

Управление контекстом в агентной программной инженерии: модель жизненного цикла и паттерны проектирования

С. А. Чернышев

Аннотация – В работе рассматривается изменение роли контекста в программной инженерии, связанное с переходом от классического написания кода к агентной разработке с использованием больших языковых моделей. Показано, что в таких системах значительная часть управляющей логики переносится из исходного кода в инструкции, спецификации, проектные правила, внешнюю память, ограничения, сведения об инструментах и промежуточные результаты работы агента. Поэтому контекст предлагается рассматривать не как разовую подсказку, а как сопровождаемый инженерный артефакт, влияющий на поведение создаваемой системы. В работе анализируются существующие подходы к использованию контекста в LLM- и агентных системах, включая внешние источники знаний, инструментальные интерфейсы, проектные инструкции, правила времени выполнения, агентские навыки и механизмы памяти. На основе проведенного анализа предложена модель жизненного цикла контекста, включающая формирование, проверку и оценку, распространение и повторное использование, наблюдение за использованием, а также сопровождение. Дополнительно вводится структура контекста $K = (S, A, I, V, Q, P)$, позволяющая фиксировать область действия, условия применения, содержание, проверяемые ограничения, критерии качества и происхождение.

Работоспособность модели проверена экспериментально на 240 прогонах кодирующих агентов двух моделей: сильной облачной `deepseek-v4.1-flash` и слабой локальной `qwen3.6-35b-a3b-mtp` на собственном `opensource` проекте со смоделированным дрейфом API, построенном по схеме единственного носителя, при котором конвенции дрейфа невыводимы из репозитория и единственным их носителем является испытываемый контекст, который после предложения предлагаемого подхода дал прирост успешности и сильному ($0,50 \rightarrow 0,93$) и слабому агенту ($0,45 \rightarrow 0,88$). Что же касается границы применимости, то без использования хуков (один из механизмов `harness` для перехвата и обработки событий) контекст компенсирует лишь дефицит знания агента, но никак не дефицит его исполнительской надежности. Другими словами, всякие «НИКОГДА», «НЕ СМЕЙ» и т.д., могут быть и вовсе проигнорированы агентом, тогда как хук детерминировано отработает в нужный момент (запуск тестов, проверка того, можно ли дать агенту доступ в данную директорию и многое другое).

Последний же результат работы заключается в том, что было предложено расширить существующую классификацию паттернов проектирования мультиагентных систем, предложенную автором в своих предыдущих работах, и добавить в нее группу контекстных паттернов проектирования, позволяющих повысить переиспользуемость, проверяемость, безопасность и управляемость контекстом в агентной разработке.

Ключевые слова – агентная разработка, контекстная инженерия, большие языковые модели, ИИ-агенты, жизненный цикл контекста, контекстные паттерны проектирования, программная инженерия.

I. ВВЕДЕНИЕ

Рывок в развитии больших языковых моделей, произошедший за последние пару лет, очень сильно отразился на некоторых ранее устоявшихся аспектах программной инженерии. Если раньше мы оперировали абстракциями, которые выражались в коде: функции, объекты (классы), модули и т.д., то с появлением в 2021 году GitHub Copilot в качестве базовой абстракции, используемой в процессе разработки стала выступать «задача» (написать класс, функцию, произвести рефакторинг файла, документировать выделенный код и т.д.), что ставилась ИИ-ассистенту. А с 2023 года, с появлением более автономных агентов для кодирования, способных работать над поставленной задачей на протяжении нескольких и более часов, разработка программного обеспечения перешла на уровень **абстракции намерений**, где базовым элементом стала выступать «цель» – какую проблему решаем, что для этого необходимо сделать, какие критерии завершения работы и т. д. На этом уровне абстракции программист на естественном языке или с помощью декларативных структур (JSON, XML и т.д.) формулирует для агента высокоуровневую цель. Сам же агент выступает в роли интеллектуального «компилятора», способного самостоятельно переводить заложенное в цель намерение в некоторую последовательность действий, закладывая план и динамически декомпозируя задачи на подзадачи, «на лету» генерируя программные модули,

Статья получена 1 июня 2026, после доработки 06.10.2026

Чернышев Станислав Андреевич, кандидат технических наук, доцент кафедры прикладной информатики, Санкт-Петербургский государственный университет аэрокосмического приборостроения, Институт информационных технологий и программирования, Санкт-Петербург, Россия (email: chernyshev.s.a@bk.ru).

выполняя их в песочнице и производя самокоррекцию за счет получаемой от среды или в последствии выполнения одного из шагов плана (например, запустить линтер и убедиться, что нет ошибок и предупреждений) обратной связи (agent runtime feedback). Это позволило нам уйти от самостоятельного написания кода реализации и проверки каждой его строчки (Code Review) к аудиту доказательств (тесты, логи и т.д.) и обоснованию решений, которые подтверждают качество работы агента.

Другими словами, если раньше основная управляющая логика выражалась нами через код, то с появлением агентных систем для кодирования, все большую долю поведения разрабатываемой системы задают инструкции, спецификации, примеры, внешняя память, правила доступа к инструментам, ограничения и фрагменты знаний, помещаемые в окно контекста агента. Ведь именно правильно подобранный контекст (и грамотно выстроенная контекстная инженерия в рамках агента) влияет на выбор действия, порядок использования инструментов, соблюдение архитектурных границ, расход токенов, устойчивость к ошибкам и безопасность выполнения.

Отсюда и вопрос: «А не стоит ли нам сместить фокус и начать рассматривать контекст как новый вид кода, либо сам программный код, как производную от контекста?». С этой точки зрения контекст должен начать рассматриваться не как разовая подсказка агенту, а как управляемый инженерный артефакт, так как от его качества напрямую зависит поведение создаваемой системы. Более того, часть действий, которые раньше приходилось жестко описывать в программе через множество условий и вспомогательных сценариев, теперь, с добавлением ИИ-модулей, может быть оформлено как повторно используемые инструкции, навыки или спецификации в виде обычного текста. Однако, такое перенесение части функциональности системы из кода в контекст допустимо только тогда, когда эти инструкции можно проверять, обновлять, версионировать и контролировать так же строго, как обычный программный код.

Цель данной статьи – рассмотреть возможность работы с контекстом, как с инженерным артефактом. А для этого требуется определить его структуру, стадии жизненного цикла, какие показатели качества необходимо контролировать, а также какие паттерны проектирования позволят сделать контекст переиспользуемым, проверяемым и безопасным.

II. ОБЗОР СУЩЕСТВУЮЩИХ ПОДХОДОВ

Среди существующих работ по использованию больших языковых моделей (LLM, БЯМ) в задачах разработки программных продуктов наибольший интерес представляют статьи, в которых рассматривается роль контекста в управлении поведением ИИ-систем или агентов. Сюда входят такие статьи, где: контекст может дополнять знания модели, задавать интерфейс действия, фиксировать правила проекта, масштабироваться, превращаться в проверяемые ограничения, оформляться как повторно используемая процедура или становиться частью эксплуатационного состояния агента.

Исходной точкой в смене парадигмы разработки программных продуктов с использованием ИИ-агентов

стали работы, в которых доказано, что качество ответа зависит не столько от того, насколько «умная» модель была выбрана, а от того, какие сведения из вне и операции в рамках окружающей среды ей доступны. Так, например, в работе [1] показано, что языковая модель лучше справляется с задачами, требующими специальных знаний, если перед ответом получает релевантные фрагменты документов из внешнего источника. В работе посвященной ReAct-подходу (Reasoning + Acting) продемонстрировано, что LLM может чередовать рассуждения и действия, если среда предоставляет ей доступ к инструментам и наблюдениям [2]. В работе [3] представлена модель Toolformer, способная определять какие внешние инструменты, с какими аргументами и когда лучше всего вызывать. А это значит, что обращение к внешним инструментам может рассматриваться, как часть управляемого поведения модели. Конечно, эти работы не решают проблему сопровождения контекста как артефакта, но задают очень важный сдвиг – управление результатом переносится из слоя настройки модели в область проектирования информационной и инструментальной среды.

Особенно отчетливо этот сдвиг проявился в программной инженерии. SWE-bench сделал объектом оценки не изолированную генерацию функции, классов или модулей, а решение реальных задач в репозиториях, где агент должен учитывать множество файлов, историю изменений, тесты и соглашения в проекте [4]. SWE-agent, в свою очередь, показал, что эффективность работы агента зависит от специально спроектированного интерфейса работы с операционной системой компьютера: навигации по репозиторию, редактирования файлов, запуска команд и интерпретации результатов [5]. Отсюда следует, что контекст и управление им намного важнее простого текста запроса. Он должен включать в себя состояние репозитория, доступные команды, протоколы взаимодействия со средой, промежуточные результаты выполнения и ограничения на допустимые действия. Как результат, эти работы положили начало целой череде разработки агентских harness (исполнительных сред) для кодирования (Claude Code, Codex, OpenCode, Pi и т.д.).

