ქაოსის ინჟინერიის გამოყენება მიკროსერვისული არქიტექტურის მდგრადობის გასაძლიერებლად: „Online Boutique“-ის ქეისის ანალიზი

გიორგი კუჭავა, იოსებ ქართველიშვილი, შალვა ვაშალომიძე

საქართველოს ტექნიკური უნივერსიტეტი

kuchavagiorgi08@gtu.ge, s.kartvelishvili@gtu.ge, vashalomidze.shalva24@gtu.ge

რეზიუმე

თანამედროვე ღრუბლოვანი მიკროსერვისული არქიტექტურები, მიუხედავად მათი მოქნილობისა და მასშტაბირებადობისა, ხშირად აწყდება რთულ გამოწვევებს, როგორიცაა ფარული ხარვეზები, კასკადური მარცხი და სისტემის არაპროგნოზირებადი ქცევა. ეს სტატია იკვლევს ქაოსის ინჟინერიის, როგორც DevOps-ის პროაქტიული მეთოდის, გამოყენებას Google-ის „Online Boutique“ აპლიკაციის მაგალითზე, რომელიც განლაგებულია Google Kubernetes Engine-ზე (GKE). Chaos Mesh-ის გამოყენებით ჩატარებულმა ექსპერიმენტებმა (CPU Hog, Memory Hog, Network Latency, Packet Loss) გამოავლინა კრიტიკული სისუსტეები, მათ შორის „ჩუმი მარცხი“, მეხსიერების ამოწურვა (OOMKilled) და არაადეკვატური timeout/retry მექანიზმები. Prometheus-ისა და Grafana-ს მეშვეობით გაზომილი KPI-ებით (პასუხის დრო, შეცდომების სიხშირე, CPU/მეხსიერების მოხმარება) და Locust-ის მიერ სიმულირებული მომხმარებლის ტრაფიკით, განხორციელდა სისტემის ოპტიმიზაცია. ეს მოიცავდა Horizontal Pod Autoscaler-ის (HPA) დანერგვას, რესურსების ლიმიტების კორექტირებასა და timeout/retry მექანიზმების გაძლიერებას.

განმეორებითმა ექსპერიმენტებმა აჩვენა სისტემის მდგრადობის სტატისტიკურად მნიშვნელოვანი გაუმჯობესება, რაც ხაზს უსვამს ქაოსის ინჟინერიის, როგორც DevOps-ის განუყოფელი ნაწილის, მნიშვნელობას. სტატია გვთავაზობს პრაქტიკულ რეკომენდაციებს სისტემის სანდოობის გაძლიერებისა და SDLC-ში ქაოსის ინჟინერიის ინტეგრირებისთვის.

საკვანძო სიტყვები: ქაოსის ინჟინერია, მიკროსერვისები, Kubernetes, DevOps, Prometheus, Grafana.

1. შესავალი.

მიკროსერვისული არქიტექტურების ზრდამ, განსაკუთრებით ღრუბლოვან გარემოში, როგორიცაა Kubernetes, მნიშვნელოვნად გაზარდა პროგრამული სისტემების მოქნილობა და მასშტაბირებადობა. თუმცა, მათი განაწილებული ბუნება, რთული დამოკიდებულებები და ჰეტეროგენული ტექნოლოგიური სტეკი (მაგ., Go, C#, Python) ქმნის გამოწვევებს, როგორიცაა მოულოდნელი ხარვეზები, კასკადური მარცხი და „ჩუმი მარცხის“ ფენომენი, რომელიც ტრადიციული მონიტორინგის მეთოდებით ძნელად გამოვლენადია. ქაოსის ინჟინერია, რომელიც თავდაპირველად Netflix-ის მიერ განვითარდა, გვთავაზობს პროაქტიულ მიდგომას, სადაც განზრახ გამოწვეული ხარვეზები (მაგ., CPU-ს გადატვირთვა, ქსელის დაგვიანება) საშუალებას იძლევა გამოვლინდეს სისტემის სუსტი წერტილები, რაც DevOps გუნდებს ეხმარება სისტემის სანდოობის გაძლიერებაში. სტატია იკვლევს ქაოსის ინჟინერიის გამოყენებას Google-ის „Online Boutique“ აპლიკაციაზე, Kubernetes-ის გარემოში. „Online Boutique“ არის ღია კოდის, მრავალკომპონენტიანი ელექტრონული კომერციის პლატფორმა, რომელიც შეიცავს 11 მიკროსერვისს, მათ შორის frontend, productcatalogservice, cartservice და recommendationservice. კვლევის მიზნებია:

- სისტემის ფარული სისუსტეების გამოვლენა Chaos Mesh-ის გამოყენებით.
- მიზნობრივი ოპტიმიზაციების განხორციელება, მათ შორის HPA-სა და timeout-ების გაუმჯობესება.
- პრაქტიკული რეკომენდაციების შეთავაზება DevOps გუნდებისთვის.
- სტატია იყენებს ემპირიულ მონაცემებს Prometheus-ის, Grafana-სა და Locust-ის მეშვეობით, რათა აჩვენოს, თუ როგორ აძლიერებს ქაოსის ინჟინერია სისტემის მდგრადობას და ხელს უწყობს SDLC-ის გაუმჯობესებას.

