

Анатомия зависания: когда
thread pool закончился, а
CPU почти пустой

Дисклеймер

Данный доклад никоим образом не связан с компаниями, с которыми автор сотрудничает.

Акт I

Узнайте свой сервис или
почему провал близко

Как там прод?

- Система работает стабильно
- Багов нет
- Работаем “нормально”



ШЕФ! ВСЁ ПРОПАЛО!

Как там прод?

- Если у вас сейчас все нормально - это не значит, что проблемы нет.
- Скорее всего, вы просто еще не дошли:
 - партнер не деградировал
 - БД не уперлась в лимиты
 - тяжелый запрос не "поймал" комбинацию условий
- Но провал может быть рядом - симптомы приходят резко.

Без глубокого понимания
“как это устроено”
нет уверенности...

С пониманием ее тем более
нет.

Но, так интереснее.

Карта доклада

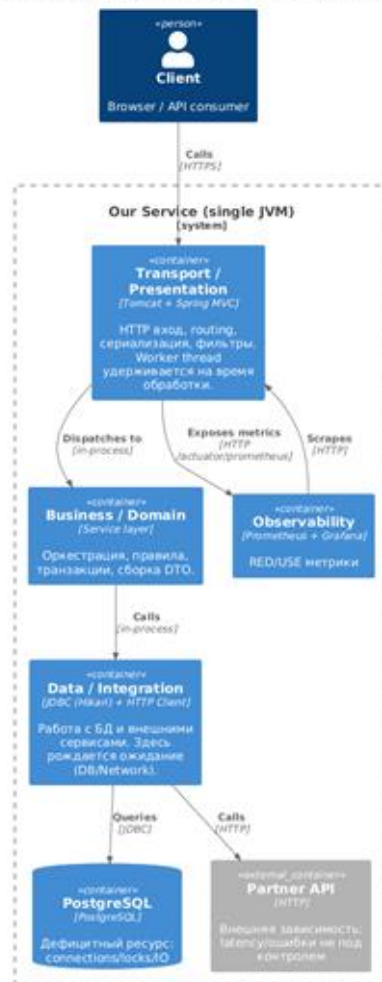
- Акт I: Узнайте свой сервис или почему провал близко
- Акт II: Расследование. RED и USE + инцидент начинается с мелкого бага
- Акт III: Спасаем корабль: Bulkhead-изоляция + Virtual Threads эксперимент
- Акт IV: Лечим тяжелый запрос: очереди, транзакции, circuit breaker, rate limiter

Что разберем

- Типовой сервис и его "скрытые" проблемы
- Observability: RED + USE, как найти root cause
- Паттерны стабильности, которые должны быть "везде"
- Спасем сервис от падения
- Потом вылечим нагруженный эндпоинт
- Virtual Threads: где помогают, где нет

Универсальная 3-tier

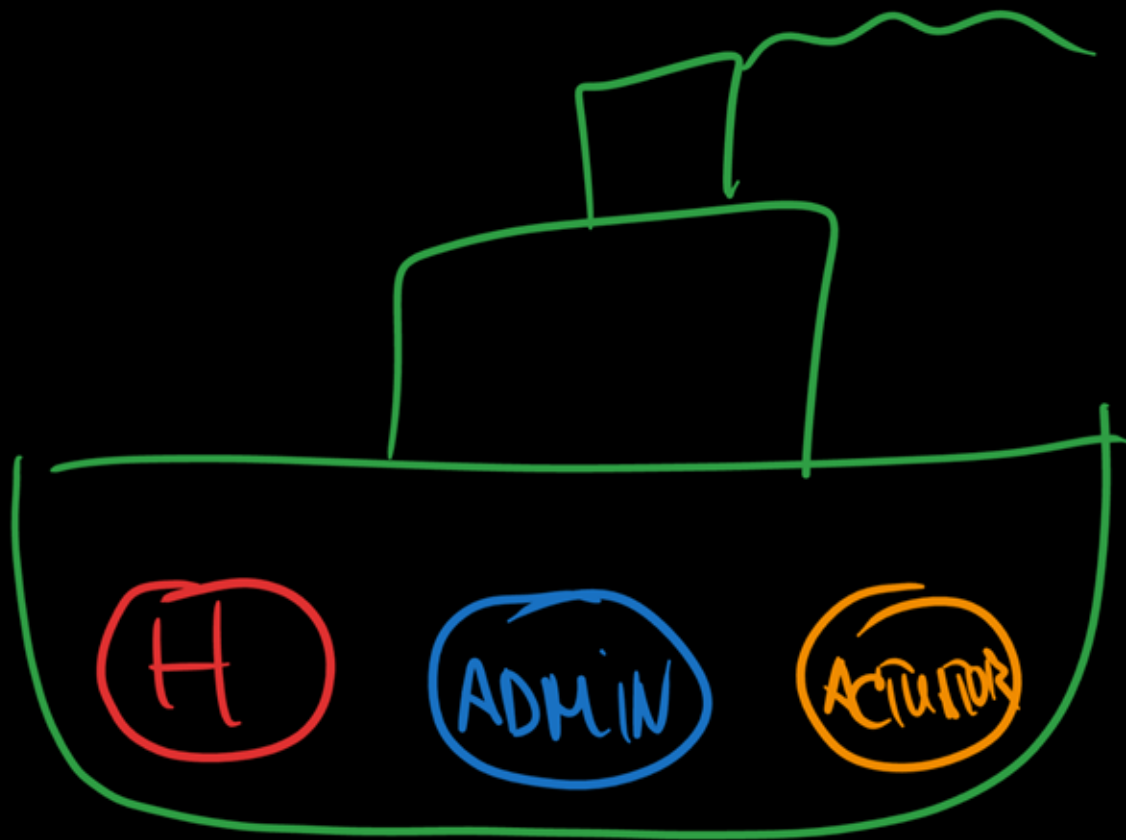
- Presentation/Transport → Business/Domain → Data/Integration
- Привязка к нашему сервису:
 - Presentation: Tomcat + Spring MVC
 - Business: Service layer, orchestration, транзакции
 - Data/Integration: Hikari + Postgres + HTTP partner

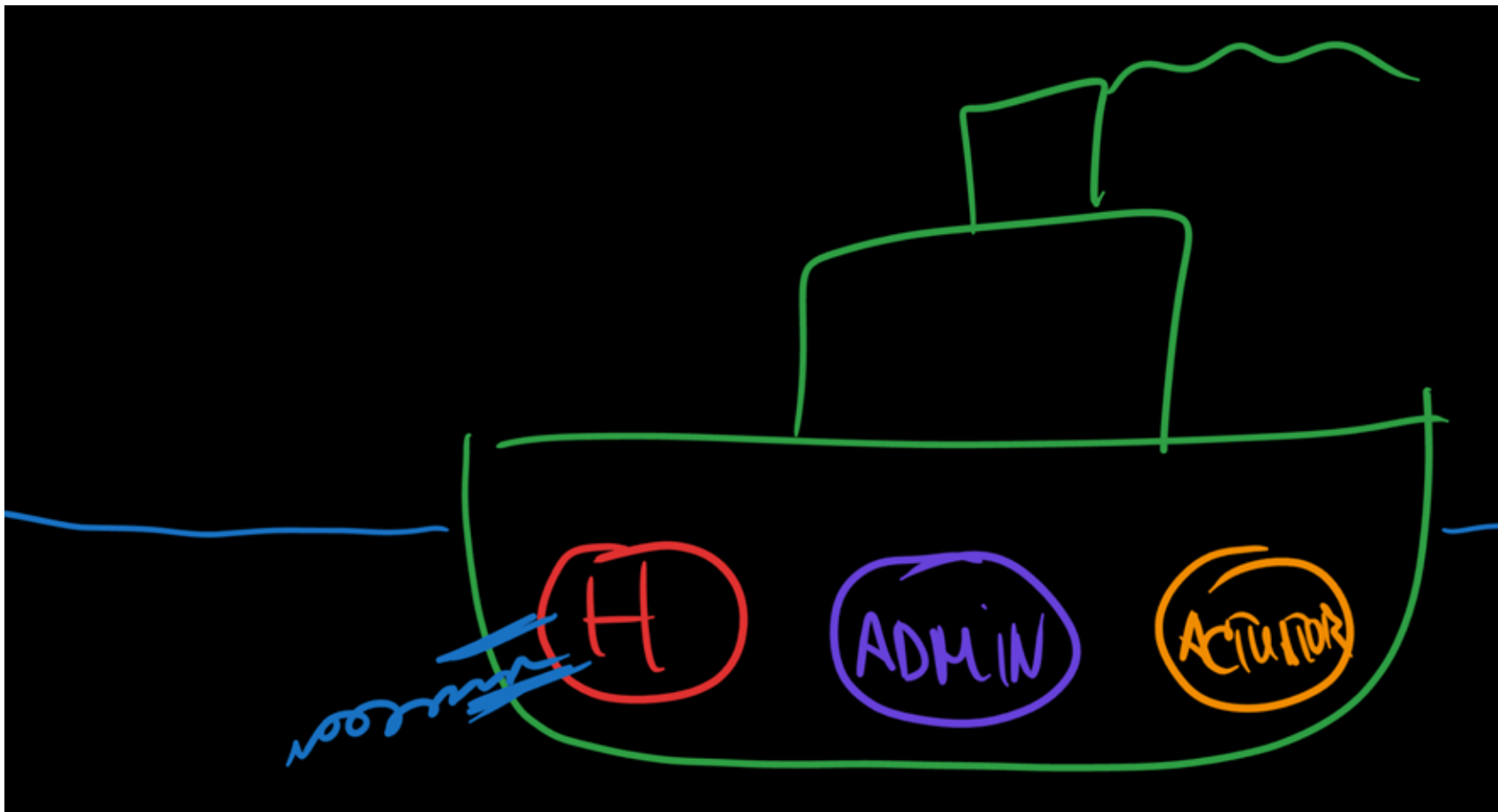


Простое приложения и
"провал" на ровном месте

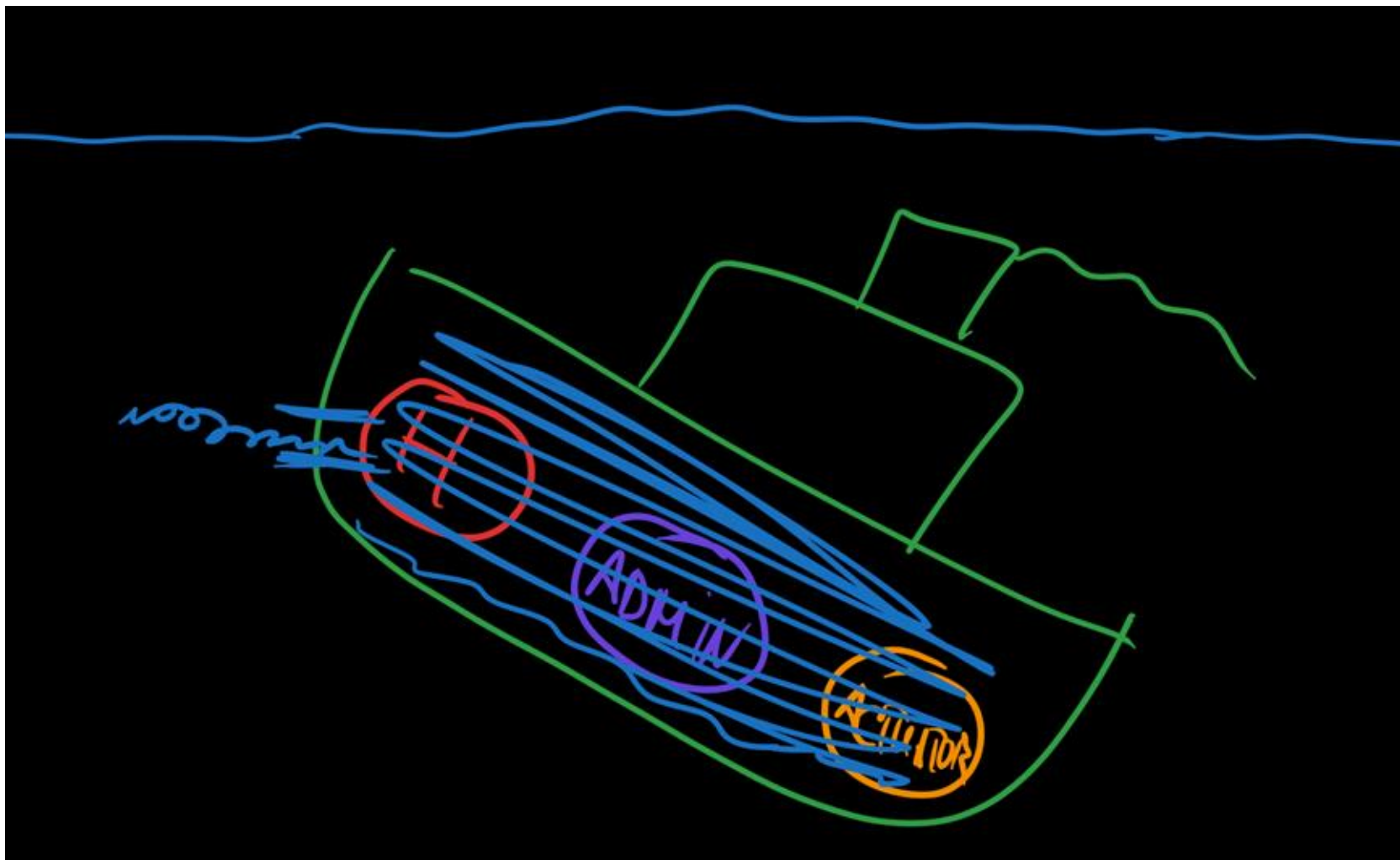
Три эндпоинта

- Heavy: `/api/v1/dashboard/{id}` - SQL + partner HTTP + DTO
- Admin: `/api/v1/admin/status` - должен жить всегда, но ходит в БД
- Actuator: `/actuator/health`, `/actuator/prometheus` - должен отвечать всегда (иначе рестарты/флаппинг)









[CODE IMAGE] Контракты endpoint'ов
(псевдо OpenAPI/контроллеры)

Почему это важно?

