НЕ ВЕРЬ

во временное

Proxmox

Kubernetes

Terraform

Ansible

GitOps

НЕ БОЙСЯ

делать руками

Registry

Observability

Secrets

Disaster Recovery

НЕ ПРОСИ

ИИ сделать за тебя

ПРАГМАТИЧНОЕ РУКОВОДСТВО

Вахтанг Мосидзе

Не верь. Не бойся. Не проси

https://litres.ru/74532482
SelfPub; 2026

Аннотация

Практическое руководство по созданию отказоустойчивой On-Premise платформы с нуля: от чистого Proxmox до High Availability Kubernetes под управлением Terraform, Ansible, AWX, Cilium eBPF и GitOps (Argo CD). Вместо оторванной от жизни теории читатель проходит полный путь построения production-ready среды на собственном оборудовании. Книга учит безопасно внедрять AI-агентов в DevOps и эксплуатацию систем: выстраивать внешнюю память проекта в Markdown, разделять права модели по уровням риска и подключать живой кластер через протокол MCP без риска разрушительных действий. На реальных сценариях разбираются критические инциденты: рассинхронизация кворума etcd при обрыве сети, ложные статусы bash-скриптов, конфликты адресации, петли CoreDNS и скрытые ловушки приоритетов переменных в AWX. Книга адресована DevOps-инженерам, системным администраторам, SRE и архитекторам. Базовый протокол надежности инфраструктуры с искусственным интеллектом прост: Не верь. Не бойся. Не проси

Содержание

Вахтанг Мосидзе
Не верь. Не бойся. Не проси

От автора

Приветствую тебя, уважаемый читатель.

Для начала расскажу немного о себе - а ты уже решай сам, верить тому, что я расскажу в книге, или нет.

Если считать по годам - а считаю я именно так, потому что цифры без контекста ничего не значат, - у меня за тридцать лет работы в ИТ набралось больше двухсот проектов и три десятка команд, которые я собирал с нуля. Пять из них были стартапами, доведёнными от пустого репозитория до первых клиентов.

Я начал свой путь в IT в далёком 1996 году. Да, я был тем «компьютерщиком», который отвечал за всё, к чему было подключено что-то сложнее лампочки: АТС, камеры, сигнализация. А так как я начинал работать в медицинской организации, то сайт, база данных пациентов, эхоскопы (о да, я видел все свои органы изнутри) и рентгеновские аппараты - тоже была моя зона ответственности.

Но это были ихие 90е, когда разделения на senior/junior, agile, frontend/backend не то что бы не было - их ещё не придумали. Так что опыт я получал на модемах 28.8 с платной

тарификацией, а первые сайты делали одной рукой в чистом HTML.

Да что я говорю, в то время юный Линус Торвальдс только начал развивать свой «just a hobby, won't be big»...

Однако времена менялись - менялись и технологии. Я видел, как менялся интернет, разработка, как начали появляться некие градации в ИТ - одним из первых разделений, пожалуй, стало появление «сисадминов» и их отделение от «программистов». Начали появляться Visual-программирование, интерфейсы становились сложнее, веб-разработка уже начала быть не просто «сайтом», а полноценной инфраструктурой.

Вместе с этим развивался и я. Сама разработка софта меня никогда не привлекала - а вот всякий хард и сети - да. Я начал постепенно изучать BSD, Solaris, Linux (да, уже). Осваивал настройку telnet и удалённое управление.

Серьёзно занимался кибербезопасностью - а надо сказать, что в то время, начало 2000х, я уже возглавлял ИТ-отдел авиакомпании, и было бы сложно отделить безопасность от реальности. И это не был разовый эпизод. Через несколько лет я откроюсь уже отдельной компанией именно под это - SOC/Cybersec/Security, реагирование на инциденты, пентесты, zero tolerance, защита периметра - моими клиентами станут банки и нефтянка, медицина и госслужбы, которым, поверьте, есть что терять. А ещё позже, уже в финтехе, буду выстраивать антифрод-мониторинг и DevSecOps-практики.

Вы спросите: «Ты ж в медицине был». Ну да. Просто в то время такого чёткого разделения на fintech/smartech/enterprise IT не было. Если ты разбирался, допустим, в настройках маршрутизации Cisco - для тебя не было разницы, чем занимается организация, на которую ты работаешь. Значение имело «откуда, куда и что должно быть». То же касалось и почти всего остального.

2000е, пожалуй, были самыми взрывными годами становления ИТ как отдельной профессии. Начали появляться полноценные онлайн-магазины, приложения. Появление первых смартфонов и вообще понятия самой мобильной разработки, облачные технологии - да и вообще облачная инфраструктура.

Как я мог при моей любви к железу и ОС отказать себе в удовольствии не быть одним из пионеров DevOps? Да, само понятие появилось в 2009 по сути, но моменты того, что вот тут что-то надо автоматизировать и запустить без участия лишних рук, назревали давно. Разумеется, у нас не было таких средств, как Jira/GitLab и Kubernetes, но у нас были bash/sed/awk и немного магии FreeBSD jails.

Время неумолимо стремилось вперёд, автоматизация становилась на поток, появились git, CI/CD, Docker, k8s.

И AI.

И тут мы, пожалуй, подходим к главному моменту. Как ИИ сочетается с DevOps? Что оно даёт? Да и вообще зачем?

Ответ тут достаточно прост. Написание хорошего

Terraform или Ansible по сути даже в наше время мало отличается от HTML 1996 года. То есть это жёсткий язык инструкций, который делается руками. Инфраструктура, если в ней больше пяти физических серверов с VM, или тем более гибридная инфраструктура, - чаще всего это реально зарисованная «от руки» карта. Это всё время, энергия и, главное, психологическая нагрузка, которая чаще всего ведёт к выгоранию.

