ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ
ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάλυση Εφαρμογών Υπολογιστικού Νέφους με Εργαλεία Κατανεμημένης Παρακολούθησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΙΣΤΙΝΑ-ΜΑΡΙΑ Π. ΑΝΔΡΩΝΑ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

Εθνικο Μετσοβιο Πολυτεχνειο

Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων

Τομεας Επικοινωνιων, Ηλεκτρονικης
Και Συστηματων Πληροφορικης

Ανάλυση Εφαρμογών Υπολογιστικού Νέφους με Εργαλεία Κατανεμημένης Παρακολούθησης

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΧΡΙΣΤΙΝΑ-ΜΑΡΙΑ Π. ΑΝΔΡΩΝΑ

Επιβλέπων : Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 8η Οκτωβρίου 2021.

....	....	....
Συμεών Παπαβασιλείου	Θεοδώρα Βαρβαρίγου	Ιωάννα Ρουσσάκη
Καθηγητής Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.	Επ. Καθηγήτρια Ε.Μ.Π.

Αθήνα, Οκτώβριος 2021

....
Χριστίνα-Μαρία Π. Ανδρωνά

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Χριστίνα-Μαρία Ανδρωνά, 2021.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ανάπτυξη εφαρμογών υπολογιστικού νέφους αποτελεί μια νέα τάση που υιοθετείται όλο και περισσότερο τα τελευταία χρόνια, αφήνοντας πίσω το μονολιθικό παραδοσιακό χαρακτήρα ανάπτυξης εφαρμογών. Τόσο το υπολογιστικό νέφος, με τα εγγενή χαρακτηριστικά του, όσο και η αρχιτεκτονική των εφαρμογών υπολογιστικού νέφους, η οποία βασίζεται σε μικροϋπηρεσίες, προσφέρουν πολλά οφέλη στις επιχειρήσεις που τα υιοθετούν. Εντούτοις, υφίστανται δυσκολίες στην παρακολούθηση των εφαρμογών αυτών, λόγω της πολυπλοκότητας και της κατανεμημένης πλέον δομής τους. Σε αυτό το πλαίσιο, έχουν αναπτυχθεί τεχνικές κατανεμημένης παρακολούθησης και ιχνηλάτησης, οι οποίες παρέχουν ορατότητα στην εσωτερική δομή και συμπεριφορά κατανεμημένων εφαρμογών που αποτελούνται από μικροϋπηρεσίες. Στην παρούσα διπλωματική εργασία, αξιοποιώντας αυτές τις μεθόδους και τεχνικές, προτείνεται η συγκέντρωση όλων των διαθέσιμων δεδομένων και μετρικών από την παρακολούθηση μιας κατανεμημένης εφαρμογής και η ενοποίησή τους σε ένα κοινό σχήμα δεδομένων. Στόχος είναι η τελική ομογενοποιημένη πληροφορία να προσφέρει μια ολοκληρωμένη εικόνα για τα κατανεμημένα συστήματα, με προηγμένες δυνατότητες ανάλυσης και ερμηνείας των επιδόσεών τους. Η προτεινόμενη αρχιτεκτονική κατανεμημένης παρακολούθησης εφαρμογών υπολογιστικού νέφους αξιολογείται με βάση μια σειρά από δοκιμές σε πειραματική υποδομή. Τα αποτελέσματα που παρουσιάζονται επιβεβαιώνουν την καταλληλότητα της προτεινόμενης λύσης για την αξιόπιστη και αποδοτική ιχνηλάτηση κατανεμημένων εφαρμογών υπολογιστικού νέφους.

Λέξεις κλειδιά

Κατανεμημένη παρακολούθηση, ιχνηλάτηση, εφαρμογές υπολογιστικού νέφους, μικροϋπηρεσίες, κατανεμημένα συστήματα, ενορχήστρωση

Abstract

Cloud-native applications development is a new trend that is adopted more and more lately, leaving behind the monolithic traditional character of developing applications. Cloud computing, with its inherent characteristics, as well as the microservices-based architecture of cloud-native applications, offer many benefits to the companies that adopt them. However, there are challenges and difficulties in understanding and observing this kind of applications, due to their complexity and distributed architecture. In this context, distributed tracing techniques have been developed to offer visibility into the internal structure and operation of a distributed software system consisting of microservices. In this thesis, by using distributed tracing methods and techniques, it is proposed to collect all the available trace data and metrics from a distributed tracing deployment and unify them in an aggregated data schema. Such data aggregation and homogenization aims to offer a better understanding of the distributed systems, with advanced analysis and performance profiling capabilities. The proposed architecture of distributed tracing in cloud-native applications is tested on a series of trials in an experimental infrastructure. The presented results confirm the suitability of the proposed solution for reliable and efficient tracing in cloud-native distributed applications.

Key words

Distributed tracing, monitoring, cloud-native applications, microservices, distributed systems, orchestration

Ευχαριστίες

Θα ήθελα να ευχαριστήσω τον καθηγητή κ. Συμεών Παπαβασιλείου για την καθοδήγησή του και την ευκαιρία που μου έδωσε με την ανάθεση αυτής της διπλωματικής εργασίας να ασχοληθώ με ένα τόσο ενδιαφέρον και σύγχρονο ερευνητικό πεδίο.
Επίσης, θα ήθελα να ευχαριστήσω θερμά τον μεταδιδάκτορα ερευνητή Αναστάσιο Ζαφειρόπουλο και την υποψήφια διδάκτορα Ελένη Φωτοπούλου για την εξαιρετική συνεργασία, την πολύτιμη βοήθειά τους και τη συνεχή υποστήριξη που μου παρείχαν κατά την εκπόνηση της διπλωματικής μου εργασίας.
Τέλος, ευχαριστώ ιδιαίτερα τους γονείς μου και την αδερφή μου για την αγάπη, την ενθάρρυνση και τη στήριξή τους όλα αυτά τα χρόνια, αλλά και τους φίλους μου που ήταν πάντα δίπλα μου.

Ανδρωνά Χριστίνα-Μαρία,
Αθήνα, Οκτώβριος 2021

Περιεχόμενα

Περίληψη 5

Abstract 7

Ευχαριστίες 9

Περιεχόμενα 10

Κεφάλαιο 1: Εισαγωγή 12
1.1 Τάση για ανάπτυξη cloud-native εφαρμογών 12
1.1.1 Ορισμός cloud-native εφαρμογών 12
1.1.2 Δομή cloud-native εφαρμογών 12
1.1.3 Πλεονεκτήματα τεχνικών ανάπτυξης εφαρμογών με βάση μικροϋπηρεσίες και containers 13
1.1.4 Δημιουργία και ενορχήστρωση εφαρμογών που εκτελούνται σε containers .14
1.2 Αξιοποίηση τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης) για cloud-native εφαρμογές 15
1.3 Βασικές προκλήσεις για την ανάπτυξη αποδοτικών τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης) 16
1.4 Συνεισφορά της εργασίας 16
1.5 Δομή της εργασίας 16

Κεφάλαιο 2: Ορισμοί και βασικές έννοιες τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης) 18
2.1 Ορισμοί για trace, span, tracer 18
2.2 Παραδείγματα με βάση μια πιλοτική εφαρμογή 21
2.3 Ορισμός των exemplars 22
2.3.1 Open Metrics και δομή των exemplars 22

Κεφάλαιο 3: Υφιστάμενα εργαλεία και τεχνικές κατανεμημένης παρακολούθησης (ιχνηλάτησης) εφαρμογών υπολογιστικού νέφους 24
3.1 Επισκόπηση τεχνικών κατανεμημένης παρακολούθησης 24
3.2 Υφιστάμενα εργαλεία για κατανεμημένη παρακολούθηση 25
3.2.1 Αρχιτεκτονική Zipkin 29
3.2.2 Αρχιτεκτονική Jaeger 30
3.2.3 Spring Cloud Sleuth 30
3.2.4 Προτυποποιημένο API Διαχείρισης Εφαρμογών (Standardized Instrumentation API) 30
3.3 Υποστήριξη κατανεμημένης παρακολούθησης από εργαλεία ενορχήστρωσης και παρακολούθησης 32

Κεφάλαιο 4: Προτεινόμενη Αρχιτεκτονική 35
4.1 Περιγραφή της προτεινόμενης λύσης 35
4.1.1 Πιλοτική υλοποίηση 35
4.1.2 Κατανεμημένη παρακολούθηση εφαρμογής 35

4.2 Αναπαράσταση δεδομένων 39
4.2.1 Zipkin 39
4.2.2 Prometheus 44
4.3 Ενοποίηση δεδομένων 47

Κεφάλαιο 5: Υποδομή δοκιμών και αποτελέσματα 49
5.1 Υποδομή υπολογιστικού νέφους 49
5.2 Φορτίο Δοκιμών 55
5.3 Αποτελέσματα 57
5.3.1 Αποτελέσματα εργαλείων παρακολούθησης 57
5.3.2 Ανάλυση δεδομένων 61

Κεφάλαιο 6: Συμπεράσματα 70

Αποθετήριο Κώδικα 71

Παράρτημα 72

Γλωσσάρι 75

Βιβλιογραφία 77

Κεφάλαιο 1: Εισαγωγή

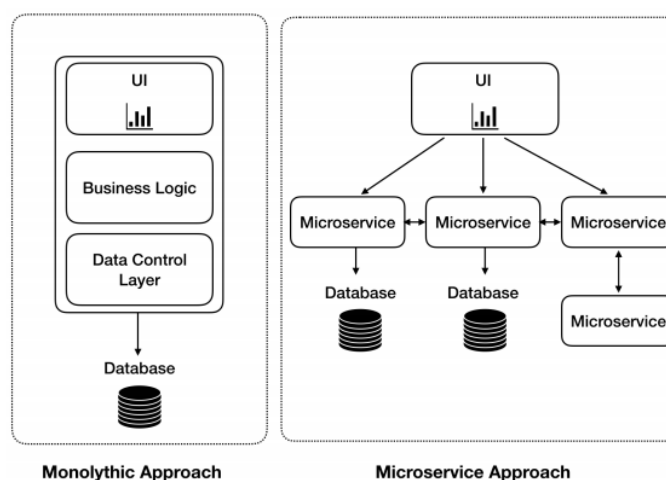
1.1 Τάση για ανάπτυξη cloud-native εφαρμογών

1.1.1 Ορισμός cloud-native εφαρμογών

Οι εγγενείς εφαρμογές υπολογιστικού νέφους (*cloud-native applications*) είναι εφαρμογές που έχουν αναπτυχθεί ειδικά για να τρέχουν σε προγραμματιζόμενες υποδομές υπολογιστικού νέφους (cloud computing). Αυτό σημαίνει ότι έχουν ως στόχο και μπορούν να αξιοποιήσουν τα εγγενή χαρακτηριστικά του [1], όπως την κατανεμημένη διαθεσιμότητα πόρων, την αξιοπιστία που προσφέρει, τη δυναμικότητα (*agility* - ικανότητα για γρήγορη ανάπτυξη και δοκιμή νέου λογισμικού, και την ανθεκτικότητα (*resilience* - αποδοχή των πιθανών αλλαγών, σφαλμάτων και αποτυχιών και αντιμετώπισή τους λόγω της δυναμικής φύσης). Έτσι οι επιχειρήσεις που έχουν στραφεί στο cloud μπορούν γρήγορα να ανταποκρίνονται στις ολοένα και αυξανόμενες απαιτήσεις χρηστών και να παρέχουν καινοτόμα χαρακτηριστικά και αναβαθμίσεις στις υπηρεσίες τους άμεσα [2].