Одним из ответов разработчиков на необходимость сохранять проектные правила стали файлы инструкций уровня репозитория или, как их еще называют, проектные инструкции – AGENTS.md, предложенный OpenAI и аналогичные документы. Их назначение состоит в том, чтобы сделать доступными для агента такие неочевидные моменты, как: архитектурные соглашения, команды сборки, правила тестирования, существующие ограничения в проекте и многое другое. Однако, эмпирические результаты по таким файлам неоднозначны. В работе [6] продемонстрировано, что наличие в репозитории проекта файла AGENTS.md может приводить к снижению медианного времени выполнения задачи и увеличению расхода выходных токенов. В работе [7] на примерах таких бенчмарков, как SWE-BENCH LITE и AGENTBENCH показано, что такие файлы могут увеличивать число шагов, повышать стоимость использования LLM при оплате токенов по API и давать лишь небольшой или отрицательный вклад в успешность выполнения задачи. Следует отметить, что это касается моментов, когда они «раздуты» или

содержат требования, плохо согласованные с решаемой в текущий момент времени задачей. Отсюда следует вывод, что не во всех случаях контекст, формируемый агентской средой исполнения, можно считать полезным. Как и программный код, он требует проектирования, проверки, минимизации, сопровождения и оценки влияния на поведение системы.

Следующая проблема, связанная с контекстом – его масштабирование. Используемый в небольших или средней сложности проектах AGENTS.md становится очень перегруженным в репозиториях крупных систем. В результате этого агенту становится сложнее выделить из общего текста именно те сведения, которые относятся к текущей задаче. В работе [8], на примере эмпирического исследования более 400 open-source-проектов, показано, что такие файлы не имеют устоявшейся структуры, а содержание заметно различается от проекта к проекту. Что же касается проблемы перегруженности единственного AGENTS.md файла, то это в основном решается тем, что знания о проекте разносятся по нескольким файлам с проектными инструкциями. А это порождает вопрос о том, как без потери качества и согласованности таких документов поместить все сведения об архитектуре, правилах разработки, принятых соглашениях, командах проверки, ограничениях модулей и т.д. Другими словами, при росте проекта контекст начинает вести себя как кодовая база – его приходится декомпозировать, задавать границы области действия, сопровождать версии, устранять противоречия и проверять влияние изменений.

К сожалению, для обеспечения повторяемости работы ИИ-агента недостаточно просто зафиксировать правила проекта в текстовом файле инструкций. Это связано с недетерминированной природой LLM из-за которой они могут по-разному интерпретировать проектные правила, игнорировать их, следовать локальным примерам из кодовой базы, противоречащим общим требованиям и многое другое. В работе [9] предложен подход, где ограничения поведения агентов задаются в виде структурированных правил времени выполнения, которые срабатывают при определенных действиях в момент его работы, что запускает анализ сложившейся ситуации. По завершению анализа производимое действие может быть принято, отклонено, либо эскалировано на уровень пользователя, чтобы он сам принял решение. Другим подходом может выступать правильная организация обратной связи для агента, стадий проверки в момент выполнения задачи посредством хуков агентской среды разработки и четко заданные границы интерфейсов реализуемых модулей для выстраивания полноценного тестового окружения. Что в одном, что в другом случае не стоит говорить о 100% повторяемости кода и повторяемости кода вообще. С учетом хорошо выстроенного тестового контура, разработчика может и не волновать, как был написан сам код модуля, главное – все unit и интеграционные тесты проходят по основным и граничным сценариям. С другой стороны, оба этих подхода вместе с SWE-bench [5] позволяют рассматривать контекст не только как обычную текстовую инструкцию для LLM, но и как источник проверяемых ограничений, влияющих на повторяемость и управляемость процессом работы агентной системы.

Что же касается процедурных знаний, то самой многообещающей формой их представления являются агентские навыки (Agents Skills), предложенные компанией Anthropic. Любой навык представляет собой повторно используемые единицы процедурного знания, включающие условие применимости, политику выполнения, условие завершения и интерфейс вызова [10]. В отличие от одноразового плана или текстовой подсказки, агентский навык задает устойчивую последовательность действий в повторяющемся классе ситуаций. Однако и у этого инструмента агентного программирования есть свои проблемы. Так, например, представленный в работе [11] бенчмарк SWE-Skills-Bench показал, что агентские навыки – не панацея на все случаи жизни. Они могут не давать существенного прироста для успешного завершения решаемой агентом задачи, либо ухудшать результат его работы, из-за противоречивости с проектным контекстом. Поэтому агентские навыки лучше всего рассматривать как программный артефакт со своей областью действия, зависимостями, версией, эволюционными тестами для проверки успешности и критериями применимости.

Следующий аспект, на который стоит обратить внимание, связан с сохранением и обновлением контекста в ходе длительной работы ИИ-агента. Для таких задач он перестает быть статическим текстом, который передается модели перед началом работы. В ходе каждой эпохи агентского цикла контекст пополняется результатами предыдущих действий: ответами LLM, данными, полученными в ходе вызова инструментов, ошибками или сведениями, что могут понадобиться агенту позже. В работе [12] представлен механизм долговременной памяти – MemoryBank, который предполагает сохранение, извлечение и обновление сведений из прошлых взаимодействий агента со средой или пользователем, что должно позволить модели учитывать в процессе своей работы накопленный ранее опыт. Но данный подход применим только при работе пользователя с одним агентом. Для мультиагентных сценариев все куда сложнее – контекст может распределяться между различными ролями, этапами работы и сообщениями, которые обмениваются агенты. Один из вариантов организации контекста в таких сценариях предложен в работе [13], где на примере специализированной системы – ChatDev, предложен подход, в котором агенты участвуют в проектировании, программировании и тестировании программного продукта через управляемые цепочки коммуникации между агентами. Как итог, результат работы зависит не только от возможностей отдельных агентов, но и от того, какие данные передаются в процессе каждой стадии работы системы между ее агентами и как они сохраняются. Из этого следует, что в продолжительных и мультиагентных задачах контекст является не разовой подсказкой, а рабочей памятью процесса. Он пополняется по мере выполнения задачи, используется разными агентами и требует поддержки таких механизмов управления контекстом, как: сохранение, обновление, передача между агентами, а также удаление устаревших или ошибочных сведений.

Как видно из обзора, контекст в агентной разработке уже давно вышел за пределы обычной текстовой подсказки. Теперь он включает в себя: внешние знания,

информацию о доступных агенту инструментах, проектные инструкции, ограничения, агентские навыки и рабочую память агента. Также, следует отметить, что проводимые в данном направлении исследования пока остаются разрозненными и не задают единого инженерного подхода к проектированию, проверке, масштабированию и сопровождению контекста.

III. ПОСТАНОВКА ЗАДАЧИ И ИСХОДНЫЕ ПОЛОЖЕНИЯ

Под контекстом в данной работе будем понимать совокупность данных, которые непосредственно влияют на выбор действий ИИ-агента, передаваемые или доступные ему через вызов различных инструментов. Сюда входят: системные и проектные инструкции, архитектурные решения, соглашения о стиле, ограничения безопасности, команды сборки и тестирования, спецификации, фрагменты памяти, политики доступа к инструментам и состояние среды и многое другое. Главное, что следует отметить, в текущей статье мы фокусируемся на внешнем контексте, как артефакте, а не на внутренних механизмах работы с контекстом агентских сред исполнения.

Можно выделить 3 признака, чем контекст отличается и от обычной проектной документации:

1. Основным потребителем контекста является не человек, а агент, способный выполнить действие на основании текста или структуры.
2. Ошибка в контексте может приводить не к недопониманию, а к генерации неверного кода, нарушению архитектуры или небезопасной цепочки действий со стороны агентов.
3. Изменение контекста меняет поведение системы так же, как изменение исходного кода меняет поведение программы.

Ниже приведены основные положения, от которых автор отталкивался в ходе проведения своих исследований:

1. **Контекст должен быть версионизуемым.** Изменение инструкции, примера или спецификации должно иметь историю, автора, область действия и объяснение влияния. Без этого невозможно «воспроизвести» результат или понять, почему при одинаковой задаче новая агентная сессия отработала иначе.
2. **Контекст должен быть проверяемым.** Наличие инструкции не гарантирует того, что агент будет ее соблюдать в процессе решения поставленной пользователем задачи. Чтобы минимизировать такие риски необходимы различные статические проверки (например, запуск линтера), сценарные оценки, повторные прогоны и приемочные критерии. Другими словами, вокруг контекста необходимо выстраивать тестовый контур, как и при работе с программным кодом.
3. **Контекст должен быть распространяемым и изолируемым.** Повторно используемые агентские навыки, спецификации и политики ИИ-агента должны иметь простой формат для распространения с описанием их предназначения, зависимостей, структуры поставляемых файлов, в какой агентской среде могут выполняться и требуются ли проверки

перед использованием.

4. **Контекст должен наблюдаться во время эксплуатации агентной системы.** Логи агентной сессии, ошибки тестов, запущенных на этапах проверки реализуемого агентом программного продукта и данные о различных инцидентах могут содержать информацию о том, какого контекста не хватило для успешного выполнения поставленной задачи, какой оказался двусмысленным, а какой подтолкнул агента к лишнему действию. Организация такой обратной связи позволит постоянно отслеживать несоответствия в текущем контексте и вносить в него жизненно необходимые изменения.

