Qwen

Создание автономного ИИ-агента для управления репозиторием: от архитектуры **LangGraph** до безопасной интеграции с **Git**

Архитектурная основа: От простого вызова инструмента к управляемому агентскому графу

Для перехода от ограниченно работающего агента, способного лишь читать и писать в существующих файлах, к системе полной автономии требуется фундаментальный пересмотр архитектуры. Первоначальная модель, вероятно, основана на парадигме ReAct (Reasoning and Acting), где агент последовательно выполняет цепочку «мысль-действие-наблюдение» [77]. Хотя эта парадигма является мощным стартовым пунктом, она не обеспечивает необходимого уровня контроля, предсказуемости и сложности для решения задачи по самостоятельному построению структуры проекта. Ключевым элементом для достижения поставленной цели является фреймворк **LangGraph**, который предоставляет низкоуровневую оркестровку для построения состоятельных, многоконтактных и контролируемых агентских систем [9] [46]. В отличие от стандартного потока в LangChain, LangGraph позволяет определять явные графы выполнения, что кардинально меняет подход к проектированию агентов [73].

Центральной концепцией LangGraph является граф, состоящий из узлов и ребер [9]. Узел в графе может представлять собой любой исполняемый блок кода: вызов большого языкового модели (LLM), вызов инструмента или даже пользовательскую функцию обновления состояния. Это позволяет декомпозировать сложную задачу на более мелкие, управляемые шаги. Например, для создания нового файла агент должен выполнить последовательность действий: проверить существование родительской директории, создать ее при необходимости, а затем записать в нее файл. В LangGraph это будет представлено как граф с несколькими узлами, где поток

управления строго регулируется логикой, определенной в ребрах графа [25]. Такой подход обеспечивает предсказуемость и возможность вмешательства или прерывания процесса на любом этапе [129]. LangGraph строится на идее, что надежность агентских систем — это прежде всего свойство их архитектуры [57]. Он позволяет создавать рабочие процессы, которые могут быть итеративными, условными и даже циклическими, что невозможно в рамках линейных цепочек LangChain [19].

Основой для управления сложностью в LangGraph служит понятие состояния. Графовый объект хранит состояние, которое является общим для всех узлов и модифицируется ими по мере выполнения [68]. Этот шарен-стате становится центральным хранилищем информации о текущем прогрессе агента, истории его действий, результатах выполнения инструментов, ошибках и контексте проекта [18]. Например, после успешного выполнения инструмента `create_directory`, результат этого выполнения (например, сообщение об успехе или путь созданной директории) добавляется в состояние. В следующем шаге LLM или другой инструмент получает доступ к этому результату и может принять решение продолжить работу, например, создав файл в только что созданной директории. Эта явная передача состояния через граф предотвращает побочные эффекты и делает поведение агента более прозрачным и воспроизводимым [68]. Для поддержки производственных приложений особенно важно использование внешних хранилищ состояний, таких как Amazon DynamoDB или MongoDB, которые обеспечивают долгосрочное хранение и возможность возобновления сеансов работы после сбоев [18] [21] [111].

Реализация расширенного туллинга в LangGraph требует правильной организации инструментов. Каждый инструмент, будь то операция с файловой системой или команда Git, должен быть представлен как отдельный узел в графе [106]. LangGraph предлагает удобный `ToolNode`, который автоматически обрабатывает вызовы нескольких инструментов [7] [106]. Однако для более сложных сценариев, когда требуется дополнительная логика между вызовами инструментов или обработка ошибок, необходимо создавать пользовательские узлы. Эти узлы могут выполнять любую логику, например, анализ вывода предыдущего инструмента и принятие решения о следующем шаге. Важно отметить, что сам инструмент должен вызываться в контексте выполнения LangGraph, чтобы корректно работала передача состояния [6]. Ребра графа определяют, какой узел будет выполнен следующим. Они могут быть статическими (например, "после `tool_node` всегда выполняется `llm_node`")

или динамическими, где логика выбора следующего узла определяется содержимым состояния после выполнения предыдущего шага [33]. Это позволяет реализовать сложные алгоритмы, такие как проверка условия перед выполнением действия или повторная попытка выполнения инструмента в случае ошибки.

Модульность является еще одним ключевым принципом при проектировании архитектуры. Система должна быть разделена на логические компоненты, каждый из которых отвечает за свою зону ответственности. Пример такой структуры можно найти в проектах типа `git-commit-agent`, которые имеют четкое разделение на папки для промтов (`prompts/`), инструментов (`tools/`) и утилит (`utils/`) [29]. Для нашей задачи можно выделить следующие модули: 1. Модуль парсера конвенций: Отвечает за загрузку и разбор Markdown-файла с архитектурными правилами. 2. Модуль инструментов **FS:** Набор высокоуровневых, безопасных инструментов для взаимодействия с файловой системой. 3. Модуль инструментов **Git:** Инструмент для взаимодействия с системой контроля версий Git. 4. Модуль управления состоянием: Логика для инициализации, обновления и сохранения состояния агента. 5. Система обработки ошибок: Модуль, который анализирует ошибки, возвращаемые инструментами, и решает, как действовать дальше (например, переформулировать запрос, сообщить пользователю или прекратить выполнение) [65] [100].

Выбор между одноразовыми и многоагентными системами также важен. Хотя LangGraph хорошо подходит для построения как одиночных, так и многоагентных систем [73], для задачи построения проекта начальная реализация, скорее всего, потребует одного мощного агента, которому предоставлены все необходимые инструменты и контекст. В более сложных сценариях можно рассмотреть разделение задачи между несколькими агентами (например, один агент отвечает за построение структуры, другой — за генерацию кода, третий — за коммиты), но это значительно усложняет координацию [32] [86]. Таким образом, архитектурная основа для решения поставленной задачи закладывается на базе LangGraph, используя его возможности для построения графов, управления состоянием и декомпозиции логики на модульные узлы, что обеспечивает необходимый уровень контроля и предсказуемости для достижения полной автономности.

Безопасное управление файловой системой: Стратегии изоляции и строгой типизации

Запрос на "полностью автономное" управление файловой системой ставит во главу угла вопрос безопасности. Прямой доступ к файловой системе через инструменты, вызываемые LLM, представляет собой одну из самых серьезных угроз в архитектуре агентских систем [89] [122]. Ошибка в инструкции, неверная интерпретация модели или намеренная манипуляция могут привести к катастрофическим последствиям: от удаления критически важных файлов (`rm -rf /`) до внедрения вредоносного кода или утечки конфиденциальных данных [39]. Поэтому реализация расширенного туллинга для файловой системы должна быть основана на принципах максимальной изоляции и строгой проверки. Наиболее надежным подходом является комбинация двух стратегий: полная изоляция среды выполнения («песочница») и предоставление агенту строго типизированного API вместо прямого доступа к файловой системе.