- Heavy может деградировать сам
- Но хуже: он может утащить ресурсы и "положить" admin/actuator
- И тогда вы теряете управление системой: мониторинг слепнет, оркестратор начинает "лечить" рестартами

Что мы имеем?

- Что получили: приложение с 3 эндпоинтами и "скрытой связностью" через общие ресурсы.
- Все они делят общие ресурсы приложения

АКТ II

История одного инцидента

Что случилось?

- Легковесный админский `/admin/status` отвечает плохо или с ошибками
- Приложение на пиковых нагрузках начинает перезапускаться

[GRAFANA IMAGE] RED:
Duration/Errors no endpoint'am

[GRAFANA IMAGE] CPU/Memory "все
OK"

Что случилось?

- Получили стартовую точку расследования: "легкие эндпоинты умерли первыми".
- Перестали смотреть на CPU как на главный индикатор.
- Осталось: понять корневую причину?

Методология RED

- Requests: RPS по endpoint
- Errors: 4xx/5xx + timeouts
- Duration: p50/p95/p99 latency
- RED отвечает: "пользователю плохо или хорошо"

Методология USE

- Utilization: сколько ресурса занято
- Saturation: сколько ждет/очередь/признаки упора в лимит
- Errors: ошибки ресурса (timeouts, rejects)
- USE отвечает: "какой ресурс стал узким горлом"

Путь запроса

- Tomcat принимает запрос
- Tomcat выделяет рабочий поток из пула
- Код берёт JDBC connection из HikariCP
- Выполняет SQL
- Делает внешний HTTP вызов и ждёт
- Собирает ответ и возвращает клиенту

А что в Tomcat?

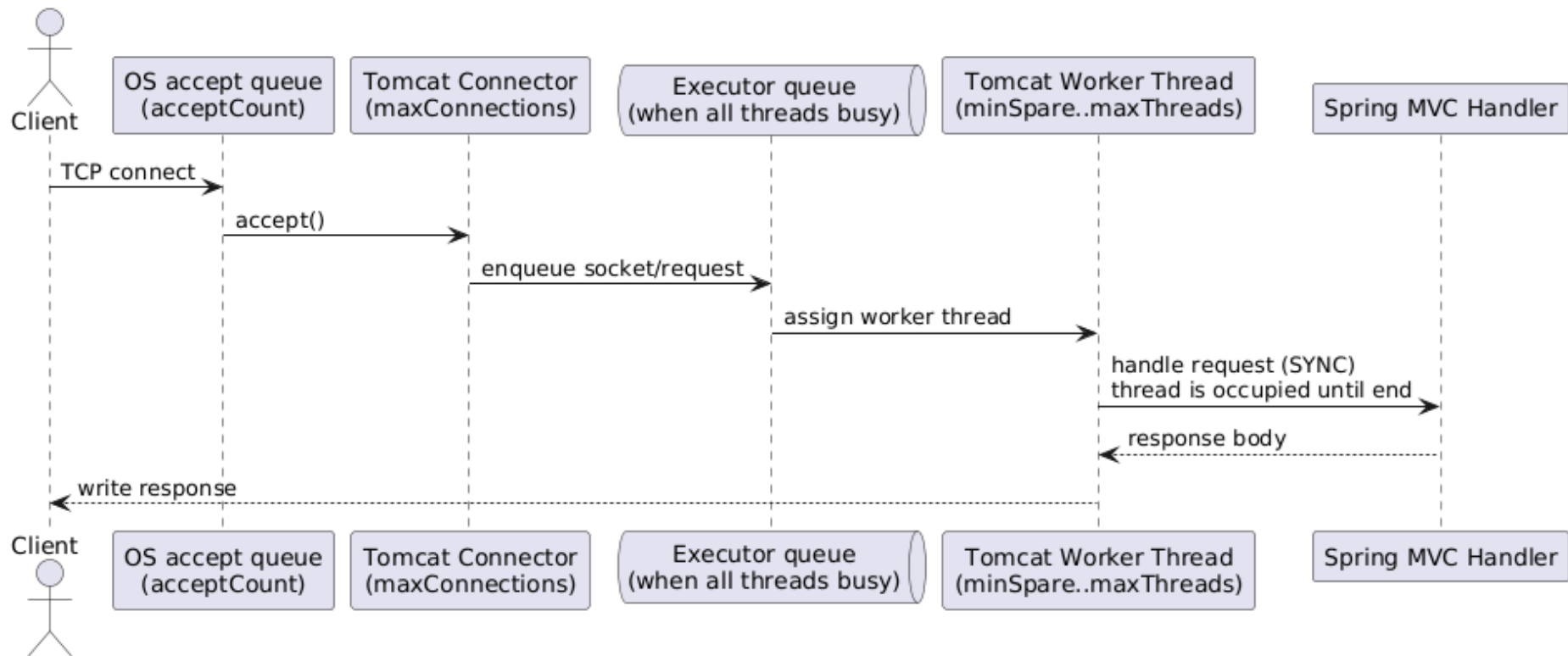
Tomcat thread pool = вход в нашу систему

- В Spring MVC “обычный” запрос занимает поток до конца обработки
- Если поток “ждёт” БД/HTTP - он всё равно занят
- Когда `busy → max`, новые запросы превращаются в очередь

Три “зоны ожидания”

- Очередь на входе (backlog/асцепт queue)
- Ожидание выделения worker thread (когда все заняты)
- Ожидание внутри worker thread (БД/HTTP/локи)

Sequence: Tomcat распределяет запросы по worker threads





HikariCP: ключевая идея

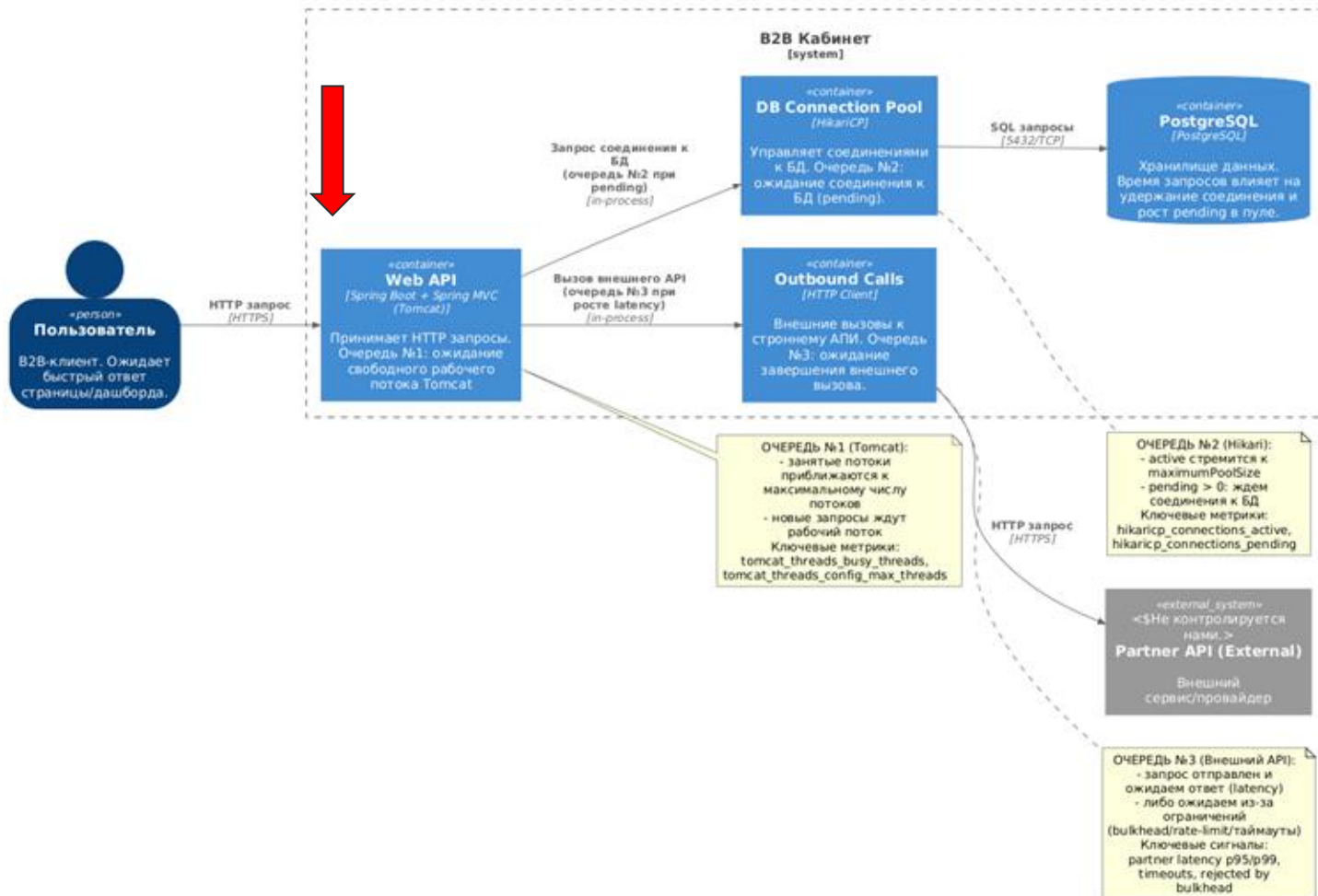
HikariCP pool = жёсткий лимит доступа к БД

- JDBC connection - конечный ресурс
- Когда пул исчерпан, `getConnection()` блокирует поток до timeout
- В MVC это значит: заблокированный поток Tomcat просто “стоит”

Три очереди в системе

- Очередь №1: Tomcat worker threads (busy/max + backlog)
- Очередь №2: Hikari pending (ожидание коннекта)
- Очередь №3: Partner latency + in-flight/timeout rate

Анатомия зависания



[GRAFANA IMAGE] USE: Tomcat
threads (util/sat)

[GRAFANA IMAGE] USE: Hikari pool
(util/sat/errors)

[GRAFANA IMAGE] USE: Partner
latency/errors (+ in-flight, если есть)

[THREAD DUMP IMAGE] Thread dump
"все ждут"

Перцентили: почему p95/p99

- p50: “типичный” запрос
- p95/p99: “хвост”, который убивает UX и бизнес-процессы
- Очереди и блокировки всегда сначала проявляются в хвостах
- Среднее может быть “красивым”, а p99 - на грани провала



RED vs USE

- Что получили: "рентген" системы: RED показывает боль, USE показывает узкое место.
- Что решили: нашли, где сатурация - это очередь ожидания, а не CPU.
- Что осталось: не лечить heavy сразу, а сначала сделать систему "непотопляемой" (admin/actuator должны жить).

Акт III

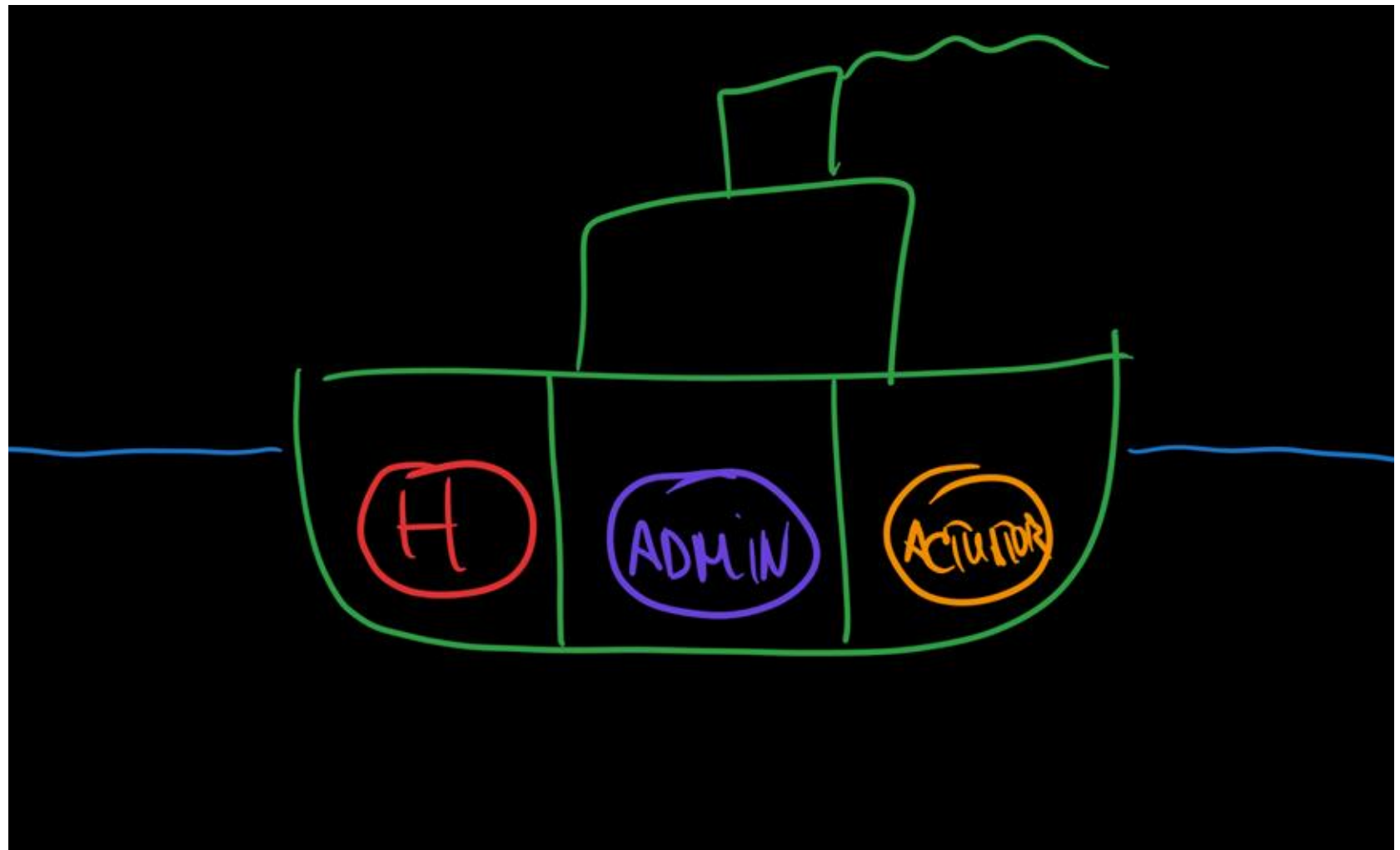
Спасаем корабль: Bulkhead-
изоляция + Virtual Threads
эксперимент

