

Разработка с использованием ИИ

Дмитрий Нестерук
@dnesteruk



О чем этот рассказ

- Очень краткий обзор status quo
- Примеры и типовые подходы
- Инфраструктурные задачи и решения
- Куда всё движется?

Введение

Часть 1

Терминология

- AI (Artificial Intelligence) — искусственный интеллект. Этот лейбл сейчас лепят на всё что автоматизируется.
- LLM (Large Language Model) — большая языковая модель. Тип ИИ разработанный для обработки и генерации человеческого языка.
- GPT (Generative Pre-trained Transformer) — тип LLM, нейросеть построенная на архитектуре трансформеров.

Large Language Model

- Размер: модели «большие» из-за большого количества параметров — внутренних настроек модели. Чем больше параметров, тем сложнее и точнее модель.
- Обучение на больших данных: веб страницы, книги, статьи, и т.п. Структура языка, смысл слов, связи.
- Генерация текста: ответы на вопросы, написание текстов (статьи, книги), кода
- Контекстная осведомленность

Generative Pre-Trained Transformer

- Тип большой языковой модели которую используют OpenAI (ChatGPT) и другие
- Обработка текста целиком, концепция «внимания» (attention) <https://arxiv.org/abs/1706.03762>
- Предобучение
- Дообучение (fine tuning)
- Генерация текста (продолжить написание, ответ на вопрос, написать статью)

Как это работает?

Дополнение следующего слова на основе статистической вероятности его появления:

public static

public static void (int? string?)

public static void Main

Прикладное применение

- Чат-боты (customer service)
- Автоматизация создания контента
- Анализ текстов
- Перевод
- Работа с кодом (написание, анализ, изменения)

Модели

- Коммерческие: ChatGPT (OpenAI), Claude (Anthropic), Bard/Gemini (Google), etc.
 - Доступны через веб-интерфейс и API
- Open-source: LLaMA (Meta), Mistral, Grok (xAI), etc.
 - Доступны для скачивания и локального использования

Коммерческие модели

Плюсы

- Качество (огромный размер модели)
- Поддержка
- Интеграции
- Масштабируемость
- Безопасность

Минусы

- Надо платить
- Непрозрачны
- Даже в платных режимах есть ограничения
- Цензура, bias 😞
- Проблемы с приватностью

OSS модели

Плюсы

- Бесплатны
- Прозрачность
- Гибкость (кастомизация, fine tuning)
- Нет привязки к вендору
- Приватность

Минусы

- Требуют существенных вычислительных ресурсов
- Поддержка?
- Баги

Как использовать?

- Коммерческие модели:
 - Зарегистрироваться на chatgpt.com, claude.ai и т.п.
 - Немного бесплатных запросов
 - Больше запросов за деньги
- OSS модели:
 - Скачать LMStudio
 - Загружать и использовать все доступные модели
- Также:
 - Использование IDE и сторонних инструментов со встроенными LLM

Запрос

- Запрос (prompt) — инструкции на основе которых LLM дает ответ
- Системный запрос — (порой скрытые) инструкции для LLM которые идут перед запросом
 - You are a helpful, smart, kind, and efficient AI assistant. You always fulfill the user's requests to the best of your ability.
- Контекст (context) — информация о текущем общении, которая вгружена в LLM. Контекст растет по мере линейного общения.
 - Чем больше контекста, тем точнее результат 😊

ЯЗЫКИ

- LLMы знают *только* языки
 - Обычные разговорные языки: английский, французский, русский, и т.п.
 - Языки программирования и представления
 - Специализированные языки различных платформ (например, regex-ы, язык формул в Excel и т.п.)
 - Специализированная поддержка конкретных языков (напр. Python)
- Для работы с другими представлениями есть
 - Сторонние системы (например Midjourney)
 - Встроенные интеграции со сторонними системами (Bing, Dall-E, и т.п.)
 - Мультимодальные интеграции (голос, видео)
 - API

LLM в разработке

Часть 2

Историческая справка

- Первое упоминание — Microsoft IntelliSense
- Механизм «статистического» code completion
 - Copilot
 - JetBrains IDE Tools/AI
 - Cursor IDE
- Дополнительный набор фич

Кодогенерация

- LLMы могут генерировать код (surprise!)
- Классы, тесты, и т.д.
- Комментарии и объяснения к существующему коду

Create a class called Person with Name and Age

```
public class Person {  
    public string Name { get; set; }  
    public int Age { get; set; }  
  
    // Constructor  
    public Person(string name, int age) {  
        Name = name; Age = age;  
    }  
  
    // Override the ToString method to return a readable  
    // representation of the object  
    public override string ToString() {  
        return $"Name: {Name}, Age: {Age}";  
    }  
}  
  
// Usage
```

Что было сделано?