1.1.2 Δομή cloud-native εφαρμογών

Αυτού του είδους οι εφαρμογές είναι κατανεμημένες καθώς αποτελούνται από πολλές μικροϋπηρεσίες (*microservices*), έτσι ώστε να υπηρετούν το σκοπό τους για υψηλές επιδόσεις και κλιμακωσιμότητα. Η αρχιτεκτονική των μικροϋπηρεσιών, όπως φαίνεται στην [Εικόνα 1], αποσυνθέτει την εφαρμογή στα μικρότερα, αυτοτελή και ειδικού σκοπού τμήματά της (μικροϋπηρεσίες) με το δικό τους αποθηκευτικό χώρο, τα οποία όμως επικοινωνούν μεταξύ τους. Η επικοινωνία τους πραγματοποιείται με διάφορα πρωτόκολλα και μεθόδους μεταφοράς όπως HTTP REST API , gRPC (Remote procedure Calls), Apache Thrift, SOAP κ.α. [3,9].

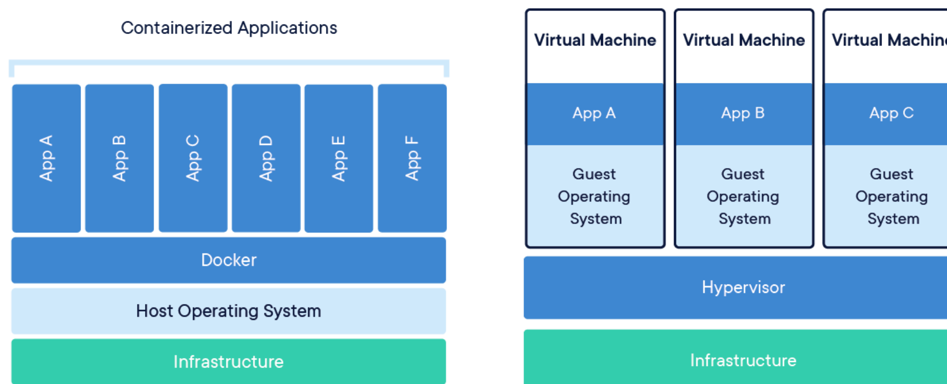


Εικόνα 1: Σύγκριση μονολιθικών εφαρμογών και εφαρμογών βασισμένων σε μικροϋπηρεσίες[1]

[1] https://elib.uni-stuttgart.de/bitstream/11682/9844/1/Application%20Performance%20Monitoring%20In%20Microservice-Based%20Systems_ValentinSeifermann.pdf

1.1.3 Πλεονεκτήματα τεχνικών ανάπτυξης εφαρμογών με βάση μικροϋπηρεσίες και containers

Κάθε μικροϋπηρεσία, αναπτύσσεται αυτόνομα με τη γλώσσα και το προγραμματιστικό πλαίσιο (framework) που ταιριάζει καλύτερα στη λειτουργικότητά της, πακετάρεται σε ένα *container* με τις εξαρτήσεις (dependencies), τον κώδικα και τα δεδομένα της και απομονώνεται σε αυτό με όλο το περιβάλλον της (runtime environment). Με αυτό τον τρόπο παρέχεται φορητότητα (portability) και συνέπεια στην εκτέλεσή του σε διαφορετικά περιβάλλοντα (υπολογιστές, παραδοσιακές υποδομές ή υποδομές υπολογιστικού νέφους).
Τα containers μπορούν, χάρη στην εικονοποίηση, να μοιράζονται τους πόρους μιας υποδομής που τα φιλοξενεί αλλά και το ίδιο λειτουργικό σύστημα, για αυτό και καταλαμβάνουν μικρότερο αποθηκευτικό χώρο σε σύγκριση με τις εικονικές μηχανές (virtual machines) [Εικόνα 2]. Έτσι πολλά containers μπορούν να τρέχουν ταυτόχρονα στον ίδιο φιλοξενητή (host), βελτιώνοντας τη χρήση και την κατανομή πόρων.



Εικόνα 2: Σύγκριση εκτέλεσης containers και εικονικών μηχανών σε μια υποδομή[2]

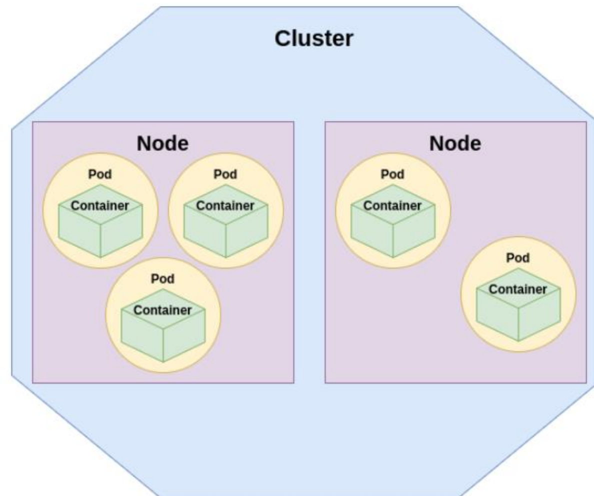
Το μικρό μέγεθος και οι λίγες απαιτήσεις των containers (μικρό overhead) στηρίζουν τη δυνατότητα γρήγορης ανάπτυξης και δοκιμής νέου λογισμικού της εφαρμογής.
Από την άλλη, η απομόνωση που παρέχουν προσφέρει αυτονομία στις μικροϋπηρεσίες και επιτρέπει την ανεξάρτητη ανάπτυξη και εκτέλεσή (deployment) τους, ακόμα και σε ξεχωριστούς εξυπηρετητές-διακομιστές (servers). Η προσέγγιση που ακολουθείται είναι χαλαρής σύζευξης (loose-coupling), όπου η κάθε μικροϋπηρεσία / τμήμα της εφαρμογής έχει ελάχιστη γνώση και εξάρτηση από τα υπόλοιπα τμήματα [2].
Ένα ακόμα βασικό πλεονέκτημα των κατανεμημένων εφαρμογών που αποτελούνται από μικροϋπηρεσίες είναι ότι οι αλλαγές στον κώδικα και οι ενημερώσεις της εφαρμογής μπορούν να γίνουν πολύ εύκολα και γρήγορα, χωρίς να προκαλούν περιόδους όπου το σύστημα δε θα είναι διαθέσιμο (downtime) ή να επηρεάζουν τη λειτουργία ολόκληρης της εφαρμογής. Μέθοδοι συνεχούς ενσωμάτωσης-συνεχούς παράδοσης (*Continuous integration and continuous delivery (CI/CD) methods*) μπορούν να υιοθετηθούν από ομάδες διαχείρισης της προγραμματιζόμενης υποδομής (DevOps), διασφαλίζοντας γρήγορο έλεγχο και προώθηση στην παραγωγή, ώστε η εφαρμογή να μείνει ενημερωμένη σύμφωνα με τις τελευταίες αλλαγές [4].
Επιπρόσθετα, η αυτονομία της κάθε μικροϋπηρεσίας επιτρέπει την ανεξάρτητη κλιμάκωσή της. Αυτό σημαίνει ότι η εφαρμογή μπορεί να προσαρμοστεί δυναμικά στην

[2] https://www.docker.com/resources/what-container

εναλλασσόμενη ζήτηση και σε δυναμικά φορτία με δυνατότητα οριζόντιας κλιμάκωσης των μικροϋπηρεσίων. Εκμεταλλευόμενες την ελαστικότητα των υποδομών υπολογιστικού νέφους στο οποίο αναπτύσσονται, οι cloud-native εφαρμογές επιτυγχάνουν ευκολότερη, γρηγορότερη και οικονομικά αποδοτικότερη κλιμάκωση [47] των τμημάτων τους.

