РСК

Группа компаний

От хаоса к прозрачности: GraphQL для унификации управления HPC-инфраструктурой

Блок 1:

Мышление графами в HPC-инфраструктуре

От хаоса к графам - проблема HPC

Как мы описываем HPC-кластер словами?

"В нашем кластере есть вычислительные узлы, на которых запускаются задачи через Slurm. У каждого узла есть диски, которые объединяются в RAID-массивы для хранилища. Хранилище предоставляется через Lustre или NFS сервисы, которые монтируются на клиентских узлах..."

Традиционный подход: множество разрозненных API

- REST API Slurm: /nodes, /jobs, /partitions
- REST API Storage: /storages, /raids, /disks
- REST API Monitoring: /metrics, /alerts

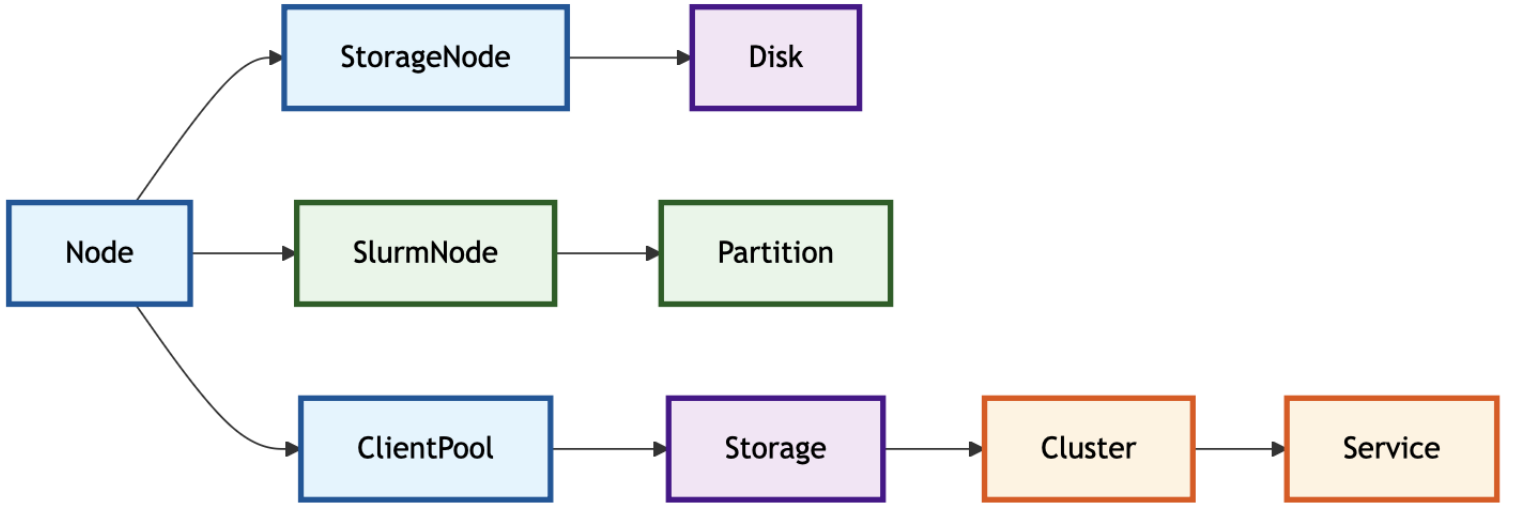
Проблема: ментальная модель ≠ API модель

- Мы мыслим связями, но работаем с isolated endpoints
- Каждый запрос - отдельный HTTP call
- Клиент собирает пазл из разрозненных данных

GraphQL - мыслить графами

Ключевая идея GraphQL: "Model your business domain as a graph"

HPC как граф связей



Естественное описание на GraphQL

```graphql
type Node {
  hostname: String!
  # Связи отражают реальную архитектуру
  storageNode: StorageNode     # У узла есть хранилище
  slurmNode: SlurmNode         # У узла есть состояние в Slurm
  clientPools: [ClientPool!]   # Узел может быть клиентом
}
```

Общий язык команды - Shared Language

Как мы говорим о HPC-системе?

