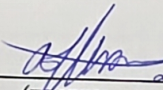
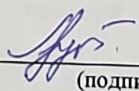
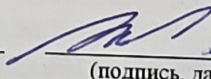
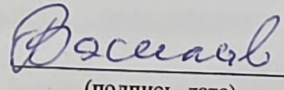


МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ
Федеральное государственное бюджетное образовательное учреждение высшего образования

**ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ**

Факультет «Вычислительная техника» (наименование)	Кафедра «Вычислительная техника» (наименование)
Направление подготовки	09.04.01 Информатика и вычислительная техника (код и наименование)
Профиль	Прикладной искусственный интеллект (наименование)

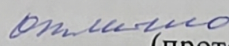
МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ
на тему:
ПРОГРАММНЫЕ СРЕДСТВА ГЕНЕРАЦИИ ДОКУМЕНТОВ
НА ОСНОВЕ ЗНАНИЙ О ПРЕДМЕТНОЙ ОБЛАСТИ

Студент	 (подпись, дата)	Иванов Александр Олегович (ФИО полностью)
Руководитель	 (подпись, дата)	Гурьянов Л.В. (фамилия, инициалы)
Нормоконтролёр	 (подпись, дата)	Бычков А.С. (фамилия, инициалы)
Рецензент	 (подпись, дата)	Василенко В.А. (фамилия, инициалы)

главный специалист
АО «НПП «Рубин»
(должность, место работы)

Работа допущена к защите

Заведующий кафедрой

Работа защищена с отметкой 
от 25.06.2025 № 15 (протокол заседания ГЭК)

Секретарь ГЭК

Пенза 2025

УТВЕРЖДАЮ

Зав. кафедрой ВТ

М.А. Митрохин

«29» 09 2023 г.

ЗАДАНИЕ

на выполнение магистерской работы

1. Студент гр. 23ВВИм1 факультета ВТ направление 09.04.01
Иванов Александр Олегович
(фамилия, имя, отчество)
2. Руководитель ВКР доцент кафедры МОиПЭВМ, ПГУ
(Должность и место работы)
Гурьянов Лев Вячеславович
(фамилия, имя, отчество)
3. Время выполнения магистерской работы с _____ по _____
4. Тема магистерской работы Программные средства генерации документов на основе знаний о предметной области
Тема утверждена приказом ПГУ № 0986/б от 29.09.2023
5. Задание на магистерскую работу: предмет и назначение разработки, функциональные, программные, технические требования и т.п.
 - 5.1. Разрабатываемый модуль предназначен для генерации документов на основе знаний о предметной области.
 - 5.2 Требования к функциональности: Модуль позволяет:
система должна автоматически формировать онтологию на основе входного текстового описания предметной области,
пользователь должен иметь возможность просматривать и редактировать автоматически сгенерированную онтологию,
принимать сообщения пользователей, и определять тип контекста входящего вопроса,
приложение должно предоставлять возможность выбора из перечня заранее загруженных шаблонов документов или загрузки собственного шаблона,
сгенерированный документ должен экспортироваться в формате DOCX,
интерфейс должен быть понятен пользователям без глубоких технических знаний.
 - 5.3 Требования к ИО: В качестве испытательного оборудования используются входные текстовые данные и шаблоны документов, поступающие от пользователей через веб-интерфейс и API системы. Хранилище онтологий и промежуточных результатов генерации документов реализуется на основе объектной системы RDF и файлового хранилища, обеспечивающих целостность и доступность данных для всех компонентов системы.
 - 5.4 Требования к ПО: Python, Streamlit, llama.cpp, Unsloth, rdflib.
 - 5.5 Требования к аппаратному обеспечению: Операционная система Microsoft Windows 10 и выше.

6. Содержание пояснительной записки магистерской работы: перечень подлежащих разработке, расчетов, обоснований, описаний

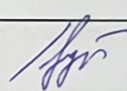
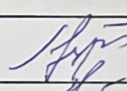
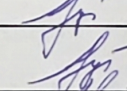
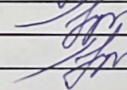
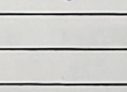
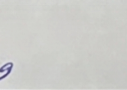
6.1 Анализ предметной области и аналогов, техническое задание

6.2 Проектирование программных средств

6.3 Разработка программных средств

6.4 Анализ результатов

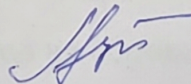
7. Календарный график выполнения магистерской работы

Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1. Анализ предметной области и аналогов, техническое задание	15 %	25.11.2023	
2. Проектирование программных средств	25 %	25.04.2024	
3. Разработка ПО	35%	25.07.2024	
4. Анализ результатов разработки	15%	25.03.2025	
5. Оформление ПЗ	10 %	30.05.2025	
6. Нормоконтроль		31.05.2025	

Дата выдачи задания

«29» 09 2023

Руководитель магистерской работы



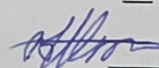
Гурьянов Л.В.

(фамилия, имя, отчество)

Задание к исполнению принял

«29» 09 2023

Студент

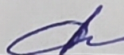


Иванов А.О.

(фамилия, имя, отчество)

Магистерскую работу к защите допустить

Декан ФВТ

 25.06.2025

/ Л.Р. Фионова

(Подпись, дата)

РЕФЕРАТ

Пояснительная записка содержит 115 листов, 15 рисунков, 21 источник, 2 приложения.

ГЕНЕРАЦИЯ ДОКУМЕНТОВ, ОНТОЛОГИИ, БОЛЬШИЕ ЯЗЫКОВЫЕ МОДЕЛИ, LORA, QLORA, UNSLOTH, GPT, CHATGPT

Объектом разработки является программный модуль для автоматизированной генерации документов на основе формализованных знаний о предметной области.

Цель работы – повышение эффективности и качества создания академических и административных документов в образовательных учреждениях посредством автоматизации генерации текстов на основе знаний о предметной области.

При разработке были последовательно решены следующие задачи: изучение предметной области и её формализация, проведение анализа существующих аналогов и подходов к автоматизированной генерации документов, обоснование выбора технологий и инструментов разработки, проектирование архитектуры и создание программного обеспечения модуля.

В результате выполнения работы был разработан программный модуль для автоматизированного создания документов на основе формализованных знаний о предметной области. Модуль реализован с использованием языка программирования Python, фреймворков FastAPI и Streamlit для разработки интерфейса и серверной части, большой языковой модели Phi-4-mini, адаптированной с помощью библиотеки Unsloth методом QLoRA, и формата хранения онтологий JSON.

Изм.	Лист	№ докум.	Подпись	Дата	ПГУ 1.090401.06.001 ПЗ			
Разраб.		Иванов А.О.		24.06.25	Программные средства генерации документов на основе знаний о предметной области	Лит.	Лист	Листов
Провер.		Гурьянов Л.В.		24.06.25			2	115
Н. Контр.		Бычков А.С.		24.06.25	Пояснительная записка	гр. 23ВВИм1		
Утв.		Митрохин М.А.		24.06.25				

СОДЕРЖАНИЕ

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	5
ВВЕДЕНИЕ	6
1 Анализ предметной области и аналогов, техническое задание	8
1.1 Обзор существующих подходов к генерации документов	8
1.2 Семантические модели как основа автоматизации	9
1.3 Техническое задание.....	10
1.4 Анализ существующих систем и платформ	11
1.4.1 OntoSyllabus	11
1.4.2 Curriculum and Syllabus Ontology	12
1.4.3 Outcome-Based Curriculum Ontology	13
1.4.4 1С:Университет	14
1.4.5 Апекс-ВУЗ	15
2 Программные средства генерации документов на основе знаний о предметной области	18
2.1 Архитектура программных средств генерации документов на основе знаний о предметной области.....	18
2.2 Структурные компоненты системы	19
2.2.1 Модуль пользовательского ввода.....	19
2.2.2 Модуль извлечения онтологии	19
2.2.3 Модуль генерации документов.....	19
2.2.4 Модуль экспорта документов	20
2.2.5 Модуль извлечения онтологии	20
2.3 Большая языковая модель	21
2.3.1 Архитектура трансформера	22
2.3.2 Подходы к дообучению БЯМ	25
2.3.3 LoRA – Low-Rank Adaptation.....	29
2.3.4 Обучение модели с использованием LoRA.....	31
2.3.5 Метод QLoRA.....	34

2.3.6 Использование PEFT и QLoRA в контексте дообучения на предметных онтологиях	37
3 Реализация программных средств генерации документов на основе знаний предметной области	42
3.1 Архитектура приложения.....	42
3.1.1 Основные компоненты и потоки данных	42
3.1.2 Основные модули и классы приложения	44
3.1.2.1 Класс OntologyExtractor.....	44
3.1.2.2 Класс OntologyManager	45
3.1.2.3 Класс DocxRenderer	47
3.1.2.4 Прочие компоненты.....	49
3.1.3 Взаимодействие модулей и классов.....	51
3.1.4 Используемые технологии и библиотеки	53
3.2 Описание процесса дообучения.....	56
3.2.1 Данные.....	56
3.2.2 Этапы обучения.....	57
4 Результаты реализации	58
4.1 Результаты дообучения языковой модели.....	58
4.2 Результаты работы с программными средствами генерации документов на основе знаний о предметной области	60
ЗАКЛЮЧЕНИЕ	66
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	68
ПРИЛОЖЕНИЕ А Листинг программы	70
ПРИЛОЖЕНИЕ А.1 Файл “docx_renderer.py”	71
ПРИЛОЖЕНИЕ А.2 Файл “llm_infer.py”	74
ПРИЛОЖЕНИЕ А.3 Файл “main.py”	77
ПРИЛОЖЕНИЕ А.4 Файл “ontology.py”	80
ПРИЛОЖЕНИЕ А.5 Файл “template_preview.py”.....	83
ПРИЛОЖЕНИЕ А.6 Файл “app.py”.....	87
ПРИЛОЖЕНИЕ Б Листинг программы.....	96

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

DOCX — формат текстовых документов Microsoft Word

Embedding (эмбеддинг) — векторное представление текстовой информации

FastAPI — веб-фреймворк для создания API

Fine-tuning — дообучение модели на специфических данных

Inference (инференс) — вывод модели, получение результата работы обученной нейронной сети

LLM (Large Language Model) — большая языковая модель

LoRA (Low-Rank Adaptation) — метод адаптации нейросетевых моделей с низким рангом

NLP (Natural Language Processing) — обработка естественного языка

Ontologies — формализованное представление знаний предметной области

Ontology-driven Generation — генерация текстов на основе онтологий

OWL (Web Ontology Language) — язык веб-онтологий

PEFT (Parameter-Efficient Fine-Tuning) — эффективное дообучение моделей с минимальным количеством параметров

Phi-4-mini — большая языковая модель от Microsoft

Prompts (промпты) — инструкции, подаваемые на вход языковым моделям для генерации текста

QLoRA (Quantized Low-Rank Adaptation) — квантованная адаптация нейросетевых моделей

RDF (Resource Description Framework) — инфраструктура описания ресурсов

Streamlit — библиотека Python для создания интерактивных веб-интерфейсов

Tokenization (токенизация) — разбиение текста на отдельные единицы (токены) для обработки языковыми моделями

Transformer — архитектура нейросетевых моделей

Vectorization (векторизация) — преобразование текстовых данных в числовые векторные представления

ВВЕДЕНИЕ

Одной из ключевых задач учебно-методической деятельности в системе высшего образования является подготовка и сопровождение различного рода документов, отражающих структуру и содержание предметной области. К таким материалам относятся рабочие программы, аннотации, справки, методические указания и иные регламентирующие документы, обеспечивающие нормативную обоснованность и соответствие требованиям образовательных стандартов.

В условиях активного внедрения цифровых платформ и модульных образовательных систем возрастает потребность в автоматизации процессов проектирования и генерации документов. Применение интеллектуальных технологий в данной области способствует снижению трудозатрат, повышению точности, согласованности и прозрачности разрабатываемой документации, а также улучшению управляемости образовательной среды.

Семантические модели, такие как онтологии, RDF-графы и концептуальные схемы (например, диаграммы Бахмана), позволяют формально описывать содержание предметной области. В отличие от произвольных текстовых формулировок, они задают чёткие связи между ключевыми понятиями, обеспечивая возможность автоматизированного анализа и обработки информации. Такой подход позволяет использовать информацию о предметной области в цифровых системах, облегчать ее обновление, сопровождение и генерацию разнообразных документов на ее основе в автоматическом или полуавтоматическом режиме [1].

Основной целью работы является создание программных средств генерации документов на основе знаний о предметной области.

Основными задачами работы является:

- анализ существующих решений и подходов к автоматизированной генерации документов в образовательной среде,
- изучение методов представления предметной области с использованием семантических моделей, включая онтологии, RDF-графы и диаграммы Бахмана,

- разработка и реализация программных средств генерации документов на основе знаний о предметной области,
- экспериментальная проверка работоспособности реализованного подхода на примере генерации документа.

Данная работа может быть полезна преподавателям, методистам, разработчикам образовательных систем и специалистам по качеству образования для автоматизации создания и анализа структурированных документов.

1 Анализ предметной области и аналогов, техническое задание

1.1 Обзор существующих подходов к генерации документов

Существующие подходы к созданию документов можно условно разделить на три категории: системы на основе шаблонов, большие языковые модели (БЯМ) и системы, основанные на онтологиях.

Системы на основе шаблонов часто интегрируются в информационные системы вузов, такие как 1С:Университет, Апекс-ВУЗ и их аналоги. Они используют реляционные базы данных для организации данных о учебных программах и курсах, и оснащены возможностью автоматического создания документов по заранее заданным макетам. Тем не менее, такие системы характеризуются жесткой привязкой к базам данных и не обладают необходимой гибкостью для применения с различными предметными областями.

Языковые модели, такие как GPT и LLaMA, имеют хорошую способность генерировать разнообразные тексты. Обученные на массивах текстовой информации, они способны самостоятельно изготовить содержательный текст для документа, основываясь лишь на кратком описании предметной области. Несмотря на высокую способность к передаче смысла, у этих моделей есть и свои недостатки: генерируемый текст может быть недостаточно структурированным, есть вероятность возникновения ошибок в документах (галлюцинации), могут возникнуть трудности с обеспечением соответствия формальным требованиям, например, государственным образовательным стандартам, таким как российский ФГОС.

Третий класс систем – онтологические платформы – обладает наибольшим потенциалом для автоматизации, поскольку использует машиночитаемые модели знаний [2]. Такие системы, как OntoSyllabus и CCSO, позволяют определить формальную структуру курса, включая темы, цели, рекомендуемую литературу и методы оценки, а затем автоматически генерировать документ на основе этой структуры. Онтологии поддерживают совместимость, машинную проверку и сопоставимость программ разных учебных заведений. Некоторые

платформы включают визуальные интерфейсы, основанные на ER-диаграммах или схемах Бахмана.

Таким образом, современные подходы охватывают как текстовую генерацию, так и структурную автоматизацию, а наиболее перспективными решениями являются гибридные системы, сочетающие сильные стороны обеих парадигм.

1.2 Семантические модели как основа автоматизации

Семантические модели знаний позволяют формализовать содержание курса, структуру образовательной программы и связи с внешними нормативами. Ключевыми стандартами семантического представления являются RDF и OWL. RDF описывает знания в виде графа триплетов, где каждое утверждение имеет форму субъект–предикат–объект. OWL расширяет RDF возможностью задавать классы, свойства, аксиомы, ограничения и логические выводы.

В контексте образования RDF/OWL используется для построения онтологий:

- онтологий дисциплин (тематика, ключевые понятия, цели обучения),
- онтологий компетенций (универсальных, профессиональных, e-CF),
- онтологий курсов и учебных планов (например, CCSO, OBC-ONTO),
- онтологий таксономий (например, таксономия Блума).

Визуальные концептуальные модели – такие как ER-диаграммы и диаграммы Бахмана – обеспечивают человеко-понятную форму представления семантической структуры. Они особенно удобны на стадии проектирования. Диаграммы Бахмана, в частности, позволяют выразить связи типа обобщение, принадлежность, ассоциация и используются для построения онтологий образовательных программ. Некоторые системы позволяют автоматически трансформировать диаграммы Бахмана в OWL-графы, сохраняя структуру и логику предметной области.

1.3 Техническое задание

Таким образом, использование семантических моделей обеспечивает автоматизацию не только генерации документов, но и валидации, логического контроля и поддержки межсистемного взаимодействия.

Цель работы

Повышение эффективности и качества генерации документов в образовательных организациях посредством автоматизированного создания документов на основе формализованных знаний о предметной области.

Описание разрабатываемого продукта Разрабатываемое программное средство представляет собой веб-приложение с графическим интерфейсом и серверной частью, использующей большую языковую модель, предварительно дообученную на специфических данных. Приложение позволяет пользователям вводить текстовое описание предметной области, просматривать и редактировать автоматически сгенерированную онтологию, выбирать шаблоны документов и экспортировать готовые документы в формате DOCX.

Входные данные

Система принимает:

- текстовое описание предметной области, введённое пользователем,
- пользовательские шаблоны документов (опционально).

Требования к разрабатываемому продукту.

Требования к надёжности:

- система должна быть доступна круглосуточно без перерывов на техническое обслуживание,
- работа системы должна осуществляться автономно, без необходимости постоянного участия технического персонала,
- время генерации документа не должно превышать одной минуты с момента отправки пользовательского запроса,
- пользователь всегда должен получать от системы результат выполнения запроса или понятное сообщение об ошибке.

Требования к функциональности.

- система должна автоматически формировать онтологию на основе входного текстового описания предметной области,
- пользователь должен иметь возможность просматривать и редактировать автоматически сгенерированную онтологию,
- приложение должно предоставлять возможность выбора из перечня заранее загруженных шаблонов документов или загрузки собственного шаблона,
- сгенерированный документ должен экспортироваться в формате DOCX и быть пригодным для дальнейшего редактирования стандартными текстовыми редакторами,
- интерфейс должен быть понятен пользователям без глубоких технических знаний и обеспечивать интуитивно понятное взаимодействие.

Требования к этической составляющей взаимодействия с пользователем:

- система не должна использовать, генерировать или предоставлять пользователю информацию, содержащую нецензурную лексику или материалы, нарушающие общепринятые этические нормы,
- перед первым использованием система должна предоставлять краткую инструкцию по работе с интерфейсом и возможностями приложения,
- система должна информировать пользователя о возможных ошибках и предоставлять рекомендации по их устранению.

Сроки реализации

Полностью функционирующая версия системы должна быть представлена к 25.06.2025 г.

1.4 Анализ существующих систем и платформ

Ряд современных систем демонстрирует реальное применение семантических моделей в образовательной практике.

1.4.1 OntoSyllabus

OntoSyllabus – это онтологическая модель, разработанная для формализованного описания структуры учебных дисциплин и автоматизированной генерации учебных документов, таких как syllabus и аннотации. Она представляет собой OWL-онтологию (Web Ontology Language),

в которой описаны основные элементы курса: темы, цели, результаты обучения, методы оценки, формы контроля, планируемые учебные действия и литература.

Основной целью OntoSyllabus является переход от неструктурированного текстового описания дисциплины к машиночитаемой модели, пригодной для автоматической генерации документов и их анализа. Онтология разрабатывалась как средство для повышения согласованности, повторного использования и совместимости академических курсов в рамках различных образовательных программ и учреждений.

Особенности:

- OntoSyllabus поддерживает экспорт учебных курсов в форматах RDF и OWL, что позволяет интегрировать её с внешними образовательными платформами (например, LMS) или аналитическими системами,

- модель обеспечивает согласование целей и результатов курса с таксономией Блума, что позволяет контролировать когнитивную нагрузку и уровень освоения материала,

- онтология расширяема и настраивается под конкретные потребности образовательной организации.

OntoSyllabus используется в научных и образовательных проектах как инструмент для проектирования курсов, автоматизации составления учебных планов и оценки полноты охвата компетенций. Также она может служить основой для систем управления качеством образования [3].

1.4.2 Curriculum and Syllabus Ontology

Curriculum and Syllabus Ontology (CCSO) – это онтологическая модель (рисунок 1), разработанная для формализованного описания структуры учебных планов и силлабусов с учётом международных требований и междисциплинарного взаимодействия. Онтология предназначена для поддержки проектирования, анализа, согласования и повторного использования учебных программ, в том числе в международных образовательных проектах.

CCSO предназначена для решения задач интероперабельности между образовательными учреждениями и системами, работающими с куррикулами

и программами дисциплин. Её структура направлена на детальное представление учебного плана и каждой дисциплины, включая связи между результатами обучения, компетенциями и оценочными методами [4].

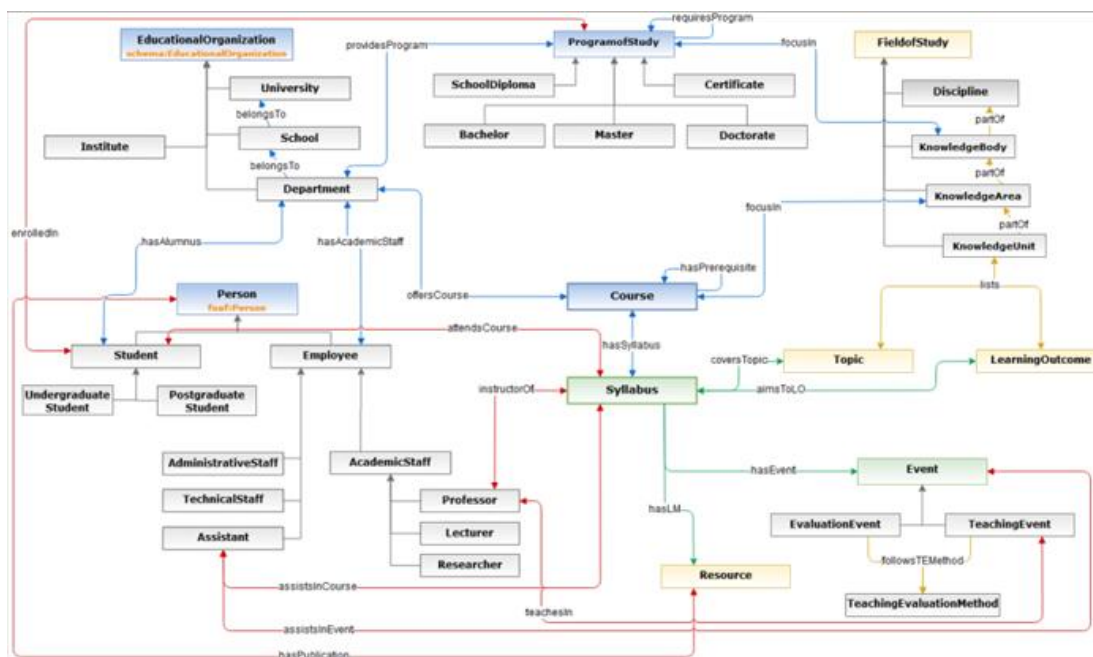


Рисунок 1 – Интерфейс работы с онтологией в CCSO

1.4.3 Outcome-Based Curriculum Ontology

OBC-ONTO (Outcome-Based Curriculum Ontology) – онтология, разработанная в контексте outcome-based education (обучения, ориентированного на результаты). Основное назначение модели – чёткое сопоставление целей курса, результатов обучения, методов оценки и содержания дисциплины.