IV. ПРЕДЛАГАЕМАЯ МОДЕЛЬ ЖИЗНЕННОГО ЦИКЛА КОНТЕКСТА

Предлагаемая модель жизненного цикла контекста состоит из пяти взаимосвязанных стадий: формирование контекста, его проверка и оценка, распространение и повторное использование, наблюдение за использованием, а также сопровождение и возврат к стадии формирования. Эти стадии образуют замкнутый контур, где полученный опыт в ходе работы ИИ-агента над поставленной задачей возвращается в контекст в виде уточненных правил, новых примеров, дополнительных ограничений или исправлений в сценарии оценки.

А. Формирование контекста

На стадии формирования контекста должны определяться данные, которые обязаны влиять на работу ИИ-агента. К ним можно отнести: цель задачи, ограничения предметной области, архитектурные правила проекта, соглашения по оформлению кода, команды проверки, примеры правильных и ошибочных решений и т.д. Главным результатом этой стадии должен стать не просто набор текста, а определение того, на кого адресован каждый из фрагментов формируемого контекста и какая у него должна быть область действия. Другими словами, на этой стадии для нас важно определить для какого агента, типа задачи, модуля или инструмента предназначен формируемый контекст и какое поведение в работе агента он должен изменить.

Само же качество формируемого контекста должно определяться его достаточностью и избирательностью. Так, например, недостаточный контекст оставляет агенту слишком много неявных допущений, а чрезмерный контекст перегружает рабочее окно модели и повышает риск того, что важные правила будут проигнорированы или вовсе забыты. Именно из-за недетерминированной природы LLM мы должны рассматривать контекст как некий артефакт со своим назначением, областью применения, источником и ожидаемым эффектом, начиная с первой стадии ЖЦ контекста.

В. Проверка и оценка контекста

Данную стадию проще всего разбить на 2 этапа. В рамках первого этапа осуществляется проверка таких формальных свойств контекста, как: корректность структуры, полнота обязательных полей, отсутствие противоречий, секретов, устаревших команд и ссылок на неподходящие версии библиотек или инструментов.

Такой вид контроля чем-то напоминает статическую проверку программного кода и позволяет обнаружить дефекты в контексте еще до того, как агент начнет выполнять задачу.

На втором этапе производится оценка влияния контекста на поведение агента. Для этого необходимо сформировать сценарий оценки, состоящий из: исходной задачи, подключаемого (формируемого на предыдущей стадии) контекста, ожидаемых действий агента и критерий приемки результата. Например, если контекст запрещает использовать определенную команду или требует запускать конкретную проверку, сценарий должен показать, соблюдает ли он это требование на практике. В идеале сценарий должен проверяться инструментами агентского harness для кодирования (хуки, логи сессий, запуск тестового окружения тестового проекта и т.д.), а в тех случаях, где результат нельзя надежно проверить обычным тестом – оценка должна опираться на анализ поведения агента другим агентом-критиком.

При таком стечении обстоятельств нельзя полностью полагаться на единичные ответы языковой модели при оценке влияния контекста на поведение агента. Поэтому предлагается ввести показатель устойчивости $p = m/k$, который показывает отношение того, на сколько запусков оценки (k) были получены удовлетворительные результаты (m). Точечная оценка p должна обязательно дополняться интервальной. Для этих целей следует использовать доверительный интервал Вильсона при уровне доверия 95 %. Так, например, результат 27 успешных прогонов из 30 дает точечную оценку 0,90 при интервальной $[0,74; 0,97]$, чего уже недостаточно для утверждения о достижении порога 0,95. Сама же интервальная форма нужна для предотвращения необоснованных выводов при малых объемах выборки и позволяет единообразно сопоставлять между собой разные версии контекста.

Выбор числа прогонов k не является произвольным, а зависит от требуемого порога устойчивости. Если в k независимых прогонах не зафиксировано ни одной неудачи, верхняя граница доверительного интервала уровня 95 % для доли неудач $\approx 3/k$ («правило трех»). Отсюда следует прямая связь порогов и объемов проверки: 30 бездефектных прогонов подтверждают $p \geq 0,9$; 60 – $p \geq 0,95$; 300 – $p \geq 0,99$. Отсюда следует, что для критичных сценариев диапазон должен составлять 0,9–1. Это связано с тем, что данный диапазон соответствует приемлемой стоимости проверки (30–60 прогонов), тогда как подтверждение более высоких порогов требует кратно большего числа прогонов. Для детерминированных проверок (линтер и тесты, запускаемые хуками самого harness) достаточно $k = 1$, поскольку их результат воспроизводим, а при граничных результатах прогоны имеет смысл продолжать до тех пор, пока нижняя граница доверительного интервала не превысит пороговое значение либо верхняя не окажется ниже него.

Для учета критичности сценариев предлагаются два взаимодополняющих друг друга подхода. Первый – разделить классы сценариев по критичности. Для каждого такого сценария задаются собственный порог p и минимальный объем прогонов (таблица 1). Критичные сценарии требуют $p \geq 0,9$ при $k \geq 30$, типовые – $p \geq 0,75$

при $k \geq 12$, а для вспомогательных порог не задается и фиксируется лишь достигнутое значение p с доверительным интервалом. Второй – рассчитывать сводный взвешенный показатель по набору сценариев j : $p_w = \sum w_j m_j / \sum w_j k_j$, где w_j – вес критичности сценария (например, 3 – безопасность и архитектурные запреты, 2 – ключевые рабочие процессы, 1 – вспомогательные сценарии). Пороговые проверки по классам следует применять при принятии решения о допуске версии контекста, а взвешенный показатель – для сводной оценки версии в целом.

Отдельной интерпретации требуют стохастические задачи, в которых даже корректный контекст не гарантирует успеха. Для них p следует трактовать не как абсолютную вероятность успеха, а как вероятность успеха при данном контексте и оценивать относительно эталонов. Для этих целей каждую неудачу необходимо отнести к одному из трех классов источников:

1. Дефект контекста (агенту не хватило сведений, инструкция двусмысленна или противоречива);
2. Ограничение модели или исполнительской среды, не зависящее от контекста;
3. Нестабильность самого сценария (нестабильные тесты, меняющееся окружение и т.д.).

Неудачи класса (1) образуют отдельную долю q_c и служат прямым входом нового витка жизненного цикла. Неудачи класса (3) устраняются стабилизацией сценария, а класс (2) задает «потолок» задачи $p_{\max}$, который невозможно устранить изменениями в контексте. Сам же вклад контекста изолируется сравнением с базовой конфигурацией p_{base} (минимальный или стандартный контекст): $\Delta p = p - p_{\text{base}}$. Контекст признается достаточным, если p статистически неотличим от $p_{\max}$ при малой доле q_c .

Таблица 1. Пороги показателя устойчивости и объемы прогонов для классов критичности сценариев

Класс сценариев	Примеры сценариев	Порог p	Объем прогонов k (правило трех)
K1 – критичные	запрет секретов в коде; схема базы данных только через миграции; запрет опасных команд	$\geq 0,9$ (цель 0,95)	30 (60); 3/30 = 0,1
K2 – типовые	сборка и тесты; стиль кода; обновление зависимостей	$\geq 0,75$	12; 3/12 = 0,25
K3 – вспомогательные	документация; генерация заготовок; типовой анализ	не задается	≥ 5 ; оценка версий

С. Распространение и повторное использование контекста

После проверки и оценки контекст оформляется в виде файла инструкций, пакета с навыком, набора спецификаций, записи в корпоративной базе знаний или в качестве конфигурации среды исполнения агента (агентского harness). На данной стадии необходимо фиксировать версию, автора контекста, его назначение, существующие зависимости, уровень доверия и т.д.

Распространение контекста принципиально отличается от копирования текста между задачами, так как при его повторном использовании важно понимать, совместимы ли все фрагменты контекста с конкретным проектом, не противоречит ли он локальным правилам, не требует ли дополнительных инструментов и не раскрывает ли сведения, которые не должны попадать в рабочее окно агента.

D. Наблюдение за использованием контекста

Несмотря на прохождения проверок полученный контекст может оказаться недостаточным для успешного решения агентом задачи или же приводить к ухудшению его в моменты реальной работы (т.е. вне тестового окружения). Именно на этот случай в модель жизненного цикла контекста и добавлена стадия наблюдения за тем, как и какие действия он осуществляет на основе переданного ему контекста. В данном случае источниками обратной связи служат журналы работы агента, замечания при проверке изменений, анализ неудачных запусков тестов с предыдущей стадии, повторяющиеся ошибки, инциденты в эксплуатации и случаи, когда агент явно сообщает о нехватке знаний.

Стадия наблюдения позволяет отвечать на такие практические вопросы, как:

- Какие инструкции были проигнорированы агентом?
- Какие данные и в какой момент времени в контексте агента слишком общие и не помогают выполнить задачу?
- Какие сведения постоянно приходится добавлять вручную?
- Какие фрагменты контекста вызывают лишние действия, рост затрат или неверные решения?
- и т.д.

Таким образом данные, получаемые при эксплуатации агента, превращаются в материал для улучшения контекста.