Первая стратегия, «песочница», заключается в запуске агента в полностью изолированном окружении, где его возможности по взаимодействию с системой строго ограничены. Это самый распространенный метод для выполнения ненадежного или потенциально опасного кода [27]. Вместо того чтобы давать агенту доступ к целой файловой системе хоста, он работает внутри своего собственного изолированного пространства. Примеры реализаций такого подхода варьируются от полноценных виртуальных машин до легковесных контейнеров и изолированных интерпретаторов. Сервис AgentBay, например, предоставляет такие сэндвичи на базе VM, где каждому сеансу агента выделяется отдельная виртуальная машина с приватной файловой системой и сетью [39]. Любая попытка агента совершить вредоносное действие, например, команду `rm -fr /`, будет строго ограничена пределами этой изолированной файловой системы, и реальная файловая система хоста останется нетронутой [39]. Другой пример — Cursor AI, который использует изолированные терминалы для своих агентов, лишая их доступа в интернет и изолируя их от основной системы [55]. Для Node.js экосистемы можно использовать Docker для создания временных контейнеров для каждого сеанса работы агента. Агент будет иметь доступ только к тому, что было смонтировано внутрь контейнера (например, исходный код проекта), и все его операции будут происходить в этой изолированной среде. Это обеспечивает первоочередную защиту от выхода за пределы разрешенной области.

Вторая стратегия, «строгая типизация API», направлена на снижение поверхности атаки на уровне самого агента. Вместо того чтобы позволять LLM генерировать произвольные пути и команды, мы создаем высокоуровневый, абстрактный набор инструментов, каждый из которых имеет четко определенную функциональность и строго типизированные параметры. Например, вместо инструмента, который принимает строку с произвольным путем, лучше создать несколько специализированных инструментов: `create_directory(path: string)`, `write_file(path: string, content: string, overwrite: boolean = false)` и `read_file(path: string)`. Функциональность этих инструментов должна быть реализована в виде серверных процессов, которые могут безопасно выполнять операции, но сами инструменты предоставляются LLM в виде вызываемых функций [71]. Для определения интерфейса этих инструментов идеально подходит библиотека Zod, которая является частью эко-системы популярных фреймворков, таких как TRPC [60]. Zod позволяет создавать схемы данных на TypeScript, которые гарантируют, что агент сможет сгенерировать только запросы с корректными типами и форматами данных. Проект `git-commit-agent` является отличным примером применения этого подхода: он использует Zod для валидации входящих параметров для команд Git, что предотвращает передачу некорректных или потенциально опасных аргументов [29]. Этот подход значительно сужает возможности агента, заставляя его действовать в рамках заранее определенных правил, и делает его поведение предсказуемым.

Наиболее надежным решением является комбинированный подход, объединяющий обе стратегии. Мы создаем безопасное, строго типизированное ядро инструментов (стратегия 2) и исполняем его в изолированной среде (стратегия 1). Логика такого подхода выглядит следующим образом: 1. Агент (LLM) генерирует вызов высокоуровневого инструмента, например, `create_directory(path="/project/src/components")`. 2. Этот вызов поступает в серверную часть, которая сначала проводит строгую валидацию параметров с помощью Zod. Она проверяет, что `path` является строкой и соответствует определенным правилам (например, не содержит `../`). 3. Если валидация пройдена, серверная часть преобразует этот высокоуровневый запрос в низкоуровневую операцию, например, `fs.mkdirSync('/sandbox/project/src/components', { recursive: true })`. 4. Эта низкоуровневая операция выполняется внутри изолированного сэндвича (Docker-контейнера).

Этот двойной уровень защиты обеспечивает максимальную надежность. Первый уровень (валидация схем) защищает от ошибок и манипуляций со стороны

самого агента. Второй уровень (изоляция среды) защищает всю систему от потенциальных уязвимостей в самом коде инструментов или от ошибок в логике валидации. Кроме того, для повышения безопасности в Node.js стоит использовать встроенные механизмы ограничения прав, такие как флаг `--permission`, который позволяет выбиратьочно предоставлять или запрещать доступ к файловой системе, сети и другим чувствительным ресурсам [122]. Также крайне важно использовать `npm ci` вместо `npm install` для установки зависимостей, чтобы строго следовать файлу блокировки (`package-lock.json`) и предотвратить подмену пакетов [122]. Регулярный аудит зависимостей с помощью `npm audit` также является обязательной мерой для снижения рисков, связанных с цепочками поставок [123].

Подход	Принцип работы	Преимущества	Недостатки
Прямой доступ	LLM получает возможность напрямую вызывать функции `fs` модуля Node.js.	Простота реализации.	Высочайший риск безопасности. Полная уязвимость системы.
Строго типизированный **API**	LLM взаимодействует с набором высокоуровневых, безопасных инструментов, интерфейсы которых строго валидируются (например, с помощью Zod).	Значительно снижает поверхность атаки. Поведение агента предсказуемо.	Требует усилий на проектирование и реализацию безопасных инструментов. Не защищает от ошибок в логике инструментов.
Песочница **(Sandboxing)**	Агент выполняется в изолированной среде (например, Docker-контейнер или виртуальная машина) с ограниченным доступом к файловой системе хоста.	Защищает хост-систему от любого вредоносного действия.	Добавляет накладные расходы на производительность и сложность развертывания. Может быть сложно настроить.
Комбинированный подход	Комбинация строго типизированного API и песочницы. Бэкенд-сервис валидирует запросы, а затем выполняет безопасные операции в изолированной среде.	Максимальный уровень безопасности. Двойная защита.	Самая сложная в реализации и эксплуатации.

Таким образом, для реализации автономного управления файловой системой необходимо отказаться от мысли о предоставлении агенту неограниченных прав. Вместо этого следует построить многоуровневую систему защиты, основанную на комбинации строгой типизации интерфейсов инструментов и их исполнения в изолированной, "песочничной" среде. Это единственный подход, который позволяет достичь желаемого уровня автономности, не жертвуя при этом безопасностью системы.

Интерпретация архитектурных конвенций из **Markdown-**файла

Ключевым требованием является способность агента строить проект на основе внешнего Markdown-файла с архитектурными конвенциями. Это требует от системы не просто механического выполнения команд, а глубокого понимания и применения семантического контекста, заложенного в этом документе. Процесс реализации этой функциональности можно разбить на три основных этапа: загрузка и парсинг Markdown-файла, преобразование его содержимого в структурированный формат, и, наконец, использование этой структурированной информации для руководства действиями агента.

Первый этап — загрузка и парсинг. Агент должен быть способен прочитать содержимое указанного Markdown-файла. Поскольку Markdown является текстовым форматом, его можно считать стандартным способом чтения файла. Однако для программной обработки сырых текстовых данных недостаточно. Необходимо преобразовать Markdown-документ в структурированный формат, такой как AST (Abstract Syntax Tree) или JSON. Это позволит агенту и его инструментам легко навигировать по документу, находить нужные разделы и извлекать информацию. Для парсинга Markdown в JavaScript/TypeScript существует множество зрелых библиотек, таких как `markdown-it`, `remark` или `commonmark` [62] [81]. Использование AST дает наибольшую гибкость, поскольку он представляет документ в виде дерева узлов, где каждый узел соответствует элементу Markdown (заголовок, параграф, список, таблица и т.д.). Это открывает возможности для не только чтения, но и модификации файла конвенций, если в будущем это потребуется. После получения AST его можно преобразовать в более удобный для LLM формат, например, в JSON-объект, содержащий ключевые правила проекта.

