



## А. Подготовка данных и парсинг сложных документов

---

### Hall of Multimodal OCR VLMs (Hugging Face)

Источник: Блог сообщества на Hugging Face (Prithiv Sakthi), октябрь 2025 [1](#) [2](#).

**Контекст:** Решает проблему извлечения текста и структуры из сложных документов (сканы, таблицы, диаграммы) с помощью мультимодальных моделей. Традиционные OCR теряют контекст таблиц и изображений; новые VLM (Vision-Language Models) обещают семантическое понимание макета документа.

**Алгоритмическая суть:** Современные OCR-VLM объединяют зрение и язык: визуальный backbone (часто ViT или Swin-трансформер) выделяет регионы текста, таблиц, графиков, а языковая модель на основе трансформера генерирует по ним структурированный вывод [3](#) [4](#). Модели обучены отвечать на вопросы по документу и извлекать смысл, сохраняя пространственные отношения. Например, Qwen3-VL (9B параметров) понимает до 40 языков, включая рукописный текст и древние скрипты, и может прямо отвечать на вопросы по схемам и таблицам [5](#) [6](#).

**Техническая реализация:** В основе – двунаправленный трансформер, где визуальные признаки (пиксели или patch-эмбеддинги) совмещаются с текстовыми токенами через модальности. Выходом модели часто служит разметка Markdown/HTML с сохранением структуры: например, табличные данные представляются в виде Markdown-таблиц, формулы – в LaTeX, графики – в виде описаний или даже код диаграмм (mermaid) [7](#) [8](#). Модели вроде PaddleOCR-VL (0.9B) выдают JSON/HTML с координатами элементов, что упрощает последующую обработку. Hugging Face демонстрирует множество таких моделей (DeepSeek-OCR, Nanonets-OCR2, Granite-Docling) со сравнимыми API для инференса.

**Метрики и результаты:** Новейшие крупные модели Chandra-9B и OlmOCR-8B достигли точности OCR-сценариев ~83% (средний score) – лидируя на мультиязычных бенчмарках [2](#). При этом даже компактные модели ~3–4B (DeepSeek, dots, Nanonets) показывают высокую точность распознавания таблиц и рукописи. Специализированный Granite-Docling-258M (от IBM) на 258 млн параметров почти догнал большие модели по извлечению структуры, оставаясь очень экономичным [9](#) [10](#).

**Почему это важно:** Это значимый прогресс – VLM-OCR превратились из простого «читалки текста» в интеллектуальных агентов для документов. Они способны ответить на вопрос из отчёта, понять подпись к графику, извлечь таблицу без потери данных. Особенно для RAG-систем это прорыв: такие модели позволяют индексировать не только сырой текст страниц, но и смысловые единицы (ячейки таблиц, подписи рисунков) в векторное хранилище. В продакшне это применимо для тысяч разнородных документов (сканы договоров, технические схемы) – теперь их можно конвертировать в пригодный вид с минимальными потерями информации.

---

### Hybrid Multimodal Document Parsing (Instill AI)

Источник: Tech-блог Instill AI, ноябрь 2024 [11](#) [12](#).

**Контекст:** Адресует проблему “галлюцинаций” чисто нейронных парсеров и потери данных в табличных структурах при парсинге PDF. В сложных PDF (много колонок, вложенные таблицы, графики) ни классические инструменты (pdfplumber, pdfminer) ни чистые VLM не дают идеально точного результата [12](#) [13](#). Здесь предлагается гибридный конвейер: сочетание правил и нейросетей для **полного** разбора документа.

**Алгоритмическая суть:** В конвейере Instill происходит параллельное извлечение: сначала эвристический парсер (на базе pdfminer/pdfplumber) выдергивает сырой текст, разбивает его по структуре (Markdown-драфт) [14](#) [15](#). Одновременно каждую страницу PDF конвертируют в

*изображение (300 DPI) и батчами подают VLM-модели. VLM видит визуальный макет страницы и итеративно улучшает черновой Markdown: поправляет разметку списков и заголовков, форматирует таблицы, добавляет недостающие подписи к картинкам* <sup>15</sup> <sup>16</sup>. Если PDF изначально скан (т.е. текст не извлечён правилами), срабатывает fallback: выполняется чистый OCR по картинкам и VLM генерирует Markdown "с нуля" <sup>17</sup>.

**Техническая реализация:** Pipeline состоит из этапов: 1) **PDF -> изображения** (каждая страница), 2) **Heuristic parse**: извлечение текста и базовой структуры (Document Operator от Instill, аналог Unstructured) <sup>12</sup> <sup>13</sup>, 3) **Batching**: страницы группируются, чтобы VLM обрабатывал сразу, сохраняя контекст разбивки по страницам <sup>18</sup>, 4) **VLM refinement**: вызов инструмента LlamaParse/ZeroxOCR – модифицированной версии LLaMA, обученной OCR – для корректировки каждого чанка Markdown на основе изображения страницы <sup>13</sup> <sup>19</sup>, 5) **Слияние**: итоговые фрагменты Markdown объединяются в полный документ. Выход – Markdown с сохранением всех структур (заголовки `#`, списки `-`, таблицы в формате Markdown, изображения как ссылки). Этот Markdown далее можно скормить в чанкеры RAG.

*Метрики и результаты: Гибридный метод на внутренних тестах Instill показал ~98% полноту текста (почти не теряет символы, в отличие от ~90% у pdfplumber на тех же сложных файлах). Точность восстановления структуры таблиц возросла на 15–20% F1 по сравнению с чистым VLM: многие мелкие таблицы, пропущенные pdfplumber, были правильно встроены в Markdown (благодаря зритальному анализу VLM). Время обработки одной страницы ~500 мс, что лишь немного больше, чем у rules-based (~300 мс), но без провалов на сложных местах* <sup>15</sup> <sup>17</sup>.

**Почему это важно:** Такой гибрид – практическое решение для продакшена. Он не "выдумывает" текст (VLM часто галлюцинирует контент, если чего-то не распознал), потому что всегда опирается на черновой парсинг. С другой стороны, он исправляет то, с чем правила не справляются – например, правильный порядок колонок таблицы, слияние вынесенных в бокноты сносок. Для систем, где нужно загрузить миллионы PDF, подход масштабируется: можно параллельно пускать pipeline на кластер, а Markdown уже хранить как источник знаний. В итоге документы любой сложности становятся RAG-ready, без ручной очистки.

---

## Docling – открытый фреймворк для парсинга документов

Источник: Презентация IBM/Linux Foundation, сентябрь 2025 <sup>20</sup> <sup>21</sup>.

**Контекст:** Docling решает проблему универсальности: парсинг **любых** форматов (PDF, DOCX, PPTX, таблицы Excel, изображения) в структурированные данные. В продакшне приходится работать не только с PDF, но и с вордовскими документами, сканами, аудио – Docling стремится покрыть максимум, сохраняя содержимое.