1.1.4 Δημιουργία και ενορχήστρωση εφαρμογών που εκτελούνται σε containers

Για τη δημιουργία των μικροϋπηρεσιών σε containers ένα από τα πιο διάσημα εργαλεία που χρησιμοποιούνται είναι το Docker. Στην πραγματικότητα ο κώδικας, οι εξαρτήσεις και το περιβάλλον της κάθε μικροϋπηρεσίας πακετάρονται σε ένα δυαδικό αρχείο που ονομάζεται εικόνα του container (*container image*), η οποία είναι διαθέσιμη και αποθηκεύεται σε ένα αποθετήριο εφαρμογών (registry). Το Docker διαθέτει το δικό του αποθετήριο, το «Docker Hub». Τα containers με τη σειρά τους δημιουργούνται τρέχοντας την εικόνα του container για την υπηρεσία που θα υλοποιήσουν [5].
Η χειροκίνητη κλιμάκωση των κατανεμημένων εφαρμογών είναι πολύ απαιτητική και δύσκολη. Αυτό οφείλεται στο μεγάλο πλήθος υποτμημάτων της εφαρμογής (μικροϋπηρεσίες) αλλά και σε παράγοντες που πρέπει να ληφθούν υπόψιν, όπως η εξισορρόπηση φορτίου (load balancing), η ορθή κατανομή πόρων και η ασφαλής λειτουργία των containers. Τα εργαλεία ενορχήστρωσης λύνουν το πρόβλημα αυτοματοποιώντας αυτές τις διαδικασίες και φροντίζοντας για τη σωστή εκτέλεση και τη διαθεσιμότητα των εφαρμογών. Το πιο διαδεδομένο και ευρέως χρησιμοποιούμενο εργαλείο ενορχήστρωσης είναι το Kubernetes (K8S) [6], το οποίο είναι ένα σύστημα ανοιχτού κώδικα με στόχο να ομαδοποιεί τα κατανεμημένα τμήματα της εφαρμογής, να τα εκτελεί και να τα διαχειρίζεται αυτόματα, με δυνατότητα κλιμάκωσής τους ανάλογα με την ανάγκη τους σε πόρους. Βασίζεται σε master-slave αρχιτεκτονική, με τα κύρια τμήματα (*master-components*) να διαχειρίζονται την κατάσταση του *cluster* (συστάδας υπολογιστών - εικονικών ή πραγματικών συνδεδεμένων μέσω δικτύου), το οποίο αποτελείται από worker κόμβους (*worker-nodes*) [46]. Ο worker κόμβος (worker-node) είναι μια υπολογιστική μονάδα με πόρους προς κατανάλωση (CPU, RAM), η οποία συνιστά δομικό στοιχείο της συστάδας (cluster) στην οποία στήνεται μια cloud-native κατανεμημένη εφαρμογή [7]. Οι κόμβοι είναι είτε φυσικά μηχανήματα (εξυπηρετητές) σε κέντρα φιλοξενίας υποδομής (data centers) είτε εικονικές μηχανές σε παρόχους υπολογιστικής υποδομής (cloud) και στον καθένα μπορεί να τρέχουν ένα ή περισσότερα *Pods*. Το Pod είναι η μικρότερη υπολογιστική μονάδα που μπορεί να δημιουργηθεί και να διαχειριστεί από το Kubernetes, η οποία εκτελεί ένα ή περισσότερα containers που θα μοιράζονται τον ίδιο αποθηκευτικό χώρο και δίκτυο. Σε κάθε Pod ιδανικά τρέχει ένα container, ώστε να απομονώνονται τα τμήματα της εφαρμογής [48]. Μέσω των Pods πραγματοποιείται η κλιμακωσιμότητα, καθώς μπορούν να πολλαπλασιαστούν με αντίγραφα όταν το Kubernetes, ως ενορχηστρωτής, το κρίνει απαραίτητο [Εικόνα 3]. Για την εκτέλεση των Pods υπάρχει ένα υψηλότερο επίπεδο, το *Deployment* (εκτελεστής). Το Deployment καθορίζει την εικόνα των containers που θα τρέξουν τα Pods αλλά και το πλήθος των αντιγράφων τους. Φροντίζει μάλιστα για την επαναδημιουργία τους σε περίπτωση που κάποιο από αυτά αποτύχει και τερματίσει. Έτσι επιτυγχάνεται μια αυτοματοποίηση στη διαχείριση των Pods και διασφάλιση της επιθυμητής κατάστασης του συστήματος από τον ίδιο τον ενορχηστρωτή.



Εικόνα 3: Δομή ενός kubernetes cluster με κόμβους, pods και containers

1.2 Αξιοποίηση τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης) για cloud-native εφαρμογές

Η κατανεμημένη μορφή των cloud-native εφαρμογών, πέρα από τα πολλά πλεονεκτήματα, φέρνει και αρκετή περιπλοκότητα. Οι μικροϋπηρεσίες μιας εφαρμογής μπορεί να είναι διασκορπισμένες σε διαφορετικές πλατφόρμες, κέντρα φιλοξενίας υποδομής (data centers) και ομάδες, δυσκολεύοντας έτσι την ορατότητα στο ίδιο το σύστημα. Πιθανά λάθη ή αποτυχίες δεν είναι εύκολο να εντοπιστούν, ούτε να ελεγχθούν αν δεν διαχειρίζονται όλα από την ίδια ομάδα. Για να διασφαλιστεί η σταθερότητα, η απόδοση και η αξιοπιστία του συστήματος, η παρακολούθησή του είναι μεγίστης σημασίας.

Τη λύση σε αυτά τα προβλήματα δίνουν *τεχνικές κατανεμημένης παρακολούθησης και ιχνηλάτησης* (*distributed tracing*), με τις οποίες μπορούμε να ερμηνεύσουμε τη συμπεριφορά ενός κατανεμημένου συστήματος και να εντοπίσουμε πού συμβαίνουν σφάλματα και συμφορήσεις (bottlenecks) σε αυτό, έχοντας πρόσβαση στην εσωτερική δομή του [8]. Με την κατανεμημένη παρακολούθηση καταγράφεται η ακολουθία συναλλαγών και διεργασιών που λαμβάνουν χώρα στην εφαρμογή, με αποτέλεσμα να κατανοείται ο ρόλος της κάθε ξεχωριστής μικροϋπηρεσίας και να παρακολουθείται η απόδοση και η κατάστασή της. Η κίνηση που εισέρχεται στο σύστημα μπορεί να καταγραφεί από άκρο σε άκρο (end-to-end) όπως και η πρόοδος του κάθε αιτήματος καθώς περνάει από τις διάφορες μικροϋπηρεσίες [9] . Μετρώντας το χρόνο που απαιτεί η κάθε μικροϋπηρεσία για να εξυπηρετήσει το κάθε αίτημα (latency), μπορούμε να ερμηνεύσουμε τη συνολική απόδοση της εφαρμογής και να εντοπίσουμε τα τμήματα που χρήζουν βελτίωσης. Έτσι υποβοηθάται η ενορχήστρωση των κατανεμημένων συστημάτων, ώστε να επιδιορθωθούν τυχόν αποτυχίες στις μικροϋπηρεσίες ή να κλιμακωθούν όπου κρίνεται απαραίτητο με βάση τις πληροφορίες από την ιχνηλάτηση.

1.3 Βασικές προκλήσεις για την ανάπτυξη αποδοτικών τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης)

Μια εφαρμογή που προγραμματίζεται κατάλληλα ώστε να παρακολουθηθεί από ένα σύστημα διαχείρισης (*instrumentation*), μπορεί να παράγει ίχνη (*traces*) για το κάθε αίτημα που δέχεται. Το εκάστοτε ίχνος (trace) μας δίνει χρήσιμη πληροφορία για το αίτημα που αναπαριστά και για την πορεία που ακολούθησε από άκρο σε άκρο, μαζί με το χρόνο που αφιέρωσε σε κάθε μικροϋπηρεσία της εφαρμογής [9]. Ανάλογα και με τον όγκο των αιτημάτων, παράγεται πολύ μεγάλος αριθμός από ίχνη. Αυτό που έχει ενδιαφέρον και αξία είναι να μπορέσουμε να εντοπίσουμε τα ίχνη που αντιστοιχούν σε αιτήματα που παρουσίασαν ιδιαίτερη συμπεριφορά, υποδεικνύοντας κάποιο σφάλμα ή κάποια συμφόρηση στο σύστημα.
Η παρακολούθηση (*monitoring*) από την άλλη του συστήματος με την καταγραφή χρήσιμων μετρικών (χρόνου απόκρισης, χρήσης cpu, μνήμης, πλήθους και ρυθμού σφαλμάτων) παίζει σημαντικό ρόλο στον έλεγχο της υγείας της εφαρμογής. Με αυτές τις μετρικές μπορούμε να εντοπίσουμε πότε παρουσιάζεται ή αναμένεται αποτυχία, λόγω π.χ. της υπέρβασης ενός κατωφλίου για το χρόνο απόκρισης ή της αύξησης του ρυθμού σφαλμάτων, χωρίς όμως να γνωρίζουμε πού οφείλεται και ποιο τμήμα ή τμήματα του κατανεμημένου συστήματος είναι υπαίτια.

1.4 Συνεισφορά της εργασίας

Παρόλο που υπάρχει δυνατότητα να συλλέξουμε πολλή και χρήσιμη πληροφορία για την εσωτερική κατάσταση του συστήματος, μέσα από μεθόδους παρατηρησιμότητας με καταγραφή μετρικών (monitoring) και ιχνηλάτησης (tracing), απουσιάζει η ενοποίησή της σε ένα ενιαίο σύστημα. Συνδυάζοντας τη γνώση από τις μετρικές για το πότε συμβαίνει και παρατηρείται μια μη φυσιολογική συμπεριφορά, με τα δεδομένα της ιχνηλάτησης για το πού ακριβώς οφείλεται και ποιο τμήμα της εφαρμογής ευθύνεται, είναι εφικτή η πλήρης ορατότητα και γνώση του συστήματος με σκοπό την επιδιόρθωσή του. Στόχος αυτής της εργασίας είναι η ανάπτυξη κατάλληλης τεχνικής κατανεμημένης παρακολούθησης που θα επιτρέπει την ενοποίηση όλης της πληροφορίας, ώστε να μπορούν τα ίχνη (traces) να συνδεθούν με τις αντίστοιχες τιμές μετρικών και να επιλεχθούν τα πιο αντιπροσωπευτικά και χρήσιμα, από τα χιλιάδες που παράγονται, για μελέτη και ερμηνεία της υπό εξέτασης συμπεριφοράς ή ανωμαλίας. Μέσα από τη συσχέτιση των επιμέρους πληροφοριών, δημιουργείται μια εικόνα για το τι συμβαίνει ακριβώς μέσα στο σύστημα μια συγκεκριμένη χρονική στιγμή και ποιοι παράγοντες επηρεάζουν μια κατάσταση.