Второй этап — формирование контекста для **LLM**. Простое передача всего JSON-объекта, полученного из Markdown, в качестве контекста может быть неэффективной, так как это может привести к превышению лимита токенов модели и снижению качества ответа. Вместо этого, информация из конвенций должна быть агрегирована и представлена в сжатом, но информативном виде. Это можно сделать несколькими способами. Во-первых, можно создать краткую сводку документа, выделяя самые важные моменты. Например, из раздела "Project Structure" можно сгенерировать строку: "Проект должен иметь следующую структуру: `/src` для исходного кода, `/tests` для тестов, `/docs` для документации". Во-вторых, ключевые правила можно представить в виде

таблицы или списка. Например, для "Code Style" можно создать таблицу с примерами именования переменных, классов и функций [86]. Такой подход, известный как "offloading context to the sandbox", позволяет использовать небольшой набор атомарных инструментов (<20), а остальную сложность делегировать файловой системе или другим сервисам [26]. Цель состоит в том, чтобы предоставить LLM только ту информацию, которая необходима для принятия решения на данном конкретном шаге, а не весь объем данных сразу [86]. Это повышает эффективность использования контекста и снижает вероятность "промывки мозгов" моделью.

Третий и самый важный этап — связь с инструментами и принятие решений. Спарсенная и структурированная информация о конвенциях становится частью состояния агента и используется им для планирования и выполнения действий. Когда агент получает задачу, например, "создать новый компонент для страницы профиля пользователя", он обращается к своему контексту. Из раздела "Project Structure" он узнает, что компоненты должны располагаться в `/src/components`. Из раздела "Naming Conventions" он понимает, что имена компонентов должны быть в PascalCase. Из "File Naming" — что файлы компонентов должны иметь суффикс `.tsx`. На основе этого синтезированного знания агент может самостоятельно сгенерировать правильный вызов инструмента: `create_directory('/src/components')` (если директория не существует) и `write_file('/src/components/UserProfileCard.tsx', '...content...')`.

Подход GitHub Spec Kit, который использует живые документы для определения технических решений, является отличным источником вдохновения для этой задачи [85]. Он вводит концепцию `constitution.md` — файла, который устанавливает непреложные принципы проекта. Аналогично, наш Markdown-файл с конвенциями может служить "конституцией" для агента. Более того, этот подход можно расширить для определения границ поведения агента, что является критически важным для его безопасности и предсказуемости. Можно выделить три уровня границ, как это рекомендуется в исследованиях GitHub [86]:

- **'Always do' (**Всегда делать**):** Действия, которые агент должен выполнять автоматически без запроса. Например, "перед коммитом запускать тесты" или "использовать Prettier для форматирования кода".

- '**Ask first**' (Спросить первым): Высокоimpactful изменения, которые требуют человеческого одобрения. Например, "изменение базы данных" или "добавление новой основной зависимости".
- '**Never do**' (Никогда не делать): Жесткие запреты, которые служат "красными линиями". Например, "никогда не редактировать файл `node_modules/`", "никогда не коммитить файлы с секретами" или "никогда не удалять корневую директорию проекта".

Эти границы также можно определить прямо в Markdown-файле, делая их частью конфигурации агента. Это позволяет централизованно управлять поведением агента, изменяя только один файл, и обеспечивает согласованность его действий на протяжении всего жизненного цикла проекта. Таким образом, интерпретация Markdown-файла — это не просто чтение данных, а сложный процесс семантического анализа, который превращает статический документ в динамическое руководство для автономного агента.

Интеграция **Git** как первого класса инструмента для агента

Интеграция Git в качестве одного из ключевых инструментов для ИИ-агента является важным аспектом, позволяющим автоматизировать управление версиями и обеспечить трассируемость всех действий агента. Запрос на интеграцию "в идеале да, работа с гит как с тулами" предполагает, что агент должен не просто знать команды Git, а уметь вызывать их как функции, получать результаты и на их основе принимать дальнейшие решения. Это превращает Git из внешнего инструмента, которым пользуется человек, в активного участника агентского рабочего процесса. Для реализации этой функциональности существуют два основных подхода: создание набора специализированных инструментов или разработка единого универсального инструмента.

Подход с набором специализированных инструментов предполагает создание отдельного инструмента для каждой основной команды Git, которую должен выполнять агент. Например, можно определить инструменты `git_init()`, `git_add(paths: list)`, `git_commit(message: str)` и `git_checkout(branch_name: str)`. Такой подход является более декомпозированным и соответствует принципу единственной ответственности.

Каждый инструмент имеет четкую и простую цель. Однако у него есть и недостатки. Во-первых, он увеличивает количество инструментов, которые модель LLM должна помнить и правильно использовать. Это может усложнить обучение и повысить вероятность ошибочных вызовов. Во-вторых, он менее гибок. Если потребуется выполнить более сложную последовательность действий, это может потребовать от LLM больше шагов и сложнее планирования.

Более элегантным и часто рекомендуемым подходом является создание единого мастер-инструмента. Этот подход был продемонстрирован в проекте `git-commit-agent` [29] [55]. Вместо множества инструментов, агенту предоставляется один мощный инструмент, например, `execute_git_command(command: str, args: dict)`. Этот инструмент принимает название команды (например, `'status'`, `'add'`, `'commit'`) и словарь с её аргументами. Такой подход значительно упрощает взаимодействие с моделью: ей нужно помнить лишь один инструмент. Логика распределения задач между различными командами Git теперь находится внутри самого инструмента. Его реализация должна включать валидацию принимаемых команд и аргументов. Например, инструмент может содержать белый список допустимых команд (`['status', 'add', 'commit', 'checkout']`) и проверять, что LLM запросил одну из них. Это обеспечивает высокий уровень безопасности, предотвращая выполнение произвольных или потенциально опасных команд.

Независимо от выбранного подхода, ключевым аспектом является обеспечение безопасности и надежности инструмента для работы с Git. Проект `git-commit-agent` является эталоном для этого благодаря нескольким важным особенностям. Во-первых, он использует Zod для строгой валидации схемы входных параметров инструмента, что гарантирует, что агент сгенерирует только корректные запросы [29]. Во-вторых, в самом инструменте реализована логика для проверки окружения: он убедится, что выполняется внутри действительного Git-репозитория, и откажется в противном случае. В-третьих, он включает специальную логику для фильтрации и блокировки потенциально опасных команд, даже если они есть в списке разрешенных [29]. Например, команда `push` может быть разрешена только с определенными флагами или в определенных условиях. Такой подход, когда безопасность встраивается в сам инструмент, а не полагается на способности модели распознать опасность, является наиболее надежным.