E. Сопровождение контекста и возврат к формированию контекста

После стадии наблюдения возможны два пути. Если агент устойчиво соблюдает правила, ошибки срабатывания остаются в пределах допустимых значений, контекст не добавляет агенту лишних действий и т.д., мы начинаем рассматривать его как проектный артефакт, который переходит в режим сопровождения. Если же на этапе наблюдения или в ходе последующего сопровождения выявлены проблемы, то начинается новый виток жизненного цикла. Повторяющаяся ошибка, неожиданное поведение агента, сбой теста или инцидент в эксплуатации оформляются в качестве входных данных для стадии формирования контекста, которые позволяют понять: что на самом деле произошло с агентом, какой фрагмент контекста был затронут, какого знания агенту не хватило, какое правило оказалось двусмысленным и многое другое. После этого измененный фрагмент контекста снова последовательно проходит стадии формирование контекста, проверка и оценка, распространение новой версии и повторное наблюдение.

В таблице 2 представлены стадии предлагаемого жизненного цикла, а также основные артефакты контекста, которые должны на них формироваться и

подходить, посредством которых осуществляется контроль за качеством формируемого контекста.

Таблица 2. Предлагаемая модель жизненного цикла контекста

Стадии жизненного цикла	Основные артефакты	Контроль качества
Формирование контекста	Цель, правила проекта, примеры решений, спецификации, фрагменты памяти, описание среды работы агента	Проверка адресата, области действия, источника сведений, отсутствия дублирования и избыточности
Проверка и оценка контекста	Структурированные инструкции, проверяемые ограничения, сценарии оценки и приемочные критерии	Формат, полнота, непротиворечивость, отсутствие секретов, повторные прогоны сценариев
Распространение и повторное использование	Версионизируемый пакет контекста, реестр, зависимости, права доступа, сведения о происхождении	Совместимость, изоляция областей действия, уровень доверия, проверка зависимостей
Наблюдение за использованием	Журналы работы агента, замечания при проверке изменений, результаты тестов, сведения об инцидентах	Выявление невыполненных инструкций, недостающего контекста, устаревших и вредных правил
Сопровождение и возврат к формированию контекста	Журнал наблюдения, решение о стабильном сопровождении, либо перечень дефектов для нового витка жизненного цикла	Контроль актуальности и версий; при дефектах - повторный запуск цикла, начиная со стадии формирования контекста

В таблице ниже приведены разграничения стадий наблюдения, проверки и оценки, сформулированные в терминах среды исполнения, метрик, артефактов агентской сессии и т.д.:

Таблица 3. Разграничение стадий наблюдения, проверки и оценки использования контекста

Признак	Проверка и оценка	Наблюдение за использованием
Среда	Тестовое окружение, фиксированные сценарии оценки	Реальные задачи в эксплуатации
Момент	До публикации версии контекста	После публикации версии
Характер процесса	Дискретные контролируемые прогоны	Непрерывный сбор сигналов
Результат	Вердикт о допуске версии (значение p с доверительным интервалом)	Перечень дефектов и улучшений для нового витка цикла
Метрики	$p = m/k$ с доверительным интервалом, нарушения проверяемых ограничений и расход токенов на прогон	Частота игнорирования инструкций, доля сессий с ручным вмешательством, инциденты, дрейф правил
Артефакты	Сценарий оценки, протокол прогонов и вердикт о допуске	Контекстная трасса, журнал инцидентов и замечания при проверке изменений

V. КОНТЕКСТ КАК ИНЖЕНЕРНЫЙ АРТЕФАКТ: СТРУКТУРА И КРИТЕРИИ КАЧЕСТВА

Исходя из представленной модели жизненного цикла контекста, к нему нужно подходить, как к самостоятельному инженерному артефакту, который не зависит от конкретного формата хранения (Markdown-файла, YAML/JSON-спецификация, пакет навыка и т.д.) и может быть описан следующим образом:

$$K = (S, A, I, V, Q, P), \quad (1)$$

где S – область действия, A – условия подключения, I – содержательная часть контекста, V – проверяемые ограничения, Q – критерии качества, а P – сведения о происхождении.

Область действия S отвечает за то, к какой части системы относится контекст: всему проекту, отдельному модулю, директории, типу задачи, роли агента, инструменту или предметной области. Чем точнее задана область действия, тем меньше риск, что правило, предназначенное для одной части проекта, начнет ошибочно влиять на другую. Например, оптимизация запросов к базе данных приводит к изменениям пользовательского интерфейса, чего быть не должно.

Условия подключения A определяют, в какой момент должен использоваться текущий контекст. Одни правила или инструкции могут загружаться всегда, другие же только при изменении определенных файлов, активации агентского навыка, работе с отдельным инструментом и т.д. Благодаря такому подходу формируемый контекст не перегружает агента лишними сведениями, а несет только нужную для решения задачи информацию.

Содержательная часть I включает в себя те данные, которыми агент должен руководствоваться на протяжении всего времени своей работы: инструкции, проектные соглашения, примеры правильных решений, ссылки на спецификации, перечень допустимых и недопустимых действий.

Проверяемые ограничения V представляют собой контролируемый набор наиболее важных правил. В него входят: архитектурные запреты, правила именования, требования к тестированию, запрет опасных команд, различные проверки зависимостей между модулями, сценарии автоматической оценки результата и т.д. Данный элемент жизненно необходим для обеспечения повторяемости, так как переносит правила, которым должен следовать агент из обычных текстовых инструкций, кои могут быть проигнорированы, в слой проверок (запуск линтера или тестов, различные скрипты валидации, хуки агентского harness). Такой подход позволяет выявлять нарушения и возвращать контекст на доработку.

Критерии качества Q применяются для оценки того, насколько контекст пригоден для использования. Хороший контекст должен быть релевантным решаемой агентом задаче. А именно: достаточным, конкретным, непротиворечивым, проверяемым, безопасным и совместимым с используемыми инструментами. Что касается недостаточного или избыточного контекста, то они способны в значительной мере ухудшить качество работы агента, из-за чего он будет читать лишние файлы, запускать не те проверки, выбирать слишком слабую или даже замудренную стратегию решения задачи и т.д.

Сведения о происхождении P фиксируют, кто именно создал или изменил контекст, когда это было сделано, на основании какого источника, какие проверки прошел, насколько ему можно доверять и многое другое. Это особенно важно для промышленного использования контекста и позволит отследить, как и какие изменения в инструкциях, агентских навыках или спецификациях отразились на поведении разрабатываемой или сопровождаемой ИИ-системе.

VI. КОНТЕКСТНЫЕ ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

Существующие классификации паттернов проектирования мультиагентных систем в основном описывают архитектуру агентов, способы их взаимодействия, миграцию, защиту, координацию и внутреннюю организацию поведения [14, 15]. Однако, с появлением LLM и их активном использовании в разработке агентных или мультиагентных систем, контекст начал рассматриваться как самостоятельный слой в системе, который задает агенту сведения о проекте, правила работы, ограничения и т.д. Поэтому в существующую классификацию целесообразно добавить новую – контекстную группу паттернов проектирования.

Среди таких паттернов особенно выделяются: горячая и холодная память, агентские навыки, проверяемая инструкция, контекстный фильтр, контекстная оценка, контекстная трасса и бюджет контекста. Естественно, это не все контекстные паттерны проектирования и если еще рассматривать те, которые используются в самих агентских средах исполнения, задавая архитектуру и их поведение, то на это уйдет слишком много времени. Поэтому в рамках текущей статьи было принято решение сосредоточиться только на тех контекстных паттернах, которые отвечают за «внешний контекст», на который может повлиять обычный пользователь таких агентских сред

программирования (harness), как: OpenCode, Claude Code, Pi, Cursor и т.д.

А. Паттерн «Горячая память»/«Горячий контекст»

Все те данные, которые загружаются в контекст агента из файлов AGENTS.md (когда поддерживается средой, то и файлов правил) вне зависимости от поставленной агенту задачи и задают его поведение верхнего уровня для текущей сессии работы, можно рассматривать как «горячую» память агента. Это именно та память, на которую агент опирается при планировании своего первого шага на стратегию сессии и, конечно же, последующих шагов. Другими словами, когда необходимо предоставить агенту начальную информацию о проекте, правила поведения и написания кода, какие документы в каком случае следует прочитать перед выполнением задачи – эти данные следует помещать в файл AGENTS.md, либо связанные с ним при наличии матрицы маршрутизации.

Главный недостаток этого паттерна проектирования – постепенное разрастание ядра контекста и его устаревание. Если в AGENTS.md начинают добавлять все подряд, данный файл превращается в перегруженную инструкцию и мешает агенту выделять действительно важные правила. А если в нем находятся устаревшие сведения – это может запутать агента и повести выполнение задачи по ложному пути. Поэтому следует ограничивать объем AGENTS.md, вынося инструкции для агента в случае его разрастания в другие файлы (перенос в область «холодной» памяти) и проводить регулярную ревизию его содержания.

В. Паттерн «Холодная память (холодный контекст)» или «Спецификация по запросу»

Текущий паттерн проектирования предлагает выносить объемные данные о проекте (архитектура, описание модулей, принятые ограничения) или проектные инструкции в отдельные файлы, куда агент будет обращаться только по необходимости. Это позволяет рассматривать весь репозиторий, в котором работает агентская среда кодирования, как холодную память агента, а также снижает постоянную нагрузку на окно контекста и позволяет агенту использовать более точные сведения только тогда, когда они действительно нужны.

В зависимости от используемой агентской среды разработки еще можно формировать специализированные правила, которые всегда, либо при определенных условиях (редактирование файлов определенного расширения, обращение к конкретной директории и т.д.) автоматически будут добавлять данные в контекст агента.

