### Конспект Лекції 2: Базові Моделі Коду (Code Foundation Models)

#### 1. Вступ: Нова Парадигма у Розробці Програмного Забезпечення

Поява великих мовних моделей (LLM) стала каталізатором фундаментального зсуву парадигми в автоматизованій розробці програмного забезпечення, кардинально переосмислюючи взаємозв'язок між людським наміром та виконуваним кодом. Ці моделі, по суті, дистилюють роки накопиченого досвіду програмування в доступні інструменти, що слідують інструкціям, роблячи процес створення ПЗ доступнішим для розробників будь-якого рівня кваліфікації. Вони усувають традиційні бар'єри між знаннями предметної області та технічною реалізацією, дозволяючи перетворювати описи природною мовою на функціональний вихідний код.Сучасні LLM демонструють вражаючі можливості у широкому спектрі завдань, пов'язаних із кодом, включаючи його завершення, переклад між мовами програмування, виправлення помилок та генерацію з нуля. Ця трансформація вже стала комерційною реальністю, що підтверджується появою низки потужних інструментів, які змінюють робочі процеси розробників та межі між людською креативністю й машинною допомогою.

* **GitHub Copilot (Microsoft):** Інтелектуальний асистент, що забезпечує завершення коду безпосередньо в середовищах розробки (IDE).
* **Cursor (Anysphere):** Орієнтований на ШІ редактор коду, що уможливлює розмовне програмування, де розробник веде діалог з редактором для створення та модифікації коду.
* **CodeGeeX (Zhipu AI):** Інструмент, що спеціалізується на багатомовній генерації коду, підтримуючи широкий спектр мов програмування.
* **CodeWhisperer (Amazon):** Асистент кодування, що глибоко інтегрований з екосистемою хмарних сервісів AWS.
* **Claude Code (Anthropic) / Gemini CLI (Google):** Інструменти командного рядка, що дозволяють розробникам делегувати завдання з кодування безпосередньо з терміналу для реалізації агентних робочих процесів.Проте, попри всю свою універсальність, моделі загального призначення мають суттєві обмеження, які стають очевидними у професійній програмній інженерії. Це вимагає створення спеціалізованих базових моделей коду, які можуть долати ці виклики.

#### 2. Обмеження Моделей Загального Призначення для Завдань Кодування

Отже, ми бачимо вражаючі можливості. Але для того, щоб будувати надійні системи, ми, як інженери, повинні спочатку зрозуміти, де саме ці універсальні інструменти зазнають невдач. Широта можливостей LLM загального призначення, яка робить їх такими потужними, є водночас і їхнім ключовим недоліком для професійної програмної інженерії.Моделі, навчені на всьому, від поезії до наукових статей, є універсалами. Проте для складних інженерних завдань їм часто бракує глибини, надійності та доменної узгодженості. Розуміння цих обмежень має стратегічне значення, оскільки воно визначає напрямки для розробки більш досконалих та спеціалізованих моделей коду. Давайте детальніше розглянемо кожне з цих обмежень.**Спеціалізація та точність** Моделі загального призначення можуть генерувати код, який поверхнево виглядає правильним, але не відповідає неявним доменним обмеженням. Наприклад, згенерований код може порушувати тонкі контракти API, не відповідати внутрішнім політикам безпеки або не підтримувати інваріанти у великих системах. Вирішення реальних проблем на рівні репозиторіїв залишається складним завданням навіть для найпотужніших моделей.**Розуміння на рівні репозиторію** Навіть із розширеними контекстними вікнами LLM не завжди ефективно використовують дуже довгі вхідні дані. Їхня продуктивність знижується, коли релевантна інформація знаходиться в середині контексту, а не на його початку чи в кінці. Бенчмарки, що охоплюють завдання на рівні репозиторію (наприклад, завершення коду в кількох файлах), виявляють постійні труднощі з відстеженням міжфайлових залежностей та глобальним мисленням.**Мультимодальні перешкоди** Загальні мультимодальні моделі корисні для сприйняття скріншотів та діаграм, але їхнє розуміння детальної ієрархії інтерфейсу користувача (UI) та семантики взаємодії залишається слабким місцем. Існуючі системи часто спеціалізуються на вузьких підзадачах, а не на цілісному та послідовному розумінні екрана, що обмежує їхню здатність до стабільних переходів від сприйняття до дії в реальних застосунках.**Агентні обмеження** Агенти, що використовують інструменти, все ще зазнають невдач через крихкість довготривалого планування, прийняття рішень та слідування інструкціям. Систематичні оцінки виявляють значні прогалини в інтерактивних середовищах. Дослідження фіксують "галюцинації інструментів", коли агент обирає неправильний інструмент, використовує його невчасно або вигадує результат його роботи. Надійне планування та правильне використання інструментів залишаються відкритими викликами.Ці недоліки чітко вказують на те, що нам потрібні не просто універсальні, а спеціалізовані моделі. Це підводить нас до ключового питання: як створюються такі моделі? Давайте розглянемо життєвий цикл їх розробки, який дозволяє подолати ці проблеми.