1.5 Δομή της εργασίας

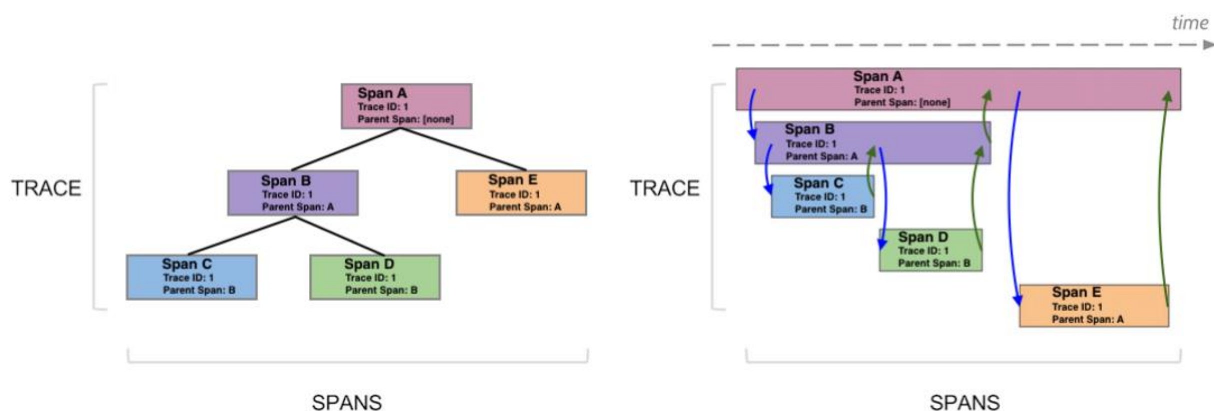
Στο Κεφάλαιο 2 της παρούσας εργασίας παρατίθενται και ερμηνεύονται όροι, έννοιες και γενικότερα βασικές αρχές της τεχνολογίας της κατανεμημένης παρακολούθησης, με χρήση επεξηγηματικών παραδειγμάτων. Στη συνέχεια, στο Κεφάλαιο 3 δίνεται μια γενική εικόνα του χώρου κατανεμημένης παρακολούθησης με επισκόπηση των τεχνικών που χρησιμοποιούνται σε εφαρμογές υπολογιστικού νέφους. Παρατίθενται και συγκρίνονται επίσης κάποια από τα σημαντικότερα διαθέσιμα εργαλεία ανοιχτού κώδικα που

χρησιμοποιούνται για ιχνηλάτηση εφαρμογών καθώς και εργαλεία ενορχήστρωσης και παρακολούθησης που μπορούν να την υποβοηθήσουν. Έχοντας αυτό το υπόβαθρο, στο Κεφάλαιο 4 παρουσιάζεται η προτεινόμενη λύση για ενοποίηση δεδομένων που συλλέγονται από την κατανεμημένη παρακολούθηση και αναπτύσσεται πιλοτική εφαρμογή. Αφού παρουσιαστούν οι δομές των δεδομένων που συλλέγονται από τα εργαλεία κατανεμημένης παρακολούθησης που θα χρησιμοποιηθούν, προτείνεται ένα ενιαίο σχήμα ενοποίησής τους. Η πιλοτική εφαρμογή που αναπτύχθηκε θα εκτελεστεί σε υποδομή υπολογιστικού νέφους, όπως αυτή περιγράφεται στο Κεφάλαιο 5, μαζί με τα κατανεμημένα εργαλεία που θα χρησιμοποιηθούν για την παρακολούθησή της. Για τη συλλογή δεδομένων εκτελείται το πειραματικό μέρος της εργασίας και στη συνέχεια, αφού τα δεδομένα αναλυθούν και επεξεργαστούν, παρουσιάζονται τα αποτελέσματα. Στο τελευταίο κεφάλαιο, εξάγονται τα βασικά συμπεράσματα της εργασίας και προτείνονται ιδέες για την εξέλιξή της.

Κεφάλαιο 2: Ορισμοί και βασικές έννοιες τεχνικών κατανεμημένης παρακολούθησης (ιχνηλάτησης)

2.1 Ορισμοί για trace, span, tracer

Με την ιχνηλάτηση, ως τεχνική κατανεμημένης παρακολούθησης, μπορούμε να καταγράψουμε την πορεία που ακολουθούν τα αιτήματα μέσα στο κατανεμημένο σύστημα από άκρη σε άκρη. Αυτό επιτυγχάνεται με τη δημιουργία ενός ίχνους (*trace)*, το οποίο είναι μοναδικό για κάθε αίτημα που δέχεται η εφαρμογή και περιέχει πληροφορίες για την διαδρομή που ακολούθησε μέσα στο κατανεμημένο σύστημα και για τις μικροϋπηρεσίες από τις οποίες πέρασε. Ένα trace είναι μια συλλογή από *spans*, το κάθε ένα από τα οποία αφορά χρονικά προσδιορισμένες διαδικασίες που περιγράφουν μια μονάδα εργασίας που πραγματοποιήθηκε από μια υπηρεσία του συστήματος [10]. Συνεπώς ένα trace περιέχει όλα τα spans, δηλαδή τις διεργασίες που έγιναν στο σύστημα για την εξυπηρέτηση του αιτήματος που αντιπροσωπεύει. Καθώς σε ένα κατανεμημένο σύστημα τα αιτήματα εξυπηρετούνται από τις διάφορες μικροϋπηρεσίες, ένα trace μπορεί να αναπαρασταθεί και ως ένας κατευθυνόμενος άκυκλος γράφος (DAG) από spans [11]. Υπεύθυνος για τη δημιουργία των spans, τη δειγματοληψία και τη διάδοσή τους ανάμεσα στις μικροϋπηρεσίες είναι ο ιχνηλάτης (*tracer*) [9], ο οποίος καταγράφει χρονικά δεδομένα σχετικά με τις διεργασίες που πραγματοποιούνται, για παράδειγμα πότε λήφθηκε ένα αίτημα και πότε στάλθηκε η απάντησή του.
Τα spans που ανήκουν στο ίδιο trace έχουν το ίδιο αναγνωριστικό *TraceID*, ώστε να μπορούν να ομαδοποιηθούν. Ωστόσο το κάθε span έχει και το δικό του αναγνωριστικό *spanID*, αλλά και ένα όνομα, το οποίο αντιπροσωπεύει τη λειτουργία την οποία περιγράφει. Μέσα σε ένα trace τα spans μπορεί να έχουν και σχέσεις μεταξύ τους. Το *root span* είναι το πρώτο span του trace και αντιπροσωπεύει όλη τη συναλλαγή που πραγματοποιήθηκε, καθώς ξεκινάει όταν λαμβάνεται το αίτημα και τελειώνει όταν ολοκληρωθούν όλες οι διαδικασίες. Αυτό το root span έχει ένα ή περισσότερα παιδιά spans (*child spans*), το καθένα από τα οποία μπορεί να έχει αντίστοιχα και τα δικά του εμφωλευμένα spans [10], όπως φαίνεται στην [Εικόνα 4]. Η σχέση γονέα-παιδιού (*parent-child*) στα spans ορίζεται όταν η μία διαδικασία καλεί και εξαρτάται από την άλλη.



Εικόνα 4: Δομή του trace και οι σχέσεις των spans[3]

[3] https://medium.com/nikeengineering/hit-the-ground-running-with-distributed-tracing-core-concepts-ff5ad47c7058

Το πλαίσιο του span (*context span*) περιέχει βασικές πληροφορίες για το κάθε span, οι οποίες θα διαδοθούν στο κάθε παιδί-span, μέσω επικεφαλίδων (headers) κατά την επικοινωνία των μικροϋπηρεσιών. Το context span περιέχει [12] :

- TraceID: Το αναγνωριστικό του trace στο οποίο ανήκει το span. Το TraceID είναι μια τυχαία ακολουθία 16 ή 32 χαρακτήρων.
- SpanID: Το αναγνωριστικό του span. Το spanID είναι τυχαία ακολουθία 16 χαρακτήρων
- ParentID: Το αναγνωριστικό του span που είναι ο γονέας του (parent span). Το span το οποίο δεν έχει parentID, άρα δεν έχει γονέα, είναι το root span.
- Debug: Πληροφορία για το αν το span πρέπει να αποθηκευτεί, ακόμα και αν δεν συμφωνεί με την πολιτική δειγματοληψίας των spans
- Sampled: Πολιτική δειγματοληψίας
- Err: Αν υπάρχει κάποιο σφάλμα

Μέσα στο context span μπορεί επίσης να περιέχονται και επιπλέον πληροφορίες για το span σε μορφή «κλειδιού-τιμής» (key-value), οι οποίες να πρέπει να μεταδοθούν μεταξύ των μικροϋπηρεσιών - spans, και ονομάζονται *baggage*.

```
SpanContext                     :
      traceId     <TraceID>
      id          <spanID>
      parentId    <parentspanID>
      Debug       <bool
      Sampled     <bool>
      Err         error
```

Με αυτές τις πληροφορίες που περιέχει το context span καθορίζεται η ταυτότητα του κάθε span και δημιουργούνται οι σχέσεις γονέα-παιδιού.