*Алгоритмическая суть: Фреймворк реализует двойной подход: (1) Classic pipeline – каскад моделей и правил, выполняющих Layout Analysis (поиск блоков текста, заголовков), Table Structure Recognition (TSR), распознавание формул, классификацию фигур, определение порядка чтения* <sup>22</sup> <sup>23</sup>. На каждой странице запускаются отдельные модели (CNN/ViT для разметки макета страницы; детектор таблиц выделяет клетки и границы; модель распознавания формул экстрагирует Latex; классификатор картинок определяет подписи и легенды). Результаты по всем страницам объединяются и формируются в единый DoclingDocument – внутреннее представление (JSON со всеми элементами и их типами) <sup>24</sup> <sup>25</sup>. (2) VLM pipeline – альтернативный режим, где используется компактная vision-language модель Granite-Docling-258M (разработана IBM) <sup>26</sup>. Она получает страницу как изображение и сразу генерирует "DocTags" – специальную разметку с тегами, представляющими структуру (заголовок, параграф, таблица и т.д.). Это ускоряет процесс, но требует мощностей на этапе генерации.

**Техническая реализация:** Docling поддерживает **форматы входа**: PDF, DOCX, PPTX, XLSX, изображения (PNG, JPG), аудио (WAV, MP3 с speech-to-text) <sup>20</sup> <sup>21</sup>. **Выходные форматы** – Markdown, HTML или "DocTags"-JSON <sup>27</sup>. Причем Markdown/HTML генерируются без потери: например, если в PDF была вложенная таблица, в Markdown она будет представлена как

вложенные таблицы (возможно, с отступами или как HTML-вставка). Метаданные (номер страницы, координаты объектов) хранятся в JSON, так что можно при необходимости восстановить оригинальное расположение. Docling интегрируется с LangChain и Langflow<sup>28</sup>, то есть сразу может работать как Loader. Он также модульный: можно подключать **другие OCR движки** (Tesseract, PaddleOCR) через плагины<sup>29</sup>.

*Метрики и результаты: На открытом наборе PDF (FinePDFs, 475 млн документов) команда добилась точности парсинга ~92% по ключевым элементам (таблицы, заголовки, параграфы) – по совокупной метрике из статьи<sup>30</sup>. Табличный парсер Docling превзошёл Apache Tika и Unstructured no F1 примерно на +8-10%. Granite-Docling VLM показал сравнимую точность (~90% извлечение) при ~5x меньшем размере модели, что важно для локального деплоя<sup>10 31</sup>.*

**Почему это важно:** IBM передала Docling в Linux Foundation, делая его открытый стандартом<sup>32</sup>. Это значит, что индустрия получила мощный инструмент без Vendor lock-in. Для RAG-систем Docling удобен тем, что **единообразно** превращает любые данные в структурированный вид. Можно одной библиотекой прокачать и PDF-отчёты, и Word-документы, и даже аудио-записи (они станут текстом с таймкодами). В крупном приложении (напр. корпоративная база знаний) это сильно снижает трудозатраты: не надо писать парсеры под каждый формат. Кроме того, DoclingDocument формат может стать стандартом обмена – модели могут прямо учиться на нём. В итоге, подготовка данных перестаёт быть узким местом RAG-пайплайна.

## В. Продвинутые стратегии чанкинга

---

### Лучшие стратегии разбиения текста для RAG (итоги 2025)

Источник: Firecrawl Blog (автор Bex Tuychiev), перевод CSDN, декабрь 2025<sup>33 34</sup>.

**Контекст:** Статья сравнивает 6 популярных подходов к **chanкингу** (разбиению документов на фрагменты) и показывает, насколько выбор стратегии влияет на успех RAG. Проблема: если бить текст на слишком большие куски – не влезет в контекст LLM, слишком маленькие – теряется смежный контекст, LLM может “пропустить” ответ. Цель – найти баланс и алгоритмически определить границы смысловых блоков.

**Алгоритмическая суть:** Рассмотрены стратегии: 1) Фиксированный размер (число символов или токенов) – самый простой, но игнорирует смысл, может рвать фразу посередине. 2) Recursive Character Split (рекурсивное деление по разделителям: параграфы, предложения, слова) – итеративно пытается разбить текст по крупным разделителям (`\n\n`, затем `\n`, затем пробелы) сохраняя порог длины<sup>35 36</sup>. 3) Semantic Chunking – основан на эмбеддингах: модель вычисляет embedding каждого предложения и меряет косинусную близость соседних предложений; если семантическая похожесть падает ниже порога – вставляется разрыв чанка<sup>37 38</sup>. Порог может выбираться адаптивно: например, разрыв, если средство упало ниже 95-го перцентиля нормального распределения по тексту, или на  $>3\sigma$  от среднего<sup>39</sup>. 4) Page-level – использовать физические страницы PDF как чанки. 5) Sentence-based – каждый отдельный запрос (например, FAQ) не разбивается вовсе, или документ делится по предложениям без слияния. 6) LLM-based – отдать весь документ модели (например GPT-4) и попросить разбить по смысловым разделам. Это дорого, но модель может учесть сложную структуру (например, секции научной статьи).

**Техническая реализация:** Реализация большинства методов доступна в инструментах: у LangChain есть `RecursiveCharacterTextSplitter` и экспериментальный `SemanticChunker`

<sup>38</sup>. `SemanticChunker` под капотом вычисляет embeddings через OpenAI или SentenceTransformers и проходит по тексту, сравнивая соседние предложения – где скачок `dissimilarity`, там `split_text` начинает новый фрагмент<sup>37 40</sup>. Также есть подходы percentile/σ, описанные выше, которые в open-source пока вручную настраивают (например, задать `threshold=0.7` или `percentiles`). **Adaptive chunking** означает, что длина чанка не фиксирована: в одном месте может быть 200 слов до смены темы, в другом – 800, если содержание однородное

<sup>41</sup> <sup>42</sup>. Для иерархичных документов применяют **Hierarchical Chunking**: сперва бьют по крупным разделам (например, заголовки H2), потом каждый раздел дополнительно режут, если он слишком велик. Это сохраняет контекст внутри подразделов. NVIDIA предложила метод **RAPTOR** – построение иерархического дерева аннотаций (резюме секций), чтобы при поиске учитывать не только мелкие куски, но и обзор верхнего уровня <sup>43</sup> <sup>44</sup>.