Системный администратор:

"У нас есть пул узлов для вычислений, которые подключены к распределенному хранилищу Lustre. Данные лежат на OSS серверах с RAID-массивами из NVMe дисков."

Пользователь кластера:

"Я запускаю задачи на партицию, а мои данные лежат в папке, которая примонтирована с файлового сервера."

DevOps инженер:

"Мне нужно отмониторить состояние всех дисков в хранилище и посмотреть, какие задачи сейчас выполняются на узлах."

GraphQL схема как общий словарь

```
# Каждый тип - это существительное из нашего языка
type Node         # Узел кластера
type Storage      # Система хранения
type Job          # Вычислительная задача
type Disk         # Физический диск

# Связи - это отношения из нашего языка
type Storage {
  clusters: [StorageCluster!]!   # "хранилище состоит из кластеров"
  clientPools: [ClientPool!]!    # "хранилище доступно пулам клиентов"
}

type Node {
  slurmNode: SlurmNode           # "узел имеет состояние в Slurm"
  storageNode: StorageNode       # "узел участвует в хранилище"
}
```

Граф вместо endpoints

Старый способ (REST): разрозненные запросы

- N+1 queries проблема
- Клиент управляет логикой агрегации
- Over-fetching: получаем много лишних полей

```
# Получить список узлов
GET /api/slurm/nodes

# Для каждого узла получить диски
GET /api/storage/nodes/{node}/disks

# Для каждого диска получить метрики
GET /api/monitoring/disks/{disk}/metrics
```

Новый способ (GraphQL): один запрос по графу

- Один HTTP запрос
- Точно те данные, которые нужны
- Сервер оптимизирует выборку данных

```
query ClusterHealth {
  nodes {
    hostname
    slurm {
      state         # Только нужные поля
      cpus
    }
    sod {
      storageDisks {
        name
        sizeBytes
        state       # Только состояние диска
      }
    }
  }
}
```

Типы как доменная модель

Полиморфизм в HPC

```
# Базовый интерфейс для всех сервисов хранения
interface StorageService {
  name: String!
  state: StorageState!
  raid: RAID
  metrics: StorageMetrics!
}

# Конкретные реализации для разных технологий
type LustreService implements StorageService {
  # ... общие поля из интерфейса
  type: LustreType!        # mgs, mds, oss
  transport: LustreNetType! # o2ib (InfiniBand), tcp
  stats: [LustreStat!]     # Специфичная статистика Lustre
}

type NFSService implements StorageService {
  # ... общие поля из интерфейса
  options: String!              # "rw,sync,no_root_squash"
  transport: NFSTransport!      # tcp, rdma
  virtualAddresses: [String!]   # IP для HA
}
```

Живые примеры запросов

```
query StorageOverview {
  storages {
    name
    type          # lustre, nfs
    state         # HEALTHY, DEGRADED, etc.
    metrics {
      capacityBytes
      usedBytes
      throughputBytesPerSec
    }
    clusters {
      name
      services {
        __typename    # Узнаем конкретный тип сервиса
        name
        state
        ... on LustreService {
          type        # mgs, mds, oss
          transport   # o2ib, tcp
        }
        ... on NFSService {
          options     # "rw,sync,no_root_squash"
          virtualAddresses
        }
      }
    }
  }
}
```

Сценарий 1:

"Покажи мне состояние хранилища"

Мысленная модель:

"Я хочу увидеть все хранилища, их состояние, сколько места занято, и какие сервисы на них работают."

Живые примеры запросов

```
query NodeDiagnostics {
  nodes(filter: {hostname: "node-05"}) {
    hostname
    # Железо и диски (от Disk Jockey)
    dj {
      disks {
        name
        sizeBytes
        driveType      # SSD, HDD
        vendor
        serialNumber
      }
      nvmeDevices { devicePath modelNumber physicalSize firmware }
    }
    # Состояние в Slurm
    slurm {
      state          # IDLE, ALLOCATED, DOWN
      cpus
      realMemory
    }
    # Участие в хранилище
    sod {
      storageDisks { name state sizeBytes }
    }
  }
}
```

Сценарий 2:

"Диагностика проблемного узла"

Мысленная модель:

"Узел node-05 ведет себя странно. Покажи все его железо, состояние в Slurm, и какие диски он предоставляет хранилищу."