Другой важной составляющей является обработка вывода команд **Git**. Агент должен быть способен "читать" результаты выполнения команд, чтобы понимать состояние репозитория. Например, после того как агент выполнил команду `git add .`, ему нужен результат этой команды. На самом деле, `git add` обычно не выводит ничего в стандартный поток вывода в случае успеха. Гораздо более ценной информацией является вывод команды `git status --porcelain`, который показывает, какие файлы были добавлены в индекс. Результат выполнения любой команды Git (стандартный вывод, стандартный поток ошибок и код завершения) становится частью контекста для следующего шага планирования агента. LLM может проанализировать этот вывод и принять решение продолжить выполнение следующей части плана, например, выполнить `git commit`. Этот цикл "выполнить команду -> проанализировать результат -> спланировать следующий шаг" является основой работы агента с Git. Примеры таких агентов, как `gut`, демонстрируют, как можно построить систему, ориентированную именно на эти взаимодействия с Git [16].

Наконец, интеграция с Git должна быть неотъемлемой частью общей архитектуры агента. Это означает, что все операции с Git должны быть реализованы как инструменты LangGraph. История выполнения Git-команд, вместе с их результатами и ошибками, должна храниться в состоянии графа [121]. Это обеспечивает полную прослеживаемость и позволяет агенту или человеку-наблюдателю отслеживать, какие шаги были предприняты и почему. В некоторых случаях может быть полезно инициировать работу агента из Git-репозитория, используя его как точку входа. Например, агент может быть запущен с указанием имени ветки или номера задачи, что позволит ему работать в рамках изолированной среды, созданной для этой задачи. Вся эта деятельность, от инициализации репозитория (`git init`) до создания коммитов и веток, должна рассматриваться как часть единого, управляемого графового процесса, а не как отдельные, разрозненные скрипты.

Обеспечение автономности, контроля и воспроизводимости

Достижение "полной автономности" не должно интерпретироваться как полное освобождение агента от контроля. Напротив, настоящая автономность в производственных системах достигается за счет сочетания предсказуемости,

безопасности и наличия механизмов контроля. Для обеспечения согласованности, воспроизводимости и надежности действий агента необходимо внедрить ряд стратегий и механизмов, которые будут управлять его поведением, логировать его действия и предоставлять человеку возможность вмешательства при необходимости.

Первым и самым важным механизмом является определение четких границ поведения. Агент должен понимать, что некоторые действия категорически запрещены. Как уже упоминалось, это можно реализовать через "конституцию" проекта, например, в виде файла `constitution.md` или встроенного в систему конфигурационного файла [85] [86]. Этот документ должен содержать три категории правил: 1. '**Always do**' **(**Всегда делать**):** Автоматические действия, которые агент выполняет без запроса. Например, "перед каждым коммитом запускать линтер и тесты" или "при создании нового файла добавлять в него шаблон лицензии". 2. '**Ask first**' **(**Спросить первым**):** Высоко-влиятельные изменения, которые требуют подтверждения от человека. Например, "добавление новой большой зависимости в `package.json`" или "модификация существующей базы данных". 3. '**Never do**' **(**Никогда не делать**):** Жесткие запреты, которые служат абсолютными "красными линиями". Например, "никогда не редактировать файлы в директории `node_modules/`", "никогда не удалять файл `.gitignore`", "никогда не коммитить файлы, содержащие API-ключи или другие секреты".

Эти границы являются фундаментом безопасности и предсказуемости агента. Они предотвращают агента от совершения случайных, но катастрофических ошибок, и делают его поведение более детерминированным. Проверка на соответствие этим границам должна происходить на каждом шаге планирования, до фактического выполнения инструмента.

Второй важной стратегией является организация процесса с "воротами" **(Gate Workflow)**. Вместо того чтобы позволять агенту создать весь проект за одну итерацию, что может привести к созданию "карточного домика" из некачественного кода, где одна ошибка приводит к множеству последующих, можно организовать работу поэтапно [86]. Этот подход вдохновлен методологией Spec-Driven Development (SDD), используемой в GitHub Spec Kit [85]. Процесс может быть разбит на следующие этапы: 1. **Specify (**Определить**):** Человек определяет высокий уровень цели. 2. **Plan (**Планирование**):** Агент, основываясь на цели и конвенциях, составляет детальный план действий. 3.

Tasks (Задачи**):** Агент разбивает план на мелкие, выполнимые задачи. 4. **Implement (**Реализация**):** Агент начинает выполнять задачи.

После выполнения каждого значительного блока задач (например, после создания всей структуры каталогов или после написания нескольких ключевых файлов) агент может остановиться и запросить подтверждение или отправить человеку отчет о проделанной работе. Человек может проверить результат, внести корректировки в план или дать добро на продолжение. Такой подход, хотя и кажется противоречащим идее "полной автономности", на самом деле делает агента более надежным и доверенным, так как он превращает его из черного ящика в очень мощного ассистента, работающего по заранее согласованному плану [3].

Третий обязательный элемент — это детальное ведение журнала **(Logging)**. Каждое действие агента, каждое решение, каждая ошибка должны быть записаны в лог. Это включает в себя: начальный запрос, сгенерированный план, вызовы инструментов с параметрами, вывод и ошибки от инструментов, а также промежуточные выводы LLM [90]. Хорошо структурированный лог является бесценным инструментом для отладки, анализа причин сбоев и обеспечения прозрачности работы агента. В современных системах наблюдаемости используются технологии, которые позволяют отслеживать выполнение агента от начала до конца, связывая вызовы инструментов с конкретными состояниями графа [124].

Наконец, для обеспечения воспроизводимости необходимо правильно управлять состоянием агента. LangGraph позволяет сохранять точки контроля, что позволяет возобновлять работу после сбоев [111]. Однако, если состояние хранится в файловой системе, это может привести к проблемам, если файлы изменяются внешними процессами. Для производственных систем настоятельно рекомендуется использовать внешние, надежные хранилища состояний, такие как базы данных. MongoDB Atlas [21] или Amazon DynamoDB [18] являются хорошими кандидатами, так как они обеспечивают долговременное хранение, масштабируемость и атомарность обновлений состояния. Это гарантирует, что агент сможет точно воспроизвести свое предыдущее состояние и продолжить работу с того же места, где он остановился, что является ключевым требованием для надежных и состоятельных систем.

Механизм	Описание	Цель
Определение границ	Задание правил 'Always do', 'Ask first', 'Never do' в конфигурационном файле (например, `constitution.md`).	Предотвращение опасных и нежелательных действий, обеспечение предсказуемости.
Ворота **(Gate Workflow)**	Поэтапное выполнение задачи с остановками для проверки и одобрения человеком на ключевых этапах.	Предотвращение накопления ошибок, обеспечение контроля над качеством результата.
Детальное ведение журнала	Запись всех действий агента, его решений, вызовов инструментов и ошибок в структурированном виде.	Обеспечение прозрачности, упрощение отладки и анализа работы агента.
Внешнее управление состоянием	Использование внешних баз данных (MongoDB, DynamoDB) для хранения состояния агента вместо файловой системы.	Обеспечение долговременного хранения, воспроизводимости и отказоустойчивости сеансов работы.

В совокупности эти механизмы позволяют перейти от идеи хаотичной автономии к созданию управляемой, безопасной и надежной агентской системы. Агент становится не "дьяволом в бутылке", которому можно все, а высококвалифицированным сотрудником, которому предоставлены инструменты и четкие инструкции, а его работа постоянно контролируется и логируется.

