3.4. «Язык» для LLM: структурированный промптинг (XML/JSON) — зачем и как?

Зачем нужен структурированный промптинг

Большие языковые модели хорошо генерируют связный текст, но не гарантируют корректный формат вывода. Когда разработчики используют LLM в автоматизированных конвейерах (с последующей обработкой вывода кодом), важно быть уверенным, что ответ удовлетворяет заранее определённой схеме. *Structured Outputs* в API OpenAI позволяет передавать модели JSON Schema и возвращать объект, который строго соответствует схеме. Документация подчёркивает три ключевых преимущества:

- **Надёжность/типовая безопасность** (*type-safety*) — модель всегда возвращает валидный JSON; нет необходимости вручную проверять и перезапрашивать ответы[1].
- **Явные отказы** — если модель не может выполнить запрос (например, из-за политики безопасности), она возвращает поле `refusal` с объяснением отказа, что позволяет программно отлавливать ошибки[1].
- **Простота промптинга** — не требуется «заставлять» модель следовать формату; схема задаёт структуру ответа, а SDK самостоятельно сериализует вывод[1].

Такие гарантии особенно полезны при построении кодовых ассистентов: результаты могут быть преобразованы в объекты вашего языка (например, Python `dataclass`) и непосредственно использоваться в приложении, что облегчает последующую обработку и тестирование.

Формы Structured Outputs: function calling vs text_format

В API OpenAI есть два способа получать структурированные ответы:

- **Function calling** — подходит, когда модель должна вызвать «функцию» (реальный API или метод) и вернуть аргументы. В этом случае вы описываете функции в запросе; модель выбирает нужную функцию и отправляет параметры.
- **response_format (json_schema)** — используется, когда необходимо отформатировать обычный ответ для пользователя в соответствии с заданной JSON Schema, а не вызывать инструменты. Документация советует: если вы «соединяете модель с инструментами, данными или действиями», лучше применять `function calling`; если просто хотите структурированный ответ для отображения пользователю — использовать `text_format/response_format`[1].

В таблице сравнения показано, что *Structured Outputs* поддерживают проверку схемы и доступны начиная с моделей `GPT-4o-mini` и более поздних снапшотов, а старые модели (например `gpt-3.5-turbo` и `gpt-4-turbo`) поддерживают лишь «JSON mode» без строгой проверки схемы[1].

Пример: извлекаем структурированную информацию

В документации приведён пример, как с помощью `Pydantic` определить схему ответа, передать её в API и получить объект без ручного парсинга. Создаётся модель `CalendarEvent` с полями `name`, `date` и `participants`. Запрос задаёт задачу: «Извлечь информацию о событии». Затем вызывается метод `client.responses.parse` с `text_format=CalendarEvent`. В результате возвращается объект `CalendarEvent`, заполненный данными из текста[1]. Такой подход защищает от ошибок разбора и упрощает проверку типов.

Пример «chain of thought» расширяет эту идею: схема `MathReasoning` содержит список шагов (`Step`) и поле `final_answer`. Модель создаёт пошаговое объяснение решения уравнения, а результат автоматически парсится в объект Python. Вывод содержит ключи `steps` (каждый шаг — объяснение и итог) и `final_answer`[1]. Этот пример показывает, что можно комбинировать генеративное рассуждение с жёсткой структурой вывода.

Как использовать Structured Outputs

Из документации следуют три основных шага[1]:

1. **Определите схему.** Опишите необходимую структуру с помощью JSON Schema или Pydantic / Zod.
2. **Передайте схему в запрос.** В `function calling` это список функций; в `response_format` — объект `{type: "json_schema", schema: ...}`.
3. **Обработайте пограничные случаи.** Модель может отказать (поле `refusal`); дополнительно добавьте инструкции, как поступать, если введённый пользователем текст не соответствует ожидаемому формату.

Документ также выделяет тонкости:

- При работе с пользовательским вводом включайте в промпт инструкции, что делать, если текст не подходит под схему; иначе модель будет «галлюцинировать», подгоняя любые данные под ваш формат[1].
- Даже при использовании Structured Outputs ответы могут содержать ошибки; корректируйте инструкции и разбивайте задачи на более мелкие подзадачи[1].
- Чтобы схема в коде и JSON Schema не расходились, используйте Pydantic/Zod или автоматически генерируйте схемы в CI / CD (например, проверяйте изменение модели, автоматически обновляя JSON Schema)[1].
- Поддерживается *стриминг*: модель может постепенно отправлять части структуры, что полезно при отображении длинных JSON полей или аргументов функции по мере их генерации[1].

Отказы и безопасное поведение

Если по соображениям безопасности модель не может выполнить запрос, API возвращает поле `refusal` с сообщением (например, `"I'm sorry, I cannot assist with that request."`)[1]. В коде следует проверить наличие `refusal` и обработать отказ (показать сообщение пользователю или переключиться на альтернативный режим)[1].

