ChatGPT

Rygorystyczne obalenie błędu architektonicznego w torze ingestu danych

Streszczenie ustaleń i wniosków

Rodzaj „błędu do obalenia" był na wejściu **nieokreślony**, więc najpierw zbudowałem katalog 5 konkretnych klas błędów, które da się obalać rygorystycznie (formalnie lub empirycznie) i które pasują do analizowanych artefaktów (repozytoria + dokumenty). Następnie — bez dodatkowych preferencji użytkownika — wybrałem typ **najbardziej ogólnie istotny i najmocniej „dowodliwy" z materiału repozytoryjnego**: błąd architektoniczny w systemie rozproszonym dotyczący odporności na przeciążenie i kaskady.

W repozytorium `DonkeyJJLove/swarm` w komponencie UDP-agregatora występuje wzorzec „**jeden pakiet UDP → jeden nowy wątek**" oraz wysyłka HTTP bez jawnego ustawienia timeoutu. To sprzyja sytuacji, w której liczba aktywnych wątków może rosnąć bez projektowego ograniczenia, co w warunkach przeciążenia przeradza się w typowy mechanizm kaskad (przeciążenie → utrata zasobów → degradacja → retry/utrata zdrowia → restart/pętla awarii). Mechanika kaskad, a także konkretne ryzyka „wątkowe" (thread overhead, thread starvation, rosnące kolejki) są dobrze opisane w literaturze inżynierii niezawodności. [1]

Wniosek: zdefiniowany błąd (fałszywe przekonanie, że „thread-per-message + service-meshowe progi" zapewnia stabilność i ograniczone zużycie zasobów w tym torze) został obalony rygorystycznie: formalnie przez kontrprzykład oparty o semantykę uruchamiania wątków oraz zachowanie wywołań HTTP bez timeoutu, oraz inżyniersko-empirycznie przez odniesienie do znanych mechanizmów kaskad i wyczerpywania zasobów. [2]
Ocena „jak bardzo naukowe" obalenie jest na końcu raportu; część wniosków wymaga doprecyzowania danych runtime (limity, ruch, metryki), których nie ma w repo.

Katalog opcji obalenia błędu przy nieokreślonym typie

Poniżej celowo proponuję typy „błędów do obalenia", które prowadzą do **innych standardów rygoru**. To jest ważne: „absolutnie naukowe" w matematyce znaczy co innego niż „absolutnie naukowe" w inżynierii systemów rozproszonych.

Proponowany typ błędu do obalenia	Krótkie streszczenie problemu	Dlaczego warto obalić	Minimalne kryteria rygoru „naukowego”
Błąd dowodu matematycznego	W dowodzie pojawia się krok nieuprawniony (np. dzielenie przez zero, niejawne założenie o niezerowości, błąd kwantyfikatorów).	Pozwala uniknąć „fałszywych fundamentów” i łańcuchów błędnych wniosków.	Formalizacja twierdzenia i założeń; poprawność w systemie aksjomatycznym; dowód kontrprzykładem lub wskazanie wadliwego kroku; możliwość weryfikacji przez niezależny rachunek/ assistant proof.
Błąd logiczny w argumentacji	Wniosek nie wynika z przesłanek (np. potwierdzanie następstwa, fałszywa dychotomia).	Ogranicza podatność na manipulację i błędne decyzje.	Rekonstrukcja argumentu w logice formalnej; rozróżnienie poprawności (validity) i prawdziwości przesłanek (soundness); kontrprzykład semantyczny.
Fałszywe twierdzenie naukowe lub popularnonaukowy mit	Twierdzenie o świecie empirycznym, które nie wytrzymuje danych (np. „X zawsze powoduje Y”).	Zmniejsza ryzyko destrukcyjnych decyzji w zdrowiu/ technice/polityce.	Operacjonalizacja pojęć; projekt falsyfikacji; replikowalność; źródła pierwotne; analiza błędów pomiaru i confounders; metaanaliza lub przegląd systematyczny (gdy adekwatne).
Błąd eksperymentu lub metryk (Goodhart, p-hacking, błędna walidacja)	Metryka mierzy nie to, co myślimy; optymalizacja KPI pogarsza outcome.	Chroni przed „wygrywaniem metryki” kosztem realnej jakości/ bezpieczeństwa.	Pre-rejestracja hipotez i primary endpoints; kontrola wielokrotnych porównań; walidacja konstruktu; raportowanie niepewności, nie tylko progu. [1]
Błąd architektoniczny w systemie (kod/ konfiguracja)	Błędne przekonanie o własnościach systemu (np. „ten wzorzec gwarantuje stabilność”), które nie zachodzi w modelu obciążenia/awarii.	To często błędy o najwyższej dźwigni: powodują kaskady, koszty i ryzyko w produkcji.	Precyzyjna własność do obalenia (inwariant); model systemu (kolejki, zasoby, granice); dowód kontrprzykładem lub przez niespójność specyfikacji; możliwość replikacji (benchmark/ test). [1]