Мутации - управление через граф

```
mutation CreateLustreOSS {
  createLustreService(input: {
    service: {
      storageName: "production-lustre"    # К какому хранилищу относится
      clusterName: "oss-cluster"          # На каком кластере запустить
      raid: {
        name: "oss-raid-01"
        disksSource: {
          diskPool: { name: "nvme-pool" }  # Откуда брать диски
        }
        zfs: {
          level: raidz2                    # RAID6 эквивалент в ZFS
          disks: 6                         # Количество дисков
        }
      }
    }
    type: oss                             # Тип Lustre сервиса
    transport: o2ib                       # InfiniBand транспорт
  }) {
    # Что нам вернуть после создания
    name
    state
}
```

Сценарий:

"Создать новый Lustre OSS"

Мысленная модель:

"Мне нужно создать Object Storage Server для Lustre на кластере oss-cluster, использовать диски из nvme-pool, настроить RAID-6 (raidz2), и подключить через InfiniBand."

От хаоса к прозрачности

Было: множество систем, разные API

Стало: единый граф предметной области

```
type Node {
  slurm: SlurmNode       # ┐
  sod: StorageNode       # ├── Все связи
  dj: DJ                 # │   в одной схеме
  clientPools: [...]     # ┘
}
```

Преимущества "мышления графами"

- **Естественность** - схема отражает ментальную модель
- **Эффективность** - один запрос вместо множества
- **Типизация** - ошибки находятся на этапе разработки
- **Эволюция** - схема растет без breaking changes
- **Документация** - схема сама себя документирует

Блок 2:

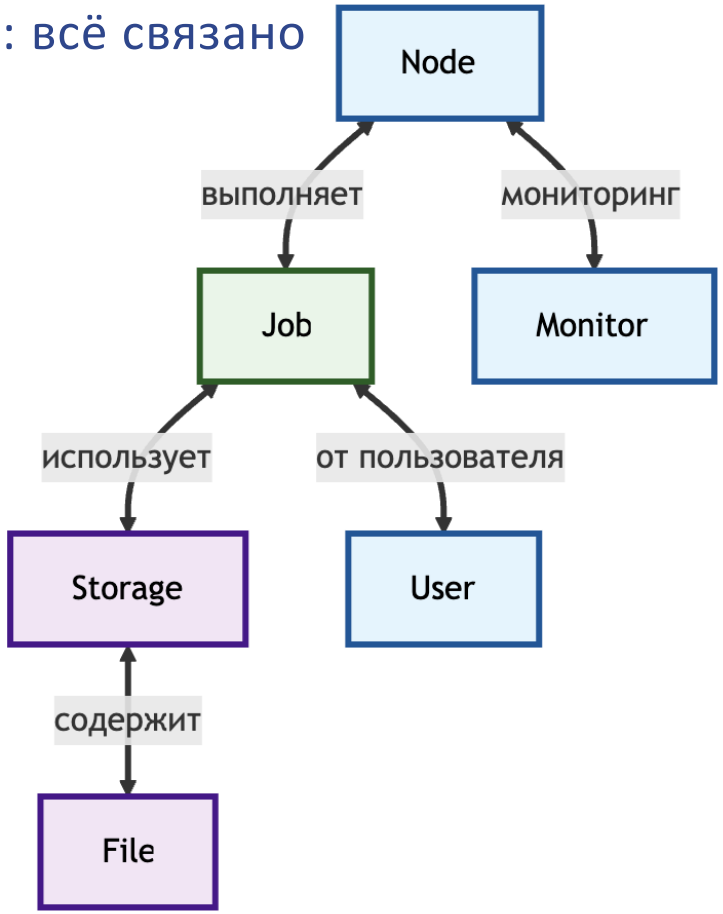
GraphQL Federation - мышление связанным графом

"Что если у нас есть отдельные GraphQL API для Slurm, Storage и Monitoring? Как объединить их в единый граф?"