2. მეთოდოლოგია.

კვლევა ჩატარდა Google Kubernetes Engine-ზე (GKE), „Online Boutique“ აპლიკაციის გამოყენებით, რომელიც წარმოადგენს ღია კოდის, მრავალკომპონენტიან ელექტრონული კომერციის პლატფორმას. აპლიკაციის ჰეტეროგენული ტექნოლოგიური სტეკი (Go, C#, Python, Node.js, Java) და რთული დამოკიდებულებები (gRPC, HTTP) მას იდეალურ ობიექტად აქცევს ქაოსის ინჟინერიისთვის. მეთოდოლოგიური ჩარჩო მოიცავდა ოთხ ფაზას: საბაზისო ანალიზი: სისტემის საწყისი მდგომარეობის გაზომვა Prometheus-ისა და Grafana-ს გამოყენებით.

KPI-ები: პასუხის დრო (Latency), შეცდომების სიხშირე (Error Rate), CPU/მეხსიერების მოხმარება.

Locust-ით სიმულირებული იყო მომხმარებლის ტრაფიკი (500 მომხმარებელი, 10 RPS).

ქაოსის ექსპერიმენტები:

Chaos Mesh-ის გამოყენებით ჩატარდა ოთხი ტიპის ექსპერიმენტი: CPU Hog: frontend-ზე CPU-ს 80%-იანი გადატვირთვა 5 წუთის განმავლობაში.

Memory Hog: მეხსიერების ხელოვნური სტრესი (500MB), რამაც გამოიწვია OOMKilled frontend-ში.

Network Latency: 500ms-იანი დაგვიანება recommendationservice-ის მოთხოვნებზე.

Packet Loss: 10%-იანი პაკეტის დაკარგვა cartservice-ისთვის.

თითოეული ექსპერიმენტი გაგრძელდა 10 წუთი, 3 გამეორებით სტატისტიკური სიზუსტისთვის.

ოპტიმიზაცია:

HPA-ს დანერგვა: Autoscaling CPU-ს 70%-იან ზღვარზე, მინიმუმ 2 პოდი, მაქსიმუმ 5.

რესურსების კორექტირება: frontend-ის მეხსიერების ლიმიტი გაიზარდა 1GB-მდე.

Timeout/Retry მექანიზმები: recommendationservice-ის timeout გაიზარდა 1 წმ-მდე, დაემატა 3 retry მცდელობა.

ვალიდაცია:

განმეორებითი ექსპერიმენტები ოპტიმიზირებულ სისტემაზე KPI-ების ცვლილებების შესაფასებლად.

მონაცემთა ანალიზი ეფუძნებოდა რაოდენობრივ (KPI-ების ცვლილებები) და თვისებრივ (სისტემის ქცევის კონტექსტუალური შეფასება) მიდგომებს. Istio გამოყენებული იყო ქსელის მონიტორინგისთვის, ხოლო Locust-მა უზრუნველყო სტაბილური დატვირთვა.

3. მსჯელობა.

კვლევის შედეგებმა გამოავლინა, რომ „Online Boutique“-ის არაოპტიმიზირებული ვერსია მოწყვლადი იყო სხვადასხვა ტიპის ხარვეზების მიმართ. „ჩუმი მარცხის“ ფენომენი,