Η αναλυτική δομή των spans με βάση τη μορφοποίηση που υποστηρίζεται από το zipkin είναι [12,13]:

- Context span, όπως περιγράφηκε παραπάνω
- Name: Το όνομα της λειτουργίας που περιγράφει (*operation span name*)
- Kind: Το πεδίο αυτό καθορίζει την ερμηνεία των πεδίων timestamp,duration και remoteEndpoint του span, όταν αυτό δεν περιορίζεται σε μια υπηρεσία (service). Αν απουσιάζει τότε το span αφορά σε διαδικασία που πραγματοποιείται τοπικά. Το Kind λαμβάνει τις τιμές :
    - Client
    - Server
- Timestamp: Η χρονική στιγμή έναρξης της λειτουργίας/του span (με ακρίβεια μικρο δευτερολέπτων)
    - Αν το πεδίο `Kind = Client` τότε είναι η στιγμή που το αίτημα στέλνεται προς τον εξυπηρετητή (server)
    - Αν το πεδίο `Kind = Server` τότε είναι η στιγμή που το αίτημα του λήφθηκε από τον εξυπηρετητή (server)
- Duration: Η διάρκεια της λειτουργίας/του span (επίσης με ακρίβεια μικρο δευτερολέπτων)
    - Αν το πεδίο `Kind = Client` τότε είναι ο χρόνος απόκρισης μεχρι να ληφθεί απάντηση ή σφάλμα

- Αν το πεδίο `Kind = Server` τότε είναι ο χρόνος που μεσολάβησε μέχρι να σταλεί απάντηση ή σφάλμα
- Shared: Αληθές αν το span προέρχεται από άλλον tracer
- localEndpoint: Περιγραφή σε επίπεδο δικτύου του κόμβου που φιλοξενεί την υπηρεσία από την οποία ξεκινάει το span
    - serviceName: Όνομα της υπηρεσίας στην οποία πραγματοποιείται ή ξεκινάει η λειτουργία
    - Ipv4: Η ipv4 διεύθυνση του κόμβου
    - Ipv6: Η ipv6 διεύθυνση του κόμβου
    - Port: Η θύρα που ακούει ο κόμβος
- remoteEndpoint :
    - Αν το πεδίο `Kind = Client` τότε ως remoteEndpoint είναι ο κόμβος server
    - Αν το πεδίο `Kind = Server` τότε ως remoteEndpoint είναι ο κόμβος client

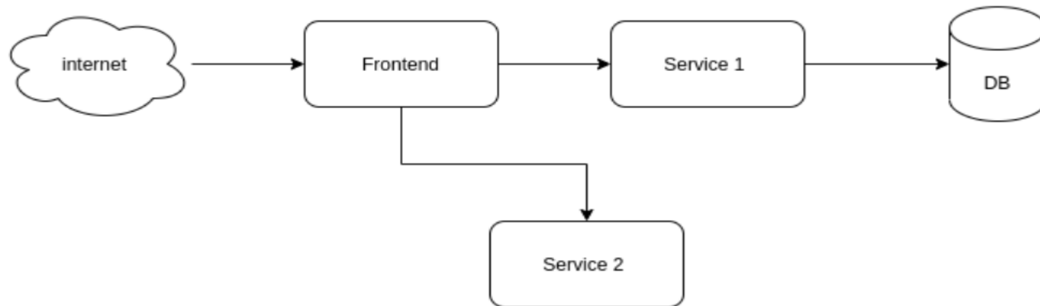
  Περιγραφή σε επίπεδο δικτύου του κόμβου που φιλοξενεί την υπηρεσία στην οποία καταλήγει το span
    - serviceName: Όνομα της υπηρεσίας στην οποία θα καταλήξει ή προέρχεται η λειτουργία
    - Ipv4: Η ipv4 διεύθυνση του κόμβου
    - Ipv6: Η ipv6 διεύθυνση του κόμβου
    - Port: Η θύρα που ακούει ο κόμβος
- Annotations: Σχολιασμοί ως μεταδεδομένα, που περιγράφουν γεγονότα με τις αντίστοιχες χρονικές στιγμές που συνέβησαν, σε ζευγάρι «timestamp-value»
- Tags: Ζευγάρια «κλειδιού-τιμής» (key-value pairs) ως μεταδεδομένα, χρήσιμα για φιλτράρισμα και ομαδοποίηση των spans με βάση αυτά.

```
SpanModel :

    traceId             <TraceID>
    name                <operation span name>
    parentId            <parentspanID>
    id                  <spanID>
    kind                <Server/Client/Producer/Consumer>
    timestamp           <begin_timestamp>
    duration            <duration>
    debug               <boolean>
    shared              <boolean>
    localEndpoint       <service name, ip, port of the service node>
    remoteEndpoint      <service name, ip, port of the service node>
    annotations         ['timestamp - value' pairs]
    tags                {key-value pairs}
```

2.2 Παραδείγματα με βάση μια πιλοτική εφαρμογή

Δίνεται ενδεικτικό παράδειγμα κατανεμημένης εφαρμογής με 3 μικροϋπηρεσίες και 1 βάση δεδομένων [Εικόνα 5]. Οι χρήστες καλούν το Frontend της εφαρμογής και αυτό με τη σειρά του καλεί την υπηρεσία 2 (Service 2) και την υπηρεσία 1 (Service 1), η οποία επικοινωνεί με τη βάση (DB).

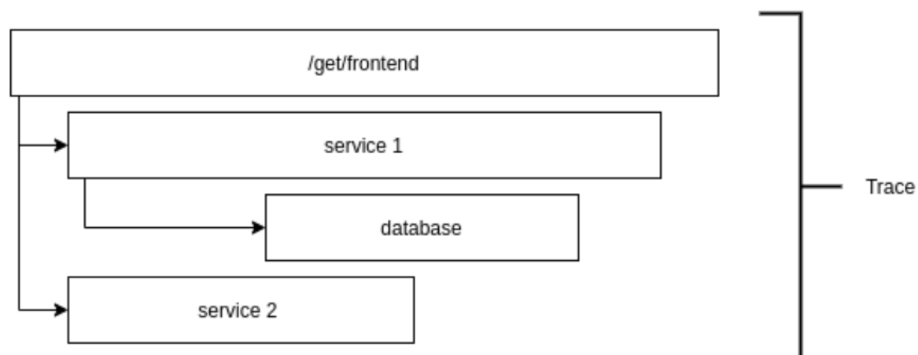


Εικόνα 5: Παράδειγμα κατανεμημένης εφαρμογής με μικροϋπηρεσίες

Από την κατανεμημένη παρακολούθηση / ιχνηλάτηση της εφαρμογής προκύπτουν 4 spans για το κάθε trace, όσες και οι διεργασίες που πραγματοποιούνται για την εξυπηρέτηση ενός αιτήματος [Εικόνα 6] :

1. Το root span "/get/frontend" που αναπαριστά όλο το αίτημα από τη στιγμή που το έλαβε η εφαρμογή μεχρι τη στιγμή που ολοκληρώθηκε
2. Το span "service 1" το οποίο αναπαριστά τις λειτουργίες που εκτελεί η υπηρεσία 1 (Service 1) όταν κληθεί από το Frontend
3. Το span "database" που εκφράζει την αναζήτηση στη βάση
4. Το span "service 2" που αναπαριστά τις λειτουργίες που εκτελεί η υπηρεσία 2 (Service 2)

Τα span "service 1" και "service 2" είναι παιδιά του "/get/frontend" span , ενώ το span "database" είναι παιδί του "service 1". Στη συγκεκριμένη περίπτωση για να ολοκληρωθεί το root span πρέπει να έχουν τελειώσει όλα τα spans-παιδιά του, θεωρώντας ότι εκτελεί σύγχρονες κλήσεις για την επικοινωνία των μικροϋπηρεσιών μεταξύ τους.



Εικόνα 6: Δομή trace με τα επιμέρους spans

2.3 Ορισμός των exemplars

Τα *exemplars* αφορούν ορισμένα traces που λειτουργούν ως αντιπροσωπευτικά παραδείγματα και εκπρόσωποι ενός μεγαλύτερου συνόλου από δεδομένα/traces που ακολουθούν ένα ίδιο μοτίβο [14]. Τα exemplars προσθέτονται ως ετικέτες (labels) σε κάποιες χρονοσειρές μετρικών με τη μορφή ζευγαριού «κλειδί-τιμή» (key-value pair). Έχουν ως στόχο να συνδέσουν τις μετρικές που συλλέγονται κατά την παρατήρηση ενός συστήματος (observability - monitoring) με τα traces που παράγονται στο σύστημα αυτό [15]. Η πιο συνήθης χρήση τους είναι να επισυνάπτουν ως κλειδί το traceID του trace που αντιστοιχεί στη τιμή μιας μετρικής μαζί με την αντίστοιχη τιμή του για αυτή την μετρική. Μέσω λοιπόν των exemplars, μπορούμε να μεταβούμε για ανάλυση σε traces που έχουν ενδιαφέρουσες τιμές σε μια συγκεκριμένη μετρική γνωρίζοντας το traceID τους, για παράδειγμα σε traces που αναπαριστούν αιτήματα με υψηλό χρόνο απόκρισης (μεγάλο latency). Έτσι διευκολύνεται η αναζήτηση και η ερμηνεία σφαλμάτων, αποτυχιών και γενικότερα της συμπεριφοράς ενός συστήματος.

2.3.1 Open Metrics και δομή των exemplars

Το OpenMetrics, που ανήκει στο Cloud Native Computing Foundation (CNCF), είναι ένα πρωτόκολλο-πρότυπο για μετάδοση cloud-native μετρικών, το οποίο υποστηρίζει τα exemplars [16]. Κατά βάση είναι παρόμοιο με το παραδοσιακό πρότυπο για αναπαράσταση και μετάδοση μετρικών, όπως το ορίζει το Prometheus, με την προσθήκη όμως των exemplars ως σχόλια με τη σήμανση # δίπλα στην τιμή μιας μετρικής. Το Prometheus αναγνωρίζει το πρότυπο OpenMetrics και μπορεί να διαβάζει και να εκθέτει μετρικές σε αυτή τη μορφή. Από την έκδοση 2.26 μάλιστα υποστηρίζεται η αποθήκευση των exemplars στη μνήμη του Prometheus ώστε να είναι διαθέσιμες για ερωτήσεις (queries) μέσα από το HTTP API του [17], ενώ στην έκδοση 2.28 μπορούν να αναπαρασταθούν γραφικά μαζί με το διάγραμμα της μετρικής [18].
Το κάθε exemplar πρέπει να αποτελείται υποχρεωτικά από ένα LabelSet (ως κλειδί), που συνήθως είναι το traceID, την τιμή του για την μετρική στην οποία επισυνάπτεται (exemplar_value) και πιθανώς και τη χρονική στιγμή που δημιουργήθηκε (timestamp) [16].

```
<metric name> <metric_value>  # {trace_id = <traceID>} <exemplar_value>
<timestamp>
```

