1 Введение в большие языковые модели

В этой главе

- ✓ Введение в генеративный ИИ (в частности, в большие языковые модели)
- ✓ Преимущества генеративного ИИ
- ✓ Где и когда не стоит применять генеративный ИИ

Хотите вы этого или нет, но вы незаметно получили повышение. Это случилось со всеми разработчиками. Почти в одночасье инженеры-программисты стали техническими менеджерами. Теперь в вашей команде появился умный и талантливый джуниор-разработчик — генеративный ИИ, который становится вашим новым партнером по программированию. Поэтому теперь в вашу ежедневную рутину должно войти наставничество, руководство и код-ревью. В этой главе представлен обзор генеративного ИИ, в частности больших языковых моделей (LLM) ChatGPT, GitHub Copilot и AWS CodeWhisperer.

ПРИМЕЧАНИЕ Это не привычная книга по программированию и не пошаговая инструкция. Вам предстоит взаимодействовать с большими языковыми моделями (LLM), и как в любом диалоге, ответы будут меняться в зависимости от модели и контекста. Результат, который вы получите, может отличаться от примеров в книге — и это нормально. Не расстраивайтесь, а исследуйте возможности. В конечном счете сам процесс обучения не менее важен, чем его результат. Может показаться, что следовать за материалом непросто. Не спешите! Проявите терпение. Если вы дис-

циплинированны и готовы к экспериментам, то сможете направлять GPT в нужное русло и использовать его так, как задумывалось в этой книге, — для развития ваших навыков программирования.

1.1. Ускорение разработки

Добро пожаловать! В мире создателей программного обеспечения настала новая эра. Теперь у каждого из нас есть не просто полезный инструмент, а настоящий друг и помощник, талантливый инженер, который поднимет ваши возможности на новый уровень. Только представьте: вы сможете разрабатывать сложные системы, очень быстро писать код и тестировать его, добиваясь беспрецедентной надежности. И все это благодаря ИИ, который учился у лучших программистов нашей планеты. В этой книге мы расскажем, как он не только поможет в решении повседневных задач, но и позволит добиться ранее недостижимых целей, ускорит разработку, повысит качество, откроет путь к невиданным инновациям.

LLM и генеративный ИИ могут значительно ускорить создание ПО: писать за вас шаблонный код. Автоматизация рутины позволит сосредоточиться на творчестве: решать проблемы, придумывать и добавлять полезные фичи.

Рассмотрим пример. Допустим, нужно, чтобы ИИ подготовил приложение Flask. Для этого можно написать такой промт:

NC Создай шаблон для приложения Flask[1].

В ответ LLM выдаст что-то похожее на следующий код на Python.

Листинг 1.1. Пример кода шаблона приложения Flask, написанный LLM

```
from flask import Flask

app = Flask(__name__)

@app.route('/')
def home():
  return "Привет, мир!"

if __name__ == '__main__':
  app.run(debug=True)
```

LLM — это системы глубокого обучения. Они прекрасно умеют отвечать человеческим языком на вопросы, заданные на нем же. Подумайте: чтобы получить готовый к работе код, достаточно просто сказать, что вам нужно. Прямо по-русски:

NC Создай класс для управления пользовательскими сессиями на Питоне.

[1] NC (Nathan Crocker) — инициалы автора книги. Так здесь и далее обозначаются промты.

А вот и возможный ответ.

Листинг 1.2. Пример кода, сгенерированного для создания класса управления сессиями

```
import uuid
from datetime import datetime, timedelta
class SessionManager:
  def __init__(self):
    self.sessions = {}
  def create_session(self, user_id):
    """ Создание новой сессии с уникальным идентификатором
    и ограничением по времени существования. """
    session_id = str(uuid.uuid4())
    # Сеанс закончится через 1 час
    expiration = datetime.now() + timedelta(hours=1)
    self.sessions[session_id] = {'user_id': user_id, 'expiration': expiration}
    return session_id

  def get_user_id(self, session_id):
    """ Получение идентификатора пользователя по идентификатору сессии
    при условии, что она еще не закончилась."""
    session_info = self.sessions.get(session_id)
    if session_info and session_info['expiration'] > datetime.now():
      return session_info['user_id']
    # Сеанс закончился или не существует
    return None

  def delete_session(self, session_id):
    """ Удаление сессии с указанным идентификатором. """
    if session_id in self.sessions:
      del self.sessions[session_id]

  def is_session_valid(self, session_id):
    """ Проверка существования сессии. """
    return session_id in self.sessions and \
      self.sessions[session_id]['expiration'] > datetime.now()
```

ИИ может помочь и при отладке: к примеру, выявить типовые фрагменты кода и посоветовать исправления. Можно попросить его проверить функцию и указать на возможные утечки памяти. И вот что можно получить в ответ.