Недостаток рассматриваемого паттерна проектирования связан с тем, что агент может не найти нужный документ, спецификацию или прочитать неподходящие для решения задачи данные. Поэтому такие документы должны иметь понятные названия, признаки применимости и связи с теми частями проекта, к которым они относятся.

С. Паттерн «Агентские навыки»

Данный паттерн проектирования был представлен компанией Anthropic в конце 2025 года и быстро завоевал популярность у инженеров. Он предлагает выносить процедурные знания агента, которые могут применяться

в различных условиях и переноситься из проекта в проект, в отдельный пакет. Такие навыки зачастую используются для выполнения агентом повторяющихся рабочих процессов: отладка, миграция, настройка CI/CD, проведения типового анализа и т.д. Другими словами, навык отличается от обычной инструкции тем, что задает воспроизводимый способ выполнения целого класса задач.

Согласно спецификации, навык должен содержать точку входа – файл SKILL.md (с двумя полями – имя и описание в YAML-блоке), содержащий последовательность действий, допустимые инструменты, критерии завершения и ожидаемый формат результата. В зависимости от сложности поставляемого или формируемого агентского навыка все его тело может поместиться как в одном файле SKILL.md, так и состоять из большего набора директорий и документов, куда могут входить шаблоны, сценарии, тесты и вспомогательные файлы.

На старте сессии агенту передаются только имена и описания навыков и после получения задачи от пользователя, он выбирает какой из них использовать для решения. В этот момент считываются данные SKILL.md, а если и этого мало, агент запрашивает чтение более специализированных документов агентского навыка. Такой механизм называется прогрессивным раскрытием и не требует загружать в «горячую память» агента все данные навыка.

Недостаток текущего паттерна проектирования состоит в том, что его можно подключить к нерелевантному проекту, где тока от него не будет от слова «совсем», навык может использовать более старые версии библиотек, нежели чем те, что используются в проекте, либо содержать старые, неактуальные данные по принятому процессу (последовательности шагов) решения задачи.

Д. Паттерн «Проверяемая инструкция»

Текущий паттерн решает проблему, когда вводимые правила поведения и следование им нельзя оставить на откуп ИИ-агента. Например, если в агент должен «использовать uv, а не pip» или «не изменять схему базы данных без миграции», такое правило желательно не только написать, но и проверить.

Само решение состоит в переводе важной текстовой инструкции в контролируемую форму: запуск статического анализатора, тестов, другого агента в качестве архитектурного-критика, перехват команд посредством хуков до и после их исполнения (например – вызов инструментов) с запуском сторонних средств автоматической проверки. Благодаря такому подходу инструкция перестает быть только рекомендацией и становится частью механизма контроля, способного повысить повторяемость поведения агента.

К недостаткам данного паттерна можно отнести то, что при слишком жестко заданных проверках плохо заданное ограничение может блокировать допустимые действия или создавать ложные срабатывания.

Е. Паттерн «Контекстный фильтр»

Агенты могут подгружать данные в контекст из различных источников (поиск в интернете, обращение через msp к внешним сервисам, работа с агентскими навыками из сомнительных источников), не все из

которых способны нести полезную для решения задачи информацию. А некоторые так и вообще могут содержать правила, приводящие к конфликтам в процессе работы, инъекции для доступа к секретам или небезопасные команды.

Одно из решений данной проблемы – добавить контекстный фильтр, который будет осуществлять проверку контекста до его загрузки в рабочую сессию агента. В идеале такой фильтр должен выявлять секреты, опасные команды, попытки навязать агенту нежелательное поведение, конфликт с политиками проекта и недостаточный уровень доверия к источнику. Его задача – не допустить, чтобы вредная или недостоверная информация стала частью рассуждения агента.

Г. Паттерн «Контекстная оценка»

Изменение даже одной инструкции в уже устоявшемся контексте может привести к совершенно другому пути решения задачи при следующем запуске агента. Это может быть другой маршрут поиска, запуск не тех команд проверки, больший расход токенов и т.д. Поэтому контекст, как и программный код, должен иметь свое тестовое окружение.

Паттерн «Контекстная оценка» предполагает создание набора сценариев, на которых оценивается влияние контекста. То есть для каждой важной инструкции задается ожидаемый эффект. Например, агент должен выбрать нужный инструмент, соблюсти правило проекта, не выполнить запрещенную команду или сгенерировать код с нужной структурой.

Критичные свойства контекста должны всегда подтверждаться синтаксическими, статическими, функциональными, интеграционными и эксплуатационными детерминированными проверками. Что касается оценкой другой языковой моделью, то она допустима только для смысловых критериев, когда мы описываем что ожидаем в качестве результата и предоставляем ей оценить – был ли он достигнут или нет.

Г. Паттерн «Контекстная трасса»

В ходе своей работы агентная система может совершать ошибки, неправильно вызывать инструменты, не активировать в нужные моменты агентские навыки, из раза в раз скатываться к одинаковым проблемам в ходе эксплуатации и многое другое. Чтобы бороться с такими неблагоприятными происшествиями необходимо реализовать полноценный слой наблюдения, который позволит отловить и преобразовать повторяющиеся проблемы в новое правило, проверку, спецификацию или навык.

Данный паттерн реализует не просто логгирование, а полноценную контекстную трассу для отслеживания любого шага агентской системы и как в ходе ее работы менялся контекст, передаваемый языковой модели. Чаще всего это JSONL-файл (JSON-файл, где каждая строка – свой отдельный JSON), который заводится на каждую агентную или субагентную сессию с такой структурой записи событий, чтобы можно было легко отследить в какие моменты и что пошло не так.

Следует отметить, что несмотря на свои достоинства, данный паттерн несет огромные риски для менее опытных инженеров. Так, например, бездумное изменение контекста из-за небольших, непроверяемых

или редко повторяемых ошибок вызванных недетерминизмом языковых моделей приведет к его загрязнению и деградации в работе агентной системы.

Н. Паттерн «Бюджет контекста»

Технически, в языковую модель можно загрузить большой объем сведений, но это далеко не всегда самое лучшее решение. Такой избыточный контекст способен отвлекать агента, снижая точность планирования последующих действий или увеличивать количество расходов на задачу токенов, а значит и стоимость его работы.

Данный паттерн проектирования рассматривает формируемые фрагменты контекста, который будет отправляться модели, по степени их важности:

1. Всегда подгружаемые фрагменты контекста;
2. Фрагменты, подключаемые по запросу (результаты вызова инструментов);
3. Фрагменты, хранимые в сжатом виде;
4. Фрагменты, наполнение которых было суммаризовано и представлено одним фрагментом, либо их наполнение заменено ссылкой на какой-либо документ, полученный в ходе работы агента;
5. Фрагменты, которые в ходе следующего шага будут исключены из контекста как устаревшие или вредные для него.

Следует отметить, что хороший контекст не всегда является максимально полным. Он должен быть достаточным для задачи и не перегружать агента нерелевантными сведениями.

VII. АПРОБАЦИЯ НА ПРАКТИЧЕСКОМ КЕЙСЕ

Работоспособность предложенной модели проверена на открытом инструментарии madharness-mini (<https://github.com/MADTeacher/madharness-mini>), реализующем проектные инструкции, слой контекста с бюджетом, агентские навыки, исполняемые хуки и трассы сессий в формате JSONL. Данный harness предоставляет пользователю минимально необходимый функционал кодингового агента и используется автором для обучения студентов основам harness-инженерии.

Для проведения серии экспериментов был выбран обучающий проект, разработанный компанией Mad Brain для книги «Основы Flutter» в соответствии с моими методическими рекомендациями (<https://github.com/MadBrains/Mad-Flutter-Practicum>) и, что самое главное – отсутствующим в обучающих выборках обеих моделей. Сам проект (lab_8 из лабораторного практикума) представляет собой приложение курсов валют и новостей, разработанный в соответствии с методологией Clean Architecture, в котором смоделирован дрейф API – обратно несовместимая эволюция репозитория (пагинация `fetchRates(CurrencyPageRequest)` с конвенцией `base: "USD"` при значении по умолчанию `"EUR"`, загрузка ленты `loadNewsFeed(NewsFeedFilter)` с семантикой кэш/сеть, чтение темы `readThemeMode()` с защитой инициализации). Некоторые прежние члены интерфейсов были помечены аннотацией устаревания `@Deprecated`, продолжают работать и используются в ходе приемки для фиксации факта того, исправил ли агент эту ситуацию. Для самих экспериментов были 2

модели: облачная `deepseek-v4.1-flash` (режим рассуждений отключен) и локальная `qwen3.6-35b-a3b-mtp` (режим рассуждений включен как свойство модели), инференс которой осуществлялся на AMD RYZEN AI MAX+ 395 (128 ГБ объединенной памяти) посредством LM Studio. Сравнивались же три конфигурации контекста разбитые по 8 подзадач в рамках внесения изменений в flutter-проект: C0 – агент без контекста; C1 – «сырой» контекст (AGENTS.md общего назначения содержащий структуру проекта, команды сборки, правило сопровождать изменения записями в журнале изменений); C2 – контекст после прохождения предложенного жизненного цикла. На каждую подзадачу выделялось по 5 прогонов ($k=5$), что вылилось в 120 запусков ($8*5*3$). В качестве оркестратора была выбрана модель GLM-5.3 (режим рассуждения high), запускаемая в харнесе ZCode, которая стартовала серии экспериментов и следила за тем, что инстансы `madharness-mini` (по факту агенты) отрабатывают в штатном режиме и каждое завершение их работы сопровождается явным запуском проверочных скриптов, собирающих статистику.