Выгоревший на работе DevOps, особенно если он senior, - это трагедия для всей компании. Цена его ошибки может исчисляться миллионами евро. Один из примеров:

KnightCapitalGroup(2012) - банкротство за 45 минут

Что произошло: финансовая компания обновляла ПО на своих серверах. Из восьми серверов DevOps-инженер случайно забыл обновить один. На этом сервере остался старый, неиспользуемый код. Когда система запустилась, этот сервер начал отправлять на биржу миллионы ошибочных ордеров.

Цена ошибки: $440 млн убытков за 45 минут. Компания не смогла оправиться от финансового удара и обанкротилась.

ИИ же помогает снять эту нагрузку. Помнить, что «там» было. Помочь быстро «прочитать» документацию. Найти «вот там я делал это руками, посмотри по history». Накидать типовую конфигурацию redis-cluster в Ansible.

Да в конце концов проверить теорию или идею намного

быстрее, чем ты успеешь допить свой кофе.

Я уже молчу о возможностях ИИ-агентов для автоматизации мониторинга и алертинга.

И если честно, я использую ИИ в своей работе каждый день. Начиная от разбора Jira и заканчивая «слушай, мне надо Playbook для автоматизации mysql user/pass».

Но эта книга посвящена тем, кто только начинает или решил начать строить свою карьеру **VibeDevOps**.

Вы спросите: почему выбрано решение on-prem? Ну хотя бы потому, что оно не сожрёт годовой бюджет маленькой страны за незакрытые ресурсы. И даёт вам право совершать ошибки, изучать всё, что вы сделали «руками», и, главное, контролировать процесс с нуля.

Удачи в изучении и happy vibecoding!

Глава 1. Место для AI

Откройте любой чат с современной нейросетью и напишите:

«Сделай мне production-ready Kubernetes-кластер».

Если вам повезёт, AI нарисует вам несколько сотен строк священного YAML, щедро сдобренных комментариями о том, как его архитектура выдержит миллион запросов в секунду. Если не повезёт - он сделает то же самое, но ещё и с чувством глубокого морального превосходства.

Так работать нельзя.

И не потому, что AI глуп: он не глуп, он - стажёр, который прочитал все книжки по инфраструктуре, но ни разу не держал в руках ни одного реального сервера. Он не знает, что у вас в шкафу два старых сервера и один коммутатор, который чихает при включении, равно как не знает и того, что ваши файлы, ваши секреты и ваши правила безопасности живут по своим, неведомым ему законам. Поскольку ни одно из этих обстоятельств до него не дошло, он заполнит пробелы сам - красиво и уверенно, но неправильно. Поэтому не верьте AI на слово: он скажет вам то, что вы хотите услышать, а хотите вы услышать, что всё готово. Он это и скажет, после чего разгребать последствия придётся вам.

Злого умысла тут нет, потому что так устроена вероятностная модель, предсказывающая наиболее вероятное про-

должение текста; а наиболее вероятное продолжение текста про кластер - это примеры из документации, где всё идеально: три гипервизора, общее хранилище и отдельные VLAN. Ничего из этого у вас нет, но AI об этом не знает и не спросит. Впрочем, это лишь половина механизма, и сама по себе она безобидна, тогда как вторая половина - ваша.

Вы хотите быстро, и не от лени: задача горит, а ответ выглядит готовым. Именно здесь два дефекта складываются в замок, поскольку я выдаю правдоподобное, вы хотите быстрое - а в момент выдачи «правдоподобно и быстро» ничем не отличается от «верно и быстро», и разница проявится позже, в другом месте. Порознь ни один из дефектов не смертелен: правдоподобную ерунду человек, привыкший проверять, ловит на первой же команде, тогда как спешащий человек, получивший явную чушь, просто её отбросит. Убивает стык.

Выглядит это обычно скучно. Агент называет вам флаг, которого нет: имя составлено по всем правилам этого инструмента, флаг встаёт в конфиг без синтаксической ошибки, проходит ревью и молча ничего не делает. Ошибка обнаружится через полгода, когда кто-то заметит, что ограничение, на которое все ссылались, никогда не работало.

Хуже того, замок затягивает себя сам: каждый раз, оказавшись случайно верным, быстрый правдоподобный ответ повышает доверие, отчего проверок становится меньше, а выборка «ну работало же» - больше, и механизм поощряет сам себя ровно до первого дорогого промаха. Ничего специфи-

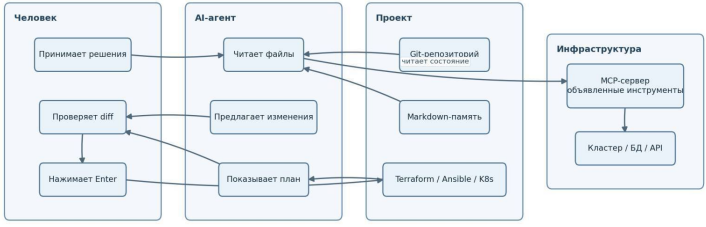
чески искусственно-интеллектуального тут нет, потому что та же конструкция работает у автопилота, у бэкапа, который никто не восстанавливал, и у сигнализации, которая год не срабатывала и именно поэтому считалась исправной. Но самое дорогое следствие состоит не в том, что вы пропустите проверку в конце, а в том, что вы испортите постановку в начале.