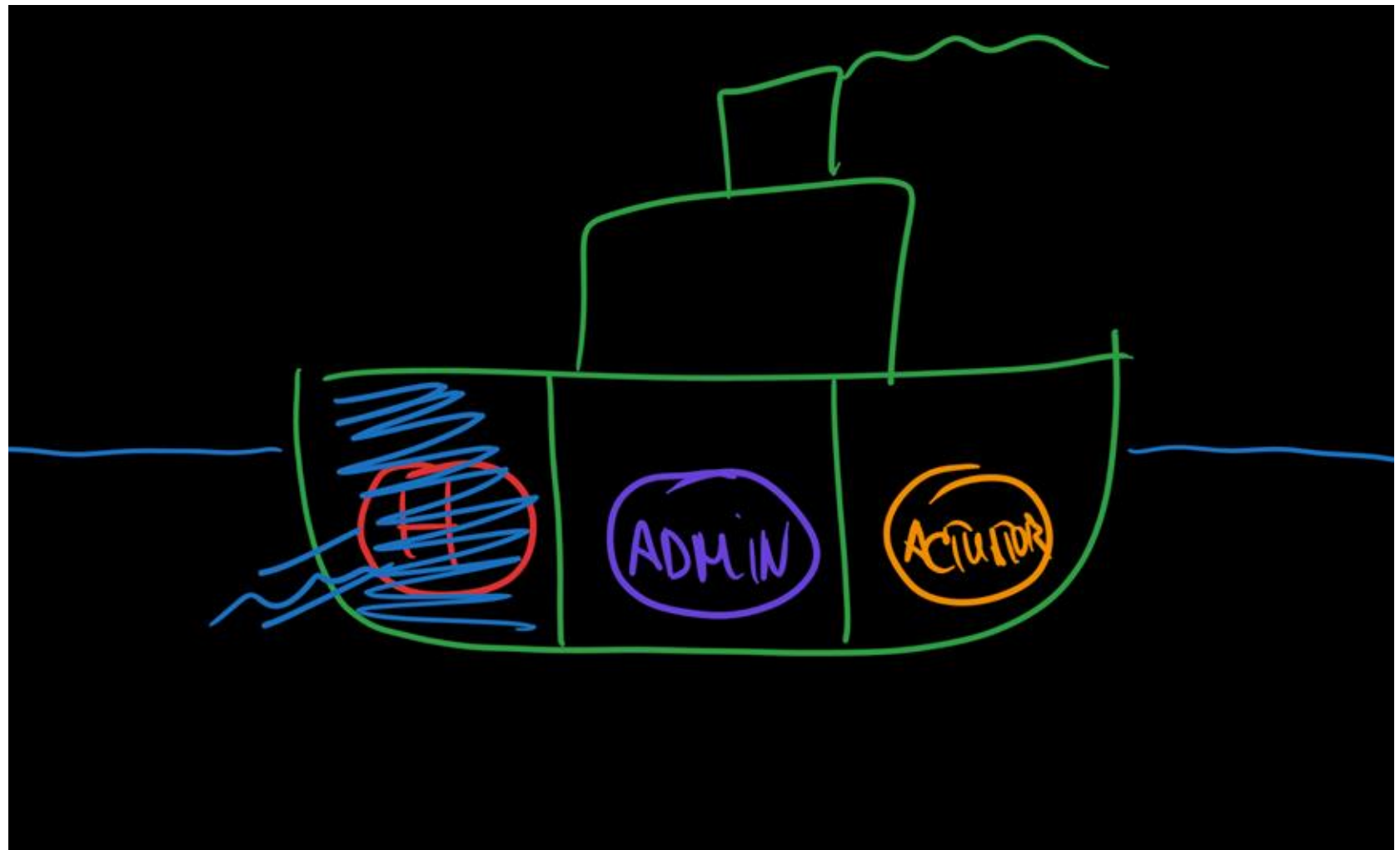
VT: проба

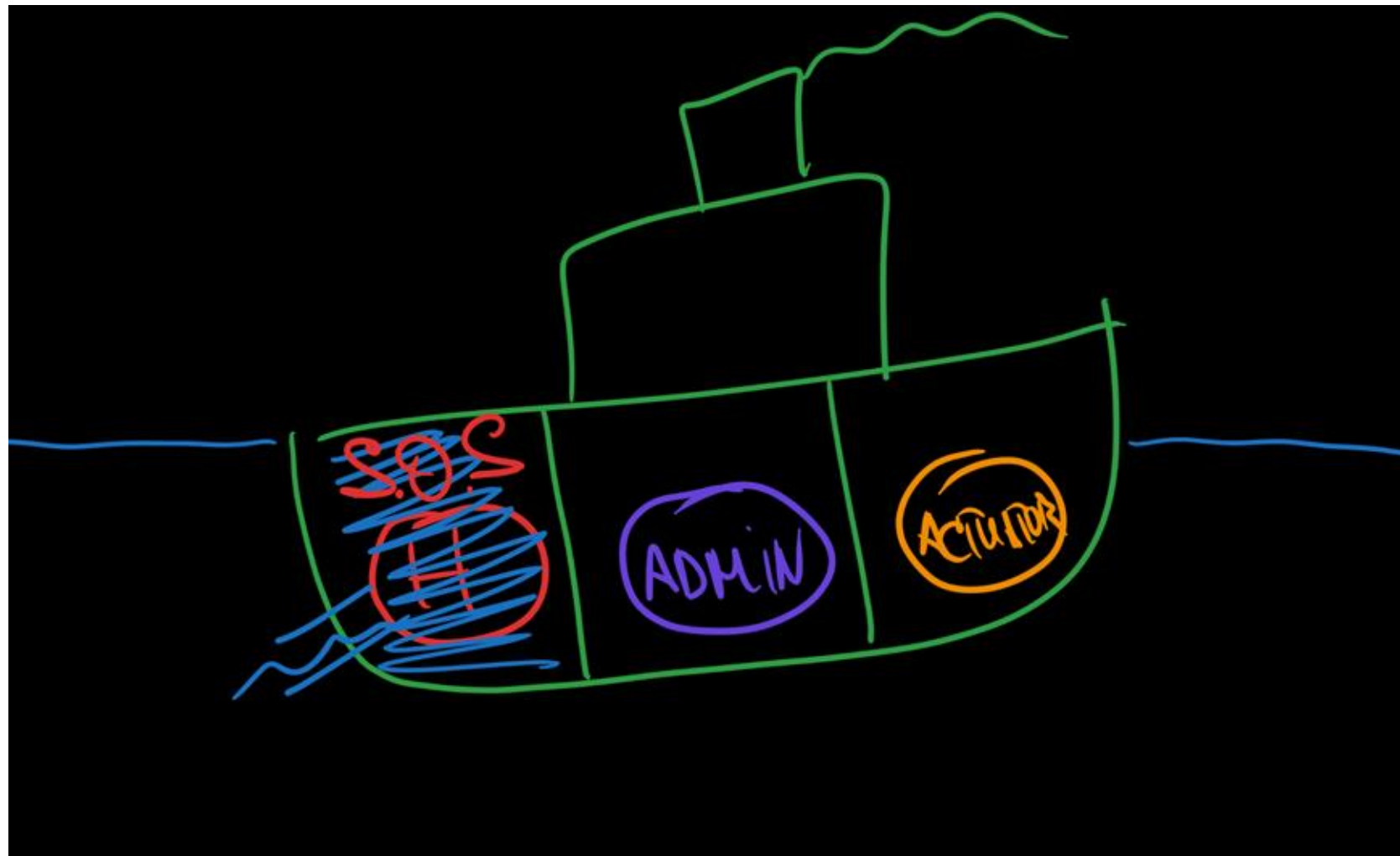
- Попробуем перевести на виртуальные потоки
- Видим, что теперь actuator стал работать быстрее и проблема рестарта ушла
- Проблема с админским эндпоинтов сохраняется.

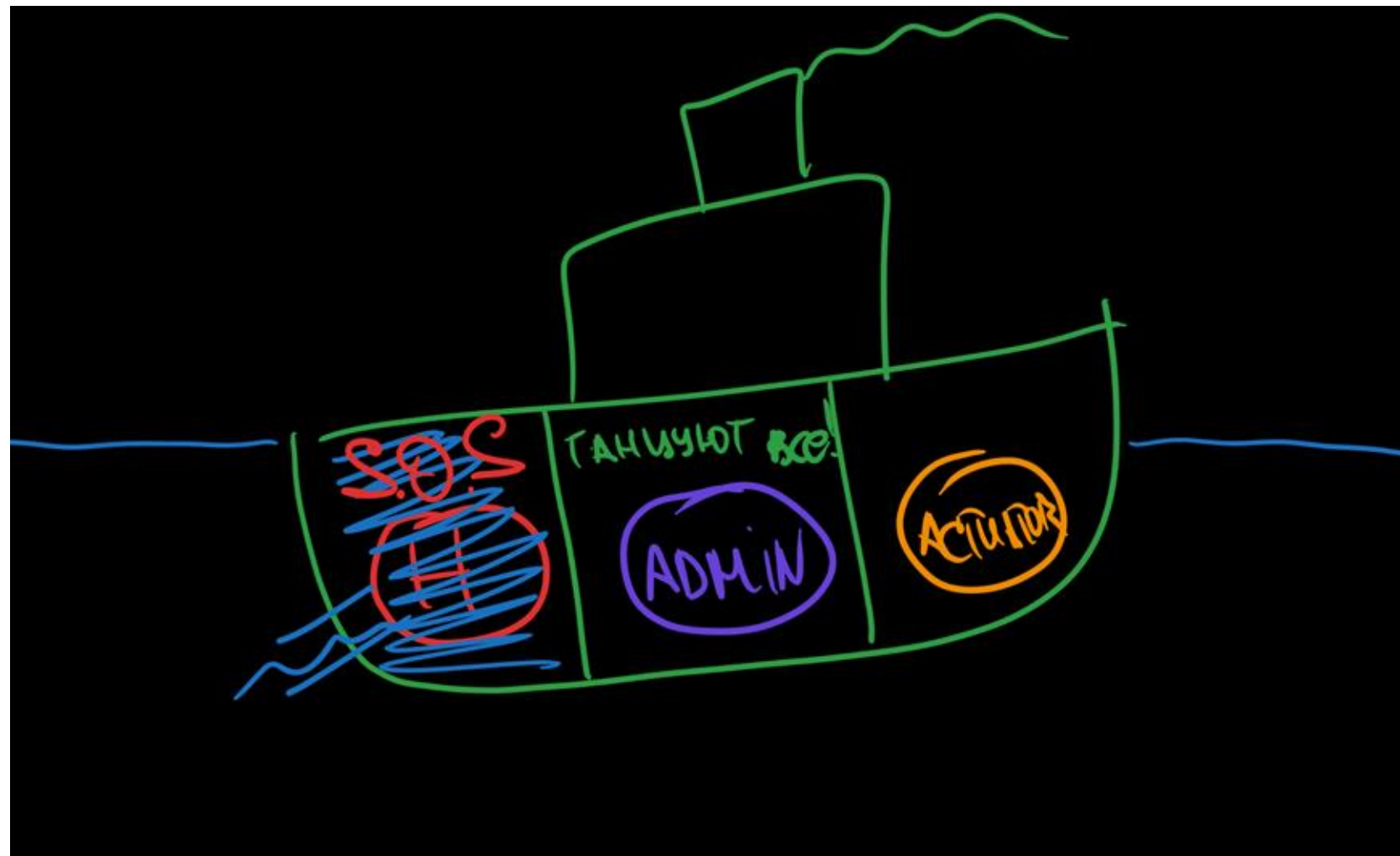
Корабль: JVM, общие ресурсы

- Общие Tomcat threads
- Общий Hikari pool
- Общие ожидания внешнего HTTP
- И именно поэтому heavy способен вытеснить admin/actuator

