# Часть 2: продвинутый промптинг и работа с кодом

## От переводчика

Представляю вашему вниманию перевод второй части статьи «Prompt Engineering» (Промпт-инжиниринг) авторства Lee Boonstra — Software Engineer Tech Lead, Office of the CTO в Google. Эта публикация продолжает цикл переводов, посвященных методам эффективного взаимодействия с большими языковыми моделями.

В первой части мы познакомились с основами промпт-инжиниринга, базовыми настройками БЯМ и ключевыми техниками промптинга. Вторая часть посвящена более продвинутым и специализированным методам, которые существенно расширяют возможности работы с языковыми моделями и позволяют решать более сложные задачи. В следующей, третьей части, мы рассмотрим лучшие практики и рекомендации, которые помогут получать от моделей еще более точные и надежные ответы.

Как и прежде, хочу отметить, что оригинальная публикация фокусируется в основном на моделях Gemini и сервисе Vertex AI от Google, однако описанные техники и рекомендации универсальны и применимы практически ко всем современным большим языковым моделям (GPT, Claude, Llama и др.).

**Обратите внимание**: Это неофициальный перевод, выполненный энтузиастом с целью сделать ценную информацию доступной русскоязычному сообществу. Результаты работы моделей с переведенными примерами промптов могут отличаться от тех, что приведены в оригинальной статье.

Некоторые технические термины оставлены без перевода или транслитерированы, поскольку они уже вошли в профессиональный жаргон русскоязычных специалистов в сфере ИИ.

Приятного чтения!

---

# Промптинг с отступлением

Промптинг с отступлением (Step-back)[8] – это техника улучшения производительности, при которой БЯМ сначала предлагается рассмотреть общий вопрос, связанный с конкретной задачей, а затем результат этого общего вопроса подается в последующий промпт для конкретной задачи. Этот "шаг назад" позволяет БЯМ активировать соответствующие фоновые знания и процессы рассуждения перед попыткой решить конкретную проблему.

Рассматривая более широкие и фундаментальные принципы, БЯМ могут генерировать более точные и содержательные ответы. Промптинг с отступлением стимулирует БЯМ мыслить критически и применять свои знания новыми и творческими способами. Он изменяет окончательный промпт, выполняющий задачу, используя больше знаний в параметрах БЯМ, чем иначе могло бы быть задействовано при прямом промптинге модели.

Это может помочь смягчить предвзятость в ответах БЯМ, фокусируясь на общих принципах вместо конкретных деталей.

Давайте рассмотрим эти примеры, чтобы понять, как промптинг с отступлением может улучшить результаты. Сначала рассмотрим традиционный промпт (Таблица 8), прежде чем сравнить его с промптом с отступлением (Таблица 9):

| | |
|---|---|
| Цель | Написать сюжетную линию для уровня шутера от первого лица. |
| Модель | gemini-pro |
| Температура | 1 |
| Лимит токенов | 1024 |
| Top-K | 40 |
| Top-P | 0.8 |
| Промпт | Напиши сюжетную линию из одного абзаца для нового уровня шутера от первого лица, который будет сложным и увлекательным. |
| Вывод | Уровень начинается с того, что отряд игрока попадает в засаду в густонаселенном городском районе, устроенную безжалостной вражеской фракцией. Игроки должны пробиваться через лабиринт задних аллей и ветхих зданий, используя скрытность и точность для уничтожения вражеских сил и избегания обнаружения. Игрок должен использовать окружающую среду в своих интересах, устанавливая ловушки, используя укрытия и предпринимая рассчитанные ходы для выживания. По пути они обнаруживают разведданные, указывающие на то, что враг планирует разрушительную атаку, и игрок должен наперегонки собрать доказательства и передать их в свой штаб, прежде чем враг сможет осуществить свои планы. |

*Таблица 8. Традиционный промпт перед сравнением с промптом с отступлением*

Когда вы установите температуру на 1, вы можете получить всевозможные творческие варианты для сюжетной линии, но это также довольно случайно и банально. Так что давайте сделаем шаг назад:

| | |
|---|---|
| Промпт | Основываясь на популярных шутерах от первого лица, назови 5 вымышленных ключевых локаций, которые способствуют созданию сложной и увлекательной сюжетной линии уровня в шутере от первого лица? |
| Вывод | 1. **Заброшенная военная база**: Обширный, постапокалиптический военный комплекс, кишащий мутировавшими солдатами и роботами-изгоями, идеально подходящий для сложного огнестрельного боя.<br>2. **Киберпанк-город**: Неоново-освещенная, футуристическая городская среда с возвышающимися небоскребами и тесными переулками, с кибернетически усовершенствованными врагами и механиками взлома.<br>3. **Инопланетный космический корабль**: Огромный инопланетный корабль, застрявший на Земле, с жуткими коридорами, секциями с нулевой гравитацией и внеземными существами для встречи.<br>4. **Город, наводненный зомби**: Опустошенный город, заполненный ордами агрессивных зомби, с интенсивными боями на ближней дистанции и решением головоломок для поиска безопасного прохода.<br>5. **Подводный исследовательский комплекс**: Глубоководная лаборатория, заполненная водой, населенная мутировавшими водными существами и требующая навыков скрытности и подводного исследования для выживания. |

*Таблица 9. Пример промптинга для самосогласованности*

Да, эти темы похоже подходят для шутера от первого лица. Давайте вернемся к исходному промпту, но на этот раз включим ответ с шагом назад в качестве контекста и посмотрим, что будет возвращено.