*Метрики и результаты: Разница между наивным и умным чанкингом колоссальна. В эксперименте NVIDIA (2024) на 5 датасетах точность ответа LLM выросла с 0.59 до 0.648 (Accuracy@1) при переходе от векторного поиска по фиксированным кускам к поиск по страницам <sup>45</sup> <sup>46</sup> – страницный подход оказался лучшим для финансовых, юридических и научных текстов (там каждая страница логически самодостаточна). Semantic chunking прибавил до +9% Recall@K по сравнению с равномерным разбиением <sup>33</sup> <sup>46</sup>, т.к. объединял связанные предложения и уменьшал "дробление" ответа между кусками. Recursive splitter (LangChain по умолчанию) в тестах ChromaDB давал ~85–90% Recall@10 без доп. затрат на вычисления эмбеддингов <sup>47</sup> <sup>46</sup> – поэтому его считают сильной базой. LLM-based разбиение дало самые осмысленные фрагменты, но сократило производительность пайплайна почти в 10 раз (очень медленно, дорого).*

**Почему это важно:** Чанкинг больше не мелочь “как угодно порежем”. От него зависит, найдётся ли нужный фрагмент. Неправильный чанкинг – главная причина, почему RAG-бот “не знает ответа” при том, что в базе знаний ответ есть. В 2025 появился консенсус: **нет универсального метода** <sup>48</sup>, нужно подбирать под задачу. Например, для FAQ вообще можно не резать (каждый документ мал и отвечает на один вопрос). Для длинного отчёта – лучше semantic или page-level, чтобы вопрос нашёл всю нужную инфу. Инженеру важно уметь быстро попробовать разные стратегии и измерить качество (например, с помощью RAGAS или LangSmith). Благо, инструменты это уже позволяют – можно переключать splitter, менять overlap, и сразу видеть влияние на Precision@k. В продакшене, вероятно, будут использовать *гибридные стратегии*: например, сперва page chunking для быстрого охвата, а затем внутри страницы – semantic split, если страница очень насыщена информацией.

## C. Graph RAG и знаниеевые графы

---

### Интеграция Neo4j-графа в RAG-пайpline

Источник: Туториал DigitalOcean Gradient, март 2025 <sup>49</sup> <sup>50</sup>.

**Контекст:** Задача – улучшить retrieval для сложных вопросов, где просто схожести текста недостаточно. Например: “Найдите все упоминания компании X и её поставщиков в отчетах 2020 года”. Векторный поиск может найти фрагменты про X, но не установит связь “поставщик” без явного текста. Решение – построить **граф знаний** по данным и делать запросы по графу.

**Алгоритмическая суть:** В pipeline сначала выполняется Named Entity Recognition (NER) по всем документам: находят сущности (персоны, организации, даты и т.д.) <sup>49</sup> <sup>51</sup>. На базе этого создаётся граф: узлы – сущности, документы, а рёбра – отношения “Document MENTIONS Entity” или даже “Entity → RELATED\_TO → Entity” (если из текста можно вывести связь) <sup>52</sup>. В примере из туториала все BBC-новости импортируются в Neo4j, где каждая статья – узел Document, все упомянутые имена – узлы Entity, и между ними рёбра MENTIONS. Multi-hop retrieval реализуется так: для пользовательского запроса делается NER (напр. “компания X” и “поставщики” станут сущностями), затем выполняется Cypher-запрос к графу: найти все Document, которые содержат узел “X” и связанные узлы типа “Supplier” <sup>53</sup>. Возвращаются ID документов, и уже по ним поднимается текст для LLM. Таким образом, поиск идет не по словам, а по связям: если компания X фигурирует рядом с компанией Y, граф может это учесть, даже если слово “поставщик” прямо не написано.

**Техническая реализация:** Используется **Neo4j** – развёрнут локально через Docker <sup>54</sup> <sup>55</sup>. Скрипт на Python с библиотекой neo4j и spaCy обходит все файлы: для каждого документа создает

узел (`MERGE (d:Document {...})`), потом для каждой распознанной сущности (`nlp(text)`) создает узел (`(e:Entity {...})`), и связь (`(d)-[:MENTIONS]->(e)`)<sup>52</sup> <sup>56</sup>. Неявные связи (например "Х поставляет Y") можно добавить отдельными ребрами типа SUPPLIES, но в примере упор на явные упоминания. При запросе pipeline делает что-то вроде: `MATCH (d:Document)-[:MENTIONS]->(e:Entity) WHERE e.name IN {entity_list} RETURN d ORDER BY count(e) DESC` – т.е. рейтингуются документы по количеству совпадавших сущностей<sup>53</sup>. Полученные топ-узы документов конвертируются обратно в текст (путём хранения ссылок или ID). В finale LLM видит не просто набор кусочков похожести, а целые документы, отфильтрованные по графовым связям.

**Метрики и результаты:** Такой графовый подход заметно повысил точность многошаговых ответов. По отзывам разработчиков, *recall* на вопросы типа "найти цепочку связанных элементов" увеличился с ~70% до 85% (т.к. граф находит документы, которые через 2-3 связи дотягиваются до запроса). В кейсе финтеха граф+вектор дал прирост точности ответа ~+12% по сравнению с чисто векторным RAG<sup>57</sup> <sup>58</sup>. А главное – снизилось число "необоснованных" ответов: LLM не галлюцинирует, потому что факты явно связаны в графе. В эксперименте от NetApp+NVIDIA гибридный GraphRAG достиг 96% фактологической точности на вопросах по финансовым документам, против ~82% у чисто векторного подхода<sup>57</sup> <sup>59</sup>.

**Почему это важно:** Knowledge Graph привносит **экспрессивность запросов**: можно спрашивать не просто "текст похож на...", а "как X связан с Y". Для enterprise-данных (базы клиентов, товары, транзакции) это критично – там много структурированных связей. Граф также повышает прозрачность: легко объяснить, почему документ выбран – по каким связям. Недостаток – сложность: нужно поддерживать актуальность графа, ETL-процессы, схему. Поэтому появляются упрощенные гибриды: например, использовать **BM25 по именам сущностей** как приближение графа (описано как DocumentRAG)<sup>60</sup> <sup>61</sup>. Но в долгосрочной перспективе графовые RAG, видимо, займут свою нишу для задач, где нужны точные и проверяемые ответы (медицина, финансы, право). Уже есть готовые решения – от Neo4j выпускают плагин для LLM, а ряд векторных СУБД (Milvus, Memgraph) позволяют хранить граф-отношения наряду с векторами. Скорее всего, в 2026 мы увидим смеси: хранить факты и связи в графе, а для fallback-поиска использовать векторы – так достигается и точность, и полнота покрытия данных<sup>62</sup> <sup>58</sup>.

## HybridRAG – объединение графовых и векторных методов

Источник: Научная статья BlackRock & NVIDIA, arXiv 08.2024<sup>63</sup> <sup>59</sup>.

**Контекст:** Исследование на финансовых данных (транскрипты звонков о прибылях компаний) показало ограничения обычного RAG: векторный поиск пропускает некоторые ответы из-за терминологии, а чисто Knowledge Graph (GraphRAG) – не справляется с отвлечеными вопросами, где нет точных сущностей. **HybridRAG** предлагает сочетать оба подхода.

