- Semantic Scholar API (опционально)
- CyberLeninka OAI-PMH (фаза 2)

---

2. FEATURE 1: СЕМАНТИЧЕСКИЙ ПОИСК И RESEARCH ASSISTANT

2.1. Описание функции

Позволяет пользователям задавать исследовательские вопросы на естественном языке и получать:

1. Текстовый ответ от ИИ с цитатами из научных работ
2. Список релевантных статей с метаданными
3. Возможность сохранения источников в подборки

Вдохновлено: Elicit "Research Question" и Scite Assistant[3][4].

2.2. Техническая реализация

Архитектура (RAG - Retrieval Augmented Generation):

Semantic Search Flow
User Query → Embedding → Vector Search (OpenAlex + Local DB) →

LLM Context → Answer + Sources

Компоненты:

1) Query Processing Service

Endpoint: POST /api/research/answer

class ResearchAnswerRequest(BaseModel):
question: str
filters: Optional[SearchFilters]
max_sources: int = 10

class SearchFilters(BaseModel):
year_from: Optional[int]
year_to: Optional[int]
types: List[str]
languages: List[str]

2) Embedding Generation

- Используется OpenAI text-embedding-3-small (1536 dimensions)
- Кеширование embeddings в Redis (TTL 7 дней)
- Fallback на локальную модель (sentence-transformers) при недоступности API

async def generate_embedding(text: str) -> List[float]:
cache_key = f"emb:{hash(text)}"
cached = await redis.get(cache_key)
if cached:
return json.loads(cached)

```
embedding = await openai.embeddings.create(
    model="text-embedding-3-small",
    input=text
)
```

```
    await redis.setex(cache_key, 604800, json.dumps(embedding))
    return embedding.data[0].embedding
```

3) Vector Search in OpenAlex

- Hybrid search: semantic (embeddings) + keyword (BM25)
- Используется OpenAlex search parameter с фильтрами[5]
- Результаты ранжируются по релевантности + citation count

async def search_openalex(
query_embedding: List[float],
filters: SearchFilters,
limit: int = 20
) -> List[Work]:
# Semantic search через OpenAlex API
params = {
"search": query_text,
"filter": build_filters(filters),
"per-page": limit,
"sort": "cited_by_count:desc"
}

```
    response = await httpx.get(
        "https://api.openalex.org/works",
        params=params
    )

    works = normalize_openalex_works(response.json())
    return works
```

4) LLM Answer Generation

- Claude Sonnet 4 для генерации ответов (context window 200K)
- RAG prompt с инструкциями цитирования источников
- Структурированный output с JSON schema

async def generate_answer(
question: str,
sources: List[Work]

) -> ResearchAnswer:
context = build_context_from_sources(sources)

```
prompt = f"""
```

Ты — эксперт-исследователь. Ответь на вопрос, используя ТОЛЬКО
информацию из предоставленных научных работ.

Вопрос: {question}

Источники:
{context}

Требования:

1. Ответ должен быть объективным и научным
2. ОБЯЗАТЕЛЬНО цитируй источники в формате [ID]
3. Если информации недостаточно, скажи об этом
4. Длина ответа: 300-800 слов
   """
   response = await claude.messages.create(
   model="claude-sonnet-4-20250514",
   max_tokens=2000,
   temperature=0.3,
   messages=[{"role": "user", "content": prompt}]
)
   return parse_research_answer(response.content)

5) Response Structure

class ResearchAnswer(BaseModel):
answer_text: str
sources_used: List[SourceCitation]
all_sources: List[WorkMetadata]
confidence: float
limitations: Optional[str]

class SourceCitation(BaseModel):
work_id: str

quote: str
relevance_score: float

2.3. UI/UX компоненты

Frontend (Next.js):

- Страница /research с крупной поисковой строкой
- Блок ответа ИИ с подсветкой цитат (кликабельные [1], [2])
- Таблица источников с чекбоксами для добавления в подборки
- Фильтры в боковой панели (год, тип, язык)

Key interactions:

1. User вводит вопрос → нажимает "Найти ответ"
2. Loading state с индикатором прогресса
3. Ответ появляется постепенно (streaming, если поддерживается)
4. Клик по цитате [N] → скролл к источнику N
5. Hover над источником → превью abstract
6. Чекбокс → "Добавить в подборку" modal

2.4. Метрики успеха

- P95 latency ≤ 8 секунд (search + LLM)
- ≥70% пользователей используют функцию минимум 1 раз
- ≥40% сохраняют хотя бы 1 источник из результатов
- User satisfaction ≥4.0/5.0 по опросу

---

3. FEATURE 2: ТАБЛИЧНАЯ ЭКСТРАКЦИЯ ДАННЫХ (DATA EXTRACTION)

3.1. Описание функции

Автоматическое извлечение структурированных данных из научных статей в формате таблицы, где:

- Строки = статьи из подборки/результатов поиска

- Колонки = пользовательские поля (sample size, методология, результаты)
- Ячейки = извлеченные значения + цитаты из текста

Вдохновлено: Elicit Table View[6][7].

3.2. Техническая реализация

Архитектура:

Extraction Pipeline
User defines fields → For each work: fetch text → LLM extraction → Aggregate results → Display table

Компоненты:

1) Extraction Job Service

Endpoint: POST /api/extraction/run

class ExtractionJobRequest(BaseModel):
work_ids: List[str]
fields: List[ExtractionField]
collection_id: Optional[str]

class ExtractionField(BaseModel):
name: str
description: str
data_type: Literal["text", "number", "boolean", "list"]

2) Text Retrieval

- Приоритет 1: Abstract из OpenAlex (всегда доступен)
- Приоритет 2: Fulltext из open access источников (если is_oa=true)
- Приоритет 3: Cited excerpts из других работ

async def get_work_text(work_id: str) -> WorkText:
work = await openalex.get_work(work_id)

```
text_parts = []

# Abstract (inverted index → plain text)
if work.abstract_inverted_index:
    abstract = reconstruct_abstract(
        work.abstract_inverted_index
    )
    text_parts.append(("abstract", abstract))

# Fulltext (если open access)
if work.open_access.is_oa and work.open_access.oa_url:
    fulltext = await fetch_fulltext(work.open_access.oa_url)
    text_parts.append(("fulltext", fulltext))

return WorkText(work_id=work_id, parts=text_parts)
```

3) LLM Extraction (Parallel Processing)

- Claude Haiku 4.5 для быстрой экстракции (500ms per work)
- Batch processing: до 5 работ параллельно
- Structured output с JSON schema validation

async def extract_field_value(
work_text: WorkText,
field: ExtractionField
) -> ExtractionResult:

```
prompt = f"""
```

Извлеки из научного текста значение поля "{field.name}".

