nil team

Александр Вяткин · Никита Шморин · Владимир Гальцев · Александр Семёнов

Все – Backend разработчики. Но в этом проекте мы почти не писали backend.

Кто мы в эпоху AI

1 **Раньше**

Мы писали код

2 **Сейчас**

Мы управляли тем, кто его пишет

Факт. AI Contribution 100%

0

Строк кода

Human

5K+

Строк кода

AI

за 1 неделю

Роли в проекте

Владимир Гальцев

AI Architect / Prompt Strategist

- формулировал системные промпты
- строил архитектуру, опираясь на AI-анализ и рассуждения
- превращал размытые идеи в чёткие инструкции для моделей

Никита Шморин

AI Orchestrator / Agent Operator

- управлял Cursor + AI agents
- разбивал задачи на цепочки промптов
- следил, чтобы агенты "не сходили с ума"

Александр Семёнов

AI QA / Signal Analyst

- валидировал поведение системы
- проверял метрики, деградацию, корректность
- ловил галлюцинации AI и возвращал в реальность

Александр Вяткин

AI UX Hacker / Vibe Engineer

- генерировал UI через AI
- доводил UX до "вау-эффекта"
- делал презентацию (ChatGPT + Gamma)

Можно бесконечно смотреть

как горит огонь

как течёт вода

как AI agent пишет код

AI-native разработка

AI — не фича

AI – разработчик

Мы не ускорили процесс

Мы его заменили

Скорость разработки

Как мы работали

ChatGPT + Gemini

ТЗ

Cursor + AI agents

Структура проекта

AI agents Opus 4.6 + Composer 1.5 + GPT 5.3 Codex

Код

AI Agents + MCP Playwright

Фиксы

AI

UI

AI Cursor + Opus 4.6 + ChatGPT + Gamma

Презентация

Мы — оркестраторы

Наша роль: мы профессионально пьём чай, пока AI пишет код

Зачем нужно нагрузочное тестирование

Сервис работает стабильно на 10 пользователях, но что произойдёт при нагрузке в 10 000 пользователей? Нагрузочное тестирование отвечает на критически важные вопросы, которые определяют успешность вашего продукта в продакшне.

Максимальный RPS

Какой пиковой нагрузки выдерживает сервис без отказов?

Точка деградации

При какой нагрузке начинается нелинейный рост латентности?

Безопасный RPS

Какой уровень нагрузки гарантированно безопасен для продакшена?

Узкое горлышко

Где: CPU, память, сеть или база данных?

Существующие инструменты тестирования – такие как ghz, k6 и Locust – не всегда поддерживают gRPC «из коробки», требуют дополнительной кодогенерации или не предоставляют real-time визуализацию метрик.

И ВСЁ ЭТО – СОБРАНО AI ЗА НЕДЕЛЮ

Что мы в итоге получаем

Load testing platform

🚀 load testing (ramping / const rps)

📊 **real-time** метрики

⚙️ авто-детекция деградации

Возможности

gRPC methods discovery by reflection

Request Data Auto-Filling

- Load testing за **10–15 минут** вместо нескольких часов
- Точка деградации – **автоматически** без ручного анализа
- Нагрузка и **real-time** метрики – в одном приложении

> **Мы не просто использовали AI**
>
> **Мы сделали инструмент, который экономит часы каждой команде**

Базовые термины

Для понимания результатов нагрузочного тестирования важно знать ключевые метрики и термины.

Термин	Описание
RPS (Requests Per Second)	Количество запросов в секунду – основная метрика нагрузки
Latency	Время ответа на один запрос (в миллисекундах)
p50 / p95 / p99	Перцентили латентности. p99 = 99% запросов быстрее этого значения
Throughput	Реальная пропускная способность (фактический RPS)
Concurrency	Количество параллельных соединений / горутин
Error Rate	Процент неуспешных запросов
Точка деградации	RPS, при котором латентность или ошибки начинают расти нелинейно
HDR Histogram	Алгоритм для точного расчёта перцентилей без большого потребления памяти

Генератор данных

Тестирование с одинаковым payload приводит к кэшированию на стороне сервиса и даёт нереалистичные результаты. nil-loader решает эту проблему.

Шаблоны с Faker-токенами

Пользователь задаёт шаблон запроса с специальными токенами:

```
{
  "user_id": "{{faker.uuid}}",
  "name": "{{faker.name}}",
  "email": "{{faker.email}}",
  "phone": "{{faker.phone}}"
}
```

Каждый запрос получает уникальные данные. Доступные токены: `uuid`, `email`, `name`, `firstName`, `lastName`, `phone`, `username`, `sentence`, `timestamp`, `int`, `bool`, `safeAlphaNum`.