OBC-ONTO помогает обеспечить прозрачность образовательного процесса за счёт прямой формализации связей: цель – результат – содержание – оценка. Онтология ориентирована на проектирование дисциплин в инженерных и технических специальностях, особенно там, где требуется соответствие международным стандартам аккредитации (например, ABET).

OBC-ONTO может применяться в системах проектирования курсов, а также в рамках автоматизированной генерации отчётов по результатам

аккредитации. Также модель используется в инструментах визуального анализа взаимосвязей между элементами курса (рисунок 2) [5].

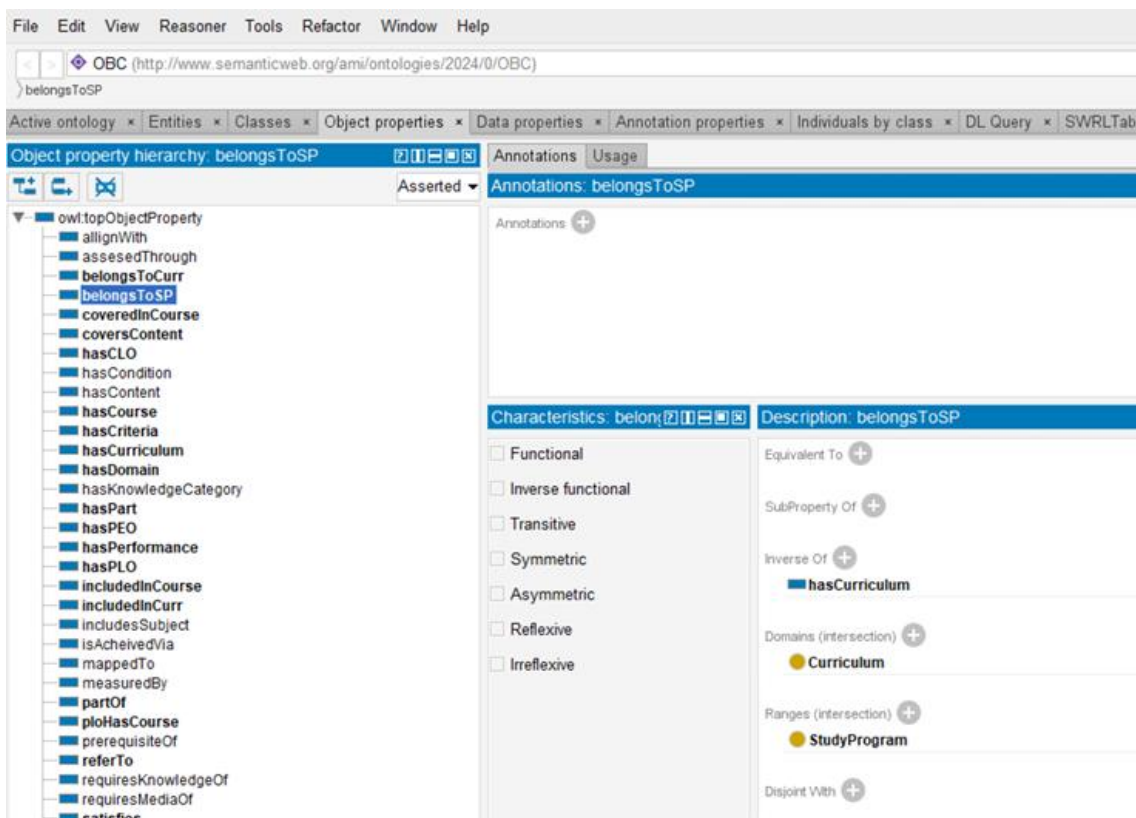


Рисунок 2 – Интерфейс программы OBC-ONTO

1.4.4 1С:Университет

1С:Университет – это модульная информационная система, созданная на платформе 1С:Предприятие, предназначенная для комплексной автоматизации управления учебным процессом в вузе. Включает модули для организации учебной и научной деятельности, кадрового учета, финансов, работы с абитуриентами и студентами.

Система предназначена для обеспечения административной и учебно-методической деятельности вуза, упрощения документооборота, учета, планирования и анализа. Она фокусируется на поддержке образовательного процесса с учётом специфики вузовских требований, таких как ФГОС, контроль часов, составление отчетов для Минобрнауки и т. д.

Особенности:

- использует таблично-реляционную модель, хранит учебные данные в виде структурированных таблиц,

- обладает мощной платформой для расширения и адаптации, но требует высокой квалификации специалистов 1С для модификации форм и шаблонов,
- не предусматривает семантического описания предметной области, поэтому автоматизация ограничена шаблонным документооборотом,
- часто интегрируется с внешними LMS или онтологическими слоями при помощи собственных API.

1С:Университет применяется как корпоративное решение в крупных университетах и является популярным выбором для вузов, использующих экосистему 1С. Подходит для административной автоматизации, однако требует внешних расширений или интеграции для реализации семантического подхода к моделированию дисциплин (рисунок 3) [6].

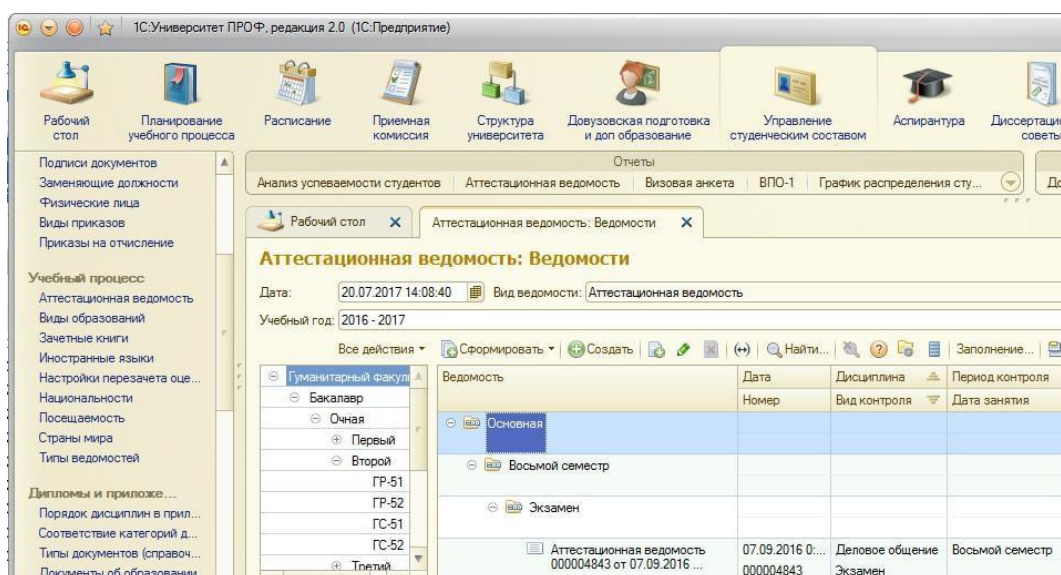


Рисунок 3 – Интерфейс программы 1С:Университет

1.4.5 Апекс-ВУЗ

Апекс-ВУЗ – это специализированная информационная система для автоматизации управления учебной, методической и административной деятельностью в высших учебных заведениях. Она ориентирована на обеспечение работы всех подразделений вуза: деканатов, учебных отделов, кафедр, методических служб, бухгалтерии и т. д.

Основная цель системы – централизованное управление учебным процессом, автоматизация составления расписаний, расчётов нагрузки,

формирования отчетных документов и шаблонных форм рабочей документации (включая рабочие программы дисциплин, учебные планы, приказы и пр.).

Особенности:

- основана на реляционной модели данных и использует встроенные редакторы шаблонов,
- поддерживает генерацию документов по утвержденным вузом или министерством шаблонам, но ограничена возможностью их гибкой настройки,
- не использует семантические модели знаний (например, онтологии), что ограничивает возможности для автоматического анализа или трансформации содержания.

Апекс-ВУЗ (рисунок 4) широко используется в российских вузах, особенно региональных, как централизованное решение для документооборота в образовательной сфере. Подходит для работы по строгим регламентам и при необходимости стандартизированной отчётности, но требует ручного ввода и шаблонной подготовки содержания [7].

The screenshot displays the Apex-VUZ web application interface. On the left is a dark sidebar with navigation links. The main content area shows a table of students under the heading 'Обучающиеся'. The table has columns for student ID, name, achievements, position, subgroup, basis of learning, and debt. The 'ЗИП-20' course is selected in the left sidebar.

#	ФИО	Достижения	Позиция	Подгруппа	Основание обучения	Долги
1	Евтушок Олеся Романовна	✓ 2	гр: 2, к: 1	1	Бюджет	21
2	Емельянов Александр Владимирович		гр: 3, к: 1	1	Внебюджет	21
3	Завгородняя Анна Юрьевна	✓ 1	гр: 20, к: 1	1	Не указано	20
4	Копцев Дмитрий Вадимович	✓ 2	гр: 19, к: 1	2	Не указано	5 / 16
5	Кретьинин Дмитрий Сергеевич		гр: 4, к: 1	1	Не указано	20
6	Кузнецова Анастасия Дмитриевна		гр: 5, к: 1	1	Не указано	20
7	Лебедев Данила Николаевич		гр: 6, к: 1	1	Не указано	20
8	Мальцев Дмитрий Владимирович		гр: 7, к: 1	1	Не указано	20
9	Мальцева Виктория Сергеевна		гр: 8, к: 1	2	Не указано	20
10	Никитин Илья Владимирович		гр: 9, к: 1	2	Не указано	20
11	Первых Денис Александрович		гр: 10, к: 1	2	Не указано	20
12	Сайфидинов Бахридин Ахлидинович		гр: 11, к: 1	2	Не указано	20
13	Сайфидинов Илья Геннадьевич		гр: 12, к: 1	2	Не указано	20

Рисунок 4 – Интерфейс ИС Апекс-ВУЗ

В таблице 1 представлены ключевые характеристики четырёх систем, использующихся для автоматизации проектирования, ведения и генерации учебной документации в высших учебных заведениях. Сравнение охватывает модель данных, наличие поддержки семантического представления знаний, возможности генерации документов, а также применимость к образовательным стандартам.

Таблица 1 – Сравнение аналогов

Система	Тип модели	Поддержка онтологий / RDF	Гибкость настройки	Генерация документов
OntoSyllabus	OWL-онтология	Да (OWL, RDF)	Высокая (расширяемая онтология)	Автоматическая генерация курса
CCSO	RDF/OWL-онтология	Да (полная поддержка)	Высокая (open-source, расширяемая)	Аннотирование, экспорт учебных курсов
OBC-ONTO	OWL-онтология	Да (OWL, RDF, reasoning)	Средняя – задана рамочно по ОВЕ-принципам	Автоматические проверки согласованности, поддержка reviewers
Апекс-ВУЗ	Реляционная модель	Нет	Средняя (фиксированные шаблоны)	Шаблонная генерация программ и планов
1С:Университет	Реляционная модель	Нет	Высокая (через настройку 1С-конфигураций)	Шаблонная отчётность, планы, приказы

2 Программные средства генерации документов на основе знаний о предметной области

2.1 Архитектура программных средств генерации документов на основе знаний о предметной области

Архитектура программных средств генерации документов на основе знаний о предметной области на основе описания дисциплины и формализованных структур знаний представлена на рисунке 5.

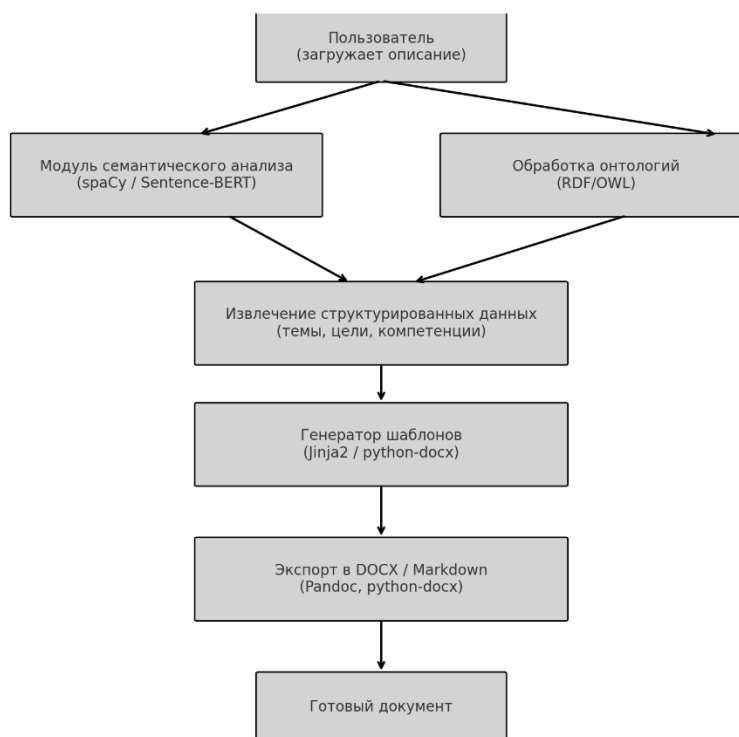


Рисунок 5 – Архитектура автоматизированной системы планирования учебной нагрузки преподавателей с интеллектуальным помощником

Архитектура информационной системы построена по модульному принципу. Модули взаимодействуют друг с другом последовательно, что упрощает архитектуру.

Для разработки информационной системы будет использоваться конвейерный принцип: начиная от ввода пользовательских данных, заканчивая заполнением шаблона документа.

2.2 Структурные компоненты системы

2.2.1 Модуль пользовательского ввода

Предоставляет интерфейс для ввода пользователем исходной информации о предметной области. Пользователю будет доступно поле для ввода информации о предметной области, а также будет доступна функциональность загрузки текстовых документов.

Предполагается наличие возможности загрузки документов формата Microsoft Office Word (*.doc, *.docx).

2.2.2 Модуль извлечения онтологии

Производит формализацию входного текста и перевод его в более структурированный и формализованный формат RDF. Предполагается возможность предпросмотра пользователем полученной онтологии для ручного редактирования.

При формировании онтологий модулем будет использоваться информация о более общих предметных областях, например об академической предметной области, либо об экономической.

Например, для случая, когда пользователю может понадобится составить описание дисциплины, модуль может извлечь следующую информацию из пользовательского текста:

- тема – “основы алгоритмизации”,
- результат – “способность применять базовые алгоритмы”,
- способ проверки – “практическое задание”.

Данный способ описания информации из предметной области позволит упростить дальнейшую машинную обработку информации, например заполнение шаблонов документов.

2.2.3 Модуль генерации документов

Модуль отвечает за генерацию текстового представления документа на основе заранее подготовленного шаблона, подготовленного заранее для более общей области деятельности: например, набор академических документов описывающий перечень работ для выполнения студентами.

Для работы с обобщенными данными модуль должен будет реализовать логику для заполнения шаблона перечислениями и повторяющимися блоками.

Пример возможного описания шаблона для данного модуля приведен в листинге 1.

Листинг 1 – Пример описания шаблона

```
Цель дисциплины: {{ goal }}
Основные темы:
{% for topic in topics %}
- {{ topic }}
{% endfor %}
```

На основе заданного шаблона пользователь может получить документ, приведенный в листинге 2.

Листинг 2 – Документ на основе шаблона

Цель дисциплины: Изучение основ алгоритмизации.

Основные темы:

- Линейные алгоритмы
- Условные операторы
- Циклы

Ожидаемые результаты:

- Уметь разрабатывать простые алгоритмы.

2.2.4 Модуль экспорта документов

Формирует готовый документ на основе заполненного шаблона. Позволяет получить документ в формате docx.

2.2.5 Модуль извлечения онтологии

Для извлечения онтологии предполагается использовать большую языковую модель (БЯМ). Для того чтобы адаптировать уже существующие модели под работу с языками описания онтологий, такими как RDF, будет использоваться процесс, называемый файнтюнингом (finetuning).

Файнтюнинг представляет собой процесс дообучения модели под работу с конкретными обучающими данными. Процесс файнтюнинга значительно менее сложен, чем процесс обучения всей модели.

Преимуществом использования большой языковой модели в данном контексте будет возможность работы модуля с любым набором непредсказуемых данных.

Основным преимуществом разрабатываемого модуля будет способность не только выполнять анализ текстового описания предметной области, но и формировать структурированное описание предметной области с учетом смысла, не основываясь лишь на поверхностном совпадении слов. Для этого будет использоваться возможность больших языковых моделей по донастройке уже обученной модели.

Возможность моделей выполнять текстовые инструкции позволит получить от модели структурированное описание элементов предметной области в необходимом для функционирования модуля текстовом формате.

Для такой задачи подойдут модели, которые умеют учитывать смысл слов не только по отдельности, но и в связке с другими. Важно, чтобы модель могла работать с контекстом и адаптироваться под формат нужного вывода. Этим требованиям соответствуют архитектуры, построенные на трансформерах. Они хорошо справляются с текстами, где значение определяется не только самими словами, но и тем, как они расположены друг относительно друга.

2.3 Большая языковая модель

Современные большие языковые модели базируются на архитектуре трансформеров, которая стала прорывом в обработке естественного языка. До появления трансформеров основой нейросетевых моделей для работы с последовательностями были рекуррентные сети (RNN) и их усовершенствованные варианты (LSTM, GRU). Однако рекуррентные архитектуры имели ряд ограничений: они обрабатывали текст последовательно, что затрудняло параллелизацию, и испытывали трудности с сохранением долговременных зависимостей в длинных текстах. Публикация в 2017 году работы “Attention is All You Need” ознаменовала переход к новой парадигме, показав, что можно отказаться от рекуррентности в пользу механизма внимания, существенно улучшив качество и эффективность моделирования языка.

2.3.1 Архитектура трансформера

Архитектура трансформера основана на механизме самовнимания (self-attention), позволяющем модели анализировать взаимосвязи между любыми парами слов во входной последовательности. В традиционной рекуррентной сети значение каждого следующего слова обрабатывается с учётом скрытого состояния, постепенно перенося информацию вдоль последовательности. В трансформере же внимание непосредственно связывает каждое слово с другими словами, вне зависимости от расстояния между ними. Это достигается путём вычисления весовых коэффициентов внимания: для каждого слова модель вычисляет вектор “запроса” (query) и сравнивает его со всеми “ключами” (key), соответствующими другим словам, чтобы определить, насколько они релевантны друг другу. На основе этих весов затем формируется взвешенная комбинация “значений” (value) всех слов, что и даёт обновлённое представление текущего слова с учётом контекста. Такой механизм самовнимания позволяет эффективно учитывать дальние зависимости в тексте – например, связь между подлежащим в начале предложения и сказуемым в конце.

Ключевым компонентом архитектуры является многоголовое внимание (multi-head attention). Вместо одного единственного веса внимания трансформер использует несколько параллельных «голов» внимания, каждая из которых фокусируется на разных аспектах взаимосвязей. Эти головы работают в разных проекциях скрытого пространства и могут улавливать различные типы зависимостей: одна голова, условно, может отражать синтаксические связи (например, соответствие подлежащего и сказуемого), другая – семантические (например, тематическую связанность слов). Результаты от всех голов конкатенируются и совместно обрабатываются, что позволяет получить более богатое представление контекста для каждого слова. После слоя внимания в трансформере следует позиционно независимый полносвязный слой (feed-forward network) – одинаковый для каждого положения в последовательности – который нелинейно преобразует полученные признаки. Наличие резидуальных связей и операций нормализации (например, layer normalization) между слоями

обеспечивает устойчивость градиентов при обучении даже для очень глубоких сетей [8].

Особое внимание уделяется учёту порядка слов, так как механизм внимания сам по себе не включает информацию о позиции. В базовом решении используются позиционные эмбединги – специальные векторные представления, добавляемые к токенам, которые кодируют их индекс или относительное положение в последовательности. Это может быть реализовано с помощью детерминированных функций (например, синусоидальные гармоники, как в оригинальном предложении трансформера) или с помощью обучаемых векторных позиций. В современных моделях также применяются усовершенствования позиционного кодирования, например, ротаторные позиционные эмбединги (RoPE), улучшающие обобщающую способность модели на больших длинах контекста. Таким образом, трансформер получает информацию не только о самих словах, но и об их расположении, что необходимо для корректного синтаксического и семантического разбора предложений.

Архитектура трансформера может работать в различных режимах – с энкодером, декодером или их комбинацией. В исходной реализации энкодер-декодер решал задачи “последовательность-в-последовательность” (например, машинный перевод), где энкодер обрабатывает исходный текст, получая контекстные представления, а декодер генерирует новый текст (перевод) на их основе. Для генеративных языковых моделей, нацеленных на продолжение текста, часто используется только декодерная часть трансформера. В таком авторегрессионном режиме декодер на каждом шаге видит предыдущие сгенерированные слова (это реализуется за счёт маскирования механизма внимания, запрещающего доступ к будущим токенам) и предсказывает следующий токен. Модели семейства GPT, например, представляют собой именно многослойный Transformer-декодер, обученный предсказывать продолжение текста по предыдущему контексту. Другая категория моделей, таких как BERT, использует только часть энкодера трансформера и обучается

восстанавливать пропущенные слова в тексте (masking), что позволяет им приобретать глубокое понимание языковых конструкций без непосредственного обучения генерации текста.

Главным преимуществом трансформерной архитектуры является её способность к масштабированию и параллельной обработке данных. В отличие от рекуррентных сетей, трансформеры не требуют последовательного прохождения сигнала через каждое слово – благодаря механизму внимания и позиционному кодированию все элементы последовательности обрабатываются одновременно, что хорошо распараллеливается на современных аппаратных ускорителях (GPU, TPU). Это открывает возможность обучения на гораздо большем объёме данных и использования моделей с огромным числом параметров. На практике трансформеры продемонстрировали отличную способность к масштабированию: увеличение числа слоёв, ширины слоёв (размерности эмбедингов) и числа голов внимания приводит к существенному росту выразительности модели. В совокупности с обучением на масштабных корпусах текста (порядка миллиардов слов и более) это позволило создать так называемые фундаментальные модели (foundation models) – универсальные языковые модели, содержащие обобщённые знания из множества источников.