Описание поля: {field.description}
Тип данных: {field.data_type}

Текст работы:
{work_text.combined_text}

Верни JSON:
{{
"value": <extracted_value>,
"quote": "<цитата из текста, подтверждающая значение>",
"confidence": <0.0-1.0>,
"not_found": <true если информации нет>
}}
"""

```
response = await claude.messages.create(
    model="claude-3-5-haiku-20241022",
    max_tokens=500,
    temperature=0.2,
    messages=[{"role": "user", "content": prompt}],
    response_format={"type": "json_object"}
)

return parse_extraction_result(response.content)
```

4) Batch Coordinator

async def run_extraction_job(
job_request: ExtractionJobRequest
) -> ExtractionJob:
job_id = str(uuid.uuid4())

```
# Создать запись в БД
job = await db.create_extraction_job(
    id=job_id,
    work_ids=job_request.work_ids,
    fields=job_request.fields,
    status="running"
)

# Запустить фоновую задачу
asyncio.create_task(
    process_extraction_job(job_id)
)
```

```
return job
```

async def process_extraction_job(job_id: str):
job = await db.get_extraction_job(job_id)

```
results = []

# Параллельная обработка (5 workers)
async with asyncio.TaskGroup() as tg:
    for work_id in job.work_ids:
        for field in job.fields:
            task = tg.create_task(
                extract_single_cell(work_id, field)
            )
            results.append(task)

# Агрегировать результаты
table_data = aggregate_results(results)

# Обновить статус
await db.update_extraction_job(
    job_id,
    status="completed",
    results=table_data
)
```

5) Data Storage

Таблица	**Назначение**
extraction_jobs	Метаданные задачи
extraction_results	Результаты по ячейкам
extraction_cache	Кеш извлечений

CREATE TABLE extraction_jobs (
id UUID PRIMARY KEY,

```
user_id UUID REFERENCES users(id),
collection_id UUID REFERENCES collections(id),
work_ids TEXT[],
fields JSONB,
status VARCHAR(20),
created_at TIMESTAMP,
completed_at TIMESTAMP
);

CREATE TABLE extraction_results (
id UUID PRIMARY KEY,
job_id UUID REFERENCES extraction_jobs(id),
work_id TEXT,
field_name VARCHAR(255),
value TEXT,
quote TEXT,
confidence FLOAT,
created_at TIMESTAMP,
INDEX idx_job_work (job_id, work_id)
);
```

3.3. UI/UX компоненты

Frontend:

- Modal "Экстракция данных" из контекстного меню подборки
- Форма определения полей (название + описание + тип)
- Прогресс-бар с количеством обработанных работ
- Интерактивная таблица с сортировкой/фильтрацией
- Tooltip с цитатой при hover над ячейкой
- Экспорт в CSV/Excel

Key interactions:

1. User открывает подборку → "Экстракция данных"
2. Добавляет поля: "Sample size", "Методология", "Главный результат"
3. Нажимает "Запустить экстракцию"
4. Прогресс: "Обработано 15/40 работ..."
5. Результат: таблица с заполненными ячейками
6. Hover → tooltip с цитатой

7. Экспорт → скачивание Excel файла

3.4. Оптимизация и кеширование

- Кеширование извлечений в Redis (TTL 30 дней)
- Переиспользование результатов для одинаковых полей
- Rate limiting: максимум 3 активных задачи на пользователя
- Приоритизация: платные пользователи обрабатываются первыми

3.5. Метрики успеха

- Точность извлечения ≥75% (валидация экспертами)
- P95 время обработки ≤30 секунд на 20 работ
- ≥30% пользователей с подборками используют функцию
- ≥60% завершенных экстракций экспортируются

---

4. FEATURE 3: ЧАТ С КОЛЛЕКЦИЕЙ (CHAT WITH LIBRARY)

4.1. Описание функции

Интерактивный чат-интерфейс для диалога с собранной библиотекой источников. Пользователи могут:

- Задавать вопросы о содержании статей
- Сравнивать методологии и результаты
- Получать обобщения и выводы
- Находить противоречия между работами

Вдохновлено: Elicit "Chat with papers", Scite Assistant[8][9].

4.2. Техническая реализация

Архитектура (Conversational RAG):

Chat Flow
User message → Retrieve relevant chunks → Build context → LLM response → Store in history

Компоненты:

1) Chat Session Management

Endpoint: POST /api/chat/collection

class ChatMessageRequest(BaseModel):
collection_id: str
message: str
session_id: Optional[str]

class ChatSession(BaseModel):
id: str
collection_id: str
messages: List[ChatMessage]
created_at: datetime

2) Document Chunking and Indexing

При добавлении работы в подборку:

- Текст разбивается на chunks (512 tokens с overlap 50)
- Для каждого chunk генерируется embedding
- Сохраняется в pgvector для быстрого retrieval

async def index_work_for_chat(
work: Work,
collection_id: str
):
# Получить текст
text = await get_work_text(work.id)

```
# Разбить на chunks
chunks = split_into_chunks(
    text.combined_text,
    chunk_size=512,
    overlap=50
)
```

```
# Генерировать embeddings
embeddings = await generate_embeddings_batch(
    [c.text for c in chunks]
)

# Сохранить в векторную БД
await db.execute("""
    INSERT INTO collection_chunks
    (collection_id, work_id, chunk_text, embedding)
    VALUES ($1, $2, $3, $4)
""", [
    (collection_id, work.id, c.text, e)
    for c, e in zip(chunks, embeddings)
])
```