| | |
|---|---|
| Промпт | Контекст: 5 увлекательных тем для шутера от первого лица:<br>1. **Заброшенная военная база**: Обширный, постапокалиптический военный комплекс, кишащий мутировавшими солдатами и роботами-изгоями, идеально подходящий для сложного огнестрельного боя.<br>2. **Киберпанк-город**: Неоново-освещенная, футуристическая городская среда с возвышающимися небоскребами и тесными переулками, с кибернетически усовершенствованными врагами и механиками взлома.<br>3. **Инопланетный космический корабль**: Огромный инопланетный корабль, застрявший на Земле, с жуткими коридорами, секциями с нулевой гравитацией и внеземными существами для встречи.<br>4. **Город, наводненный зомби**: Опустошенный город, заполненный ордами агрессивных зомби, с интенсивными боями на ближней дистанции и решением головоломок для поиска безопасного прохода.<br>5. **Подводный исследовательский комплекс**: Глубоководная лаборатория, заполненная водой, населенная мутировавшими водными существами и требующая навыков скрытности и подводного исследования для выживания.<br>Выбери одну из тем и напиши сюжетную линию из одного абзаца для нового уровня шутера от первого лица, который будет сложным и увлекательным. |
| Вывод | В сердце мрачной бездны лежит разрушенный подводный исследовательский комплекс, стоящий как свидетельство человеческих амбиций и их катастрофических последствий. Окутанный тьмой, пульсирующий гулом неисправного оборудования и преследуемый эхом утонувших душ, комплекс теперь стал рассадником кошмара, рожденного из глубин моря. Игрок, элитный боец морской пехоты, оснащенный продвинутым водолазным снаряжением и экспериментальным оружием, отправлен в этот водный мир ужаса. Вооруженный стойкостью и находчивостью, он должен ориентироваться в коварных коридорах, решать загадочные головоломки и противостоять жутким морским чудовищам, скрывающимся в глубинах. От борьбы с давящим давлением глубокого моря до перехитривания хищных водных хищников, каждый момент в этом неизведанном подводном мире испытывает пределы человеческой выносливости и мужества. |

*Таблица 10. Пример промптинга для самосогласованности*

Вот это выглядит как интересная видеоигра! Используя техники промптинга с отступлением, вы можете повысить точность своих промптов.

# Цепочка рассуждений (CoT)

Промптинг с цепочкой рассуждений (Chain of Thought, CoT)[9] — это техника улучшения способностей рассуждения БЯМ путем генерации **промежуточных шагов рассуждения**. Это помогает БЯМ генерировать более точные ответы. Вы можете комбинировать его с промптингом с несколькими примерами для получения лучших результатов на более сложных задачах, требующих рассуждения перед ответом, поскольку это сложно с промптингом с нулевым примером в цепочке рассуждений.

У CoT много преимуществ. Во-первых, он требует мало усилий, но при этом очень эффективен и хорошо работает с готовыми БЯМ (без необходимости в дополнительной настройке). Вы также получаете интерпретируемость с промптингом CoT, поскольку можете учиться на ответах БЯМ и видеть шаги рассуждения, которые были выполнены. Если есть сбой, вы сможете его идентифицировать. Цепочка рассуждений, похоже, улучшает устойчивость при переходе между разными версиями БЯМ. Это означает, что производительность вашего промпта должна меньше колебаться между разными БЯМ, чем если бы ваш промпт не использовал цепочки рассуждений. Конечно, есть и недостатки, но они довольно интуитивно понятны.

Ответ БЯМ включает рассуждения в цепочке размышлений, что означает больше выходных токенов, а это, в свою очередь, означает, что предсказания стоят дороже и занимают больше времени.

Чтобы объяснить следующий пример в Таблице 11, давайте сначала попробуем создать промпт, который не использует промптинг CoT, чтобы продемонстрировать недостатки большой языковой модели.

| | |
|---|---|
| Промпт | Когда мне было 3 года, мой партнер был в 3 раза старше меня. Сейчас мне 20 лет. Сколько лет моему партнеру? |
| Вывод | 63 года |

*Таблица 11. Пример промпта, пытающегося решить математическую задачу*

Ой! Это явно неправильный ответ. Действительно, БЯМ часто испытывают трудности с математическими задачами и могут давать неверные ответы — даже для такой простой задачи, как умножение двух чисел. Это связано с тем, что они обучены на больших объемах текста, а математика может требовать другого подхода. Итак, давайте посмотрим, улучшат ли промежуточные шаги рассуждения результат.

| | |
|---|---|
| Промпт | Когда мне было 3 года, мой партнер был в 3 раза старше меня. Сейчас мне 20 лет. Сколько лет моему партнеру? Давай размышлять пошагово. |

| | |
|---|---|
| Вывод | 1. Предположим, мой текущий возраст - "x" лет.<br>2. Когда мне было 3 года, мой возраст составлял 3 года.<br>3. Возраст моего партнера в то время был в 3 раза больше моего, что означает, что возраст моего партнера составлял 3 * 3 = 9 лет.<br>4. Сейчас мне 20 лет, что означает, что мой возраст увеличился на 20 - 3 = 17 лет с тех пор, как мне было 3 года.<br>5. Поскольку возраст моего партнера был 9 лет, когда мне было 3 года, и мой возраст увеличился на 17 лет, возраст моего партнера также должен был увеличиться на 17 лет.<br>6. Следовательно, текущий возраст моего партнера составляет 9 + 17 = 26 лет.<br>Итак, моему партнеру 26 лет. |

Таблица 12. Пример промптинга с цепочкой рассуждений

Отлично, теперь окончательный ответ верен. Это произошло потому, что мы четко проинструктировали БЯМ объяснить каждый шаг, вместо того чтобы просто вернуть ответ. Интересно видеть, что модель складывает 17 увеличенных лет. В моей голове я бы взял разницу в годах между моим партнером и мной и добавил их. (20+(9-3)). Давайте поможем модели мыслить немного больше как я.