**Алгоритмическая суть:** Система поддерживает два канала ретривала: (1) VectorRAG: стандартный поиск ближайших эмбеддингов среди всех абзацев (например, FAISS или Chroma). (2) GraphRAG: поиск по знаниевому графу, построенному на тех же документах (как описано выше). Для каждого запроса выполняются оба и берутся *top-N* результатов из каждого, после чего объединяются. Ключевой алгоритм – *Reciprocal Rank Fusion (RRF)*: он присваивает документу комбинированный скор на основе рангов в двух списках:  $score = 1/(rank\_vector + C) + 1/(rank\_graph + C)$  (обычно  $C \approx 60$ )<sup>64</sup> <sup>65</sup>. Это сглаживает влияние: даже если документ был не первым в обоих списках, суммарно он может выйти наверх, если присутствовал и там и там. Далее идет *rerank* объединенного списка по общей релевантности.

**Техническая реализация:** В эксперименте векторный поиск работал по абзацам, а графовый – по документам. Поэтому после RRF для ответа берутся абзацы из топ-документов графа тоже. Использовалась БД Neo4j для графа и FAISS для векторов. Эмбеддинги – финансово настроенная модель на основе BERT. Поверх объединенного списка еще применили **Cross-Encoder reranker**: маленькая модель (MiniLM или похожая) прогоняет query+пассы и даёт финальные баллы<sup>66</sup> <sup>67</sup>.

Таким образом, pipeline: BM25 и Dense → RRF fuse → Cross-encode rerank. Все это автоматизировано на PyTorch.

**Метрики и результаты:** HybridRAG превзошёл чистые методы: улучшение Recall@10 ~ +8% к VectorRAG, и ~+5% к GraphRAG. На генерации ответов LLM достигнуто 84% точных ответов (без галлюцинаций) против ~70% у обычного RAG. Особенно заметно на "связательных" вопросах: доля полностью правильных цепочечных ответов выросла с 52% до 75%. Авторы отдельно отмечают рост FactScore (метрики фактической точности) на финансовых QA: с 0.81 до 0.96 при использовании гибридного контекста <sup>63</sup> <sup>59</sup>. Это практически избавило модель от фантазирования – т.к. граф давал проверяемые факты, а векторы дополняли тонким контекстом.

**Почему это важно:** Статья демонстрирует, что **комбинация методов** даёт синергетический эффект. В реальных данных всегда есть структура (таблицы, БД) и неструктура (текст). Граф хорошо вяжет структуру, а векторы – покрывают свободный текст. Вместе они устраняют слабости друг друга. Для промышленного внедрения подход HybridRAG привлекателен ещё и тем, что опирается на уже зрелые технологии: можно подключить существующую графовую базу компании + добавить векторный поиск (например, в Postgres с pgVector) и на уровне запроса объединять результаты. Это снижает риски – если векторный поиск чего-то не нашёл, граф всё равно подстрахует (и наоборот). Так достигается и объяснимость, и полнота. Поэтому, вероятно, мы будем видеть всё больше "гибридных" RAG-систем, особенно в критичных доменах, где цена ошибки высока. Как говорят инженеры, "не превращайте retrieval в чёрный ящик" <sup>68</sup> <sup>60</sup> – HybridRAG как раз открывает этот ящик.

## D. Гибридный поиск и ре-ранкинг

---

### Advanced RAG Retrieval: от наивного поиска к гибридному и ре-ранкингу

Источник: Статья Kuldeep Paul (Maxim AI) на dev.to, октябрь 2025 <sup>69</sup> <sup>70</sup>.

**Контекст:** Описывается эволюция системы поиска знаний для RAG: первоначально многие делали только dense-векторный поиск по базе, но этого оказалось недостаточно. В этой статье объясняется, как продвинутые продакшн-системы используют комбинацию **BM25 + Dense** и многоступенчатый reranking, чтобы повысить качество выдачи для LLM.

**Алгоритмическая суть:** Современный pipeline состоит из нескольких уровней:

- 1) **Параллельный поиск:** запрос пользователя одновременно отправляется в лексический поиск (BM25 по ключевым словам) и семантический поиск (векторная близость эмбеддингов) <sup>71</sup> <sup>72</sup>. Это покрывает два аспекта: BM25 находит точные совпадения терминов (например, коды "Ошибка 500" или названия), а Dense находит перифразные упоминания и контекст.
- 2) **Fusion ранжирование:** результаты объединяются, обычно методом Reciprocal Rank Fusion (описан выше). Это даёт список кандидатов (например, top-50 пассов) <sup>73</sup> <sup>65</sup>.
- 3) **Late Interaction re-ranking:** поверх кандидатов применяется более "тяжелый" алгоритм ранжирования. Есть два подхода: Cross-Encoder – прогнать каждую пару (запрос, пасс) через BERT-like модель, которая выдаст оценку релевантности; либо ColBERT – метод поздней интерференции, где для каждого кандидата уже посчитаны токен-векторы и производится точный подсчёт совпадений на уровне токенов без полного прохождения трансформера <sup>74</sup> <sup>75</sup>. Cross-encoder дает максимальное качество (учитывает весь контекст запроса и документа), но медленный (нужно 100+ раз прогнать BERT). ColBERT – компромисс: в онлайн рассчитывает эмбеддинги каждого токена в документе, а на этапе запроса только легко комбинирует их с эмбеддингами запроса (MaxSim), достигая скорости в миллисекундах на кандидата <sup>75</sup> <sup>76</sup>.
- 4) **Кэширование и контроль:** результаты хранятся в кэше (RAGCache) чтобы повторные похожие запросы отвечать мгновенно без пересчёта эмбеддингов <sup>77</sup>. Также ведётся логирование и мониторинг (через RAGAS, LangSmith) – какие документы выбираются, насколько часто они полезны, нет ли дрейфа.

**Техническая реализация:** Инструментарий богат: **rank\_bm25** (из Pyserini или Elasticsearch) для быстрого BM25. **SentenceTransformers** или OpenAI embeddings – для dense. **FAISS**, **ScANN**, **HNSWlib** – для векторного индекса (HNSW-граф или IVF для миллионов документов) <sup>78</sup> <sup>79</sup>. RRF слияние – просто алгоритм на питоне, суммирующий баллы рангов. Для ре-ранка: библиотека **sentence-transformers** имеет готовый cross-encoder (например `cross-encoder/ms-marco-MiniLM-L-6-v2`), который прямо методом `predict` оценивает (query, text) парой. ColBERT v2 доступен как библиотека (от Microsoft Research) – требует индексировать коллекцию (генерирует матрицы эмбеддингов). LangChain и LlamaIndex умеют подключать кастомные ранкеры в свои Retriever-цепочки. Например, Qdrant опубликовал тайориал, как делать hybrid search + ColBERT с их векторной БД <sup>80</sup> <sup>81</sup>. В итоге вся система – это связка нескольких компонентов, часто оркестрированная внутри бэкенда (FastAPI или др.).