Далее под конвенцией будем понимать проектное соглашение о правильном использовании API, не выраженное в его сигнатуре. То есть нарушающий конвенцию код может быть скомпилирован и пройти тесты полигона, тогда как само нарушение будет зафиксировано в момент структурной проверки приемки.

Тестовое окружение для экспериментов было построено по схеме единственного носителя конвенций, то есть побочные носители знаний (документация, некоторые экраны как примеры, ряд тестов) были удалены из репозитория, а пометки `@Deprecated` намеренно не сообщали, чем заменены прежние члены. При таком подходе единственным носителем актуального знания о конвенциях дрейфа является испытываемый контекст. Чтобы это гарантировать дополнительно выполнялись статические (сканер утечек в кодовой базе по регулярным выражениям) и динамические (три агента без контекста решали самую конвенционно-зависимую и не справились) проверки. В противном случае проверялась бы наблюдательность агента, а не передача знаний контекстом, что было выявлено в ходе подготовки тестового полигона. Так, например, на изначальном проекте, содержащем дополнительные источники знаний в репозитории `deepseek` закрыла все задачи на 1, в то время как `qwen3.6-35b-a3b-mtp` при работе с `madharness-mini` завершила серию в 40 прогонов каждого варианта контекста с удручающими показателями, даже с учетом наличия дополнительного источника знаний в репозитории. Как оказалось дело было в одном флаге запуска инструмента `shell`, который `harness` предоставлял модели. Поскольку система изначально разрабатывалась на MacOS, она не учитывала специфику Windows на которой и проводился эксперимент, ввиду чего инструмент запуска команд работал без оболочки операционной системы, и на Windows обращение по имени «flutter build» (`uv run ...`, `python ...` или `dart ...`) не разрешались на уровне операционной системы (фактический исполнитель – `flutter.bat`). В ходе анализа трасс было выявлено, что сильная модель преодолевала эту особенность самостоятельно. То есть `deepseek` в той же самой среде

сам подобрал рабочую форму команды, тогда как ни один прогон слабой модели не содержал ни одного успешного вызова верификации, из-за чего агент расходовал ходы на неработающие варианты или заявлял проверку без фактического вывода. В итоге C2 для `qwen` = 0,35. Эволюция контекста, в которой были добавлены в AGENTS.md раздел «Среда исполнения» (исполнимые формы команд взятые из трассы `deepseek` и трактовка ошибок запуска) и ряд других улучшений позволила при повторном прогоне C2 улучшить показатель до 0,95 [0,83; 0,99] при снижении входных токенов с 819 до 512 тыс., среднего числа ходов с 38,1 до 25,2 и времени прогона с 1133 до 298 с. Именно эта эволюция контекста C2 и была в последствии использована при проведении основной серии экспериментов, после того как были убраны дополнительные контекстные знания из репозитория проекта и исправлена ошибка в `madharness-mini`. Другими словами, контекст C2 был структурирован в соответствии с $K = (S, A, I, V, Q, P)$, куда вошли: инструкции корневого уровня, отдельные AGENTS.md для каталогов слоев приложения, навык `migration-v2` с картой кода, конвенциями дрейфа и разделом «Среда исполнения» (исполнимые формы команд верификации), а также исполняемый хук, блокирующий записи с обращениями к `@Deprecated`-членам.

В первой серии экспериментов с `deepseek-v4.1-flash` на очищенном полигоне проверялось главное предположение – дает ли эволюция жизненного цикла контекста прирост там, где актуальное знание невозможно вывести из репозитория. Итоговая успешность C2 составила 0,93 [0,80; 0,97] против 0,50 [0,35; 0,65] у C0 и 0,57 [0,42; 0,71] у C1. Прирост к контролю +0,43, а на пяти задачах с конвенциями дрейфа (T1, T2, T4–T6) +0,52 (0,88 против 0,36). По результатам анализа трасс агентских сессий C0/C1 27 неудач отнесены к классу (1) (дефицит несообщенной конвенции, ожидаемый эффект схемы единственного носителя), у C2 данных неудач не наблюдалось. 13 неудач к классу (2) – исчерпание лимита ходов в циклах правок, досрочная сдача с красными тестами, неполная многочастная документация. Навык `migration-v2` активирован во всех 40 прогонах C2, а хук проверки обращения к `@Deprecated`-членам хоть и запускался каждый раз, но на деле так и понадобился.

В таблице 4 приведены результаты запуска тестового полигона с использованием `deepseek-v4.1-flash`:

Таблица 4. Результат запуска тестового полигона с большой облачной моделью deepseek-v4.1-flash

Задача (группа)	k	C0: m; p [ДИ]	C1: m; p [ДИ]	C2: m; p [ДИ]
T1 (курсы валют)	5	0; 0,00	0; 0,00	5; 1,00
T2 (курсы валют)	5	0; 0,00	0; 0,00	3; 0,60 [0,23; 0,88]
T3 (новости)	5	5; 1,00	5; 1,00	5; 1,00
T4 (новости)	5	0; 0,00	2; 0,40 [0,12; 0,77]	4; 0,80 [0,38; 0,96]
T5 (настройки)	5	4; 0,80 [0,38; 0,96]	5; 1,00	5; 1,00
T6 (настройки)	5	5; 1,00	5; 1,00	5; 1,00
T7 (смешанная)	5	2; 0,40 [0,12; 0,77]	1; 0,20 [0,04; 0,62]	5; 1,00
T8 (смешанная)	5	4; 0,80 [0,38; 0,96]	5; 1,00	5; 1,00
Конвенции v2 (T1, T2, T4–T6)	25	9; 0,36 [0,20; 0,55]	12; 0,48 [0,30; 0,67]	22; 0,88 [0,70; 0,96]
Итого	40	20; 0,50 [0,35; 0,65]	23; 0,57 [0,42; 0,71]	37; 0,93 [0,80; 0,97]

Во второй серии для запуска полигона использовалась локальная модель qwen3.6-35b-a3b-mtp. Прирост контекста воспроизводится практически без потерь: C2 = 0,88 [0,74; 0,95] против C0 = 0,45 [0,31; 0,60] и C1 = 0,38 [0,24; 0,53]. В ходе анализа трасс агентских сессий можно прийти к выводу, что текст без конвенций не помогает слабому агенту, тогда как сильный извлекал из него опоры (+0,07 в первой серии). В отличие от 40 неудач у deepseek, qwen допустила 52. Из них: 30 класса (1) – все у C0/C1, у C2 снова ноль; 5 неудач C2 (класс (2)) смещаются из знаний в исполнительскую дисциплину (в двух случаях конвенция применена семантически, но литеральная форма (forceRefresh: true / NewsFeedFilter.all) не воспроизведена); 3 – сессия завершилась досрочно (остановка с вопросом о разрешении верификации, пустой ответ модели, разрушение файлов при редактировании); 1 – у C0/C1 добавляются неполная многочастная документация изменений (фрагментов журнала изменений меньше, чем затронутых частей) и пропущенные импорты dart:async. Также было выявлено, что слабая модель самостоятельно не активирует агентские навыки (0 из 40 активаций в C1), тогда как при их явном указании вызов навыка может быть проигнорирован (в C2 навык migration-v2 активирован в 25 из 40 прогонов). Результаты данной серии прогонов тестового полигона приведены в таблице 5:

Таблица 5. Результат локального запуска тестового полигона с моделью qwen3.6-35b-a3b-mtp

Задача (группа)	k	C0: m; p [ДИ]	C1: m; p [ДИ]	C2: m; p [ДИ]
T1 (курсы валют)	5	0; 0,00	0; 0,00	5; 1,00
T2 (курсы валют)	5	0; 0,00	0; 0,00	5; 1,00
T3 (новости)	5	5; 1,00	5; 1,00	4; 0,80 [0,38; 0,96]
T4 (новости)	5	0; 0,00	0; 0,00	4; 0,80 [0,38; 0,96]
T5 (настройки)	5	5; 1,00	5; 1,00	5; 1,00
T6 (настройки)	5	4; 0,80 [0,38; 0,96]	5; 1,00	5; 1,00
T7 (смешанная)	5	3; 0,60 [0,23; 0,88]	0; 0,00	3; 0,60 [0,23; 0,88]
T8 (смешанная)	5	1; 0,20 [0,04; 0,62]	0; 0,00	4; 0,80 [0,38; 0,96]
Конвенции v2 (T1, T2, T4–T6)	25	9; 0,36 [0,20; 0,55]	10; 0,40 [0,23; 0,59]	24; 0,96 [0,80; 0,99]
Итого	40	18; 0,45 [0,31; 0,60]	15; 0,38 [0,24; 0,53]	35; 0,88 [0,74; 0,95]

В таблице 6 приведены данные о среднем количестве расходуемых ресурсов на каждый из вариантов прогона разработанного тестового полигона. Из них можно сделать вывод, что структурированный контекст способен заменять агенту самостоятельную разведку изменившегося репозитория, уменьшая количество требуемых для задачи токенов и времени работы (как и наоборот, плохой контекст способен приводить к перерасходу токенов и увеличению времени работы агента):

