 Егор Иванов

Подключение AI Coding Agent'a на проекты автотестов

Обновлено 17 апреля 2025, 17:48

Введение

Современные AI-ассистенты для разработки, или **AI Coding Agents**, эволюционировали от простых автодополнений до мощных инструментов, способных понимать контекст проекта, генерировать код, выполнять рефакторинг и даже запускать команды. Их интеграция в процесс разработки автотестов уже сегодня может принести значительную пользу.

Задачи в автотестировании часто включают повторяющиеся элементы, что требует написания большого объема кода с похожими структурами, такими как тестовые сценарии, Page Objects и работа с API. **AI Coding Agents**, особенно при правильной настройке с использованием **Custom Modes** и **Memory Bank** (о чем пойдет речь в этом руководстве), могут значительно облегчить эту работу, позволяя автоматизаторам сосредоточиться на более сложных и творческих задачах.

Хотя ожидать от агента идеального кода "из коробки" пока рано, текущие модели уже способны генерировать весьма неплохие заготовки тестов, которые требуют лишь небольшой доработки. Это позволяет существенно ускорить процесс написания автотестов и освободить время инженеров для более сложных и творческих задач.

Данное руководство описывает практические шаги по настройке и использованию AI Coding Agent'a (на примере **Roo Code**) в проекте автотестов для достижения максимальной эффективности.

Custom Modes - это возможность использовать агента в отдельном режиме со своими изолированными правилами.
Memory Bank - один из подходов по эффективному наполнению контекста при работе с LLM-агентом.

1. Настройка Roo Code плагина

Под **AI Coding Agent** здесь понимается AI-ассистент для IDE (в нашем случае, для VS Code), имеющий **Agent Mode**, т.е. способность самостоятельно редактировать файлы, запускать команды и выполнять другие действия в проекте.

Как и в доке про Прикладное использование AI (LLM) для разработки и тестирования рекомендуется использовать **Roo Code**.

Почему Roo Code:
Ключевое преимущество – **Custom Modes**. В отличие от инструментов с фиксированным набором режимов, Roo Code позволяет создавать собственные "личности" AI-ассистента, точно настраивая его инструкции и доступные инструменты под конкретные задачи и роли (например, можно создать специализированный режим для написания тестов в котором агент будет заточен именно под данную работу).

1.1 Установка расширения

- Ищем `Roo Code` в маркетплейсе VS Code.
- Устанавливаем.

Скриншот

1.2 Настройка игнорируемых файлов (`.rooignore`)

> **Критически важно**
>
> Это **обязательный шаг** перед тем, как разрешить LLM-агенту доступ к вашему проекту!
> Всегда проверяйте, какие `.ignore` файлы поддерживает используемый инструмент, и правильно их настраивайте. Помните, что даже при использовании внутренних моделей не стоит передавать им токены, пароли и другую чувствительную информацию.

При работе с Roo Code для исключения файлов и директорий из контекста агента используется файл `.rooignore` в корне проекта. Синтаксис аналогичен `.gitignore` .
Подробнее: Roo Code Docs - Ignore File Interactions

Рекомендуемый порядок создания `.rooignore` :

1. Создаем файл `.rooignore` в корне проекта.
2. Копируем в него содержимое `.arcignore` .
3. **Внимательно проверяем** и удаляем из `.rooignore` те пути, которые *могут быть полезны* агенту для понимания контекста (например, папки с тестовыми данными, сгенерированными отчетами типа Allure, локальными конфигами).

Отличие от `.arcignore`:
В `.rooignore` должны попасть только те файлы, которые агент *не должен* видеть ни при каких обстоятельствах (секреты, нерелевантные бинарники, тяжелые логи), а не просто те, которые не коммитятся в репозиторий. Например, если у LLM-агента будет доступ к папке с **Allure**-отчетом, он потенциально сможет лучше разобраться с причиной падения теста, проанализировав информацию из отчета.