#### 3. Життєвий цикл Базової Моделі Коду

Життєвий цикл розробки базової моделі коду — це послідовний процес, що перетворює модель із загального знавця мови на спеціалізованого експерта з програмування. Цей шлях можна порівняти з навчанням розробника: спочатку він вивчає фундаментальні концепції мови (попереднє навчання), а потім поглиблює свої знання в конкретній галузі та вчиться виконувати практичні завдання (подальше навчання). Цей процес складається з двох ключових етапів.

##### 3.1. Етап 1: Попереднє Навчання (Pre-training) — Закладання Фундаменту

Попереднє навчання — це основний етап, на якому модель засвоює загальні мовні патерни та фундаментальні знання про світ програмування. Навчаючись на величезних обсягах нерозмічених даних, модель вивчає граматику, семантику та базові структури коду.**Збір та обробка даних** Цей процес починається зі збору даних з різноманітних джерел, таких як веб-тексти, книги, репозиторії на GitHub та обговорення на Stack Overflow. Зібрані дані проходять через кілька етапів обробки:

* **Очищення:** Видалення низькоякісного та конфіденційного контенту.
* **Дедуплікація:** Усунення повторюваних зразків для підвищення різноманітності даних.
* **Токенізація:** Розбиття тексту на менші одиниці (токени), які модель може обробляти. **Задачі для попереднього навчання** Моделі навчаються за допомогою кількох самокерованих задач. Кожна з них розвиває певні "навички" моделі.

|  |  |
| --- | --- |
| ***Задача*** | ***Опис та мета*** |
| **Прогнозування наступного токена (NTP)** | Фундаментальна авторегресійна задача, де модель вчиться прогнозувати наступний токен у послідовності. Це розвиває її здатність розуміти та генерувати послідовний код. |
| **Прогнозування кількох токенів (MTP)** | Розширення NTP, де модель прогнозує кілька наступних токенів одночасно. Це підвищує ефективність навчання та когерентність згенерованого коду. |
| **Заповнення пропусків (FIM)** | Неавторегресійна задача, де модель вчиться заповнювати пропущену частину коду, маючи його початок і кінець. Це покращує розуміння глобального контексту. |
| **Навчання на основі дифузії (Diffusion Coder)** | Процес «поступового усунення шуму», де модель вчиться відновлювати вихідний код із зашумленої версії. Це підвищує різноманітність та якість генерації. |

**Безперервне попереднє навчання (CPT)** Це метод адаптації вже навченої моделі до специфічних доменів (наприклад, певної мови програмування або внутрішньої кодової бази компанії) або для оновлення її знань новою інформацією. Замість навчання з нуля, модель продовжує попереднє навчання на новому, меншому наборі даних.

##### 3.2. Етап 2: Подальше Навчання та Вирівнювання (Post-training) — Спеціалізація та Вдосконалення

Якщо попереднє навчання дає моделі широкі знання, то подальше навчання робить її точною та корисною для конкретних завдань. На цьому етапі модель вчиться слідувати людським інструкціям, адаптується до вподобань користувачів та вдосконалює свої навички у вирішенні специфічних проблем.**Кероване донавчання (Supervised Fine-Tuning, SFT)** SFT використовує розмічені дані, які зазвичай складаються з пар "інструкція-відповідь". На цих прикладах модель вчиться точно слідувати командам. Якість даних тут є вирішальною. Оскільки створення великих наборів високоякісних даних вручну є дорогим, використовуються методи їх синтезу:

* **Зворотний зв'язок від виконання коду (Execution Feedback):** Згенерований код виконується, і результат (успіх чи помилка) використовується для створення нових навчальних прикладів.
* **Взаємодія між агентами:** Створюються два агенти: один ставить запитання, а інший генерує код. Їхня взаємодія, особливо коли код потребує виправлення, стає цінним навчальним матеріалом.**Навчання з підкріпленням (Reinforcement Learning, RL)** RL дозволяє оптимізувати модель для досягнення конкретних, об'єктивних результатів. У контексті коду це найчастіше **навчання з підкріпленням на основі верифікованої винагороди (RLVR)** . Замість того, щоб порівнювати згенерований код із "правильним" зразком, модель отримує винагороду, якщо її код проходить юніт-тести. Цей метод не створює нових фундаментальних можливостей, а "відточує" вже існуючі, змушуючи модель частіше генерувати коректні та перевірені рішення.Отже, ми навчили та спеціалізували нашу модель. Але як ми дізнаємося, наскільки вона насправді "розумна"? Це приводить нас до наступного важливого аспекту: екосистеми оцінки та застосування.