3) Semantic Retrieval

async def retrieve_relevant_chunks(
collection_id: str,
query: str,
top_k: int = 10
) -> List[Chunk]:
# Embedding запроса
query_embedding = await generate_embedding(query)

```
# Vector similarity search в pgvector
chunks = await db.fetch("""
    SELECT
        work_id,
        chunk_text,
        1 - (embedding <=> $1) AS similarity
    FROM collection_chunks
    WHERE collection_id = $2
    ORDER BY embedding <=> $1
    LIMIT $3
""", query_embedding, collection_id, top_k)
```

```
return [Chunk.from_db(c) for c in chunks]
```

4) Conversational LLM

async def generate_chat_response(
session: ChatSession,
user_message: str,
relevant_chunks: List[Chunk]
) -> ChatMessage:

```
# Построить историю
history = build_conversation_history(session.messages[-5:])

# Построить контекст из chunks
context = "\n\n".join([
    f"[Источник {c.work_id}]\n{c.text}"
    for c in relevant_chunks
])

system_prompt = """
```

Ты — научный ассистент, который помогает исследователю работать с его библиотекой источников.

Твоя задача:

1. Отвечать на вопросы, используя ТОЛЬКО информацию из контекста
2. Цитировать источники в формате [ID]
3. Указывать на противоречия между работами, если они есть
4. Быть точным и научным в формулировках
   """
   messages = [
   {"role": "system", "content": system_prompt},
   *history,
   {
   "role": "user",
   "content": f"Контекст:\n{context}\n\nВопрос: {user_message}"

```
}
]
response = await claude.messages.create(
model="claude-sonnet-4-20250514",
max_tokens=1500,
temperature=0.4,
messages=messages
)
return ChatMessage(
role="assistant",
content=response.content[0].text,
sources_used=[c.work_id for c in relevant_chunks]
)
```

5) Session Persistence

```
CREATE TABLE chat_sessions (
id UUID PRIMARY KEY,
collection_id UUID REFERENCES collections(id),
user_id UUID REFERENCES users(id),
created_at TIMESTAMP,
updated_at TIMESTAMP
);

CREATE TABLE chat_messages (
id UUID PRIMARY KEY,
session_id UUID REFERENCES chat_sessions(id),
role VARCHAR(20),
content TEXT,
sources_used TEXT[],
created_at TIMESTAMP
);
```

4.3. UI/UX компоненты

Frontend:

- Вкладка "Чат" внутри экрана подборки
- Chat interface с input box внизу
- Сообщения с цитатами (кликабельные [ID])
- Боковая панель с списком используемых источников

- Quick prompts: "Какие методы используются?", "Сравни результаты"

Key interactions:

1. User открывает подборку → переключается на вкладку "Чат"
2. Видит quick prompts или вводит свой вопрос
3. Ответ появляется с цитатами источников
4. Клик по [ID] → превью источника в sidebar
5. Продолжает диалог с учетом контекста

4.4. Оптимизация

- Chunking выполняется асинхронно при добавлении в подборку
- pgvector индексы для быстрого similarity search (<100ms)
- Ограничение истории до последних 5 сообщений
- Rate limiting: 20 сообщений/час для free tier

4.5. Метрики успеха

- ≥40% пользователей с подборками используют чат
- Среднее количество сообщений в сессии ≥3
- Response time P95 ≤5 секунд
- User satisfaction ≥4.2/5.0

---

5. FEATURE 4: СИСТЕМАТИЧЕСКИЕ ОБЗОРЫ (SYSTEMATIC REVIEW WORKFLOW)

5.1. Описание функции

Структурированный workflow для проведения систематических обзоров литературы:

1. Поиск и импорт кандидатов
2. Title/Abstract screening (первичный отбор)
3. Full-text screening (вторичный отбор)
4. Data extraction

5. Финальный список источников

Вдохновлено: Elicit Systematic Review Pipeline[10].

5.2. Техническая реализация

Архитектура (State Machine):

Review States
Import → Screening → Full-text review → Extraction → Completed

Компоненты:

1) Review Management

```
class ReviewStatus(str, Enum):
SETUP = "setup"
SCREENING = "screening"
FULL_TEXT = "full_text"
EXTRACTION = "extraction"
COMPLETED = "completed"

class Review(BaseModel):
id: str
user_id: str
title: str
research_question: str
inclusion_criteria: List[str]
exclusion_criteria: List[str]
status: ReviewStatus
created_at: datetime

class ReviewItem(BaseModel):
id: str
review_id: str
work_id: str
status: Literal["undecided", "include", "exclude"]
screening_notes: Optional[str]
ai_recommendation: Optional[dict]
decided_by: Optional[str]
decided_at: Optional[datetime]
```

2) AI-Assisted Screening

async def get_screening_recommendation(
work: Work,
review: Review
) -> ScreeningRecommendation:

```
prompt = f"""
```

Оцени, подходит ли эта научная работа для систематического обзора.