Pydantic: валидация и генерация схем

`Pydantic` — популярная библиотека на Python для валидации данных и генерации схем. Документация отмечает, что `Pydantic` — «наиболее широко используемая библиотека для проверки данных»; она позволяет описывать, как должны выглядеть данные, в чистом Python, а затем автоматически валидировать ввод[2]. Несколько важных возможностей:

- **Сериализация и генерация JSON Schema.** Pydantic умеет сериализовать модели в Python `dict`, в JSON-ready `dict` и в JSON-строку. При этом можно исключать поля, unset-значения и значения по умолчанию[2]. Для любой модели Pydantic можно сгенерировать JSON Schema, что облегчает создание самодокументируемых API. Библиотека совместима с последней версией спецификации JSON Schema 2020-12 и OpenAPI 3.1[2].
- **Строгий режим и преобразование типов.** По умолчанию Pydantic пытается привести данные к нужному типу (например, строку `"10"` привести к числу). Но в `strict mode` типы не приводятся, и при несоответствии схемы выбрасывается ошибка. При этом Pydantic может валидировать JSON за один шаг, сохраняя строгие проверки и выполняя разумные преобразования (например, превращать ISO-строки в `datetime`)[2]. Это полезно, когда вывод LLM должен строго соответствовать типам.

Guardrails AI: валидация LLM-вывода

`Guardrails AI` предоставляет высокоуровневую обёртку вокруг LLM, которая помогает получить валидный JSON и проверить его на соответствие схеме. В руководстве «Generate Structured Data» отмечается, что Guardrails обеспечивает несколько интерфейсов, чтобы ответы LLM всегда были валидным JSON, который затем можно проверить валидаторами Guardrails[3]. Процесс обычно выглядит так:

1. **Создайте Pydantic-модель и guard.** Описываете структуру данных; поля Pydantic поддерживают встроенные валидаторы (например, регулярные выражения). Далее с помощью `Guard.for_pydantic` создаётся «guard» для этой модели[3].
2. **Выберите метод генерации.** Guardrails поддерживает:
   - `tool/function calling` — модель GPT-4o/4-turbo/3.5-turbo вызывается в режиме вызова функций, а Guardrails формирует правильный формат аргументов[3];
   - `prompt updates` — добавление `complete_json_suffix_v3` к промпту, чтобы модель сгенерировала валидный JSON[3];
   - `constrained decoding` — для моделей Hugging Face (например, через `jsonformer`) Guardrails обеспечивает ограниченное декодирование, которое гарантирует правильный формат[3];
   - `JSON Mode` — для моделей OpenAI, которые поддерживают `response_format={"type": "json_object"}`[3];
   - `Strict JSON Mode` — использование схемы (`response_format_json_schema`) для жёсткого контроля структуры[3].

Guardrails снимает с разработчика необходимость разбираться с тонкостями каждого режима и обеспечивает проверку входов/выходов, что улучшает безопасность и устойчивость приложений.

Outlines: библиотека для гарантированной структуры

`Outlines` — ещё один инструмент для управления структурой вывода. Главная страница библиотеки поясняет, что Outlines обеспечивает гарантированные структурированные ответы напрямую от любой LLM. Основные преимущества:

- **Работает с любыми моделями** (OpenAI, Ollama, vLLM, Hugging Face и др.) и использует общий интерфейс `model(prompt, output_type)`[4].
- **Гарантирует валидную структуру** — избавляет от необходимости вручную парсить нестандартный JSON или исправлять разбитый вывод[4].
- **Независимость от провайдера** — можно переключать модели без изменения кода[4].
- **Богатое описание структуры** — поддерживает JSON Schema, регулярные выражения и контекстно-свободные грамматики[4].
- **Высокая производительность** — библиотека добавляет лишь микро-задержку и компилирует структуру один раз, а не на каждый запрос[4].

Использование Outlines похоже на Pydantic: вы определяете класс (например, `Customer` с полями `name`, `urgency`, `issue`), передаёте его вместе с текстом в модель — и получаете объект `Customer`, гарантированно удовлетворяющий схеме[4].

Практические рекомендации

- **Используйте теги или XML/JSON-обёртки.** При формировании промптов обозначайте блоки примеров и требуемого формата явными тегами (`<example>...</example>`, `<output>...</output>`). Это помогает модели придерживаться структуры и облегчает парсинг. В RAG-системах или генерации кода можно использовать XML-теги как «рамки поведения», чтобы отделять инструкции, данные и ответы.
- **Определяйте схемы через Pydantic.** Такой подход упрощает генерацию JSON Schema и позволяет статически проверять соответствие типов. Для Python-кодовой базы это наиболее естественный способ описать структуру ответа.
- **Выбирайте режим generation.** При интеграции с API OpenAI используйте `response_format` для пользовательских ответов, а `function calling` — для вызова инструментов. Для моделей Hugging Face рассмотрите Guardrails с `constrained decoding` или Outlines.
- **Обрабатывайте отказы и ошибки.** Проверяйте наличие поля `refusal` и предусматривали fallback-стратегию. Для неопределённых запросов добавляйте в промпт инструкции, как действовать при некорректном вводе.
- **Интегрируйте в CI/CD.** Автоматически генерируйте JSON Schema из Pydantic и проверяйте соответствие в CI. Это поможет избегать рассинхронизации между кодом и схемой и повысит надёжность LLM-пайплайнов.

Вывод

Структурированный промптинг позволяет «научить» LLM говорить на чётко определённом «языке» с жёсткой схемой. Использование JSON Schema, Pydantic, Guardrails или Outlines обеспечивает детерминированность и предсказуемость вывода, а XML/JSON-теги помогают разделять инструкции и данные. Для разработчиков это означает меньший риск

галлюцинаций, простое тестирование и возможность надёжно интегрировать LLM в существующие процессы.

Источники

[1] Источник: platform.openai.com
[2] Источник: docs.pydantic.dev
[3] Источник: guardrailsai.com
[4] Источник: dottxt-ai.github.io