Человеку, которому нужен готовый ответ, и задача формулируется как готовый ответ: не «расширь control-plane», а «подними count до пяти»; не «разберись, почему секрет не доехал до пода», а «перезапусти оператор». Решение сидит внутри вопроса, и агенту остаётся его выполнить. Спросить, свободны ли адреса, которые формула выдаст следующим двум узлам, ему в такой постановке негде: его об этом не спрашивали. Поиск кнопки «сделай хорошо» ломает саму кнопку, причём механически, через сужение вопроса. В главе 18 вы увидите обе формулировки на одной задаче: при второй cidrhost выдаст новым узлам управления адреса .12 и .13, а это worker-01 и worker-02, которые в этот момент работают.

Браузерный чат хорош ровно для одного: спросить, почему systemd завершил сервис с кодом 203/EXEC, получить объяснение про отсутствующий executable и закрыть вкладку - на этом его полезность заканчивается. Инфраструктура же живёт в файлах, в Git diff, в планах terraform plan, в логах проверок и в архитектурных решениях, которые мучи-

тельно понадобятся через неделю в три часа ночи, а жить в этом мире браузерный чат не умеет, потому что умеет только говорить. Поэтому нам нужен простой рабочий контур, состоящий из редактора с открытым проектом, Git-репозитория, терминала, AI-агента, умеющего читать и предлагать изменения файлов, и Markdown-файлов, в которых хранится память проекта. Конкретный редактор и агент при этом не важны, поскольку названия продуктов меняются быстрее, чем инженеры успевают обновлять скриншоты в документации; важны возможности: агент должен видеть разрешённые файлы, показывать изменения до применения, работать в ограниченном каталоге и спрашивать подтверждение перед командами, способными изменить внешнюю систему.

Вот как это выглядит в схеме:



Вручную печатать каждую строку человек не обязан, зато обязан понимать границу изменения, увидеть diff и решить, можно ли нажимать Enter. Разрешения для этого по-

лезно сразу разделить на три класса, и делается это не из мании контроля, а в качестве страховки: классы отличаются друг от друга тем, сколько стоит вернуть всё назад, если решение окажется неверным.

Чтение - просмотр файлов, поиск по репозиторию, чтение локальных логов. Обычно это можно разрешить без подтверждения, если в область чтения не попадают секреты. Если агент может прочитать ~/.ssh/id_rsa - вы уже проиграли.

Локальные изменения - редактирование файлов проекта. Разрешаем в пределах текущей области задачи, но обязательно проверяем diff: агент его показывает, вы его смотрите и вы же решаете, принять или отклонить. Без этого шага агент - просто очень быстрый вандал.

Внешнее воздействие - terraform apply, kubectl apply, изменение Proxmox, установка пакетов, отправка коммита, обращение к production API. Здесь требуется явное подтверждение человека, причём сознательное и осознанное, а не «ну, вроде всё правильно, давай».

Разница между классами измеряется не строгостью, а обратимостью: файл можно вернуть через Git, удалённый volume иногда тоже можно вернуть - если у вас был backup, если restore проверяли и если сегодня ваш счастливый день, - но строить процесс вокруг последнего условия не стоит. Отсюда следует, что интеграция агента в редактор - это ещё не повод выдавать ему все разрешения. Многие инструменты предлагают удобный режим автоматического подтвержде-

ния, в котором агент сам читает файлы, запускает команды и принимает изменения; на демонстрации это выглядит волшебно, тогда как в инфраструктурном репозитории волшебство заканчивается примерно там, где в shell history появляется первая команда с правами администратора.

Начните с минимального режима, открыв агенту только каталог проекта и ничего сверх него. Не добавляйте домашний каталог целиком: рядом могут лежать ~/.ssh, kubeconfig, конфиги облачных CLI и другие вещи, которые совершенно не нужны для написания README. Отключите автоматический запуск терминальных команд и автоматическое принятие diff, а если инструмент умеет отдельно подтверждать чтение и запись, чтение внутри проекта можно разрешить постоянно, оставив запись с просмотром.

Проведите скучный, но полезный тест. Назвав корень доступного workspace, попросите агента показать список файлов верхнего уровня и заранее запретите ему что-либо создавать или запускать команды вне репозитория; в пустом проекте ответ должен получиться почти пустым. Затем попросите его предложить создание файла permission-test.txt, но изменение не принимайте: diff обязан появиться до того, как файл окажется в рабочем дереве, после чего предложение следует отменить. Так мы проверяем не интеллект модели, а механику инструмента - где проходит граница workspace, что считается подтверждением и можете ли вы остановить изменение. Если же агент без вопроса читает соседние ка-

талоги или выполняет команды, не утешайте себя тем, что «там пока ничего секретного», а исправьте разрешения сейчас, поскольку эта фраза обладает удивительной способностью переживать миграции, смену сотрудников и появление production kubeconfig в том же каталоге.

Теперь, когда границы понятны, стоит разобраться с тем, что за этими границами лежит. Агент, которого мы только что настроили, видит файлы, и этого достаточно, пока вы пишете код. Но инфраструктура тем и отличается от кода, что её состояние живёт не в файлах: сколько подов в CrashLoopBackOff прямо сейчас, какой узел держит виртуальный адрес, что лежит в таблице pg_stat_replication - ничего из этого нет в репозитории, всё это есть только в кластере.

И вот здесь начинается то, что превращает работу с агентом в утомительный ритуал. Вы выполняете kubectl describe, копируете вывод, вставляете в чат. Агент отвечает: «покажите ещё логи оператора». Вы выполняете, копируете, вставляете. Он просит kubectl get events. Вы выполняете, копируете, вставляете. Заметьте, кем вы стали в этой схеме. Транспортом. Живым проводом между терминалом и моделью, который вручную переносит байты туда и обратно; это не вайбкодинг, это работа курьером при собственном ассистенте — и на третьем круге вы поймаете себя на мысли, что быстрее было разобраться самому.