Практическая реализация и технологический стек

Для практической реализации полностью автономного ИИ-агента на базе LangChain + LangGraph в стеке Node.js/TypeScript необходимо выбрать подходящий набор инструментов и следовать определенным лучшим практикам разработки. Заявленный стек (Node.js, TypeScript) является отличной основой, поскольку LangGraph.js является зрелой и широко поддерживаемой библиотекой, которая получила стабильный выпуск версии 1.0 и используется крупными компаниями, такими как Uber, LinkedIn и GitLab [42] [46].

Технологический стек и ключевые библиотеки:

- Ядро агента: **LangGraph.js**. Именно эта библиотека станет основой для построения графа выполнения, управления состоянием и оркестрации взаимодействия между LLM и инструментами [9] [73].
- Работа с файловой системой: Для операций с файлами внутри изолированной среды (песочницы) можно использовать стандартный `fs` модуль Node.js. Однако для построения безопасных инструментов предпочтительнее создавать высокоуровневые абстракции, которые

инкапсулируют логику проверки и формирования команд. Для парсинга и манипуляции с Markdown-файлами с конвенциями отлично подойдут библиотеки `remark` или `mdast-util-to-jsx`, которые работают с AST [81].

- Работа с **Git:** Для вызова команд Git из Node.js-скриптов можно использовать библиотеки `simple-git` или `isomorphic-git`. Однако, как было показано, более безопасным подходом является создание собственного инструмента-обертки, который будет валидировать команды и аргументы, как это сделано в проекте `git-commit-agent` [29].
- Строгая типизация и валидация**: Zod** является де-факто стандартом для валидации схем данных в экосистеме TypeScript. Его следует использовать для определения интерфейсов всех инструментов, чтобы гарантировать, что LLM будет генерировать только корректные запросы [29] [60].
- Хранение состояния **(Production):** Для производственных приложений, где важна отказоустойчивость и воспроизводимость, состояние агента не должно храниться на локальной файловой системе. Рекомендуется использовать внешние базы данных. **MongoDB Atlas** с его Vector Search является отличным вариантом, так как он хорошо интегрируется с LangChain/LangGraph и позволяет хранить не только состояние, но и векторные представления контекста [21] [37]. Amazon DynamoDB также является мощным и масштабируемым решением для хранения состояния [18].
- Безопасность**:** Для повышения безопасности самого Node.js приложения следует применять лучшие практики: использовать флаги `--frozen-intrinsics` и `--disable-proto` для защиты от заморочки прототипов, использовать `npm ci` для установки зависимостей, чтобы строго следовать `package-lock.json`, и регулярно проводить аудит уязвимостей с помощью `npm audit` [122][123].

Пошаговый план реализации**:**

1. Настройка проекта**:** Создайте новый проект на Node.js с TypeScript. Настройте `tsconfig.json` в соответствии с лучшими практиками, например, как описано в руководстве DigitalOcean [87]. Установите ключевые зависимости: `@langchain/langgraph`, `@langchain/core` (для моделей), `zod`, `mongodb` (или другую БД для состояния) и инструменты для работы с Git и Markdown.

2. Создание инструментов: Разработайте набор высокоуровневых, безопасных инструментов для файловой системы и Git.

- **FS-**инструменты: Создайте инструменты вроде `createDirectory(path: string)`, `writeFile(path: string, content: string)`, `readFile(path: string)`. Каждый инструмент должен быть обернут в Zod-схему, которая валидирует типы и значения параметров. Реализуйте логику внутри инструментов, которая будет выполнять безопасные операции с файловой системой (например, с использованием `fs.mkdirSync` с опцией `recursive: true`).
- **Git-**инструмент: Реализуйте единый мастер-инструмент `gitTool(command: string, args: Record)`. Эта функция должна содержать белый список допустимых команд, валидировать аргументы с помощью Zod и выполнять команды через `child_process` или специализированную библиотеку, обрабатывая вывод и ошибки.

1. Разработка парсера конвенций: Создайте функцию, которая принимает путь к Markdown-файлу, читает его и преобразует в структурированный JSON. Используйте `remark` для парсинга. Затем напишите логику для извлечения ключевой информации (структура проекта, соглашения об именовании, границы и т.д.) и представления ее в виде объекта, который будет легко использоваться агентом.

2. Проектирование графа **LangGraph:** Определите узлы и ребра вашего графа.

- Узлы:
- `agent_node`: Узел, который принимает контекст (включая задачу, историю и конвенции) и LLM, и генерирует следующее действие (например, вызов инструмента или сообщение).
- `tool_node`: Узел, который принимает решение LLM и выполняет соответствующий инструмент. Результат выполнения (результат или ошибка) возвращается в состояние.
- `check_for_finish_node`: Узел, который анализирует результат выполнения инструмента и решает, нужно ли продолжать выполнение графа или задача завершена.

- Ребра: Определите логику переходов. Например, после `tool_node` граф переходит в `check_for_finish_node`. Если задача не завершена, он может вернуться к `agent_node` для нового раунда планирования.

1. Управление состоянием: Определите структуру состояния графа. Она должна включать в себя историю сообщений, результаты выполнения инструментов, текущий прогресс построения проекта и любую другую релевантную информацию. Настройте сохранение состояния в выбранной базе данных (например, MongoDB) для обеспечения воспроизводимости.

2. Сборка и запуск: Создайте скрипт для запуска агента. Он должен принимать задачу от пользователя, инициализировать граф, загружать и парсить файл с конвенциями, инициализировать состояние и запускать цикл выполнения графа LangGraph.

3. Тестирование и развертывание: Тщательно протестируйте каждый инструмент и узел отдельно. При развертывании в производственной среде убедитесь, что агент выполняется в изолированной среде, например, в Docker-контейнере, и имеет минимально необходимые права доступа.

В заключение, создание полностью автономного ИИ-агента для управления файловой системой и Git является сложной, но вполне достижимой задачей. Успех зависит не от одной "магической" функции, а от комплексного подхода, сочетающего мощь LangGraph для оркестрации, строжайшую безопасность через изоляцию и валидацию, а также тщательное проектирование архитектуры с учетом контроля и предсказуемости. Представленный технологический стек и пошаговый план дают прочную основу для построения надежной и мощной системы, которая сможет выполнять сложные задачи по построению проектов с минимальным вмешательством человека.