Dalsza część raportu wybiera ostatnią opcję (błąd architektoniczny), bo jest najbardziej „ogólnie istotna" dla systemów SaaS/AI i jednocześnie da się ją zdefiniować w sposób ścisły na podstawie artefaktów repo.

Materiał dowodowy i zakres analizy w GitHub [3]

Badanie oparłem na artefaktach z repozytoriów wskazanych przez użytkownika (w szczególności `sbom`, `swarm`, `ai_platform`, `HA2D`, `chunk-chunk`, `writeups`) oraz na źródłach pierwotnych/official dotyczących semantyki wątków, timeoutów i mechaniki kaskad w systemach. [2]

Wymiary, które muszą być jawne, aby „dowód" był naukowo uczciwy

Wymogi użytkownika sugerują, że sam „dowód logiczny" bez kontekstu wdrożeniowego może być pozorny. Poniżej mapuję kluczowe wymiary i stan danych.

Wymiar	Co należy ustalić	Co da się ustalić z repo/ dokumentów	Czego brakuje i jest nieokreślone
Twierdzenia repo (claims)	Jaką własność system deklaruje (np. odporność na kaskady)	W dokumentacji polityk mesh jest nacisk na „resilience" (rate limiting, circuit breakers itd.).	Brak jednoznacznej, mierzalnej definicji „stabilności" dla całego toru UDP→HTTP.
Kod (mechanizm)	Co program naprawdę robi w runtime	W `swarm/aggregator/aggregator.py` jest wzorzec: `recvfrom()` + `Thread(...).start()` per pakiet; w `requests.post()` brak jawnego `timeout`.	Brak telemetryki runtime (liczba wątków, czasy obsługi, błędy).
Architektura	Gdzie są progi, gdzie są kolejki, gdzie backpressure	Istnieją polityki Envoy/Istio (globalny rate limit, outlier detection). [4]	Nie wiadomo, czy UDP ingress jest w jakikolwiek sposób limitowany (NetworkPolicy, iptables, bufory, QoS).
Założenia kosztowe	Jaki jest koszt awarii i degradacji	Materiały projektowe akcentują potrzebę metryk jednostkowych i falsyfikowalności.	Brak danych billing/ FinOps oraz kosztów incydentów.
Metryki i testowalność	Jak mierzyć „kaskadę" i „amortyzację"	Jest propozycja metryk przeciążeniowych (p95/p99, MTTR, error budget) w materiałach.	Brak wyników testów obciążeniowych oraz definicji WU (work unit) w samym torze swarm.

Wymiar	Co należy ustalić	Co da się ustalić z repo/ dokumentów	Czego brakuje i jest nieokreślone
Powtarzalność	Czy da się odtworzyć wynik	Kod i konfiguracje są jawne; można zbudować test.	Brak „golden workload” i procedury prerejestracji testu; brak danych produkcyjnych.
Dostępność danych	Czy są logi, trace, profiling	W repo są wzorce eventów/ telemetrii (np. w `sbom`).	Brak rzeczywistych logów/trace z toru swarm.
Ograniczenia prawne/ etyczne	Jak testować bez szkody i zgodnie z prawem	Źródła UE podkreślają zasady minimalizacji, celu i bezpieczeństwa przy przetwarzaniu danych osobowych. [5]	Nie wiemy, czy telemetryka obejmuje dane osobowe (np. identyfikatory urządzeń/osób); brak DPIA/opisu retencji.