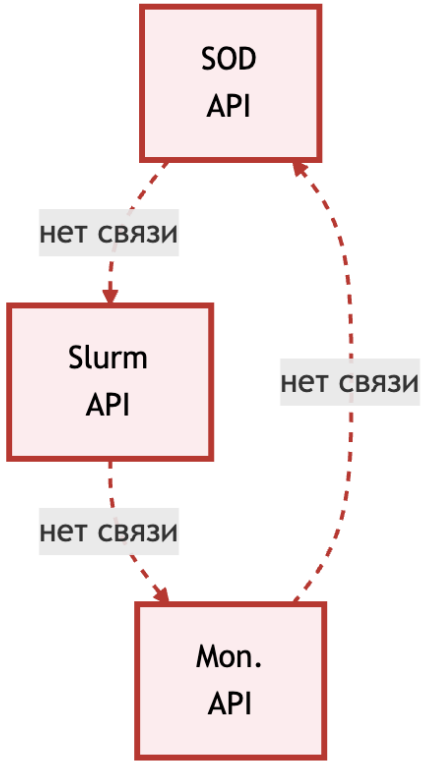
Фрагментированные системы разрывают граф понимания

"В нашем кластере есть вычислительные узлы, на которых запускаются задачи через Slurm. У каждого узла есть диски, которые объединяются в RAID-массивы для хранилища. Хранилище предоставляется через Lustre или NFS сервисы, которые монтируются на клиентских узлах..."

Реальность: всё связано



Инструменты: всё разделено



Конкретная проблема из нашей схемы

Вопрос: "На каком диске лежат данные задания job-123456?"

Сегодня нужно:

1. Slurm API → где выполняется задание? → node05.cluster
2. SOD API → какие диски на node05? → nvme0n1, sda1
3. Файловая система → где папка задания?
   → /scratch/user1/job-123456
4. SOD API → какой диск для /scratch? → nvme0n1

Проблема: 4 запроса, ручная логика, хрупкие предположения.

Потерянное знание о связях

```
# Что ДОЛЖНО быть возможно:
query JobStorage($jobId: ID!) {
  job(id: $jobId) {
    workingDirectory
    nodes {
      hostname
      disks { device mountPath }  # ← Эта связь потеряна!
    }
  }
}
```

Сейчас это невозможно — разные API не знают друг о друге.