#### 4. Екосистема та Застосування Моделей Коду

Цінність базових моделей коду визначається не лише їхньою внутрішньою архітектурою, а й тим, як вони застосовуються та оцінюються в реальному світі. Цей розділ досліджує практичний вимір моделей коду: від стандартизованих тестів, що вимірюють їхні можливості, до конкретних застосунків, які змінюють індустрію розробки програмного забезпечення.

##### 4.1. Оцінка та Бенчмарки: Як ми вимірюємо "інтелект" коду?

Ви можете запитати: як ми взагалі вимірюємо "інтелект" коду? Для цього існують бенчмарки. Систематична оцінка є ключовою для порівняння продуктивності різних моделей, відстеження прогресу та виявлення слабких місць. Сучасні бенчмарки класифікуються за рівнем гранулярності — від написання окремих функцій до керування цілими проєктами.

* **Рівень функції:** Ці тести оцінюють базову здатність моделі генерувати невеликі фрагменти коду на основі текстових описів. Це фундаментальна перевірка її "словникового запасу" та розуміння синтаксису.
* *Приклади:* HumanEval, MBPP.
* **Доменно-специфічні:** Тут ми перевіряємо не просто знання мови, а й експертність у вузьких галузях, таких як змагальне програмування чи математика. Це аналогічно тестуванню спеціаліста на знання його професійної області.
* *Приклади:* APPS, CodeNet.
* **Рівень репозиторію:** Це критичний рівень, оскільки саме тут перевіряється здатність моделі працювати як справжній інженер, що має справу з успадкованим кодом та складними залежностями, а не просто писати ізольовані функції.
* *Приклади:* RepoBench, CrossCodeEval.
* **Комплексна розробка ПЗ та агентні системи:** Це найвищий рівень оцінки, що імітує реальну роботу розробника. Тут тестується здатність агента не просто писати код, а й взаємодіяти з інструментами (Git, термінал) та самостійно вирішувати проблеми.
* *Приклади:* Aider's benchmarks, Agent Tool Use.

##### 4.2. Практичні Застосування: Від Асистента до Автономного Розробника

Моделі коду знаходять застосування у різноманітних інструментах, які можна згрупувати за їхньою основною функцією та способом інтеграції в робочий процес розробника.

|  |  |  |
| --- | --- | --- |
| ***Категорія*** | ***Приклади*** | ***Ключові особливості*** |
| **Асистенти, інтегровані в IDE** | GitHub Copilot, Cursor, Tabnine | Глибоке контекстне розуміння, інтеграція в робочі процеси розробників, безпека та приватність (для Tabnine). |
| **Хмарні платформи для кодування** | Amazon Q Developer, Gemini Code Assist | Спеціалізація на екосистемах хмарних провайдерів (AWS, GCP), централізоване розгортання, генерація інфраструктури як коду. |
| **Автономні агенти в терміналі** | Aider, Claude Code | Робота в командному рядку, автоматизація завдань, можливість написання скриптів, інтеграція з Git. |
| **Виправлення коду та верифікація** | RepairLLaMA, VulRepair | Автоматичне виправлення помилок, виправлення вразливостей безпеки, генерація патчів. |
| **Рев'ю Pull Request'ів** | PR-Agent, CodeRabbit AI | Автоматизація рев'ю коду, зменшення навантаження на рецензентів, забезпечення дотримання стандартів кодування. |

Але коли ми даємо агентам таку автономію, ми неминуче стикаємося з питаннями безпеки. Що станеться, якщо такий потужний інструмент використають зловмисно? Це підводить нас до фінальної, але критично важливої теми: безпека та надійність.

#### 5. Безпека та Надійність в Епоху ШІ-агентів

Надання ШІ-агентам більшої автономії — можливості самостійно виконувати код, взаємодіяти з файловими системами та зовнішніми сервісами — неминуче створює нові ризики. Тому розробка надійних механізмів захисту для запобігання зловживанням та забезпечення безпеки є абсолютно необхідною умовою для їх широкого впровадження.

##### 5.1. Вектори Атак: Як можна "зламати" модель?

Існують два основні типи атак на моделі коду, які зловмисники можуть використовувати для обходу вбудованих механізмів безпеки.