1.3. Настройка используемых LLM моделей

Качество генерации кода напрямую зависит от возможностей используемой LLM.
Рекомендуется настроить и попробовать в Roo Code как минимум следующие модели:

- **Claude 3.7 через Eliza:**
  - **Плюсы:** Модель хорошо пишет код.
  - **Минусы:** Платная, есть бюджетный лимит, **нельзя передавать NDA-данные**.
  - **Настройка в Roo Code:**
    - Провайдер: `Anthropic`
    - Base URL: `https://api.eliza.yandex.net/raw/anthropic`
    - Token: ссылка
    - Модель: `claude 3.7 sonet`
- **DeepSeek V3-0324 (внутренняя инсталляция):**
  - **Плюсы:** Безопасна для **NDA-кода**, нет бюджетного лимита (внутренняя).
  - **Минусы:** Качество генерации кода может быть ниже, чем у Claude.
  - **Настройка:**
    - Провайдер: `OpenAI Compatible`
    - Base URL: `http://deepseek-openai.yandex-team.ru/deepseek-v3/v1`
    - Token: ссылка
    - Модель: `deepseek-0324`

Список прочих моделей можно посмотреть здесь

Шаги по настройке

1. Отркываем VS Code
2. Отрываем плагин Roo Code
3. Настраиваем параметры необходимой модели с выбором `Use custom base URL` и параметрам описанным для используемой модели.

   Скриншот

Общие рекомендации по выбору модели:

- **Экспериментируйте:** С хорошо подготовленным банком памяти многие модели могут показывать хорошие результаты. Пробуйте разные модели для разных задач (генерация тестов, рефакторинг, написание документации). Результаты смогут зависеть от сложности задачи и вашего стека (Python/Java/JS/TS).
- **Баланс:** Учитывайте качество генерации, скорость ответа, стоимость (для внешних моделей) и ограничения по NDA.

После первоначальной настройки уже можно использовать Roo Code со стандартными режимами, но для максимально продуктивной работы с автотестами рекомендуется настроить **Memory Bank** и, возможно, кастомный режим.

2. Настройка Memory Bank (Банка Памяти)

Банк Памяти (Memory Bank) — это механизм, который предоставляет LLM-агенту структурированную **"долговременную память"** о контексте вашего проекта. Благодаря **Memory Bank** агент может быстрее "вспомнить" особенности проекта в начале каждого нового диалога, эффективнее наполняя свой **рабочий контекст**. Это позволяет **экономить токены** и **повышает точность ответов**, так как модель изначально лучше понимает, как устроен проект и где искать нужную информацию.

Сравнение подходов:

- **Без Memory Bank:**
  1. Даем задачу агенту (написать тест).
  2. Он пытается сам собрать контекст (может быть неполно или неоптимально).
  3. Выполняет задачу (возможно, с ошибками или не в стиле проекта).
  4. Мы указываем на ошибки, даем примеры, дополняем контекст вручную в чате.
  5. Агент исправляет с учетом уточнений (контекст теряется в следующем диалоге).
  6. При старте нового далога весь накопленный контекст теряется и придется заново его наполнять
- **С Memory Bank:**
  1. Даем задачу агенту.
  2. Агент автоматически считывает релевантную информацию из **Memory Bank** (заранее подготовленную).

3. Агент выполняет задачу, учитывая контекст проекта (более качественно и в нужном стиле).
4. Мы корректируем результат (если нужно).
5. Агент обновляет **Memory Bank** нашими корректировками (автоматически или по команде), сохраняя знания.
6. Последующие задачи выполняются еще лучше.

Для автотестов (и не только) это критически важно, так как позволяет агенту "знать":

- Какие **фикстуры** существуют и как их использовать.
- Какие **хелперы** доступны для общих действий (логин, работа с API, генерация данных).
- Как устроены **Page Objects** или другие абстракции UI.
- Какие **общие паттерны** и **базовые классы** используются в ваших тестах.
- Какие **примеры тестов** считаются эталонными по стилю и структуре.

Существуют готовые **шаблоны Memory Bank**, например, Roo Code Memory Bank. Однако стандартные шаблоны могут не полностью подходить под специфику вашего проекта или задачи, поэтому часто требуется их доработка или создание собственного решения с нуля.

2.1 Использование стандартного Memory Bank (и его доработка)

Стандартный **Memory Bank** Roo Code (Roo Code Memory Bank) предоставляет базовый набор правил и структуру для хранения информации о проекте. Его можно установить, следуя инструкциям в репозитории (Install Script).

После установки вы можете:

- **Использовать как есть:** Если стандартные правила вас устраивают.
- **Доработать:** Отредактировать flow работы агента:
  - Например, добавить правило "не галлюцинировать" или "всегда использовать определенный стиль комментариев"
  - Отредактировать описание содержимого файлов **Memory Bank**, чтобы агент прописывал в них более релевантную информацию
    и тд

Пример правила для Memory Bank в direct

2.2 Создание собственного flow для отдельного режима

Для автотестов и других специфичных задач часто более эффективно создать полностью кастомный flow работы с **Memory Bank** в рамках **отдельного Custom Mode** Roo Code. Это дает максимальную гибкость.