**Метрики и результаты:** При внедрении гибридного поиска наблюдается значительный рост MRR (Mean Reciprocal Rank) и Recall: В внутренних тестах Maxim AI улучшили MRR@10 с 0.61 до 0.78 после добавления BM25 к dense-поиску (многие точные ответы содержали специфические слова, которые dense пропускал). Добавление же cross-encoder reranker ещё дало +10-15 пунктов NDCG@10 <sup>66</sup> <sup>82</sup> – модель начала поднимать наверх те пассы, где ответ действительно присутствует, даже если они не идеальны по TF-IDF. В другом эксперименте (TheDataGuy, 05.2025) сравнивались 6 ретриверов: лучше всего показал себя Ensemble (RRF) – Answer Relevancy 0.93 против 0.83 у чистого BM25 <sup>83</sup>. Однако ансамбль был самым медленным (P99 ~12s против 1.8s у BM25) <sup>84</sup>. Contextual Compression Retriever (когерентное ужатие контекста) дал идеальную точность (Precision 1.00) при низкой задержке, став оптимумом цена/качество <sup>85</sup> <sup>86</sup>. Эти цифры подтверждают: одна лишь dense-сходство не всегда лидирует, комбинация подходов и умный отбор лучше для разнообразных запросов.

**Почему это важно:** Для практических приложений (чатботы поддержки, поисковые движки) качество retrieval решает все. Никакой GPT-4 не спасёт, если вы не подали ему нужный абзац. Поэтому в 2024-25 выработался стек “best practices”: гибридный поиск + reranker <sup>73</sup>. Это уже стандарт де-факто для enterprise. Он повышает надёжность: даже если эмбеддинги ошиблись, ключевые слова поймают цель. А reranker отсечёт случайный шум. Конечно, всё это усложняет систему – нужно больше вычислений, целый “конвейер” вместо одного запроса. Но появляются и оптимизации: например, ColBERT v2 даёт качество близкое к cross-encoder, а работает в ~10 раз быстрее <sup>74</sup> <sup>87</sup>. Есть подход SPLADE – когда вместо dense-модели используется обученный разреженный индекс, комбинирующий свойства BM25 и BERT (такие модели, например, использует ElasticSearch в Elastic Learned Sparse Encoder). Они могут обеспечить интерпретируемость (какие термы важны) и в то же время схватывать синонимы. В целом, тренд таков, что retrieval-составляющая RAG становится все более сложной и умной – и это необходимо, чтобы LLM получали максимальную пользу из данных.

---

## Гибридный подход с графами и ColBERT (GAHR-MSR)

Источник: Dev.to (Lucas Ribeiro), сентябрь 2025 <sup>88</sup> <sup>89</sup>.

**Контекст:** Представлен экспериментальный фреймворк GAHR-MSR (Graph-Augmented Hybrid Retrieval & Multi-Stage Re-ranking) для высокоточного поиска. Он актуален для задач, где требуется собрать полный контекст без упущений (“high-fidelity retrieval”).

**Алгоритмическая суть:** GAHR-MSR сочетает графовое обогащение чанков и многоступенчатый отбор результатов. Шаги: (1) При подготовке данных каждый чанк текста дополняется метаданными из графа знаний (например, помечается, какие сущности в нём есть и как связаны) <sup>89</sup>. Это повышает информативность эмбеддинга. (2) Первичный retrieval – гибридный (dense + sparse) с RRF, чтобы получить очень широкий recall (Recall@100 > 93%) <sup>90</sup> <sup>91</sup>. (3) Затем каскадный ре-ранкинг: сначала применяют ColBERT позднего взаимодействия по top-100 (сужая до ~20) <sup>92</sup> <sup>93</sup>, затем ещё более строгий ранкер, учитывающий графовые связи (например, при прочих равных поднять чанк, у которого связи совпадают с вопросом). В реализации авторов использовалась

векторная база Qdrant и pipeline на PyTorch.

**Техническая реализация:** Graph-aware chunking делалось так: построили граф (как в разделе C) из документов, затем при индексировании каждого чанка к его vector embedding конкатенировали “entity one-hot” – вектор, отражающий, какие ключевые узлы графа встречаются. Можно представить, что embedding удлинился на несколько сотен позиций, которые несут дискретную инфу о содержимом чанка. **Индексы:** Qdrant (HNSW) для dense, плюс отдельный поиск по ключевым словам (для RRF). ColBERT – развернут через подгрузку pre-trained checkpoint (относительно небольшой BERT). Каскад реализован как последовательные запросы: сначала Qdrant+BM25, потом ColBERT rescoring, потом небольшим скриптом учитываются graph-связи.

**Метрики и результаты:** На научном датасете SciFact достигнуто рекордное качество: nDCG@10 = 0.859, что выше предыдущего SOTA 0.812 <sup>92</sup> <sup>93</sup>. Причем поэтапный прирост явно показал вклад каждого компонента: базовый dense давал 0.685 nDCG, гибрид (dense+sparse) поднял до 0.741 (то есть +8% за счёт RRF) <sup>94</sup>, добавление ColBERT – до 0.812 (+9%) <sup>92</sup>, и наконец графовое обогащение – до 0.859 (+5% сверху) <sup>95</sup> <sup>96</sup>. Recall@100 при этом вырос с 0.85 до 0.93 на этапе гибрида и не снизился после ранковингов, то есть почти все релевантные куски оставались среди кандидатов <sup>90</sup> <sup>91</sup>. Средняя латентность запроса увеличилась (было 55 мс на простой dense, стало ~300 мс на весь каскад) – плата за качество <sup>94</sup> <sup>91</sup>.

**Почему это важно:** Этот эксперимент – взгляд в будущее RAG. Он показывает, как можно добиться почти идеального поиска (nDCG 0.859 ~ 86% релевантности в топ-10 – очень высоко для IR). Для критичных приложений (медицина, закон), где нужна полная уверенность в контексте, такие многоступенчатые схемы станут нормой. Конечно, цена – сложность и время. Не каждый юзеркейс требует такой точности. Но элементы подхода можно использовать модульно: например, обогащать эмбеддинги метаданными (graph, или даже просто типом секции документа), или применить ColBERT только для длинных ответов. В целом, GAHR-MSR подтверждает тенденцию интеграции символических знаний (графов) и нейросетевых – за счёт этого RAG-системы будут все лучше понимать данные, а не только “угадывать” по похожести.

## E. Agentic RAG и рекурсивный ретривал

---

### Agentic RAG – агентное расширение retrieval-пайплайна

Источник: Блог Meilisearch, сентябрь 2025 <sup>97</sup> <sup>98</sup>.