Wniosek metodologiczny: „obalać” można już teraz **własność logiczno-architektoniczną** (brak projektowej granicy zasobów), ale pełna ocena skutków ekonomicznych i produkcyjnych wymaga brakujących danych.

Wybrany błąd i precyzyjne twierdzenie do obalenia

Dlaczego ten wybór

Wybrałem błąd architektoniczny, bo: - ma bezpośrednią dźwignię na niezawodność i koszty systemów, - da się go zdefiniować jako własność (inwariant) i obalić kontrprzykładem, - jego obalenie nie wymaga wchodzenia w kontrowersje ontologiczne ani w niepewne dane zewnętrzne.

Konkretny błąd do obalenia

Błędne przekonanie (B) — sformułowane precyzyjnie:

> W torze `UDP → aggregator → HTTP POST → aggregator-api`, zastosowanie strategii „każdy odebrany pakiet UDP obsłuż w osobnym wątku”, w połączeniu z politykami service mesh (rate limiting, outlier detection/circuit breaking), zapewnia stabilność w sensie inżynierskim: istnieje projektowy, niezależny od ruchu limit na równoczesne zużycie kluczowych zasobów procesu agregatora (w szczególności liczby żywych wątków), a zatem tor nie może wejść w reżim kaskady zasobowej.

To jest „dobre” twierdzenie do obalenia, bo zawiera sprawdzalną własność: *istnienie stałej granicy na liczbę aktywnych wątków wynikającej z projektu*, a nie z przypadkowych limitów OS/kubleta.

Materiał repozytoryjny, na którym opiera się obalenie

Najistotniejsze fragmenty kodu/konfiguracji (źródło: wskazane repozytoria; linki poniżej):

- `swarm/aggregator/aggregator.py`: pętla `recvfrom()` i `threading.Thread(...).start()` dla każdego pakietu, wątek daemon; w `handle_message` jest `requests.post(...)` bez parametru `timeout`.
- `swarm/infrastructure/istio/policies/rate-limit.yaml`: filtr HTTP rate limit na ingressgateway, z `failure_mode_deny: true`.
- `swarm/infrastructure/istio/policies/circuit-breaker.yaml`: `outlierDetection` dla `aggregator-api-service...`, bez jawnych limitów puli połączeń w tej konkretnej wersji pliku.

Repo-odnośniki (URL w bloku kodu z uwagi na zasady cytowania):

```
https://github.com/DonkeyJJLove/swarm/blob/master/aggregator/aggregator.py
https://github.com/DonkeyJJLove/swarm/blob/master/aggregator/mqtt_bridge/
mqtt_bridge.py
https://github.com/DonkeyJJLove/swarm/blob/master/infrastructure/istio/
policies/rate-limit.yaml
https://github.com/DonkeyJJLove/swarm/blob/master/infrastructure/istio/
policies/circuit-breaker.yaml
```

Dowód obalający błędne przekonanie

Definicje i założenia

Definicja zasobu krytycznego: Niech `N(t)` oznacza liczbę wątków procesu `aggregator` będących w stanie „alive” w chwili `t`.

Definicja stabilności w sensie B: Istnieje stała `K` taka, że dla każdego `t` zachodzi `N(t) ≤ K`, a `K` wynika z projektu (np. z rozmiaru puli wątków i kolejki), nie z przypadkowego limitu zasobów OS.

Założenia, które muszą być jawne (i są realistyczne):

A1. Każde wywołanie `Thread.start()` uruchamia `run()` „w osobnym wątku sterowania” (separate thread of control). [6]
A2. Jeśli wątek nie zakończy `run()`, to pozostaje „alive” (w sensie `is_alive()`). [6]
A3. Wywołania HTTP w bibliotece Requests **domyślnie nie timeoutują**, o ile timeout nie zostanie ustawiony jawnie; bez timeoutu kod może „wisieć” długo. [7]
A4. Obciążenie wejściowe może mieć postać „bursta”: skończona liczba `M` pakietów może nadejść w krótkim interwale (to nie jest instrukcja ataku; to standardowa własność ruchu w systemach sieciowych i testach obciążeniowych).
A5. System ma skończone zasoby (RAM/limity wątków), więc przekroczenie pewnych progów powoduje degradację, a w skrajności restarty/OOM-kill (to jest empiryczne założenie o świecie, nie o logice programu).