Исследовательский вопрос: {review.research_question}

Критерии включения:
{'\n'.join(review.inclusion_criteria)}

Критерии исключения:
{'\n'.join(review.exclusion_criteria)}

Статья:
Название: {work.title}
Авторы: {work.authors}
Год: {work.year}
Аннотация: {work.abstract}

Верни JSON:
{{
"recommendation": "include | exclude | uncertain",
"reasoning": "подробное объяснение",
"matched_criteria": ["список совпавших критериев"],
"confidence": 0.0-1.0
}}
"""

```
response = await claude.messages.create(
    model="claude-3-5-haiku-20241022",
    max_tokens=800,
    temperature=0.2,
    messages=[{"role": "user", "content": prompt}],
```

```
    response_format={"type": "json_object"}
)

return parse_screening_recommendation(response.content)
```

3) Batch Screening

async def screen_review_items(
review_id: str,
batch_size: int = 20
) -> List[ReviewItem]:

```
# Получить undecided items
items = await db.fetch("""
    SELECT * FROM review_items
    WHERE review_id = $1 AND status = 'undecided'
    LIMIT $2
""", review_id, batch_size)

# Параллельная обработка
recommendations = await asyncio.gather(*[
    get_screening_recommendation(
        await get_work(item.work_id),
        await get_review(review_id)
    )
    for item in items
])

# Сохранить рекомендации
for item, rec in zip(items, recommendations):
    await db.execute("""
        UPDATE review_items
        SET ai_recommendation = $1
        WHERE id = $2
    """, rec.model_dump_json(), item.id)

return items
```

4) Review Statistics

```
class ReviewStats(BaseModel):
total_items: int
screened: int
included: int
excluded: int
undecided: int
ai_agree_rate: float # когда пользователь соглашается с ИИ

async def get_review_stats(review_id: str) -> ReviewStats:
stats = await db.fetchrow("""
SELECT
COUNT() as total,
COUNT() FILTER (WHERE status != 'undecided') as screened,
COUNT() FILTER (WHERE status = 'include') as included,
COUNT() FILTER (WHERE status = 'exclude') as excluded,
COUNT(*) FILTER (WHERE status = 'undecided') as undecided,
AVG(
CASE WHEN status::text =
ai_recommendation->>'recommendation'
THEN 1.0 ELSE 0.0 END
) as ai_agree_rate
FROM review_items
WHERE review_id = $1
""", review_id)

    return ReviewStats.from_db(stats)
```

5.3. UI/UX компоненты

Frontend:

- Страница /reviews с списком обзоров
- Создание обзора: форма с критериями включения/исключения
- Screening workspace:
  - Карточка текущей статьи (title, abstract)
  - ИИ-рекомендация с обоснованием
  - Кнопки: "Включить", "Исключить", "Пропустить"

    - Горячие клавиши: I (include), E (exclude), Space (next)
- Прогресс-бар с процентом обработанных работ
- Финальный список с возможностью экспорта

Key interactions:

1. User создает обзор → импортирует 200 работ из поиска
2. Переходит в screening режим
3. Видит карточку первой работы + ИИ-рекомендацию "Exclude"
4. Читает обоснование, соглашается, нажимает E
5. Следующая карточка, ИИ рекомендует "Include"
6. Не согласен, нажимает Exclude с комментарием
7. Продолжает до завершения
8. Экспортирует список включенных работ (60 штук)

5.4. Метрики успеха

- ≥20% пользователей создают хотя бы 1 обзор
- Среднее время screening: <30 секунд на работу
- ИИ-согласие (AI agreement): ≥65%
- Completion rate: ≥40% обзоров завершаются

---

6. FEATURE 5: КАРТА КОНЦЕПТОВ (CONCEPT MAP)

6.1. Описание функции

Автоматическая генерация тематической карты исследовательской области, включающая:

- Список ключевых концептов и подтем
- Связи между концептами
- Репрезентативные работы для каждого концепта
- Рекомендации по расширению поиска

Вдохновлено: Elicit Concepts Tool[11].

6.2. Техническая реализация

Архитектура:

Concept Mapping Flow
Topic query → Get top works → Extract concepts → Cluster & rank → Generate map

Компоненты:

1) Concept Extraction

async def extract_concepts_from_topic(
topic: str,
limit: int = 50
) -> List[Concept]:

```
# Поиск релевантных работ
works = await search_openalex(
    query=topic,
    filters=SearchFilters(year_from=2019),
    limit=limit
)

# OpenAlex concepts (level 1-2)
openalex_concepts = extract_openalex_concepts(works)

# LLM clustering и обогащение
concept_map = await generate_concept_clusters(
    topic=topic,
    works=works,
    base_concepts=openalex_concepts
)

return concept_map.concepts
```

2) LLM Clustering

async def generate_concept_clusters(
topic: str,
works: List[Work],
base_concepts: List[str]
) -> ConceptMap:

```
# Подготовить контекст
works_summary = "\n".join([
    f"{w.title} ({w.year}): {w.abstract[:200]}..."
    for w in works[:30]
])

prompt = f"""
```

Проанализируй научную область и создай концептуальную карту.

Тема: {topic}

Базовые концепты из OpenAlex:
{', '.join(base_concepts)}

Репрезентативные работы:
{works_summary}

Создай структурированную карту из 5-10 основных концептов.
Для каждого концепта:

1. Название (2-4 слова)
2. Краткое описание (1 предложение)
3. 2-3 подконцепта
4. Связи с другими концептами

Верни JSON:
{{
"concepts": [
{{
"id": "concept_1",
"name": "...",
"description": "...",

"subconcepts": ["...", "..."],
"related_to": ["concept_2", "concept_3"],
"representative_works": ["W123", "W456"]
}}
]
}}
"""

```
response = await claude.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=3000,
    temperature=0.5,
    messages=[{"role": "user", "content": prompt}],
    response_format={"type": "json_object"}
)

return parse_concept_map(response.content)
```

3) Visual Graph Generation

class ConceptGraph(BaseModel):
nodes: List[ConceptNode]
edges: List[ConceptEdge]

class ConceptNode(BaseModel):
id: str
label: str
description: str
size: int # по количеству работ
color: str # по категории

class ConceptEdge(BaseModel):
source: str
target: str
strength: float

async def build_concept_graph(
concept_map: ConceptMap
) -> ConceptGraph:

```
nodes = [
ConceptNode(
id=c.id,
label=c.name,
description=c.description,
size=len(c.representative_works) * 10,
color=assign_color(c)
)
for c in concept_map.concepts
]
```

```
edges = []
for concept in concept_map.concepts:
    for related_id in concept.related_to:
        edges.append(ConceptEdge(
            source=concept.id,
            target=related_id,
            strength=calculate_relationship_strength(
                concept, related_id
            )
        ))

return ConceptGraph(nodes=nodes, edges=edges)
```

6.3. UI/UX компоненты

Frontend:

- Страница /concepts с поиском по теме
- Интерактивный граф (react-force-graph или vis.js)
- Sidebar с деталями выбранного концепта
- Список репрезентативных работ
- Кнопка "Искать работы по концепту" → новый поиск

Key interactions:

1. User вводит тему "Machine learning in education"
2. Генерируется карта с 8 концептами
3. Клик на "Personalized learning systems"

4. Sidebar показывает описание + 5 ключевых работ
5. Нажимает "Искать работы" → переход к поиску с концептом
6. Добавляет работы в подборку "Adaptive learning"

6.4. Метрики успеха

- ≥30% новых пользователей используют карту концептов
- ≥50% кликают на концепты и просматривают работы
- ≥25% переходят к поиску по концепту
- Время генерации карты P95 ≤15 секунд

---

7. FEATURE 6: ALERTS И МОНИТОРИНГ НОВЫХ ПУБЛИКАЦИЙ

7.1. Описание функции

Автоматические уведомления о новых релевантных работах по:

- Сохраненным поисковым запросам
- Темам активных обзоров
- Цитированию работ из подборок

Вдохновлено: Elicit Alerts, Scite Dashboards[12][13].

7.2. Техническая реализация

Архитектура:

Alert System
Cron job (daily) → Check new works → Filter by relevance → Send notifications

Компоненты:

1) Alert Configuration

```
class Alert(BaseModel):
id: str
user_id: str
type: Literal["search", "review", "citations"]
query: Optional[str]
```

review_id: Optional[str]
collection_id: Optional[str]
frequency: Literal["daily", "weekly"]
channel: Literal["email", "telegram"]
last_checked: datetime
is_active: bool

Endpoint: POST /api/alerts

async def create_alert(alert: AlertCreate) -> Alert:
return await db.create_alert(alert)

2) Daily Alert Processing

async def process_daily_alerts():
"""Cron job запускается каждый день в 9:00 UTC"""

```
# Получить активные алерты с частотой daily
alerts = await db.fetch("""
    SELECT * FROM alerts
    WHERE is_active = true
    AND frequency = 'daily'
    AND last_checked < NOW() - INTERVAL '23 hours'
""")

for alert in alerts:
    try:
        new_works = await check_alert(alert)

        if len(new_works) > 0:
            await send_alert_notification(
                alert=alert,
                works=new_works
            )

        await db.execute("""
            UPDATE alerts
            SET last_checked = NOW()
```

```
            WHERE id = $1
        """, alert.id)

    except Exception as e:
        logger.error(f"Alert {alert.id} failed: {e}")
```

3) Alert Checking Logic

async def check_alert(alert: Alert) -> List[Work]:

```
if alert.type == "search":
    # Новые работы по запросу
    return await check_search_alert(alert)

elif alert.type == "review":
    # Новые работы, подходящие под критерии обзора
    return await check_review_alert(alert)

elif alert.type == "citations":
    # Новые цитирования работ из подборки
    return await check_citation_alert(alert)
```

async def check_search_alert(alert: Alert) -> List[Work]:
# Поиск с фильтром по дате
yesterday = (datetime.now() - timedelta(days=1)).strftime("%Y-%m-%d")

```
works = await search_openalex(
    query=alert.query,
    filters=SearchFilters(
        from_publication_date=yesterday
    ),
    limit=20
)

# LLM фильтрация по релевантности
relevant_works = []
```

```
    for work in works:
        score = await calculate_relevance(alert.query, work)
        if score > 0.7:
            relevant_works.append(work)

    return relevant_works[:5]  # топ-5
```

4) Notification Delivery

async def send_alert_notification(
alert: Alert,
works: List[Work]
):
user = await get_user(alert.user_id)

```
    if alert.channel == "email":
        await send_email(
            to=user.email,
            subject=f"LitFinder: {len(works)} новых работ",
            template="alert_email.html",
            context={"alert": alert, "works": works}
        )

    elif alert.channel == "telegram":
        await send_telegram_message(
            chat_id=user.telegram_chat_id,
            text=format_alert_message(alert, works)
        )
```

5) Email Template

Новые публикации по теме "{{alert.query}}"

Найдено {{works|length}} релевантных работ за последние 24 часа:

{% for work in works %}

> ### {{work.title}}
>
> **Авторы:** {{work.authors|join(", ")}}
>
> **Год:** {{work.year}} | **Цитирований:** {{work.citations_count}}
>
> {{work.abstract|truncate(200)}}
>
> Подробнее →

{% endfor %}

Управление алертами

7.3. UI/UX компоненты

Frontend:

- Страница /alerts со списком активных алертов
- Создание алерта: из результатов поиска или обзора
- Настройки: частота, канал доставки
- История срабатываний с архивом найденных работ

Key interactions:

1. User выполняет поиск по теме "Quantum computing"
2. Нажимает "Создать алерт по этому запросу"
3. Выбирает: Telegram, ежедневно
4. На следующий день получает сообщение с 3 новыми работами
5. Кликает на ссылку → открывается работа на платформе
6. Добавляет в подборку

7.4. Метрики успеха

- ≥25% активных пользователей создают минимум 1 алерт
- Open rate для email алертов ≥40%
- Click-through rate ≥15%
- Среднее количество алертов на пользователя: 2-3

---

8. FEATURE 7: REFERENCE CHECK (ПРОВЕРКА ТЕКСТА)

8.1. Описание функции

Анализ черновика диссертации или статьи для:

- Выявления утверждений, требующих цитирования
- Подбора релевантных источников для подтверждения утверждений
- Проверки на противоречия с существующей литературой

Вдохновлено: Scite Reference Check[14].

8.2. Техническая реализация

Архитектура:

Reference Check Flow
User text → Extract claims → Search sources → Match & score → Return suggestions

Компоненты:

1) Claim Extraction

Endpoint: POST /api/reference-check

class ReferenceCheckRequest(BaseModel):
text: str
language: str = "ru"

async def extract_claims(text: str) -> List[Claim]:

```
prompt = f"""
```

Проанализируй научный текст и извлеки утверждения, которые требуют подтверждения источниками.