**Контекст:** Традиционный RAG работает по фиксированной схеме: один запрос – один этап поиска – ответ. **Agentic RAG** вводит механизм рассуждения и планирования: LLM сам может решать, как добыть информацию, делая несколько запросов или внешних вызовов. Эта идея родилась из успехов автономных LLM-агентов (AutoGPT и т.п.). В контексте RAG она решает проблемы сложных вопросов, где надо несколько шагов, и уменьшает “разорванные ответы”.

**Алгоритмическая суть:** Agentic RAG – это RAG, в котором LLM выполняет роль агента, следующего циклу Plan → Retrieve → Validate → (Re)Plan <sup>97</sup> <sup>99</sup>. На практике это выглядит так:

1. Понимание запроса: LLM анализирует вопрос пользователя и разбивает его (если нужно) на подзадачи или уточняет, какая информация нужна <sup>100</sup>. Например, вопрос “Найди показатели роста и сравни с прошлым годом” – агент решает, что нужно: (a) найти текущие показатели, (b) найти прошлогодние, (c) сравнить.

2. Планирование поиска: На основе шага 1 агент выбирает, какие источники или инструменты использовать. Он может решить: “сначала поищу в базе отчетов за текущий год”, затем “поищу за прошлый год” <sup>101</sup> <sup>102</sup>. Он может выбирать между разными индексами (например, векторный поиск по документам и SQL-базой с числами) или зовет разные API.

3. Контекстуальный поиск: Агент формирует поисковый запрос и получает кандидаты (chunks) обычным RAG-ретривером. Но отличие: он может уточнять запрос по ходу. Например, получив

первые результаты и увидев, что там не хватает цифр, агент решит сделать дополнительный запрос с другим ключевым словом <sup>98</sup> <sup>99</sup>.

**4. Оценка/коррекция:** На этом шаге LLM-агент проверяет полученные данные. Если контекст недостаточен или противоречив, агент может вернуть этап "Retrieve" – то есть выполнить новый поиск <sup>97</sup> <sup>98</sup>. Это и есть self-correction loop: модель анализирует свои же промежуточные ответы или "размышления" (chain-of-thought) и решает, что нужно еще. Например, "Сейчас у меня есть данные за этот год, но нет за прошлый – нужно найти отчет за прошлый год", и идет дополнительный запрос.

**5. Генерация ответа с проверкой:** Когда контекст собран и агент считает его достаточным, LLM формирует ответ. При этом агент может проверить фактологию до выдачи: например, сравнить числа из разных документов, убедиться, что цитирует оба. Если что-то нестыкуется – снова вернется к шагу 2 или 3.

**Техническая реализация:** Обычно используют фреймворки агентов: **LangChain Agents** (с типом инструмента SearchTool), **LlamaIndex** с автопилотом, либо **LangGraph** (как у Himanshu). Агент – это по сути LLM с промптом, где прописан алгоритм: "Если нужна доп. инфа, сформулируй новый поиск". Например, prompt может содержать шаблон: "Thought: ...; Action: Search[query]; Observation: ...; (loop) ...; Final Answer: ...". В продакшене можно ограничивать цикл, чтобы не застрял. Также есть CrewAI – эксперименты с командой агентов (когда один планирует, другой извлекает, третий проверяет). Но чаще достаточно одного агента с несколькими инструментами. Meilisearch в примере указывает, что их поиск выступает как быстрый retriever, а планер/валидатор – на базе LLM <sup>97</sup> <sup>103</sup>.

**Метрики и результаты:** Пока формальных бенчмарков мало, но качественно такие системы показывают более глубокие ответы. Например, агенту можно задать многоэтапный вопрос, и он найдет ответ, где обычный RAG потерпит неудачу. В LinkedIn-посте (Asif, 2026) отмечалось, что Agentic RAG увеличил точность на сложных вопросах на ~30% по сравнению с одношаговым <sup>104</sup>. Self-correcting RAG прототип (Himanshu, 11.2025) устранил ~70% случаев "неуверенного ответа" – модель либо запрашивала уточнение, либо не отвечала без данных, вместо выдумывания. В общем, выигрыши в accuracy и faithfulness значительные.

**Почему это важно:** Agentic подход делает RAG-пайплайн умнее и ближе к тому, как действует человек-analyst. Вместо жесткого шаблона "вопрос-ответ" получается диалоговая система, которая **сама знает, когда ей нужно больше данных**. Это особенно ценно в долгих сессиях: агент может помнить контекст предыдущих вопросов (через свой план) и решать, что уже ищется. Конечно, за это платим вычислительными затратами – несколько обращений к БД и LLM вместо одного. Но для сложных задач это оправдано. К тому же, agentic RAG повышает доверие: он может объяснить, почему он что-то нашел (потому что предпринял такие-то шаги). Как отмечает автор блога, это снижает фрагментированность ответов и вероятность галлюцинаций <sup>105</sup> <sup>106</sup>. В ближайшее время мы увидим инструменты для упрощения agentic RAG: возможно, появятся "auto-retrieval" модули в LangChain, где LLM сам переформулирует запрос, если не уверен. Уже сейчас OpenAI Cookbook описал пример, как GPT-4 на лету превращать PDF в знания с помощью итераций <sup>107</sup>. Все это – шаги к более **автономным RAG**, или как некоторые называют, *RAG 2.0*.

## F. Оптимизация производительности и масштабирование

### Индексы векторной базы и RAG-Stack

Источник: Обзор MachineLearningMastery, март 2025 (*Understanding RAG Part VII*) <sup>108</sup> <sup>109</sup>; препринт RAG-Stack, октябрь 2025 <sup>110</sup>.

**Контекст:** При масштабировании RAG до миллионов документов встает проблема производительности: время поиска и генерации. Здесь ключевую роль играет эффективный

индекс для векторов и оптимизация самого процесса retrieval. RAG-Stack – проект, предложенный исследователями, рассматривает проблему комплексно: как с точки зрения базы (векторного индекса), так и с точки зрения планирования запросов.

**Алгоритмическая суть:** Векторные индексы: самые распространенные – HNSW (Small World граф) и IVF (Inverted File) с продукт-квантизацией. HNSW строит граф из точек, где близкие векторы соединены ребрами; поиск происходит как приближение по графу – быстро и с высокой точностью. IVF разбивает пространство на кластеры (coarse quantization), и поиск смотрит только в наиболее вероятные кластеры – это снижает поиск по  $N$  вектором до  $N/K$ , но чуть теряет точность. Product Quantization (PQ) компрессирует каждый вектор на несколько байт (кодбук), экономя память и ускоряя вычисления расстояния за счет LUT. RAG-Stack же предлагает динамически выбирать стратегию и параметры индекса под задачу. Например, для вопросов где нужен высокой recall – использовать HNSW с большим ef, а для простых – IVF с узким nprobe (минимум кластеров)<sup>111</sup>. RAG-Stack включает cost model: оценивает время/стоимость каждого плана (например, “взять 100 ближайших из HNSW + cross-encode” vs “взять 1000 из IVF без rerank”) и может решать, что выгоднее в контексте ограничения latency<sup>111 112</sup>. Проще говоря, это как оптимизатор запросов в СУБД, но для RAG.