Lemat o braku projektowej granicy wątków

Lemat L1: W rozpatrywanej implementacji agregatora nie występuje żadna konstrukcja ograniczająca liczbę wątków `N(t)` przez stałą wynikającą z projektu.

Uzasadnienie (z konstrukcji kodu): dla każdego pakietu odebranego przez `recvfrom()` wykonywane jest `Thread(...).start()` bez warunku i bez semafora/puli. W takim schemacie jedyna „granica" to: (i) przypadkowa szybkość CPU, (ii) limity jądra/środowiska uruchomieniowego, (iii) to, czy wątki kończą się szybko. Żaden z tych czynników nie jest projektowym `K`.

L1 jest obserwacją konstrukcyjną, ale do obalenia B potrzebujemy mocniejszego kroku: kontrprzykładu, w którym `N(t)` przekracza dowolny próg.

Kontrprzykład obalający B

Twierdzenie T: Dla każdego `K ∈ ℕ` istnieje przebieg zdarzeń zgodny z A1–A4, w którym w pewnej chwili `t*` zachodzi `N(t*) > K`.

Dowód (kontrprzykładem):

1. Weź dowolne `K ∈ ℕ`.
2. Rozważ `K+1` pakietów UDP, które docierają do pętli `recvfrom()` w bardzo krótkim interwale (A4).
3. Dla każdego odebranego pakietu program wywołuje `Thread.start()`, co — zgodnie z A1 — uruchamia `handle_message` w osobnym wątku sterowania. [6]
4. W `handle_message` następuje `requests.post(...)` bez jawnego timeoutu. Z A3 wynika, że istnieją sytuacje, w których wywołanie nie kończy się szybko (może „wisieć" długo, bo brak timeoutu). [7]
5. Skonstruuj (teoretycznie) środowisko, w którym odpowiedź serwera docelowego nie nadchodzi przez pewien czas `Δ > 0` (to może wynikać z przeciążenia downstream, problemów sieciowych lub awarii; nie jest to dodatkowe założenie o kodzie, tylko dopuszczalny stan świata). Wtedy każdy z `K+1` wątków pozostaje alive co najmniej przez `Δ`. [7]
6. W chwili `t*` tuż po uruchomieniu `(K+1)`-tego wątku wszystkie `(K+1)` wątki są alive, więc `N(t*) ≥ K+1 > K`. (A2 formalizuje znaczenie „alive".) [6]

Ponieważ `K` było dowolne, obalamy istnienie stałej granicy `K` wynikającej z projektu. To przeczy treści błędnego przekonania B. □

Dlaczego polityki service mesh nie „ratują" obalonej własności

W repozytorium są polityki typu rate limiting i outlier detection, ale one nie wprowadzają projektowego `K` dla wątków agregatora:

- **Rate limit HTTP (Envoy)** dotyczy ruchu HTTP i zachowania proxy w razie „over limit" (429) albo błędu usługi limitującej przy `failure_mode_deny=true` (fail-closed). [8]
  To nie jest mechanizm ograniczający liczbę wątków w procesie UDP-agregatora, bo wątki powstają *przed* tym, zanim proxy zwróci 429/500 downstream, i mogą wisieć na braku timeoutu. [7]

- **Outlier detection / circuit breaking (Envoy/Istio)** polega na wyłączaniu z LB endpointów uznanych za „outliers” (ejection) według algorytmu outlier detection. [9]
  To mechanizm ochrony ruchu *do upstreamów*, a nie limit wątków w komponencie, który produkuje żądania. W szczególności ejection nie zastępuje backpressure w komponencie „producer” (agregator), który może dalej generować pracę i zużywać zasoby.