Federation восстанавливает связность через типы

Ключевая идея: один тип - много источников знания

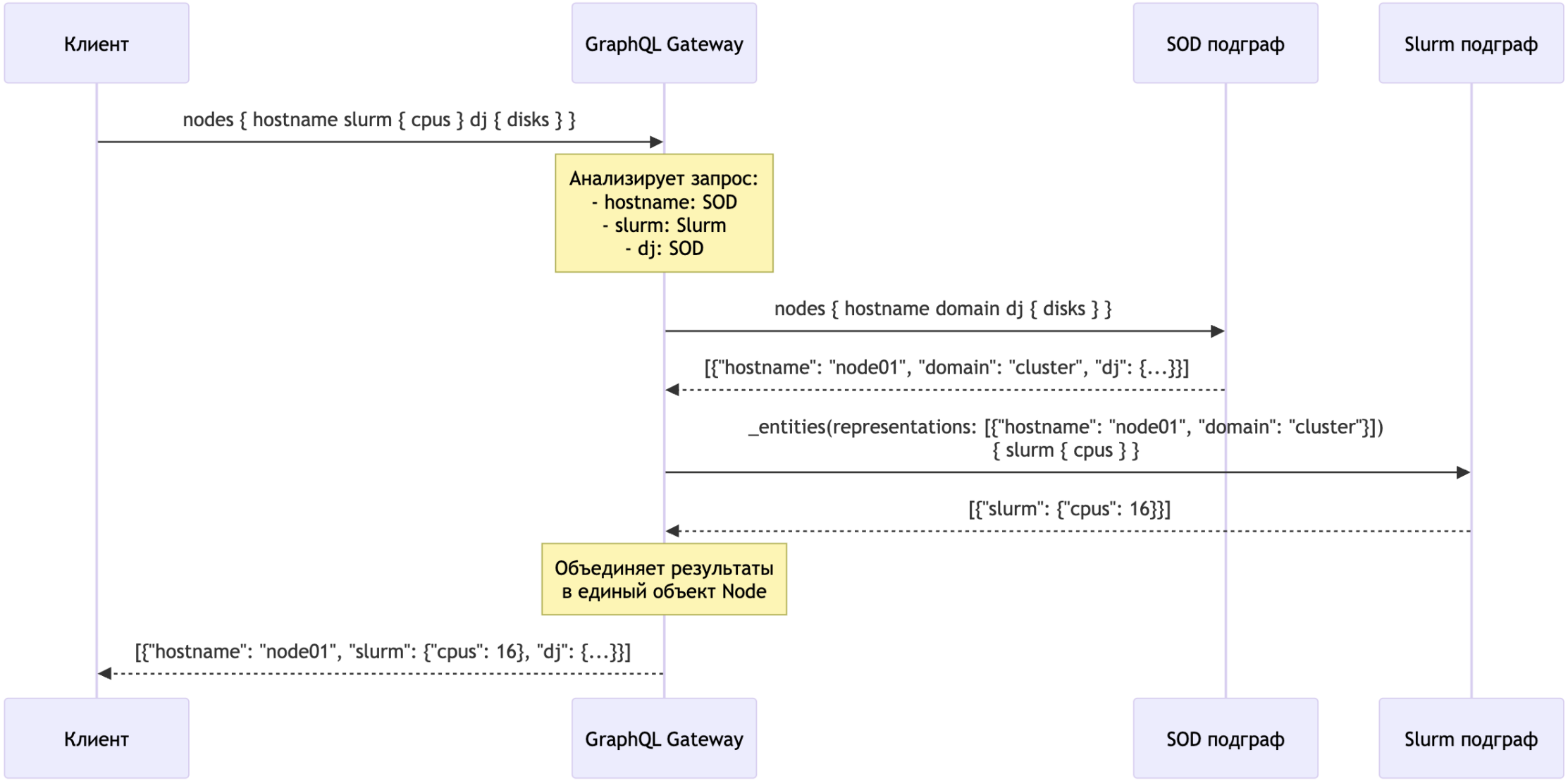
"Тип Node существует один, но каждый подграф знает о нём что-то своё. Federation соединяет эти знания в единый объект."

```
# SOD подграф: "владелец" типа Node
type Node @key(fields: "hostname domain") {
  hostname: String!
  domain: String!
  dj: DJ!                # Знаю про железо
  sod: StorageNode!      # Знаю про хранилище
}

# Slurm подграф: расширяет Node своим знанием
type Node @key(fields: "hostname domain") {
  hostname: String!
  domain: String!

  slurm: SlurmNode             # Добавляю своё знание
}
```