Преимущества:

- Можно ограничить набор файлов, считываемых из **Memory Bank** для конкретной задачи (экономия токенов и повышение релевантности).
- Можно более точно прописать правила, когда и как обновлять **Memory Bank**.
- Настройки для тестов полностью изолированы от других режимов, т.е. отдельный режим агента для написания тестов не будет влиять на другие режимы в тч стандартные

Для этого потребуется описать отдельный режим в Roo Code и описать flow работы с банком памяти в рамках этого режима.

2.2.1 Настройка отдельного режима (Custom Mode)

Roo Code позволяет создавать свои режимы через файл `.roomodes` (Custom Modes Docs).

Шаги:

1. **Определить режим:** Добавляем или изменяем файл `.roomodes` в корне проекта. Пример:

```
{
  "customModes": [
    {
        "slug": "mode-name",
        "name": "Mode Display Name",
        "roleDefinition": "Mode's role and capabilities",
        "groups": ["read", "edit"],
        "customInstructions": "Additional guidelines"
    }
  ]
}
```

Пример описания режима `AQA` в проекте автотестов Афиши

После этого у вас появится новый режим в Roo Code:

Кастомный AQA режим

2. **Создать файл правил:** Создайте файл правил для режима (например, `.roo/rules-<mode-name>/rules.yaml`).

Примечание: Файлы правил могут иметь формат `.md`, `.txt`, но рекомендуется использовать **YAML (`.yaml`)**, так как он позволяет структурировать инструкции и использовать ссылки на блоки в yaml для их переиспользования моделью. Модели достаточно хорошо понимают такой алгоритм описания документов, такой же подход используется в (Roo Code Memory Bank)

```
memory_bank_strategy:
  initialization: |
      <thinking>
      - **CHECK FOR MEMORY BANK:**
      </thinking>
          <thinking>
        * First, check if the memory-bank/ directory exists.
          </thinking>
          <list_files>
          <path>.</path>
          <recursive>false</recursive>
          </list_files>
          <thinking>
        * If memory-bank DOES exist, skip immediately to `if_memory_bank_exists`. #Здесь ссылаемся на блок "if
          </thinking>
  if_no_memory_bank: |
      1. **Inform the User:**
          "No Memory Bank was found. I recommend creating one to  maintain project context.
```

2.2.2 Настройка правил Memory Bank для режима

Инструкции по взаимодействию с **Memory Bank** являются **ключевой частью файла правил** вашего кастомного режима (`rules.yaml` из предыдущего шага).

Как разработать эти правила:

1. **Изучите стандартные примеры описания flow работы Memory Bank:**
   Посмотрите, как работа с памятью описана в стандартных банках, примеры:

- (Roo Code Memory Bank
- https://docs.cline.bot/improving-your-prompting-skills/cline-memory-bank#cline-memory-bank-custom-instructions-copy-this
  Используйте их как отправную точку. Также полезно заметить, что в стандартных правилах для Roo Code активно используются **инструменты (`tools`)**, доступные агенту (определяются базовым промптом Roo Code).
  Использование `tools` (таких как `<list_files>`, `<read_file>`, `<edit_file>`) в ваших правилах позволяет точнее управлять поведением агента и автоматизировать его действия.

  Как посмотреть tools и пример использования

2. **Используйте LLM:**
   Откройте ваш **rules file** (`rules.yaml`) и попросите агента (в режиме с доступом к этому файлу) помочь вам: *"Сгенерируй flow (инструкции) для взаимодействия с **Memory Bank** в формате YAML для Roo Code, взяв за основу стандартный подход [ссылка на стандартный], но с фокусом на задачи автотестирования: агент должен искать фикстуры, хелперы, примеры тестов и обновлять память после коррекций"*.
3. **Адаптируйте под себя:**
   Доработайте сгенерированные инструкции, чтобы они соответствовали вашему workflow и структуре вашего **Memory Bank**.
4. **Проверьте базовую работоспособность:**
   Убедитесь, что агент в вашем кастомном режиме корректно интерпретирует ваши правила и понимает команды, связанные с **Memory Bank** (например, *"обнови **Memory Bank**"*), пытаясь взаимодействовать с ним согласно заданному flow.

Пример описания flow работы с банком памяти в проекте автотестов Афиши .