Листинг 1.3. Использование LLM для поиска возможных утечек памяти в Python

```
def process_data():
  large_data = [x for x in range(1000000)] # Большой список из чисел
  result = sum(large_data)
  print("Результат обработки:", result)
  # Большой блок данных не удален, переменная все еще ссылается на него
  # Потенциально может остаться в памяти
process_data()
```

По мнению ChatGPT, в этом коде возможна утечка памяти, так как большой список не очищается после работы с ним и может сохраняться в памяти дольше, чем нужно. В улучшенном варианте ИИ предлагает явно очищать список или присваивать ему None после использования, чтобы ускорить освобождение памяти, особенно в средах, где сборщик мусора не освобождает ее сразу.

Листинг 1.4. Улучшенный LLM-код: потенциальная утечка устранена

```
def process_data():
  large_data = [x for x in range(1000000)] # Большой список из чисел
  result = sum(large_data)
  print("Результат обработки:", result)
  # Удаление ссылки на список для высвобождения памяти
  large_data = None
process_data()
```

Кроме того, ИИ может дать советы по рефакторингу кода, сделать его более понятным и эффективным, как, например, в следующих двух примерах.

Листинг 1.5. Избыточный код до предложенного рефакторинга

```
class DataProcessor:
  def __init__(self, data):
    self.data = data

  def process_data(self):
    if self.data is not None:
      if len(self.data) > 0:
        processed_data = []
        for d in self.data:
          if d is not None:
            if d % 2 == 0:
              processed_data.append(d)
        return processed_data
      else:
        return []
      else:
        return []

processor = DataProcessor([1, 2, 3, 4, None, 6])
result = processor.process_data()
print("Обработанные данные:", result)
```

После рефакторинга код стал более читаемым, удобным для сопровождения и соответствующим лучшим практикам языка.

Листинг 1.6. Код после рефакторинга LLM намного короче исходного

```
class DataProcessor:
  def __init__(self, data):
    self.data = data or []
```

```
  def process_data(self):
    return [d for d in self.data if d is not None and d % 2 == 0]

processor = DataProcessor([1, 2, 3, 4, None, 6])
result = processor.process_data()
print("Обработанные данные:", result)
```

LLM умеют гораздо больше, чем просто генерировать код: они достаточно умны, чтобы помочь в проектировании программной архитектуры. Поэтому программист при работе с ИИ может вести себя более творчески и стратегически, запрашивать не отдельные фрагменты кода, а задавать общие цели и требования к функциональности. В ответ LLM может предложить архитектурные решения, подходящие паттерны проектирования или даже сформировать структуру всей системы. Этот подход экономит время и дает возможность использовать опыт, накопленный ИИ в ходе обучения, предлагая оптимизации и идеи, которые разработчик мог бы не рассмотреть сразу. Благодаря такой гибкости LLM становится незаменимым инструментом в креативных и итеративных процессах разработки ПО. Мы подробно разберем это в главе 3.

ИИ повышает качество и безопасность кода и документации, что гарантирует их соответствие самым высоким стандартам. Например, при интеграции новой библиотеки ИИ автоматически предлагает наиболее безопасные и эффективные способы ее применения и помогает избежать распространенных проблем с безопасностью.

Наконец, обучение новым языкам программирования и фреймворкам становится значительно проще. ИИ может в реальном времени анализировать контекст и предоставлять релевантные рекомендации, помогая не только понять, но и применять новые концепции на практике. Например, если вы переходите на новый фреймворк, скажем Dash, ваш ИИ-ассистент сможет мгновенно сгенерировать примеры кода и предоставить детальные пояснения, адаптированные под ваш проект.