CSV-источник

Подстановка значений из CSV в запросы через `{{csv.column}}`.

Подходит для кастомных и реальных данных (например, `order_id`, `user_id`, способы оплаты) и точечной проработки нужных кейсов.

Точка деградации

> Формулу и подход "точки деградации" разработала нейросеть, опираясь на best practices индустрии.

Baseline p99

Первые 5 секунд – сбор нормальной латентности при низкой нагрузке

Проверка условий

Каждую секунду проверяется: p99 вырос > 2.5x от baseline? Или error rate > 15%?

Фиксация точки

Если условие выполнилось – фиксируем точку деградации и останавливаем тест

Degradation at

RPS, при котором началась деградация

Recommended

Безопасный RPS = 80% от точки деградации

Max stable

Максимальный RPS при стабильной работе

Именно нейросеть подсказала это определение, собрала формулу и предложила её как практичное решение на основе best practices индустрии.

Эффект от использования в цифрах

MVP версия nil-loader уже сегодня может дать ощутимый практический эффект для команд, которые тестируют gRPC-сервисы.

10

минут до первого отчёта

0

строк клиентского кода. Нет нагрузочных скриптов.

4

ключевые метрики в одном экране

1 = 3

1 прогон = 3 результата

2

инструмента в одном. Нагрузка и метрики

Что и как мы использовали в итоге

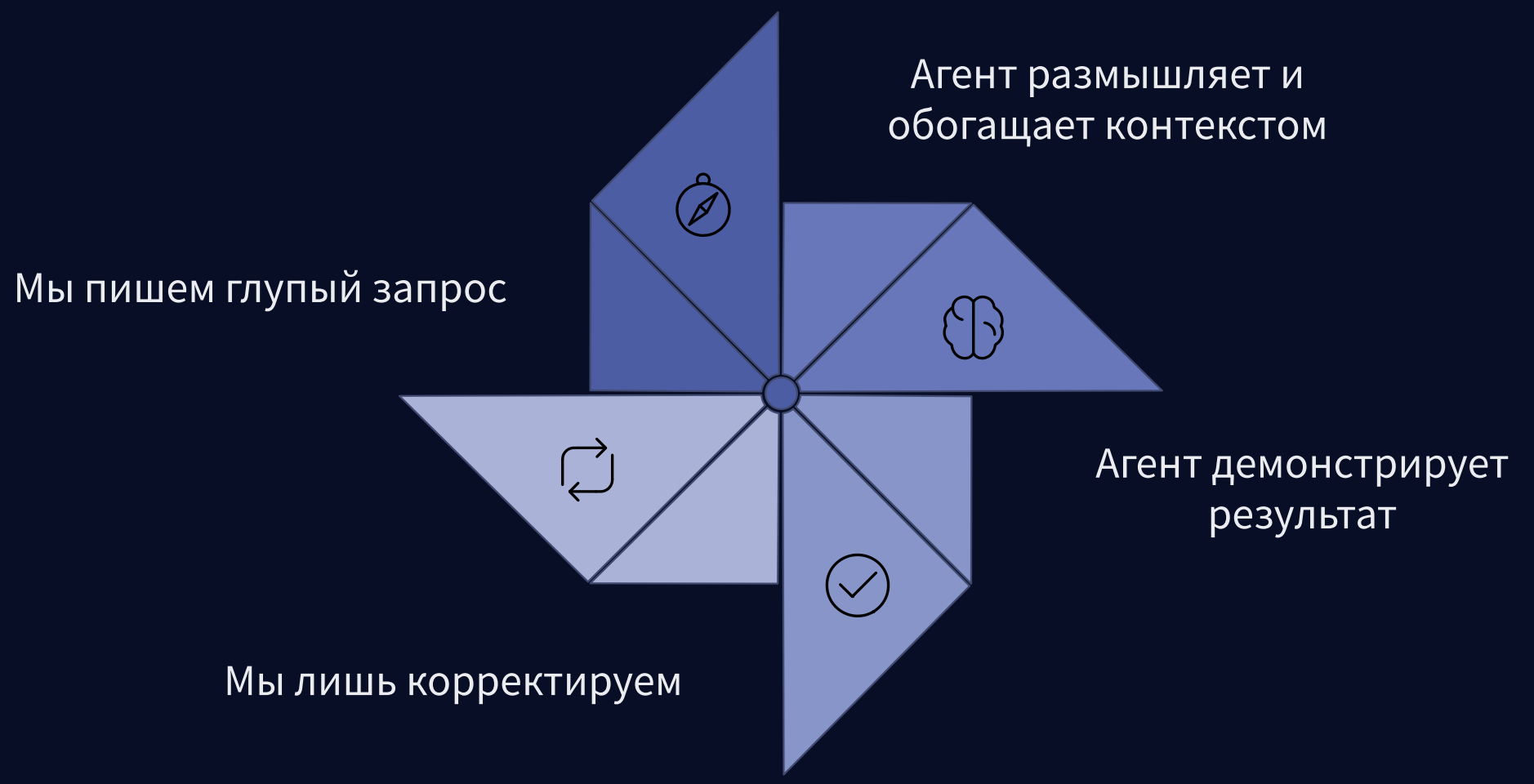
Выбранные компоненты обеспечивают высокую производительность и простоту интеграции