Στο ακόλουθο παράδειγμα [Εικόνα 7] οι μετρικές παρουσιάζονται σε OpenMetrics πρότυπο με τα TraceIDs να εκτίθενται σε σχόλιο δίπλα από αυτές ως labels των exemplars, μαζί με την τιμή τους για την μετρική και τη χρονική στιγμή σύλληψής τους.

```
# TYPE foo histogram
foo_bucket{le="0.01"} 0
foo_bucket{le="0.1"} 8 # {} 0.054
foo_bucket{le="1"} 11 # {trace_id="KOO5S4vxi0o"} 0.67
foo_bucket{le="10"} 17 # {trace_id="oHg5SJYRHA0"} 9.8 1520879607.789
foo_bucket{le="+Inf"} 17
foo_count 17
foo_sum 324789.3
foo_created  1520430000.123
```

Εικόνα 7: Μετρικές με χρήση exemplars σε OpenMetrics μορφοποίηση[4]

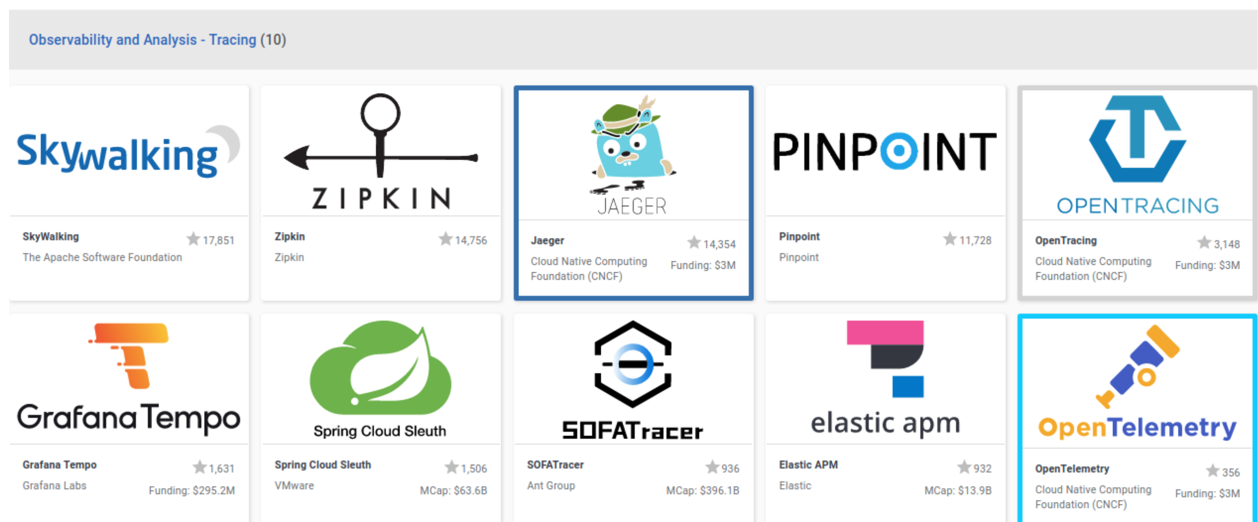
Μέσα από το HTTP API του Prometheus ζητώντας τα exemplars (query_exemplars) για μια συγκεκριμένη μετρική, τα λαμβάνουμε σε αυτή τη μορφή [19]:

```
"Exemplars":[
{"labels":{"traceID":"5d50451ede18a73f"}, "value":"0.076717818",
"timestamp":1624350348.532}
]
```

[4] https://promlabs.com/blog/2021/04/01/whats-new-in-prometheus-2-26

Κεφάλαιο 3: Υφιστάμενα εργαλεία και τεχνικές κατανεμημένης παρακολούθησης (ιχνηλάτησης) εφαρμογών υπολογιστικού νέφους

Παράλληλα με την ανάγκη για κατανεμημένη παρακολούθηση (ιχνηλάτηση) των cloud-native εφαρμογών, δημιουργήθηκαν και τα αντίστοιχα εργαλεία που διευκολύνουν αυτή τη διαδικασία. Στόχος τους είναι να παράγουν και να συλλέγουν τα δεδομένα ιχνηλάτησης (traces) από τις εφαρμογές, να τα επεξεργάζονται, να τα αναλύουν και να τα παρουσιάζουν γραφικά στον χρήστη. Στην [Εικόνα 8] παρουσιάζονται τα εργαλεία ανοιχτού κώδικα που ανήκουν στο Cloud Native Computing Foundation (CNCF) και μπορούν να χρησιμοποιηθούν για κατανεμημένη παρακολούθηση εφαρμογών υπολογιστικού νέφους.



Εικόνα 8: Εργαλεία ανοιχτού κώδικα CNCF για κατανεμημένη παρακολούθηση εφαρμογών υπολογιστικού νέφους[5]

3.1 Επισκόπηση τεχνικών κατανεμημένης παρακολούθησης

Η παρακολούθηση των κατανεμημένων εφαρμογών πραγματοποιείται κυρίως με *Application Performance Monitoring Tools (APM)* (εργαλεία και συστήματα παρακολούθησης της απόδοσης εφαρμογών). Αυτά τα εργαλεία έχουν ως στόχο τη συλλογή και ανάλυση μετρικών ώστε να ελέγχουν την απόδοση των εφαρμογών, να εντοπίζουν προβλήματα σε αυτές και να προειδοποιούν για πιθανά σφάλματα [20]. Για να γίνει αυτό, δημιουργούν ένα επίπεδο αναφοράς και κατώφλια για τις μετρικές, τα οποία, όταν ξεπεραστούν, σηματοδοτούν πιθανή υπολειτουργία του συστήματος. Οι μετρικές που συλλέγονται αφορούν τόσο στον εξυπηρετητή (χρήση της cpu, της μνήμης), όσο και στην ίδια την εφαρμογή (χρόνος απόκρισης, ρυθμός σφαλμάτων). Τα APM εργαλεία υποστηρίζουν και κατανεμημένη παρακολούθηση (distributed tracing) ερμηνεύοντας την

---

[5] https://landscape.cncf.io/card-mode?category=tracing&grouping=category&license=open-source&sort=stars

τοπολογία του συστήματος και παρέχοντας πληροφορίες για spans και traces που συλλέχθηκαν από την εφαρμογή. Από την [Εικόνα 8] τα «Skywalking», «Pinpoint» και «Elastic Apm», ανήκουν σε αυτή την κατηγορία συστημάτων παρακολούθησης.

3.2 Υφιστάμενα εργαλεία για κατανεμημένη παρακολούθηση

Το πρώτο βήμα για την κατανεμημένη παρακολούθηση είναι ο καθορισμός κατάλληλων σημείων παρακολούθησης (instrumentation) των εφαρμογών και των υπηρεσιών τους, η κατάλληλη δηλαδή ρύθμισή τους με τη χρήση βιβλιοθηκών από συστήματα παρακολούθησης, ώστε να δημιουργούν δεδομένα ιχνηλάτησης (traces) κατά την εξυπηρέτηση αιτημάτων και την πραγματοποίηση εσωτερικών διεργασιών. Αυτά τα δεδομένα που παράγονται, από τις ειδικά διαμορφωμένες (instrumented) εφαρμογές, συλλέγονται στη συνέχεια από ένα σύστημα αποθήκευσης και ανάλυσης των traces (backend), το οποίο πρέπει να είναι συμβατό με τη μορφή που αυτά παρέχονται, ώστε να μπορέσει να τα ομαδοποιήσει και να τα επεξεργαστεί.

Το 2010 με το άρθρο της Google «Dapper» [26] παρουσιάστηκε το βασικό τους εργαλείο για κατανεμημένη παρακολούθηση στα πολύπλοκα συστήματά τους. Το πρώτο εργαλείο ιχνηλάτησης ανοιχτού κώδικα, εμπνευσμένο από το Dapper, παρουσιάστηκε από το Twitter το 2012 με το όνομα Zipkin [Εικόνα 8], το οποίο τώρα διατηρείται από την κοινότητά του. Το Zipkin συλλέγει και ομαδοποιεί χρονικά δεδομένα (timing data) με στόχο την ανάλυση των επιδόσεων κατανεμημένων εφαρμογών [21]. Ακολουθώντας την ίδια νοοτροπία με το Zipkin αλλά διαφορετική υλοποίηση, το 2015 η Uber ανέπτυξε το Jaeger, το δικό της εργαλείο κατανεμημένης παρακολούθησης ανοιχτού κώδικα, το οποίο ανήκει από το 2017 στο Cloud Native Computing Foundation [Εικόνα 8] [22]. Η γενική αρχιτεκτονική και των δύο αυτών ευρέως διαδεδομένων εργαλείων είναι κοινή. Τα κατανεμημένα συστήματα, έχοντας προγραμματιστεί και ρυθμιστεί να να παράγουν και εκθέτουν δεδομένα ιχνηλάτησης, στέλνουν τα traces τους στον αντίστοιχο συλλέκτη (trace-collector) που έχουν υλοποιήσει. Αυτός με τη σειρά του καταγράφει τα δεδομένα αυτά, τα ομαδοποιεί, δημιουργεί τις σχέσεις μεταξύ τους, τα αποθηκεύει και τα διαθέτει για αναζήτηση από το χρήστη και παρουσίαση στη διεπαφή των χρηστών (User Interface - UI) [25] . Στον [Πίνακα 1] γίνεται σύγκριση των 2 αυτών βασικών εργαλείων.