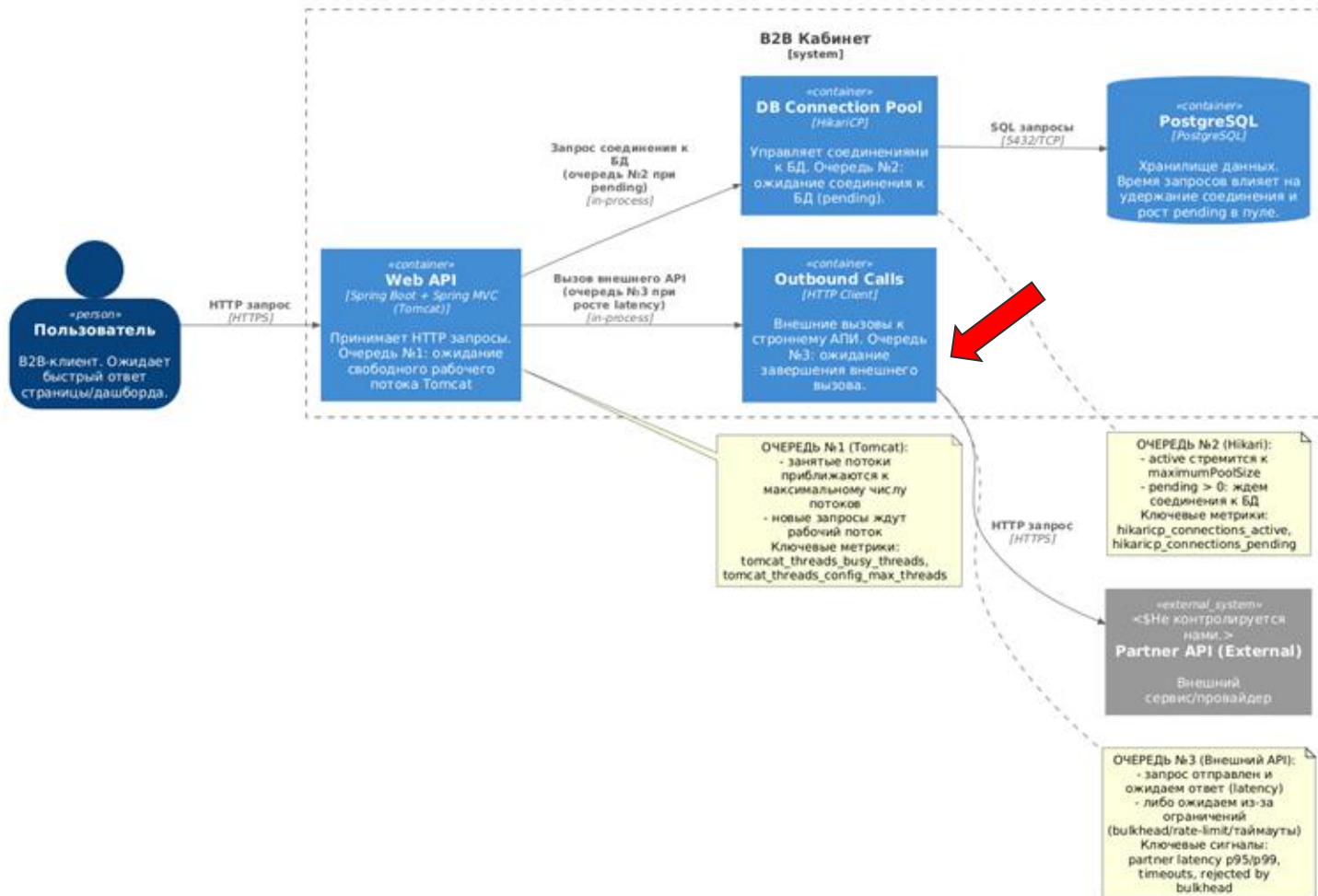








Анатомия зависания



Bulkhead

(ограничиваем in-flight)

Bulkhead: что решает

- Изолировать конкретную интеграцию и ресурсы под нее
- Ограничивает количество одновременных вызовов к партнёру
- Не даёт “размножать ожидание” и забивать систему потоками/коннектами
- Превращает лавину таймаутов в управляемый отказ

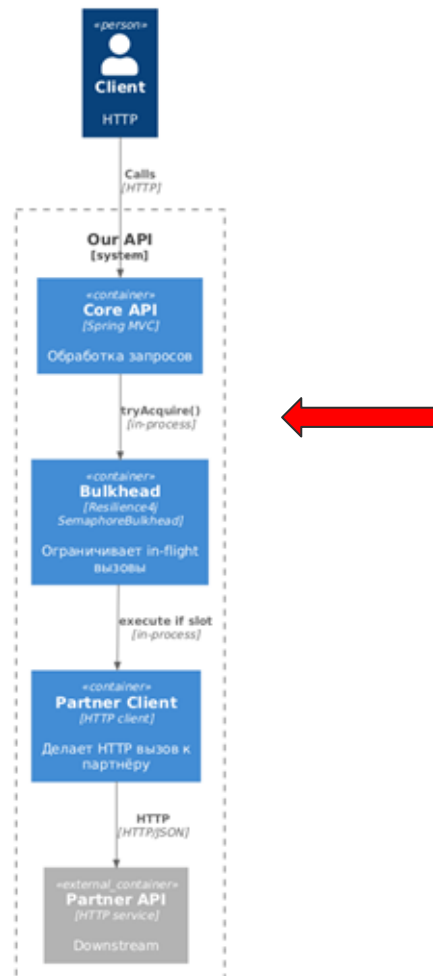
Bulkhead: 2 реализации

- Semaphore bulkhead: ограничиваем конкурентность без отдельного пула
- Thread-pool bulkhead: изоляция на отдельные потоки + bounded queue
- Главное: ограничить in-flight и управлять ожиданием слота.

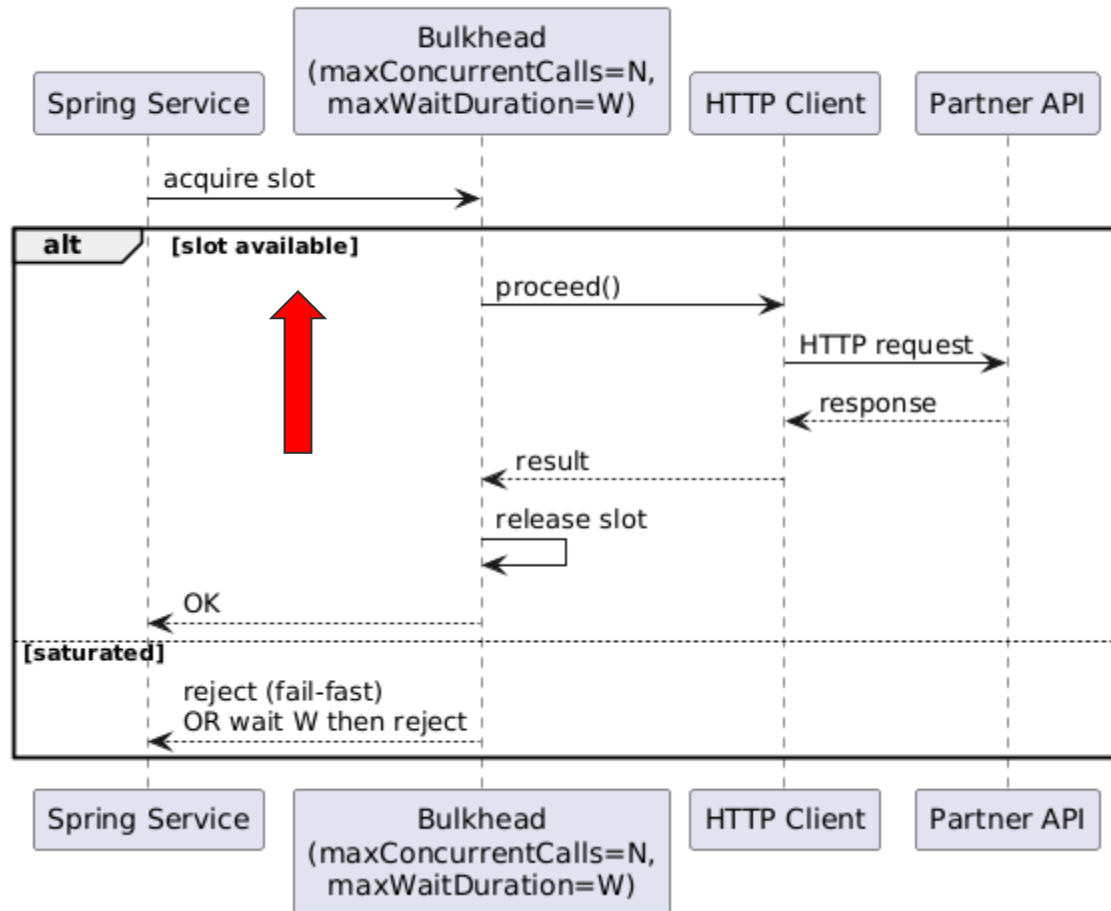
Bulkhead: что настраиваем

- ``maxConcurrentCalls``: сколько параллельных вызовов к партнёру выдержим
- ``maxWaitDuration``: готовы ли ждать слот или падаем сразу
- Метрики: in-flight, rejected calls, wait time на слот

C4 Container: Bulkhead для Partner API



Sequence: Bulkhead ограничивает in-flight к партнёру



Bulkhead

- Что получили: правильную цель: "сначала выживаемость, потом производительность".
- Что решили: объяснили, почему admin/actuator падают и как концептуально это лечится.
- Что осталось: показать внедрение bulkhead и эффект на метриках (и не убить мониторинг).

[CODE IMAGE] Конфиг Bulkhead для heavy

[CODE IMAGE] Конфиг Bulkhead для heavy

[CODE IMAGE] Конфиг Bulkhead для heavy

[GRAFANA IMAGE] До/после:
Actuator живет

[GRAFANA IMAGE] До/после: Admin
стал предсказуем

Проблема актуатора

- Если readiness/metrics постоянно ходят в БД → создают доп. конкуренцию
- Hikari один, запросов стало больше
- Итог: observability превращается в нагрузку на дефицитный ресурс

[CODE IMAGE] Правильный health:
liveness без БД, readiness аккуратно

Bulkhead

- Что получили: система не "тонет": actuator/admin выживают.
- Что решили: убрали каскад "health не отвечает → рестарты → хуже".
- Что осталось: heavy все еще плох - теперь лечим его как причину.

VT: обещание и реальность

- VT делают ожидание потоков дешевле
- Но не создают больше JDBC connections
- Не ускоряют партнера
- Вывод: VT - усилитель, а не лекарство

[CODE IMAGE] Переключатель Virtual
Threads

[GRAFANA IMAGE] VT помогли
actuator

[GRAFANA IMAGE] VT HE помогли
admin (упор в БД/Hikari)

VT: обещание и реальность

- Что получили: честную картину по VT.
- Что решили: улучшили "сервисность" части путей (actuator).
- Что осталось: главный тяжелый запрос все равно упирается в DB/downstream → лечим фундаментально.

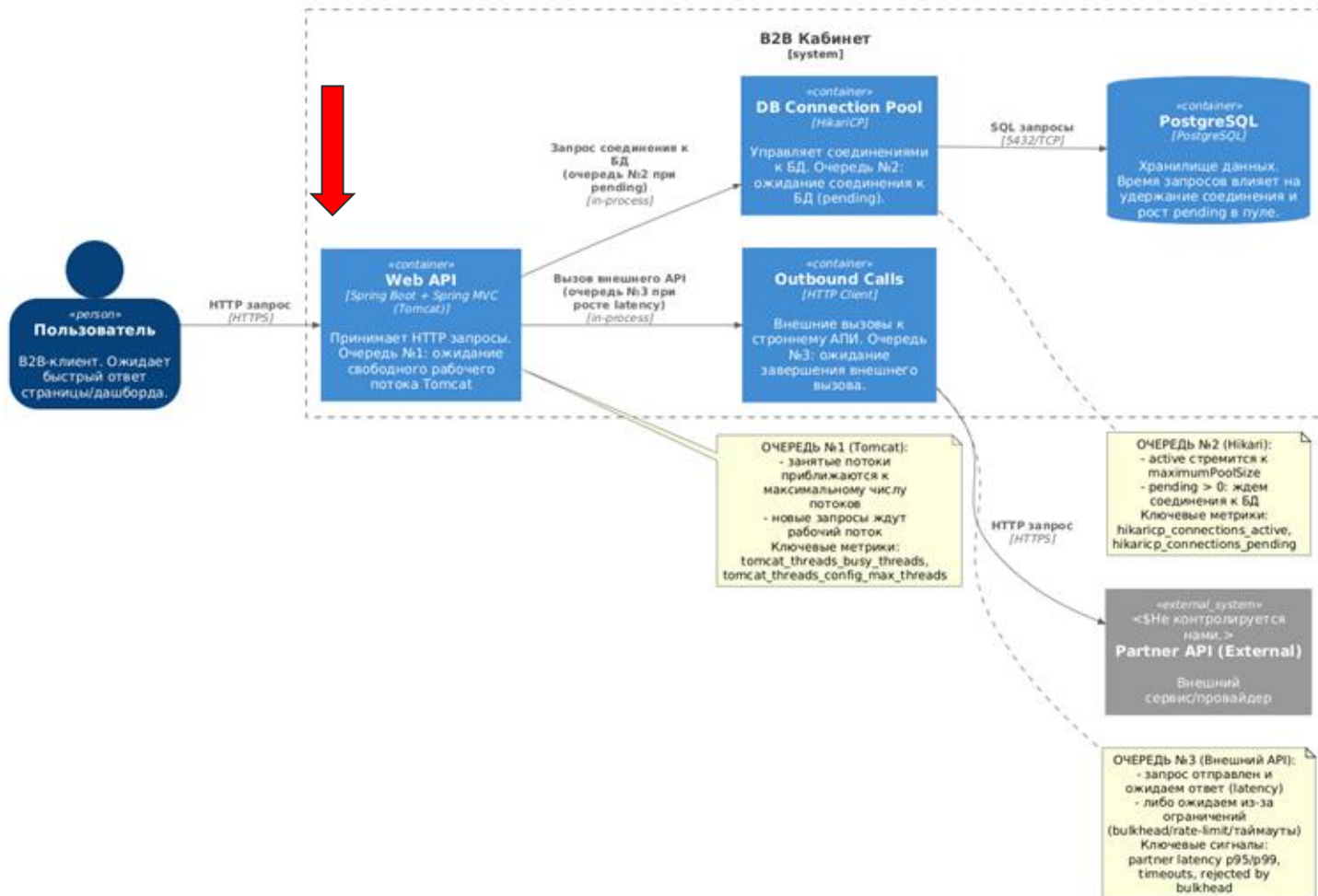
АСТ IV:

"Лечим heavy: очереди,
транзакции, circuit breaker,
rate limiter"