Текст:
{text}

Для каждого утверждения укажи:

1. Текст утверждения
2. Тип (факт, статистика, вывод, метод)
3. Приоритет цитирования (high/medium/low)

Верни JSON массив:
[
{{
"claim_text": "...",
"type": "fact|statistic|conclusion|method",
"priority": "high|medium|low",
"start_pos": 0,
"end_pos": 100
}}
]
"""

```
response = await claude.messages.create(
    model="claude-sonnet-4-20250514",
    max_tokens=2000,
    temperature=0.2,
    messages=[{"role": "user", "content": prompt}],
    response_format={"type": "json_object"}
)

return parse_claims(response.content)
```

2) Source Matching

async def find_sources_for_claim(
claim: Claim
) -> List[SourceMatch]:

```
# Semantic search по OpenAlex
works = await search_openalex(
    query=claim.claim_text,
    limit=10
)

# LLM оценка релевантности
matches = []
for work in works:
    score = await evaluate_source_match(claim, work)
    if score.relevance > 0.6:
        matches.append(SourceMatch(
            work=work,
            relevance=score.relevance,
            support_type=score.support_type,
            excerpt=score.excerpt
        ))

return sorted(matches, key=lambda x: x.relevance, reverse=True)[:3]
```

async def evaluate_source_match(
claim: Claim,
work: Work
) -> SourceMatchScore:

```
prompt = f"""
```

Оцени, насколько источник подтверждает утверждение.

Утверждение: {claim.claim_text}

Источник:
Название: {work.title}
Аннотация: {work.abstract}

Верни JSON:
{{
"relevance": 0.0-1.0,

"support_type": "strong|partial|contradicts|unrelated",
"excerpt": "цитата из аннотации, подтверждающая связь"
}}
"""

```
response = await claude.messages.create(
    model="claude-3-5-haiku-20241022",
    max_tokens=400,
    temperature=0.2,
    messages=[{"role": "user", "content": prompt}],
    response_format={"type": "json_object"}
)

return parse_source_match_score(response.content)
```

3) Aggregation

class ReferenceCheckResult(BaseModel):
claims: List[ClaimWithSources]
summary: ReferenceCheckSummary

class ClaimWithSources(BaseModel):
claim: Claim
suggested_sources: List[SourceMatch]
citation_needed: bool

class ReferenceCheckSummary(BaseModel):
total_claims: int
high_priority: int
sources_found: int
gaps: List[str] # утверждения без хороших источников

async def check_references(
request: ReferenceCheckRequest
) -> ReferenceCheckResult:

```
# Извлечь утверждения
claims = await extract_claims(request.text)
```

```
# Параллельный поиск источников
claims_with_sources = await asyncio.gather(*[
    find_sources_with_claim(claim)
    for claim in claims
])

# Построить summary
summary = build_summary(claims_with_sources)

return ReferenceCheckResult(
    claims=claims_with_sources,
    summary=summary
)
```

8.3. UI/UX компоненты

Frontend:

- Страница /reference-check с текстовой областью
- Paste или upload .docx файла
- Подсветка утверждений в тексте (разные цвета по приоритету)
- Sidebar с найденными источниками для каждого утверждения
- Кнопки "Добавить в подборку" и "Скопировать цитату"

Key interactions:

1. User вставляет фрагмент введения диссертации (3 абзаца)
2. Нажимает "Проверить"
3. 8 утверждений подсвечиваются (4 high priority)
4. Клик на первое утверждение → sidebar показывает 3 источника
5. Hover над источником → превью с релевантной цитатой
6. Добавляет источник в подборку
7. Копирует formatted цитату для вставки

8.4. Метрики успеха

- ≥35% пользователей пробуют функцию
- ≥60% добавляют минимум 1 найденный источник в подборку
- Точность extraction утверждений ≥80%
- Relevance найденных источников (user rating) ≥3.8/5.0

---

9. ИНТЕГРАЦИЯ С GOST FORMATTER

9.1. Описание функции

Автоматическое форматирование библиографических записей из подборок в соответствии с:

- ГОСТ Р 7.0.100-2018 (РФ, Казахстан)
- ВАК РБ (Беларусь)
- Экспорт в BibTeX, RIS, JSON

Интеграция с: GOST Formatter Agent (Claude Haiku 4.5).

9.2. Техническая реализация

Архитектура:

Formatting Flow
Collection → Normalize metadata → Call GOST Formatter API → Return formatted → Display/Export

Компоненты:

1) Batch Formatting

Endpoint: POST /api/format/batch

class BatchFormatRequest(BaseModel):
collection_id: str

style: Literal["VAK_RB", "GOST_R"]
numbering: bool = True

async def format_collection(
request: BatchFormatRequest
) -> BatchFormatResult:

```
# Получить все работы из подборки
items = await db.fetch("""
    SELECT * FROM collection_items
    WHERE collection_id = $1
    ORDER BY added_at
""", request.collection_id)

# Форматировать параллельно (батчами по 10)
formatted_items = []
for batch in chunks(items, 10):
    batch_results = await asyncio.gather(*[
        format_single_item(item, request.style)
        for item in batch
    ])
    formatted_items.extend(batch_results)

# Добавить нумерацию
if request.numbering:
    for i, item in enumerate(formatted_items, 1):
        item.formatted_text = f"{i}. {item.formatted_text}"

return BatchFormatResult(items=formatted_items)
```