Благодаря трансформерной архитектуре современные большие языковые модели достигают впечатляющих результатов. Модели вроде GPT-3, GPT-4, PaLM, LLaMA и другие содержат десятки и сотни миллиардов параметров, что стало возможным именно за счёт эффективной архитектуры с механизмом внимания. Такая архитектура позволяет модели учитывать тончайшие нюансы контекста и выбирать релевантные факты из своего огромного запаса знаний, накопленного на этапе предварительного обучения. Кроме того, трансформеры относительно легко адаптируются под различные задачи: после предобучения на общем корпусе текстов модель можно дообучить для конкретной задачи или домена. Эта универсальность и гибкость, заложенные на архитектурном уровне, объясняют, почему трансформеры стали стандартом де-факто для LLM. С их помощью модель может воспринимать длинные входные тексты (контекст

объёмом в несколько страниц) и генерировать связные осмысленные ответы, сохраняя при этом согласованность и стилевое единство. Таким образом, трансформеры сформировали ту самую архитектурную основу, на которой строятся все современные достижения в области больших языковых моделей.

2.3.2 Подходы к дообучению БЯМ

Классический подход к дообучению языковой модели предполагал тонкую настройку (fine-tuning) всех её параметров на новые данные. Иными словами, после этапа предварительного обучения на массивном корпусе текстов модель дополнительно обучалась на целевом датасете для конкретной задачи, причём градиентным спуском обновлялись все веса нейросети. Этот метод хорошо работал для относительно небольших моделей первых поколений (таких как BERT с ~110 млн параметров), позволяя адаптировать универсальную модель под задачи классификации, ответов на вопросы и т. п. Однако по мере роста размеров моделей данный подход стал наталкиваться на серьёзные трудности. Во-первых, полное дообучение громадной модели требует огромных вычислительных ресурсов и памяти: при миллиардах параметров нужно хранить не только сами веса, но и градиенты таких же размеров, что выходит далеко за возможности одиночного графического процессора среднего класса. Во-вторых, обновление всех параметров повышает риск переобучения на небольшом целевом датасете и может привести к «забыванию» моделью ранее приобретённых знаний (явление катастрофического забывания). Наконец, для каждой новой задачи приходится хранить полный набор параметров модели, что затрудняет распространение и развёртывание таких специализированных версий, учитывая их гигантский объём.

Осознание этих ограничений привело к развитию методов параметрически эффективного дообучения (Parameter-Efficient Fine-Tuning, PEFT). Суть PEFT заключается в том, чтобы адаптировать модель к новой задаче, изменяя лишь небольшую долю её параметров либо добавляя относительно немного новых весов, тогда как основная масса параметров остаётся неизменной. Таким образом достигается несколько целей одновременно: сокращается объём вычислений и

требуемой памяти, уменьшается риск переобучения, а базовые знания модели сохраняются. Замораживание большинства слоёв означает, что модель не теряет общей языковой компетентности и ранее усвоенных фактов, а обучение концентрируется только на тех новых параметрах, которые отвечают за конкретную задачу. Экспериментально было показано, что, несмотря на столь сильное сокращение числа обучаемых весов, качество модели на новой задаче часто почти не снижается по сравнению с полным finetuning. Например, при добавлении специальных адаптерных слоёв объёмом всего около 3–4 % от исходного числа параметров модель может достичь качества, отстающего менее чем на один процентный пункт от варианта с обучением всех параметров. Это значит, что большая часть весов действительно не требует подстройки для решения новых узких задач, и их замораживание не мешает модели обучиться необходимым навыкам [9].

Были предложены различные подходы к параметрически эффективному дообучению. Один из ранних методов – адаптеры (Adapters), внедряемые непосредственно в архитектуру модели. В этом подходе в каждом слое трансформера добавляются небольшие дополнительные блоки с сильно уменьшенной внутренней размерностью. Внутри такого адаптера входные d -мерные признаки сначала проецируются в гораздо меньшее пространство (например, размерности m), затем проходят через нелинейность, после чего проецируются обратно в d -мерное пространство, плюс предусмотрена сквозная связь (skip-connection) между входом и выходом адаптера. Параметров у адаптеров на несколько порядков меньше, чем у самой модели. При дообучении обновляются только веса этих дополнительных слоёв, тогда как исходные веса трансформера остаются неизменными. Исследования показали высокую эффективность адаптеров: модель почти не теряет в качестве, хотя численно изменяется лишь несколько процентов параметров. Более того, адаптеры обладают модульностью: их можно отключать или переключать, используя одну и ту же базовую модель для разных задач и подгружая соответствующий адаптер для каждой из них [10].

Другой важный класс методов PEFT – так называемое промт-обучение (prompt tuning) и родственные техники. Эти подходы не добавляют новых слоёв внутрь модели, а вводят дополнительные обучаемые векторы на входе или на промежуточных этапах обработки. В простейшем варианте используется набор soft prompts – виртуальных токенов, которые вставляются в начало входной последовательности, причём их эмбединги обучаются совместно с моделью для оптимального выполнения требуемой задачи. Более развитый вариант – prefix tuning (настройка префиксов), когда к каждому блоку трансформера добавляется специальная последовательность из нескольких обучаемых векторов (префикс). Эти векторы служат как бы дополнительным контекстом для модели: при тренировке они настраиваются так, чтобы направлять модель к нужным ответам, тогда как исходные веса не меняются. Важно, что ни параметры основных слоёв, ни сами исходные токены при этом не изменяются – подстраиваются лишь добавленные «виртуальные» векторы. Это делает метод весьма универсальным: его можно применить практически к любой трансформерной модели без изменения её архитектуры и без большого числа новых параметров [11].

Особого упоминания заслуживает метод LoRA (Low-Rank Adaptation), ставший одним из наиболее популярных и эффективных PEFT-подходов для больших языковых моделей. LoRA был разработан в 2021 году и эксплуатирует гипотезу о том, что изменение весов при дообучении имеет низкую внутреннюю размерность. Проще говоря, когда мы адаптируем огромную модель к новой задаче, фактические изменения в матрицах весов можно выразить как матрицу низкого ранга. LoRA воплощает эту идею, вводя для некоторых крупных весовых матриц (в практике – матриц внутри механизмов внимания или проекций в FFN-блоках) их низкоранговые поправки. Вместо того чтобы напрямую модифицировать существующую матрицу размерности $N \times M$, метод добавляет к ней произведение двух гораздо меньших матриц размерностей $N \times r$ и $r \times M$. Параметр r (ранг аппроксимации) выбирается относительно небольшим (как правило, 4, 8 или 16), так что число новых параметров составляет лишь малую долю от $N \times M$. Эти добавочные матрицы

инициализируются, например, нулевыми значениями, после чего обучаются на новом датасете, тогда как исходная матрица остаётся замороженной. При использовании модели в работе базовые (предобученные) веса и выученная низкоранговая поправка суммируются – таким образом, эффекты от дообучения накладываются на изначальные знания модели, не затирая их. Метод LoRA показал впечатляющие результаты: за счёт всего нескольких миллионов дополнительных параметров можно достичь качества, сравнимого с полным finetuning огромной модели, при этом требуя в десятки раз меньше памяти на этапе обучения [12].

Общим свойством всех описанных подходов является значительное снижение ресурсных затрат на адаптацию моделей. Поскольку подавляющая часть параметров модели не изменяется, нет необходимости хранить и обновлять градиенты для них, что существенно экономит память. Время обучения также сокращается, ведь оптимизировать приходится намного меньшее число весов. Эти методы сместили парадигму разработки: теперь вместо того, чтобы создавать и обучать отдельную модель под каждую задачу, можно взять одну универсальную LLM и быстро адаптировать её под разные цели, не держа десятки копий полной модели. Достаточно хранить небольшие «дельта-модули» – например, набор адаптеров или матриц LoRA – для каждой задачи, что облегчает развёртывание в продуктивной среде. Стоит также отметить, что параметрически эффективное дообучение обычно лучше сохраняет исходные возможности модели: замороженная основа продолжает обеспечивать общее «понимание» языка, а добавленные параметры лишь слегка корректируют поведение модели. Это помогает избежать деградации качества на тех аспектах, которые не связаны с новой задачей – модель не утрачивает ранее приобретённые знания и навыки. В совокупности, эволюция методов fine-tuning от полного обновления весов ко всё более экономичным техникам позволила исследователям и инженерам эффективно использовать большие модели даже в условиях ограниченных вычислительных ресурсов. Следующим шагом в этом направлении стало появление подходов, комбинирующих описанные идеи с

сжатием моделей – например, метод QLoRA, объединяющий низкоранговую адаптацию с 4-битным квантованием для ещё большей эффективности.

2.3.3 LoRA – Low-Rank Adaptation

Модели вроде GPT и BERT содержат десятки миллиардов параметров. Их полное дообучение требует значительных вычислительных мощностей и объёма памяти. В исследовательской или прикладной среде это часто делает использование таких моделей затруднительным. Особенно если задача предполагает адаптацию под конкретный формат данных или новую предметную область

В 2021 году появился метод LoRA – Low-Rank Adaptation. Его цель – сократить число параметров, которые нужно дообучать. Вместо обновления исходных весов, к ним добавляется специальная корректировка. Эта корректировка представляет собой произведение двух маленьких матриц (рисунок 6). Их размер выбирается заранее. Основные параметры остаются неизменными. Обучаются только эти добавочные матрицы.

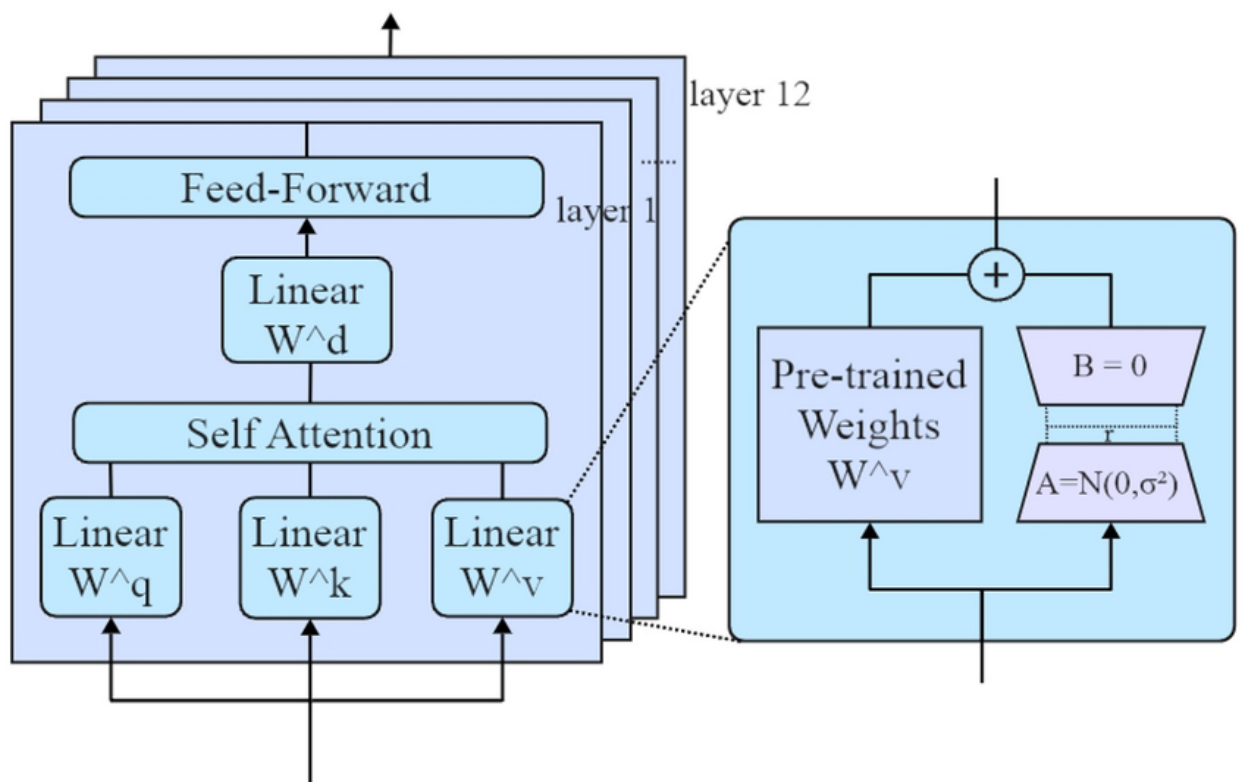


Рисунок 6 – Архитектура моделей с применением LoRA

Преимущество такого подхода в том, что он снижает нагрузку на память и ускоряет обучение. Это особенно полезно, если выборка небольшая, а модель всё ещё должна работать с текстами разного типа. LoRA чаще всего применяют в задачах генерации текста, классификации, распознавания намерений и подобных. Его удобно использовать, если цель – быстро адаптировать поведение модели к новым данным без глубокого вмешательства в её структуру.

Метод хорошо сочетается с существующими трансформерными архитектурами. Он не требует значительной переработки модели и легко интегрируется в готовую систему. В большинстве случаев LoRA даёт достаточно хорошее качество, особенно если задача не требует полной перестройки модели или радикального изменения её поведения.

Чтобы использовать LoRA, в модель ничего кардинально не добавляют. Её структура остаётся прежней. Основная идея в том, что вместо обновления весов в линейном слое (обычно это проекция в attention или FFN-блоке), к этим весам прибавляется небольшая поправка. Она рассчитывается отдельно и не затрагивает основную матрицу.

На практике берётся стандартный линейный слой – матрица весов W . К ней добавляют две новые матрицы: A и B . Матрица A сжимает вход до низкого ранга, матрица B восстанавливает размерность обратно. Получается что-то вроде:

$$W' = W + BA$$

Важно, что W остаётся замороженной, то есть её значения не меняются. Обновляются только A и B . Их размер обычно на порядок меньше, чем у W . Поэтому обучение требует меньше памяти и проходит быстрее.

В трансформерах такие вставки делают в тех слоях, которые больше всего влияют на поведение модели. Чаще всего это attention-блоки и слои в feed-forward сети. Иногда LoRA применяют только к части модели – например, к последним слоям или только к self-attention. Это зависит от задачи и объёма ресурсов [12].

Встраивание происходит на этапе подготовки модели. Слой заменяют на версию с LoRA. Это делается через небольшую обёртку в коде. Если

используется библиотека вроде PEFT (из HuggingFace), всё добавляется через конфигурацию, без ручного переписывания слоёв.

Такой способ интеграции делает LoRA удобным выбором, особенно когда модель уже обучена, и нужно быстро адаптировать её под новый набор данных или формат выходного текста.

Параметры LoRA задаются заранее, до начала обучения. Обычно их не так много, но от них зависит, насколько эффективно модель будет дообучаться. Основной параметр – это ранг, он обозначает размерность промежуточного пространства между матрицами A и B.

2.3.4 Обучение модели с использованием LoRA

Обучение модели (рисунок 7) с LoRA почти не отличается от обычного процесса finetuning. Основная разница в том, какие параметры участвуют в обновлении. При старте обучения все веса основной модели остаются замороженными. Обновляются только те параметры, которые добавлены в LoRA-слоях. Это позволяет сильно уменьшить объём градиентов, и, как следствие, снизить требования к GPU и памяти.

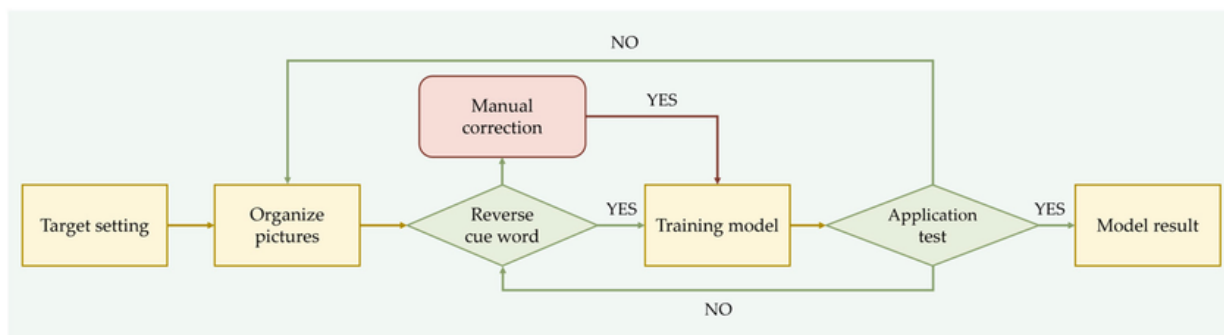


Рисунок 7 – Процесс обучения модели

Перед запуском обучения формируется датасет, соответствующий задаче. Это может быть парный набор «вход – ответ» для генерации, или «текст – метка» для классификации. Никаких специальных требований к структуре данных нет. LoRA работает поверх уже готовой модели, поэтому формат обучающих примеров должен соответствовать изначальной архитектуре.

Далее происходит стандартная токенизация текста. Обычно используется та же токенизаторная схема, что и при обучении исходной модели. Токены

подаются в модель, на выходе получается либо метка (в случае классификации), либо предсказанная последовательность (в случае генерации). Потери рассчитываются стандартно – через кросс-энтропию или другую функцию, в зависимости от задачи.

Важно, что в процессе обратного распространения ошибки (backpropagation) обновляются только параметры, отмеченные как trainable. В случае с LoRA это только матрицы A и B. Все остальные параметры не получают градиенты. Это сильно снижает нагрузку и позволяет запускать обучение даже на одной видеокарте средней мощности.

Периодически модель сохраняется. Обычно сохраняются только LoRA-параметры, а не полная копия всей модели. Это экономит место. При инференсе или валидации эти параметры подгружаются поверх исходной модели. Если использовать библиотеку вроде PEFT, это делается одной командой.

Процесс обучения можно контролировать с помощью стандартных метрик. В классификации это точность, F1-мера и т. п. В генерации – BLEU, ROUGE или лог-перепроверка качества предсказаний вручную. Переобучение встречается редко, но его можно отслеживать через валидационную выборку и регуляризацию.

Весь процесс обычно занимает меньше времени, чем полный finetuning. Даже при большом объёме данных обучение проходит быстрее, поскольку обновляется небольшая часть модели. Это особенно важно, если модель большая, а оборудование ограничено.

Если обозначить размер входного тензора как d , а размер выхода как k , то без LoRA линейный слой содержит матрицу размером $k \times d$. При использовании LoRA добавляются две матрицы: A ($r \times d$) и B ($k \times r$), где r – это ранг. Чем он меньше, тем меньше параметров обучается, но и степень адаптации модели ограничена. Если выбрать слишком маленькое значение, LoRA не сможет внести нужные изменения. Если слишком большое – вырастут требования к памяти, и эффект экономии снизится. На практике часто

используют значения от 4 до 64. Оптимальный ранг подбирается эмпирически, иногда – через перебор.

Второй параметр – это α . Он отвечает за масштаб влияния LoRA-компоненты. После расчёта BA результат умножается на α / r , чтобы нормализовать добавку. Это сделано для того, чтобы не перегрузить основную матрицу весов. α можно воспринимать как коэффициент уверенности в том, что LoRA должна вносить заметные изменения. Если он слишком мал, LoRA-компонента почти не влияет. Если велик – может возникнуть переобучение. Стандартные значения – 16, 32 или 64.

Также указывается, к каким слоям применять LoRA. Например, можно указать только q и v проекции в attention-механизме. Это уменьшает нагрузку и даёт хорошую адаптацию, особенно если модель большая. Иногда LoRA подключают и к feedforward-слоям, но это делается реже.

Наконец, есть параметр dropout, который добавляет регуляризацию. Он применяется к LoRA-компоненте во время обучения, чтобы уменьшить переобучение. Как и в других моделях, dropout отключается при инференсе.

Все параметры задаются через конфигурацию. Если используется библиотека вроде PEFT, их можно передать через словарь или файл. В случае кастомной реализации – задаются вручную перед добавлением слоёв.

После завершения обучения модель с LoRA-компонентами можно использовать для инференса. При этом сама структура модели остаётся прежней. Загружается исходная предобученная модель, к ней подключаются сохранённые LoRA-параметры. Это позволяет быстро восстановить адаптированное поведение, не загружая полную копию модели с изменёнными весами.

Если используется библиотека вроде PEFT (Parameter-Efficient Fine-Tuning) из экосистемы HuggingFace, процесс загрузки выглядит просто. Сначала загружается базовая модель, такая же, как использовалась при обучении. Затем через специальную функцию подключаются LoRA-настройки. Они могут храниться как в отдельной папке, так и в виде архива.

Процесс дообучения изображен на листинге 3.

Листинг 3 – Пример описания шаблона

```
from peft import PeftModel, PeftConfig
from transformers import AutoModelForCausalLM

base_model =
AutoModelForCausalLM.from_pretrained("имя_базовой_модели")
peft_model = PeftModel.from_pretrained(base_model,
"путь_к_LoRA_параметрам")
```

После этого модель можно использовать как обычную: подавать текст, получать ответы, строить генерации или классифицировать. Все LoRA-изменения применяются на лету. Если не подключать их, модель будет вести себя как базовая.

Вес LoRA-параметров составляет десятки мегабайт, в отличие от нескольких гигабайт полной модели. Поэтому такую настройку можно быстро загружать и использовать в разных задачах. Например, можно держать одну базовую модель и несколько LoRA-конфигураций для разных предметных областей – и переключаться между ними без перезапуска всей системы.

Также есть возможность «слияния» параметров. Это процедура, при которой LoRA-компоненты складываются с основными весами и сохраняются как единая модель. Тогда для использования не требуется отдельная загрузка LoRA-слоёв [12].

2.3.5 Метод QLoRA

Метод QLoRA (Quantized Low-Rank Adaptation) был предложен как дальнейшее развитие идей PEFT, позволяющее ещё сильнее снизить затраты памяти при дообучении больших моделей. Если LoRA решает проблему избыточного числа обучаемых параметров, то QLoRA нацелена на уменьшение самого объёма модели в памяти во время обучения. Главная мотивация заключалась в том, чтобы сделать возможным дообучение моделей с десятками миллиардов параметров на оборудовании, недоступном для классического обучения (например, на одном GPU с ограниченной памятью). В традиционном подходе даже при использовании LoRA необходимо загрузить все веса базовой LLM в память с высокой разрядностью (обычно в формате FP16, то есть 16-бит

с плавающей запятой). Для модели объёмом, скажем, 65 млрд параметров это означает сотни гигабайт памяти, что выходит за рамки возможностей одиночного ускорителя. QLoRA предлагает решить эту проблему путём агрессивного сжатия весов: модель загружается в квантованном виде с пониженной точностью, обычно 4 бита на параметр. Это сразу уменьшает память, требуемую для хранения модели, примерно в 4 раза (по сравнению с 16-битным представлением), делая возможным работу с системами ранее запредельного размера.

Ключевая особенность QLoRA – использование специальной схемы квантования под названием NF4 (NormalFloat 4-bit), которая позволяет сохранить максимальную точность при столь сильном сжатии. В отличие от наивного равномерного квантования, которое просто масштабирует диапазон значений на 4-битную сетку, формат NF4 распределяет уровни квантования с учётом реального статистического распределения весов. Проще говоря, в NF4 маленькие по абсолютной величине веса (а значительная часть параметров трансформеров близка к нулю) кодируются с более высокой относительной точностью, тогда как крайне большие или редкие значения могут квантоваться более грубо. Такая неравномерная квантизация позволяет минимизировать погрешность: авторы QLoRA показали, что модель в формате 4 бита (NF4) практически не уступает по качеству той же модели, представленной в обычном FP16. Более того, чтобы снизить накладные расходы на хранение квантованных коэффициентов, применяется техника “двойного квантования” – она означает, что сами параметры квантования (например, масштабы квантования) тоже хранятся в сжатом виде вместо 16-битного, что дополнительно экономит память. В совокупности эти меры обеспечивают высокую точность даже при радикальном сокращении разрядности [13].