Как Gateway собирает единый объект



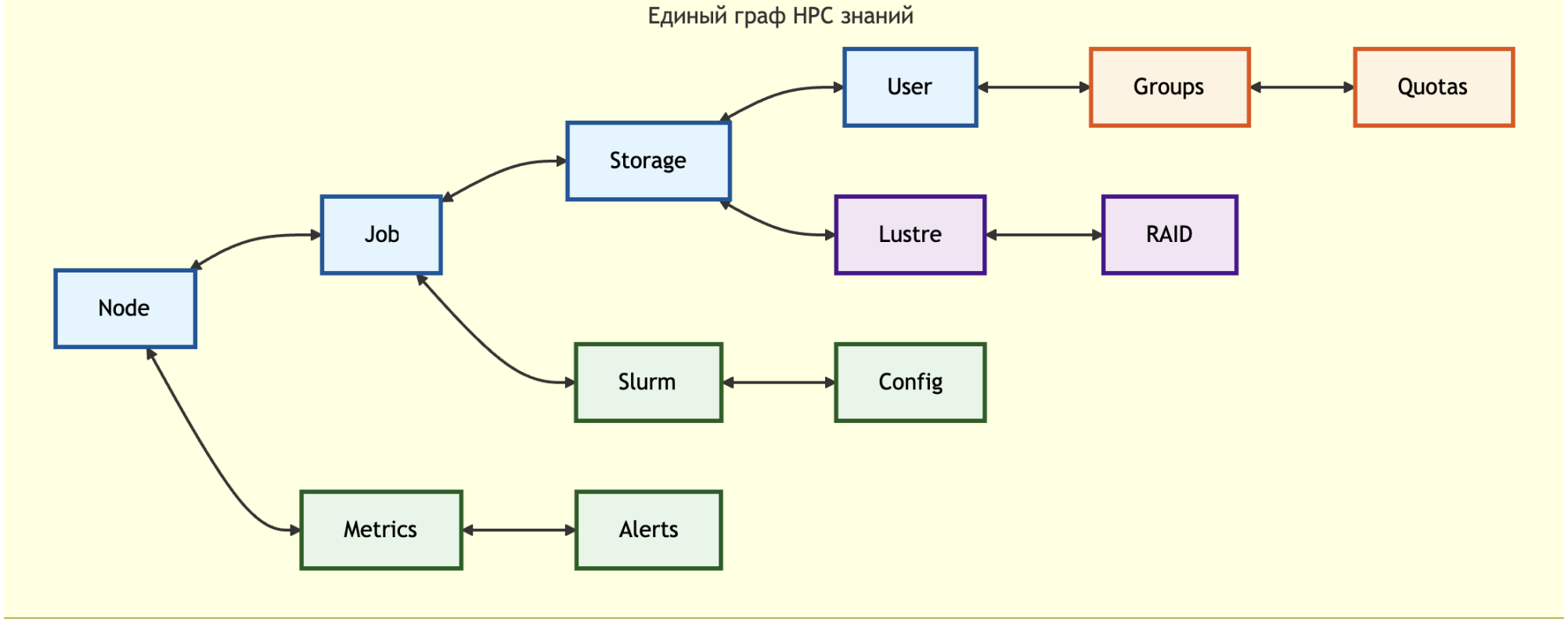
Блок 3:

LLM-агенты - интеллектуальная навигация по графу знаний

От связанного графа к интеллектуальному взаимодействию

"Federation восстановила связи, но граф HPC-системы огромен. Тысячи узлов, миллионы связей, сотни типов данных. Как навигировать в этой сложности?"

НО: Граф стал ОГРОМНЫМ и СЛОЖНЫМ!



Барьер сложности для пользователей

Проблемы навигации:

- Как найти нужную информацию в огромном графе?
- Какой путь по связям даёт оптимальный ответ?
- Как связать проблему в одной части с причиной в другой?

Ключевая идея: LLM становится экспертом по структуре графа и умеет:

- **Понимать семантику** связей между типами
- **Планировать навигацию** по оптимальным путям
- **Переводить** между естественным языком и GraphQL
- **Анализировать** паттерны в данных

Пример сложности:

```
# Чтобы понять "почему job медленный", нужно знать:
query JobPerformanceDebug {
  job(id: $jobId) {
    nodes {                          # Где выполняется
      slurm { allocMemory }          # Сколько памяти выделено
      dj {
        disks { temperature }  # Температура дисков
        nvmeDevices { wearLevel } # Износ SSD
      }
      metrics {                      # Текущая нагрузка
        cpuUsagePercent
        memoryUsagePercent
        diskIOPS
      }
    }
    storage {                        # Где хранятся данные
      clusters {
        services { metrics { throughputBytesPerSec } }
      }
    }
  }
}
```

Интроспекция как карта для навигации по графу

GraphQL даёт LLM структурную карту

"Интроспекция превращает схему в живую документацию. LLM изучает граф и становится экспертом по его структуре."