Таблица 6. Средние ресурсы на прогон по сериям и конфигурациям (исходящие токены qwen3.6-35b-a3b-mtp включают рассуждения режима thinking)

Серия	Вариант	Вх. токены (тыс.)	Исх. токены* (тыс.)	Ходы	Время, с
1 (deepseek)	C0	2317	7,1	49,2	147
1 (deepseek)	C1	2862	7,3	51,2	147
1 (deepseek)	C2	1474	5,8	40,6	113
2 (qwen)	C0	916	14,4	35,0	1401
2 (qwen)	C1	1046	15,6	38,0	1454
2 (qwen)	C2	645	13,6	28,0	1288

Следует отметить, что проведенные эксперименты ограничены одним инструментарием (madharness-mini и ZCode, как оркестратор эксперимента), двумя моделями легкого класса и малыми выборками (k = 5) с перекрывающимися доверительными интервалами. В виду чего приводятся только оценки и интервалы, без пороговых утверждений по «правилу трех». Большая облачная модель (deepseek) использовалась без режима рассуждений, тогда как для маленькой локальной модели (qwen) он был включен. Тем не менее проведенная серия экспериментов количественно подтверждает ключевые положения модели эволюции жизненного цикла контекста. А именно то, что жизненный цикл контекста дает прирост ровно там, где закрывает реальный, невыводимый из кода дефицит знания агента вне зависимости от типа выбранной модели.

Ознакомиться с трассами агентских сессий, оформлением задач и тем, как в целом было выстроено окружение проводимых серий экспериментов можно в следующем репозитории:

https://github.com/MADTeacher/context_evals_article

VIII. ОБСУЖДЕНИЕ РЕЗУЛЬТАТОВ

С переходом от обычного взаимодействия с языковыми моделями к построению агентных и мультиагентных систем на их основе, контекст перестает быть вспомогательным текстовым фрагментом к запросу пользователя и становится самостоятельным инженерным артефактом. Он задает то, какие знания доступны агенту в текущий момент времени, в рамках каких ограничений он работает и какие действия может совершать и многое другое. К сожалению, несмотря на такие возможности контекста и на то, что его изменение способно менять поведение агентной системы, результат такого изменения остается менее детерминированным, чем при изменении обычного программного кода.

Предложенная модель жизненного цикла, включающая стадии от формирования до сопровождения (и возврат к первой стадии при наличии выявленных проблем), позволяет рассматривать контекст как полноценный проектный артефакт. Это очень важно, поскольку произвольное накопление в репозитории разрабатываемого программного продукта проектных инструкций, спецификаций, агентских навыков и правил использования инструментов способно приводить к ухудшению качества работы агентных систем. В этом смысле идея «контекст как код» означает не буквальное тождество контекста и исходного кода программы, а необходимость применять к контексту инженерные практики контроля качества.

Другими словами, контекст должен быть достаточным для решения задачи, ограниченным по области действия, непротиворечивым и проверяемым. Как раз за это и отвечают предложенные контекстные паттерны проектирования мультиагентных систем: «горячая» и «холодная» память, агентские навыки, проверяемые инструкции, контекстные фильтры, трассы, оценки и бюджет контекста.

Экспериментальная апробация уточняет эти положения, варьируя два фактора – силу агента и выводимость знания из репозитория. Когда актуальное знание нельзя вывести из проекта и окружения, контекст после эволюции жизненного цикла становится единственным каналом его передачи (+0,43 у сильной, +0,42 у слабой). «Сырой» контекст в этих условиях бесполезен обеим моделям на конвенционных задачах (0,36–0,48 против 0,88–0,96 у C2), а слабой модели так и вовсе – делает только хуже (0,38 против 0,45). Замыкание жизненного цикла произошло в ходе подготовки к серии экспериментов, когда была найдена ошибка в madharness-mini у инструмента shell. Явное незнание среды исполнения (исполнимых форм команд) оказалось фатально для слабого агента (0,35) и незаметно для сильного, но нивелировалось эволюцией контекста по триажу трасс (0,35 → 0,95).

Верхнюю границу прироста успешности задает сам агент, а не контекст. Так, например, у слабой модели

неудачи C2 объясняются исполнительской дисциплиной (неполная документация изменений, литеральные формы, досрочные завершения сессий), а у сильной – исчерпанием лимита ходов. Исходя из этого можно предположить, что без использования специализированных хуков (один из механизмов harness для перехвата и обработки событий) контекст компенсирует только лишь дефицит знания, но никак не дефицит исполнительской надежности агента. Это отчасти подтверждает и подсчет активации агентских навыков агентами в ходе экспериментов (в C2 – 40 из 40 прогонов у сильной модели и 25 из 40 у слабой, хотя они явно упоминались в контексте).

Следует отметить, что предлагаемый в статье подход работы с контекстом имеет ряд естественных ограничений. Прежде всего, он относится к внешнему контексту, которым может управлять пользователь или разработчик агентной среды: инструкциям, спецификациям, навыкам, правилам подключения документов и проверкам. Внутренние механизмы агентских сред исполнения остаются за пределами рассмотрения. Это связано с тем, что чаще всего разработка таких систем ведется с использованием SDK или фреймворков, которые предоставляют уже готовые механизмы работы с внутренним контекстом, реализовав их посредством собственного видения паттерна «Бюджет контекста». С другой стороны, даже управление внешним контекстом позволяет повысить детерминированность агентных систем в рамках конкретного проекта.

IX. ЗАКЛЮЧЕНИЕ

В статье обсуждается, что при переходе к агентной разработке контекст становится одним из ключевых факторов, определяющих поведение агентных систем, построенных вокруг больших языковых моделей. Инструкции, спецификации, проектные правила, навыки, ограничения и сведения об инструментах начинают выполнять роль управляющего слоя, от которого зависит качество выполнения задач.

Основным результатом работы является модель жизненного цикла контекста, включающая формирование, проверку и оценку, распространение и повторное использование, наблюдение за использованием, а также сопровождение. Дополнительно была предложена структура контекста $K = (S, A, I, V, Q, P)$, позволяющая фиксировать область действия, условия применения, содержание, проверяемые ограничения, критерии качества и происхождение контекстного артефакта. Ее апробация в ходе серии экспериментов на 240 прогонов кодирующих агентов (по 120 на каждую из используемых моделей: deepseek-v4.1-flash и qwen3.6-35b-a3b-mtp) над проектом со смоделированным дрейфом API подтвердили ее работоспособность и уточнили границы применимости. Например, когда конвенции дрейфа невыводимы из репозитория, сообщить их агенту способен только контекст, после его эволюции жизненного цикла. В виду чего полученный прирост на задачах средней сложности может и вовсе не зависеть от силы модели (0,93 [0,80; 0,97] у C2 для deepseek и (0,88) [0,74; 0,95] для qwen), в

то время как «сырой» контекст бесполезен. Что же касается границы применимости, то без использования хуков (один из механизмов harness для перехвата и обработки событий) контекст компенсирует лишь дефицит знания агента, но никак не дефицит его исполнительской надежности. Другими словами, всякие «НИКОГДА», «НЕ СМЕЙ» и т.д., могут быть и вовсе проигнорированы агентом, тогда как хук детерминировано отработает в нужный момент (запуск тестов, проверка того, можно ли дать агенту доступ в данную директорию и многое другое).

Последний же результат работы заключается в том, что было предложено расширить существующую классификацию паттернов проектирования мультиагентных систем, предложенную автором в своих предыдущих работах, и добавить в нее группу контекстных паттернов проектирования.

Таким образом, контекст в агентной разработке следует рассматривать не как набор подсказок, а как сопровождаемый инженерный артефакт. Он должен быть достаточным, проверяемым, совместимым с проектом, ограниченным по области действия и наблюдаемым в эксплуатации. Только при таких условиях контекст может служить надежным средством управления поведением агентной системы.