Справка

1. Built CodeWarden, an AI-Powered Code Review Tool - LinkedIn https://www.linkedin.com/posts/sounishnath_python-crewai-gemini-activity-7342486774926974976-YtPN

2. Prometheus: Unified Knowledge Graphs for Issue Resolution ... - arXiv https://arxiv.org/html/2507.19942v1
3. David Mieloch Portfolio https://davidmieloch.com/enterprise-ai-development-framework
4. Cektitle | PDF | Microsoft Sql Server - Scribd https://www.scribd.com/document/752533028/cektitle
5. https://udd.debian.org/cgi-bin/attic/sources_in_un... https://udd.debian.org/cgi-bin/attic/sources_in_unstable_but_not_in_testing_by_popcon_max.cgi
6. Using member functions of a class as LangGraph tools https://stackoverflow.com/questions/79807713/using-member-functions-of-a-class-as-langgraph-tools
7. How to Build a LangGraph AI Agent in Minutes ... - DEV Community https://dev.to/getstreamhq/how-to-build-an-ai-powered-ui-components-generator-in-minutes-langgraph-gpt-4-stream-chat-sdk-165n
8. An Absolute Beginner's Guide to LangGraph.js https://techcommunity.microsoft.com/blog/educatordeveloperblog/an-absolute-beginners-guide-to-langgraph-js/4212496
9. Master LangChain in Typescript - A Practical Guide - DEV Community https://dev.to/mohsenkamrani/master-langchain-in-typescript-a-practical-guide-4ifc
10. Build a Secure LangChain RAG Agent Using FGA and LangGraph ... https://www.linkedin.com/posts/auth0_build-a-secure-langchain-rag-agent-using-activity-7303834453380919297-wSZI
11. Lass: Scaffold a modern package boilerplate for Node.js https://lass.js.org/
12. there is node.js scaffolding to create new project? like in visual ... https://stackoverflow.com/questions/63051614/there-is-node-js-scaffolding-to-create-new-project-like-in-visual-studio-for-c
13. NodeJS搭建博客系统(三)用markdown写静态博客原创 https://blog.csdn.net/u011781521/article/details/53524706
14. The Architect's Blueprint for Autonomous Context: Engineering AI ... https://www.linkedin.com/pulse/architects-blueprint-autonomous-context-engineering-ai-james-melvin-xjnpe
15. Lessons Learned from Agentic Coding Failures | Iqbal Y. posted on ... https://www.linkedin.com/posts/iyusuf_%3F%3F%3F-%3F%3F%3F%3F%3F%3F%3F-%3F%3F%3F%3F%3F%3F-%3F%3F%3F-activity-7418511992572956672-k16B
16. Introducing gut: a CLI-based AI agent for git | Jerry Liu posted on the ... https://www.linkedin.com/posts/jerry-liu-64390071_introducing-gut-an-ai-agent-for-git-activity-7352792089857810432-1DAk
17. git - Create remote repository from local files that are already on a ... https://stackoverflow.com/questions/8732977/create-remote-repository-from-local-files-that-are-already-on-a-local-repo

18. Artificial Intelligence – AWS Database Blog https://aws.amazon.com/blogs/database/category/artificial-intelligence/feed/
19. Adaptive Integrates with MLflow for Closed-Loop Optimization https://www.linkedin.com/posts/madhur-prashant-781548179_agent-agentoptimization-activity-7404578268999340032-5F2p
20. Building Production-Ready AI Agents with Next.js and LangGraph.js https://dev.to/ialijr/building-production-ready-ai-agents-with-nextjs-and-langgraphjs-1a79
21. Build an AI Agent with LangGraph.js and MongoDB Atlas https://www.mongodb.com/docs/atlas/ai-integrations/langgraph-js/build-agents/
22. #ai | Eduardo Ordax | 58 comments - LinkedIn https://www.linkedin.com/posts/eordax_ai-activity-7418764200040173568-tAgA
23. From Zero to Fiori: Building SAP Apps with AI Agents (And Why I ... https://community.sap.com/t5/sap-cap-blog-posts/from-zero-to-fiori-building-sap-apps-with-ai-agents-and-why-i-use-markdown/ba-p/14288142
24. [PDF] Language Understanding in the Human Machine Era 2025 ... https://aclanthology.org/2025.luhme-1.pdf
25. Building AI Agents In Action: Architectures, Algorithms, and Source ... https://blog.csdn.net/universsky2015/article/details/157049379
26. How agents can use filesystems for context engineering - LinkedIn https://www.linkedin.com/posts/harrison-chase-961287118_how-agents-can-use-filesystems-for-context-activity-7398405604312170497-eQb9
27. [PDF] Architecting Resilient LLM Agents: A Guide to Secure Plan-then ... https://arxiv.org/pdf/2509.08646?
28. How to fix 'fs module not found' error when using Langchain ... https://stackoverflow.com/questions/76353315/how-to-fix-fs-module-not-found-error-when-using-langchain-document-loaders-in
29. Teaching an AI to `git commit`: building a tool-aware agent in ... https://dev.to/gevik/teaching-an-ai-to-git-commit-building-a-tool-aware-agent-in-langchainjs-4557
30. keywords:model-context-protocol - npm search https://www.npmjs.com/search?q=keywords:model-context-protocol
31. Debian -- Source Packages in "sid", Subsection misc https://packages.debian.org/source/sid/mips64el/misc/
32. Choosing the right agent framework: LangGraph, CrewAI, AI SDK ... https://www.linkedin.com/posts/tylerfolkman_the-agent-framework-you-pick-matters-way-activity-7409256137583202304-4utq
33. LangGraph: How to determine which node triggered the current ... https://stackoverflow.com/questions/79626240/langgraph-how-to-determine-which-node-triggered-the-current-node-during-executi

34. Attila Magyar - ai #vim #linux - LinkedIn https://www.linkedin.com/posts/attila-magyar-41b07983_ai-vim-linux-activity-7363098584910643200-UWI-
35. JavaScript and TypeScript tooling overview https://tooling.js.org/
36. (PDF) English Homophones and Spelling - Academia.edu https://www.academia.edu/36558860/English_Homophones_and_Spelling
37. Get Started with the LangChain JS/TS Integration - Atlas - MongoDB https://www.mongodb.com/docs/atlas/ai-integrations/langchain-js/
38. LangChain literally reverse-engineered Claude Code to build ... https://www.linkedin.com/posts/shubhamsaboo_langchain-literally-reverse-engineered-claude-activity-7360859332487401481-9KAN
39. AgentBay: A Hybrid Interaction Sandbox for Seamless Human-AI ... https://arxiv.org/html/2512.04367v1
40. From ReAct to Ralph Loop A Continuous Iteration Paradigm for AI ... https://www.alibabacloud.com/blog/from-react-to-ralph-loop-a-continuous-iteration-paradigm-for-ai-agents_602799
41. Development & testing · Cloudflare Workers docs https://developers.cloudflare.com/workers/development-testing/
42. Setting Claude Code hooks for AI discipline | Joey Kudish posted on ... https://www.linkedin.com/posts/jkudish_what-guardrails-do-you-put-on-your-ai-coding-activity-7414557411040198658-25OU
43. [PDF] Amazon Bedrock AgentCore - Developer Guide https://docs.aws.amazon.com/pdfs/bedrock-agentcore/latest/devguide/bedrock-agentcore-dg.pdf
44. A Survey of LLM-Driven AI Agent Communication - arXiv https://arxiv.org/html/2506.19676v3
45. Comprehensive Agentic AI v2.0 Learning Roadmap | PDF - Scribd https://www.scribd.com/document/892544253/Comprehensive-Agentic-AI-v2-0-Learning-Roadmap
46. langchain/langgraph - NPM https://www.npmjs.com/package/@langchain/langgraph
47. langchain/mcp-adapters - NPM https://www.npmjs.com/package/@langchain/mcp-adapters
48. @langchain/xai - npm https://www.npmjs.com/package/@langchain/xai
49. Newest 'langchain-js' Questions - Stack Overflow https://stackoverflow.com/questions/tagged/langchain-js?tab=Newest
50. @langchain/aws - npm https://www.npmjs.com/package/@langchain/aws
51. Bca 2021 24 | PDF | World Wide Web - Scribd https://www.scribd.com/document/727327742/BCA-2021-24