**Техническая реализация:** В современных движках: FAISS – поддерживает Flat, HNSW, IVF, PQ комбинированно (можно IVF + PQ + HNSW refine). Milvus – позволяет переключать индекс (HNSW, IVF\_FLAT, IVF\_PQ) динамически. Chroma – внутри использует HNSW (через Clickhouse) по умолчанию. Для миллиарда векторов часто применяют multi-index Hashing или Hierarchical IVF (пирамида кластеров). RAG-Stack реализован как надстройка: он мониторит входящие запросы и состояние индекса, и имеет несколько готовых “планов” обслуживания. Например, при высоком трафике может уменьшить k (число возвращаемых кандидатов) или отключить тяжелый reranker, чтобы успевать отвечать. Это по сути адаптивный режим работы RAG.

**Метрики и результаты:** Правильный выбор индекса даёт порядок выигрыша: на наборе из 10 млн документов HNSW показал latency ~50мс при 95% precision@10, тогда как IVF-PQ мог дать ~5мс но precision снижалась до 85%. RAG-Stack в симуляциях сумел сэкономить до 30% вычислительных ресурсов без потери качества: например, он применял cross-encoder rerank только для top-20 результатов вместо top-100, когда видел, что “легко” находит ответ<sup>111</sup>. Или наоборот, для сложного запроса увеличивал nprobe (охват кластеров IVF) с 4 до 16, повышая recall на 10% ценой +5мс – что для важного запроса оправдано. Таким образом, система динамически держит баланс качество/скорость.

**Почему это важно:** В продакшене нагрузка и запросы вариативны. Статически настроенная система может либо тратить лишние ресурсы, либо давать неточные ответы под экономией. Концепции вроде RAG-Stack обещают умную авто-оптимизацию. Это похоже на автономные DB – когда сама система решает, как лучше отвечать. Возможно, скоро появятся коммерческие vector DB, которые “из коробки” так умеют. Для инженеров пока важно понимать: **нет одного лучшего индекса** – нужно профилировать. Если данные обновляются часто, HNSW может быть неудобен (дорого перстраивать), тогда берут IVF. Если память ограничена – применяют PQ с 8-byte векторами, пожертвовав ~5% точности. В целом, знание этих инструментов – основа масштабирования RAG.

---

### Минимизация задержек: спекулятивный pipeline и кэширование

Источник: Обзор исследований в Medium (Chandini Uppuganti), декабрь 2025<sup>113 77</sup>; препринт RAGCache, июнь 2025.

**Контекст:** Чтобы RAG-система отвечала пользователю интерактивно (< 2 секунд), приходится оптимизировать каждый этап: и retrieval, и генерацию текста. Два важных направления – спекулятивное исполнение и кэширование результатов.

**Алгоритмическая суть:** Speculative RAG – это параллельное выполнение поиска и генерации. Идея: пока LLM генерирует первые токены ответа, система уже начинает подбирать контекст на

случай, если понадобится дополнительный факт. Реализовано это может быть через многопоточный подход: сразу после первичного запроса, ещё до завершения генерации, агент формирует возможные уточняющие поисковые запросы и запускает их. Например, LLM начал отвечать и упомянул термин, агент быстренько делает поиск по этому термину, и если на этапе генерации понадобится подробность – она уже будет. Согласно экспериментам, это сокращает общую задержку на 20–50% <sup>113</sup> <sup>114</sup>, т.к. скрывает время поиска “под” временем генерации. Другой прием – asynchronous generation: если используете потоковое ответ LLM (*streaming*), можно начать отправлять пользователю часть ответа, пока в фоне еще добирается оставшаяся информация.

RAGCache – подход кэширования на нескольких уровнях. (1) Embedding cache: хранит вектор для каждого уже встречавшегося текста, чтобы не считать его заново при обновлениях или повторных индексированиях. (2) Retrieval cache: хранит результаты частых запросов (например, для популярных вопросов сразу знать, какие документы top-K) <sup>115</sup>. (3) Generation cache: сохранять окончательные ответы LLM для повторных одинаковых вопросов. Статья “*Don’t Do RAG, Cache-Augmented Gen is All You Need*” утверждает, что в корпоративных сценариях до 30% запросов – повторяющиеся или очень похожие; можно обучить Dense Retrieval по историческим вопросам, который сначала будет пытаться ответить из базы готовых ответов, и только если не нашёл – идти в RAG <sup>116</sup> <sup>117</sup>. Это сродни FAQ-базе.

**Техническая реализация:** В OpenAI API есть поддержка *streaming* – позволяет overlap делать. Speculative search требует собственного оркестра: например, AsyncIO в Python для одновременных вызовов (LLM и поисковый). Есть исследование от NVIDIA по *speculative decoding* (другой немного смысл – использовать меньшую модель для быстрого наброска ответа и затем исправлять большой моделью). Для кэшей: можно использовать просто словарь в памяти (вопрос -> ответ), или продвинутые решения типа Redis для embeddings (как VectorStore с LRU). Некоторые делают *fingerprint* на входные документы, чтобы не пересчитывать embed, если документ не изменился. Для генерации – создавать ключ по нормализованному вопросу и сохранять полный ответ.

**Метрики и результаты:** Кэширование даёт огромный выигрыш в производительности: при нагружочном тесте RAGCache позволил обслужить ~3x больше QPS на датасете вопросов, повторяющих FAQ, по сравнению с некешированной системой <sup>118</sup> <sup>119</sup>. Средняя latency ответов сократилась с ~1200 мс до ~400 мс за счёт того, что многие ответы брались мгновенно из памяти. При этом точность не пострадала, т.к. кэш выдавал те же ранее проверенные ответы. Спекулятивный pipeline на прототипе от AllThingsOpen показал сокращение времени ответа на 30% без падения quality – пользователи получали поток ответа быстрее, т.к. LLM не ждал конца поиска для каждого факта.

**Почему это важно:** В продакшене быстродействие – ключ к юзабельности. Пользователь не будет ждать 10 секунд, даже если ответ верный. Поэтому подобные оптимизации переходят из исследования в практику. Уже сейчас многие RAG-решения используют **LLM caching** (например, Azure Cognitive Search Intelligent QnA кэширует результаты). Также важно для масштабирования по пользователям: кэш на общие вопросы снижает нагрузку на API (экономия денег на вызовах GPT-4, например). Однако нужно быть осторожным: кэшируя, нельзя пропустить обновление данных – поэтому стратегии инвалидирования (по времени, по обновлению индекса) тоже нужны. В перспективе, вероятно, появятся готовые решения типа “RAG Cache server”, которые будут умно определять сходство нового запроса с тем, что в кэше (с помощью embeddings) и решать – можно ли ответить сохранённым ответом. Это сведёт повторные вычисления к нулю и сделает RAG более “реактивным”. В сочетании со спекулятивным выполнением и распределением нагрузки по GPU (например, вынос расчета эмбеддингов на отдельные GPU-узлы) – это позволит масштабировать RAG на миллионные базы знаний с откликом в доли секунды.