Таблица 12 — это пример промптинга с нулевым примером в цепочке рассуждений. Промптинг с цепочкой рассуждений может быть очень мощным в сочетании с одним или несколькими примерами, как видно в Таблице 13:

| | |
|---|---|
| Промпт | Q: Когда моему брату было 2 года, я был в два раза старше него. Сейчас мне 40 лет. Сколько лет моему брату? Давай размышлять пошагово.<br>A: Когда моему брату было 2 года, мне было 2 * 2 = 4 года. Это разница в возрасте в 2 года, и я старше. Сейчас мне 40 лет, поэтому моему брату 40 - 2 = 38 лет. Ответ — 38.<br>Q: Когда мне было 3 года, мой партнер был в 3 раза старше меня. Сейчас мне 20 лет. Сколько лет моему партнеру? Давай размышлять пошагово.<br>A: |
| Вывод | Когда мне было 3 года, моему партнеру было 3 * 3 = 9 лет. Это разница в возрасте в 6 лет, и мой партнер старше. Сейчас мне 20 лет, поэтому моему партнеру 20 + 6 = 26 лет. Ответ — 26. |

*Таблица 13. Пример промптинга с цепочкой рассуждений и одним примером*

Цепочка рассуждений может быть полезна для различных сценариев использования. Например, для генерации кода, для разбиения запроса на несколько шагов и сопоставления этих шагов с конкретными строками кода. Или для создания синтетических данных, когда у

вас есть какая-то затравка, например: "*Продукт называется XYZ, напишите описание, направляя модель через предположения, которые вы бы сделали на основе данного названия продукта*". В целом, любая задача, которую можно решить путем "обсуждения", является хорошим кандидатом для цепочки рассуждений. Если вы можете объяснить шаги для решения проблемы, попробуйте цепочку рассуждений.

Пожалуйста, обратитесь к блокноту[10], размещенному в репозитории GitHub GoogleCloudPlatform, где более подробно рассматривается промптинг CoT.

# Самосогласованность

Хотя большие языковые модели продемонстрировали впечатляющие успехи в различных задачах обработки естественного языка, их способность к рассуждению часто рассматривается как ограничение, которое нельзя преодолеть только увеличением размера модели. Как мы узнали в предыдущем разделе о промптинге с цепочкой рассуждений, модель можно побудить генерировать шаги рассуждения, подобно тому, как человек решает проблему. Однако CoT использует простую стратегию "жадного декодирования", что ограничивает её эффективность. Самосогласованность[11] объединяет сэмплирование и голосование большинством для генерации разнообразных путей рассуждения и выбора наиболее согласованного ответа. Это улучшает точность и согласованность ответов, генерируемых БЯМ.

Самосогласованность даёт псевдо-вероятностную оценку правильности ответа, но, очевидно, имеет высокую стоимость.

Она следует следующим шагам:

1. Генерация разнообразных путей рассуждения: БЯМ предоставляется один и тот же промпт несколько раз. Высокая настройка температуры стимулирует модель генерировать различные пути рассуждения и перспективы на проблему.
2. Извлечение ответа из каждого сгенерированного ответа.
3. Выбор наиболее распространенного ответа.

Давайте рассмотрим пример системы классификации электронной почты, которая классифицирует письмо как ВАЖНОЕ или НЕВАЖНОЕ. Промпт с нулевым примером цепочки рассуждений будет отправлен БЯМ несколько раз, чтобы увидеть, различаются ли ответы после каждой отправки. Обратите внимание на дружеский тон, выбор слов и сарказм, использованные в письме. Всё это может запутать БЯМ.

| | |
|---|---|
| Промпт | EMAIL:<br>Привет,<br>Я видел, что вы используете Wordpress для вашего сайта. Отличная система управления контентом с открытым исходным кодом. Я тоже использовал её в прошлом. Она поставляется с множеством отличных пользовательских плагинов. И её довольно легко настроить.<br>Я заметил ошибку в контактной форме, которая возникает при выборе поля имени. Посмотрите прикрепленный скриншот, где я ввожу текст в поле имени. Обратите внимание на окно предупреждения JavaScript, которое я вызвал.<br>Но в остальном это отличный сайт. Мне нравится его читать. Не стесняйтесь оставить ошибку на сайте, потому что это даёт мне больше интересных вещей для чтения.<br>С уважением,<br>Гарри Хакер.<br>Классифицируй вышеуказанное письмо как ВАЖНОЕ или НЕВАЖНОЕ. Размышляй пошагово и объясни свой выбор. |
| Попытка 1 | **Шаг 1: Определить цель письма.** Цель письма - проинформировать получателя об ошибке в контактной форме на их веб-сайте. Отправитель также упоминает, что ему нравится читать веб-сайт и что они оставляют ошибку на месте на время.<br>**Шаг 2: Оценить потенциальное влияние ошибки.** Ошибка в контактной форме может позволить атакующему выполнить произвольный JavaScript-код на веб-сайте. Это может привести к тому, что атакующий сможет получить контроль над веб-сайтом, украсть конфиденциальную информацию или обезобразить веб-сайт.<br>**Шаг 3: Рассмотреть убедительность отправителя.** Отправитель письма утверждает, что является хакером по имени Гарри. Однако нет возможности проверить личность отправителя. Возможно, отправитель является законным хакером, который пытается помочь получателю, но также возможно, что отправитель является злоумышленником, который пытается обмануть получателя, чтобы тот кликнул на вредоносную ссылку или загрузил вредоносный файл.<br>**Заключение: ВАЖНОЕ** Исходя из потенциального влияния ошибки и убедительности отправителя, письмо следует классифицировать как **ВАЖНОЕ**. Получатель должен немедленно принять меры для исправления ошибки и защиты своего веб-сайта от атак. |