Решение называется **MCP**(*ModelContextProtocol*) — открытый стандарт, по которому агент получает не доступ к ва-

шему терминалу, а набор объявленных инструментов. Между агентом и системой встаёт отдельный процесс, MCP-сервер: он умеет ровно то, что заявил, и ничего сверх того, поэтому агент не выполняет kubectl — он вызывает инструмент «прочитать поды в namespace», а сервер сам решает, во что это превратить. Обратите внимание, как это ложится на три класса разрешений, которые мы только что разобрали: MCP-сервер, умеющий только читать состояние кластера, — это ровно первый класс, то есть чтение без подтверждения. Сервер, умеющий применять манифесты, — уже третий, и подключать его надо с тем же выражением лица, с каким вы выдаёте кому-то права на terraform apply. Разница между этими двумя серверами не в модели и не в промпте: у одного в объявленном наборе инструментов есть запись в кластер, у другого её нет, и выяснить это можно, открыв конфиг и прочитав четыре строки.

Конфигурация живёт в настройках вашего агентского клиента и в общем виде выглядит так:

TERMINAL

```
{
"mcpServers": {
"k8s-readonly": {
"command": "npx",
"args": ["-y", "mcp-server-kubernetes"],
"env": {
"KUBECONFIG": "/home/operator/.kube/config-readonly",
```

```
  "ALLOW_ONLY_NON_DESTRUCTIVE_TOOLS":
"true"
    }
    },
    "platform-db": {
    "command": "postgres-mcp",
    "args": ["--access-mode=restricted"],
    "env": {
    "DATABASE_URI": "${PLATFORM_DB_URI}"
    }
    }
    }
    }
```

Три детали здесь важнее самих строк.

Первая: KUBECONFIG указывает на **отдельный-**kubeconfig, а не на ваш административный. Агент получает те права, которые лежат в этом файле, и ни одним больше. Если вы дадите ему свой admin.conf, никакой MCP не спасёт - вы просто вручили ключи от кластера процессу, который запускается через npx. Флаг ALLOW_ONLY_NON_DESTRUCTIVE_TOOLS убирает из объявленного набора всё разрушающее, но это второй рубеж, а не первый: права в kubeconfig решают, флаг лишь сужает то, что и так разрешено.

Вторая: у сервера базы выставлен --access-mode=restricted - режим, в котором доступны только читающие транзакции.

Тот же принцип, что и с кластером: не «мы договорились, что агент не пишет», а «пишущих инструментов в наборе нет».

Третья: DATABASE_URI - это ссылка на переменную окружения, а не строка подключения. Значение приходит из окружения, в котором стартует клиент, а туда попадает из менеджера паролей. Пароль в этом файле не появляется никогда. К главе 14 мы заведём для этого Vault и поставим прекоммит хук, который такую строку не пропустит, - и было бы неловко, если бы он сработал на примере из первой главы. Синтаксис подстановки у клиентов разный: где-то ${VAR}, где-то переменные наследуются из окружения и в конфиге ключ вообще не пишут. Сверьтесь со своим; неизменно одно - в файле лежит ссылка, а не значение.

Рабочий вариант этого конфига лежит в labs/mcp/config.json - там же записано, какого сервера в нём намеренно нет и почему. Экосистема шевелится быстро: серверы переименовываются, меняют флаги, иногда их бросают. Летом 2025-го авторы протокола вынесли часть своих же эталонных серверов в отдельный архив как неготовые к эксплуатации - в том числе тот, что работал с PostgreSQL, и в нём к тому моменту нашли внедрение SQL. Поэтому проверяйте, что пакет, который вы прописываете, жив и поддерживается, - а не копируйте листинг из книги, включая эту. Важна форма: команда запуска, окружение, объявленный набор инструментов.

И сразу о том, чего MCP **не**даёт. Он не отбирает у агента доступ к терминалу. Если ваш клиент умеет запускать shell - он продолжит уметь, MCP тут ни при чём. Ограничение работает только на своём канале: агент не может через MCP-сервер сделать то, чего этот сервер не объявил. Разница между «агент ограничен» и «агент ограничен вот в этом одном месте» существенная, и путать их дорого. К главе 23 мы вернёмся к ней всерьёз, когда будем строить структурные заборы вокруг агента, которому доверено действовать самостоятельно.

Пока запомните одно: без MCP агент в инфраструктуре слеп, и вы обречены быть его глазами, тогда как с MCP он смотрит сам - в пределах того, что вы ему открыли. И раз уж речь зашла об открытом и закрытом, сделаем последнее различение, без которого дальше будет путаница: слово «агент» обозначает в этой книге две разные вещи.

Есть **агент-консультант**- тот, с кем вы обсуждаете. Он читает репозиторий, задаёт вопросы, предлагает план и ошибается дёшево: его ошибка живёт в тексте, пока вы её не приняли, и стоит она ровно тех двух минут, которые вы потратили на чтение. Широта - его работа, и чем шире он смотрит, тем полезнее; половина того, что он предложит, вам не понадобится, и это нормальная цена.

И есть **агент-исполнитель**- то, что применяет уже принятое решение. Он должен быть буквальным и скучным: делать ровно сказанное и ничего сверх того. Его достоинство

не в сообразительности, а в предсказуемости, потому что пока он не проявляет инициативы, всё мышление остаётся в одном месте и его можно отревьюить целиком.