При дообучении с QLoRA сама базовая модель остаётся замороженной и представленной в 4-битном формате, а обучаемые параметры – низкоранговые матрицы адаптации (LoRA) – добавляются аналогично обычному LoRA-методу. Обучение происходит таким образом, что градиенты вычисляются только для

этих добавленных матриц, а веса базовой модели не меняются. В реализации QLoRA используются специальные оптимизированные библиотеки (например, пакет BitsAndBytes), поддерживающие эффективные операции с 4-битными тензорами и смешанные вычисления. Обычно во время обратного распространения ошибки часть операций выполняется в более высокой точности (например, градиенты аккумулируются в FP16), чтобы не потерять чувствительность из-за квантования. Тем не менее на прямом проходе и при хранении модели используется 4-битное представление, что и даёт основной выигрыш. Благодаря этому подходу стало возможным запускать дообучение ранее недоступных по масштабу моделей. Например, эксперименты показали, что 65-миллиардную LLaMA можно успешно дообучить на одном сервере с 48 ГБ памяти GPU, тогда как классический метод потребовал бы несколько таких GPU или более дорогие специализированные системы.

Важно подчеркнуть, чем QLoRA отличается от «классического» LoRA. В методе LoRA предполагалось, что есть возможность оперировать с полными весами модели в стандартном формате, и акцент делался только на сокращении числа параметров, которые нужно обновлять. QLoRA же комбинирует эту идею с сжатием: модель в памяти занимает в разы меньше места, поскольку почти все её веса хранятся в низкой точности. Это означает, что узким местом становится уже не объём модели, а лишь матрицы LoRA и накладные данные алгоритма обучения. Таким образом, QLoRA радикально снижает требования к аппаратуре: то, что ранее требовало распределённого обучения или облачной инфраструктуры, теперь может быть выполнено на одном относительно доступном устройстве. За это приходится платить определённой сложностью и временными затратами. Во-первых, операции с квантованными числами менее эффективно реализуются на GPU, поэтому обучение с QLoRA обычно идёт медленнее: по оценкам, на 20–30 % больше времени на эпоху по сравнению с аналогичной моделью без квантования. Во-вторых, хотя качество модели в 4-битном формате остаётся высоким, в отдельных случаях наблюдается незначительное снижение точности по сравнению с 16-битным аналогом,

особенно на очень чувствительных к численным ошибкам задачах. Тем не менее на большинстве прикладных задач QLoRA демонстрирует практически такое же качество, что и полноразмерное дообучение, и при этом обеспечивает огромную экономию памяти.

Метод QLoRA стал востребованным в ситуациях, когда требуется адаптировать большие языковые модели в условиях ограниченных ресурсов. Он вобрал в себя лучшие черты LoRA – сохранение подавляющей части весов неизменными и обучение компактных адаптивных модулей – и усилил их через использование квантования. На практике QLoRA открыла возможность многим исследователям и разработчикам работать с LLM, ранее недоступными по объёму: вместо того чтобы довольствоваться более мелкими моделями, теперь можно дообучить практически такую же мощную модель, но сжатую на время обучения. Это особенно актуально для тонкой настройки модели под узкоспециализированные домены, когда важно сохранить качество генерации, близкое к исходной крупной модели, но нет возможности развернуть полноценное обучение. QLoRA продолжает линию развития методов PEFT, показывая, что даже без изменения архитектуры модели можно чрезвычайно эффективно адаптировать её под новые задачи, оптимально используя доступные вычислительные ресурсы [13].

2.3.6 Использование PEFT и QLoRA в контексте дообучения на предметных онтологиях

Большие языковые модели можно эффективно приспособлять под специфические предметные области, используя формализованные онтологии этих областей. Предметная онтология представляет собой строго структурированное описание знаний в некоторой сфере: она включает перечень ключевых понятий (классов), их определения, а также отношения между ними (например, иерархия “класс–подкласс”, ассоциации, причинно-следственные связи и др.). Онтологии обычно задаются в виде графовых структур или в формате логических триплетов (как в стандартах RDF/OWL), что делает их машиночитаемыми. Имея такую онтологию, мы, по сути, располагаем сжатым и

проверенным человеком корпусом знаний по определённой тематике. Задача дообучения LLM на основе онтологии заключается в том, чтобы «вливать» эти структурированные знания в модель либо научить модель эффективно использовать их при генерации текста. Это особенно актуально для прикладных интеллектуальных систем: например, в медицинской сфере модель, обученная на медицинской онтологии, будет лучше оперировать терминологией и фактами о заболеваниях, в юридической области модель, адаптированная под онтологию законодательства, будет более точно понимать отношения между правовыми понятиями и выдавать консистентные ответы.

Использование PEFT-подходов, таких как LoRA и QLoRA, существенно упрощает дообучение модели на данных онтологии. Дело в том, что размер онтологического датасета обычно невелик по сравнению с объёмами текстов, на которых обучались сами LLM. Он может содержать тысячи или десятки тысяч утверждений, определений или пар «вопрос–ответ» на основе понятий онтологии – но не миллиарды примеров. Поэтому нет необходимости и смысла перенастраивать все параметры огромной модели: гораздо эффективнее ограничиться небольшим числом дополнительных весов, позволяющих аккуратно подстроить модель под требуемый домен. Замораживание основной массы весов LLM при обучении на материалах онтологии означает, что общие языковые способности и мировые знания модели сохраняются, а все изменения сконцентрируются на новых знаниях из предметной области. Это снижает риск того, что модель утратит какую-либо полезную информацию из базовой подготовки: например, адаптируя модель к медицинской онтологии, мы не хотим, чтобы она перестала понимать общие фразы на бытовые темы – и PEFT-методики помогают избежать такого эффекта. Кроме того, обучение небольшого набора параметров требует куда меньше данных, что соответствует самой природе онтологий: они компактны, и добавочные параметры способны выучиться даже на этом ограниченном материале без переобучения.

Дообучение с помощью методов вроде LoRA на онтологическом знании обычно предполагает составление специального обучающего корпуса,

извлечённого или сгенерированного на основе самой онтологии. Сама по себе онтология (как формальная структура) может быть преобразована в текстовые данные, понятные модели. Один подход заключается в генерации наборов вопросов и ответов по содержанию онтологии. Например, для каждой концепции можно сформулировать вопрос "Что такое X?" и подготовить ответ, взяв определение X из онтологии. Также можно создавать утверждения на естественном языке, отражающие заданные связи: скажем, предложения вида "X является подтипом Y согласно онтологии Z" или "В данной предметной области A связано с B отношением R". Модель, обучаясь на таких примерах, учится воспроизводить факты из онтологии и правильно использовать терминологию. В некоторых случаях целевой формат вывода модели тоже может быть структурированным: например, если необходимо, чтобы LLM извлекала из произвольного текста факты для пополнения базы знаний, её можно обучить выдавать ответы в виде JSON-объектов с полями для понятий и отношений. Применение PEFT позволяет внедрить новые знания без полного переобучения: модель «помнит» основной язык и общий мир, но приобретает способность излагать и структурировать информацию согласно предоставленной онтологии.

Метод QLoRA особенно полезен, когда для работы нам нужна очень мощная модель, а ресурсы ограничены. Часто предметная онтология охватывает узкую, но глубоко проработанную область знаний, для которой качество ответов имеет критическое значение. Использование крупной модели (десятки или сотни миллиардов параметров) было бы желательным – ведь она обладает более развитым «пониманием» и генерирует текст высокого качества, – но до недавнего времени дообучить такую модель на специфических данных могли лишь крупные организации. Благодаря QLoRA ситуация меняется: можно взять, к примеру, 70-миллиардную базовую модель, квантовать её до 4 бит и обучить с LoRA-адаптациями на данных онтологии, имея всего одну-две видеокарты. При этом итоговый адаптер (набор новых весов) получится весьма компактным, его можно распространять без включения самой модели (что удобно с точки зрения лицензирования и обмена данными). Разработчики, обладающие экспертными

онтологиями, теперь получают возможность самостоятельно создать на их основе разговорного агента или консультационную систему, не передавая конфиденциальные данные третьим лицам и не полагаясь исключительно на внешние API крупных LLM. Таким образом, сочетание PEFT и онтологического подхода снижает барьер входа: доменные знания, аккумулированные в онтологиях, легко интегрируются в модель и начинают работать в интерактивном режиме на благо пользователей.

Практические результаты применения дообучения на предметных онтологиях демонстрируют ряд преимуществ. Во-первых, модель, обученная на онтологии, более последовательна и не противоречит сама себе в рамках данного домена: она опирается на единую систему терминов и отношений, заданную онтологией. Это значит, что ответы модели в разное время или при разной формулировке запроса будут согласованными (например, определение термина не будет произвольно меняться от запроса к запросу, а останется стабильным и совпадающим с онтологическим определением). Во-вторых, такая модель меньше склонна к фактическим “галлюцинациям” относительно знаний из своей области: если чего-то нет в онтологии, она с меньшей вероятностью начнет выдумывать ответ, скорее признает отсутствие информации или даст осторожный, обтекаемый комментарий. Конечно, полностью исключить вероятность галлюцинаций сложно, но наличие строгого набора знаний дисциплинирует генерацию. В-третьих, LLM, настроенная на онтологию, способна автоматически генерировать высоко структурированные документы или ответы. Например, в образовательной сфере можно получать учебные программы или аннотации дисциплин: модель будет заполнять текстом шаблон документа, следуя структуре, заданной онтологией учебной программы. Такое сочетание формального каркаса (онтологии) и языковой гибкости (LLM) позволяет создавать контент, одновременно соответствующий нормативным требованиям и изложенный понятным человеческим языком. Подводя итог, можно сказать, что дообучение больших языковых моделей с использованием PEFT на базе предметных онтологий открывает путь к созданию

интеллектуальных систем, глубоко освоивших конкретную область. Эти системы наследуют от LLM общие языковые навыки и мощь генерации текста, дополняя их точными знаниями из формализованной базы.

3 Реализация программных средств генерации документов на основе знаний о предметной области

3.1 Архитектура приложения

Архитектура приложения разделена на фронтенд (интерфейс пользователя) и бекенд (серверная логика). В качестве фронтенда используется приложение на Streamlit, которое запускается локально и предоставляет веб-интерфейс. Бекенд реализован как веб-сервис на основе FastAPI, работающий на локальном хосте (по умолчанию на порту 8000). Фронтенд и бекенд взаимодействуют через REST API: Streamlit-приложение отправляет HTTP-запросы к FastAPI-серверу и обрабатывает ответы. Код приложения приведен в приложении А.

3.1.1 Основные компоненты и потоки данных

Ввод и извлечение онтологии: пользователь через веб-интерфейс вводит текст (описание предметной области) в соответствующее поле и нажимает кнопку “Извлечь онтологию”. Фронтенд формирует запрос к эндпоинту `/extract_ontology` бекенда, передавая введенный текст в формате JSON. Сервер принимает запрос, вызывает модуль извлечения онтологии (языковую модель), формирует структуру онтологии и возвращает её в виде JSON с списком концептов и отношений. Если запрос успешен, фронтенд отображает уведомление об успешном извлечении и сохраняет полученную онтологию во внутреннем состоянии сессии.

Визуализация и редактирование онтологии: после получения онтологии бекендом, фронтенд переходит в режим редактирования. Отображается граф онтологии: узлы соответствуют концептам (с отображением их меток), а рёбра – отношениям между ними. Для построения графической схемы используется библиотека NetworkX и отрисовка с помощью matplotlib. Пользователь может анализировать полученную структуру. Предусмотрена возможность редактирования: например, изменить названия или описания концептов, добавить или удалить связи (интерфейс для правки может включать текстовые поля или таблицы, синхронизированные с данными онтологии). После внесения

изменений пользователь нажимает кнопку “Сохранить онтологию”, и фронтенд отправляет обновлённые списки концептов и отношений на сервер через запрос к эндпоинту `/update_ontology`. Бекенд в ответ выполняет обновление внутренней модели онтологии и возвращает обновлённую структуру (или подтверждение). Таким образом, обеспечивается консистентность данных онтологии на сервере и на клиенте.

Выбор шаблона и генерация документа: Пользователь переключается на вкладку генерации документа. Интерфейс запрашивает у сервера список доступных шаблонов (эндпоинт `/templates`) и отображает, например, стандартный шаблон `sample_template.docx` в выпадающем списке. Пользователь может либо выбрать один из готовых шаблонов, либо загрузить свой файл шаблона (DOCX) через поле загрузки. После выбора шаблона пользователь нажимает кнопку “Сгенерировать документ”. Фронтенд отправляет запрос на эндпоинт `/render_doc`, передавая текущую онтологию (JSON) и файл шаблона (либо из списка, либо загруженный) в теле запроса. Бекенд принимает шаблон и сохраняет его временно, затем с помощью модуля генерации документа объединяет данные онтологии с шаблоном и формирует итоговый Word-файл. Сгенерированный документ сохраняется на сервере (в директории `backend/output`) под уникальным именем, которое возвращается в ответе (поле `document_path`). Фронтенд предоставляет пользователю ссылку для скачивания полученного файла. Пользователь может скачать сгенерированный документ и при необходимости продолжить работу (например, заново отредактировать онтологию или сгенерировать другой документ).

Для реализации серверной части используется FastAPI (включая интеграцию с Pydantic для определения схем данных запросов/ответов), что обеспечивает удобное определение REST-эндпоинтов и автоматическую генерацию документации API. На стороне фронтенда Streamlit позволяет быстро создать интерактивный веб-интерфейс без необходимости ручной разработки HTML/JS, что ускоряет прототипирование. Обмен данными происходит в

формате JSON (для онтологии) и двоичных файлов (для шаблона и сгенерированного документа).

Стоит отметить, что в архитектуре заложено хранение онтологии в структуре RDF-графа на стороне сервера. Хотя взаимодействие с фронтендом осуществляется через JSON, серверный компонент преобразует онтологию в семантический граф. Это может пригодиться для расширяемости: RDF-граф позволяет выполнять сложные запросы, проверки консистентности или интеграцию со сторонними онтологиями, если потребуется.

3.1.2 Основные модули и классы приложения

3.1.2.1 Класс `OntologyExtractor`

За извлечение онтологии из текста отвечает класс `OntologyExtractor`, определённый в модуле `backend/llm_infer.py`. Его задача – воспользоваться языковой моделью для преобразования произвольного текстового описания в структурированные данные (концепты и отношения).

В конструкторе `__init__` этого класса происходит инициализация модели LLM. В проекте используется локальная модель на основе LLaMA, загружаемая через библиотеку `llama-cpp` (Python-обёртка для C++ реализации LLaMA). В исходном коде видно, что `OntologyExtractor` загружает модель из файла `phi4-objfield-q4_K_M.gguf` с заданными параметрами контекста и потоков. Эта модель представляет собой заранее подготовленную (обученную или дообученную) небольшую языковую модель, оптимизированную для задачи выделения онтологий. Использование `llama-cpp` позволяет выполнять инференс модели локально на CPU, без внешних API-запросов, что удобно в автономной среде.

Основной метод класса – `extract_ontology(text: str) -> Dict`. Он формирует входной prompt для LLM на основе переданного текста. В prompt явно задаётся требуемый формат ответа – JSON-объект с полями “`concepts`” и “`relations`” (каждый концепт содержит идентификатор, название и описание, каждая связь описывается исходным и целевым концептом и типом отношения). В шаблоне prompt предусмотрен пример JSON-структуры и правила: например, модель должна использовать русский язык для меток и описаний и завершать вывод

специальным маркером “</json>”. Такое описание гарантирует, что модель сгенерирует ответ именно в ожидаемом формате. После подготовки prompt передаётся модели для генерации. Параметры генерации подобраны консервативно: temperature низкая (0.0), что нацелено на детерминированный выход, и указано стоп-слово “</json>” для остановки генерации после окончания JSON-объекта.

Получив от модели строку с JSON, `OntologyExtractor` осуществляет разбор (парсинг) JSON в словарь Python (используется `json.loads`). Далее, для надежности, результат может быть проверен на соответствие ожидаемой структуре (например, наличие требуемых полей). Полученный словарь онтологии возвращается вызывающему коду (в данном случае – обработчику эндпоинта FastAPI). Таким образом, `OntologyExtractor` инкапсулирует работу с LLM и преобразование неструктурированного текста в первичную онтологию.

3.1.2.2 Класс `OntologyManager`

Класс `OntologyManager` (модуль `backend/ontology.py`) отвечает за внутреннее представление онтологии на стороне сервера и операции по её обновлению и проверке. Он использует библиотеку `RDFlib` для работы с RDF-графами, тем самым представляя онтологию как набор семантических триплетов (предмет–связка–объект). При инициализации (`__init__`) создаётся пустой граф и задаётся пространство имён (`Namespace`) для онтологии, например “`http://example.org/ontology/`”, привязанное с префиксом `ex`. Это пространство имён применяется для формирования URI ресурсов в графе (например, `ex:C1` для концепта с идентификатором `C1`).

В `OntologyManager` определены вспомогательные методы для добавления в граф концептов и отношений на основе словарного представления.

`_concept_to_rdf(concept: Dict)` добавляет в RDF-граф новый узел-концепт. Каждый концепт из JSON-структуры имеет поля `id`, `label` (название) и `description` (описание). Метод генерирует URI для концепта на основе его `id` и пространства имён (например, `id="C1"` станет URI `http://example.org/ontology/C1`). Затем в граф добавляются тройки: что данный URI является экземпляром класса `ex:Concept`

(используется предикат `rdf:type`), а также задаются его название и описание (предикаты `rdfs:label` и кастомный `ex:description` соответственно). В результате каждое понятие представлено в графе с семантикой: тип “Concept”, человекочитаемое имя и текстовое описание.

`_relation_to_rdf(relation: Dict)` добавляет в граф отношение между двумя концептами. В структуре JSON отношения задаются через поля “from” (идентификатор исходного концепта), “to” (идентификатор связанного концепта) и “type” (тип связи). Метод берёт соответствующие URI для исходного и целевого концептов (через пространство имён) и также генерирует URI для типа отношения. После этого в граф добавляется одна RDF-тройка: субъект – URI исходного концепта, предикат – URI отношения, объект – URI целевого концепта. Таким образом, связи хранятся как ориентированные ребра с помеченным типом (например, если “type”: “subclass”, то в графе будет тройка `ex:C1 ex:subclass ex:C2`).

`update_ontology(concepts: List[Dict], relations: List[Dict]) -> Dict` принимает на вход обновлённый список концептов и отношений (например, отредактированных пользователем), и выполняет пересоздание внутреннего RDF-графа. В начале работы метод очищает предыдущий граф (инициализирует новый Graph и заново привязывает префикс `ex`). Затем последовательно вызывает `_concept_to_rdf` для каждого концепта и `_relation_to_rdf` для каждого отношения из переданных списков. Таким образом, граф пересобирается с нуля на основе актуальных данных. После обновления графа метод возвращает обновленную онтологию в виде словаря (JSON-структуры) через вызов `self.to_json()`. Метод `to_json()` осуществляет обратное преобразование: обход RDF-графа и формирование списка словарей концептов и отношений (по сути, формат, ожидаемый фронтендом). Помимо этого, в `OntologyManager` предусмотрен метод `validate_ontology()`, который проверяет корректность структуры графа – например, наличие необходимых свойств у каждого концепта, соответствие идентификаторов в отношениях существующим узлам и т. п. (точные проверки закодированы внутри функции). В случае сложных изменений онтологии

(например, удаление концепта, к которому были отношения) такой валидатор позволил бы своевременно обнаружить проблему.

Использование RDF-графа внутри `OntologyManager` делает систему потенциально расширяемой: при необходимости можно добавить семантические отношения, привязать онтологию к внешним словарям или выполнять SPARQL-запросы для сложного анализа. В текущей реализации эти возможности не используются в полной мере, однако сама структура классов намекает на задел для работы с онтологиями на семантическом уровне.

3.1.2.3 Класс `DocxRenderer`

Модуль `backend/docx_renderer.py` содержит класс `DocxRenderer`, отвечающий за формирование выходного документа (DOCX) на основе шаблона и данных онтологии. Этот компонент позволяет отделить логику подготовки содержимого от собственно интерфейса и онтологии. Он использует библиотеку `docxtpl` (`DocxTemplate`), которая предоставляет механизм шаблонизации для файлов Word (DOCX). Внутри DOCX-файла-шаблона можно разместить плейсхолдеры (метки в двойных фигурных скобках, а также конструкции Jinja2 для циклов и условий), которые затем автоматически заменяются на реальные данные.

При создании экземпляра `DocxRenderer` в конструкторе задаётся папка для выходных документов (по умолчанию `output` внутри директории `backend`). Если этой папки нет, она создаётся автоматически.

`_prepare_template_data(ontology: Dict) -> Dict`: этот вспомогательный метод преобразует онтологию (в виде словаря с списками концептов и связей) в структуру данных, удобную для подстановки в шаблон. В реализации он формирует новый список `concepts_list`, проходя по каждому концепту из `ontology["concepts"]`. Для каждого концепта собирается словарь `concept_data` с ключами: `"id"` (идентификатор), `"name"` (метка/название концепта), `"description"` (описание) и `"related_concepts"` (список связанных концептов). Поле `"related_concepts"` заполняется путём поиска всех отношений, где данный концепт является исходным (`relation["from"] == concept["id"]`), для каждого

такого отношения находится целевой концепт и добавляется его название и тип связи. В результате “related_concepts” представляет собой список словарей, каждый из которых содержит “name” связанного понятия и “relation” – тип связи (например, “Подтема” или ”пример” – зависит от содержимого онтологии). Если у концепта нет исходящих отношений, список остаётся пустым. Каждый сформированный concept_data добавляется в concepts_list. После обработки всех концептов метод возвращает словарь с ключевыми данными для шаблонизации: “concepts”: concepts_list, “generated_date” <текущая дата/время>, “total_concepts”: <число концептов>, “total_relations”: <число отношений>. Здесь generated_date генерируется автоматически (в формате строки) – время создания документа, а total_concepts и total_relations позволяют отобразить в документе статистику по онтологии. Структура возвращаемого словаря соответствует переменным, ожидаемым в шаблоне документа (как будет рассмотрено ниже).

render_document(template_path: str, ontology: Dict) -> str: основной метод генерации. Он принимает путь к DOCX-шаблону и онтологию (словарь). Метод загружает шаблон, затем вызывает описанный выше _prepare_template_data, получая template_data – словарь с данными для подстановки. Далее метод запускает рендеринг шаблона: doc.render(template_data). Библиотека docxtpl проходит по документу и заменяет все placeholders соответствующими значениями из template_data. В том числе обрабатываются циклы: в шаблоне предусмотрен блок {% for concept in concepts %} ... {% endfor %}, поэтому docxtpl повторит вложенные элементы (название, описание концепта и т. д.) для каждого концепта из списка. Аналогично обрабатываются условные конструкции, например вывод списка связанных концептов, только если он не пустой (в шаблоне используется {% if concept.related_concepts %} ... {% endif %}). После успешной замены данных DocxRenderer генерирует имя выходного файла – обычно включающее временную метку, например “document_20250615_131724.docx”. Выходной путь формируется внутри директории output, и документ сохраняется. Метод возвращает строку с путем сохранённого файла. Если во время обработки произошла ошибка (например,

отсутствует требуемая переменная в шаблоне), выбрасывается исключение, которое обрабатывается на уровне эндпоинта FastAPI и переводится в HTTP ошибку 500.