| | |
|---|---|
| Попытка 2 | **Объяснение**:<br><br>1. **Отсутствие срочности**: Письмо не передает никакого ощущения срочности или чувствительности ко времени. Оно начинается с обычного приветствия и представляет информацию в будничном тоне.<br>2. **Некритичное сообщение об ошибке**: Сообщаемая ошибка в контактной форме описывается как интересная, но не критичная. Отправитель письма даже предлагает, чтобы владелец веб-сайта оставил ошибку на месте.<br>3. **Отсутствие личного влияния**: Письмо не указывает на какое-либо прямое или личное влияние на владельца веб-сайта. Оно просто упоминает, что отправителю нравится читать веб-сайт.<br>4. **Отсутствие запроса на действие**: Письмо явно не запрашивает каких-либо действий от владельца веб-сайта. Оно не просит исправить ошибку или предоставить какие-либо конкретные предложения по улучшению.<br>5. **Намерение отправителя**: Намерение отправителя, кажется, больше сфокусировано на обмене наблюдением и выражении интереса к веб-сайту, чем на том, чтобы поднять серьезную проблему или запросить помощь.<br>**Заключение: НЕВАЖНОЕ** |
| Попытка 3 | **Шаг 1: Определить цель письма.** Цель письма - проинформировать получателя об ошибке в контактной форме на их веб-сайте. Отправитель также упоминает, что ему нравится читать веб-сайт и что они оставляют ошибку на месте на время.<br>**Шаг 2: Оценить потенциальное влияние ошибки.** Ошибка в контактной форме может позволить атакующему выполнить произвольный JavaScript-код на веб-сайте. Это может привести к тому, что атакующий сможет получить контроль над веб-сайтом, украсть конфиденциальную информацию или обезобразить веб-сайт.<br>**Шаг 3: Рассмотреть убедительность отправителя.** Отправитель письма утверждает, что является хакером по имени Гарри. Однако нет возможности проверить личность отправителя. Возможно, отправитель является законным хакером, который пытается помочь получателю, но также возможно, что отправитель является злоумышленником, который пытается использовать ошибку в своих целях.<br>**Заключение:** Исходя из вышеперечисленных факторов, письмо следует классифицировать как **ВАЖНОЕ**. Ошибка в контактной форме является серьезным риском безопасности, и личность отправителя не установлена. Получатель должен немедленно принять меры для исправления ошибки и расследовать утверждения отправителя. |

*Таблица 14. Пример промптинга для самосогласованности*

Вы можете использовать приведенный выше промпт и попытаться увидеть, возвращает ли он согласованную классификацию. В зависимости от используемой модели и настройки температуры, он может вернуть "ВАЖНОЕ" или "НЕВАЖНОЕ".

Генерируя несколько цепочек рассуждений и выбирая наиболее часто встречающийся ответ ("ВАЖНОЕ"), мы можем получить более стабильно правильный ответ от БЯМ.

Этот пример показывает, как промптинг с самосогласованностью может быть использован для улучшения точности ответа БЯМ путем рассмотрения нескольких перспектив и выбора наиболее согласованного ответа.

## Дерево рассуждений (ToT)

Теперь, когда мы знакомы с цепочкой рассуждений и самосогласованностью, давайте рассмотрим дерево рассуждений (Tree of Thoughts, ToT)[12]. Оно обобщает концепцию промптинга с цепочкой рассуждений, поскольку позволяет БЯМ исследовать несколько различных путей рассуждения одновременно, а не просто следовать одной линейной цепочке мыслей. Это изображено на Рисунке 1.

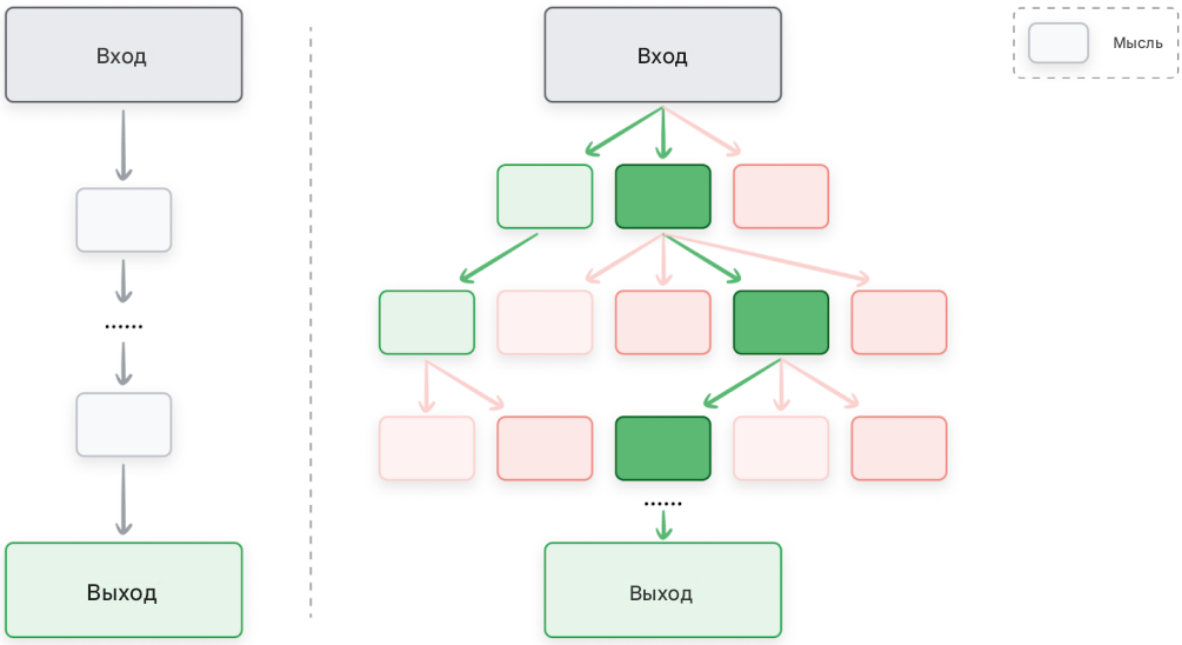


*Рисунок 1. Визуализация промптинга с цепочкой рассуждений слева и дерева рассуждений справа*

Этот подход делает ToT особенно хорошо подходящим для сложных задач, требующих исследования. Он работает, поддерживая дерево мыслей, где каждая мысль представляет собой связную языковую последовательность, которая служит промежуточным шагом к решению проблемы. Затем модель может исследовать различные пути рассуждения, разветвляясь от разных узлов дерева.

Есть отличный блокнот, который более подробно показывает дерево рассуждений (ToT), основанный на статье "Large Language Model Guided Tree-of-Thought".[9]

# ReAct (рассуждения и действия)

Рассуждения и действия (ReAct)[13] - это парадигма, позволяющий БЯМ решать сложные задачи, используя рассуждения на естественном языке в сочетании с внешними инструментами (поиск, интерпретатор кода и т.д.), позволяющими БЯМ выполнять определенные действия, такие как взаимодействие с внешними API для получения информации, что является первым шагом к моделированию агентов.

ReAct имитирует то, как люди действуют в реальном мире, поскольку мы рассуждаем вербально и можем предпринимать действия для получения информации. ReAct показывает хорошие результаты по сравнению с другими подходами к промпт-инжинирингу в различных областях.