განსაკუთრებით Memory Hog ექსპერიმენტში, აჩვენა, რომ frontend მალავდა შეცდომებს ქეშირებული ან არასრული ინფორმაციის მიწოდებით, რაც სტანდარტული მონიტორინგის მეტრიკებით (მაგ., Error Rate) არ ჩანდა. ეს ხაზს უსვამს მაღალი დონის SLI-ების (Service Level Indicators) მნიშვნელობას, რომლებიც ზომავს, მაგ., timeout-ების სიხშირეს ან retry მცდელობებს..NET-ის Garbage Collector-ის გავლენა cartservice-ზე Memory Hog ექსპერიმენტებში გამოავლინა, რომ ტექნოლოგიური სტეკის სპეციფიკა გადამწყვეტია ქაოსის ექსპერიმენტების დაგეგმვისას. Garbage Collector-მა გაანეიტრალა მეხსიერების სტრესი, რაც მიუთითებს, რომ ხარვეზის ტიპი უნდა იყოს მორგებული აპლიკაციის შიდა მექანიზმებზე. HPA-ს შეზღუდვები ნაჩვენები იყო CPU Hog ექსპერიმენტში, სადაც არაადეკვატური სკალირებამ გამოიწვია პასუხის დროის მკვეთრი ზრდა (200ms-მდე). HPA-ს დანერგვისა და რესურსების კორექტირების შემდეგ, სისტემამ აჩვენა სტაბილური ქცევა (~60ms) სტრესის ქვეშ. Network Latency ექსპერიმენტმა გამოავლინა recommendationservice-ის timeout-ების ნაკლოვანებები, რამაც გამოიწვია მომხმარებლის გამოცდილების (UX) დეგრადაცია. Retry მექანიზმების დამატებამ ეს პრობლემა მნიშვნელოვნად შეამსუბუქა.

დამოკიდებულებების განსხვავებული მდგრადობა გამოვლინდა Packet Loss ექსპე-რიმენტში. კრიტიკული სერვისის (productcatalogservice) ხარვეზმა გამოიწვია მყისიერი დეგრადაცია, ხოლო არაკრიტიკული recommendationservice-ის ხარვეზმა გააქტიურა Circuit Breaker შაბლონი, რაც სისტემის თვითდაცვის მექანიზმის ნიშანია. ეს აჩვენებს, რომ ქაოსის ინჟინერია არა მხოლოდ სისუსტეებს, არამედ სისტემის ძლიერ მხარეებსაც ავლენს. შეზღუდვები: კვლევა შემოიფარგლა ერთი აპლიკაციით („Online Boutique“) და GKE გარემოთი, რაც ზღუდავს შედეგების განზოგადებას. მონაცემთა შრის (მაგ., Redis, SQL) ან გარე სერვისების (მაგ., API-ების) ხარვეზები არ განხილულა. სამომავლო კვლევებმა უნდა შეისწავლოს: მონაცემთა შრის ხარვეზები: მონაცემთა ბაზის მიუწვდომლობის გავლენა.
უსაფრთხოების ქაოსის ინჟინერია: RBAC ან DDoS სცენარების ტესტირება.
Service Mesh: Istio-ს მდგრადობის პოლიტიკების ანალიზი.

4. შედეგები:

საბაზისო ანალიზი:
პასუხის დრო: ~50ms frontend-ისთვის (500 მომხმარებელი, 10 RPS).
შეცდომების სიხშირე: <0.1%.
CPU/მეხსიერება: ნაგულისხმევი ლიმიტების ფარგლებში (500MB frontend-ისთვის).
ქაოსის ექსპერიმენტები:
CPU Hog:
CPU-ს 80%-იანი გადატვირთვა გამოიწვია პასუხის დროის ზრდა 200ms-მდე.
გამოვლინდა HPA-ს არარსებობის გამო სკალირების ნაკლებობა.
Memory Hog:
500MB მეხსიერების სტრესმა გამოიწვია OOMKilled frontend-ში.
„ჩუმი მარცხი“: სისტემამ მიაწოდა ქეშირებული, არასრული ინფორმაცია.
Network Latency:
500ms-იანი დაგვიანება recommendationservice-ზე გამოიწვია UX-ის დეგრადაცია.
Timeout-ების ნაკლებობამ გააუარესა შეცდომების მართვა.
Packet Loss:
10%-იანმა პაკეტის დაკარგვამ გამოიწვია cartservice-ის კასკადური ხარვეზები.

Circuit Breaker-მა დაიცვა სისტემა არაკრიტიკული სერვისებისთვის.
ოპტიმიზაციის შედეგები:
HPA-ს დანერგვა: პასუხის დრო სტაბილიზირდა ~60ms-ზე სტრესის ქვეშ, CPU-ს 70%-იან ზღვარზე autoscaling-ით.
რესურსების კორექტირება: frontend-ის მეხსიერების ლიმიტის გაზრდამ (1GB) აღმოფხვრა OOMKilled.
Timeout/Retry გაუმჯობესება: recommendationservice-ის timeout-ის (1 წმ) და retry-ის (3 მცდელობა) დამატებამ გაზარდა სისტემის სტაბილურობა.
შეცდომების მეტრიკები: ოპტიმიზაციის შემდეგ Error Rate გახდა უფრო ადეკვატური, რაც ასახავდა სისტემის რეალურ მდგომარეობას.