Πίνακας 1: Σύγκριση εργαλείων Zipkin - Jaeger

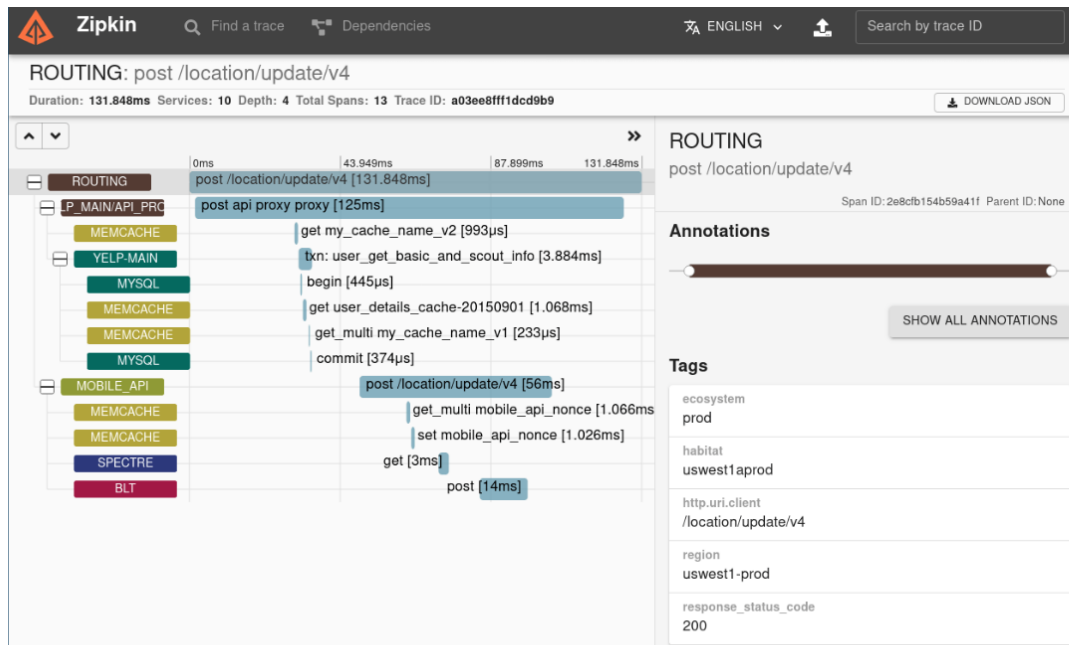
Zipkin [23]	**Jaeger** [24]
Github (ανοιχτού κώδικα από το 2012) • 2.8 χιλιάδες Forks • 14.7 χιλιάδες Stars	Github (ανοιχτού κώδικα από το 2015) • 1.7 χιλιάδες Forks • 14.2 χιλιάδες Stars
	Ανήκει στο Cloud Native Computing Foundation (CNCF)
Γραμμένο σε Java	Γραμμένο σε Golang
Διαθέτει δικές του βιβλιοθήκες για παρακολούθηση, δημιουργία και συλλογή	Υποστηρίζει και χρησιμοποιεί εγγενώς το OpenTracing API για παρακολούθηση

traces (βιβλιοθήκες για instrumentation)	συστημάτων (instrumentation). Συνεπώς, μπορεί να υποστηρίξει όλα τα προγραμματιστικά πλαίσια (frameworks) που υποστηρίζει και το OpenTracing, όπως: • Python-django • Python-flask • Java-spring • node.js
Βιβλιοθήκες του Zipkin: • C#: Zipkin4net library • Go: zipkin-go library • Java: brave library • Javascript: zipkin-js library • Ruby: zipkin-ruby library • Scala: zipkin-finagle library • PHP: zipkin-php library	Βιβλιοθήκες του Jaeger βασισμένες σε OpenTracing API: • C#: jaegetracing/jaeger-client-csharp • Go: jaegetracing/jaeger-client-go • Java: jaegetracing/jaeger-client-java • Python: jaegetracing/jaeger-client-python • Node.js: jaegetracing/jaeger-client-node • C++: jaegetracing/jaeger-client-cpp
Συμβατότητα με άλλα συστήματα και βιβλιοθήκες για παρακολούθηση και διαχείριση συστημάτων (instrumentation): • OpenTracing • OpenCensus • OpenTelemetry	Συμβατότητα με άλλα συστήματα και βιβλιοθήκες για παρακολούθηση και διαχείριση συστημάτων (instrumentation): • OpenTracing (εγγενώς) • OpenCensus • OpenTelemetry
Δυνατότητα αποθήκευσης δεδομένων (traces και spans) σε: • Cassandra (εγγενής συμβατότητα) • Elasticsearch (εγγενής συμβατότητα) • My-SQL(εγγενής συμβατότητα) • Άλλες εξωτερικές βάσεις δεδομένων (third party back-ends) • Εσωτερικά στη μνήμη (in-memory)	Δυνατότητα αποθήκευσης δεδομένων (traces και spans) σε: • Cassandra (εγγενής συμβατότητα) • Elasticsearch (εγγενής συμβατότητα) • Εσωτερικά στη μνήμη (in-memory) • Και άλλα συστήματα αποθήκευσης δεδομένων που είναι υπό δοκιμή από την κοινότητα

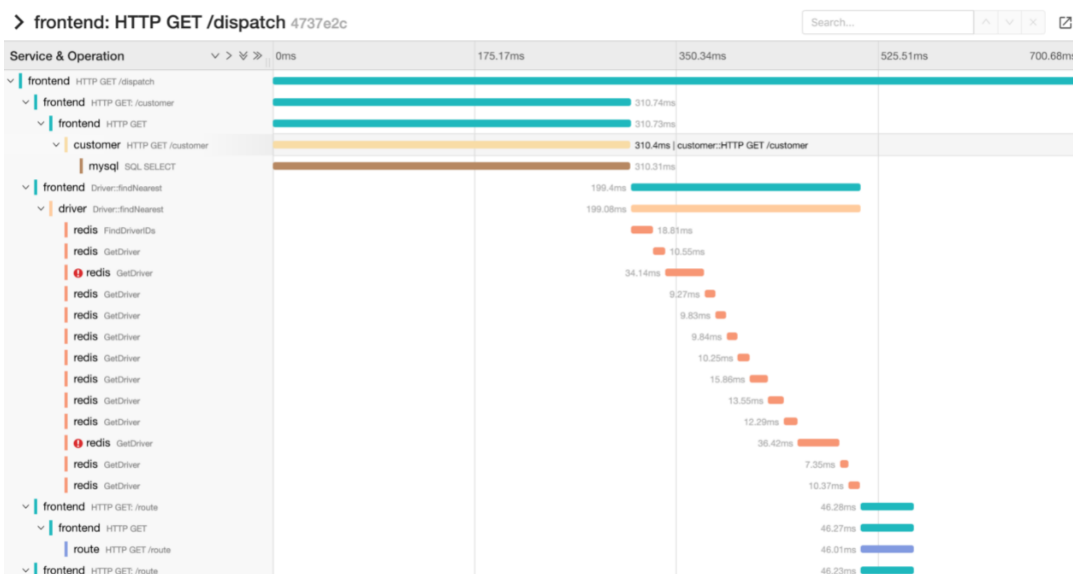
Δυνατότητα δειγματοληψίας στα traces:
• Σταθερός ρυθμός δειγματοληψίας
    ○ 1: κρατάει όλα τα traces
    ○ 0: δεν κρατάει κανένα
• Πιθανοτική δειγματοληψία με Spring Cloud Sleuth
	Δυνατότητα δειγματοληψίας στα traces:
	• Σταθερός ρυθμός δειγματοληψίας
	    ○ 1: κρατάει όλα τα traces
	    ○ 0: δεν κρατάει κανένα
	• Πιθανοτική δειγματοληψία
	• Δειγματοληψία με σταθερό ρυθμό
	• Remote, όπου ο Jaeger agent προτείνει την κατάλληλη στρατηγική δειγματοληψίας

Παρέχει Web UI [Εικόνα 9] Το dashboard μπορεί να χρησιμοποιηθεί	Παρέχει Web UI [Εικόνα 10] Το dashboard μπορεί να χρησιμοποιηθεί για

για να οπτικοποιήσει τα traces με τα αντίστοιχα spans και πληροφορίες για αυτά, αλλά και τις σχέσεις των services μεταξύ τους.	να οπτικοποιήσει τα traces με τα αντίστοιχα spans και πληροφορίες για αυτά, αλλά και τις σχέσεις των services μεταξύ τους.
Πρωτόκολλα για τη μεταφορά των spans, από τις εφαρμογές στις οποίες συλλέχθηκαν προς τους συλλέκτες: • HTTP • Kafka • Scribe	Πρωτόκολλα για τη μεταφορά των spans, από τις εφαρμογές στις οποίες συλλέχθηκαν προς τους συλλέκτες: • HTTP • UDP
Η αρχιτεκτονική του Zipkin [Εικόνα 11] αποτελείται από: • Συλλέκτη (Collector) • Προγραμματιστικές διεπαφές (API) για ερωτήσεις (Query API) • Διεπαφές χρήστη (UI) • Αποθηκευτικό χώρο για τα spans	Η αρχιτεκτονική του Jaeger [Εικόνα 12] αποτελείται από: • Πράκτορας (Agent) • Συλλέκτη (Collector) • Προγραμματιστικές διεπαφές (API) για ερωτήσεις (Query API) • Διεπαφές χρήστη (UI) • Αποθηκευτικό χώρο για τα spans
	Το jaeger είναι συμβατό με το API του zipkin, μπορεί, δηλαδή, ο συλλέκτης του jaeger να δεχτεί, μέσω HTTP, spans που ακολουθούν τη μορφοποίηση (format) του zipkin (Thrift, JSON v1/v2 and Protobuf) έχοντας παραχθεί από βιβλιοθήκες (instrumentation) του zipkin.
Τρόποι εκτέλεσης: • Java (.jar) • Docker (εικόνα: `openzipkin/zipkin`) • Από τον πηγαίο κώδικα στο github (ιδίως για ανάπτυξη νέων χαρακτηριστικών)	Τρόποι εκτέλεσης: • Ως ένα all-in-one εκτελέσιμο αρχείο που όλα τα τμήματα του jaeger τρέχουν από αυτό • Με Docker είτε ως κατανεμημένο σύστημα με ξεχωριστά όλα τα επιμέρους τμήματά του (agent, collector, query, ingester) είτε με την εικόνα: `jaegertracing/all-in-one` που τα περιέχει όλα
API για πρόσβαση στα traces και spans σε JSON μορφή, αλλά και σε πληροφορίες για τις υπηρεσίες (services) και τις εξαρτήσεις τους [13] `http://localhost:9411/api/v2`	API για πρόσβαση στα traces και spans σε JSON μορφή `http://localhost:16686/api/`



Εικόνα 9: Αναπαράσταση trace στο Web UI του Zipkin[6]