Промптинг ReAct работает, объединяя рассуждения и действия в петлю мысль-действие. БЯМ сначала рассуждает о проблеме и генерирует план действий. Затем он выполняет действия в плане и наблюдает результаты. Затем БЯМ использует наблюдения для обновления своих рассуждений и генерации нового плана действий. Этот процесс продолжается, пока БЯМ не достигнет решения проблемы.

Чтобы увидеть это в действии, вам нужно написать некоторый код. В фрагменте кода 1 я использую фреймворк langchain для Python вместе с VertexAI (google-cloud-aiplatform) и пакетами google-search-results pip.

Для запуска этого примера вы должны создать (бесплатный) ключ SerpAPI с сайта https://serpapi.com/manage-api-key и установить переменную окружения SERPAPI_API_KEY.

Далее давайте напишем некоторый код Python, с задачей для БЯМ выяснить: *Сколько детей имеют известного отца, выступающего в группе Metallica.*

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import VertexAI

prompt = "Сколько детей у участников группы Metallica?"
llm = VertexAI(temperature=0.1)
tools = load_tools(["serpapi"], llm=llm)
agent = initialize_agent(tools, llm,
agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
agent.run(prompt)
```

*Фрагмент кода 1. Создание агента ReAct с LangChain и VertexAI*

Фрагмент кода 2 показывает результат. Обратите внимание, что ReAct делает цепочку из пяти поисков. Фактически, БЯМ просматривает результаты поиска Google, чтобы выяснить имена участников группы. Затем она перечисляет результаты как наблюдения и связывает мысль для следующего поиска.