Дополнительно в DocxRenderer имеется метод `get_template_variables(template_path: str)` – он загружает шаблон и возвращает список всех переменных, используемых в нём (используется метод `doc.get_undeclared_template_variables()` из `docxtpl`). Это может пригодиться для отладки или для отображения пользователю списка доступных плейсхолдеров. В текущем фронтенде данная функция напрямую не вызывается, но её наличие облегчает расширение функционала (например, автоматически сопоставлять поля онтологии с полями шаблона).

Вместе с DocxRenderer в проекте присутствует файл `sample_template.docx` – это шаблон документа по умолчанию. Судя по исходникам, этот шаблон содержит заранее заготовленную структуру: заголовки для документации, места для списка концептов, их описаний и связанных понятий. Шаблон был автоматически сгенерирован с помощью вспомогательного скрипта (`create_template.py` или `init_templates.py`), который программно создает DOCX с нужными плейсхолдерами (например, вставляет текст `{{generated_date}}`, `{{total_concepts}}` и блоки Jinja2 для перечня концептов и их связей). Такой подход позволяет пользователю иметь сразу готовый пример оформления, который можно изменить под свои нужды. Также пользователь может загрузить свой шаблон – главное, чтобы в нём были используются понятные `docxtpl`-плейсхолдеры, соответствующие структуре данных (например, переменные `concepts`, `concept.name` и т. д., как в стандартном шаблоне).

3.1.2.4 Прочие компоненты

Помимо основных классов, в проекте есть несколько вспомогательных модулей и компонентов.

FastAPI эндпоинты и схемы. В файле `backend/main.py` на стороне сервера определены маршруты API. В частности:

- POST /extract_ontology – вызывает OntologyExtractor и возвращает JSON онтологии,
- POST /update_ontology – принимает отредактированную онтологию, обновляет через OntologyManager и возвращает актуализированную версию,
- POST /render_doc – принимает файл шаблона и онтологию, генерирует документ через DocxRenderer и возвращает путь к файлу,
- GET /templates – возвращает список доступных шаблонов (файлов .docx) в директории шаблонов.

Для валидации входных данных используются Pydantic-модели: класс `TextInput(BaseModel)` описывает JSON со строковым полем `text` (для запроса извлечения онтологии), а класс `OntologyUpdate(BaseModel)` описывает ожидаемые структуры списка концептов и связей для обновления. Эти классы обеспечивают проверку типов и облегчают документацию API (схемы автоматически попадают в интерактивную документацию Swagger UI).

Streamlit интерфейс. Во фронтенде (файл `webui/app.py`) код написан процедурно, без дополнительных классов, но важно упомянуть его логику. Приложение разделено на две вкладки: «Редактор Онтологии» и «Генерация Документа». Используются сессии Streamlit (`st.session_state`) для хранения онтологии между шагами. При нажатии кнопок вызываются соответствующие REST-запросы через библиотеку `requests`. Граф онтологии визуализируется с помощью `NetworkX`: создается граф, добавляются узлы (идентификатор и метки) и рёбра (с типами отношений в качестве меток). Для раскладки графа применяется алгоритм силы (`nx.spring_layout`), узлы и связи рисуются средствами `matplotlib`, и граф выводится в Streamlit через `st.pyplot`. Интерфейс также содержит формы для редактирования: текстовое поле для ввода исходного текста, выпадающий список для выбора шаблона, элемент загрузки файла для пользовательского шаблона, кнопки запуска операций. Благодаря `st.spinner` и сообщениям `st.success/st.error` обеспечивается обратная связь в процессе работы (например, отображается индикатор во время генерации онтологии или документа).

Обработка шаблонов. Уже упомянутые скрипты `create_template.py` и `init_templates.py` служат для программного создания шаблона документа. Они используют библиотеку `python-docx` для добавления в документ необходимых разделов и плейсхолдеров. Этот код, по сути, генерирует файл `sample_template.docx` с демонстрационной структурой, содержащей поля: дата генерации, количество концептов, количество связей, а также перечисление всех концептов (с названиями, описаниями и списком связанных концептов для каждого). Наличие таких скриптов позволяет автоматически воссоздать шаблон, если он отсутствует, и потенциально упрощает обновление структуры шаблона в будущем.

3.1.3 Взаимодействие модулей и классов

В процессе работы приложения упомянутые классы тесно взаимодействуют, каждый выполняя свою роль в общей цепочке обработки данных. Ниже описан сценарий полного цикла работы, иллюстрирующий взаимодействие компонентов.

Запуск и ввод данных: Пользовательский интерфейс Streamlit запускается и подключается к локальному API FastAPI. Пользователь вводит текстовое описание предметной области (например, небольшой обзор темы) и инициирует процесс извлечения онтологии.

Извлечение онтологии (Backend): запрос `/extract_ontology` поступает на сервер. FastAPI вызывает функцию, которая создаёт (или использует существующий) экземпляр `OntologyExtractor` и передаёт ей введённый текст. `OntologyExtractor` формирует prompt и вызывает LLM для генерации онтологии. Получив JSON-структуру, она возвращается как результат эндпоинта.

Отображение онтологии (Frontend): ответ сервера содержит онтологию в JSON (список концептов и связей). Streamlit-приложение сохраняет его (например, в `st.session_state.ontology`) и визуализирует: строится граф отношений, выводится список концептов. Пользователь видит извлечённые ключевые понятия (с краткими описаниями) и связи между ними. Предположим, для учебного текста об определённой теме были выделены концепты “Основное

понятие”, “Подтема1”, “Подтема2” и отношения типа “подчиняется” (subtopic) или ассоциации примеров.

Редактирование онтологии: если пользователь замечает неточности или хочет уточнить структуру, он может отредактировать онтологию. Например, переименовать какой-то концепт, удалить лишний либо добавить недостающий. В UI это может быть реализовано через интерактивные виджеты (например, таблицы или текстовые поля для каждого концепта). После внесения правок пользователь нажимает “Сохранить онтологию”. Приложение отправляет обновлённые данные на эндпоинт `/update_ontology`.

Обновление онтологии (Backend): Серверный обработчик принимает изменённые списки концептов и связей. Он вызывает метод `OntologyManager.update_ontology`, передавая эти данные. Внутри `update_ontology` происходит пересоздание RDF-графа: удаляется старая структура, заново добавляются все концепты (с их типами, названиями, описаниями) и все отношения. Таким образом, сервер получает консистентное представление онтологии с учётом правок. Затем граф снова преобразуется в JSON и возвращается в ответе.

Выбор шаблона: на втором этапе пользователь выбирает шаблон оформления документа. По запросу фронтенда `/templates` сервер возвращает список файлов шаблонов в директории (например, `["sample_template.docx"]`). Пользователь может выбрать этот шаблон или загрузить свой. Если загружается новый файл, фронтенд отправляет его напрямую при запросе генерации.

Формирование документа: при нажатии кнопки генерации документ отправляется запрос `/render_doc`. Сервер получает файл шаблона (в виде `UploadFile`) и текущую версию онтологии (например, тот же JSON, возможно скорректированный на предыдущем шаге). Эндпоинт сохраняет шаблон во временный файл и вызывает `DocxRenderer.render_document`, передав путь шаблона и онтологию. `DocxRenderer` готовит данные для шаблона: формирует для каждого концепта структуру со списком связанных концептов, а также вычисляет сводную информацию (дата, количество элементов). Затем

происходит заполнение шаблона: библиотека `docxtempl` производит подстановку данных в DOCX (в каждый плейсхолдер и цикл). Результат сохраняется на диск (в `backend/output/document_*.docx`). Путь к файлу возвращается клиенту.

Завершение – получение документа: Фронтенд Streamlit получает путь к сгенерированному документу. Для удобства пользователя интерфейс формирует ссылку для скачивания. Пользователь кликает по ссылке и загружает готовый DOCX-документ. В документе, согласно шаблону, присутствует раздел с общей информацией (дата генерации, количество понятий и связей), далее перечислены все концепты: каждый – с названием (в виде заголовка), указанием идентификатора, описанием, а также списком “Связанные концепты” (если для данного понятия были указаны связанные). Таким образом, документ автоматически содержит структурированное изложение предметной области, полученное из исходного текста.

На этом цикл работы может повторяться: пользователь при желании меняет исходный текст или правит онтологию и генерирует новый документ, пробуя разные шаблоны оформления.

Взаимодействие между классами происходит преимущественно на стороне бекенда под управлением FastAPI: `OntologyExtractor` занимается только разовым созданием онтологии из текста, далее `OntologyManager` поддерживает целостность данных при обновлениях, а `DocxRenderer` выполняет финальную стадию – сборку документа. Эти компоненты связаны логически (выход одного является входом для другого), но разделены физически по разным этапам обработки. Благодаря этому код приложения структурирован модульно, что облегчает понимание и возможную замену частей (например, можно заменить модель LLM на другую, не затрагивая модуль генерации документа).

3.1.4 Используемые технологии и библиотеки

Проект построен с применением ряда современных фреймворков и библиотек, каждая из которых отвечает за свою часть функциональности.

FastAPI – высокопроизводительный Python-фреймворк для создания RESTful API. Здесь он используется для разработки серверной части: объявления

эндпоинтов (`/extract_ontology`, `/update_ontology`, `/render_doc` и др.), обработки HTTP-запросов и возвращения JSON-ответов. FastAPI обеспечивает валидацию входных данных через Pydantic-модели (например, `TextInput`, `OntologyUpdate`) и автоматически настраивает CORS (Cross-Origin Resource Sharing) для взаимодействия со Streamlit-фронтом. В продакшн-режиме приложение запускается с помощью ASGI-сервера Uvicorn.

Streamlit – фреймворк для создания веб-интерфейсов для данных и ML-приложений на Python. В данном проекте Streamlit выбран для реализации простого и интерактивного пользовательского интерфейса. Он позволяет быстро собрать веб-страницу с формами ввода, кнопками и графиками, написав минимум кода на Python. Благодаря этому разработчик мог сосредоточиться на логике приложения, а не на тонкостях веб-разработки.

LLM и библиотека `llama-cpp` – в качестве движка для выполнения инференса большой языковой модели выбрана реализация LLaMA на C++ (через Python-пакет `llama_cpp`). Это позволяет загружать локальный файл модели (`phi4-objfield.gguf`) и выполнять генерацию текста на CPU. Подходящая модель была заранее обучена для задачи выделения онтологий, используя инструменты Hugging Face Transformers и PEFT (Parameter-Efficient Fine-Tuning) – в кодовой базе присутствуют упоминания этих библиотек (скрипт `merge.py` загружает модель `unsloth/Phi-4-mini-instruct` и применяет LoRA веса). Однако на этапе исполнения приложения участие HuggingFace не требуется – используется непосредственно результат обучения, загруженный через `llama-cpp`. Выбор локальной LLM обеспечивает автономность решения (нет зависимости от внешних API) и потенциально быструю генерацию после первоначальной загрузки модели.

RDFlib – популярная библиотека на Python для представления графов знаний в формате RDF. В проекте она отвечает за хранение и манипулирование онтологией: позволяет определить namespace, добавлять триплеты (через `Graph.add`) и выполнять сериализацию/десериализацию графа. RDFlib предоставляет готовые RDF-термины, такие как `RDF.type` и `RDFS.label`, которые

используются при формировании триплетов. Несмотря на то, что возможности RDF (запросы SPARQL, логический вывод) в проекте не используются, применение этой библиотеки закладывает фундамент для более сложной работы с онтологиями.

DocxTemplate (docxtpl) – библиотека, расширяющая python-docx для поддержки шаблонов. Она читает DOCX-файл как шаблон, ищет внутри конструкции вида `{{variable}}` и Jinja2-выражения, позволяя программисту передать словарь данных для подстановки. В проекте docxtpl используется для автоматической генерации документов: разработчик определил в шаблоне места, куда нужно подставить названия, описания и связи, а DocxRenderer передаёт эти значения через метод `doc.render()`. Таким образом, формирование итогового документа сводится к заполнению готового макета.

python-docx – нижележащая библиотека для работы с Word-документами. Она используется опосредованно (внутри docxtpl) и непосредственно в скриптах генерации шаблона. С её помощью в коде `create_template.py` создаётся новый документ, добавляются параграфы и заголовки, содержащие шаблонные метки (например, `{{total_concepts}}`). Python-docx обеспечивает доступ к структуре DOCX на уровне абзацев, таблиц, стилей, что позволило программно сформировать шаблон без использования MS Word вручную.

NetworkX и Matplotlib – связка библиотек для визуализации графов. NetworkX используется для хранения структуры онтологии (как графа узлов и рёбер) на стороне фронтенда исключительно для визуализации. Он не связан напрямую с онтологией на сервере (там используется RDFlib), но фронтенд, получив списки концептов и отношений, строит из них граф `nx.Graph/DiGraph`, чтобы удобно рассчитать координаты узлов для отображения. Matplotlib применяется для рисования этого графа – метод `nx.draw_networkx` позволяет указать позиционирование, цвета, подписи узлов и рёбер. Streamlit поддерживает вывод графиков Matplotlib, поэтому полученное изображение графа выводится пользователю. Эти библиотеки помогают наглядно представить сложную

структуру знаний в понятной форме (схемой), что ценно в интерактивной работе с онтологиями.

Прочие: В проекте используются и другие вспомогательные библиотеки. Pydantic (часть FastAPI) – для декларативного описания моделей данных запросов (валидирует типы полей JSON автоматически). Requests – на стороне фронтенда для отправки HTTP-запросов к API. urllib3, Retry – в генераторе датасета для надёжной загрузки данных из внешнего API (с повторными попытками при ошибках соединения). Logging – для логирования процессов (например, вывод prompt и длины текста перед отправкой в LLM). Эти инструменты обеспечивают устойчивость и информативность работы приложения, хотя и не являются центральными с точки зрения архитектуры.

3.2 Описание процесса дообучения

3.2.1 Данные

Для обучения использован собственный датасет в формате JSONL, подготовленный для этапа Supervised Fine-Tuning (SFT). Датасет представляет собой набор пар запрос–ответ на русском языке, где пользователь задаёт вопрос, а ассистент (модель) отвечает, как эксперт. Ключевые характеристики данных:

- направленность: вопросы охватывают различные предметные области, ассистент должен давать экспертные, содержательные ответы по заданным темам,

- объём: около 250 примеров (пар вопрос–ответ) в обучающей выборке,

- структура: файл jsonl, каждая строка которого содержит JSON-объект с полями “input” (текст запроса пользователя) и “output” (текст ответа ассистента) [15],

- подготовка данных: для каждой пары формировался диалог со специальными токенами ролей, в начале добавлялось системное сообщение (<|system|>), задающее тон: “Ты эксперт по предметным областям.”, далее следовал пользовательский запрос (<|user|> с содержимым из »input») и ожидаемый ответ ассистента (<|assistant|> с содержимым из »output»), полученная объединённая строка затем токенизировалась токенизатором модели

с ограничением длины 2048 токенов, последовательности дополнялись до этой максимальной длины, а метки для обучения устанавливались равными токенизированному ответу (то есть модель обучалась предсказывать ответ по заданному контексту), таким образом, задача обучения сводилась к условному генеративному моделированию: модель училась продолжать диалог ответом ассистента на пользовательский запрос [16].

3.2.2 Этапы обучения

Обучение производилось на базе предварительно обученной языковой модели Phi-4-mini с использованием метода QLoRA (Low-Rank Adaptation с квантованием) для эффективной донастройки модели. Были выполнены следующие шаги:

Загрузка модели: в качестве исходной модели взята unsloth/Phi-4-mini-instruct – компактная декодерная трансформер-модель (~2,23 млрд параметров), способная обрабатывать последовательности до 2048 токенов. Модель была загружена через библиотеку Unsloth в 4-битном квантованном виде (4-bit weight quantization) для экономии памяти. Архитектурно Phi-4-mini представляет собой 32-слойный Transformer-декодер с размерностью скрытого слоя 3072, применяющий RMSNorm и ротарные позиционные эмбединги (RoPE) [17].

Адаптация с LoRA: к загруженной модели применена техника LoRA для добавления небольшого числа обучаемых параметров. Слои модели были расширены низкоранговыми адаптерами (ранг LoRA = 8, коэффициент $\alpha = 16$) без добавления dropout. При этом основные веса модели остаются замороженными, обучаются только параметры адаптеров (всего ~4,46 млн параметров, что составляет лишь 0,2% от полного размера модели). Такой подход позволяет эффективно обучать большую модель на небольшом датасете даже с ограниченными вычислительными ресурсами.

Фреймворк Unsloth: для организации обучения использовалась библиотека Unsloth, предоставляющая удобные обёртки над Hugging Face Transformers [18]. С помощью FastLanguageModel из Unsloth модель была подготовлена к обучению (включая применение 4-бит квантования и LoRA-адаптеров) и

интегрирована с Trainer из HuggingFace Transformers [19]. Unsloth также обеспечивает ускоренное обучение (например, за счёт собственных оптимизаций вычислений) и совместим с 4-битными весами [20].

Гиперпараметры обучения: обучение проводилось с следующими основными настройками:

- размер батча: 1 пример на GPU (с накоплением градиента в 8 шагов, что эквивалентно эффективному batch size = 8,
- число эпох: 3 полных прохода по обучающей выборке,
- темп обучения: начальный коэффициент обучения (learning rate) $2 \times 10^{(-4)}$ с оптимизатором AdamW (в 8-битном режиме для экономии памяти),
- прочее: логи сохранялись каждые 10 шагов (logging_steps=10), модель сохранялась по завершении каждой эпохи (save_strategy="epoch"), обучение проводилось в смешанной точности: веса модели – 4-bit, градиенты – в более высокой точности (bf16/FP32), градиентный чекпоинтинг и компиляция модели (torch_compile) были отключены, поскольку на таком небольшом батче и модели они не требовались.

4 Результаты реализации

4.1 Результаты дообучения языковой модели

Динамика функции потерь. В ходе обучения функция потерь заметно уменьшилась, свидетельствуя об успешном усвоении моделью тренировочных примеров. По мере прохождения эпох потери снизились с примерно ~0.8 (в начале первой эпохи) до порядка ~0.4 к концу третьей эпохи (примерные оценки)

(рисунок 7). Это означает, что модель значительно приблизилась к точному воспроизведению ответов на обучающих данных.

```
==((====))== Unsloth - 2x faster free finetuning | Num GPUs used = 1
  \ \  /|    Num examples = 250 | Num Epochs = 3 | Total steps = 96
0^0/ \_/ \   Batch size per device = 1 | Gradient accumulation steps = 8
 \ _____/ Data Parallel GPUs = 1 | Total batch size (1 x 8 x 1) = 8
"-_____"     Trainable parameters = 8,912,896/2,234,321,920 (0.40% trained)
wandb: WARNING The `run_name` is currently set to the same value as `TrainingArguments.output_dir`.
wandb: Currently logged in as: dr3ids (dr3ids-myself-germany) to https://api.wandb.ai. Use `wandb l
Tracking run with wandb version 0.19.11
Run data is saved locally in /content/wandb/run-20250615_201443-s49vkh05
Syncing run phi4-finetuned to Weights & Biases (docs)
View project at https://wandb.ai/dr3ids-myself-germany/huggingface
View run at https://wandb.ai/dr3ids-myself-germany/huggingface/runs/s49vkh05
`use_cache=True` is incompatible with gradient checkpointing. Setting `use_cache=False`.
Unsloth: Will smartly offload gradients to save VRAM!
[96/96 22:14, Epoch 3/3]
```

Step	Training Loss
10	0.766900
20	0.556000
30	0.491800
40	0.426400
50	0.408400
60	0.412500
70	0.380300
80	0.373600
90	0.368200

```
TrainOutput(global_step=96, training_loss=0.45857539772987366, metrics={'train_runtime': 1384.9136,
'train_samples_per_second': 0.542, 'train_steps_per_second': 0.069, 'total_flos':
1.1965475781568512e+16, 'train_loss': 0.45857539772987366})
```

Рисунок 7 – Процесс дообучения модели

Качество полученной модели: fine-tuning позволил модели Phi-4-mini адаптироваться к стилю и содержанию обучающего набора. Ассистент начал формулировать ответы в соответствии с заданным системным сообщением – как “эксперт по предметным областям”. На вопросы, похожие на тренировочные,

модель отвечает уверенно и развернуто, демонстрируя усвоенные факты и формулировки (рисунок 8).

```
INFO:llm_infer:Sending prompt to LLM...
INFO:llm_infer:Prompt length: 2699 characters
llama_perf_context_print:      load time =   98587.30 ms
llama_perf_context_print: prompt eval time =   98585.33 ms /   614 tokens (  160.56 ms per token,    6.23 tokens per second)
llama_perf_context_print:      eval time =  159141.43 ms /   342 runs (  465.33 ms per token,    2.15 tokens per second)
llama_perf_context_print:      total time =  258768.99 ms /   956 tokens
INFO:llm_infer:
=====
INFO:llm_infer:RAW LLM OUTPUT START
INFO:llm_infer:=====
INFO:llm_infer:{
  "concepts": [
    {"id": "C1", "label": "Архитектура трансформеров", "description": "Современная архитектура трансформеров, используемая в качестве основы для генерации документов."},
    {"id": "C2", "label": "PEFT", "description": "Методы параметрически эффективного дообучения, включающие адаптеры, промт-обучение, LoRA и QLoRA."},
    {"id": "C3", "label": "Автоматизированная система генерации документов", "description": "Система, состоящая из веб-приложения, серверной части и нейросетевого модуля."},
    {"id": "C4", "label": "Датасет", "description": "Специализированный набор данных, предназначенный для дообучения нейросетевой модели."},
    {"id": "C5", "label": "Phi-4-mini", "description": "Модель, использованная для дообучения с помощью QLoRA."}
  ],
  "relations": [
    {"from": "C1", "to": "C2", "type": "Используется"},
    {"from": "C3", "to": "C1", "type": "Основана на"},
    {"from": "C3", "to": "C2", "type": "Включает"},
    {"from": "C4", "to": "C5", "type": "Используется для"}
  ]
}
INFO:llm_infer:=====
INFO:llm_infer:RAW LLM OUTPUT END
INFO:llm_infer:=====
INFO:llm_infer:Successfully parsed ontology:
INFO:llm_infer:Concepts: 5
INFO:llm_infer:Relations: 4
```

Рисунок 8 – Вывод языковой модели

Целевое применение: Полученная модель планируется к использованию в качестве специализированного чат-бота или справочной системы, способной отвечать на вопросы из различных областей знаний на русском языке. Благодаря компактному размеру (2,2 млрд параметров) и дообучению, её можно внедрить в приложение, где требуется экспертный ассистент, например для образовательных целей или для поддержки пользователей в предметно-ориентированных задачах.