Как только исполнитель начинает быть полезным - по дороге чинить, дополнять, улучшать «раз уж я здесь», - проверять приходится не результат, а его суждение, а суждение не показывает ни terraform plan, ни git diff. Хорошая новость, впрочем, состоит в том, что исполнителей не надо изобретать, поскольку вы их и так построите: AWX прогоняет роль ровно так, как записано в Job Template, а Argo CD приводит кластер к тому, что лежит в Git, не имея собственного мнения о том, правильно ли это. Идеальные исполнители - тупые, буквальные, с полным журналом сделанного, и к главе 20 консультант будет отдавать работу им, а не выполнять её сам.

Создадим пустую директорию, инициализируем Gitи откроем проект в редакторе:

TERMINAL

```
mkdir vibecode-onprem
cd vibecode-onprem
git init code .
```

Если команда code у вас не установлена, откройте каталог через интерфейс редактора, поскольку на инженерный смысл эксперимента это не влияет. Предположим теперь, что мы строим учебную on-prem платформу на Proxmox: она должна быть похожа на реальную, но мы пока не выбрали се-

ти, домены, версии ОС, модель хранения и даже количество гипервизоров, и это не досадные пробелы, которые агенту следует заполнить. Это непринятые решения.

Открыв агента, ставим задачу - и не просим его сделать кластер, а просим подготовиться к этому.

ИНЖЕНЕР

Мы начинаем новый DevOps-проект в текущем репозитории. Цель первой сессии - подготовить рабочую память проекта и правила взаимодействия. Не пиши Terraform, Ansible, Kubernetes-манифесты и shell-скрипты. Сначала изучи только текущее содержимое репозитория и покажи краткий план. Затем предложи содержимое файлов README.md, ai_initial.md, docs/architecture.md, docs/decisions.md, docs/incidents.md, docs/runbooks.md и plans/current-task.md. Не придумывай IP-адреса, домены, версии, секреты, существующие узлы и уже принятые решения. Все неизвестные вынеси в открытые вопросы. В ai_initial.md зафиксируй порядок чтения проектной памяти, разделение read-only и mutating actions, запрет destructive-команд и внешних изменений без подтверждения, обязательные plan, diff, verification и handoff. В plans/current-task.md должны быть Goal, Scope, Out of Scope, Done, To Do и Open Questions. Не дублируй один текст во всех документах: у каждого файла одна роль. После подготовки файлов покажи diff и остановись. Ничего не устанавливай, не обращайся к внешней инфраструктуре и не создавай дополнительные файлы.

Это не заклинание и не «идеальный промпт», а инженерная задача, в которой определены результат, область работы, неизвестные, запреты и точка остановки. Особенно важна последняя фраза: без неё деятельный агент может решить, что семь документов - лишь разминка перед созданием main.tf. При этом хороший агент не обязан немедленно создавать файлы, потому что сначала он должен обнаружить решения, которые нельзя принять из доступного контекста.

AI

Агент: Я понял границы первой сессии. Перед созданием документов уточню три вещи. Проект останется учебным или его правила должны быть пригодны для production-like среды? Нужны ли сейчас правила для нескольких агентов или оставляем одну роль? Какой язык документов предпочтителен?

Вопросы полезны, потому что ответы меняют артефакты: строгость предохранителей зависит от назначения проекта, язык - от команды, а роли агентов - от будущего рабочего процесса. Вопрос «Какой ваш любимый цвет YAML?» можно было бы не задавать.

ИНЖЕНЕР

Инженер: Стенд учебный, но процесс проектируем как production-like. Добавь роли исследователя, исполнителя и ревьюера, но без автоматической параллельной записи в одни файлы. Создавай семь согласованных файлов, покажи diff и остановись.

Теперь квота доверия выдана ровно на одну локальную операцию, а значит - не запускать Terraform, не стучаться в Proxmox API и не готовить «заодно» кластер, поскольку область задачи есть бетонный забор вокруг неё, а не декоративная рамка в Jira. Документы памяти при этом не должны быть клонами друг друга, ведь если одна и та же архитектура полностью пересказана в пяти местах, через месяц у вас будет пять разных архитектур. И все они будут называться актуальными.

README.mdотвечает на вопрос: «Куда я попал и что читать первым?»

FILEREADME.md

Vibecode On-Prem Platform

Учебный путь от пустого Proxmox к управляемой on-prem платформе.

Start here

1. Прочитай `ai_initial.md`.
2. Открой `plans/current-task.md`.
3. Проверь факты в `docs/architecture.md`.
4. Посмотри решения и открытые вопросы в `docs/decisions.md`.

Current status

Stage zero. Инфраструктура и сетевой план ещё не определены.

README не обязан содержать историю цивилизации, полный inventory и инструкцию по восстановлению etcd, по-

тому что если точку входа приходится читать пятнадцать минут - это уже не карта, а экскурсия. Следующий файл, **ai_initial.md**, назван намеренно нейтрально, поскольку разные агенты распознают собственные служебные файлы; если ваш инструмент требует AGENTS.md, CLAUDE.md или другой файл - перенесите туда тот же смысл.

FILE ai_initial.md

AI operating contract

Before work

1. Прочитай `README.md`, `ai_initial.md` и `plans/current-task.md`.

2. Прочитай только релевантные документы и исходники.

3. Сверь Goal, Scope, Out of Scope и Open Questions.

4. До редактирования покажи план.

Permissions

- Read-only checks внутри scope разрешены.

- Локальные изменения разрешены только после согласования плана.

- Внешние и mutating actions требуют явного подтверждения.

- Destructive actions требуют отдельного предупреждения, плана восстановления и подтверждения.

Never

- не выдумывай отсутствующие факты;

- не показывай и не записывай секреты;

- не расширяй scope словом «заодно»;

- не объявляй задачу готовой без verification.