```
> Начинаем новую цепочку AgentExecutor...
У Metallica 4 участника.
Действие: Поиск
Ввод действия: Сколько детей у Джеймса Хэтфилда?
Наблюдение: трое детей
Размышление: 1/4 участников Metallica имеют 3 детей
Действие: Поиск
Ввод действия: Сколько детей у Ларса Ульриха?
Наблюдение: 3
Размышление: 2/4 участников Metallica имеют 6 детей
Действие: Поиск
Ввод действия: Сколько детей у Кирка Хэмметта?
Наблюдение: Хэмметт женат на своей второй жене Лани с 1998 года. У них двое сыновей, Анхель
(р. 29 сентября 2006) и Винченцо (р. 28 июня 2008).
Размышление: 3/4 участников Metallica имеют 8 детей
Действие: Поиск
Ввод действия: Сколько детей у Роберта Трухильо?
Наблюдение: 2
Размышление: 4/4 участников Metallica имеют 10 детей
Итоговый ответ: 10
```

*Фрагмент кода 2. Выполнение агента ReAct в терминале*

Фрагмент кода 2 показывает, что группа Metallica состоит из четырех участников. Затем он ищет каждого участника группы, чтобы узнать общее количество детей и суммирует общее число. В итоге возвращает общее количество детей как окончательный ответ.

Промптинг ReAct на практике требует понимания того, что вам постоянно приходится повторно отправлять предыдущие промпты/ответы (и обрезать дополнительный сгенерированный контент), а также настраивать модель с соответствующими примерами/инструкциями. Пожалуйста, обратитесь к блокноту[14], размещенному в репозитории GitHub GoogleCloudPlatform, который более подробно показывает фактические входные и выходные данные БЯМ с более сложным примером.

## Автоматический промпт-инжиниринг

К этому моменту вы, возможно, осознали, что написание промпта может быть сложным. Не было бы здорово автоматизировать этот процесс (написать промпт для написания

промптов)? Что ж, есть метод: автоматический промпт-инжиниринг (Automatic Prompt Engineering, APE). Этот метод[15] не только устраняет необходимость ввода данных человеком, но и улучшает производительность модели в различных задачах.

Вы будете запрашивать модель для генерации большего количества промптов. Оцените их, возможно, измените хорошие и повторите.

Например, вы можете использовать автоматический промпт-инжиниринг, который поможет с обучением чат-бота для интернет-магазина футболок с атрибутикой музыкальных групп. Мы хотим выяснить различные способы, которыми клиенты могли бы сформулировать свой заказ на покупку футболки с атрибутикой группы.

1. Напишите промпт, который будет генерировать варианты вывода. В этом примере я использую gemini-pro для генерации 10 инструкций. См. Таблицу 15:

| | |
|---|---|
| Промпт | У нас есть интернет-магазин футболок с музыкальной атрибутикой, и для обучения чат-бота нам нужны различные способы заказа: "Одна футболка Metallica размера S". Сгенерируй 10 вариантов с той же семантикой, но сохраняя то же значение. |
| Вывод | 1. Я хотел бы приобрести футболку Metallica размера S.<br>2. Можно заказать футболку Metallica маленького размера?<br>3. Я хотел бы купить одну футболку Metallica размера S.<br>4. Одна футболка Metallica, маленький размер, пожалуйста.<br>5. Я хочу одну футболку Metallica размера S.<br>6. Я хотел бы оформить заказ на одну футболку Metallica маленького размера.<br>7. Футболка Metallica, размер S, одна штука, пожалуйста.<br>8. Я ищу одну футболку Metallica размера S.<br>9. Можете оформить для меня одну футболку Metallica размера S?<br>10. Одна футболка Metallica маленького размера, пожалуйста. |

*Таблица 15. Использование БЯМ для генерации промптов*

2. Оцените всех кандидатов инструкций, оценивая кандидатов на основе выбранной метрики. Например, вы можете использовать BLEU (Bilingual Evaluation Understudy) или ROUGE (Recall-Oriented Understudy for Gisting Evaluation).
3. Выберите кандидата инструкции с наивысшей оценкой оценки. Этот кандидат будет окончательным промптом, который вы можете использовать в своем программном приложении или чат-боте. Вы также можете настроить выбранный промпт и снова оценить.

# Промптинг для кода

Gemini фокусируется в основном на текстовых промптах, что также включает написание промптов для возврата кода. Давайте перейдем к Vertex AI Studio и протестируем эти промпты, чтобы посмотреть некоторые примеры кода.

# Промпты для написания кода

Gemini также может быть разработчиком и помочь вам с написанием кода на любом языке программирования по вашему выбору. Как разработчику, это может помочь вам ускорить процесс написания кода.

Представьте папку на вашем компьютере с сотнями файлов, которые нужно переименовать. Переименование каждого файла заняло бы у вас много времени. Вы немного знаете Bash и могли бы написать скрипт для автоматизации этого, но это тоже может занять некоторое время. Так что давайте напишем промпт. Вы можете написать промпт в общедоступном потребительском чат-боте Gemini, или если вы больше обеспокоены конфиденциальностью, вы можете написать эти промпты в своей учетной записи Google Cloud и открыть Vertex AI Studio. Преимущество Vertex AI Studio заключается в том, что вы можете настроить температуру и т.д.

| | |
|---|---|
| Название | 1_bash_rename_files_in_folder |
| Цель | Написать промпт для создания кода на Bash для переименования файлов в папке. |
| Модель | gemini-pro |
| Температура | 0.1 |
| Top-K | N/A |
| Промпт | Напиши фрагмент кода на Bash, который запрашивает имя папки. Затем он берет содержимое папки и переименовывает все файлы внутри, добавляя приставку "draft" к имени файла. |

| | |
|---|---|
| Вывод | ```bash #!/bin/bash<br># Запрос имени папки<br>echo "Введите имя папки: "<br>read folder_name<br># Проверка существования папки<br>if [ ! -d "$folder_name" ]; then<br>echo "Папка не существует."<br>exit 1<br>fi<br># Получение списка файлов в папке<br>files=( "$folder_name"/* )<br># Переименование каждого файла, добавляя "draft" к имени файла<br>for file in "${files[@]}"; do<br>new_file_name="draft_$(basename "$file")"<br>mv "$file" "$new_file_name"<br>done<br>echo "Файлы успешно переименованы."<br>``` |