- Интуитивно хороший работающий код
- Код был сгенерирован на C# (если не указать язык, получим Python)
- Атрибуты реализованы как get-set свойства
- Были выбраны «разумные» типы свойств
- Вещи которые мы совсем не просили
 - Инициализирующий конструктор
 - ToString()
 - Комментарии (порой не очень полезные)
 - Пример использования кода

Ключевые проблемы

- Настройки coding style
- Настройки проекта (namespace)
- Что генерировать а что нет, и как это генерировать
- Следует как-то формализовать правила генерации заранее

Преамбула

- Максимальное количество настроек проекта
- Описывается в свободной (но максимально краткой) форме
- Дописывается по мере использования LLM

Типичная преамбула

You are a world-class C# and .NET developer. I will be asking you to write C# code according to the following rules. Generate only the code I ask for using the LATEST version of the C# language. Do not generate class members I did not ask for. When generating properties, unless there is specific behavior in getters or setters, use public fields instead of properties.

Enumerations: Whenever you generate an enum, be sure to define a ``DescriptionAttribute`` for each enum case.

Namespace: All code should be generated in namespace `ElectricalAIDemo`, namespace should be declared without indentation (i.e., a line ``namespace Xyz;`` before class declaration).

Coding style: Use two spaces for indentation. Use ``var`` where possible. Field or constructor argument names should not begin with ``_``. Use target-typed ``new`` where possible. Use collection initializer expressions where possible.

Documentation: unless requested otherwise, generate XML documentation for every member of the type (class, interface, record, enum and so on) as well as the type itself. Also, be sure to document any tricky parts of code inside methods.

Unit tests: When I ask to create unit tests, use NUnit and ``Assert.That()`` syntax. Do not create unit tests unless I asked for them explicitly.

IMPORTANT: if in my instructions I mention a type that you do not know about then, before generating ANY code, you need to ask me for a definition of the type you do not know. Only when you've encountered all unknown types can you generate code.

Что не так?

- Любую информацию которой нет, LLM будет «додумывать»
 - «Галлюцинации»
- Еще хуже: каждый запуск даст радикально отличающийся результат
- Лучший результат: когда максимум информации прописывается
- Но тогда какой смысл в LLM?
- Решение: *метазапросы*

Метазапрос

- Метазапрос = запрос, результатом которого является один или более запросов для LLM
- Зачем?
- Лень полностью эксплицировать (детально описывать) задачу
- Поэтому:
 - 1) Просим LLM максимально детализировать задачу
 - 2) Редактируем детализацию

Пример метазапроса

A DIN rail is a standardized metal rail for holding modular equipment. Each DIN rail can store a fixed number of standard modules, where a standard module is 18mm wide. Therefore, when talking about a DIN rail, we are concerned with:

- The number of standard modules this rail can hold
- The modules actually attached to this DIN rail. Each module is of ``Module`` type. A module is placed on the DIN rail in any position that is empty, i.e., there is enough space to store the module at that particular location. Typically, but not always, we try to find an empty space immediately following the last module already on the DIN module, if there are any. When placing a module on the DIN rail, we store a reference to that module as well as its position from the left of the DIN rail, measured in millimeters.

Create a ChatGPT prompt that asks to create the ``DINRail`` class according to this specification.

Результат метазапроса 1

Please generate a C# class named `DINRail` within the `ElectricalAIDemo` namespace. The class should have the following members:

1. A field to store the number of standard modules the rail can hold.
2. An inner class named `ModulePlacement` with the following members:
 - A field of type `Module` named `Module` (the `Module` class is defined elsewhere).
 - An integer field named `Position` representing the position of the module from the left of the DIN rail, measured in millimeters.
3. A list of `ModulePlacement` objects representing the modules actually attached to the DIN rail.
4. A method `PlaceModule` that takes a `Module` as a parameter and tries to place it on the DIN rail in the next available position. This method should:
 - Find the next available position for the module.
 - Store a reference to the module and its position in a new `ModulePlacement` object.
 - If there is not enough space to place the module, the method should return `false`. Otherwise it should return `true`.

Результат метазапроса 2

****Requirements:****

- Use the latest version of the C# language.
- Namespace declaration should not be indented.
- Use public fields instead of properties unless specific behavior in getters or setters is required.
- Use two spaces for indentation.
- Use `var` where possible.
- Field or constructor argument names should not begin with `_`.
- Use target-typed `new` where possible.
- Use collection initializer expressions where possible.
- Include XML documentation for the class, its members, and any tricky parts of code inside methods.