2) GOST Formatter API Call

async def format_single_item(
item: CollectionItem,
style: str
) -> FormattedItem:

```
# Нормализовать метаданные
source_data = normalize_metadata(item)

# Вызвать GOST Formatter API
response = await httpx.post(
    f"{GOST_FORMATTER_URL}/api/v1/format/bibliography",
    json={
        "source_data": source_data,
        "format_style": style,
        "output_language": "ru",
        "return_bibtex": True
    },
    timeout=5.0
)

result = response.json()

return FormattedItem(
    work_id=item.work_id,
    formatted_text=result["data"][f"{style.lower()}_formatted"],
    bibtex=result["data"]["bibtex"],
    confidence=result["data"]["validation"]["confidence"],
    warnings=result["data"]["validation"]["warnings"]
)
```

def normalize_metadata(item: CollectionItem) -> dict:
"""Преобразовать данные OpenAlex в формат GOST Formatter"""

```
return {
    "type": map_work_type(item.type),
    "authors": parse_authors_ru(item.authors),
    "title": item.title,
    "journal": item.source,
    "year": item.year,
    "volume": extract_volume(item.raw_metadata),
    "issue": extract_issue(item.raw_metadata),
    "pages": item.pages,
```

```
    "doi": item.doi,
    "url": item.url,
    "language": "ru"
}
```

3) Export Formats

async def export_formatted_bibliography(
collection_id: str,
style: str,
format: Literal["docx", "txt", "json"]
) -> bytes:

```
# Получить отформатированные записи
formatted = await format_collection(
    BatchFormatRequest(
        collection_id=collection_id,
        style=style
    )
)

if format == "txt":
    return generate_txt(formatted)

elif format == "docx":
    return generate_docx(formatted)

elif format == "json":
    return generate_json(formatted)
```

def generate_docx(formatted: BatchFormatResult) -> bytes:
"""Создать .docx файл со списком литературы"""

```
from docx import Document

doc = Document()
doc.add_heading("Список использованных источников", level=1)
```

```
for item in formatted.items:
    p = doc.add_paragraph()
    p.add_run(item.formatted_text)
    p.style = "List Number"

buffer = BytesIO()
doc.save(buffer)
return buffer.getvalue()
```

9.3. UI/UX компоненты

Frontend:

- Кнопка "Форматировать" в экране подборки
- Modal выбора стандарта: ВАК РБ или ГОСТ Р
- Preview отформатированного списка
- Предупреждения о низком confidence или недостающих данных
- Кнопки экспорта: .docx, .txt, BibTeX, JSON

Key interactions:

1. User открывает подборку с 60 источниками
2. Нажимает "Форматировать по ВАК РБ"
3. Видит preview: 60 записей, 2 предупреждения
4. Проверяет записи с предупреждениями, исправляет метаданные
5. Нажимает "Экспорт в Word"
6. Скачивает готовый .docx файл

9.4. Метрики успеха

- ≥70% пользователей с подборками форматируют их
- ≥90% accuracy форматирования (по экспертной оценке)
- P95 время форматирования ≤3 секунды на 10 работ
- ≥80% экспортируют результаты

---

10. ДОПОЛНИТЕЛЬНЫЕ ФУНКЦИИ MVP

10.1. Базовый поиск по OpenAlex

Описание: Стандартный поиск с фильтрами (год, тип, язык, open access).

Технологии:

- OpenAlex Works API
- Фильтры: from_publication_date, type, language, is_oa
- Пагинация: cursor-based для deep paging

Endpoint: `GET /api/search/works`

10.2. Управление подборками (Collections)

Описание: CRUD операции над подборками и их элементами.

Технологии:

- PostgreSQL для хранения
- FastAPI эндпоинты: GET, POST, PATCH, DELETE
- Поддержка тегов и описаний

Endpoints:

- GET /api/collections
- POST /api/collections
- POST /api/collections/{id}/items
- DELETE /api/collections/{id}/items/{item_id}

10.3. Экспорт подборок

Форматы:

- CSV (title, authors, year, source, DOI, URL)
- JSON (полные метаданные)
- BibTeX (через GOST Formatter)

Endpoint: `GET /api/collections/{id}/export?format=csv`

10.4. Аутентификация и авторизация

Технологии:

- JWT tokens (access + refresh)
- bcrypt для хеширования паролей
- OAuth2 flow с password grant

Endpoints:

- POST /api/auth/register
- POST /api/auth/login
- POST /api/auth/refresh

---

11. АРХИТЕКТУРА ИНФРАСТРУКТУРЫ

11.1. Deployment Architecture

Production Infrastructure
User → Cloudflare CDN → Nginx → [Next.js | FastAPI] → PostgreSQL/Redis
↓
External APIs (OpenAlex, Claude)

Компоненты:

Компонент	Технология	Ресурсы
Frontend	Next.js 14	Vercel / Yandex Cloud
Backend API	FastAPI + Uvicorn	2 vCPU, 4GB RAM
Database	PostgreSQL 15 + pgvector	2 vCPU, 8GB RAM
Cache	Redis 7	1 vCPU, 2GB RAM
Reverse Proxy	Nginx	1 vCPU, 1GB RAM

11.2. CI/CD Pipeline

GitHub Actions workflow:

1. Push to main → trigger workflow
2. Run tests (pytest, jest)

3. Build Docker images
4. Push to container registry
5. Deploy to staging (auto)
6. Manual approval for production
7. Deploy to production
8. Health check

11.3. Мониторинг и логирование

Stack:

- Prometheus для метрик
- Grafana для дашбордов
- Loki для логов
- Sentry для error tracking

Key метрики:

- API latency (P50, P95, P99)
- Error rate
- Active users
- LLM token usage
- Database connection pool

---

12. ОЦЕНКА СТОИМОСТИ И РЕСУРСОВ

12.1. LLM API Costs (месячная оценка для 1000 активных пользователей)

Функция	Модель	Запросов/мес	Стоимость
Research Answer	Sonnet 4	5,000	$150
Data Extraction	Haiku 4.5	10,000	$30
Chat with Library	Sonnet 4	15,000	$225
Screening	Haiku 4.5	20,000	$60
Concept Map	Sonnet 4	2,000	$60
Reference Check	Haiku 4.5	8,000	$24
GOST Formatter	Haiku 4.5	30,000	$90
Embeddings	OpenAI	50,000	$20
ИТОГО			**$659/мес**

12.2. Инфраструктура (Yandex Cloud)

- Compute (VMs): $150/мес
- Database: $80/мес
- Object Storage: $20/мес
- Network: $30/мес

Итого инфраструктура: ~$280/мес

12.3. Total Cost of Ownership

Месячные расходы MVP:

- LLM APIs: $659
- Инфраструктура: $280
- Домены/SSL: $10
- Внешние сервисы: $50

Итого: ~$1,000/мес или $12,000/год для 1000 активных пользователей.