Session close

Покажи diff и результаты проверок. Обнови только изменившиеся документы памяти.

Зафиксируй незавершённое и назови следующий безопасный шаг.

«Обнови память» не означает «перепиши все документы свежими словами»: если архитектура не изменилась, docs/architecture.md трогать не нужно, иначе факты начнут мутировать при каждом handoff, как рыба в рассказе человека, вернувшегося с рыбалки. Сам же **architecture.md**содержит подтверждённые факты, тогда как гипотезы и варианты живут в decisions.md.

FILE decisions.md

Architecture

Current state

- Stage: zero.
- Managed infrastructure: none.
- Confirmed nodes: none.

Target direction

Proxmox → bastion → Terraform-managed VM → Ansible/AWX → Kubernetes → platform services → GitOps.

Unknowns

- network and VLAN plan;
- DNS zones;
- Ubuntu and Proxmox versions;

- storage model;
- failure domains.

Разделение Current state и Target direction обязательно, поскольку «мы хотим три control plane» не означает «у нас есть три control plane», а в инфраструктуре будущее время особенно любит маскироваться под настоящее. Следующий файл, **decisions.md**, хранит решения и их цену: не только итог, но и причину выбора, альтернативы и последствия, - а пока решения нет, туда записывается открытый вопрос.

FILE decisions.md

```
# Decisions
## ADR template
- Date:
- Status: proposed | accepted | superseded
- Context:
- Alternatives:
- Decision:
- Consequences:
- Verification:
```

Последствия нужны потому, что архитектурное решение без цены подозрительно похоже на рекламу. Что до **incidents.md**, это память о том, что уже болело, и пока инцидентов нет - единственный момент проекта, когда данный файл выглядит оптимистично.

FILE incidents.md

```
# Incidents
```

Template
- Symptom:
- Hypotheses:
- Checks:
- Root cause:
- Fix:
- Guardrail:

Формат здесь важнее полноты: не «Kubernetes сломался», а что увидел оператор, какой командой проверил гипотезу и что изменилось после исправления. Следом идёт **runbooks.md**- действия под давлением, и регламент это не лекция, а последовательность команд с условиями запуска, ожидаемым выводом и точкой остановки.

FILE runbooks.md

Runbooks
Repository bootstrap check
1. Выполнить `git status --short`.
2. Убедиться, что изменены только файлы текущего scope.
3. Просмотреть `git diff --check` и `git diff`.
4. Запустить настроенный secret scanner, если он есть.
5. Не коммитить, пока Open Questions выданы за факты.

Человек открывает регламент не для духовного роста: обычно у него уже красный мониторинг, два сообщения от руководителя и коллега, который спрашивает: «Ну что там?» Последний файл, **current-task.md**, работает как оператив-

ная память, и в нём всегда должен быть ответ на вопрос «что мы делаем сейчас и что мешает закончить?»; меняется он чаще остальных именно потому, что играет роль оперативной памяти, а не семейного архива.

FILE current-task.md

```
# Current task: bootstrap project memory
## Goal
Создать проектную память и контракт работы с AI.
## Scope
- семь согласованных Markdown-файлов;
- протокол начала и закрытия сессии.
## Out of scope
- IaC и Kubernetes-код;
- установка инструментов;
- внешняя инфраструктура и секреты.
## Done
- [x] Согласованы роли документов.
## To do
- [ ] Проверить diff.
- [ ] Проверить, что неизвестные не выданы за факты.
## Open questions
- сеть, DNS, версии, storage и topology.
```

Представим вполне реалистичное продолжение, в котором агент создал семь файлов, а затем сообщает: «Для удобства я также добавил setup.sh, базовый .github/workflows/validate.yml и шаблон Terraform provider. Это ускорит сле-

дующий этап». Звучит разумно - и именно поэтому опасно, ведь дополнительные файлы не входили в область задачи, их требования не обсуждались, а workflow к тому же способен выполнять действия при будущем push. Выяснять, хороший ли там код, нам не нужно. Нарушен сам контракт.

ИНЖЕНЕР

Инженер: Стоп. Дополнительные файлы не были разрешены. Покажи их diff отдельно, удали только созданные тобой вне области задачи файлы и не меняй семь согласованных документов. Затем добавь в ai_initial.md guardrail: новые артефакты вне списка требуют отдельного согласования.

Получилась первая учебная цепочка. Антипаттерн: агент расширяет задачу «полезной» инициативой. Инцидент: в репозитории появляются нерассмотренные исполняемые файлы. Исправление: лишние артефакты удалены после проверки их происхождения. Предохранитель: список разрешённых файлов считается закрытым, расширение требует подтверждения. Важно не превращать это в истерику «AI снова всё испортил», поскольку агент сделал то же, что иногда делает увлечённый инженер, решивший немного улучшить соседнюю систему. Предохранитель нужен не потому, что одна сторона глупая, а потому, что обе стороны деятельные.

Остановившись, проверяем рабочее дерево:

TERMINAL

```
git status --short
```

git diff --check

git diff -- README.md ai_initial.md docs plans

Ожидаем ровно семь новых файлов, отсутствие ошибок пробелов и понятный diff. Затем ищем очевидные маркеры секретов и случайно придуманные адреса:

TERMINAL

rg -n -i 'password|token|secret|private.?key|BEGIN .* KEY' README.md ai_initial.md docs plans

rg -n '10\.|172\.(1[6-9]|2[0-9]|3[01])\.|192\.168\.' README.md ai_initial.md docs plans

Это sanity check - быстрая проверка здравого смысла, в силу того что регулярное выражение не знает всех форматов токенов, DNS-имён и внутренних идентификаторов. В реальном проекте понадобятся secret scanner, правила исключения чувствительных каталогов из контекста и внимательный просмотр diff. Наконец, проверяем, усвоил ли агент собственный контракт:

ИНЖЕНЕР

Прочитай ai_initial.md. Назови действия, которые разрешены без подтверждения, действия, требующие подтверждения, и три причины, по которым ты обязан остановиться.