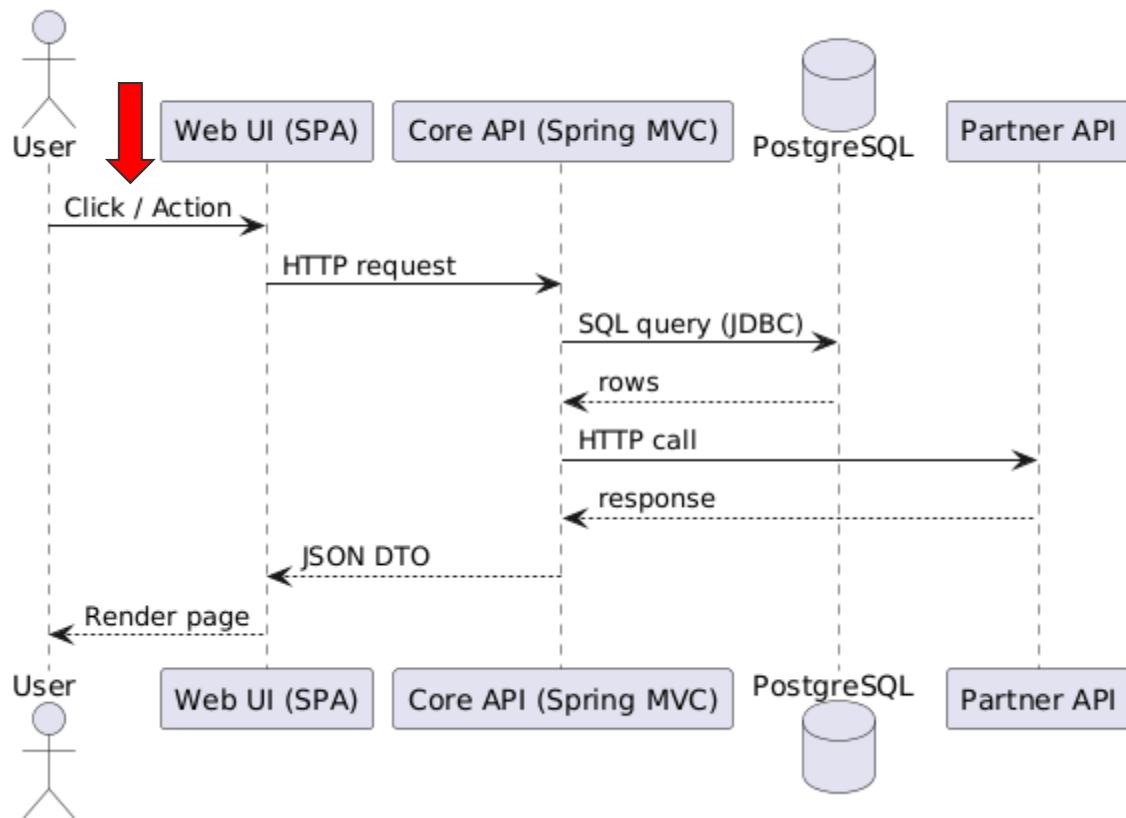
Путь запроса

- Tomcat принимает запрос
- Tomcat выделяет рабочий поток из пула
- Код берёт JDBC connection из HikariCP
- Выполняет SQL
- Делает внешний HTTP вызов и ждёт
- Собирает ответ и возвращает клиенту

Анатомия зависания



Sequence: стандартный запрос (MVC + DB + Partner)



[GRAFANA IMAGE] USE: Tomcat
busy/max во время heavy деградации

[GRAFANA IMAGE] USE: Hikari
pending pactet

[GRAFANA IMAGE] USE/RED: Partner
latency/timeouts

Как читать эту корреляцию

- Если Tomcat busy растёт и partner latency растёт → узкое место downstream

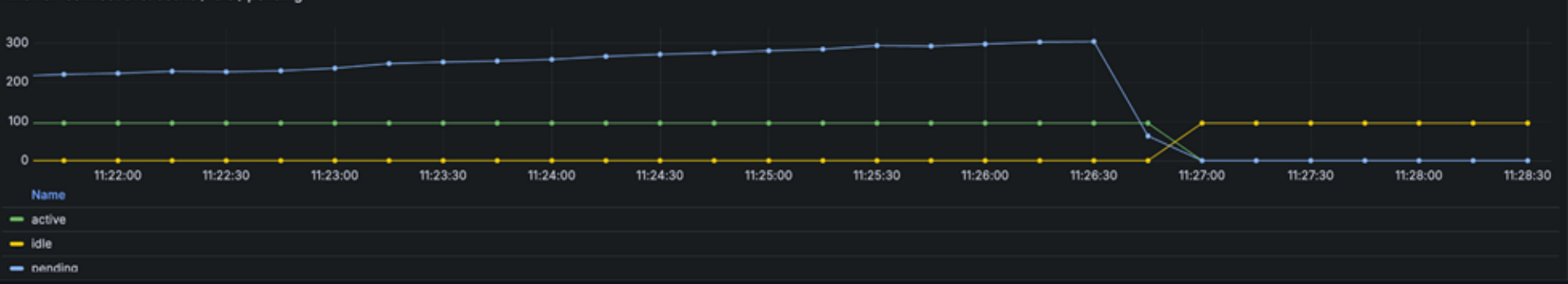
Tomcat threads: busy vs max



Tomcat threads: current



HikariCP connections: active / idle / pending



Partner API latency p95 (demo.partner.call)



Partner API latency p99 (demo.partner.call)



Как читать эту корреляцию

- Если Tomcat busy растёт и partner latency растёт → узкое место downstream
- Если Hikari pending растёт без роста partner latency → вероятны длинные запросы/локи/утечки connection

Tomcat threads: busy vs max



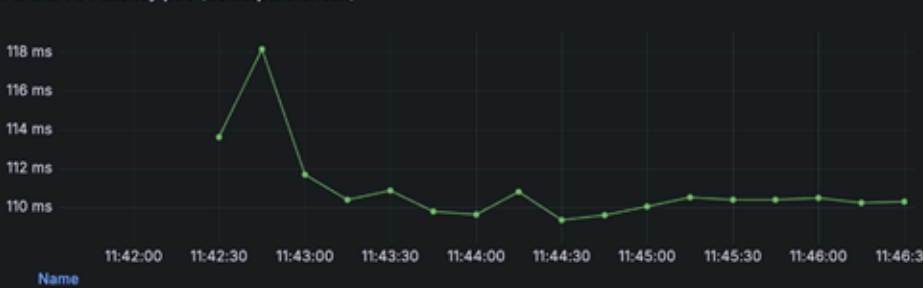
Tomcat threads: current



HikariCP connections: active / idle / pending



Partner API latency p95 (demo.partner.call)



Partner API latency p99 (demo.partner.call)



Как читать эту корреляцию

- Если Tomcat busy растёт и partner latency растёт → узкое место downstream
- Если Hikari pending растёт без роста partner latency → вероятны длинные запросы/локи/утечки connection
- CPU при этом может быть низким во всех случаях



Thread dump

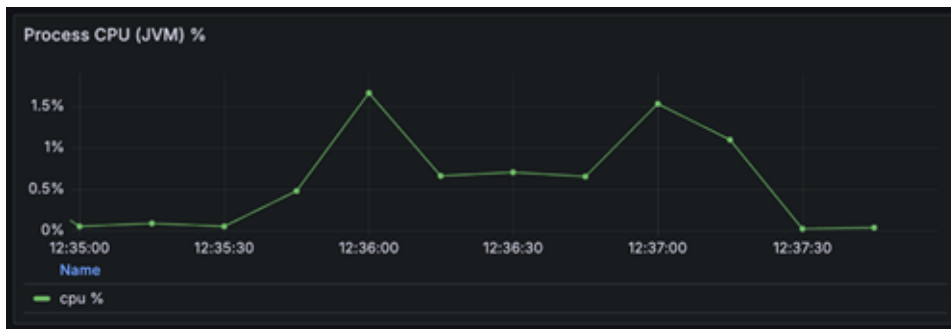
3 типовых класса “ожиданий”:

1. WAITING/TIMED_WAITING на `getConnection()` (Hikari)
2. TIMED_WAITING на socket read / HTTP client (ждём партнёра)
3. BLOCKED на lock (локи в коде или на уровне JDBC/DB)

```
"threadName": "http-nio-8080-exec-67",
"threadId": 54,
"blockedTime": 32,
"blockedCount": 286,
"waitedTime": 8933,
"waitedCount": 7,
"lockName": "java.util.concurrent.CompletableFuture$Signaller@52d37f62",
"lockOwnerId": -1,
"daemon": true,
"inNative": false,
"suspended": false,
"threadState": "WAITING",
"priority": 5,
"stackTrace": [
  {
    "moduleName": "java.base",
    "moduleVersion": "21.0.10",
    "methodName": "park",
    "lineNumber": -2,
    "className": "jdk.internal.misc.Unsafe",
    "nativeMethod": true
  }
]
```

```
"threadName": "http-nio-8080-exec-67",
"threadId": 134,
"blockedTime": 0,
"blockedCount": 0,
"waitedTime": 39245,
"waitedCount": 11,
"lockName": "java.util.concurrent.CompletableFuture$Signaller@2678c2bb",
"lockOwnerId": -1,
"daemon": true,
"inNative": false,
"suspended": false,
"threadState": "WAITING",
"priority": 5,
"stackTrace": [
  {
    "moduleName": "java.base",
    "moduleVersion": "21.0.10",
    "methodName": "park",
    "lineNumber": -2,
    "className": "jdk.internal.misc.Unsafe",
    "nativeMethod": true
  }
]
```

```
"threadName": "http-nio-8080-exec-67",
"threadId": 134,
"blockedTime": 0,
"blockedCount": 1,
"waitedTime": 61700,
"waitedCount": 21,
"lockName": "java.util.concurrent.CompletableFuture$Signaller@1ff839bd",
"lockOwnerId": -1,
"daemon": true,
"inNative": false,
"suspended": false,
"threadState": "WAITING",
"priority": 5,
"stackTrace": [
  {
    "moduleName": "java.base",
    "moduleVersion": "21.0.10",
    "methodName": "park",
    "lineNumber": -2,
    "className": "jdk.internal.misc.Unsafe",
    "nativeMethod": true
  }
]
```



Порядок лечения

1. Time budget + timeouts + fail-fast
2. Circuit breaker
3. Транзакции (короткие) + DB bulkhead
4. Rate limiter на входе
5. VT - только как усилитель, если нужно

Time budget + fail-fast

- Если у запроса нет бюджета - он превращается в "вечное ожидание"
- Таймауты = управление очередью
- Быстрый отказ лучше массового зависания

[CODE IMAGE] Таймауты HTTP
клиента + общий deadline

Circuit Breaker

-

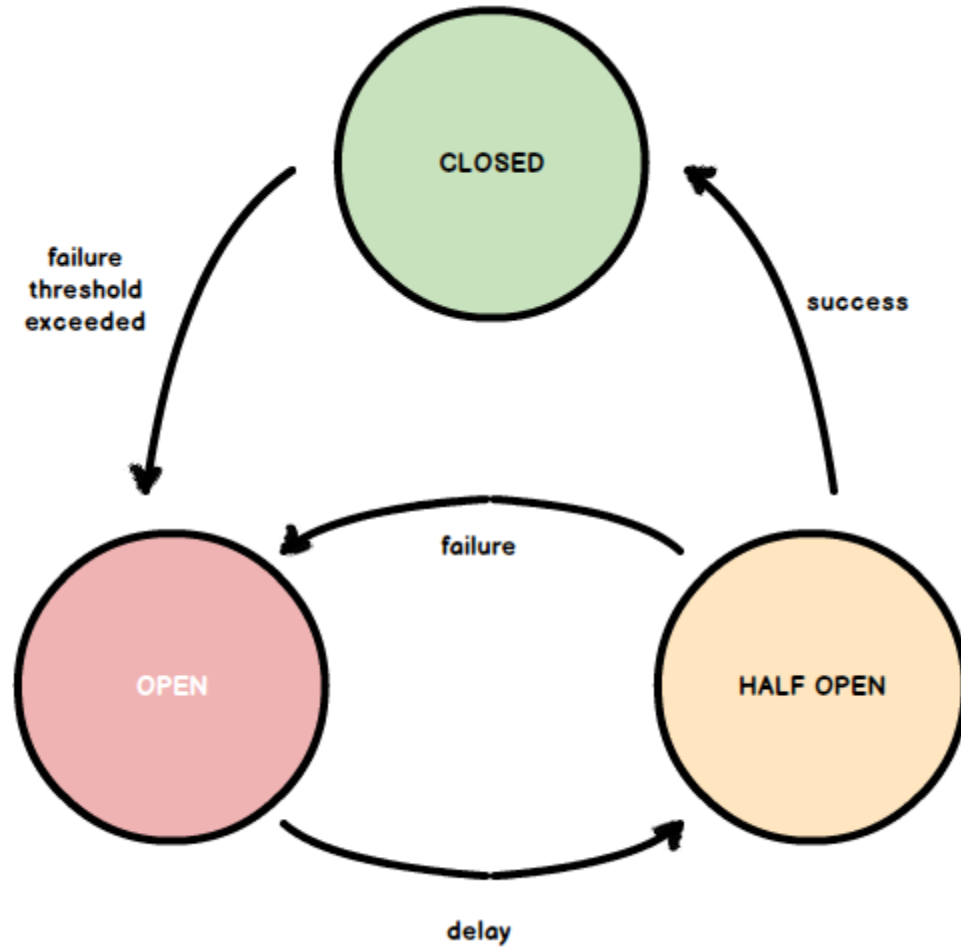
не добиваем
деградирующий
downstream

Circuit Breaker: что решает

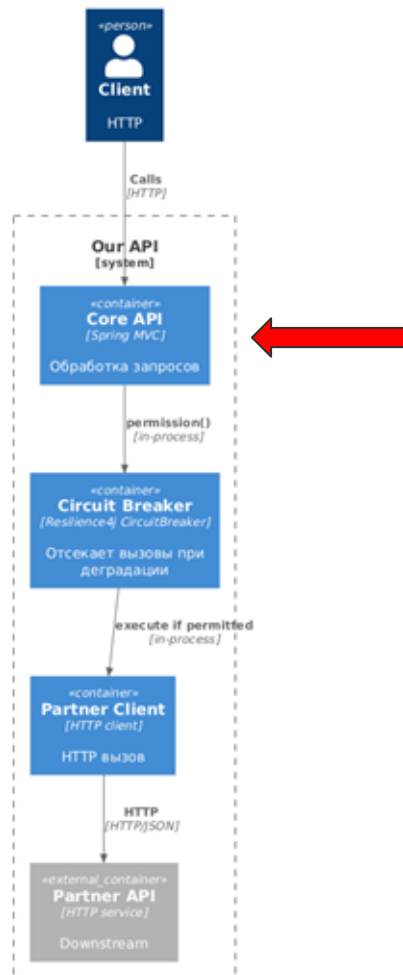
- Когда downstream деградирует, мы перестаём слать туда запросы “на авось”
- Убираем бессмысленное ожидание, освобождаем ресурсы
- Сервис остаётся живым и предсказуемо деградирует