Листинг 1.7. Пример кода, сгенерированный LLM, демонстрирующий использование библиотеки

```
import dash
from dash import dcc, html
from dash.dependencies import Input, Output
import pandas as pd
import plotly.express as px

# Создание тестовых данных
dates = pd.date_range(start='1/1/2020', periods=100)
prices = pd.Series(range(100)) + pd.Series(range(100))/2
# Простая последовательность для имитации изменения цен на акции
data = pd.DataFrame({'Дата': dates, 'Цена': prices})
# Инициализация приложения Dash (обычно происходит в главном модуле)
app = dash.Dash(__name__)

# Определение структуры приложения
```

```
app.layout = html.Div([
  html.H1("График цен на акции"),
  dcc.DatePickerRange(
    id='date-picker-range',
    start_date=data['Date'].min(),
    end_date=data['Date'].max(),
    display_format='MMM D, YYYY',
    start_date_placeholder_text='Start Period',
    end_date_placeholder_text='End Period'
  ),
  dcc.Graph(id='price-graph'),
])

# Коллбэк для обновления графика по входным данным
@app.callback(
  Output('price-graph', 'figure'),
  Input('date-picker-range', 'start_date'),
  Input('date-picker-range', 'end_date')
)
def update_graph(start_date, end_date):
  filtered_data = data[(data['Date'] >=
          start_date) & (data['Date'] <= end_date)]
  figure = px.line(filtered_data, x='Date',
          y='Price', title='Изменение цен на акции с течением времени')
return figure

# Запуск приложения
if __name__ == '__main__':
  app.run_server(debug=True)
```

Результат работы этого кода Dash показан на рис. 1.1.

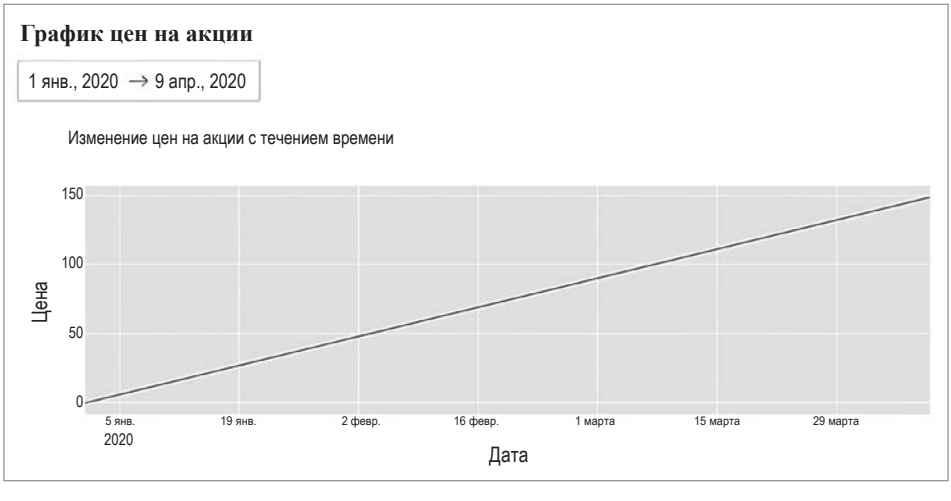

Рис. 1.1. Дашборд с графиком цен на акции, созданный ChatGPT в ответ на запрос «Создай простой дашборд на основе Dash»

Настоящая сила LLM раскрывается при их интеграции со средой разработки (IDE). Инструмент GitHub Copilot от Microsoft позволяет при помощи ИИ в режиме реального времени генерировать код напрямую в IDE, например в Visual Studio Code. Мы рассмотрим это в главе 4.

Эта книга не только объяснит ключевые концепции, но и покажет на практических примерах, как можно использовать LLM для повышения продуктивности и улучшения качества кода. Мы начнем с настройки окружения, а закончим решением довольно сложных задач. Вы узнаете, как максимально эффективно применять интеллектуальные инструменты в повседневной разработке.

1.2. LLM глазами разработчика ПО

Как и в любом практическом руководстве, в этой книге не будет много теории. Но в этом разделе я приведу наиболее важные сведения о том, как добиться от нашего нового помощника максимальной пользы.

> **Тем, кто хотел бы знать больше**
>
> Желающим поближе познакомиться с теорией, на которой строятся LLM, нейронные сети, а также генеративный ИИ, стоит прочесть эти две книги: «Строим LLM с нуля» Себастьяна Рашки (издательство «Питер», 2025 г.) и «The Complete Obsolete Guide to Generative AI» Дэвида Клинтона (David Clinton) (издательство «Manning», 2024 г.).