Компонент	Технология	Обоснование
Backend	Go 1.25	Высокая производительность, простая конкуррентность через goroutines, статическая сборка
gRPC Client	jhump/protoreflect	Dynamic Client без кодогенерации, работает с reflection и .proto файлами
WebSocket	gorilla/websocket	Надёжная реализация, удобная интеграция с HTTP-сервером
Frontend Charts	Chart.js 4	Интерактивные графики для дашборда MVP
Smart IDE	Cursor + AI agents	Широкий выбор агентов в одной среде
AI Agents	Opus 4.6, Composer 1.5, GPT Codex 5.3	Лучшие модели для кодинга

Минимальные зависимости, максимальное использование стандартной библиотеки Go

Промпт инжениринг нам не очень то и нужен

При работе с кодовой базой и четком понимании требований, промпт инжениринг в значительной степени излишен. Методология RACE в таких случаях также не нужна, поскольку модель уже обладает всем основным контекстом и сама его неплохо себе создает.



Что действительно работает

- Базовые архитектурные принципы
- Использование MCP для доступа к реальной среде
- Дебаг в процессе разработки
- Итеративное влияние на результаты

> Модель сама доходит до нужного результата через рассуждения, нам нужно только направлять

Борьба с галлюцинациями AI

Использование продвинутых AI-агентов требует надёжных методов верификации.

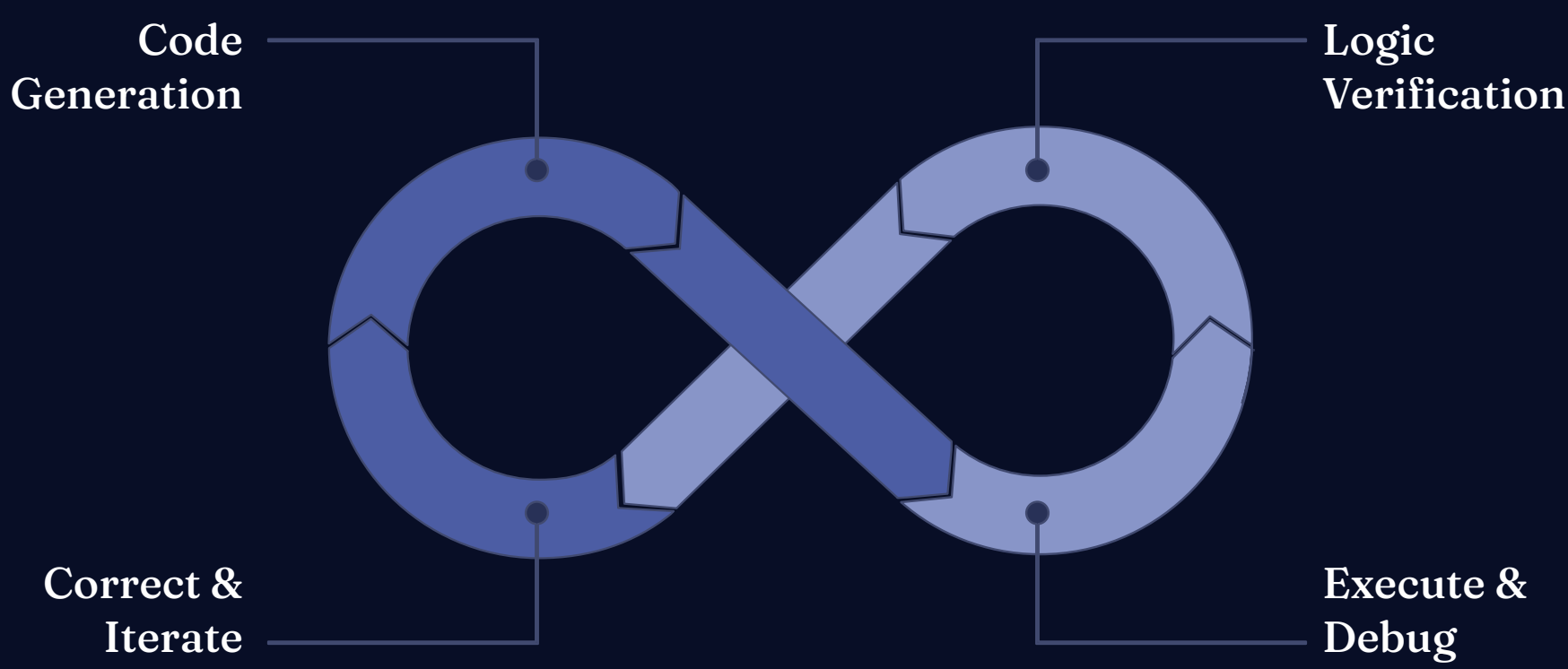
- **Что такое галлюцинации:** Нейросети генерируют правдоподобный, но неверный код или информацию, которые выглядят убедительно, но не работают.
- **Почему происходят:** Модели предсказывают следующий токен на основе вероятностей, а не фактических знаний. Они могут "выдумывать" детали, которые звучат логично.