4.2 Результаты работы с программными средствами генерации документов на основе знаний о предметной области

На первом этапе работы в интерфейсе генератора документации пользователь вводит исходный текст, содержащий знания о предметной области. Примером такого текста может служить описание семантических моделей знаний, представленных стандартами RDF и OWL, их связи с внешними нормативами и концепциями образовательной области. Пользователь нажимает кнопку “Извлечь Онтологию”, после чего система успешно выполняет

извлечение онтологической структуры. Успешность операции подтверждается системным уведомлением “Онтология успешно извлечена!” (рисунок 9).

Генератор Документации

Редактор Онтологии Генерация Документа

Ввод Текста

Введите или вставьте текст:

Семантические модели знаний позволяют формализовать содержание курса, структуру образовательной программы и связи с внешними нормативами. Ключевыми стандартами семантического представления являются RDF и OWL. RDF описывает знания в виде графа триплетов, где каждое утверждение имеет форму субъект-предикат-объект. OWL расширяет RDF возможностью задавать классы, свойства, аксиомы, ограничения и логические выводы.

В контексте образования RDF/OWL используется для построения онтологий:

– онтологий дисциплин (тематика, ключевые понятия...

Извлечь Онтологию

Онтология успешно извлечена!

Визуализация Онтологии

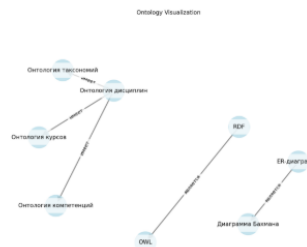


Рисунок 9 – Успешное извлечение онтологии

На втором этапе полученная онтология визуализируется в виде графа. В представленном графе узлы соответствуют отдельным концептам (например, “Онтология дисциплин”, “Онтология компетенций”, “RDF”, “OWL” и др.), а связи между ними иллюстрируют отношения (“имеет”, “является”) (рисунок 10).

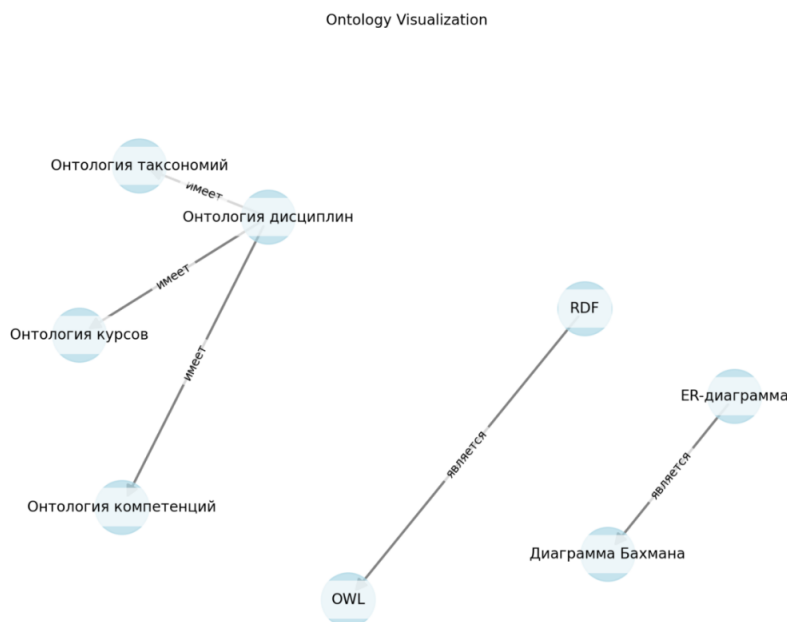


Рисунок 10 – Визуализация онтологии

Следующим шагом предоставляется возможность пользователю отредактировать извлеченную онтологию в специальном интерфейсе. Интерфейс содержит отдельные блоки для каждого концепта, в которых пользователь может изменять название и описание, а также удалять ненужные концепты. Также предусмотрена возможность редактирования или удаления связей между концептами (рисунок 11).

Редактирование Онтологии

The interface is divided into two main sections: 'Концепты:' (Concepts) and 'Связи:' (Relationships).

Концепты:

- Концепт Онтология дисциплин**
 - Название: Онтология дисциплин
 - Описание: Онтология, описывающая предметную область, ключевые понятия и цели обучения.
 - Удалить Концепт
- Концепт Онтология компетенций**
 - Название: Онтология компетенций
 - Описание: Онтология, описывающая универсальные, профессиональные и е-CF компетенции.
 - Удалить Концепт
- Концепт Онтология курсов**
 - Название: Онтология курсов
 - Описание: (empty)
 - Удалить Концепт

Связи:

- Связь 1**
 - От: Онтология дисц...
 - К: Онтология комп...
 - Тип: имеет
 - Удалить Связь
- Связь 2**
 - От: Онтология дисц...
 - К: Онтология курсов
 - Тип: имеет
 - Удалить Связь
- Связь 3**
 - От: Онтология дисц...
 - К: Онтология таксо...
 - Тип: имеет
 - Удалить Связь

Рисунок 11 – Редактирование онтологии

Далее пользователь переходит к этапу генерации документа, где доступен выбор предустановленного шаблона или загрузка своего собственного шаблона

в формате DOCX. Предусмотрена возможность предпросмотра шаблона, где отображаются все доступные переменные, которые будут автоматически заполнены данными из онтологии. Среди переменных представлены общие данные о документации (дата генерации, количество концептов, связей), а также детальные описания концептов и их взаимосвязей (рисунок 12).

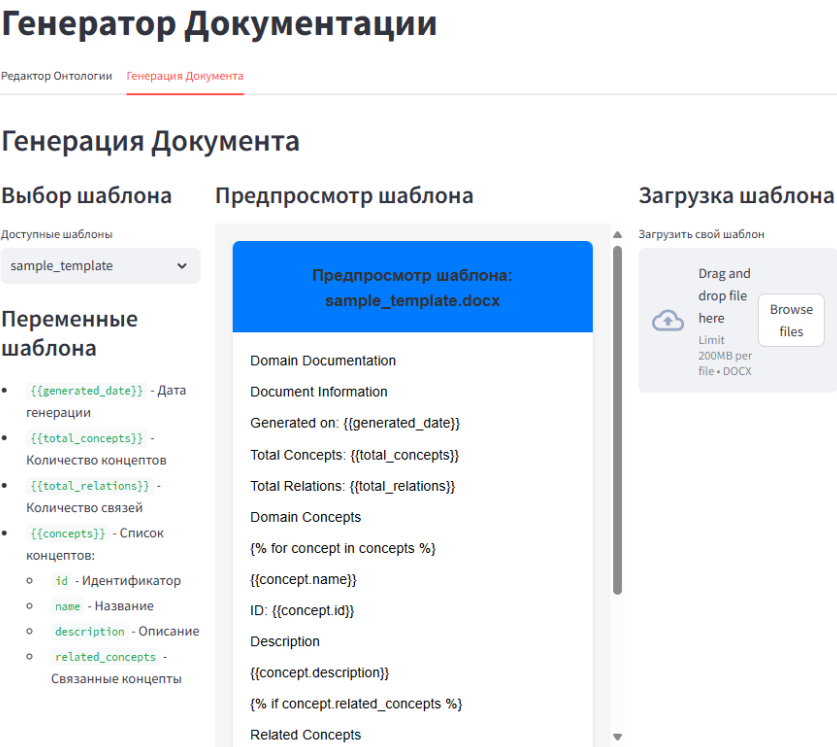


Рисунок 12 – Выбор предустановленного шаблона

Важной составляющей процесса является информация о самом шаблоне документа, включая количество страниц, слов и размер файла, а также список переменных, доступных в шаблоне (рисунок 13).

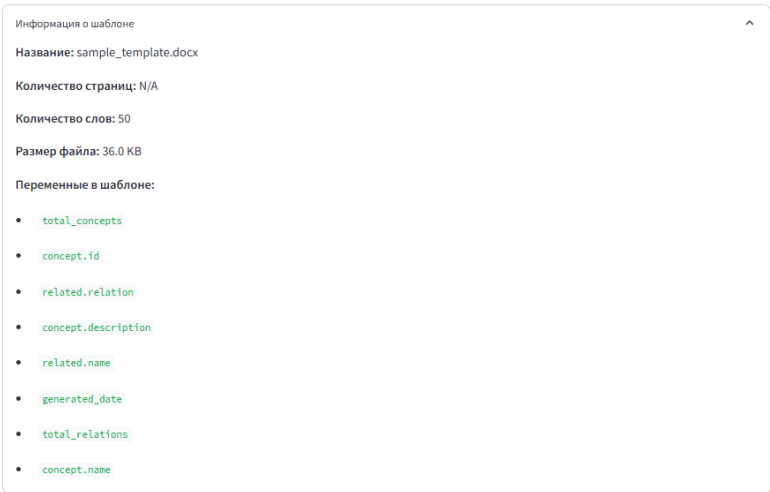
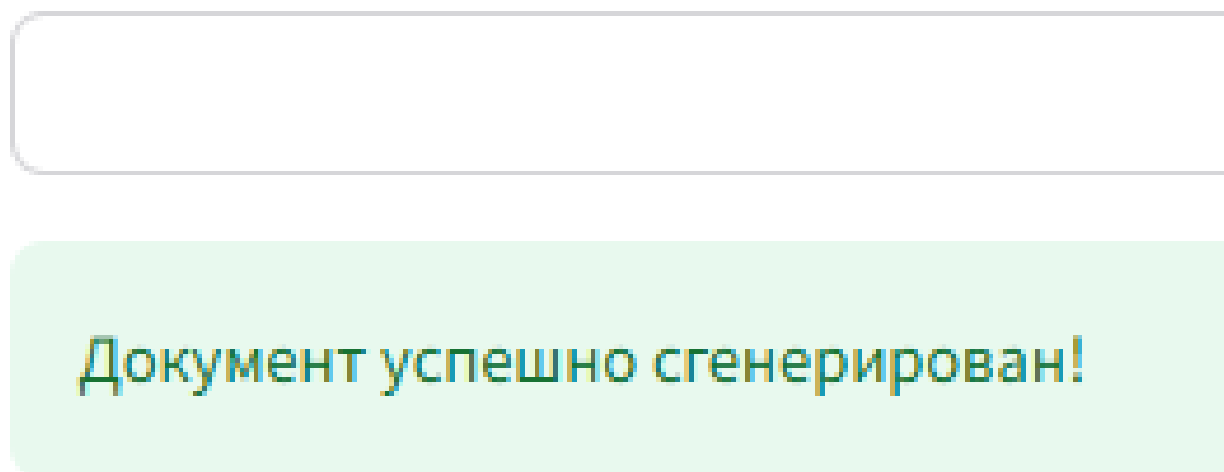


Рисунок 13 – Выбор предустановленного шаблона

После того как пользователь завершает выбор и редактирование шаблона, он инициирует процесс генерации документа. Система формирует готовый к загрузке файл и подтверждает успешность генерации соответствующим уведомлением с возможностью скачивания готового документа (рисунок 14).



[Скачать Документ](#)

Рисунок 14 – Успешная генерация документа

Итоговый сгенерированный документ представляет собой структурированный и наполненный информацией файл в формате DOCX, который включает сведения о доменной документации: дату генерации, количество концептов и отношений, а также подробное описание каждого концепта и связанных с ним других концептов. Например, концепт “Онтология дисциплин” описывается как онтология, охватывающая предметную область, ключевые понятия и цели обучения. Также перечисляются связанные с ним концепты, например “Онтология компетенций”, “Онтология курсов” и “Онтология таксономий” с указанием типа связи (“имеет”) (рисунок 15).

Domain Documentation

Document Information

Generated on: 2025-06-23 03:12:11

Total Concepts: 8

Total Relations: 5

■ Domain Concepts

Онтология дисциплин

ID: C1

Description

Онтология, описывающая предметную область, ключевые понятия и цели обучения.

Related Concepts

- Онтология компетенций (имеет)
- Онтология курсов (имеет)
- Онтология таксономий (имеет)

Рисунок 15 – Пример генерации документа

Таким образом, реализованное программное средство обеспечивает эффективное создание и управление документами на основе структурированных знаний о предметной области, демонстрируя практическую применимость и удобство в решении задач автоматизации документирования.

ЗАКЛЮЧЕНИЕ

В ходе выполнения настоящей работы были достигнуты следующие результаты:

- проведен анализ существующих решений и подходов к автоматизированной генерации документов в образовательной среде,
- проведён анализ современной архитектуры трансформеров и возможностей её применения в качестве базовой технологии для генерации документов на основе знаний о предметной области,
- изучены и подробно описаны методы параметрически эффективного дообучения (PEFT), включая адаптеры, промпт-обучение, LoRA и QLoRA, выделены их преимущества и ограничения,
- предложена архитектура автоматизированной системы генерации документов, состоящая из веб-приложения с графическим интерфейсом, серверной части, основанной на языке Python, и нейросетевого модуля, реализованного с применением подходов PEFT,
- проведена практическая реализация системы с использованием языка программирования Python, фреймворков FastAPI и Streamlit, а также библиотеки Unsloth для эффективного дообучения языковой модели с помощью QLoRA,
- собран специализированный датасет на основе предметных онтологий, предназначенный для целевого дообучения нейросетевой модели,
- выполнено дообучение модели Phi-4-mini методом QLoRA с использованием собранного датасета, что позволило адаптировать большую языковую модель к специфическим задачам генерации документов на основе формализованных знаний,
- результаты дообучения продемонстрировали возможность высокоэффективного и ресурсосберегающего переноса знаний предметных областей в большие языковые модели, обеспечивая существенное снижение затрат на вычислительные ресурсы и память при сохранении высокого качества генерации текстовых документов.

Полученные результаты подтверждают перспективность применения PEFT-методов и, в частности, подхода QLoRA для решения практических задач генерации документов на основе структурированных знаний. Реализованная система способна существенно ускорить и автоматизировать процессы подготовки специализированных документов, таких как учебные программы, аннотации и отчётная документация, обеспечивая при этом высокую точность и согласованность генерируемых материалов. Разработанный подход и проведённые эксперименты могут быть полезны при создании аналогичных систем и в других предметных областях.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Золкин А. Л., Ахмадуллин Ф. Р. Проектирование онтологий для экспертных систем: учебное пособие. – СПб: Лань, 2025. – 132 с.
2. Лапшин В. А. Онтологии в компьютерных системах. – М.: URSS, 2010. – 224 с.
3. Chicaiza, J., Tapia-León, M., Aveiga, C., Suárez-Figueroa, M. C. Ontological Model for the Semantic Description of Syllabuses. – Proceedings of ACM conference, 2020. – 12 p.
4. Kiptoo, C. C., et al. Curriculum Course Syllabus Ontology (CCSO). – ESWC Satellite Events, 2018. – 8 p.
5. Aminah, S., Alfa Krisnadhi, A., Hidayanto, A. N. Ontological Framework for the Analysis of Outcome-Based Curriculum in Higher Education. – IEEE Access, 2025, Vol. 13, pp. 31497–31516.
6. 1С: УНИВЕРСИТЕТ. Автоматизация управления вузом. – Официальный сайт 1С: решения ВУЗ. – URL: <https://solutions.1c.ru/catalog/university> (дата обращения: 23.06.2025).
7. Апекс-ВУЗ. Система автоматизации образовательного процесса. – Официальный сайт Апекс-ВУЗ. – URL: <https://apeks-vuz.ru/> (дата обращения: 23.06.2025).
8. Vaswani A., Shazeer N., Parmar N., et al. Attention is All You Need // Proc. of NIPS, 2017. – Pp. 5998–6008.
9. Brown T., Mann B., Ryder N., et al. Language Models are Few-Shot Learners // Advances in Neural Information Processing Systems, 2020. – Vol. 33. – Pp. 1877–1901.
10. Houlsby N., Giurgiu A., Jastrzebski S., et al. Parameter-Efficient Transfer Learning for NLP // Proc. of ICML, 2019. – Pp. 2790–2799.
11. Li X. L., Liang P. Prefix-Tuning: Optimizing Continuous Prompts for Generation // Proc. of ACL, 2021. – Pp. 4582–4597.

12. Hu E., Shen Y., Wallis P., et al. LoRA: Low-Rank Adaptation of Large Language Models // arXiv preprint arXiv:2106.09685, 2021.
13. Dettmers T., Pagnoni A., Holtzman A., et al. QLoRA: Efficient Finetuning of Quantized LLMs // Proc. of ICML, 2023.
14. Xie H., Si J., Xie P., et al. Parameter-Efficient Fine-Tuning for Large Models: A Survey. – arXiv:2403.14608, 2024.
15. Шунейко А. А. Корпусная лингвистика. – М.: Юрайт, 2020. – 222 с.
16. Большакова Е. И., Клышинский Э. С., Ландэ Д. В., и др. Автоматическая обработка текстов на естественном языке и компьютерная лингвистика: учеб. пособие. – М.: НИУ ВШЭ, 2017. – 272 с.
17. Javaheripi M., Bubeck S. Phi-2: The surprising power of small language models // Microsoft Research Blog, 12.12.2023.
18. Кайда Н. Делаем конкурента DeepSeek R1-Zero на домашней пекарне: метод GRPO в Unsloth // Proglab, 12.02.2025.
19. Wolf T., Debut L., Sanh V., et al. Transformers: State-of-the-Art Natural Language Processing // Proc. of EMNLP 2020: System Demonstrations, 2020. – Pp. 38–45.
20. Transformers (Hugging Face) – URL: <https://huggingface.co/docs/transformers/index> (дата обращения: 23.06.2025).
21. Шатухин Д. Исследователи из Unsloth сжали DeepSeek R1 на 80% от оригинального размера // Хабр (новости), 28.01.2025.

ПРИЛОЖЕНИЕ А

(обязательное)

Программные средства генерации документов на основе знаний о предметной
области

Листинг программы

ПРИЛОЖЕНИЕ А.1

Файл “docx_renderer.py”

```
from docxtpl import DocxTemplate
from typing import Dict
import os
from datetime import datetime

class DocxRenderer:
    def __init__(self):
        self.output_dir = "output"
        os.makedirs(self.output_dir, exist_ok=True)

    def _prepare_template_data(self, ontology: Dict) -> Dict:
        """
        Prepare data for template rendering from ontology.

        Args:
            ontology (Dict): Ontology in JSON format

        Returns:
            Dict: Data ready for template rendering
        """
        # Convert concepts to a more template-friendly format
        concepts_list = []
        for concept in ontology["concepts"]:
            concept_data = {
                "id": concept["id"],
                "name": concept["label"],
                "description": concept["description"],
                "related_concepts": []
            }

            # Find related concepts
            for relation in ontology["relations"]:
                if relation["from"] == concept["id"]:
                    related_concept = next(
                        (c for c in ontology["concepts"] if
                         c["id"] == relation["to"]),
                        None
                    )
                    if related_concept:
                        concept_data["related_concepts"].append({
                            "name": related_concept["label"],
                            "relation": relation["type"]
                        })

            concepts_list.append(concept_data)

        return {
            "concepts": concepts_list,
```

```

        "generated_date": datetime.now().strftime("%Y-%m-%d
%H:%M:%S"),
        "total_concepts": len(concepts_list),
        "total_relations": len(ontology["relations"])
    }

    def render_document(self, template_path: str, ontology: Dict)
-> str:
    """
    Render a document from template using ontology data.

    Args:
        template_path (str): Path to the DOCX template
        ontology (Dict): Ontology in JSON format

    Returns:
        str: Path to the generated document
    """
    try:
        # Load template
        doc = DocxTemplate(template_path)

        # Prepare data
        template_data = self._prepare_template_data(ontology)

        # Render document
        doc.render(template_data)

        # Generate output filename
        timestamp = datetime.now().strftime("%Y%m%d_%H%M%S")
        output_filename = f"document_{timestamp}.docx"
        output_path = os.path.join(self.output_dir,
output_filename)

        # Save document
        doc.save(output_path)

        return output_path

    except Exception as e:
        raise ValueError(f"Failed to render document:
{str(e)}")

    def get_template_variables(self, template_path: str) -> list:
    """
    Get list of variables used in the template.

    Args:
        template_path (str): Path to the DOCX template

    Returns:
        list: List of variable names used in the template
    """

```

```
try:
    doc = DocxTemplate(template_path)
    return doc.get_undeclared_template_variables()
except Exception as e:
    raise ValueError(f"Failed to get template variables:
{str(e)}")
```


ПРИЛОЖЕНИЕ А.2

Файл “llm_infer.py”

```
from llama_cpp import Llama
import json
from typing import Dict, List
import logging
import re

# Configure logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

class OntologyExtractor:
    def __init__(self):
        self.model_path = "../models/phi4-objfield-q4_K_M.gguf"
        self.llm = Llama(
            model_path=self.model_path,
            n_ctx=2048, # Context window
            n_threads=4, # Number of CPU threads to use
            n_gpu_layers=0 # CPU only
        )

    def _create_prompt(self, text: str) -> str:
        return f"""Extract an ontology from this text:

{text}

Return a single JSON object with this exact structure:

{{
  "concepts": [
    {{ "id": "C1", "label": "Concept1", "description":
"Description1" }},
    {{ "id": "C2", "label": "Concept2", "description":
"Description2" }}
  ],
  "relations": [
    {{ "from": "C1", "to": "C2", "type": "relation1" }}
  ]
}}

Rules:
1. Use Russian for labels, descriptions, and relation types, end
your response with </json> marker
2. Extract as much entities and relations as possible from the
text, but do not hallucinate

JSON OUTPUT:
<json>"""

    def extract_ontology(self, text: str) -> Dict:
```

```

prompt = self._create_prompt(text)
logger.info("Sending prompt to LLM...")
logger.info(f"Prompt length: {len(prompt)} characters")

output = self.llm(
    prompt,
    max_tokens=4096,
    temperature=0.0,
    stop=["</json>"], # Single clear stop marker
    echo=False # Don't include the prompt in the output
)

response = output["choices"][0]["text"].strip()
logger.info("\n" + "="*50)
logger.info("RAW LLM OUTPUT START")
logger.info("="*50)
logger.info(response)
logger.info("="*50)
logger.info("RAW LLM OUTPUT END")
logger.info("="*50 + "\n")

try:
    ontology = json.loads(response)

    # Validate the structure
    if not isinstance(ontology, dict):
        logger.error("Response is not a dictionary")
        raise ValueError("Response is not a dictionary")
    if "concepts" not in ontology or "relations" not in
ontology:
        logger.error("Missing required fields: concepts or
relations")
        raise ValueError("Missing required fields:
concepts or relations")
    if not isinstance(ontology["concepts"], list) or not
isinstance(ontology["relations"], list):
        logger.error("Concepts and relations must be
lists")
        raise ValueError("Concepts and relations must be
lists")

    # Validate each concept
    for concept in ontology["concepts"]:
        if not isinstance(concept, dict):
            raise ValueError(f"Invalid concept format:
{concept}")
        if "id" not in concept or "label" not in concept
or "description" not in concept:
            raise ValueError(f"Concept missing required
fields: {concept}")
        if not concept["label"] or not
concept["description"]:

```

```

        raise ValueError(f"Concept has empty label or
description: {concept}")

    # Validate each relation
    for relation in ontology["relations"]:
        if not isinstance(relation, dict):
            raise ValueError(f"Invalid relation format:
{relation}")
        if "from" not in relation or "to" not in relation
or "type" not in relation:
            raise ValueError(f"Relation missing required
fields: {relation}")
        if not relation["type"]:
            raise ValueError(f"Relation has empty type:
{relation}")

    logger.info("Successfully parsed ontology:")
    logger.info(f"Concepts: {len(ontology['concepts'])}")
    logger.info(f"Relations:
{len(ontology['relations'])}")
    return ontology
except json.JSONDecodeError as e:
    logger.error(f"JSON parsing error: {e}")
    logger.error(f"Error position: {e.pos}")
    logger.error(f"Line: {e.lineno}, Column: {e.colno}")
    raise ValueError(f"Failed to parse LLM output as JSON:
{e}")

# Initialize the extractor
extractor = OntologyExtractor()

def extract_ontology_from_text(text: str) -> Dict:
    """
    Extract ontology from text using the LLM model.

    Args:
        text (str): Input text to extract ontology from

    Returns:
        Dict: Extracted ontology in the format:
        {
            "concepts": [
                {"id": "C1", "label": "Concept", "description":
"Description"}
            ],
            "relations": [
                {"from": "C1", "to": "C2", "type": "relation"}
            ]
        }
    """
    return extractor.extract_ontology(text)

```

ПРИЛОЖЕНИЕ А.3

Файл “main.py”

```
from fastapi import FastAPI, UploadFile, File, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from fastapi.responses import FileResponse, HTMLResponse
from pydantic import BaseModel
from typing import List, Dict, Optional
import json
import os
from pathlib import Path

from llm_infer import extract_ontology_from_text
from ontology import OntologyManager
from docx_renderer import DocxRenderer

app = FastAPI(title="Documentation Generator API")

# Enable CORS
app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_credentials=True,
    allow_methods=["*"],
    allow_headers=["*"],
)

# Initialize components
ontology_manager = OntologyManager()
docx_renderer = DocxRenderer()

# Ensure directories exist
os.makedirs("templates", exist_ok=True)
os.makedirs("output", exist_ok=True)

class TextInput(BaseModel):
    text: str

class OntologyUpdate(BaseModel):
    concepts: List[Dict]
    relations: List[Dict]

@app.post("/extract_ontology")
async def extract_ontology(input_data: TextInput):
    try:
        ontology = extract_ontology_from_text(input_data.text)
        return ontology
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/update_ontology")
async def update_ontology(ontology: OntologyUpdate):
```

```

    try:
        updated_ontology = ontology_manager.update_ontology(
            ontology.concepts,
            ontology.relations
        )
        return updated_ontology
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/render_doc")
async def render_document(ontology: OntologyUpdate):
    try:
        # Get the first available template
        templates = [f for f in os.listdir("templates") if
f.endswith('.docx')]
        if not templates:
            raise HTTPException(status_code=404, detail="No
templates found")

        template_path = os.path.join("templates", templates[0])

        # Generate document
        doc_path = docx_renderer.render_document(
            template_path,
            ontology.dict()
        )

        # Return the full path
        return {"document_path": doc_path}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/download/{document_name}")
async def download_document(document_name: str):
    try:
        doc_path = os.path.join("output", document_name)
        if not os.path.exists(doc_path):
            raise HTTPException(status_code=404, detail="Document
not found")
        return FileResponse(
            doc_path,
            media_type="application/vnd.openxmlformats-
officedocument.wordprocessingml.document",
            filename=document_name
        )
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/templates")
async def list_templates():
    templates = []
    if os.path.exists("templates"):

```

```

        templates = [f for f in os.listdir("templates") if
f.endswith('.docx')]
        return {"templates": templates}

@app.get("/template/preview/{template_name}")
async def get_template_preview(template_name: str):
    try:
        from template_preview import TemplatePreviewRenderer

        renderer = TemplatePreviewRenderer()
        preview_html =
renderer.render_template_preview(template_name)

        if preview_html:
            return HTMLResponse(content=preview_html)
        else:
            raise HTTPException(status_code=404, detail="Template
not found or preview failed")
    except ImportError as e:
        raise HTTPException(status_code=500, detail=f"Template
preview module not available: {str(e)}")
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.get("/template/info/{template_name}")
async def get_template_info(template_name: str):
    try:
        from template_preview import TemplatePreviewRenderer

        renderer = TemplatePreviewRenderer()
        info = renderer.get_template_info(template_name)

        if info:
            return info
        else:
            raise HTTPException(status_code=404, detail="Template
not found")
    except ImportError as e:
        raise HTTPException(status_code=500, detail=f"Template
preview module not available: {str(e)}")
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)

```

ПРИЛОЖЕНИЕ А.4

Файл “ontology.py”

```
# from typing import List, Dict, Optional
import json
from rdflib import Graph, Literal, BNode, Namespace, URIRef
from rdflib.namespace import RDF, RDFS

class OntologyManager:
    def __init__(self):
        self.graph = Graph()
        self.ns = Namespace("http://example.org/ontology/")
        self.graph.bind("ex", self.ns)

    def _concept_to_rdf(self, concept: Dict) -> URIRef:
        """Convert a concept dictionary to RDF triples."""
        concept_uri = self.ns[concept["id"]]
        self.graph.add((concept_uri, RDF.type, self.ns.Concept))
        self.graph.add((concept_uri, RDFS.label,
Literal(concept["label"])))
        self.graph.add((concept_uri, self.ns.description,
Literal(concept["description"])))
        return concept_uri

    def _relation_to_rdf(self, relation: Dict) -> None:
        """Convert a relation dictionary to RDF triples."""
        from_uri = self.ns[relation["from"]]
        to_uri = self.ns[relation["to"]]
        relation_uri = self.ns[relation["type"]]
        self.graph.add((from_uri, relation_uri, to_uri))

    def update_ontology(self, concepts: List[Dict], relations:
List[Dict]) -> Dict:
        """
        Update the ontology with new concepts and relations.

        Args:
            concepts (List[Dict]): List of concept dictionaries
            relations (List[Dict]): List of relation dictionaries

        Returns:
            Dict: Updated ontology in JSON format
        """
        # Clear existing graph
        self.graph = Graph()
        self.graph.bind("ex", self.ns)

        # Add concepts
        for concept in concepts:
            self._concept_to_rdf(concept)

        # Add relations
```

```

        for relation in relations:
            self._relation_to_rdf(relation)

        # Convert back to JSON format
        return self.to_json()

def to_json(self) -> Dict:
    """
    Convert the current ontology to JSON format.

    Returns:
        Dict: Ontology in JSON format
    """
    concepts = []
    relations = []

    # Extract concepts
    for s, p, o in self.graph.triples((None, RDF.type,
self.ns.Concept)):
        concept = {
            "id": str(s).split("/")[-1],
            "label": str(self.graph.value(s, RDFS.label)),
            "description": str(self.graph.value(s,
self.ns.description))
        }
        concepts.append(concept)

    # Extract relations
    for s, p, o in self.graph.triples((None, None, None)):
        if p != RDF.type and p != RDFS.label and p !=
self.ns.description:
            relation = {
                "from": str(s).split("/")[-1],
                "to": str(o).split("/")[-1],
                "type": str(p).split("/")[-1]
            }
            relations.append(relation)

    return {
        "concepts": concepts,
        "relations": relations
    }

def validate_ontology(self) -> bool:
    """
    Validate the current ontology structure.

    Returns:
        bool: True if ontology is valid, False otherwise
    """
    # Check if all concepts have required properties
    for s, p, o in self.graph.triples((None, RDF.type,
self.ns.Concept)):

```



```

        if not self.graph.value(s, RDFS.label):
            return False
        if not self.graph.value(s, self.ns.description):
            return False

        # Check if all relations reference existing concepts
        concept_ids = {str(s).split("/")[-1] for s, p, o in
self.graph.triples((None, RDF.type, self.ns.Concept))}

        for s, p, o in self.graph.triples((None, None, None)):
            if p != RDF.type and p != RDFS.label and p !=
self.ns.description:
                from_id = str(s).split("/")[-1]
                to_id = str(o).split("/")[-1]
                if from_id not in concept_ids or to_id not in
concept_ids:
                    return False

        return True

```

ПРИЛОЖЕНИЕ А.5

Файл “template_preview.py”

```
from docx import Document
import os
import re
from typing import Optional, Dict, List

class TemplatePreviewRenderer:
    def __init__(self):
        self.templates_dir = "templates"
        self.output_dir = "output"
        os.makedirs(self.output_dir, exist_ok=True)

    def render_template_preview(self, template_name: str,
sample_data: dict = None) -> Optional[str]:
    """
    Render a preview of the DOCX template with sample data.

    Args:
        template_name (str): Name of the template file
        sample_data (dict): Sample data to populate the
template

    Returns:
        str: HTML preview or None if error
    """
    try:
        template_path = os.path.join(self.templates_dir,
template_name)
        if not os.path.exists(template_path):
            return None

        # Load the document
        doc = Document(template_path)

        # Extract text content
        content = []
        for paragraph in doc.paragraphs:
            if paragraph.text.strip():
                content.append(f"<p>{paragraph.text}</p>")

        # Extract table content
        for table in doc.tables:
            table_html = "<table border='1' style='border-
collapse: collapse, width: 100%, margin: 10px 0, '>"
            for row in table.rows:
                table_html += "<tr>"
                for cell in row.cells:
                    table_html += f"<td style='padding: 8px,
border: 1px solid #ddd, '>{cell.text}</td>"
                table_html += "</tr>"
```

```

        table_html += "</table>"
        content.append(table_html)

# Join all content
html_content = "\n".join(content)

# Add styling
styled_html = f"""
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <style>
        body {{
            font-family: Arial, sans-serif,
            margin: 20px,
            background-color: #f5f5f5,
            line-height: 1.6,
        }}
        .preview-container {{
            background: white,
            padding: 20px,
            border-radius: 8px,
            box-shadow: 0 2px 4px rgba(0,0,0,0.1),
            max-width: 800px,
            margin: 0 auto,
        }}
        .preview-header {{
            background: #007bff,
            color: white,
            padding: 10px,
            margin: -20px -20px 20px -20px,
            border-radius: 8px 8px 0 0,
            text-align: center,
        }}
        p {{
            margin: 10px 0,
            padding: 0,
        }}
        h1, h2, h3, h4, h5, h6 {{
            color: #333,
            margin: 15px 0 10px 0,
        }}
        .template-variable {{
            background-color: #fff3cd,
            border: 1px solid #ffeaa7,
            padding: 2px 4px,
            border-radius: 3px,
            font-family: monospace,
        }}
    </style>
</head>
<body>

```

```

        <div class="preview-container">
            <div class="preview-header">
                <h3>Предпросмотр шаблона:
{template_name}</h3>
            </div>
            <div class="content">
                {html_content}
            </div>
        </div>
    </body>
</html>
    """

    return styled_html

except Exception as e:
    print(f"Error rendering template preview: {str(e)}")
    return None

def get_template_info(self, template_name: str) -> dict:
    """
    Get basic information about the template.

    Args:
        template_name (str): Name of the template file

    Returns:
        dict: Template information
    """
    try:
        template_path = os.path.join(self.templates_dir,
template_name)
        if not os.path.exists(template_path):
            return {}

        doc = Document(template_path)

        # Extract text content
        content = []
        for paragraph in doc.paragraphs:
            content.append(paragraph.text)

        # Count words
        full_text = " ".join(content)
        word_count = len(full_text.split())

        # Extract template variables
        variables = []
        var_pattern = r'\{\{([^\}]+)\}\}'
        matches = re.findall(var_pattern, full_text)
        variables = list(set(matches)) # Remove duplicates

        # Count paragraphs and tables

```

```

        paragraph_count = len([p for p in doc.paragraphs if
p.text.strip()])
        table_count = len(doc.tables)

        return {
            "name": template_name,
            "paragraph_count": paragraph_count,
            "table_count": table_count,
            "word_count": word_count,
            "variables": variables,
            "file_size": os.path.getsize(template_path)
        }

    except Exception as e:
        print(f"Error getting template info: {str(e)}")
        return {}

def highlight_variables(self, text: str) -> str:
    """
    Highlight template variables in text.

    Args:
        text (str): Text to process

    Returns:
        str: Text with highlighted variables
    """
    var_pattern = r'(\{\{[^\}]+\}\})'
    return re.sub(var_pattern, r'<span class="template-
variable">\1</span>', text)

```

ПРИЛОЖЕНИЕ А.6

Файл “app.py”

```
import streamlit as st
import requests
import json
import os
from pathlib import Path
import networkx as nx
import matplotlib.pyplot as plt
import io

# API endpoint
API_URL = "http://localhost:8000"

# Configure page to use full width
st.set_page_config(layout="wide")

def main():
    st.title("Генератор Документации")

    # Initialize session state for ontology if not exists
    if "ontology" not in st.session_state:
        st.session_state.ontology = None

    # Create tabs
    tab1, tab2 = st.tabs(["Редактор Онтологии", "Генерация
Документа"])

    # Tab 1: Combined Text Input and Ontology Editor
    with tab1:
        # Create two columns for text input and ontology
        visualization
        col1, col2 = st.columns([1, 1])

        with col1:
            st.subheader("Ввод Текста")
            text_input = st.text_area("Введите или вставьте
текст:", height=300)

            if st.button("Извлечь Онтологию"):
                if text_input:
                    with st.spinner("Извлечение онтологии..."):
                        try:
                            response = requests.post(
                                f"{API_URL}/extract_ontology",
                                json={"text": text_input}
                            )
                            if response.status_code == 200:
                                st.session_state.ontology =
response.json()
```

```

        st.success("Онтология успешно
извлечена!")
    else:
        st.error(f"Ошибка:
{response.text}")
    except Exception as e:
        st.error(f"Ошибка: {str(e)}")
    else:
        st.warning("Пожалуйста, введите текст.")

with col2:
    st.subheader("Визуализация Онтологии")
    if st.session_state.ontology:
        # Create a directed graph
        G = nx.DiGraph()

        # Add nodes with their labels
        for concept in
st.session_state.ontology["concepts"]:
            G.add_node(concept["id"],
                        label=concept["label"],
                        description=concept["description"])

        # Add edges with their types
        for relation in
st.session_state.ontology["relations"]:
            G.add_edge(relation["from"], relation["to"],
                        type=relation["type"])

        # Create a mapping of IDs to labels for better
visualization
        id_to_label = {concept["id"]: concept["label"] for
concept in st.session_state.ontology["concepts"]}

        # Create the graph visualization
        pos = nx.spring_layout(G, k=2, iterations=50) #
Increased k for more spacing

        # Create the figure with increased size
        plt.figure(figsize=(15, 10)) # Increased figure
size

        # Draw nodes with larger size and more padding
        nx.draw_networkx_nodes(G, pos,
                                node_color='lightblue',
                                node_size=2000, # Increased
node size
                                alpha=0.7)

        # Draw edges with arrows
        nx.draw_networkx_edges(G, pos,
                                edge_color='gray',
                                arrows=True,

```

```

arrow size                                arrowsize=20, # Increased
width                                    width=2) # Increased edge

padding                                # Draw labels with increased font size and more
nx.draw_networkx_labels(G, pos,
                        labels=id_to_label,
                        font_size=12, # Increased
font size
                        font_family='sans-serif',
                        bbox=dict(facecolor='white',
                                edgecolor='none',
                                alpha=0.7,
                                pad=10)) #
Increased padding

                        # Draw edge labels with increased font size
edge_labels = nx.get_edge_attributes(G, 'type')
nx.draw_networkx_edge_labels(G, pos,

edge_labels=edge_labels,
                        font_size=10, #
Increased font size

bbox=dict(facecolor='white',
edgecolor='none',
                        alpha=0.7,
                        pad=5)) #
Added padding

plt.title("Ontology Visualization", pad=20) #
Added padding to title
plt.axis('off') # Hide axes

# Add more margin space around the graph
plt.margins(x=0.2, y=0.2) # Increased margins

# Display the graph
st.pyplot(plt)
else:
    st.info("Извлеките онтологию из текста для
просмотра визуализации")

# Ontology editing section (full width)
if st.session_state.ontology:
    st.subheader("Редактирование Онтологии")

# Create two columns for concepts and relations
edit_coll, edit_col2 = st.columns(2)

```



```

        with edit_coll:
            st.write("Концепты:")
            for i, concept in
enumerate(st.session_state.ontology["concepts"]):
                with st.expander(f"Концепт
{concept['label']}"):
                    expanded=True):

st.session_state.ontology["concepts"][i]["label"] = st.text_input(
                    "Название", concept["label"],
key=f"label_{i}"
                )

st.session_state.ontology["concepts"][i]["description"] =
st.text_area(
                    "Описание", concept["description"],
key=f"desc_{i}", height=100
                )
                if st.button("Удалить Концепт",
key=f"del_{i}"):

st.session_state.ontology["concepts"].pop(i)
                    st.experimental_rerun()

                if st.button("Добавить Новый Концепт"):
                    new_id =
f"C{len(st.session_state.ontology['concepts']) + 1}"
                    st.session_state.ontology["concepts"].append({
                        "id": new_id,
                        "label": "",
                        "description": ""
                    })
                    st.experimental_rerun()

        with edit_col2:
            st.write("Связи:")
            for i, relation in
enumerate(st.session_state.ontology["relations"]):
                with st.expander(f"Связь {i+1}",
expanded=True):

                    col1, col2 = st.columns(2)
                    with col1:

st.session_state.ontology["relations"][i]["from"] = st.selectbox(
                        "От", [c["id"] for c in
st.session_state.ontology["concepts"]],
                        index=[c["id"] for c in
st.session_state.ontology["concepts"]].index(relation["from"]),
                        key=f"from_{i}",
                        format_func=lambda x:
next((c["label"] for c in st.session_state.ontology["concepts"] if
c["id"] == x), x)
                    )
                    with col2:

```

```

st.session_state.ontology["relations"][i]["to"] = st.selectbox(
    "K", [c["id"] for c in
st.session_state.ontology["concepts"]],
    index=[c["id"] for c in
st.session_state.ontology["concepts"]].index(relation["to"]),
    key=f"to_{i}",
    format_func=lambda x:
next((c["label"] for c in st.session_state.ontology["concepts"] if
c["id"] == x), x)
)

st.session_state.ontology["relations"][i]["type"] = st.text_input(
    "Тип", relation["type"],
key=f"type_{i}"
)
if st.button("Удалить Связь",
key=f"del_rel_{i}"):
st.session_state.ontology["relations"].pop(i)
st.experimental_rerun()

if st.button("Добавить Новую Связь"):
st.session_state.ontology["relations"].append({
    "from":
st.session_state.ontology["concepts"][0]["id"],
    "to":
st.session_state.ontology["concepts"][0]["id"],
    "type": ""
})
st.experimental_rerun()

# Save ontology changes button (full width)
if st.button("Сохранить Изменения Онтологии"):
    try:
        response = requests.post(
            f"{API_URL}/update_ontology",
            json=st.session_state.ontology
        )
        if response.status_code == 200:
            st.success("Онтология успешно обновлена!")
        else:
            st.error(f"Ошибка: {response.text}")
    except Exception as e:
        st.error(f"Ошибка: {str(e)}")

# Tab 2: Document Generation
with tab2:
    st.header("Генерация Документа")

    if st.session_state.ontology:
        # Get available templates

```

```

try:
    response = requests.get(f"{API_URL}/templates")
    if response.status_code == 200:
        templates = response.json()["templates"]

        if templates:
            # Create three columns for template
            selection, preview, and upload
            col1, col2, col3 = st.columns([1, 2, 1])

            with col1:
                st.subheader("Выбор шаблона")
                selected_template = st.selectbox(
                    "Доступные шаблоны",
                    templates,
                    format_func=lambda x:
x.replace('.docx', ''))

            if selected_template:
                # Show template variables
                st.subheader("Переменные шаблона")
                st.markdown("""
генерации
Количество концептов
Количество связей
концептов:
Связанные концепты

                - `{{generated_date}}` - Дата
                - `{{total_concepts}}` -
                - `{{total_relations}}` -
                - `{{concepts}}` - Список
                    - `id` - Идентификатор
                    - `name` - Название
                    - `description` - Описание
                    - `related_concepts` -
                """)

            with col2:
                st.subheader("Предпросмотр шаблона")
                if selected_template:
                    # Show template preview
                    try:
                        # Get template preview
                        preview_response =
requests.get(f"{API_URL}/template/preview/{selected_template}")
                        if
preview_response.status_code == 200:
                            # Display the HTML preview

st.components.v1.html(preview_response.text, height=600,
scrolling=True)
                    else:

```

```

документе**

st.markdown("""
### Структура документа
1. **Заголовок документа**
2. **Информация о
    - Дата генерации
    - Количество концептов
    - Количество связей
3. **Концепты домена**
    - Название концепта
    - ID
    - Описание
    - Связанные концепты
""")
except Exception as e:
    st.markdown("""
### Структура документа
1. **Заголовок документа**
2. **Информация о документе**
    - Дата генерации
    - Количество концептов
    - Количество связей
3. **Концепты домена**
    - Название концепта
    - ID
    - Описание
    - Связанные концепты
""")

# Show template information
try:
    info_response =
requests.get(f"{API_URL}/template/info/{selected_template}")
    if info_response.status_code
== 200:
        info =
info_response.json()
        with
st.expander("Информация о шаблоне"):
st.write(f"**Название:** {info.get('name', 'N/A')}")
st.write(f"**Количество страниц:** {info.get('page_count',
'N/A')}")
st.write(f"**Количество слов:** {info.get('word_count', 'N/A')}")
        st.write(f"**Размер
файла:** {info.get('file_size', 0) / 1024:.1f} KB")

        if
info.get('variables'):
st.write("**Переменные в шаблоне:**")

```

```

for var in
info['variables']:
    st.write(f"--
`{var}`")
except Exception as e:
    st.error(f"Не удалось
загрузить информацию о шаблоне: {str(e)}")

with col3:
    st.subheader("Загрузка шаблона")
    # Upload custom template
    uploaded_file =
st.file_uploader("Загрузить свой шаблон", type=['docx'])

    if uploaded_file:
        template_path =
f"temp_{uploaded_file.name}"
        with open(template_path, "wb") as
f:

f.write(uploaded_file.getvalue())
        selected_template = template_path
st.success("Шаблон успешно
загружен!")

        # Generate Document button (full width)
        if st.button("Сгенерировать Документ",
use_container_width=True):
            with st.spinner("Генерация
документа..."):
                try:
                    # Send request with ontology
data
                    response = requests.post(
                        f"{API_URL}/render_doc",

json=st.session_state.ontology
                    )

                    if response.status_code ==
200:
                        doc_path =
response.json()["document_path"]
                        st.success("Документ
успешно сгенерирован!")

                        # Create download link
                        download_url =
f"{API_URL}/download/{os.path.basename(doc_path)}"
                        st.markdown(f'<a
href="{download_url}" target="_blank">Скачать Документ</a>',
unsafe_allow_html=True)
                else:

```

```

                                st.error(f"Ошибка:
{response.text}")
                                except Exception as e:
                                    st.error(f"Ошибка: {str(e)}")
                                else:
                                    st.warning("Нет доступных шаблонов.
Пожалуйста, загрузите шаблон.")
                                except Exception as e:
                                    st.error(f"Ошибка: {str(e)}")
                                else:
                                    st.info("Пожалуйста, извлеките онтологию из текста во
вкладке Редактор Онтологии.")

if __name__ == "__main__":
    main()

```

ПРИЛОЖЕНИЕ Б

(обязательное)

Программные средства генерации документов на основе знаний о предметной
области

Презентация

Министерство образования и науки Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
Политехнический институт
Кафедра «Вычислительная техника»
Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ НА ТЕМУ:
**ПРОГРАММНЫЕ СРЕДСТВА ГЕНЕРАЦИИ ДОКУМЕНТОВ НА
ОСНОВЕ ЗНАНИЙ О ПРЕДМЕТНОЙ ОБЛАСТИ**

выполнил: студент гр. 23ВВИм1 Иванов А. О.
руководитель: доцент кафедры МОиПЭВМ Гурьянов Л. В.