LLM понимает семантику связей

```
# Из интроспекции LLM изучает, что:
type Node {
  hostname: String!
  slurm: SlurmNode      # Node СВЯЗАН с планированием задач
  dj: DJ!               # Node СОДЕРЖИТ железо
  metrics: NodeMetrics # Node ИМЕЕТ состояние
}

type Job {
  id: ID!
  nodes: [Node!]!       # Job ВЫПОЛНЯЕТСЯ на узлах
  user: User!           # Job ПРИНАДЛЕЖИТ пользователю
}

type Storage {
  disks: [Disk!]!       # Storage СОСТОИТ из дисков
  services: [StorageService!]! # Storage ПРЕДОСТАВЛЯЕТ сервисы
}
```

Автоматическое планирование запросов

Пользователь: "Почему мои задания медленные?"

LLM планирует навигацию:

1. User → найти пользователя
2. User.jobs → получить задания
3. Job.nodes → узлы выполнения
4. Node.metrics + Node.dj.disks → проверить нагрузку и железо
5. **Автоматически формирует оптимальный запрос**

```
query DebugSlowJobs($userId: ID!) {
  user(id: $userId) {
    jobs(filter: {state: RUNNING}) {
      id name
      nodes {
        hostname
        metrics { cpuUsagePercent memoryUsagePercent }
        dj {
          disks { temperature readErrors }
        }
      }
    }
  }
}
```

Практические сценарии - интеллектуальное решение задач

Принцип работы:

- LLM получает естественноязыковую задачу от оператора
- Использует introspection для понимания структуры данных
- Планирует оптимальную последовательность GraphQL-запросов
- Интерпретирует результаты и предлагает решения

Автоматическое планирование запросов: LLM анализирует схему и автоматически выбирает стратегию:

- Минимальные запросы для получения нужных данных
- Использование федеративных связей между сервисами
- Оптимизация с учётом производительности и прав доступа
- Контекстное понимание предыдущих запросов в диалоге

Интеллектуальная маршрутизация: LLM понимает топологию федерации и при запросах, требующих данные из нескольких сервисов, автоматически определяет минимальный набор сервисов и планирует порядок **запросов.**

PCK BasIS
AI платформа управления

HUMAN:

Я хочу увидеть все хранилища, их состояние, сколько места занято, и какие сервисы на них работают.

AI:

Вот информация о доступных хранилищах в вашей системе:

Хранилище: lstore

- **Состояние**: LIVE (активно и работает)
- **Использовано места**: 20.00 MB
- **Общий объем**: 8.45 TB
- **Доступно свободного места**: 8.45 TB

Хранилище использует технологию Lustre (высокопроизводительная распределенная файловая система) и находится в полностью рабочем состоянии. На данный момент используется очень небольшой объем пространства - менее 1% от общей емкости.

В системе обнаружено только одно хранилище с именем "lstore". Если вы хотите получить информацию о сервисах, работающих на этом хранилище, или о кластерах, его составляющих, я могу дополнительно запросить эту информацию.

PCK BasIS
AI платформа управления

Сделай отчет об использовании вычислительных ресурсов для аккаунта rsc за первые полгода 2024

AI:

Вот отчет об использовании вычислительных ресурсов для аккаунта **rsc** за первые полгода 2024 года:

Общая статистика:

- **Количество задач (jobs)**: 31
- **Общее время выполнения задач**: 2116 секунд
- **Общее CPU-время**: 2116 секунд
- **Эффективность использования CPU (CPU Efficiency)**: 100%
- **Период отчета**:
  - Начало: 2024-01-01
  - Окончание: 2024-06-30

Интерпретация данных:

Все задачи, запущенные от имени аккаунта **rsc** в указанный период, показали высокую эффективность использования процессорного времени — **100%**, что говорит о том, что ресурсы использовались оптимально без потерь. Общее количество задач — 31, суммарное время работы и CPU-время совпадают, что может указывать на однопоточные задачи или равномерное распределение нагрузки.

Если вам нужна более детальная информация по пользователям или конкретным задачам в рамках этого аккаунта, я могу дополнительно запросить эти данные.

Спасибо!

РСК

rscgroup.ru
hq@rsc-tech.ru