Skutki inżynierskie i powiązanie z literaturą pierwotną

Formalny kontrprzykład mówi tylko: „nie ma projektowej granicy `N(t)`”. Aby przejść do wniosku o kaskadach, potrzebujemy A5 (świat ma skończone zasoby). Wtedy rosnące `N(t)` prowadzi do klasycznych symptomów przeciążenia: koszt wątków w RAM, thread starvation, rosnące kolejki i w skrajności niestabilność procesu, co jest typową przyczyną kaskad. [1]

Wprost: literatura SRE wymienia w sekcji o wątkach, że gdy serwer „dodaje wątki w miarę potrzeby”, narzut wątków może zużyć zbyt dużo RAM; omawia też thread starvation oraz współzależności zasobów w przeciążeniu. [1]

Możliwe kontrargumenty i ograniczenia dowodu

Kontrargument C1: „UDP i bufory jądra zrzucą pakiety, więc wątków nie będzie nieskończenie dużo.”
Odpowiedź: to może ograniczyć *faktyczną* maksymalną szybkość wzrostu `N(t)` w danym środowisku, ale nie tworzy projektowej granicy `K`. Dowód obala B w sensie „gwarancji architektonicznej”; bufor OS jest parametrem środowiska, nie inwariantem projektu.

Kontrargument C2: „W praktyce `requests.post` zwykle szybko odpowiada.”
Odpowiedź: B mówi o stabilności jako gwarancji w torze, a nie o „typowym przypadku”. Brak timeoutu oznacza, że istnieją dopuszczalne (i w systemach rozproszonych częste) stany świata, w których wywołanie trwa długo. To wystarczy do kontrprzykładu. [7]

Kontrargument C3: „Kubernetes ma limity; OOM-kill i restart to ‘mechanizm stabilizacji’.”
Odpowiedź: restart jest mechanizmem *odtworzeniowym*, nie stabilizacyjnym. Jeśli przyczyna (bursty + brak backpressure) trwa, restart prowadzi do crash-loop i może pogłębiać kaskadę (usługa wraca i natychmiast jest bombardowana). To jest opisane jako typowy mechanizm w kaskadach przy przeciążeniu. [1]

Ograniczenie L1: brak danych o realnym ruchu i limitach
Dowód obala gwarancję projektową bez telemetryki. Natomiast *siła praktyczna* wniosku („to na pewno uderzy w produkcji”) zależy od: rozkładu ruchu, zasobów, topologii, a także od tego, czy upstream jest stabilny. Do potwierdzenia/kalibracji potrzebne są dane wymienione w końcowej sekcji „dalsza weryfikacja”.

Alternatywne ścieżki dowodowe i porównanie

Poniższa tabela pokazuje, że obalenie tego typu błędu można przeprowadzić kilkoma ścieżkami o innym udziale formalizmu i empirii.

Metoda obalenia	Jaki błąd obala	Rdzeń dowodu	Formalne vs empiryczne	Co jest potrzebne	Główne ograniczenie
Kontrprzykład semantyczny wątków i timeoutów (użyty w raporcie)	„Istnieje projektowa granica na liczbę aktywnych wątków w agregatorze”	Dla dowolnego K konstruujemy przebieg z `N(t)>K` przez burst + brak timeoutu	Wysoko formalne (semantyka start/is_alive) + minimalne empiryczne (świat ma opóźnienia) [10]	Kod + podstawowe fakty o bibliotece	Nie wyznacza progu, przy którym *praktycznie* nastąpi awaria
Argument z teorii kolejek / stabilności (λ vs μ)	„System jest stabilny przy dowolnym natężeniu wejścia”	Jeśli średnia szybkość napływu przekracza zdolność obsługi, kolejki/zasoby nasycają się	Mieszane: formalizm zależny od modelu (M/M/1 itp.) + empiryczna estymacja parametrów	Oszacowania λ, μ, rozkłady czasów obsługi	Wrażliwość na błędną parametryzację modelu
Empiryczny test obciążeniowy z prerejestracją endpointów	„W tym torze nie ma kaskad zasobowych”	Kontrolowany load + pomiar (threads, RSS, p99, błędy)	Empiryczne z rygorem metodologicznym (prerejestracja, powtarzalność)	Środowisko testowe, generator ruchu, telemetryka	Koszt i ryzyko testów; wymaga danych/konfiguracji
Dowód niespójności dokumentacji i konfiguracji (wariant alternatywny)	„Polityka X jest wdrożona, bo README tak mówi”	README opisuje limity connectionPool, ale konkretna wersja YAML może ich nie zawierać	Formalne (porównanie artefaktów)	Dostęp do wersji README i YAML, wersjonowanie	Obala roszczenie dokumentacyjne, niekoniecznie własność runtime
Dowód z literatury SRE o kaskadach	„Wątki/queue mogą rosnąć bezkarnie”	Mechanizmy przeciążenia i kaskad: thread overhead, queue saturation, retries	Empiryczno-teoretyczne (best practices) [1]	Źródła pierwotne i interpretacja	To argument ogólny; bez kodu i kontrprzykładu bywa „zbyt ogólny”