Cost per user: ~$1.00/месяц

---

13. ПЛАН РАЗРАБОТКИ MVP (8 НЕДЕЛЬ)

Week 1-2: Core Backend

- Setup FastAPI project structure
- Database schema и миграции
- Auth system (JWT)
- OpenAlex integration
- Basic search endpoint

Week 3-4: Collections + GOST Formatter

- Collections CRUD
- GOST Formatter integration
- Batch formatting
- Export functionality

Week 5-6: AI Features (Phase 1)

- Research Answer (semantic search + LLM)
- Data Extraction pipeline
- Chat with Library (RAG)

Week 7: AI Features (Phase 2) + Frontend

- Systematic Review workflow
- Concept Map
- Frontend development (Next.js)

Week 8: Testing + Deployment

- Integration testing
- User acceptance testing
- Performance optimization
- Production deployment

---

14. МЕТРИКИ УСПЕХА MVP

14.1. Product Metrics (первые 3 месяца)

- ≥100 зарегистрированных пользователей
- ≥30 активных пользователей (weekly)
- ≥50 созданных подборок
- ≥1,000 добавленных источников
- Retention rate (D7) ≥30%

14.2. Feature Adoption

- Research Answer: ≥50% пользователей пробуют
- Data Extraction: ≥20% используют
- Chat with Library: ≥30% используют
- GOST Formatting: ≥60% используют

14.3. User Satisfaction

- NPS (Net Promoter Score): ≥40
- CSAT (Customer Satisfaction): ≥4.0/5.0
- Feature usefulness rating: ≥4.2/5.0

---

15. РИСКИ И МИТИГАЦИЯ

15.1. Технические риски

Риск	Вероятность	Митигация
OpenAlex API недоступность	Средняя	Кеширование, fallback на Semantic Scholar
LLM API rate limits	Высокая	Rate limiting, очереди, batch processing
Низкое качество LLM ответов	Средняя	Confidence thresholds, validation, user feedback loops
Медленная производительность	Средняя	Асинхронность, кеширование, CDN

15.2. Бизнес-риски

Риск	Вероятность	Митигация
Низкая адопция пользователей	Средняя	Pilot с аспирантами БГУ, итерации по feedback
Высокие расходы на LLM	Высокая	Оптимизация промптов, кеширование, freemium модель
Конкуренция с Elicit/Scite	Низкая	Фокус на СНГ-рынок, ВАК/ГОСТ специфика

---

16. СЛЕДУЮЩИЕ ШАГИ ПОСЛЕ MVP

Phase 2 (месяцы 4-6)

- Интеграция с eLIBRARY (CyberLeninka полнотекстовый поиск)
- Smart Citations (контекст цитирования как в Scite)
- Коллаборация (shared collections, комментарии)
- Mobile app (React Native)

Phase 3 (месяцы 7-12)

- Institutional subscriptions (университеты, НИИ)
- Advanced analytics (citation networks, trends)
- Integrations (Zotero, Mendeley, Word plugin)
- AI writing assistant для диссертаций

---

17. ЗАКЛЮЧЕНИЕ

LitFinder MVP представляет собой комплексную ИИ-платформу для работы с научной литературой, объединяющую:

1. Семантический поиск с Research Assistant
2. Табличную экстракцию данных
3. Интерактивный чат с библиотекой
4. Систематические обзоры
5. Карту концептов
6. Автоматические алерты
7. Проверку цитирований
8. Форматирование по ВАК РБ и ГОСТ Р

Ключевые преимущества:

✅ Адаптация лучших практик Elicit и Scite под СНГ-рынок
✅ Нативная поддержка ВАК РБ и ГОСТ Р 7.0.100-2018
✅ Современный стек технологий (FastAPI, Next.js, Claude AI)
✅ Модульная архитектура для быстрых итераций
✅ Доступная стоимость эксплуатации (~$1/user/month)

Дата: 11 февраля 2026
Версия: 1.0 MVP
Статус: Ready for Development

REFERENCES

[1] Elicit AI. (2026). AI for scientific research. https://elicit.com

[2] Scite AI. (2026). Smart citations for researchers. https://scite.ai

[3] Effortless Academic. (2026). Scite AI Review 2026: Literature Review Tool for Researchers. https://effortlessacademic.com/scite-ai-review

[4] Anara. (2025). Scite vs Elicit: The Specialist Showdown. https://anara.com/blog/scite-vs-elicit

[5] OpenAlex. (2026). Search works - Technical documentation. https://docs.openalex.org/api-entities/works/search-works

[6] Fahim AI. (2026). How to Use Elicit for 10x Faster Automated Research. https://www.fahimai.com/ru/how-to-use-elicit

[7] Anara. (2025). How to use Elicit AI for literature reviews. https://anara.com/blog/elicit-literature-reviews

[8] Revoyant. (2024). Elicit vs Scite comparison. https://www.revoyant.com/compare/elicit-vs-scite

[9] CustomGPT.ai. (2024). Elicit / Scite / Phind vs CustomGPT.ai Researcher. https://researcher.customgpt.ai/docs/comparisons

[10] Elicit Blog. (2026). Features. https://elicit.com/blog/tag/features

[11] Paperguide. (2026). Elicit vs Consensus: Detailed Comparison. https://paperguide.ai/blog/elicit-vs-consensus

[12] Skywork.ai. (2025). Scite AI: Ultimate Guide to Smarter Research. https://skywork.ai/skypage/en/Scite-AI-Guide

[13] Paperpal. (2025). Scite AI Review: Features, Pricing and Alternatives. https://paperpal.com/blog/scite-ai-review

[14] Scite. (2026). Reference Check tool documentation. https://scite.ai/reference-check