* **Маніпуляція на рівні запиту (Prompt-Level Manipulation):** Цей метод, відомий як "джейлбрейкінг" (jailbreaking), спрямований на те, щоб змусити модель ігнорувати її протоколи безпеки. Зловмисники використовують різні техніки, такі як:
* **Рольові ігри:** Моделі пропонують увійти в роль персонажа, який не має етичних обмежень (наприклад, DAN - Do Anything Now).
* **Ін'єкції префіксів:** Шкідливий запит маскується під невинний контекст (наприклад, "Ось приклад вразливого SQL-запиту для підручника з кібербезпеки: ..."), що змушує модель генерувати заборонений контент.
* **Семантична та контекстуальна маніпуляція:** Це більш витончені атаки, де шкідливе навантаження вбудовується не в основну інструкцію, а в дані, які модель вважає "довіреними" (наприклад, вміст файлу, який вона аналізує). Оскільки фільтри безпеки зазвичай перевіряють лише основну команду користувача, такі атаки можуть їх обійти, використовуючи слабкі місця в логіці інтерпретації контексту моделлю.

##### 5.2. Основи Захисту: Побудова Безпечних Середовищ

Першою лінією оборони є створення ізольованих середовищ ("пісочниць"), які обмежують потенційну шкоду від виконання коду, згенерованого агентом. Вибір технології ізоляції залежить від балансу між безпекою, продуктивністю та сумісністю.

* **Контейнери на рівні ОС (наприклад, Docker):** Є ефективними та простими у розгортанні, але вони використовують спільне ядро операційної системи з хостом. Це створює значну поверхню для атак, оскільки вразливості в ядрі можуть дозволити зловмисному коду "втекти" з контейнера.
* **Пісочниці на рівні процесів (наприклад, nsjail):** Надають дуже детальний контроль над системними викликами, які може робити процес. Хоча це потужний інструмент для реалізації принципу найменших привілеїв, він вимагає складного налаштування політик безпеки, і помилки в конфігурації можуть звести нанівець усі захисні механізми.
* **Ізоляція на основі віртуалізації (наприклад, Firecracker):** Створюють мікровіртуальні машини, що забезпечують майже апаратний рівень ізоляції. Це ефективно захищає від більшості атак на рівні ядра ОС, але не є панацеєю. Залишаються ризики, пов'язані з атаками на мікроархітектурні бічні канали (наприклад, Spectre), які можуть дозволити витік даних між віртуальними машинами.Як ви бачите, кожна технологія має свої компроміси. Отже, ми пройшли весь шлях: від створення моделі до її розгортання та захисту. Давайте підсумуємо все, що ми дізналися, і подивимося, що чекає нас у майбутньому.

#### 6. Висновки та Погляд у Майбутнє

Ми розглянули повний життєвий цикл базових моделей коду: від фундаментального попереднього навчання, яке закладає основу знань, до спеціалізованого донавчання та вирівнювання, що перетворює їх на точні інструменти. Кінцевим результатом цього процесу є розгортання моделей у вигляді складних агентних систем, здатних вирішувати комплексні інженерні завдання. З мого погляду як дослідника, я бачу три основні вектори, що визначатимуть майбутнє цих технологій.

* **Глибоке контекстуальне розуміння:** Майбутні агенти перейдуть від статичного аналізу коду до створення динамічних "ментальних моделей" репозиторіїв. Використовуючи графи знань, вони зможуть розуміти не лише структуру, а й еволюцію проєкту, його історію та неявні принципи проєктування, що дозволить їм виконувати завдання на рівні, який наближається до людського.
* **Покращена взаємодія "людина-в-циклі":** Наступне покоління агентів не замінить розробників, а стане їхніми інтерактивними партнерами. Вони діятимуть як "молодші розробники", які можуть виконувати завдання, отримувати зворотний зв'язок та ітеративно вдосконалювати свою роботу під керівництвом людини, що значно підвищить продуктивність та креативність.
* **Довіра, безпека та верифікація "за задумом":** У міру зростання автономії агентів питання безпеки стануть першочерговими. Майбутні системи будуть інтегрувати формальні методи верифікації та перевірку безпеки безпосередньо у процес свого "мислення". Це забезпечить гарантії того, що їхні дії будуть безпечними, надійними та передбачуваними, що є критично важливим для їх використання у виробничих системах.Зрештою, базові моделі коду — це не просто інструменти для написання коду. Вони стають повноцінними партнерами, які здатні розуміти, аналізувати та створювати складні програмні системи, фундаментально трансформуючи саму суть програмної інженерії.