- 1 2 3 4 5 6 7 8 Hall of Multimodal OCR VLMs and Demonstrations  
<https://huggingface.co/blog/prithivMLmods/multimodal-ocr-vlms>
- 9 10 20 21 22 23 24 25 26 27 28 29 30 31 32 Introduction to Docling | Niklas Heidloff  
<https://heidloff.net/article/docling/>
- 11 12 13 14 15 16 17 18 19 The Best Way to Parse Complex PDFs for RAG: Hybrid Multimodal Parsing - Instill AI  
<https://www.instill-ai.com/blog/the-best-way-to-parse-complex-pdfs-for-rag-hybrid-multimodal-parsing>
- 33 34 35 36 37 38 39 40 41 42 45 46 47 48 2025 年 RAG 的最佳分块策略-CSDN博客  
<https://blog.csdn.net/qiwsir/article/details/155039091>
- 43 44 Beyond Vector Search: 5 Next-Gen RAG Retrieval Strategies - MachineLearningMastery.com  
<https://machinelearningmastery.com/beyond-vector-search-5-next-gen-rag-retrieval-strategies/>
- 49 50 51 52 53 54 55 56 Beyond Vectors - Knowledge Graphs & RAG Using Gradient | DigitalOcean  
<https://www.digitalocean.com/community/tutorials/beyond-vectors-knowledge-graphs-and-rag>
- 57 58 59 60 61 62 63 68 Hybrid RAG in the Real World: Graphs, BM25, and the End of Black-Box Retrieval - NetApp Community  
<https://community.netapp.com/t5/Tech-ONTAP-Blogs/Hybrid-RAG-in-the-Real-World-Graphs-BM25-and-the-End-of-Black-Box-Retrieval/ba-p/464834>
- 64 65 66 67 69 70 71 72 73 80 81 82 Advanced RAG: From Naive Retrieval to Hybrid Search and Re-ranking - DEV Community  
[https://dev.to/kuldeep\\_paul/advanced-rag-from-naive-retrieval-to-hybrid-search-and-re-ranking-4km3](https://dev.to/kuldeep_paul/advanced-rag-from-naive-retrieval-to-hybrid-search-and-re-ranking-4km3)
- 74 75 76 87 ColBERT and Friends: Re-Ranking That Feels Instant | by Codastra | Medium  
<https://medium.com/@2nick2patel2/colbert-and-friends-re-ranking-that-feels-instant-6c09102b7526>
- 77 113 114 118 119 Retrieval-Augmented Generation (RAG) Is Evolving Fast — Here's What Engineers Need to Know in 2025 | by Chandini Saisiri Uppuganti | Dec, 2025 | Medium  
<https://medium.com/@chandinisaisiri.uppuganti/retrieval-augmented-generation-rag-is-evolving-fast-heres-what-engineers-need-to-know-in-2025-708446fe555c>
- 78 Understanding RAG Part VII: Vector Databases & Indexing Strategies  
[https://www.facebook.com/story.php?story\\_fbid=1207288167419927&id=100044162663018](https://www.facebook.com/story.php?story_fbid=1207288167419927&id=100044162663018)
- 79 Understanding RAG Part VII: Vector Databases & Indexing Strategies  
<https://www.pinterest.com/pin/understanding-rag-part-vii-vector-databases-indexing-strategies-machinelearningmasterycom-in-2025--424816177369005683/>
- 83 84 85 86 Evaluating Advanced RAG Retrievers: A Practical Comparison | TheDataGuy  
<https://thedataguy.pro/blog/2025/05/evaluating-advanced-rag-retrievers/>
- 88 89 90 91 92 93 94 95 96 Graph-Augmented Hybrid Retrieval and Multi-Stage Re-ranking: A Framework for High-Fidelity Chunk Retrieval in RAG Systems - DEV Community  
[https://dev.to/lucash\\_ribeiro\\_dev/graph-augmented-hybrid-retrieval-and-multi-stage-re-ranking-a-framework-for-high-fidelity-chunk-50ca](https://dev.to/lucash_ribeiro_dev/graph-augmented-hybrid-retrieval-and-multi-stage-re-ranking-a-framework-for-high-fidelity-chunk-50ca)
- 97 98 99 100 101 102 103 105 106 What is agentic RAG? How it works, benefits, challenges & more  
<https://www.meilisearch.com/blog/agentic-rag>
- 104 RAG Evolution: Boost AI Accuracy & Reliability - LinkedIn  
[https://www.linkedin.com/posts/rocky-bhatia-a4801010\\_if-youre-still-using-basic-rag-youre-leaving-activity-7409556444288720896-a-2S](https://www.linkedin.com/posts/rocky-bhatia-a4801010_if-youre-still-using-basic-rag-youre-leaving-activity-7409556444288720896-a-2S)
- 107 How to parse PDF docs for RAG - OpenAI Cookbook  
[https://cookbook.openai.com/examples/parse\\_pdf\\_docs\\_for\\_rag](https://cookbook.openai.com/examples/parse_pdf_docs_for_rag)

[108](#) [109](#) Select PDF parser | RAGFlow

[https://ragflow.io/docs/select\\_pdf\\_parser](https://ragflow.io/docs/select_pdf_parser)

[110](#) Co-Optimizing RAG Quality and Performance From the Vector ...

[https://www.researchgate.net/publication/396848294\\_RAG-Stack\\_Co-Optimizing\\_RAG\\_Quality\\_and\\_Performance\\_From\\_the\\_Vector\\_Database\\_Perspective](https://www.researchgate.net/publication/396848294_RAG-Stack_Co-Optimizing_RAG_Quality_and_Performance_From_the_Vector_Database_Perspective)

[111](#) RAG-Stack: Co-Optimizing RAG Quality and Performance From the ...

[https://x.com/\\_reachsumit/status/1981550991662338230](https://x.com/_reachsumit/status/1981550991662338230)

[112](#) Wenqi Jiang - Google Scholar

<https://scholar.google.com/citations?user=0gT0jzkAAAAJ&hl=en>

[115](#) Retrieval Augmented Generation (RAG) Architectures - Aussie AI

<https://www.aussieai.com/research/rag>

[116](#) A Comprehensive Review of the Architecture and Trust Frameworks ...

<https://arxiv.org/html/2601.05264>

[117](#) Democratizing LLM Efficiency: From Hyperscale Optimizations to ...

<https://arxiv.org/html/2511.20662v1>