Двухагентная проверка

- Первый агент (Composer) генерирует код
- Второй агент (Opus) перепроверяет логику и корректность

MCP Playwright для верификации

- Прямой доступ к браузеру
- Запуск решения в реальной среде
- Дебаг консоли и сетевых запросов
- Корректировка кода на основе реальных результатов



Эта комплексная система обеспечивает непрерывную проверку и корректировку AI-генерируемого кода, минимизируя галлюцинации и повышая надежность.

Агент юзает очень много ИИ, ии.. что делать?

Проблема:

Два агента (Composer + Opus) генерируют много стандартных команд (grep, glob, git log и т.д.)

- Каждый вызов инструмента расходует контекст
- Риск упереться в лимиты токенов при работе с большими проектами

Решение - RTK (https://www.rtk-ai.app/):

- Подписывается на хуки агента
- Перехватывает вызовы команд
- Подменяет полный вывод на сокращённый, сохраняя суть
- Результат: сокращение контекста вполовину без потери смысла

Без RTK (примерно 1430 токенов)

Полный вывод git log --stat -10 с полными коммитами, авторами, датами, полным списком изменённых файлов и статистикой по каждому файлу, подсказками

С RTK (сокращено вполовину)

Компактный формат:

- Ветка: master...origin/master
- Только измененные файлы
- Неотслеживаемые файлы: tests/

Итог: Эффективное использование контекста позволило работать с двумя агентами без потенциального упора в лимиты и +- сохранения смысла

Функциональность MVP

1

Запуск нагрузочного теста

Тестирование одного gRPC-метода (unary) с режимами Constant и Ramping

2

Обнаружение сервисов

Автоматическое обнаружение через gRPC Reflection и загрузка .proto файлов

3

Генерация данных

Faker-шаблоны и CSV-источник для создания тестовых данных

4

Real-time дашборд

Визуализация p50/p95/p99, Target vs Actual RPS, Success Rate, ошибки по gRPC-кодам