3. Работа с Memory Bank

После того как настроен flow работы с **Memory Bank** (используется стандартный или кастомный), можно начинать с ним работать.
Процесс включает **инициализацию**, **использование** для генерации кода и последующую **поддержку**.

3.1 Инициализация и первичное наполнение

Для **первоначального наполнения ("обучения") Memory Bank** под конкретный тип задач подготовьте набор **репрезентативных примеров** , результат выполнения которых вы сможете легко проверить.
В случае с генерацией автотестов этим набором может быть несколько простых и типовых тест-кейсов.

Рекомендуется начинать с тестов, для которых уже есть необходимые хелперы и Page Objects, чтобы агент меньше "придумывал" и больше учился на существующих сущностях.

Пошаговый процесс первичного наполнения:

1. **Запуск инициализации:** Запустите диалог с агентом в нужном режиме (например, `AQA Mode` или `architect`, если используется стандартный банк) и инициируйте процесс создания/проверки **Memory Bank**. Как именно это сделать, должно быть описано в правилах вашего агента (часто достаточно команды `hello` или подобной).
2. **Первичное сканирование (если предусмотрено правилами):** Некоторые flow предполагают автоматическое сканирование проекта командой типа `update memory bank` для сбора базовой информации о хелперах, фикстурах и т.д.
3. **Целевое обучение (Итеративно через задачи):**
   - Поставьте агенту задачу из подготовленного набора (например, `"Напиши тест для сценария X, используя фикстуру Y и хелпер Z."`).
   - **Валидация и Корректировка:** Проверьте сгенерированный код. Укажите агенту на ошибки, предложите правильные варианты использования сущностей проекта.
   - **Обновление Памяти: Обязательный шаг!** После корректировки убедитесь, что агент обновил **Memory Bank** (либо дайте команду `update memory bank`, если он не сделал это сам), чтобы запомнить правильный подход и контекст.
   - **Итерации:** Повторяйте шаги генерации-корректировки-обновления для нескольких типовых задач. Если замечаете, что агент плохо наполняет **Memory Bank** или возникают другие проблемы именно с процессом работы **Memory Bank** (memory bank flow), то исправляйте **файл правил** (`rules.yaml`) и пробуйте выполнить шаги по инициализации заново.

> ⚠️ **Критически важно: Всегда проверяйте содержимое Memory Bank** (папка `memory-bank` или та которая описана в вашем flow) после команд обновления, особенно на этапе инициализации. Неверная или неполная информация приведет к ошибкам в будущем.

Пример наполненного Memory Bank в автотестах Афиши

3.2 Использование (генерация кода и обновление)

После инициализации и первичного наполнения **Memory Bank** готов к **повседневному использованию**.

Пошаговый процесс работы:

1. **Запуск:** Выберите режим работы, для которого настроен **Memory Bank** (например, `AQA Mode`).
2. **Генерация Кода:** Поставьте задачу агенту (например, `"Напиши тест для сценария [...]."`). Агент должен использовать информацию из **Memory Bank** для генерации более релевантного и соответствующего стандартам проекта кода.
3. **Валидация и Корректировка:** Проверьте результат, при необходимости укажите на ошибки или предложите улучшения.
4. **Обновление Памяти:** Если в процессе диалога появилась новая полезная информация (например, вы объяснили агенту новый паттерн или использование API), попросите агента обновить **Memory Bank**, чтобы он запомнил это для будущих задач.
5. **Валидация Обновления:** Кратко проверьте, что агент корректно добавил информацию в файлы **Memory Bank**.

3.3 Поддержка и актуализация

Memory Bank — это **"живой" артефакт**, требующий регулярной поддержки и актуализации, так же как и код вашего проекта.

- **Добавление нового:** При появлении новых общих хелперов, фикстур, базовых классов или устоявшихся паттернов — обучайте агента и обновляйте **Memory Bank**.
- **Обновление существующего:** Если изменяются сигнатуры функций, логика базовых компонентов или Page Objects, отраженных в памяти, — обновите соответствующие записи в **Memory Bank**.
- **Удаление устаревшего:** Периодически проводите ревизию **Memory Bank** и удаляйте информацию об устаревших или более не используемых сущностях и паттернах.
- **Рефакторинг:** При значительных изменениях в структуре проекта или тестового фреймворка может потребоваться рефакторинг не только кода, но и структуры/содержимого **Memory Bank**.