Что дальше?

- Редактируем детализированную спецификацию
- Дописываем нужные аспекты
- Выносим все «общие» требования в преамбулу
- Выполняем запрос и забираем код

«Обратные» запросы

- Иногда все же проще написать код чем объяснять
- Но хочется чтобы все равно был актуальный запрос!
- Что делать?
- «Вывести» запрос из кода

Метазапросы и рекурсия

- Мета-запросы могут работать рекурсивно
- Можно описать концепции на верхнем уровне
- LLM может «проработать» концепцию на несколько уровней
- Конечными листьями дерева может быть конкретная реализация (код)

Проблема зависимостей

- Допустим что у Person есть Address
- Если начать использовать Address без контекста, LLM «догаллюцинирует» его в моменте
 - String property, separate class, etc.
- Каким-то образом нужно вгрузить зависимости в контекст
- Хорошие идеи
 - DIP (зависимость от абстрактных типов)
 - Запрос на краткое содержание

Create a summary of the following class. A summary lists the public API of the class together with information about what the class members do. Keep it short and do not include any extra information.

Class Summary: `Cable`

Properties

- `Insulation` `Insulation { get: set: }`
 - Gets or sets the insulation type of the cable.
- `Wire[]` `Wires { get: }`
 - Gets the array of wires the cable is composed of.
- `int CableCount { get: }`
 - Gets the number of wires in the `Wires` property.

Constructor

- `Cable(Insulation insulation, params Wire[] wires)`
 - Initializes a new instance of the `Cable` class with the specified insulation type and set of wires.
 - Throws an `ArgumentException` if no wires are provided.

Тесты

- LLMы прекрасно генерируют юнит-тесты классов
- Можно использовать подход метазапросов
- Специфику можно определить в преамбуле
- Проверяйте тестовые данные

Рефакторинги

- LLM может сделать рефакторинг
- Не обладает общей картиной проекта/решения
- Нет семантической информации
- Теоретические возможности существенно превышают эвристики на AST
- Например: Name \rightarrow FirstName + LastName

Следование шаблонам

- LLMы очень хорошо понимают генерацию текста на основе шаблонов
- Например, просим `Person.ToString()` выводить пример
Hello, my name is John, I am 30 years old.
- Получаем:

```
public override string ToString()  
{  
    return $"Hello, my name is {Name} and I am {Age} years old.";  
}
```

Перспективы

Часть 3

Best Practices

- Сохраняем и регенерируем код на основе метазапросов и результирующих запросов
- Слабая связанность кода (абстракции, DI, посредники и т.п.)
- Разделение типов на AI-generated и «ручную» часть
 - Partial classes

Макросы (привет C++!)

- При автоматизации кодогенерации, все запросы являются one-shot (весь контекст собирается одним запросом)
- И в ручном и в автоматическом режиме приходится повторять инструкции
- LLMу можно определить макросы, например через \
- Это позволяет писать укороченные инструкции

We're going to introduce a macro system. Whenever I write a macro prefixed with `\`, you're going to replace that macro wholesale with what the body of the macro is. Macros will be defined using `\foo = bar` syntax, so whenever you see `\foo`, replace it completely with `bar`. Understood?

Understood! Whenever you define a macro with the syntax `\foo = bar`, I'll replace `\foo` with `bar` in any of your messages that use it. Just let me know what macros you'd like to set up!

Главная ложка дегтя

- Любые знания LLM являются статистическими
- Если задача известна, LLM решит ее наиболее «общим» способом
- Если задача неизвестна или имеет свою специфику, LLM не решит ее вообще
- Более того, LLM может не решить ее даже при экспликации пошагового решения
- Чем ближе к boilerplate тем полезней LLM-ы

Отсутствие вида сверху

- LLM не видит весь проект целиком, и
- Не трекает изменения в проекте
- Загрузить в контекст весь проект невозможно
 - Например, ChatGPT Enterprise ctx size = 128k
- Не видит связи между спецификацией и кодом
 - До момента выполнения

Заключение

- LLM-ы являются конкурентным преимуществом для программистов
- Возможности LLM ограничены, увольняться пока рано
- Эффективная работа с LLM является навыком
- Максимальная эффективность достигается путем написания своих собственных интеграций

Спасибо за внимание!

- @dnesteruk
- nesteruk.wordpress.com
- youtube.com/@soloondotnet