52. 333333 23135851162 the 13151942776 of 12997637966 ftp://ftp.cs.princeton.edu/pub/cs226/autocomplete/words-333333.txt
53. Introducing MCP Agent Mail: A Tool for Coding Agents - LinkedIn https://www.linkedin.com/posts/jeffreyemanuel_github-dicklesworthstonemcpagentmail-activity-7388557611622903808-oh0H
54. Parsing options for your data source - Amazon Bedrock https://docs.aws.amazon.com/bedrock/latest/userguide/kb-advanced-parsing.html
55. Building a tool-aware AI agent for git commit in LangChain.js https://www.linkedin.com/posts/gyaansetu-ai_teaching-an-ai-to-git-commit-building-activity-7391109562528878592-4Uv-
56. @toolsdk.ai/registry - npm https://www.npmjs.com/package/@toolsdk.ai/registry
57. [PDF] Architectures for Building Agentic AI - arXiv https://arxiv.org/pdf/2512.09458
58. An API-first approach to building Node.js applications https://developers.redhat.com/articles/2022/10/18/api-first-approach-building-nodejs-applications
59. issue with typescript output scaffolding (keep structure) https://stackoverflow.com/questions/35516200/issue-with-typescript-output-scaffolding-keep-structure
60. Zod and the Joy of Single Sources of Truth - DEV Community https://dev.to/codinsonn/the-joy-of-single-sources-of-truth-277o
61. Sisifo Architecture vs Phoenix Architecture: Limitations and Risks https://www.linkedin.com/posts/uberto_process-over-magic-beyond-vibe-coding-activity-7421055271956869120-cVyQ
62. Time to build a markdown parser and processor (MDL Log #1) https://dev.to/mortoray/time-to-build-a-markdown-parser-and-processor-edl-log-1-44b1
63. How would you go about parsing Markdown? - Stack Overflow https://stackoverflow.com/questions/605434/how-would-you-go-about-parsing-markdown
64. Building an AI Skills Executor in .NET with Azure OpenAI https://devblogs.microsoft.com/foundry/dotnet-ai-skills-executor-azure-openai-mcp/
65. Handling LangGraph ToolExceptions in a REPL - Stack Overflow https://stackoverflow.com/questions/79818757/handling-langgraph-toolexceptions-in-a-repl
66. Optimizing LLM Workflows with Structured Transformers - LinkedIn https://www.linkedin.com/posts/matthewffox_github-foundry81open-notebook-transformations-activity-7421557445258534912-O6-2
67. Everything you might have missed in Java in 2025 - JVM Weekly https://www.jvm-weekly.com/p/everything-you-might-have-missed-886
68. Generative AI With LangChain Build Production-Ready LLM ... - Scribd https://www.scribd.com/document/923991015/Generative-AI-With-LangChain-Build-

Production-ready-LLM-Applications-and-Advanced-Agents-Using-Python-LangChain-And-LangGraph

69. Choosing the right RAG framework for your AI project - LinkedIn https://www.linkedin.com/posts/leadgenmanthan_5-rag-frameworks-for-ai-applications-activity-7360954663501807617-eCI4
70. A Survey of Vibe Coding with Large Language Models - arXiv https://arxiv.org/html/2510.12399v1
71. Deploying MCP servers with Ray Serve: A guide - LinkedIn https://www.linkedin.com/posts/lindahaviv_as-more-developers-move-their-mcp-servers-activity-7370849929235439616-4Dbh
72. Fundamentals and Practical Implications of Agentic AI - arXiv https://arxiv.org/html/2505.19443v1
73. [PDF] Thoughtworks. (2025). Technology Radar (Vol. 32) https://www.thoughtworks.com/content/dam/thoughtworks/documents/radar/2025/04/tr_technology_radar_vol_32_en.pdf
74. The Tech Stack for Building AI Apps in 2025 - DEV Community https://dev.to/copilotkit/the-tech-stack-for-building-ai-apps-in-2025-12l9
75. LangGraph 7 - Platform - Agentic RAG、监督、SQL代理 - CSDN博客 https://blog.csdn.net/lovechris00/article/details/148036752
76. bing.txt - FTP Directory Listing ftp://ftp.cs.princeton.edu/pub/cs226/autocomplete/bing.txt
77. Learn to build ReAct agents with LangGraph: A beginner's guide https://www.linkedin.com/posts/ernestprovo3_machine-learning-mastery-has-just-dropped-activity-7397681737834590208-IyfO
78. github精选Agent学习repo 原创 - CSDN博客 https://blog.csdn.net/polanpan/article/details/154491638
79. From frustration to 700 stars | Romuald Czlonkowski | 33 comments https://www.linkedin.com/posts/czlonkowski_from-frustration-to-700-stars-the-n8n-mcp-activity-7347885940050120706-YhuI
80. Simple Index - SUSTech Open Source Mirrors https://mirrors.sustech.edu.cn/pypi/simple/
81. How can I parse Markdown into an AST, manipulate it, and write it ... https://stackoverflow.com/questions/67797326/how-can-i-parse-markdown-into-an-ast-manipulate-it-and-write-it-back-to-markdo
82. Agentic app with LangGraph or Foundry Agent Service (Node.js) https://learn.microsoft.com/en-us/azure/app-service/tutorial-ai-agent-web-app-langgraph-foundry-node

83. Deep Agents JS: A Powerful Agent Framework for JS Ecosystem https://www.linkedin.com/posts/langchain_deep-agents-js-deep-agents-is-now-available-activity-7392236824448077825-pO0j
84. @houtini/lm - npm https://www.npmjs.com/package/@houtini/lm
85. Diving Into Spec-Driven Development With GitHub Spec Kit https://developer.microsoft.com/blog/spec-driven-development-spec-kit
86. How to write a good spec for AI agents - Addy Osmani https://addyosmani.com/blog/good-spec/
87. How To Set Up a New TypeScript Project - DigitalOcean https://www.digitalocean.com/community/tutorials/typescript-new-project
88. Build Rich-Context AI Apps with Anthropic's MCP - LinkedIn https://www.linkedin.com/posts/andrewyng_new-course-mcp-build-rich-context-ai-apps-activity-7328435643234037760-fZYo
89. [PDF] agentic-frameworks-practical-considerations-for-building-ai ... - Elastic https://www.elastic.co/fr/pdf/agentic-frameworks-practical-considerations-for-building-ai-augmented-security-systems.pdf
90. AI Agents: Mostly Software Engineering, Not Just AI - LinkedIn https://www.linkedin.com/posts/annievella_ai-agents-are-about-90-software-engineering-activity-7357635523890327552-75k_
91. (PDF) The Rise of Agentic AI: A Review of Definitions, Frameworks ... https://www.researchgate.net/publication/395264831_The_Rise_of_Agentic_AI_A_Review_of_Definitions_Frameworks_Architectures_Applications_Evaluation_Metrics_and_Challenges
92. AI Agent Deployment Challenges and Best Practices - LinkedIn https://www.linkedin.com/posts/muhammad-saad-ar_we-talk-a-lot-about-ai-agents-how-smart-activity-7422881438963970048-I9K_
93. Node JS executable application framework or folder structure [closed] https://stackoverflow.com/questions/53516912/node-js-executable-application-framework-or-folder-structure
94. Build Your Own Frontend Scaffolding CLI Tool with Node.js https://dev.to/hexshift/build-your-own-frontend-scaffolding-cli-tool-with-nodejs-1oge
95. Formatting Instructions For NeurIPS 2025 - arXiv https://arxiv.org/html/2505.15216v2
96. Workers llms-full.txt - Cloudflare Docs https://developers.cloudflare.com:2053/workers/llms-full.txt
97. How To Set Up a Node Project With Typescript - DigitalOcean https://www.digitalocean.com/community/tutorials/setting-up-a-node-project-with-typescript