БИБЛИОГРАФИЯ

- [1] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S. and Kiela, D. (2020) "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", *Advances in Neural Information Processing Systems (NeurIPS 2020)*, 33, pp. 9459–9474.
- [2] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R. and Cao, Y. (2023) "ReAct: Synergizing reasoning and acting in language models", *11th International Conference on Learning Representations (ICLR 2023)*, doi: 10.48550/arXiv.2210.03629.
- [3] Schick, T., Dwivedi-Yu, J., Dessì, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N. and Scialom, T. (2023) "Toolformer: Language models can teach themselves to use tools", *Advances in Neural Information Processing Systems*, 36, pp. 68539–68551. doi: 10.52202/075280-2997.
- [4] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. and Narasimhan, K.R. (2024) "SWE-bench: Can language models resolve real-world GitHub issues?", *12th International Conference on Learning Representations (ICLR 2024)*, doi: 10.48550/arXiv.2310.06770.
- [5] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K.R. and Press, O. (2024) "SWE-agent: Agent-computer interfaces enable automated software engineering", *Advances in Neural Information Processing Systems*, 37, pp. 50528–50652. doi: 10.52202/079017-1601.
- [6] Lulla, J.L., Mohsenimofidi, S., Galster, M., Zhang, J.M., Baltes, S. and Treude, C. (2026) "On the impact of AGENTS.md files on the efficiency of AI coding agents", in *Proceedings of Journal Ahead Workshop (JAWs) 2026 (ICSE JAWs 2026)*, New York: ACM, 5 p. arXiv:2601.20404, doi: 10.48550/arXiv.2601.20404.
- [7] Gloaguen, T., Mündler, N., Mueller, M.N., Raychev, V. and Vechev, M. (2026) "Evaluating AGENTS.md: Are repository-level context files helpful for coding agents?", in *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, doi: 10.48550/arXiv.2602.11988.
- [8] Mohsenimofidi, S., Galster, M., Treude, C. and Baltes, S. (2026) "Context engineering for AI agents in open-source software", in *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*, Rio de Janeiro, Brazil, 13–14 April 2026. New York: ACM, 5 p. arXiv:2510.21413. doi: 10.48550/arXiv.2510.21413.
- [9] Sharma, R.K. (2026) "ContextCov: Deriving and enforcing executable constraints from agent instruction files", arXiv:2603.00822, doi: 10.48550/arXiv.2603.00822.
- [10] Wang, H., Poskitt, C.M. and Sun, J. (2026) "AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents", in *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, Rio de Janeiro, Brazil, 12–18 April 2026. New York: ACM, 12 p. arXiv:2503.18666. doi: 10.48550/arXiv.2503.18666.
- [11] Han, T., Zhang, Y., Song, W., Fang, C., Chen, Z., Sun, Y. and Hu, L. (2026) "SWE-Skills-Bench: Do agent skills actually help in real-world software engineering?", arXiv:2603.15401, doi: 10.48550/arXiv.2603.15401.
- [12] Zhong, W., Guo, L., Gao, Q., Ye, H. and Wang, Y. (2024) "MemoryBank: Enhancing large language models with long-term memory", *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17), pp. 19724–19731. doi: 10.1609/aaai.v38i17.29946.
- [13] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z. and Sun, M. (2024) "ChatDev: Communicative agents for software development", in Ku, L.-W., Martins, A. and Srikumar, V. (eds.) *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Bangkok, Thailand, 11–16 August 2024. Stroudsburg, PA: Association for Computational Linguistics, pp. 15174–15186. doi: 10.18653/v1/2024.acl-long.810.
- [14] Чернышев С.А. Классификация общих шаблонов проектирования мультиагентных систем // Программные продукты и системы. 2022. Т. 35, № 4. С. 670–679. DOI: 10.15827/0236-235X.140.670-679
- [15] Чернышев С.А. Общие программные шаблоны проектирования мультиагентных систем: монография. Москва: РУСАЙНС, 2022. 134 с.

Context Management in Agentic Software Engineering: A Lifecycle Model and Design Patterns

S. A. Chernyshev

Abstract – The article examines the changing role of context in software engineering associated with the transition from traditional code writing to agentic development based on large language models. It is shown that, in such systems, a significant part of the control logic is transferred from source code to instructions, specifications, project rules, external memory, constraints, information about available tools, and intermediate results of the agent's work. Therefore, context is proposed to be considered not as a one-time prompt, but as a maintainable engineering artifact that affects the behavior of the system being developed. The paper analyzes existing approaches to the use of context in LLM-based and agentic systems, including external knowledge sources, tool interfaces, project instructions, runtime rules, agent skills, and memory mechanisms. Based on this analysis, a context lifecycle model is proposed, including formation, verification and evaluation, distribution and reuse, usage monitoring, and maintenance. In addition, the paper introduces the structure of context $K = (S, A, I, V, Q, P)$, which makes it possible to define the scope, conditions of application, content, verifiable constraints, quality criteria, and provenance of a contextual artifact. The article also describes contextual design patterns aimed at improving the reusability, verifiability, safety, and controllability of context in agentic development.

The model was validated experimentally in 240 runs of coding agents built on two LLMs: the strong cloud-based deepseek-v4.1-flash and the weak local qwen3.6-35b-a3b-mtp. The runs used the author's own open-source project with a simulated API drift, designed under the single-carrier scheme, in which drift conventions are underivable from the repository and the context under test is their only carrier. After applying the proposed approach, this context increased the success rate of both the strong ($0.50 \rightarrow 0.93$) and the weak agent ($0.45 \rightarrow 0.88$). As for the applicability boundary, without hooks (a harness mechanism for intercepting and handling events) context compensates only for the agent's knowledge deficit and in no way for its lack of execution reliability. In other words, instructions such as "NEVER" or "DO NOT" may be ignored by the agent altogether, whereas a hook fires deterministically at the right moment (running tests, checking whether the agent should be given access to a particular directory, and much more).

Finally, the study proposes extending the author's previously developed classification of design patterns for multi-agent systems by adding a group of context design patterns aimed at improving the reusability, verifiability, security, and manageability of context in agent-based software development.

Keywords – agentic development; context engineering; large language models; AI agents; context lifecycle; contextual design patterns; software engineering.

REFERENCES

- [1] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S. and Kiela, D. (2020) "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks", *Advances in Neural Information Processing Systems (NeurIPS 2020)*, 33, pp. 9459–9474.
- [2] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K.R. and Cao, Y. (2023) "ReAct: Synergizing reasoning and acting in language models", *11th International Conference on Learning Representations (ICLR 2023)*. doi: 10.48550/arXiv.2210.03629.
- [3] Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Hambro, E., Zettlemoyer, L., Cancedda, N. and Scialom, T. (2023) "Toolformer: Language models can teach themselves to use tools", *Advances in Neural Information Processing Systems*, 36, pp. 68539–68551. doi: 10.52202/075280-2997.
- [4] Jimenez, C.E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O. and Narasimhan, K.R. (2024) "SWE-bench: Can language models resolve real-world GitHub issues?", *12th International Conference on Learning Representations (ICLR 2024)*. doi: 10.48550/arXiv.2310.06770.
- [5] Yang, J., Jimenez, C.E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K.R. and Press, O. (2024) "SWE-agent: Agent-computer interfaces enable automated software engineering", *Advances in Neural Information Processing Systems*, 37, pp. 50528–50652. doi: 10.52202/079017-1601.
- [6] Lulla, J.L., Mohsenimofidi, S., Galster, M., Zhang, J.M., Baltes, S. and Treude, C. (2026) "On the impact of AGENTS.md files on the efficiency of AI coding agents", in *Proceedings of Journal Ahead Workshop (JAWs) 2026 (ICSE JAWs 2026)*. New York: ACM, 5 p. arXiv:2601.20404, doi: 10.48550/arXiv.2601.20404.
- [7] Gloaguen, T., Mündler, N., Mueller, M.N., Raychev, V. and Vechev, M. (2026) "Evaluating AGENTS.md: Are repository-level context files helpful for coding agents?", in *ICLR 2026 Workshop on Memory for LLM-Based Agentic Systems*, doi: 10.48550/arXiv.2602.11988.
- [8] Mohsenimofidi, S., Galster, M., Treude, C. and Baltes, S. (2026) "Context engineering for AI agents in open-source software", in *Proceedings of the 23rd International Conference on Mining Software Repositories (MSR '26)*, Rio de Janeiro, Brazil, 13–14 April 2026. New York: ACM, 5 p. arXiv:2510.21413. doi: 10.48550/arXiv.2510.21413.
- [9] Sharma, R.K. (2026) "ContextCov: Deriving and enforcing executable constraints from agent instruction files", arXiv:2603.00822, doi: 10.48550/arXiv.2603.00822.
- [10] Wang, H., Poskitt, C.M. and Sun, J. (2026) "AgentSpec: Customizable runtime enforcement for safe and reliable LLM agents", in *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26)*, Rio de Janeiro, Brazil, 12–18 April 2026. New York: ACM, 12 p. arXiv:2503.18666. doi: 10.48550/arXiv.2503.18666.
- [11] Han, T., Zhang, Y., Song, W., Fang, C., Chen, Z., Sun, Y. and Hu, L. (2026) "SWE-Skills-Bench: Do agent skills actually help in real-world software engineering?", arXiv:2603.15401, doi: 10.48550/arXiv.2603.15401.
- [12] Zhong, W., Guo, L., Gao, Q., Ye, H. and Wang, Y. (2024) "MemoryBank: Enhancing large language models with long-term memory", *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(17), pp. 19724–19731. doi: 10.1609/aaai.v38i17.29946.
- [13] Qian, C., Liu, W., Liu, H., Chen, N., Dang, Y., Li, J., Yang, C., Chen, W., Su, Y., Cong, X., Xu, J., Li, D., Liu, Z. and Sun, M. (2024) "ChatDev: Communicative agents for software development", in Ku,

- L.-W., Martins, A. and Srikumar, V. (eds.) Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), Bangkok, Thailand, 11–16 August 2024. Stroudsburg, PA: Association for Computational Linguistics, pp. 15174–15186. doi: 10.18653/v1/2024.acl-long.810.
- [14] Chernyshev S.A. Classification of common design patterns for multi-agent systems. *Software & Systems*, 2022, vol. 35, no. 4, pp. 670–679 (in Russ.). DOI: 10.15827/0236-235X.140.670-679.
- [15] Chernyshev, S.A. (2022) *Common Software Design Patterns for Multi-Agent Systems*. St. Petersburg, 155 p. (in Russ.)