> 📌 **Помните:** Актуальность **Memory Bank** напрямую влияет на качество работы LLM-агента. Устаревшая информация в памяти может приводить к генерации некорректного кода.

Ревью Memory Bank: Файлы банка памяти лежат в репозитории, соответственно будут попадать в ревью. Крайне важно относиться к ревью этих файлов также серьезно как к документации проекта.

3.4 Общие рекомендации (Советы)

- **Итеративность:** Не пытайтесь наполнить **Memory Bank** сразу всем. Начинайте с основного (базовые классы, ключевые хелперы/фикстуры) и постепенно добавляйте детали по мере необходимости.
- **Качество важнее количества:** Лучше иметь в памяти меньше информации, но точной и релевантной, чем много устаревших или неверных примеров. Если есть легаси-код, который не должен служить примером, не используйте его для обучения агента.
- **Указывайте на важное:** При обучении агента явно акцентируйте его внимание на ключевых аспектах: конкретные строки кода, важные параметры функций, правильные способы обработки ошибок и т.д. (эта информация должна сохраняться в **Memory Bank**).

- **Структуризация:** Когда файлов в **Memory Bank** станет много, используйте подпапки или попросите LLM помочь реструктурировать банк для лучшей организации.
- **Консистентность:** Следите, чтобы информация в **Memory Bank** была консистентной и не противоречила сама себе.

4. Дополнительные правила для агента (User Rules)

Memory Bank отвечает за **контекст проекта** (что в нем есть, как это используется). Однако, помимо этого, агенту нужно **задать общие правила поведения** и требования к результату его работы (стиль кода, структура тестов, обязательные элементы и т.д.).

Эти правила (часто называемые "User Rules" или "Custom Instructions") также прописываются в **файле правил кастомного режима** (`rules.yaml`), создание которого описано в разделе `2.2.1` . Они могут включать:

- Требования к стилю кода (использование линтеров, форматеров: black, isort, flake8).
- Структура тестов (именование, использование классов/функций, паттерн AAA, Gherkin-комментарии).
- Обязательное использование определенных фикстур/хелперов для конкретных действий.
- Правила логирования, обработки исключений.
- Требования к комментариям и документации.
- Язык ответа агента ("Всегда отвечай пользователю на русском").

Пример в котором указано что нужно избегать комментариев кода и отвечать на русском.

5. Q/A (Вопросы и Ответы)

- **Q:** А **Memory Bank** вообще полезен и работает?
  A: Да, посмотреть это можно на примерах которые показывались на встрече по агентам.
- **Q:** Агент не использует нужную фикстуру/хелпер.
  A:
  1. Проверьте **Memory Bank** (наличие, корректность записи - см. разделы `3.1` , `3.3`).
  2. Проверьте правила **Custom Mode** (`rules.yaml` , см. раздел `4`): нет ли конфликтов, правильно ли указано использовать эту сущность?
  3. Укажите на сущность явно в запросе при постановке задачи.
- **Q:** Агент генерирует код в старом стиле / использует устаревший паттерн.
  A:
  1. Убедитесь, что новые примеры/паттерны есть в **Memory Bank** (`3.1` , `3.3`), а старые удалены или помечены как устаревшие.
  2. Проверьте и обновите User Rules (`4`), явно указав на новый стиль/паттерн.
  3. Явно укажите на новый паттерн в запросе при постановке задачи.
- **Q: Memory Bank** стал слишком большим/медленным.
  A:
  1. Удалите устаревшую и нерелевантную информацию (`3.3`).
  2. Реструктуризуйте (`3.4`).
  3. Уменьшите детализацию хранимой информации (например, храните только сигнатуры функций вместо полного кода, если это возможно).

[!] Добавьте другие актуальные для вас вопросы и ответы.

6. Заключение

Правильно настроенный LLM-агент (Roo Code), интегрированный в IDE с использованием **Custom Modes** и актуального **Memory Bank**, может стать мощным инструментом для команды автотестирования. Он помогает ускорить разработку тестов, повысить консистентность кода и служит базой знаний по используемым в проекте паттернам и сущностям.

Ключ к успеху — это не разовая настройка, а итеративный процесс:

- Тщательная первоначальная настройка правил агента.
- Внимательное наполнение **Memory Bank** релевантными примерами.
- Регулярная актуализация **Memory Bank** и правил по мере развития проекта.
- Включение **Memory Bank** в процесс код-ревью.

О сервисе Сообщество Конфиденциально © 2025 ООО «Яндекс» Infrastructure

написании качественных автотестов.