Diagram struktury dowodu i zależności logicznych

```
flowchart TD
  A[Wejście: błąd nieokreślony] --> B[Wybór typu: błąd architektoniczny]
  B --> C[Twierdzenie B: istnieje projektowy limit K na liczbę wątków]
  C --> D[Obserwacja kodu: Thread.start() per pakiet]
  D --> E[Lemat: start() uruchamia run() w osobnym wątku]
  E --> F[Obserwacja kodu: requests.post bez timeout]
  F --> G[Lemat: bez timeoutu żądanie może wisieć długo]
```

```
G --> H[Kontrprzykład: burst K+1 pakietów + opóźniony upstream]
H --> I[Wynik: N(t*) >= K+1]
I --> J[Sprzeczność z twierdzeniem B]
J --> K[Wniosek: błąd obalony; brak gwarancji stabilności]
K --> L[Warstwa empiryczna: skończone zasoby -> ryzyko kaskady]
```

Semantyka `start()` i brak timeoutu są kluczowymi węzłami dowodu. [10]

Ocena rygoru, formalność kontra empiryka, wniosek i dalsza weryfikacja

Które elementy obalenia są formalne, a które empiryczne

Element	Status	Uzasadnienie
„`Thread.start()` uruchamia `run()` w osobnym wątku sterowania”	Formalne (specyfikacja biblioteki)	Wynika bezpośrednio z dokumentacji języka/biblioteki. [6]
„Wątek jest alive od startu do zakończenia run()”	Formalne	Definicja `is_alive()` w dokumentacji. [6]
„Requests domyślnie nie timeoutuje; bez timeoutu żądanie może wisieć długo”	Formalne (specyfikacja zachowania biblioteki)	Opisane w dokumentacji Requests. [7]
„Dla dowolnego K istnieje przebieg z N(t)>K”	Formalne (kontrprzykład)	Wynika logicznie z trzech punktów powyżej + konstrukcji bursta.
„To może wywołać kaskady/ awarie w praktyce”	Empiryczne (ale silnie ugruntowane)	Zależy od skończonych zasobów i realnego obciążenia; mechanika kaskad jest dobrze opisana w literaturze SRE. [1]
„Service mesh rate limiting/ outlier detection nie tworzą limitu wątków w procesie”	Mieszane	Formalne co do semantyki proxy (429/500, ejection) + empiryczne co do topologii wdrożenia. [4]

Czy błąd został obalony naukowo według kryteriów

Według kryteriów zdefiniowanych dla „błędu architektonicznego” (precyzyjna własność, model, dowód kontrprzykładem, oparcie o źródła pierwotne, możliwość replikacji):

- **Precyzyjne twierdzenie do obalenia**: tak (istnienie projektowego limitu `K` na liczbę wątków).
- **Dowód kontrprzykładem**: tak (konstrukcja dla dowolnego `K`).
- **Oparcie o źródła pierwotne/official**: tak — dokumentacja `threading` i Requests oraz literatura SRE o kaskadach. [2]
- **Replikowalność**: częściowa — logiczne obalenie jest replikowalne z samego kodu, ale praktyczny wpływ wymaga testu i metryk.