Если ответ расплывчатый - проблема не обязательно в модели: возможно, правила написаны как корпоративные ценности на стене, то есть звучат благородно, но понять, что делать руками, по ним невозможно. Впрочем, самая чест-

ная проверка наших документов происходит не сейчас, а после закрытия чата, потому что новая сессия не знает устных договорённостей, интонации и того, что вчера под словом «кластер» вы имели в виду учебный стенд. Дайте новому агенту короткую команду:

ИНЖЕНЕР

Восстанови контекст проекта. Сначала прочитай README.md, ai_initial.md и current-task.md, затем только те документы, на которые они ссылаются для текущей задачи. Ничего не меняй. Верни Goal, Scope, Out of Scope, подтверждённые факты, неизвестные и следующий безопасный шаг.

Хороший ответ помещается в несколько абзацев и ссылается на файлы, тогда как плохой начинает заново интервьюировать вас о назначении проекта или уверенно сообщает параметры, которых в документах нет; в первом случае память работает, во втором - найдите, какой факт потерян или в каких двух документах он записан по-разному. Проектная память, кстати, тоже требует уборки: завершённая задача не должна вечно лежать в current-task.md, её результат переносится в архитектуру, решение, инцидент или историю Git, устаревшее решение получает статус superseded, а не исчезает, а регламент удаляется или исправляется, если команды больше не соответствуют системе. Иначе через полгода агент будет экономно читать четыре файла и экономно приходить к неверному выводу.

После проверки не закрываем ноутбук со словами «вроде нормально»: завтрашний агент не помнит сегодняшний разговор, а завтрашний вы, строго говоря, тоже помните его несколько оптимистичнее, чем было на самом деле.

Команда на завершение:

ИНЖЕНЕР

Закрой сессию и подготовь handoff. Проверь diff всех изменённых файлов и перечисли выполненные проверки. Обнови current-task.md, а остальные документы - только если содержащиеся в них факты действительно изменились. Зафиксируй открытые вопросы. Ничего не коммить и не переходи к следующей задаче. Назови один следующий безопасный шаг.

Хороший ответ содержит не настроение, а свидетельства:

FILEcurrent-task.md

Handoff

- Созданы семь файлов проектной памяти.
- Удалены три неразрешённых артефакта вне scope.
- В ai_initial.md добавлен guardrail на создание новых файлов.
- Выполнены git status --short, git diff --check и ручной просмотр diff.
- Внешняя инфраструктура не изменялась.
- Открыты вопросы сети, DNS, версий, storage и topology.
- Следующий шаг: описать ownership и desired state до выбора инструментов автоматизации.

Слово «готово» без списка проверок ничего не стоит. Это просто хорошее настроение агента.

За одну сессию мы не построили ни одной виртуальной машины, зато получили рабочее место, память проекта, границы разрешений, протокол вопросов, проверку diff, первый инцидент и предохранитель. Теперь новый агент способен за несколько минут понять, где находится проект, что считается фактом и к чему ему нельзя прикасаться. Проверьте результат руками: закройте текущий чат, начните новую сессию и попросите агента прочитать точку входа, назвать текущую цель, неизвестные и запрещённые действия. Если для восстановления контекста требуется пересказать всю вчерашнюю беседу, память проекта не работает; если достаточно нескольких файлов, первый слой платформы уже построен. Он не отвечает ни на одном сетевом порту. Это пока его лучшее свойство.

И раз уж рабочее место готово, назовём три правила, вокруг которых собрана вся остальная книга. Они вынесены в её название, и у каждого два чтения: про инфраструктуру и про того, кто вам в ней помогает.

Не верь.В инфраструктуре - временным решениям: они переживают своих авторов, свои тикеты и иногда своих создателей. В работе с агентом - его интонации. Он одинаково уверенно пишет проверенное и выдуманное, потому что уверенность в его тексте не измерение, а выученная манера. Скидку делайте в обе стороны - и на «безусловно», и на «по-

жалуй». Единственный неподдельный сигнал: может ли он предъявить проверку, которая упадёт, если он неправ.

Не бойся.В инфраструктуре - делать руками и лезть внутрь, вместо того чтобы верить зелёному индикатору. В работе с агентом - того, что проверка выглядит как недоверие. Она не отношения выясняет, она и есть форма работы. Вопрос «с чем ты сверялся?» продуктивен. Вопрос «ты уверен?» - нет: у агента нет доступа к собственной уверенности, и он ответит «да» ровно так же убедительно, как отвечал всё остальное.

Не проси.В инфраструктуре - сделать за тебя то, за что отвечаешь ты. В работе с агентом - определить, что значит «готово». Вот это нельзя отдавать никогда: критерий готовности агент задаст как «то, что я произвёл», и не из хитрости - у него просто нет другого источника. Сюда же относится вторая подмена, менее заметная: он с той же готовностью определит за вас и то, о чём именно его попросили.

Дальше эти три правила не повторяются мантрой - они проверяются, и в каждой главе найдётся место, где одно из них можно нарушить и посмотреть, что из этого выйдет.

В следующей главе мы разберём desired state и владельцев изменений. До появления Terraform, Ansible и Kubernetes нужно решить, кто отвечает за VM, операционную систему, кластерные объекты и приложения. Иначе три замечательных инструмента начнут исправлять друг друга, а инженер получит редкую возможность наблюдать вечный двигатель в

пределах собственного дата-центра.