5. დასკვნა.

ქაოსის ინჟინერიის გამოყენებამ „Online Boutique“-ის მაგალითზე ნათლად აჩვენა, რომ ეს მეთოდი ეფექტურად ავლენს ფარულ სისუსტეებს, რომლებიც ტრადიციული მონიტორინგით შეუმჩნეველი რჩება. გამოვლენილი პრობლემები, როგორიცაა „ჩუმი მარცხი“, HPA-ს შეზღუდვები და timeout-ების ნაკლებობა, მიზნობრივი ოპტიმიზაციებით (HPA, რესურსების კორექტირება, timeout/retry) გამოსწორდა, რამაც გამოიწვია სისტემის მდგრადობის სტატისტიკურად მნიშვნელოვანი გაუმჯობესება.

პრაქტიკული რეკომენდაციები:

ქაოსის ინჟინერიის ინტეგრირება SDLC-ში: რეგულარული ექსპერიმენტები სატესტო და წარმოების გარემოში.
რესურსების მართვა: ემპირიულად განსაზღვრული requests/limit პარამეტრები Kubernetes-ში.
SLI-ების გაღრმავება: Timeout-ების, retry-ებისა და Circuit Breaker-ის მონიტორინგი.
ტექნოლოგიური სტეკის გათვალისწინება: ხარვეზის ტიპის მორგება აპლიკაციის შიდა მექანიზმებზე (მაგ., .NET Garbage Collector).

სამომავლო კვლევა:

მონაცემთა შრის (Redis, SQL) ხარვეზების ანალიზი.
უსაფრთხოების ქაოსის ინჟინერია (მაგ., DDoS, RBAC).
Service Mesh-ის (Istio) მდგრადობის პოლიტიკების შესწავლა.

ქაოსის ინჟინერია ადასტურებს თავის ღირებულებას, როგორც DevOps-ის განუყოფელი ნაწილი, და ხელს უწყობს უფრო სანდო და პროგნოზირებადი ღრუბლოვანი სისტემების შექმნას.

გამოყენებული ლიტერატურა:

1. დემჩენკო ნ. (2025). DevOps-ის ეფექტიანობის ოპტიმიზაცია ქაოსური ინჟინერიის გამოყენებით, ბიზნესისა და ტექნოლოგიების უნივერსიტეტი. 80 გვ.
2. Basiri, A., Behnam, N., de Rooij, R., Hochstein, L., Kosewski, L., Reynolds, J., & Rosenthal, C. (2016). Chaos Engineering: Building Confidence in System Behavior through Controlled Experiments. Netflix Technology Blog.
3. Rosenthal, C., & Jones, N. (2020). Chaos Engineering: System Resiliency in Practice. O’Reilly Media. 267 p.
4. Zhang, X., Liu, Y., & Wang, J. (2022). Performance Optimization in Kubernetes-Based Microservices. Journal of Cloud Computing, 11(3), 45–56p.

The Use of Chaos Engineering to Enhance the Resilience of Microservice Architectures: A Case Study of "Online Boutique"

Giorgi Kuchava, Ioseb Kartvelishvili, Shalva Vashalomidze

Georgian Technical University

kuchavagiorgi08@gtu.ge, s.kartvelishvili@gtu.ge, vashalomidze.shalva24@gtu.ge

Abstract

Modern cloud-native microservice architectures, despite their flexibility and scalability, often face complex challenges such as latent failures, cascading errors, and unpredictable system behavior. This article explores the application of chaos engineering, a proactive DevOps method, using the example of Google's "Online Boutique" application deployed on Google Kubernetes Engine (GKE). Experiments conducted with Chaos Mesh (CPU Hog, Memory Hog, Network Latency, Packet Loss) revealed critical vulnerabilities, including "silent failures," memory exhaustion (OOMKilled), and inadequate timeout/retry mechanisms. Using KPIs measured through Prometheus and Grafana (response time, error rate, CPU/memory usage) and simulated user traffic via Locust, system optimization was achieved. This included implementing the Horizontal Pod Autoscaler (HPA), adjusting resource limits, and enhancing timeout/retry mechanisms. Repeated experiments demonstrated statistically significant improvements in system resilience, underscoring the importance of chaos engineering as an integral part of DevOps. The article provides practical recommendations for enhancing system reliability and integrating chaos engineering into the SDLC.

Keywords: Chaos Engineering, Microservices, Kubernetes, DevOps, Prometheus, Grafana.