5

Автодетектор деградации

Расчёт точки отказа и рекомендуемого RPS

6

Экспорт отчёта

JSON-отчёт по результатам теста для анализа

Ограничения и интеграции

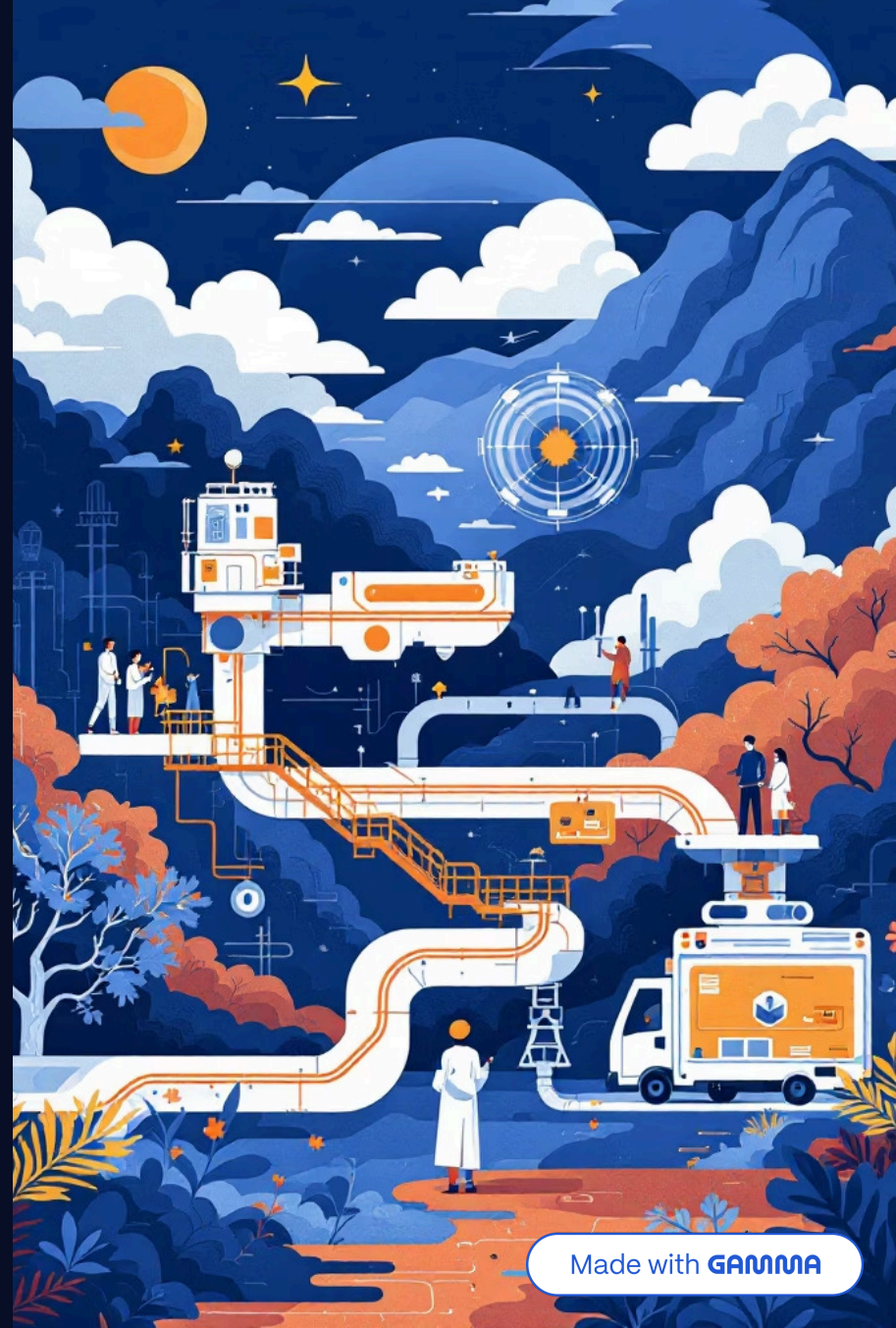
Текущие ограничения

- Один gRPC-метод за раз
- Только unary-методы
- Данные в памяти
- Нет истории тестов
- Один инстанс

CI/CD интеграция (to be)

nil-loader можно запускать в пайплайне (GitLab CI / GitHub Actions) как отдельный шаг для автоматической проверки SLA.

- Поднять тестовый сервис
- Запустить nil-loader
- Выполнить Ramping-тест
- Проверить метрики



Будущее развитие

QA Profile

QA могут настраивать платформу под себя. Свои сервисы, свои шаблоны.

История тестов

Сохранение и сравнение результатов

HTTP/REST

Поддержка помимо gRPC

Streaming-методы

Client/server/bidi streaming

CI/CD интеграция

CI/CD интеграция с автоматической проверкой SLA

Итоги

Real-time UI Графики, логи, ошибки через WebSocket	Умная нагрузка Constant / ramping + worker pool
Уникальные данные Faker + CSV шаблоны	Автодетекция деградации Точка отказа, рекомендуемый RPS
Точная телеметрия HDR Histogram, посекундные snapshots	Без кодогенерации Reflection + dynamic messages

AI Contribution 100%.

Неделя работы вместо 2 месяцев.

Переосмысление процесса разработки.

Спасибо за внимание

Проект разработан командой nil-team

Участники:

- Владимир Гальцев
- Никита Шморин
- Александр Семёнов
- Александр Вяткин
- AI Agents