Глава 2. Не просите AI чинить: научите его расследовать

Инженер показывает AI одну строку ошибки и пишет: «Почини».

AI отвечает мгновенно и предлагает очистить кэш, перезапустить сервис, переустановить пакет, отключить файрвол и добавить запись в /etc/hosts, причём какое-нибудь из этих действий иногда даже помогает. После этого система снова работает, причина остаётся неизвестной, а в инфраструктуре появляется ещё одно временное решение с прекрасными шансами пережить автора.

Так рождаются технические легенды. «Этот сервис по вторникам надо рестартовать два раза». «DNS здесь особенный». «Строку не трогай, без неё всё падает». Никто уже не помнит, откуда взялась строка, зато она заботливо переносится из одного поколения конфигурации в другое, как фамильное проклятие, и через три года, когда вы будете мигрировать на новый кластер, кто-нибудь обязательно спросит: «А мы перенесли эту магическую строку?» - и ответить не сможет никто. Но перенесут. На всякий случай.

AI особенно опасен в таком режиме вовсе не оттого, что плохо ищет ошибки; беда ровно в обратном - он очень быстро находит правдоподобные объяснения, и когда контекста

мало, правдоподобие начинает подменять доказательство. Модель узнаёт знакомый симптом и достраивает остальную историю по наиболее вероятному шаблону, иногда угадывая. Вот здесь и важно понять, что в production слово «иногда» перестаёт быть статистической погрешностью и становится бомбой с часовым механизмом, которая просто ждёт своего часа.

Задача этой главы - не устанавливать софт: мы снова ничего не будем ставить. Да, опять. Да, так надо. Научиться нужно другому - расследовать сбой вместе с AI: собрать свидетельства, отделить факт от гипотезы, проверить версии по одной, найти не только непосредственную причину, но и системную, а затем сохранить знание в проекте. Это ещё не Linux-диагностика, ею займёмся позже; сейчас мы строим сам процесс мышления, которым потом будем разбирать Linux, сеть, Terraform, Kubernetes и всё остальное, что умеет ломаться с гораздо большим размахом. А начинать придётся с терминов, потому что слово «ошибка» обычно используют сразу для пяти разных вещей.

Симптом- то, что заметил человек или мониторинг: страница не открывается, команда завершилась неуспешно, файл не появился. Симптом сообщает, где болит, но редко объясняет почему.

Свидетельство- наблюдаемый факт: точная команда, время, код возврата, строка лога, diff, состояние процесса. Свидетельство можно повторно проверить или привязать к

источнику.

Гипотеза- возможное объяснение свидетельств: «имя не резолвится из-за отсутствующей DNS-записи» - гипотеза, которая становится полезной только вместе с проверкой, способной её опровергнуть.

Непосредственная причина- конкретное условие, из-за которого операция не выполнилась именно сейчас: например, в конфигурации остался «enter_your_ip_here».

Системная причина- объяснение того, почему неправильное условие прошло через процесс незамеченным: скрипт не проверил заглушку, проигнорировал ошибку curl, напечатал сообщение об успехе и вернул код 0. Исправить только имя узла - значит убрать непосредственную причину и сохранить фабрику следующих инцидентов.

Исправление(*fix*) восстанавливает правильное поведение, тогда как **предохранитель**(*guardrail*) меняет код или процесс так, чтобы тот же класс ошибки не проходил бесшумно. Перезапустить сервис можно считать исправлением; добавить проверку конфигурации до запуска и функциональный health check после - предохранитель.

AI должен работать именно с этой цепочкой, и если агент перескакивает от симптома прямо к исправлению, он не расследует. Он участвует в лотерее с очень убедительными комментариями.

В будущем нам понадобится stage-zero bootstrap - минимальный ручной этап, который подготовит вход в инфра-

структуру до появления полноценной автоматизации; поскольку никакой инфраструктуры пока нет, проверим только локальную механику будущей предстартовой проверки. Представим, что агент предложил такой черновик:

TERMINAL

```
#!/usr/bin/env bash
MANAGEMENT_HOST="${MANAGEMENT_HOST:-control.example.invalid}"
echo "Checking management endpoint..."
curl -s "https://${MANAGEMENT_HOST}/health" || true
echo "Stage-zero preflight completed successfully"
```

Домен .invalid зарезервирован именно для примеров и не должен разрешаться в реальный адрес. Запускаем скрипт:

TERMINAL

```
$ ./stage-zero-preflight.sh
Checking management endpoint...
Stage-zero preflight completed successfully
$ echo $?
0
```

На экране нет даже ошибки: curl работает в тихом режиме, а || true превращает любой его результат в успех. Скрипт не проверил точку подключения, но сообщил, что проверил, и это хуже честного падения, поскольку красный статус хотя бы мешает идти дальше, тогда как ложный зелёный открывает ворота следующему шагу. || true в продакшене - это как заклеить лампочку Check Engine на приборной панели изо-

лентой. Машина едет, индикатор не горит, все счастливы. До первого перегрева.

Не исправляйте код сразу: сейчас важнее правильно поставить задачу агенту. Новая сессия начинается с восстановления памяти проекта, после чего агенту назначается роль исследователя - read-only, без редактирования и выполнения команд. Мы не написали «ты Senior SRE с тридцатилетним опытом и IQ размером с дата-центр», поскольку ролевая биография не заменяет протокол, а по факту является продуктом маркетинга и попыткой сыграть на «комплексе менеджера». Настоящий Senior SRE не говорит «я всё знаю» - он говорит «я не знаю, но знаю, как это проверить». Именно этому мы учим агента.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.