*Таблица 16. Использование Gemini для написания кода*

Это похоже на хороший код - он даже документирован! Однако, поскольку БЯМ не могут рассуждать и повторяют данные обучения, важно сначала прочитать и протестировать ваш код.

Момент, которого мы все ждали, действительно ли это работает?

Давайте сначала попробуем его с тестовой папкой, содержащей только несколько файлов, которые нужно переименовать из filename.txt в draft_filename.txt.

1. Скопируйте вывод из Таблицы 16 (без текста обертки `bash`), и вставьте его в новый файл с названием: "**rename_files.sh**".
2. Откройте окно терминала и введите: **. rename_files.sh**. Он попросит ввести имя папки, например, **test.** и нажмите enter.
3. Скрипт, похоже, работает нормально. Вы увидите сообщение: **Файлы успешно переименованы.** Когда вы посмотрите в папку test, заметите, что все файлы были идеально переименованы в **draft_filename.txt.**

Это работает!

# Промпты для объяснения кода

Как разработчик, когда вы работаете в командах, вам приходится читать чужой код. Gemini может помочь вам и с этим. Давайте возьмем вывод кода из Таблицы 16, удалим комментарии и попросим большую языковую модель объяснить, что происходит, см. Таблицу 17:

| | |
|---|---|
| Название | 1_bash_rename_files_in_folder |

| | |
|---|---|
| Цель | Написать промпт для создания кода на Bash для переименования файлов в папке. |
| Модель | gemini-pro |
| Температура | 0.1 |
| Лимит токенов | 1024 |
| Top-K | N/A |
| Top-P | 1 |
| Промпт | Объясни мне этот Bash-скрипт:<br>`#!/bin/bash`<br>`echo ""Введите имя папки: "`<br>`read folder_name`<br>`if [ ! -d "$folder_name" ]; then`<br>`echo "Папка не существует."`<br>`exit 1`<br>`fi`<br>`files=( "$folder_name"/* )`<br>`for file in "${files[@]}"; do`<br>`new_file_name="draft_$(basename "$file")"`<br>`mv "$file" "$new_file_name"`<br>`done`<br>`echo "Файлы успешно переименованы."` |
| Вывод | Этот Bash-скрипт выполняет следующие задачи:<br>1. **Пользовательский ввод**:<br>- Запрашивает у пользователя ввод имени папки с помощью команд echo и read. Введенное имя папки сохраняется в переменной folder_name.<br>2. **Проверка существования папки**:<br>- Проверяет, существует ли папка с указанным именем, используя условие [ ! -d "$folder_name" ]. Если папка не существует, выводится сообщение об ошибке ("Папка не существует.") и скрипт завершается с кодом выхода 1.<br>3. **Список файлов**:<br>- Получает список всех файлов внутри указанной папки с помощью команды files=( "$folder_name"/* ). Это создает массив с именем files, содержащий пути ко всем файлам в папке.<br>4. **Переименование файлов**:<br>- Перебирает каждый файл в массиве files с помощью цикла for.<br>- Для каждого файла создает новое имя файла, используя команду new_file_name="draft_$(basename "$file")". Это новое имя файла создается путем добавления префикса к исходному имени файла со строкой "draft_".<br>- Затем переименовывает файл с помощью команды mv "$file" "$new_file_name", которая перемещает файл к новому имени файла.<br>5. **Сообщение об успехе**:<br>- После переименования всех файлов выводится сообщение об успехе ("Файлы успешно переименованы."), указывающее, что операция переименования прошла успешно. |

*Таблица 17. Промпт для объяснения кода*

# Промпты для перевода кода

Bash-код из Таблицы 16, похоже, работает нормально. Однако этот скрипт мог бы быть действительно многоразовым, если бы он запрашивал имена файлов, в идеале он должен работать как отдельное приложение с пользовательским интерфейсом. В качестве отправной точки, Python был бы лучшим языком для (веб-)приложения, чем Bash. БЯМ могут помочь с переводом кода с одного языка на другой. Посмотрите пример в Таблице 18:

| | |
|---|---|
| Название | 1_python_rename_files |
| Цель | Написать промпт для перевода Bash-кода на Python |
| Модель | gemini-pro |
| Температура | 0.1 |
| Лимит токенов | 1024 |
| Top-K | N/A |
| Top-P | 1 |
| Промпт | Переведи приведенный ниже Bash-код в фрагмент Python.<br><br>`#!/bin/bash`<br>`echo "Введите имя папки: "`<br>`read folder_name`<br>`if [ ! -d "$folder_name" ]; then`<br>`echo "Папка не существует."`<br>`exit 1`<br>`fi`<br>`files=( "$folder_name"/* )`<br>`for file in "${files[@]}"; do`<br>`new_file_name="draft_$(basename "$file")"`<br>`mv "$file" "$new_file_name"`<br>`done`<br>`echo "Файлы успешно переименованы."` |
| Вывод | `python`<br>`import os`<br>`import shutil`<br>`# Получаем имя папки от пользователя`<br>`folder_name = input("Введите имя папки: ")`<br>`# Проверяем, существует ли папка`<br>`if not os.path.isdir(folder_name):`<br>`print("Папка не существует.")`<br>`    exit(1)`<br>`# Получаем список всех файлов в папке`<br>`files = os.listdir(folder_name)`<br>`# Перебираем файлы`<br>`for file in files:`<br>`    # Создаем новое имя файла с префиксом "draft_"`<br>`    new_file_name = f"draft_{file}"`<br>`# Перемещаем файл к новому имени`<br>`shutil.move(os.path.join(folder_name, file),`<br>`os.path.join(folder_name, new_file_name))`<br>`# Выводим сообщение об успехе`<br>`print("Файлы успешно переименованы.")` |

*Таблица 18. Промпт для перевода кода с Bash на Python*

Прочитайте и просмотрите код. Скопируйте вывод из промпта и вставьте его в новый файл: file_renamer.py. Протестируйте код, открыв окно терминала и выполнив следующую команду: python file_renamer.py.

**ПРИМЕЧАНИЕ**: При запросе кода (на Python) в Language Studio в Vertex AI вам нужно будет нажать на кнопку 'Markdown'. В противном случае вы получите простой текст, в котором отсутствует правильное отступление строк, что важно для запуска кода Python.

# Промпты для отладки и проверки кода

Давайте вручную внесем некоторые изменения в код из Таблицы 18. Он должен запрашивать у пользователя префикс имени файла и записывать этот префикс прописными буквами. См. пример кода в Фрагменте кода *3, но какая досада. Теперь он возвращает ошибки Python!