Начнем с простого определения, что такое LLM и чем она может быть полезна. Это поможет вам правильно объяснить ее коллегам или руководству. *Большая языковая модель* — это модель искусственного интеллекта, которая анализирует, понимает и генерирует текст, написанный обычным человеческим языком, используя огромные объемы данных, на которых она была обучена. Эти модели относятся к области глубокого обучения и особенно хорошо работают в задачах обработки естественного языка (NLP, natural language processing).

Как следует из названия, *большая* языковая модель отличается не только размером набора данных, на которых она обучалась, но также своей сложностью и количеством параметров. Современные модели, например GPT-4 от OpenAI, могут содержать сотни миллиардов параметров.

При обучении LLM читают и анализируют огромный объем самых разнообразных текстов, включая статьи, книги и материалы из интернета. Это позволяет моделям понимать структуру языка, нюансы речи и сложные закономерности человеческого общения.

Большинство LLM основаны на архитектуре Transformer — модели глубокого обучения с механизмом внутреннего внимания (self-attention), которые позволяют оценивать важность слов в составе предложения независимо от их места в нем, а значит, генерировать тексты, более значимые с точки зрения контекста. Типичная архитектура Transformer состоит из кодировщика (encoder) и декодировщика (decoder), каждый из которых включает многоуровневую структуру.

Понимание архитектуры LLM помогает более эффективно использовать их возможности, а также учитывать их ограничения при практическом применении. Модели постоянно развиваются, что обещает появление новых, все более сложных инструментов, которые разработчики смогут использовать для улучшения своих проектов.

1.3. Когда применять ИИ, а когда не стоит

Генеративный ИИ (а значит, и большие языковые модели) — это не универсальное решение. Чтобы максимизировать их пользу и избежать возможных проблем, важно понимать, когда их применение оправданно, а когда может быть неэффективным или даже проблемным. Начнем с того, в каких случаях стоит использовать LLM.

- Повышение производительности: автоматическая генерация шаблонного кода, создание документации, получение подсказок прямо в IDE. В главах 3 и 4 мы поговорим о том, как применять GitHub Copilot для более эффективного кодинга.
- Обучение и исследования: использование ИИ для изучения новых языков программирования и фреймворков, генерации примеров кода и пояснений. В главе 5 мы посмотрим, как с помощью ИИ ускорить обучение и освоение новых технологий.
- Автоматизация рутинных задач: использование ИИ для автоматизации тестирования ПО, ввода данных и других повторяющихся задач, чтобы освободить время для чего-то более сложного. В главе 7 мы обсудим автоматизацию тестирования и сопровождения кода.

Есть и ситуации, когда использовать LLM не следует. Применение ИИ в средах с чувствительными или проприетарными данными может привести к утечке информации. Тому есть несколько причин. Одна из них состоит в том, что для работы с моделью необходим контекст, а значит, по меньшей мере часть вашего кода отправится за пределы защищенной среды и даже, возможно, послужит учебными данными при дальнейшем обучении модели. Но в главе 9 мы рассмотрим несколько способов решить эту проблему.

Возможности ИИ также могут быть ограничены в случаях, когда решение задачи требует точности и квалификации. Поскольку LLM, как правило, вносит в ответы некую долю случайности (иногда это называют *галлюцинациями*), результат может оказаться довольно далеким от правильного. Поэтому сгенерированный код обязательно нужно проверить и лишь затем использовать на практике.

Генеративный ИИ обладает множеством преимуществ. Но использовать его нужно с умом, учитывая и контекст применения, и потребности конкретного проекта. Четко понимая, когда можно полностью положиться на LLM, а когда нужно действовать с осторожностью, разработчик сможет повысить свою производительность до максимума и гарантировать этичное и эффективное применение этих технологий.

Итоги

- Генеративный ИИ — одновременно и эволюция, и революция. С одной стороны, это еще один инструмент в арсенале разработчика, который просто продолжает развитие технологий, а с другой — он кардинально меняет сам подход к программированию.
- Будущее разработки ПО за генеративным ИИ. Даже мифический разработчик с максимальной продуктивностью не сможет работать так эффективно, как программист с LLM-помощником, в силах которого генерировать более качественный код значительно быстрее и дешевле, чем те, у кого такого помощника нет. Лучше потратить время на то, чтобы объяснить ИИ, что нужно сделать и как, чем писать код вручную.
- Доверяйте ИИ, но всего проверяйте его результат.