98. AI-Driven Development: Modernizing a Decade-Old Website in 3 Days https://caseywest.com/ai-driven-development-modernizing-a-decade-old-website-in-3-days
99. Full Stack AI SaaS Roadmap | PDF | Databases | Cloud Computing https://www.scribd.com/document/900674262/Full-Stack-AI-SaaS-Roadmap
100. Radar 33 | PDF | Artificial Intelligence - Scribd https://www.scribd.com/document/969854738/radar-33
101. Every engineer knows tool calling is what unlocks their agent's ... https://www.linkedin.com/posts/pauliusztin_every-engineer-knows-tool-calling-is-what-activity-7394365996637171712-r0cs
102. Deep Research with Open-Domain Evaluation and Multi-Stage ... https://www.researchgate.net/publication/396459043_DeepResearchGuard_Deep_Research_with_Open-Domain_Evaluation_and_Multi-Stage_Guardrails_for_Safety
103. A Systematic Review of the Rise of Conversational Agents for Visual ... https://ieeexplore.ieee.org/iel8/6287639/10820123/11271508.pdf
104. Generative AI For Software Development by Balasubramaniam S https://www.scribd.com/document/889886095/Generative-AI-for-Software-Development-by-Balasubramaniam-S
105. [XML] Planet Mozilla https://planet.mozilla.org/rss10.xml
106. 分类工具技巧下的文章 - aneasystone's blog https://www.aneasystone.com/category/tools/
107. GitHub | Laimonas Sutkus ⭕ | 13 comments - LinkedIn https://www.linkedin.com/posts/laimonas-sutkus_github-callsyaiai-openapi-automatically-activity-7393558110986043392-Cxau
108. GitHub Copilot documentation https://docs.github.com/copilot
109. Artificial Intelligence - arXiv https://arxiv.org/list/cs.AI/new
110. Computer Science - arXiv https://arxiv.org/list/cs/new
111. Industrial Engineering and Operations Management - Springer Link https://link.springer.com/content/pdf/10.1007/978-3-031-98235-4.pdf
112. [PDF] A Survey of Vibe Coding with Large Language Models https://www.researchgate.net/publication/396498909_A_Survey_of_Vibe_Coding_with_Large_Language_Models/fulltext/68ef0fc4220a341aa1553778/A-Survey-of-Vibe-Coding-with-Large-Language-Models.pdf?origin=scientificContributions
113. Astro 怎样实现内容集合管理？ - 飞书文档 https://docs.feishu.cn/v/wiki/LCoYw0IU1iIogjkW5u1cvRDqn1d/ai

114. CLI coding agents: OpenAI Codex, Claude Code, Gemini CLI https://www.linkedin.com/posts/matt-koppenheffer_quick-thoughts-on-cli-coding-agents-since-activity-7372353748645363713-wfQ1

115. [PDF] Volume 33 Nov. 2025 - Thoughtworks Technology Radar https://www.thoughtworks.com/content/dam/thoughtworks/documents/radar/2025/11/tr_technology_radar_vol_33_en.pdf

116. 重新定义AI编程协作:深入解析Claude Code多智能体系统架构转载 https://blog.csdn.net/sd7o95o/article/details/153217228

117. Schematica API Documentation Platform | Amir Zouerami posted on ... https://www.linkedin.com/posts/amir-zouerami_check-out-schematica-schematica-is-the-activity-7400979314411966464-0BNJ

118. The Realities of Spec-Driven Development (SDD) - LinkedIn https://www.linkedin.com/posts/rarni_%F0%9D%97%A7%F0%9D%97%9F%F0%9D%97%97%F0%9D%97%A5%F0%9D%97%94-%F0%9D%97%A5%F0%9D%97%B2%F0%9D%97%AE%F0%9D%97%B9%F0%9D%97%B6%F0%9D%98%81%F0%9D%98%86-%F0%9D%97%96%F0%9D%97%B5%F0%9D%97%B2%F0%9D%97%B0%F0%9D%97%B8-%F0%9D%97%BC%F0%9D%97%BB-activity-7385303295797387264-J-8F

119. Helping Users Update Intent Specifications for AI Memory at Scale https://dl.acm.org/doi/10.1145/3746059.3747778

120. Announcing BaseMessage: Safe Agent State Evolution - LinkedIn https://www.linkedin.com/posts/vsh1996_langgraph-langchain-agenticai-activity-7414310214705000449-huxr

121. Mastering Multi-Agent Systems | PDF | Fault Tolerance - Scribd https://www.scribd.com/document/941425531/Mastering-Multi-Agent-Systems

122. Node.js — Security Best Practices https://nodejs.org/en/learn/getting-started/security-best-practices

123. Node js techniques or steps to secure an application - Stack Overflow https://stackoverflow.com/questions/49975568/node-js-techniques-or-steps-to-secure-an-application

124. llmz - npm https://www.npmjs.com/package/llmz?activeTab=readme

125. Paramanantham Harrison's Post - LinkedIn https://www.linkedin.com/posts/paramanantham_want-to-have-better-output-from-claude-code-activity-7417896343224016896-JpDx

126. "12 Open Source Tools Every Developer Should Know" | Faraz ... https://www.linkedin.com/posts/gfaraz_12-open-source-tools-every-developer-should-activity-7338338408240463875-8nPe

127. Singapore | PDF | Artificial Intelligence - Scribd https://www.scribd.com/document/938063459/Singapore

128. OTC CatchUp Summaries - Our Tech Community https://catchup.ourtech.community/summary/

129. LangGraph State Machines: Managing Complex Agent Task Flows ... https://dev.to/jamesli/langgraph-state-machines-managing-complex-agent-task-flows-in-production-36f4

130. Profile for GitHub - Linknovate https://www.linknovate.com/affiliation/github-435897/all/?query=design%20parameter%20pre-setting

131. Typescript file-naming conventions [closed] - Stack Overflow https://stackoverflow.com/questions/60498245/typescript-file-naming-conventions