```
import os
import shutil
folder_name = input("Введите имя папки: ")
prefix = input("Введите строку для добавления в начало имени файла: ")
text = toUpperCase(prefix)
if not os.path.isdir(folder_name):
    print("Папка не существует.")
    exit(1)
files = os.listdir(folder_name)
for file in files:
    new_filename = f"{text}_{file}"
shutil.move(os.path.join(folder_name, file),
os.path.joi(folder_name, new_file_name))
print("Файлы успешно переименованы.")
```

*Фрагмент кода 3. Сломанный Python-скрипт*

Ой! Похоже на ошибку:

```
Приведенный ниже Python-код выдает ошибку:
Traceback (most recent call last):
  File "/Users/leeboonstra/Documents/test_folder/rename_files.py", line 7,
in <module>
  text = toUpperCase(prefix)
NameError: name 'toUpperCase' is not defined
```

*Фрагмент кода 4. Я сломал Python-код*

Давайте посмотрим, сможем ли мы попросить большую языковую модель отладить и проверить код. Посмотрите на Таблицу 19:

| | |
|---|---|
| Название | 1_python_debug_code |
| Цель | Написать промпт для отладки и проверки Python-кода. |
| Модель | gemini-pro |
| Температура | 0.1 |
| Лимит токенов | 1024 |
| Top-K | N/A |
| Top-P | 1 |
| Промпт | Приведенный ниже Python-код выдает ошибку:<br>Traceback (most recent call last):<br>File "/Users/leeboonstra/Documents/test_folder/rename_files.py", line 7, in<br>text = toUpperCase(prefix)<br>NameError: name 'toUpperCase' is not defined<br>Выясни, что не так, и объясни, как я могу улучшить код.<br>python<br>import os<br>import shutil<br># Получаем имя папки от пользователя<br>folder_name = input("Введите имя папки: ")<br>prefix = input("Введите строку для добавления в начало имени файла: ")<br>text = toUpperCase(prefix)<br># Проверяем, существует ли папка<br>if not os.path.isdir(folder_name):<br>print("Папка не существует.")<br>exit(1)<br># Получаем список всех файлов в папке<br>files = os.listdir(folder_name)<br># Перебираем файлы<br>for file in files:<br><br># Создаем новый файл с префиксом "draft_"<br>new_filename = f"{text}_{file}"<br># Перемещаем файл к новому имени<br>shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name))<br># Выводим сообщение об успехе<br>print("Файлы успешно переименованы.") |

| | |
|---|---|
| Вывод | Код вызывает функцию 'toUpperCase' для преобразования 'prefix' в верхний регистр, но такая функция не определена. Чтобы исправить эту проблему, вы можете использовать метод 'upper()' класса строк, который преобразует заданную строку в верхний регистр. Модифицированный код представлен ниже:<br>`import os`<br>`import shutil`<br>`# Получаем имя папки от пользователя`<br>`folder_name = input("Введите имя папки: ")`<br>`prefix = input("Введите строку для добавления в начало имени файла: ")`<br>`text = prefix.upper()`<br>`# Проверяем, существует ли папка`<br>`if not os.path.isdir(folder_name):`<br>`print("Папка не существует.")`<br>`exit(1)`<br>`# Получаем список всех файлов в папке`<br>`files = os.listdir(folder_name)`<br>`# Перебираем файлы`<br>`for file in files:`<br>`# Создаем новое имя файла с префиксом "draft_"`<br>`new_filename = f"{text}_{file}"`<br>`# Перемещаем файл к новому имени`<br>`shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_filename))`<br>`# Выводим сообщение об успехе`<br>`print("Файлы успешно переименованы.")`<br>Дополнительно, есть еще несколько улучшений, которые можно сделать в коде:<br>1. В оригинальном коде расширение файла не сохраняется. Лучше извлечь расширение из исходного имени файла и добавить его к новому имени файла.<br>2. Код предполагает, что в имени папки нет пробелов. Однако это хорошая практика, чтобы обрабатывать пробелы в имени папки, заменяя их подчеркиваниями или дефисами.<br>3. Код не обрабатывает ошибки, которые могут возникнуть во время процесса переименования. Было бы лучше обернуть вызов shutil.move в блок try...except, чтобы отлавливать любые потенциальные ошибки.<br>4. Код строит полные пути для исходных и новых файлов. Лучше использовать функцию os.path.join для этого.<br>Вот улучшенный код с этими предложениями:<br>`python`<br>`# Получаем расширение файла`<br>`ext = os.path.splitext(file)[1]`<br>`# Создаем новое имя файла с префиксом и расширением`<br>`new_filename = f"{prefix}_{os.path.splitext(file)[0]}{ext}"`<br>`# Строим полные пути для исходных и новых файлов`<br>`old_path = os.path.join(folder_name, file)`<br>`new_path = os.path.join(folder_name, new_filename)`<br>`# Пробуем переместить файл`<br>`try:`<br>`shutil.move(old_path, new_path)`<br>`except Exception as e:`<br>`print(f"Произошла ошибка при перемещении файла {file}: {e}")`<br>`# Выводим сообщение об успехе` |

*Таблица 19. Промпт для отладки и проверки Python-кода*

Это отлично. Модель не только рассказала мне, как решить проблему, но и обнаружила, что в моем коде было еще больше ошибок и как их исправить. Последняя часть промпта содержала предложения по общему улучшению кода.

## Мультимодальный промптинг

Промптинг для кода все еще использует обычную большую языковую модель. Мультимодальный промптинг — это отдельная задача, он относится к технике, при которой вы используете несколько форматов ввода для направления большой языковой модели, вместо того чтобы полагаться только на текст. Это может включать комбинации текста, изображений, аудио, кода или даже других форматов, в зависимости от возможностей модели и задачи.

# Источники и ссылки

Примечание переводчика: некоторые ссылки из оригинальной публикации не работают.


8. Zheng, L., et al., 2023, Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models:

https://openreview.net/pdf?id=3bq3jsvcQ1

9. Wei, J., et al., 2023, Chain of Thought Prompting:

https://arxiv.org/pdf/2201.11903.pdf

10. Google Cloud Platform, 2023, Chain of Thought and React:

https://github.com/GoogleCloudPlatform/generative-ai/blob/main/language/prompts/examples/chain_of_thought_react.ipynb

11. Wang, X., et al., 2023, Self Consistency Improves Chain of Thought reasoning in language models:

https://arxiv.org/pdf/2203.11171.pdf

12. Yao, S., et al., 2023, Tree of Thoughts: Deliberate Problem Solving with Large Language Models:

https://arxiv.org/pdf/2305.10601.pdf

13. Yao, S., et al., 2023, ReAct: Synergizing Reasoning and Acting in Language Models:

https://arxiv.org/pdf/2210.03629.pdf

14. Google Cloud Platform, 2023, Advance Prompting: Chain of Thought and React:

https://github.com/GoogleCloudPlatform/applied-ai-engineering-samples/blob/main/genaion-vertex-ai/advanced_prompting_training/cot_react.ipynb

15. Zhou, C., et al., 2023, Automatic Prompt Engineering - Large Language Models are Human-Level Prompt Engineers:

https://arxiv.org/pdf/2211.01910.pdf

# Завершение второй части

Мы подробно рассмотрели продвинутые техники промпт-инжиниринга, которые значительно расширяют возможности взаимодействия с большими языковыми моделями:

- **Промптинг с отступлением (Step-back prompting)** позволяет БЯМ сначала обдумать общую концепцию, а затем применить ее к конкретной задаче, что существенно повышает качество ответов для сложных вопросов.
- **Цепочка рассуждений (Chain of Thought)** дает возможность модели пошагово объяснять свой ход мыслей, что особенно полезно для задач, требующих логических рассуждений.
- **Самосогласованность (Self-consistency)** через создание множественных путей рассуждения и выбор наиболее часто встречающегося ответа помогает повысить точность результатов.
- **Дерево рассуждений (Tree of Thoughts)** расширяет концепцию цепочки рассуждений, позволяя БЯМ исследовать несколько параллельных путей рассуждения одновременно.
- **ReAct (рассуждения и действия)** объединяет процессы рассуждения с возможностью использования внешних инструментов, что делает БЯМ более функциональными для решения практических задач.
- **Автоматический промпт-инжиниринг** демонстрирует, как можно использовать сами языковые модели для создания и улучшения промптов.
- **Промптинг для кода** показывает, как эффективно использовать БЯМ для написания, объяснения, перевода и отладки программного кода.
- **Мультимодальный промптинг** открывает перспективы использования нескольких форматов ввода для более эффективного взаимодействия с моделями.

Эти техники не просто теоретические концепции — они представляют собой практические инструменты, которые можно немедленно применять для улучшения результатов работы с большими языковыми моделями.

Что дальше?

В третьей, заключительной части цикла мы сфокусируемся на лучших практиках промпт-инжиниринга, которые помогут вам максимально эффективно применять все изученные техники:

- Оптимальные способы предоставления примеров в промптах
- Проектирование простых и эффективных промптов
- Точное описание желаемого результата
- Использование JSON и схем для структурирования данных
- Документирование экспериментов с промптами
- Методы коллективной работы над промптами
- Специфические рекомендации для работы с цепочками рассуждений
- ...и многое другое

Третья часть станет практическим руководством, которое поможет вам систематизировать полученные знания и выработать методологию для создания эффективных промптов под конкретные задачи.

Надеюсь, эта вторая часть углубила ваше понимание возможностей промпт-инжиниринга. Буду рад вашим комментариям, вопросам и обсуждению практического опыта работы с продвинутыми техниками промптинга.

---

Revision #2
Created 2026-02-16 09:01:36 UTC by Эд Кукса
Updated 2026-02-16 09:08:51 UTC by Эд Кукса