Пенза, 2025



Актуальность темы работы

В условиях активного внедрения цифровых платформ и модульных образовательных систем возрастает потребность в автоматизации процессов проектирования и генерации документов. Применение интеллектуальных технологий в данной области способствует снижению трудозатрат, повышению точности, согласованности и прозрачности разрабатываемой документации, а также улучшению управляемости образовательной среды.

Данная работа может быть полезна преподавателям, методистам, разработчикам образовательных систем и специалистам по качеству образования для автоматизации создания и анализа структурированных документов.

Анализ требований

От разрабатываемого приложения ожидаются следующие характеристики и функции:

- система должна автоматически формировать онтологию на основе входного текстового описания предметной области,
- пользователь должен иметь возможность просматривать и редактировать автоматически сгенерированную онтологию,
- приложение должно предоставлять возможность выбора из перечня заранее загруженных шаблонов документов или загрузки собственного шаблона,
- сгенерированный документ должен экспортироваться в формате DOCX и быть пригодным для дальнейшего редактирования стандартными текстовыми редакторами,
- интерфейс должен быть понятен пользователям без глубоких технических знаний и обеспечивать интуитивно понятное взаимодействие.

Методы и средства решения поставленной задачи

Для извлечения онтологии предполагается использовать большую языковую модель (БЯМ). Для того чтобы адаптировать уже существующие модели под работу с языками описания онтологий, такими как RDF, будет использоваться процесс, называемый файнтюнингом (finetuning).

Для реализации серверной части будет использоваться язык Python и библиотека FastAPI (включая интеграцию с Pydantic для определения схем данных запросов/ответов), что обеспечивает удобное определение REST-эндпоинтов и автоматическую генерацию документации API.

На стороне фронтенда Streamlit позволяет быстро создать интерактивный веб-интерфейс без необходимости ручной разработки HTML/JS, что ускоряет прототипирование.

Техническое задание

Цель работы: Повышение эффективности и качества генерации документов посредством автоматизированного создания документов на основе формализованных знаний о предметной области

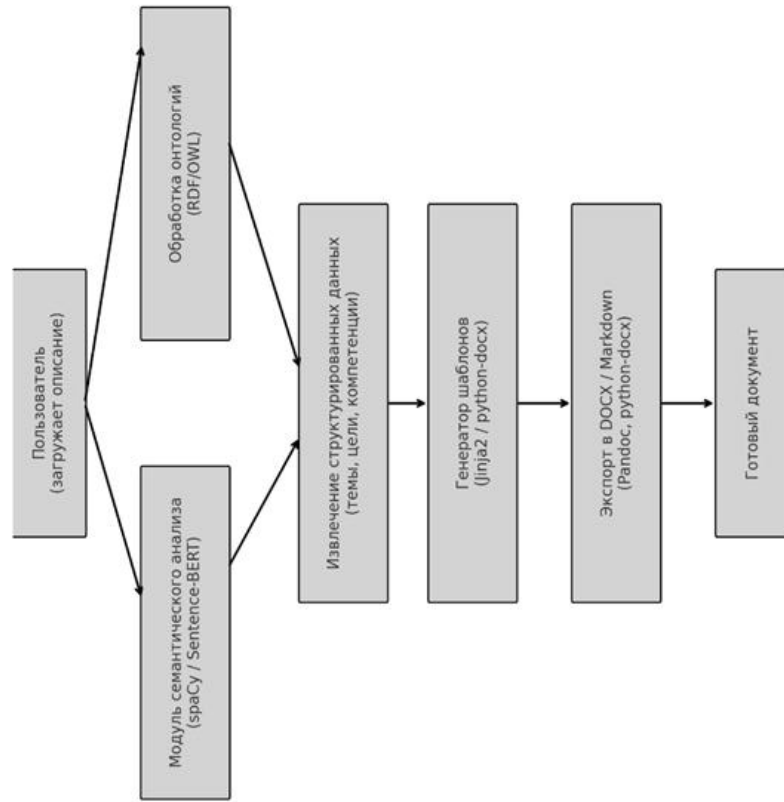
Основные функции разрабатываемой программы:

- формирование онтологии на основе входного текстового описания предметной области,
- возможность просматривать и редактировать автоматически сгенерированную онтологию,
- возможность выбора из перечня заранее загруженных шаблонов документов или загрузки собственного шаблона,
- сгенерированный документ должен экспортироваться в формате DOCX
- интерфейс должен быть понятен пользователям без глубоких технических знаний

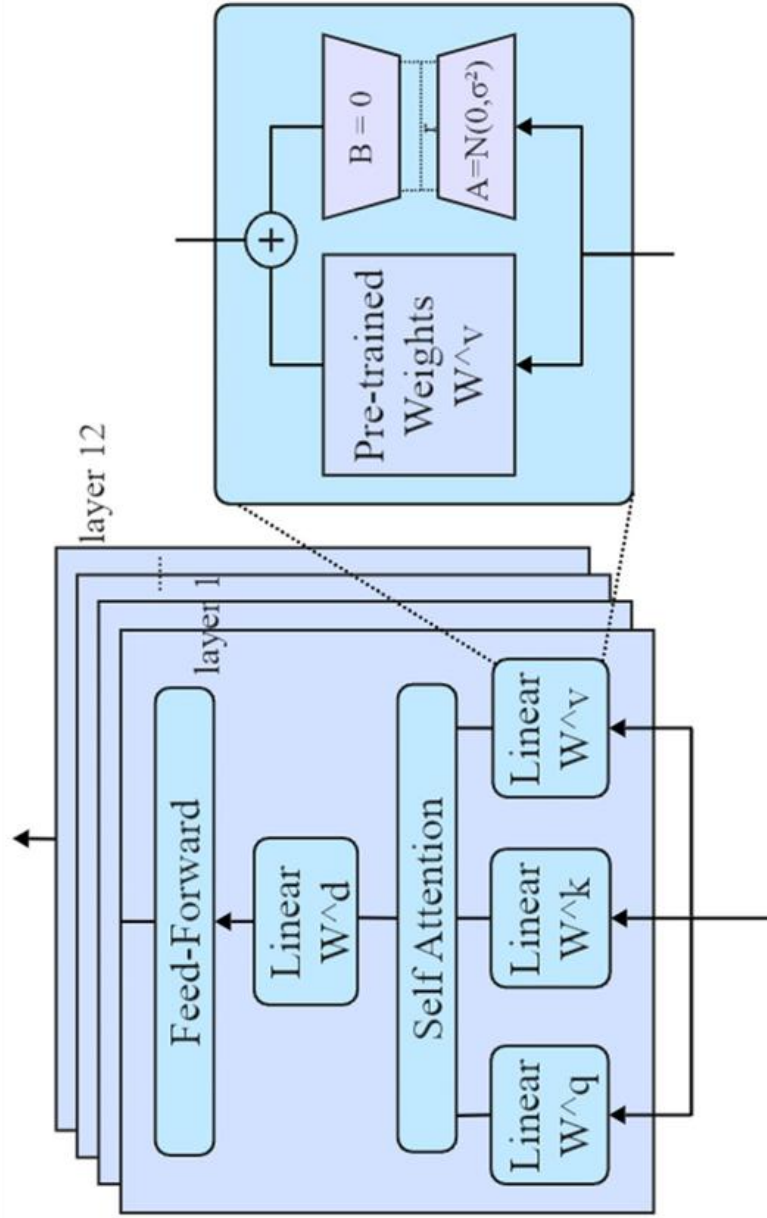
Сравнение аналогов

Система	Тип модели	Поддержка онтологий / RDF	Гибкость настройки	Генерация документов
OntoSyllabus	OWL-онтология	Да (OWL, RDF)	Высокая (расширяемая онтология)	Автоматическая генерация учебного плана
CCSO	RDF/OWL-онтология	Да (полная поддержка)	Высокая (open-source, расширяемая)	Аннотирование, экспорт учебных курсов
OBC-ONTO	OWL-онтология	Да (OWL, RDF)	Средняя – задана рамочно по OBE-принципам	Автоматические проверки согласованности, поддержка reviewers
Алекс-ВУЗ	Реляционная модель	Нет	Средняя (фиксированные шаблоны)	Шаблонная генерация программ и планов
1С:Университет	Реляционная модель	Нет	Высокая (через настройку 1С-конфигураций)	Шаблонная отчётность, планы, приказы

Проектирование программных средств



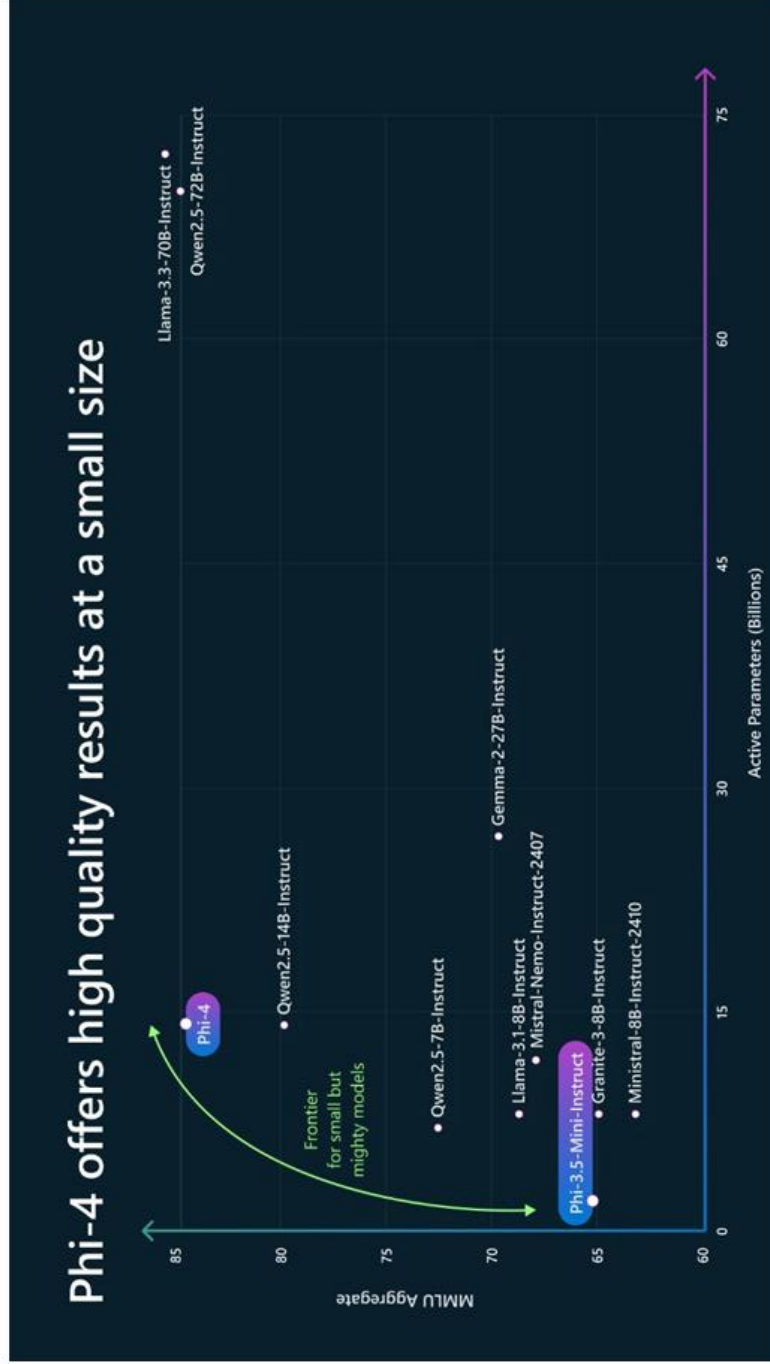
Low-Rank Adaptation



Используемые технологии



Используемая модель



Пример данных для дообучения

```
1 {  
2   "input": "Рассмотрим предметную область, связанную с разработкой и функционированием интернет-магазина  
по продаже книг. Интернет-магазин предоставляет пользователям ...",  
3   "output": {  
4     "concepts": [  
5       {  
6         "id": "C1",  
7         "label": "Интернет-магазин книг",  
8         "description": "Платформа для онлайн-продажи книг."  
9       },  
10      ...  
11    ],  
12    "relations": [  
13      {  
14        "from": "C1",  
15        "to": "C5",  
16        "type": "Содержит"  
17      },  
18      ...  
19    ]  
20  }  
21 }
```

Дообучение методом SFT

```
1 dataset_path = "/content/ontology_train_dataset.jsonl"
2 dataset = load_dataset("json", data_files={"train": dataset_path})["train"]
3
4 def format_ontology(example):
5     return {
6         "text": f"<|system|>\nВы эксперт по предметным областям.\n<|user|>\n{example['input']}\n<|assistant|>\n{example['output']}"
7     }
8
9 dataset = dataset.map(format_ontology)
```

Процесс обучения

```
==((====))== Unsloth - 2x faster free finetuning | Num GPUs used = 1
\\  /| Num examples = 250 | Num Epochs = 3 | Total steps = 96
0^0/ \^/ \ Batch size per device = 1 | Gradient accumulation steps = 8
\  / \^/ \ Data Parallel GPUs = 1 | Total batch size (1 x 8 x 1) = 8
"-_____" Trainable parameters = 8,912,896/2,234,321,920 (0.40% trained)
wandb: WARNING The 'run_name' is currently set to the same value as 'TrainingArguments.output_dir'.
wandb: Currently logged in as: dr3ids (dr3ids-myself-germany) to https://api.wandb.ai. Use `wandb l`
Tracking run with wandb version 0.19.11
Run data is saved locally in /content/wandb/run-20250615_201443-s49vkh05
Syncing run phi4-finetuned to Weights & Biases (docs)
View project at https://wandb.ai/dr3ids-myself-germany/huggingface
View run at https://wandb.ai/dr3ids-myself-germany/huggingface/runs/s49vkh05
`use_cache=True` is incompatible with gradient checkpointing. Setting `use_cache=False`.
Unsloth: Will smartly offload gradients to save VRAM!
```

96/96 22:14, Epoch 3/3]

Step Training Loss

10	0.766900
20	0.556000
30	0.491800
40	0.426400
50	0.408400
60	0.412500
70	0.380300
80	0.373600
90	0.368200

```
TrainOutput(global_step=96, training_loss=0.45857539772987366, metrics={'train_runtime': 1384.9136,
'train_samples_per_second': 0.542, 'train_steps_per_second': 0.069, 'total_flos':
1.1965475781568512e+16, 'train_loss': 0.45857539772987366})
```

Результат обучения

```
INFO:llm_infer:Sending prompt to LLM...
INFO:llm_infer:Prompt length: 2699 characters
llama_perf_context_print: load time = 98587.30 ms / 614 tokens ( 169.56 ms per token, 6.23 tokens per second)
llama_perf_context_print: prompt eval time = 98585.33 ms / 342 runs ( 465.33 ms per token, 2.15 tokens per second)
llama_perf_context_print: eval time = 159141.43 ms / 956 tokens
llama_perf_context_print: total time = 258768.99 ms /
INFO:llm_infer:
=====
INFO:llm_infer:RAW LLM OUTPUT START
INFO:llm_infer:=====
INFO:llm_infer:{
  "concepts": [
    {"id": "C1", "label": "Архитектура трансформеров", "description": "Современная архитектура трансформеров, используемая в качестве основы для генерации документов."},
    {"id": "C2", "label": "RFT", "description": "Методы параметрически эффективного дообучения, включающие адаптеры, промт-обучение, LoRA и QLoRA."},
    {"id": "C3", "label": "Автоматизированная система генерации документов", "description": "Система, состоящая из веб-приложения, серверной части и нейросетевого модуля."},
    {"id": "C4", "label": "Dataset", "description": "Специализированный набор данных, предназначенный для дообучения нейросетевого модуля."},
    {"id": "C5", "label": "phi-4-mini", "description": "Модель, использованная для дообучения с помощью QLoRA."}
  ],
  "relations": [
    {"from": "C1", "to": "C2", "type": "Используется"},
    {"from": "C3", "to": "C1", "type": "Основана на"},
    {"from": "C3", "to": "C2", "type": "Включает"},
    {"from": "C4", "to": "C5", "type": "Используется для"}
  ]
}
INFO:llm_infer:=====
INFO:llm_infer:RAW LLM OUTPUT END
INFO:llm_infer:=====
INFO:llm_infer:Successfully parsed ontology:
INFO:llm_infer:Concepts: 5
INFO:llm_infer:Relations: 4
```

Реализация пользовательского интерфейса

Генератор Документации

Редактор Онтологий Генерация Документа

Ввод Текста

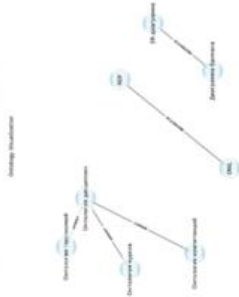
Введите или вставьте текст:

Семантические модели знаний позволяют формализовать содержание курса, структуру образовательной программы и связи с внешними нормативами. Ключевыми стандартами семантического представления являются RDF и OWL. RDF описывает знания в виде графа триплетов, где каждое утверждение имеет форму субъект-предикат-объект. OWL расширяет RDF возможностью задавать классы, свойства, аксиомы, ограничения и логические выводы. В контексте образования RDF/OWL используется для построения онтологий:
– онтологий дисциплин / тематика, ключевые понятия

Извлечь Онтологию

Онтология успешно извлечена!

Визуализация Онтологии



Редактирование Онтологии

Концепты:

Концепт Онтология дисциплин

Название: Онтология дисциплин

Описание: Онтология, описывающая предметную область, ключевые понятия и цели обучения.

Удалить Концепт

Связи:

Связь 1

От: Онтология дисц...

К: Онтология комп...

Тип: имеет

Удалить Связь

Концепт Онтология компетенций

Название: Онтология компетенций

Описание: Онтология, описывающая универсальные, профессиональные и e-CE компетенции.

Удалить Концепт

Связь 2

От: Онтология дисц...

К: Онтология курсов

Тип: имеет

Удалить Связь

Концепт Онтология курсов

Название: Онтология курсов

Описание:

Удалить Концепт

Связь 3

От: Онтология дисц...

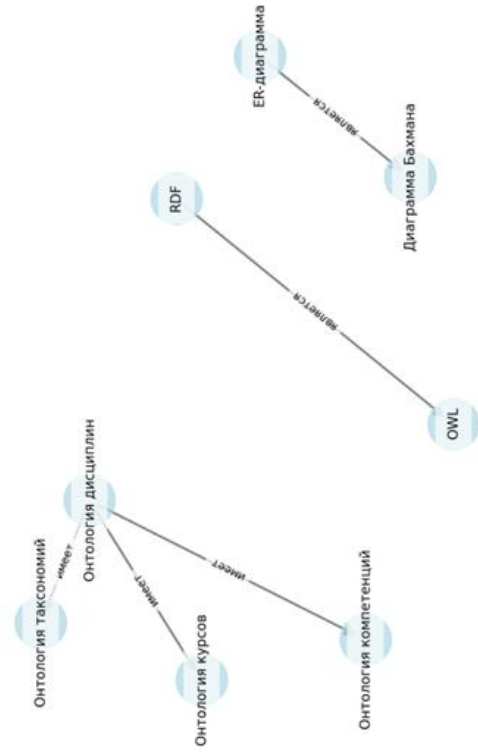
К: Онтология таксо...

Тип: имеет

Удалить Связь

Реализация пользовательского интерфейса

Ontology Visualization



Генератор Документации

Ресурсы: [Онтология](#) [Пользовательский интерфейс](#)

Генерация Документа

Выбор шаблона: шаблоны: **Интерфейс**

Предпросмотр шаблона

Загрузка шаблона

Перменные шаблона

Детали шаблона

Загрузка шаблона

Результат генерации

Domain Documentation	
Document Information	
Generated on:	2025-06-23 03:12:11
Total Concepts:	8
Total Relations:	5
Domain Concepts	
Онтология дисциплин	
ID:	C1
Description	Онтология, описывающая предметную область, ключевые понятия и цели обучения.
Related Concepts	
	• Онтология компетенций (имеет)
	• Онтология курсов (имеет)
	• Онтология таксономий (имеет)

Заключение

Полученные результаты подтверждают перспективность применения PEFT-методов и, в частности, подхода QLoRA для решения практических задач генерации документов на основе структурированных знаний. Реализованная система способна существенно ускорить и автоматизировать процессы подготовки специализированных документов, таких как учебные программы, аннотации и отчётная документация, обеспечивая при этом высокую точность и согласованность генерируемых материалов. Разработанный подход и проведённые эксперименты могут быть полезны при создании аналогичных систем и в других предметных областях.

Доклад окончен.
СПАСИБО ЗА ВНИМАНИЕ!