Circuit Breaker: логика

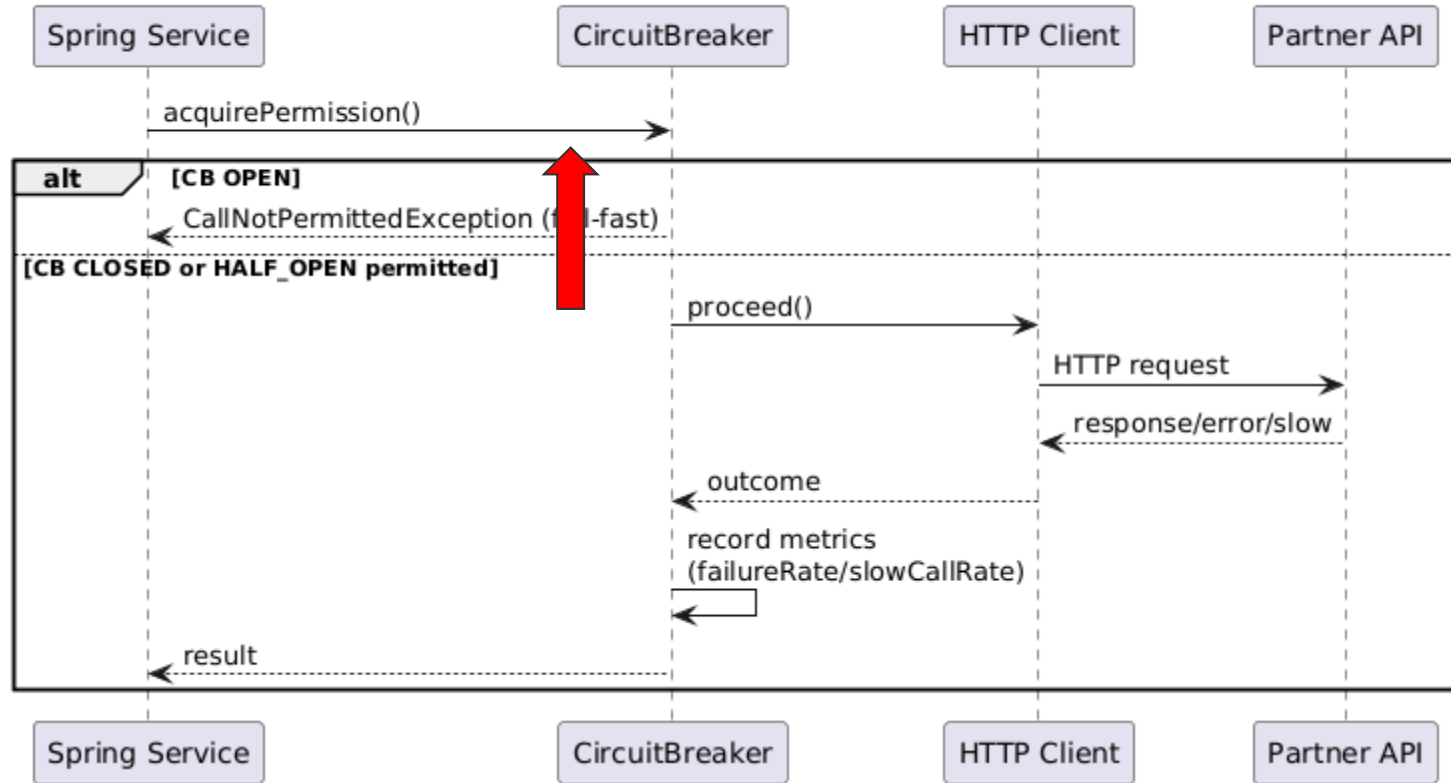
- CLOSED: всё нормально, считаем статистику
- OPEN: слишком много ошибок/медленных вызовов → режем вызовы быстро
- HALF_OPEN: пробуем ограниченное число запросов, проверяем восстановление



C4 Container: Circuit Breaker перед Partner API



Sequence: Circuit Breaker быстрый отказ при OPEN



настраиваем

- окно статистики (сколько запросов учитываем)
- порог ошибок
- порог медленных вызовов
- время в OPEN
- сколько тестовых вызовов в HALF_OPEN

ГРАФИКИ или АНИМАЦИЯ для СВ +
ВН

Bulkhead vs CB

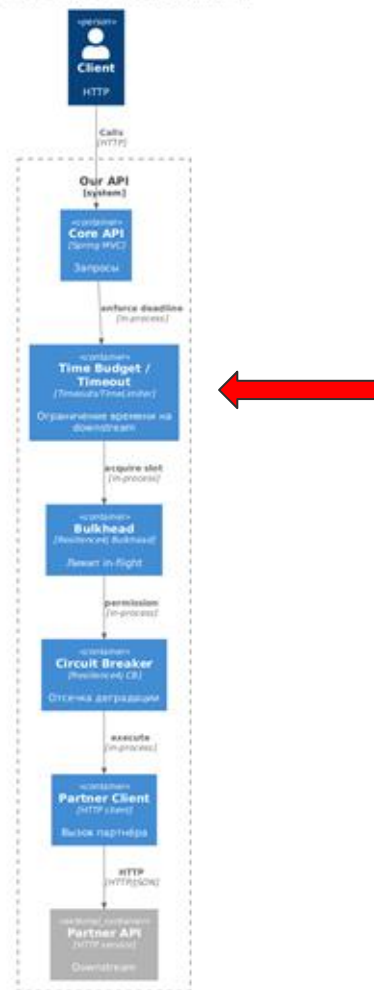
- Bulkhead: ограничивает параллельность (in-flight)

Bulkhead vs CB

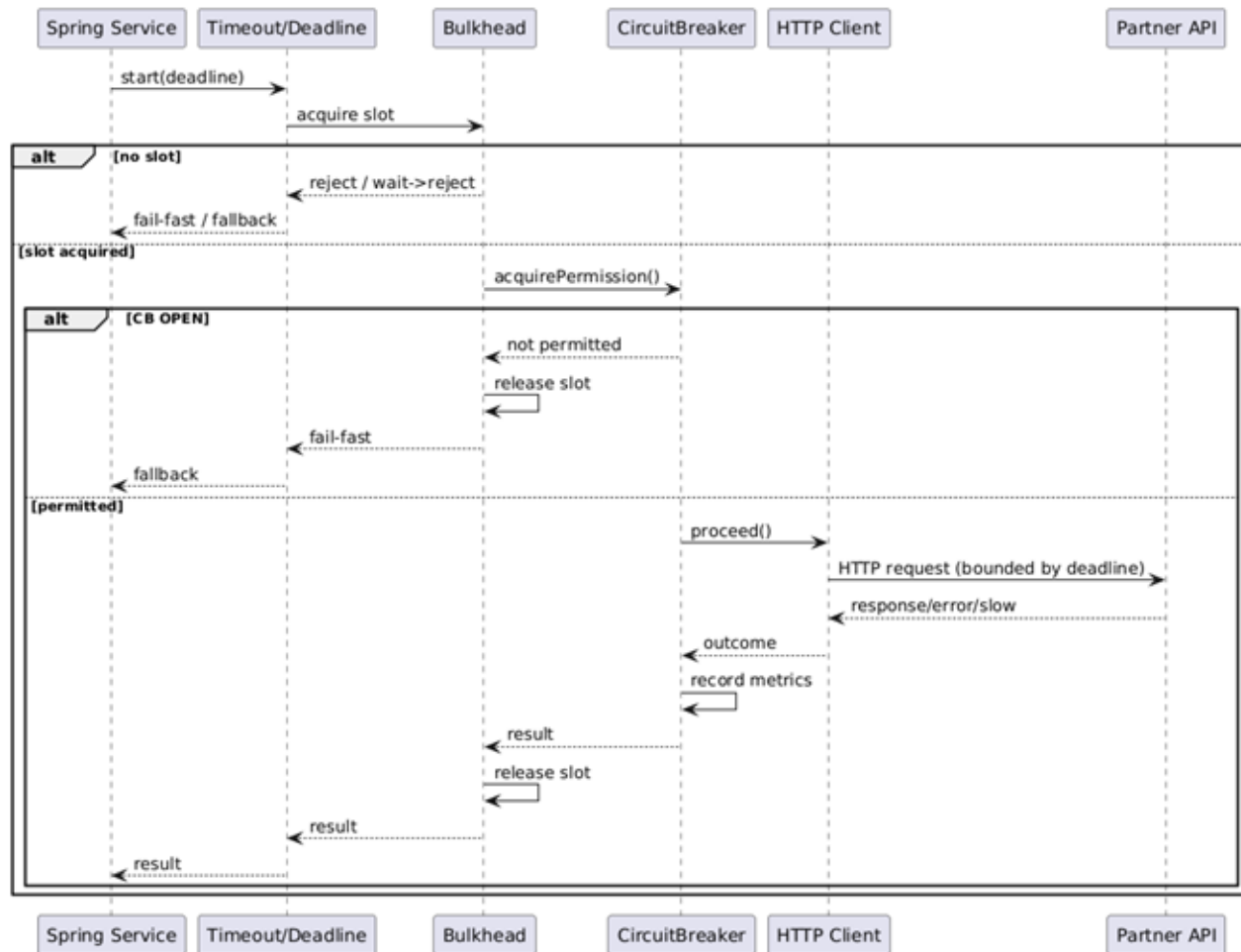
- Bulkhead: ограничивает параллельность (in-flight)
- Circuit Breaker: отсекает деградацию (не тратим ресурсы на “плохой” downstream)

Bulkhead vs CB

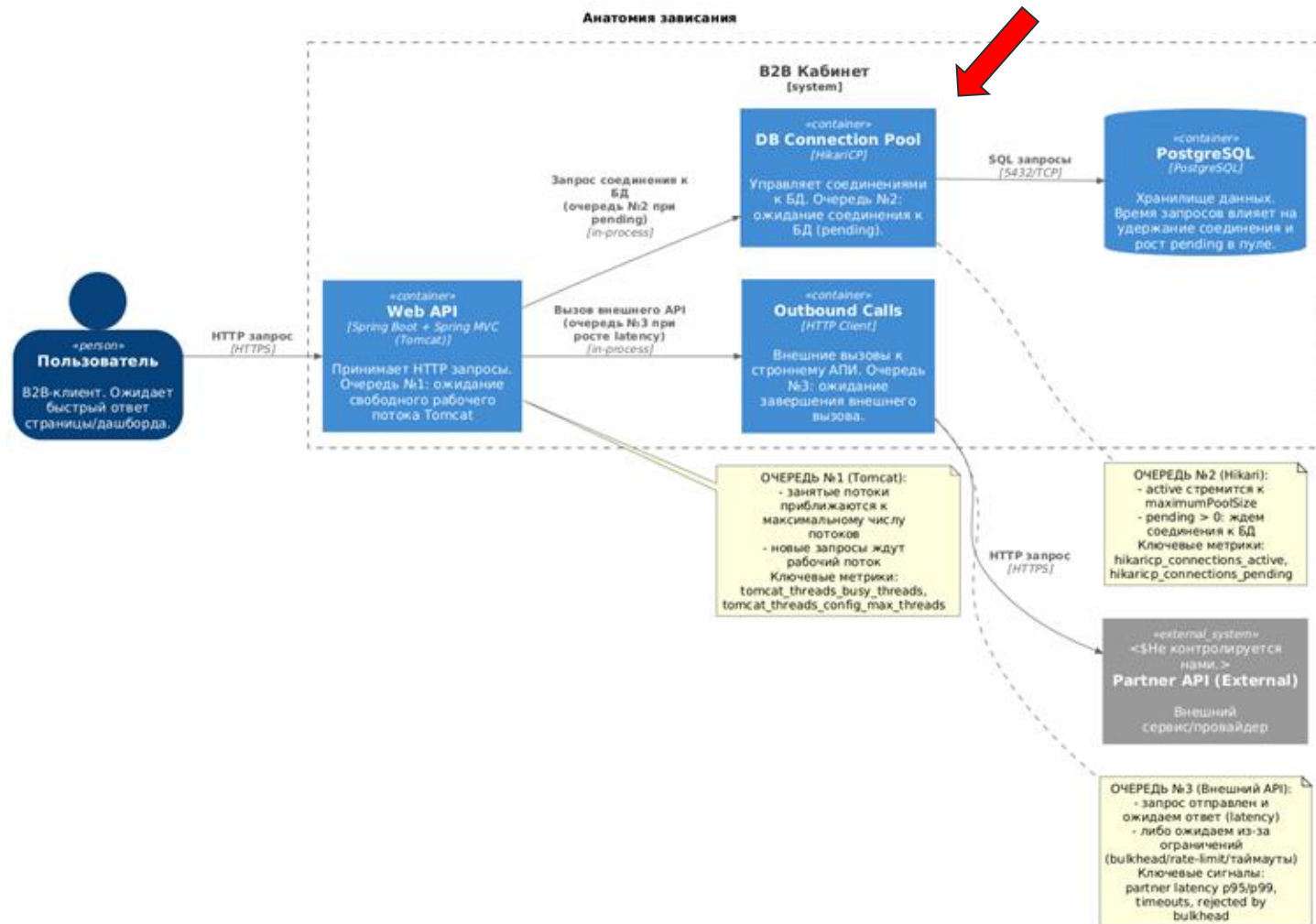
- Bulkhead: ограничивает параллельность (in-flight)
- Circuit Breaker: отсекает деградацию (не тратим ресурсы на “плохой” downstream)
- Сильный эффект получается только в комбинации.