Werdykt: błąd został obalony **rygorystycznie** w sensie formalnym (brak gwarancji projektowej) i **uzasadniony inżyniersko** co do ryzyk kaskad. Pełna „twarda" predykcja produkcyjna (jak szybko i przy jakim ruchu „pęknie") wymaga danych empirycznych.

Dane brakujące do „domknięcia" obalenia w reżimie produkcyjnym

Aby przejść od obalenia gwarancji do ilościowego ryzyka i planu mitygacji, potrzebne są (nie są w repo):

1. **Telemetria procesu aggregator**: `N(t)` (liczba wątków), RSS, CPU, czas obsługi `handle_message`, odsetek błędów JSON/HTTP.
2. **Charakterystyka wejścia UDP**: rozkład burstów, pps, źródła ruchu.
3. **SLO/SLA** toru: np. dopuszczalny p99 opóźnienia i poziom strat danych.
4. **Konfiguracje runtime**: limity kontenera (requests/limits), ulimit wątków/plików, ustawienia buforów socket.
5. **Zasady legal/etyka telemetryki**: czy payload zawiera dane osobowe; jeśli tak, projekt powinien spełniać zasady legalności, minimalizacji, ograniczenia celu, retencji i bezpieczeństwa. [5]

Bezpieczeństwo i zgodność

Raport nie zawiera instrukcji działań niezgodnych z prawem ani wskazówek umożliwiających szkodzenie systemom (np. „jak przeprowadzić flood"). Analiza posługuje się abstrakcyjnymi modelami obciążenia, które są standardem w inżynierii niezawodności i testach.

Dalsze badania i weryfikacje

1. **Test obciążeniowy w izolacji** z prerejestracją primary endpoints: `max N(t)`, p99 latencji, OOM-kill, degradacja; zgodnie z praktyką „load test until components break" i z zaleceniami unikania kaskad poprzez kontrolę kolejek/zasobów. [1]
2. **Wprowadzenie projektowego ogranicznika**: pula wątków + ograniczona kolejka (fail-fast / load shedding), zamiast „thread per packet"; SRE opisuje, że kolejki powinny być kontrolowane i zwykle lepiej odrzucać żądania wcześnie niż gromadzić długi ogon. [1]
3. **Jawne timeouty i retry policy** w kliencie HTTP: brak timeoutu jest znanym źródłem blokad. [7]
4. **Spójność progów między warstwami**: dopiero po ustabilizowaniu zasobów aplikacji ma sens strojenie `failure_mode_deny`, outlier detection i ewentualnych limitów connectionPool (Istio/Envoy). [11]
5. **Audyt danych i DPIA** (jeśli telemetryka obejmuje dane osobowe): doprecyzować podstawę prawną, retencję i minimalizację danych. [12]

---

[1] https://sre.google/sre-book/addressing-cascading-failures/
https://sre.google/sre-book/addressing-cascading-failures/

[2] [6] [10] https://docs.python.org/3/library/threading.html
https://docs.python.org/3/library/threading.html

[3] [9] https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/outlier.html
https://www.envoyproxy.io/docs/envoy/latest/intro/arch_overview/upstream/outlier.html

[4] [8] [11] https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rate_limit_filter
https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/rate_limit_filter

5 https://www.edpb.europa.eu/sme-data-protection-guide/faq-frequently-asked-questions/answer/what-are-basic-processing_en
https://www.edpb.europa.eu/sme-data-protection-guide/faq-frequently-asked-questions/answer/what-are-basic-processing_en

7 https://requests.readthedocs.io/en/stable/user/advanced/
https://requests.readthedocs.io/en/stable/user/advanced/

12 https://commission.europa.eu/law/law-topic/data-protection/rules-business-and-organisations/principles-gdpr/overview-principles/what-data-can-we-process-and-under-which-conditions_en
https://commission.europa.eu/law/law-topic/data-protection/rules-business-and-organisations/principles-gdpr/overview-principles/what-data-can-we-process-and-under-which-conditions_en