Sequence: deadline + bulkhead + circuit breaker (правильный конвейер)



Длинная транзакция и защита БД



```

1  ProfileRepository.java
2
3  > import ...
4
5
6  @Repository & Evgeniy Suleimanov
7  public class ProfileRepository {
8
9      private final JdbcTemplate jdbcTemplate; 3 usages
10
11      public ProfileRepository(JdbcTemplate jdbcTemplate) { & Evgeniy Suleimanov
12          this.jdbcTemplate = jdbcTemplate;
13      }
14
15      public Profile findById(int id) { 1 usage & Evgeniy Suleimanov
16          return jdbcTemplate.queryForObject(
17              sq: "select id, name, segment, previous_score from profiles where id = ?",
18              (ResultSet rs, int rowNum) -> new Profile(
19                  rs.getInt( columnLabel: "id"),
20                  rs.getString( columnLabel: "name"),
21                  rs.getString( columnLabel: "segment"),
22                  rs.getInt( columnLabel: "previous_score")
23              ),
24              id
25          );
26      }
27
28      /**
29       * Updates previous_score for a profile. Returns number of updated rows (0 or 1).
30       */
31      public int updatePreviousScore(int id, int previousScore) { 1 usage & Evgeniy Suleimanov
32          return jdbcTemplate.update(
33              sq: "update profiles set previous_score = ? where id = ?",
34              previousScore, id
35          );
36      }
37  }

```

```

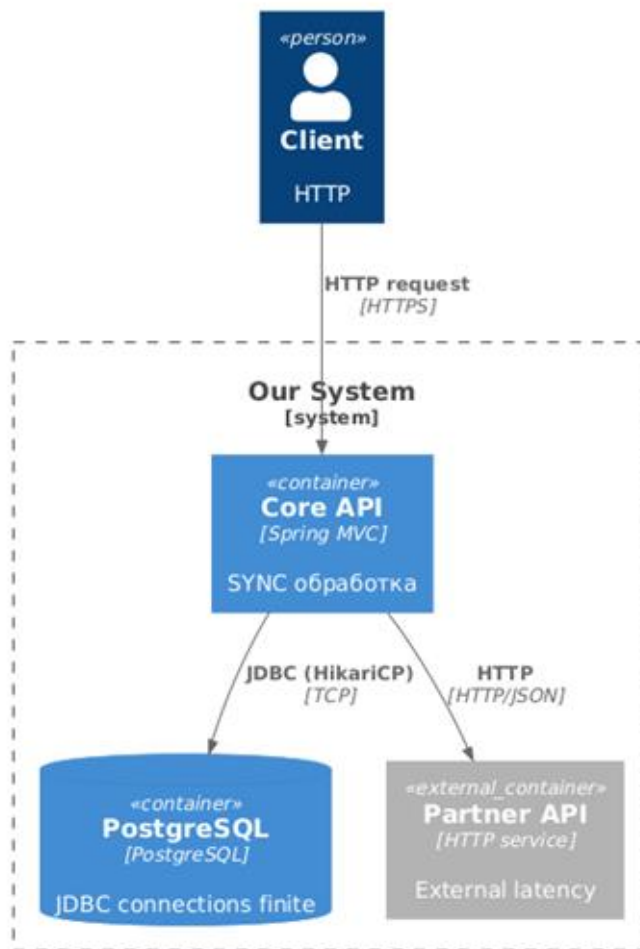
3  > import ...
10
11  @Service 3 usages & Evgeniy Suleimanov
12  public class DashboardService {
13
14      private final ProfileRepository profileRepository; 3 usages
15      private final PartnerClient partnerClient; 2 usages
16
17      public DashboardService(ProfileRepository profileRepository, PartnerClient partnerClient) { & Evgeniy Suleimanov
18          this.profileRepository = profileRepository;
19          this.partnerClient = partnerClient;
20      }
21
22      @Transactional 1 usage & Evgeniy Suleimanov
23      public DashboardResponse getDashboard(int id) {
24          Profile profile = profileRepository.findById(id);
25          PartnerResponse partner = partnerClient.getPartnerInfo(id);
26
27          profileRepository.updatePreviousScore(id, partner.score());
28
29          return new DashboardResponse(
30              profile.id(),
31              profile.name(),
32              profile.segment(),
33              profile.previousScore(),
34              new DashboardResponse.PartnerInfo(partner.id(), partner.status(), partner.score())
35          );
36      }
37  }
38

```

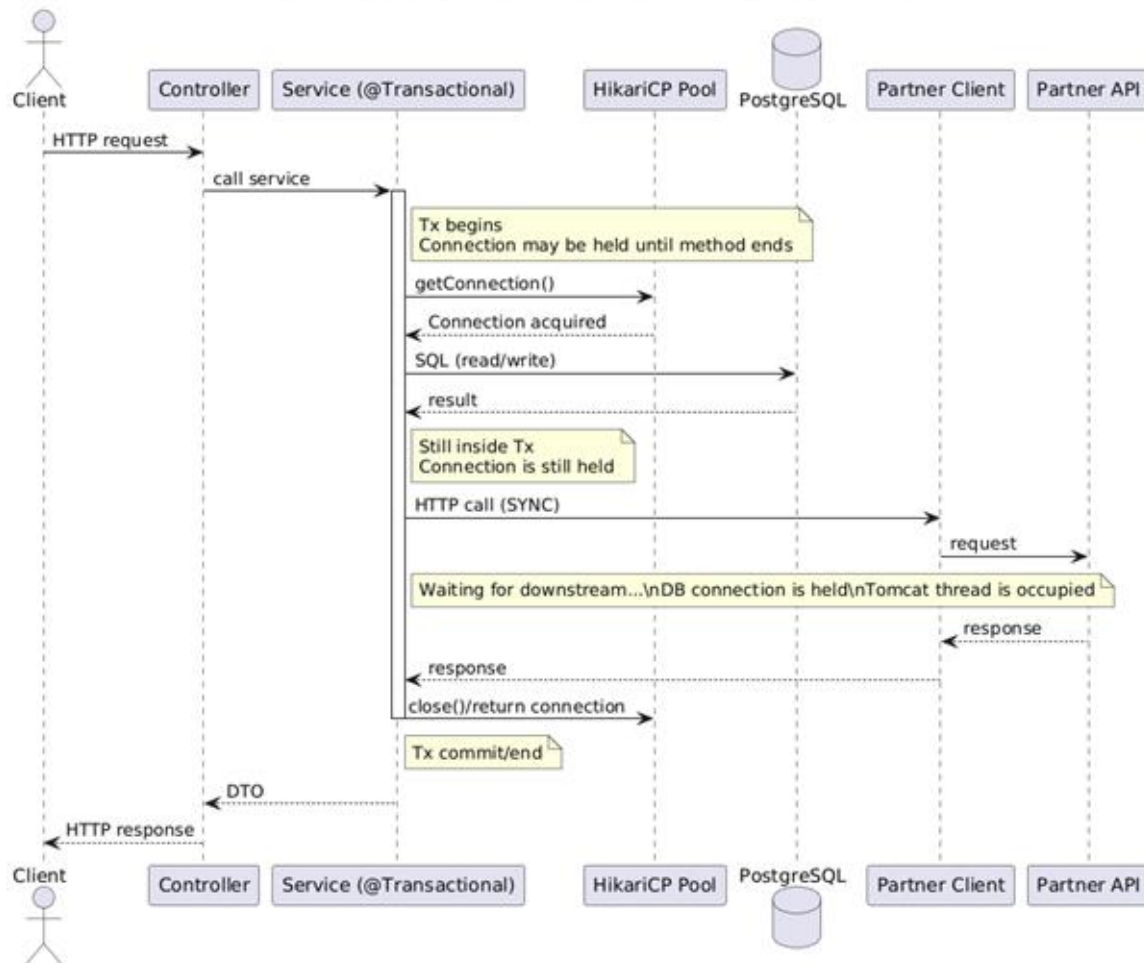
Длинная транзакция

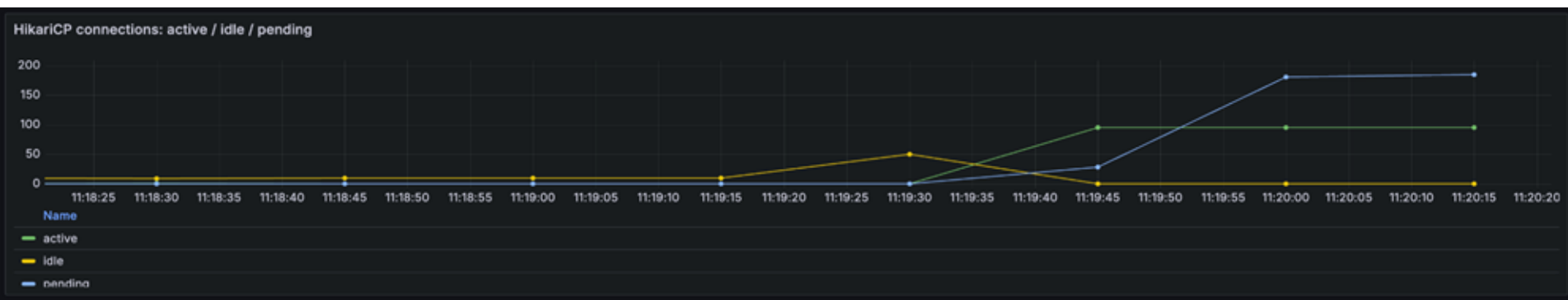
- Транзакция часто означает: connection держится до конца метода
- Если внутри есть внешний HTTP вызов → держим connection, пока ждём сеть
- Итог: Hikari pending растёт, Tomcat threads заняты ожиданием, хвост latency улетает
- Не для всех бизнес-сценариев это валидно

C4 Container: Tx + DB connection + External wait



Sequence: анти-паттерн — внешний HTTP внутри транзакции





Как лечим транзакции

- Сужаем транзакцию: только то, что действительно должно быть атомарным
- Убираем сетевые ожидания из транзакции
- Разделяем “подготовку данных” и “commit”
- Принцип: transaction = короткая и предсказуемая



application.yml

ProfileRepository.java

ProfileService.java ×



...

```
1 package ru.jug.demo.hangdemo.service;
2
3 > import ...
4
5
6
7 @Service 2 usages
8 public class ProfileService {
9
10     private final ProfileRepository profileRepository; 2 usages
11
12     public ProfileService(ProfileRepository profileRepository) {
13         this.profileRepository = profileRepository;
14     }
15
16     @Transactional 1 usage
17     public void updatePreviousScore(int id, int score) {
18         profileRepository.updatePreviousScore(id, score);
19     }
20 }
21
```

```
app > src > main > java > ru > jug > demo > hangdemo > service > DashboardService > getDashboard
application.yml ProfileRepository.java ProfileService.java DashboardService.java
2
3 > import ...
10
11 @Service 3 usages
12 public class DashboardService {
13
14     private final ProfileRepository profileRepository; 2 usages
15     private final PartnerClient partnerClient; 2 usages
16     private final ProfileService profileService; 2 usages
17
18
19     public DashboardService(ProfileRepository profileRepository, PartnerClient partnerClient, ProfileService profileService) {
20         this.profileRepository = profileRepository;
21         this.partnerClient = partnerClient;
22         this.profileService = profileService;
23     }
24
25     public DashboardResponse getDashboard(int id) { 1 usage
26         Profile profile = profileRepository.findById(id);
27         PartnerResponse partner = partnerClient.getPartnerInfo(id);
28
29         profileService.updatePreviousScore(id, partner.score());
30
31         return new DashboardResponse(
32             profile.id(),
33             profile.name(),
34             profile.segment(),
35             profile.previousScore(),
36             new DashboardResponse.PartnerInfo(partner.id(), partner.status(), partner.score())
37         );
38     }
39 }
40
```

Bulkhead на БД

- Hikari ограничивает connections, но не ограничивает количество потоков, которые ждут connection
- Bulkhead на уровне repository/service ограничивает конкурентность “тяжёлых операций”
- Результат: меньше pending, меньше забитых потоков Tomcat

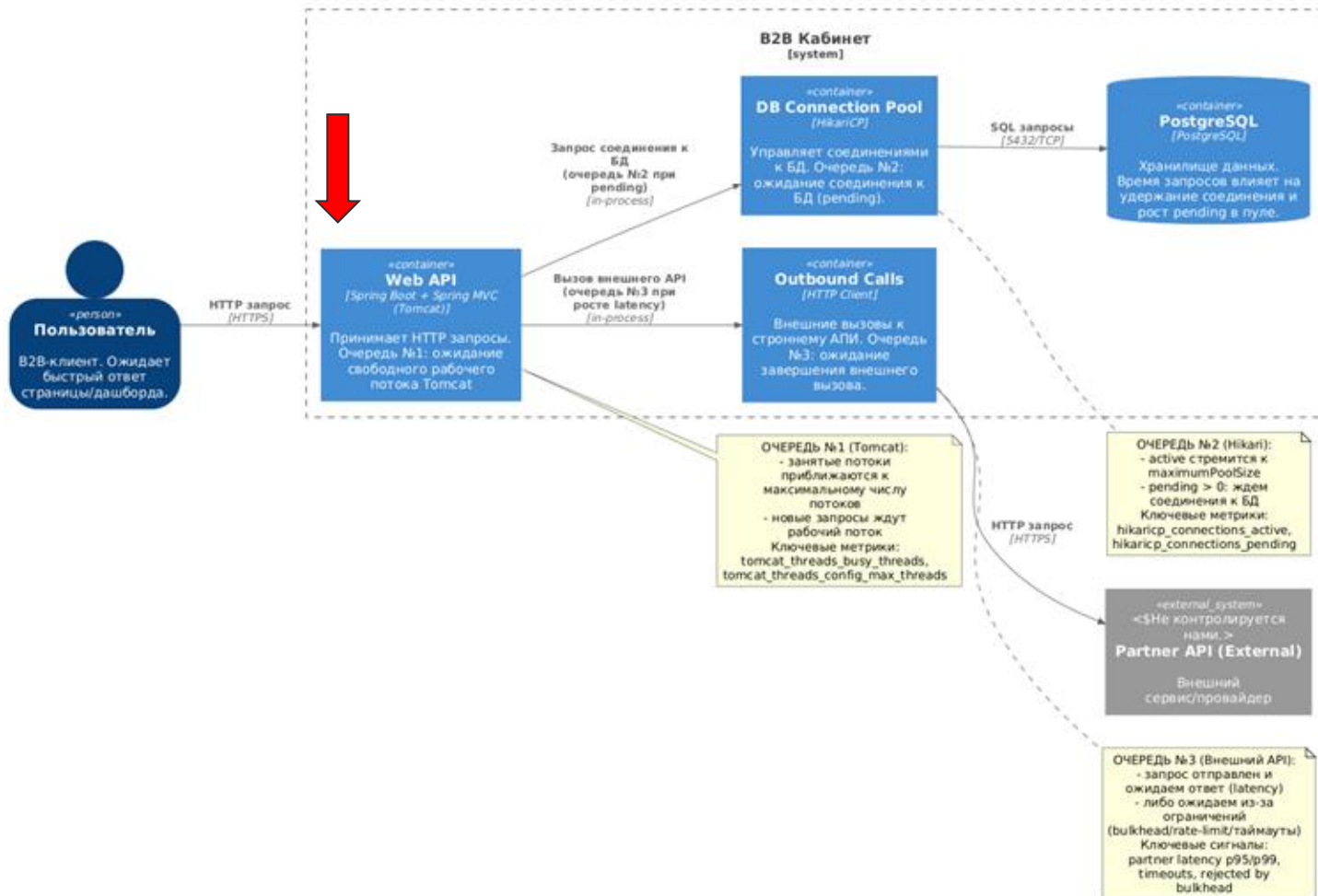
```
app > src > main > java > ru > jug > demo > hangdemo > repo > ProfileRepository > ProfileRepository
application.yml ProfileRepository.java ProfileService.java DashboardService.java
1 package ru.jug.demo.hangdemo.repo;
2
3 import ...
4
5 @Repository
6 public class ProfileRepository {
7
8     private final JdbcTemplate jdbcTemplate; @Usage
9     private final DbBulkhead dbBulkhead; @Usage
10
11     public ProfileRepository(JdbcTemplate jdbcTemplate, DbBulkhead dbBulkhead) {
12         this.jdbcTemplate = jdbcTemplate;
13         this.dbBulkhead = dbBulkhead;
14     }
15
16     public Profile findById(int id) { @Usage
17         return dbBulkhead.call(() ->
18             jdbcTemplate.queryForObject(
19                 sql "select id, name, segment, previous_score from profiles where id = ?",
20                 (ResultSet rs, int rowNum) -> new Profile(
21                     rs.getInt(columnLabel "id"),
22                     rs.getString(columnLabel "name"),
23                     rs.getString(columnLabel "segment"),
24                     rs.getInt(columnLabel "previous_score")
25                 ),
26                 id
27             );
28         );
29     }
30
31     public int updatePreviousScore(int id, int previousScore) { @Usage
32         return dbBulkhead.call(() ->
33             jdbcTemplate.update(
34                 sql "update profiles set previous_score = ? where id = ?",
35                 previousScore, id
36             );
37         );
38     }
39 }
40
41
42
```

```
app > src > main > java > ru > jug > demo > hangdemo > bulkhead > DbBulkhead > call
application.yml ProfileRepository.java DbBulkhead.java ProfileService.java DashboardService.java

1  > import
2
3
4
5
6
7
8
9
10
11 public class DbBulkhead { 8 usages
12
13     private final Semaphore semaphore; 3 usages
14     private final long waitMs; 2 usages
15
16
17     private final Counter rejected; 2 usages
18     private final Timer guardedDbOps; 2 usages
19
20     public DbBulkhead(DbBulkheadProperties props, MeterRegistry registry) { 1 usage
21         this.semaphore = new Semaphore(props.maxConcurrent(), true);
22         this.waitMs = props.waitMs();
23
24         this.rejected = Counter.builder(name("demo_db_bulkhead_rejected_total"))
25             .description("DB bulkhead rejected calls (could not acquire permit fast enough)")
26             .register(registry);
27
28         this.guardedDbOps = Timer.builder(name("demo_db_bulkhead_op_seconds"))
29             .description("Time spent executing DB operations under bulkhead (includes JDBC time)")
30             .publishPercentileHistogram()
31             .register(registry);
32     }
33
34     public <T> T call(Callable<T> action) { 2 usages
35         boolean acquired = false;
36         try {
37             acquired = semaphore.tryAcquire(waitMs, TimeUnit.MILLISECONDS);
38             if (!acquired) {
39                 rejected.increment();
40                 throw new DbBulkheadRejectedException("DB busy: bulkhead limit reached");
41             }
42             return guardedDbOps.recordCallable(action);
43         } catch (DbBulkheadRejectedException e) {
44             throw e;
45         } catch (Exception e) {
46             throw new RuntimeException(e);
47         } finally {
48             if (acquired) semaphore.release();
49         }
50     }
51
52     > public void run(Runnable action) { call() -> { action.run(); return null; }; }
53 }
```

Наш API

Анатомия зависания



Просто увеличить maxThreads”

- Увеличение потоков часто:
 - увеличивает конкуренцию за БД
 - увеличивает количество ожидающих потоков
 - ухудшает tail latency
- **Вывод: тюнинг потоков работает только вместе с лимитами ниже по стеку**

app > src > main > resources > application.yml

application.yml

```
1  server:
2    port: ${SERVER_PORT:8080}
3    tomcat:
4      mbeanregistry:
5        enabled: ${TOMCAT_MBEANREGISTRY_ENABLED:true}
6      threads:|
7        max: ${TOMCAT_THREADS_MAX:500}
8        min-spare: ${TOMCAT_THREADS_MIN_SPARE:50}
9        accept-count: ${TOMCAT_ACCEPT_COUNT:500}
```

Rate limiter

- Пропускная способность системы конечна: БД/партнёр/пулы ограничены
- Если входной поток выше → очередь растёт → p99 растёт
- Rate limiter вводит честный контракт:
 - лучше быстрый 429/503
 - чем массовые 30-секундные зависания

Как выбрать лимит

- Смотрим метрики после внедрения timeouts + bulkhead + CB + фикса Tx
- Определяем стабильный потолок throughput без роста очередей
- Ставим лимит чуть ниже этого потолка (с запасом)
- Проверяем в нагрузке и корректируем

p95 latency (dashboard)



p99 latency (dashboard)



RPS (dashboard)



Tomcat threads: busy vs max



Tomcat threads: current







Tomcat threads: busy vs max



Tomcat threads: current



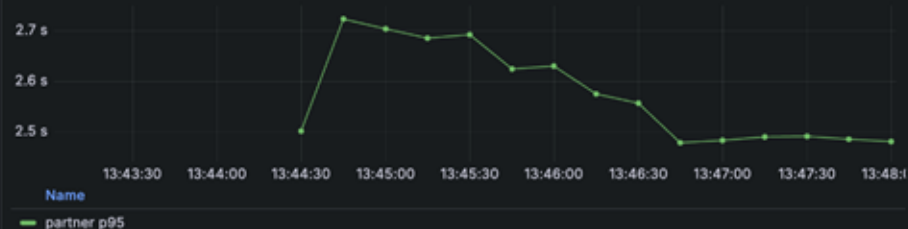
HikariCP connections: active / idle / pending



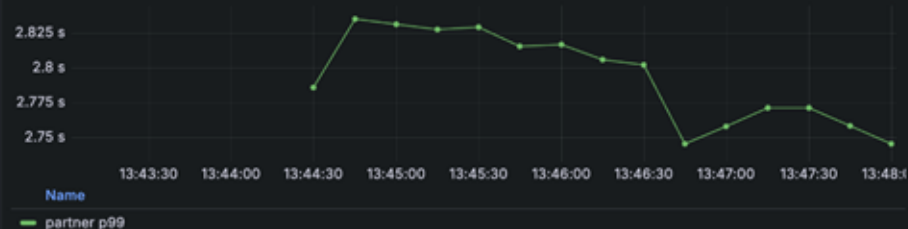
HikariCP peaks (max over 30s): active / pending



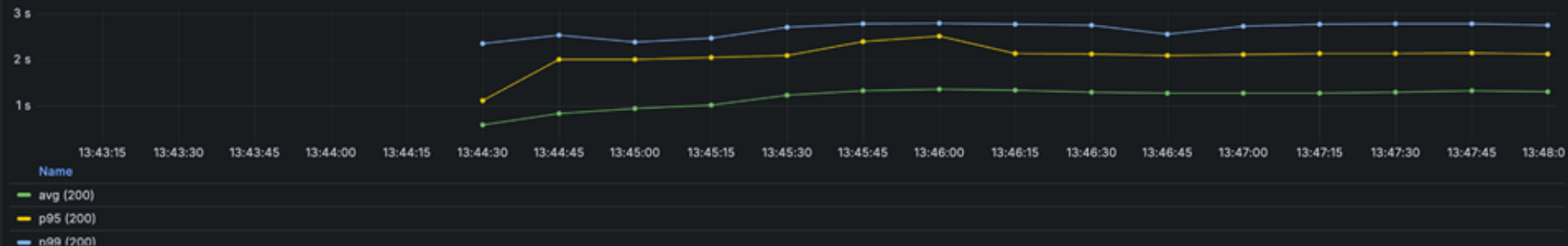
Partner API latency p95 (demo.partner.call)



Partner API latency p99 (demo.partner.call)



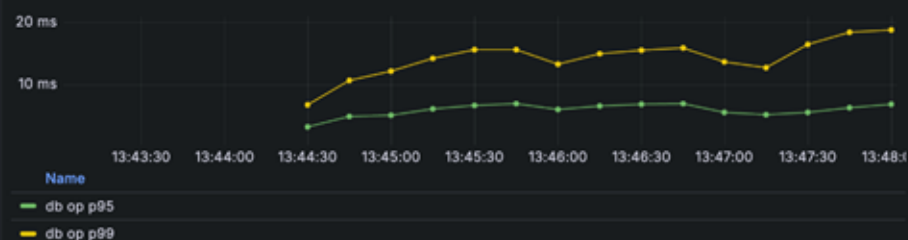
Latency (200): avg / p95 / p99 (dashboard)



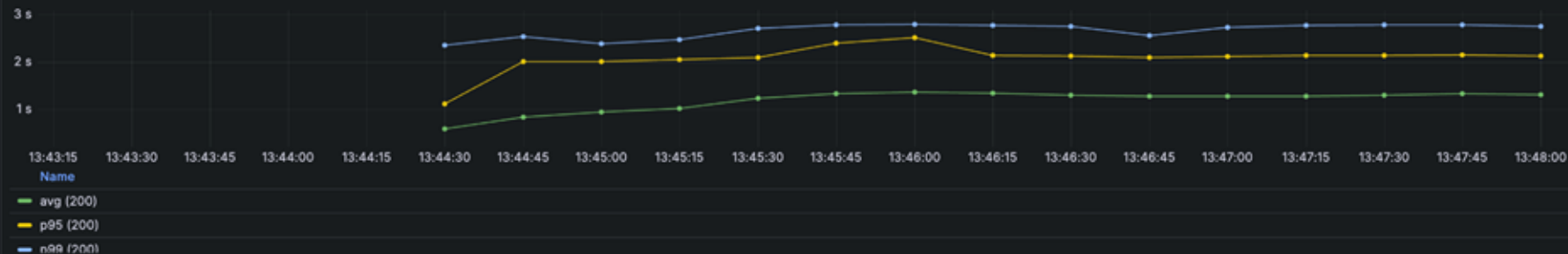
DB bulkhead rejected (RPS) — demo_db_bulkhead_rejected_total



DB bulkhead op latency p95/p99 — demo_db_bulkhead_op_seconds



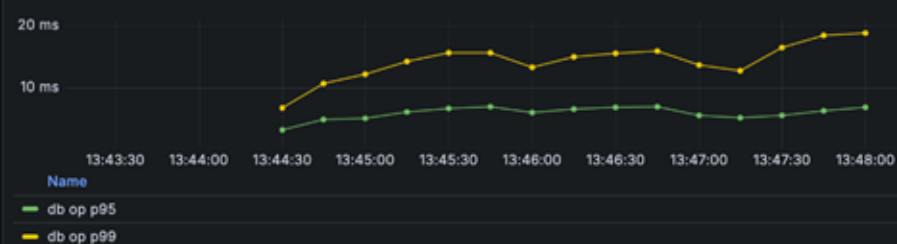
Latency (200): avg / p95 / p99 (dashboard)



DB bulkhead rejected (RPS) — demo_db_bulkhead_rejected_total



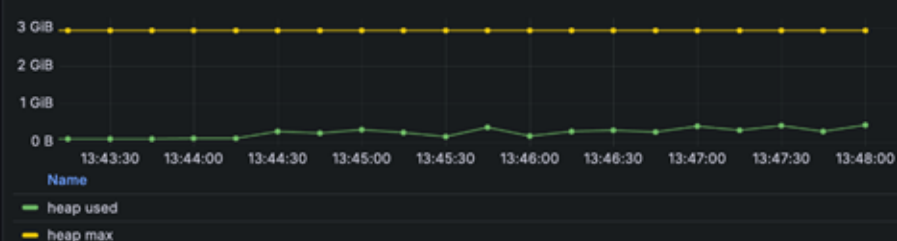
DB bulkhead op latency p95/p99 — demo_db_bulkhead_op_seconds



Process CPU (JVM) %



JVM Heap used vs max



Итог

- Низкий CPU не означает, что система здорова
- Здоровье = контроль очередей ожидания
- Три главные очереди: Tomcat / Hikari / Partner
- Порядок лечения важнее “модных технологий”
- Virtual Threads помогают, но только на подготовленном фундаменте

Практический чеклист

1. Дашборд “3 очереди” в Grafana
2. Time budget + timeouts на внешние зависимости
3. Bulkhead на партнёра (in-flight лимит)
4. Circuit breaker на партнёра
5. Убрать сетевые ожидания из транзакций + bulkhead на БД
6. Rate limiter на входе
7. Потом - Virtual Threads (если есть смысл)

Песочница

- Репозиторий с кодом
- Конфиги (Tomcat/Hikari/Resilience patterns)
- Grafana dashboard JSON
- Скрипты нагрузки / сценарии деградации
- Инструкция “как воспроизвести локально”
- Ссылки

Q&A