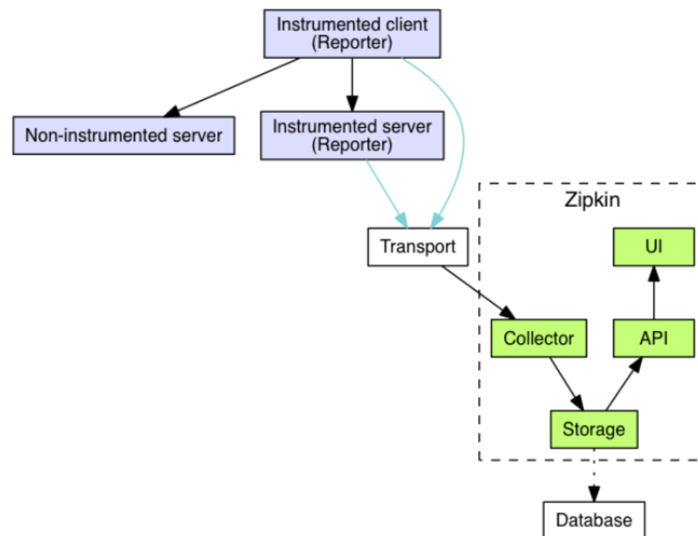
Εικόνα 10: Αναπαράσταση trace στο Web UI του Jaeger[7]

[6] https://zipkin.io/

[7] https://www.jaegertracing.io/docs/1.25/frontend-ui/

3.2.1 Αρχιτεκτονική Zipkin

Το Zipkin αποτελεί μια αυτόνομη λύση για κατανεμημένη ιχνηλάτηση, καθώς παρέχει τις δικές του βιβλιοθήκες (zipkin client library) και tracer, για κατάλληλο προγραμματισμό της εφαρμογής ώστε να παράγει trace δεδομένα, αλλά και backend - χώρο αποθήκευσης, συσχέτισης και ομαδοποίησης των spans.

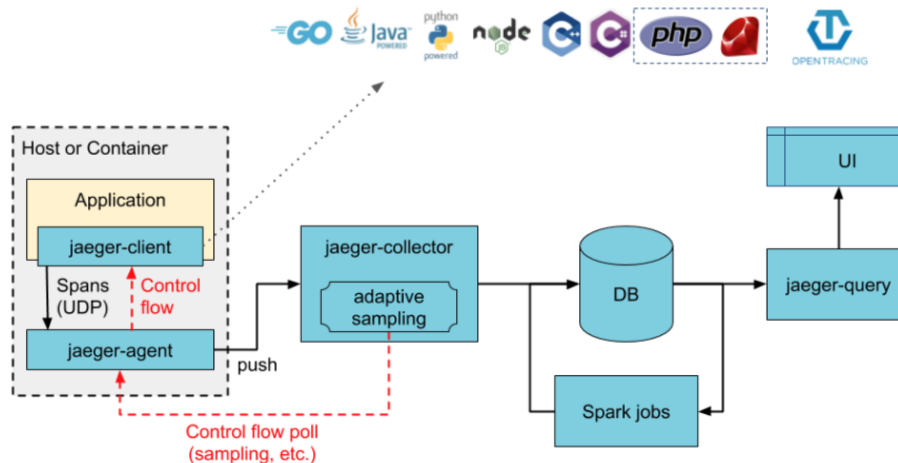


Εικόνα 11: Διάγραμμα αρχιτεκτονικής του Zipkin[8]

Συγκεκριμένα, με χρήση του zipkin client library δημιουργείται ένας tracer (ιχνηλάτης) ο οποίος είναι υπεύθυνος για τη δημιουργία των spans με τα δεδομένα τους. Όπως παρουσιάζεται στην [Εικόνα 11], όλα τα spans τα οποία δημιουργούνται στέλνονται από τον Ανταποκριτή (Reporter) μέσω κάποιου πρωτοκόλλου μεταφοράς (HTTP, Kafka, Scribe) στο Zipkin backend ασύγχρονα και συγκεκριμένα στον Συλλέκτη (Collector). Ο ρόλος του Συλλέκτη ως δαίμονα είναι να δεικτοδοτήσει (index) τα spans και να τα αποθηκεύσει στον αποθηκευτικό χώρο (Cassandra, ElasticSearch, MySQL, εσωτερικά στη μνήμη (in-memory) χωρίς μόνιμη αποθήκευση ή σε εξωτερική εξωτερική βάση δεδομένων). Έπειτα, μέσω του JSON API του Zipkin, είναι εφικτή η συλλογή των trace δεδομένων μέσω ερωτημάτων (queries) και η παρουσίασή τους σε Web UI με φιλική προς το χρήστη εικονοποίηση. Μέσα από αυτό, μάλιστα, τα traces μπορούν να ομαδοποιηθούν με βάση το όνομα της υπηρεσίας, ελάχιστη ή μέγιστη διάρκεια του trace, χρονικές στιγμές, tags και σχολιασμούς (annotations) που φέρουν.

[8] https://zipkin.io/pages/architecture.html

3.2.2 Αρχιτεκτονική Jaeger



Εικόνα 12: Διάγραμμα αρχιτεκτονικής του Jaeger[9]

Το Jaeger δεν διαθέτει δικές του βιβλιοθήκες αλλά υποστηρίζει εγγενώς και χρησιμοποιεί το OpenTracing API για ρύθμιση της εφαρμογής ώστε να παράγει trace δεδομένα. Έτσι η κάθε μικροϋπηρεσία μπορεί να δημιουργεί spans όταν λαμβάνει ένα νέο αίτημα και να του επισυνάπτει τη χρήσιμη πληροφορία του με τα IDs. Το Jaeger, σε αντίθεση με το Zipkin, όπως φαίνεται στην [Εικόνα 12] έχει έναν jaeger agent (δαίμονα δικτύου) που συλλέγει spans μέσω UDP, τα ομαδοποιεί και τα προωθεί στον Συλλέκτη (jaeger-collector), ο οποίος με τη σειρά του, αφού τα ταξινομήσει, τα αποθηκεύει στη βάση δεδομένων (Cassandra, Elasticsearch, Kafka ή εσωτερικά στη μνήμη). Κάνοντας ερωτήσεις (jaeger-query) στη βάση μπορούμε να αναζητήσουμε πληροφορία για τα traces και να την παρουσιάσουμε μέσω του web UI στο χρήστη.

3.2.3 Spring Cloud Sleuth

Το Spring Cloud Sleuth [27] παρέχει αυτόματα κατανεμημένη παρακολούθηση και ιχνηλάτηση Spring Boot εφαρμογών, υλοποιώντας την βιβλιοθήκη Brave [28] του Zipkin. Συνεπώς τα δεδομένα που παράγονται και συλλέγονται, μπορούν να μεταφερθούν με απόλυτη συμβατότητα στο Zipkin backend για ανάλυση και παρουσίαση.

3.2.4 Προτυποποιημένο API Διαχείρισης Εφαρμογών (Standardized Instrumentation API)

Στις σύγχρονες cloud-native εφαρμογές, οι εκατοντάδες μικροϋπηρεσίες οι οποίες τις απαρτίζουν, υλοποιούνται κατά κύριο λόγο με διαφορετικές τεχνολογίες και σε διαφορετική γλώσσα η καθεμία. Για να μπορέσει πιο εύκολα μια τέτοια εφαρμογή να παρακολουθηθεί και να προγραμματιστεί ώστε να μπορεί να ιχνηλατηθεί, προτάθηκε η προτυποποίηση (standardization) των αρχών και του API ιχνηλάτησης [9]. Σε αυτό το πλαίσιο έχουν αναπτυχθεί το OpenTracing, OpenCensus και OpenTelemetry. Το καθένα από αυτά παρέχει ένα σύνολο από βιβλιοθήκες, APIs, frameworks που υλοποιούν κάποιες προδιαγραφές ώστε

[9] https://www.jaegertracing.io/docs/1.26/architecture/

να μπορούν να δημιουργηθούν δεδομένα trace από τις μικροϋπηρεσίες, χωρίς να δεσμεύονται σε κάποιο πάροχο (vendor neutral). Παρόλ'αυτά, δεν παρέχουν κάποιο σύστημα συλλογής και αποθήκευσης των trace δεδομένων που παράγονται, όπως διαθέτει το Zipkin και το Jaeger. Μπορούν όμως να εξάγουν τα δεδομένα τους σε κάποιο backend από τα προαναφερθέντα.

Πίνακας 2: Σύγκριση OpenTracing - OpenCensus - OpenTelemetry [9]

Open Tracing [29]	**Open Census** [30]	**Open Telemetry** [31]
Ανοιχτού κώδικα από το 2016	Ανοιχτού κώδικα από το 2018	Ανοιχτού κώδικα από το 2019, ως ένωση των Open Tracing και Open Census
Έργο του CNCF [Εικόνα 8]	Εμπνευσμένο από το πρόγραμμα Census της Google	Έργο του CNCF [Εικόνα 8]
Στόχος είναι να παρέχει ένα τυποποιημένο μηχανισμό για παρακολούθηση και ιχνηλάτηση με τις βιβλιοθήκες και τα πακέτα να είναι ανεξάρτητα από γλώσσες και παρόχους.	Στόχος του Census και μετέπειτα και του OpenCensus είναι να παρέχει μια ενιαίο μέθοδο για παρακολούθηση εφαρμογών με σύλληψη traces και μετρικών από τις υπηρεσίες αυτόματα, ενσωματώνοντας τεχνολογίες όπως gRPC.	Στόχος ήταν η ενοποίηση των δύο προτύπων που προσέφεραν το OpenTracing και το OpenCensus σε ένα κοινό, ώστε να επιτρέπεται η εύκολη μεταφορά δεδομένων (portability)
Παρέχει API για δημιουργία, διαχείριση των spans και μετάδοσή τους μεταξύ των services, ανεξάρτητα από τον πάροχο (vendor). Το API αποτελείται από tracer, span και span context.	Είναι ένα ολόκληρο SDK για ιχνηλάτηση με traces και μετρικές, με πλήρη υλοποίηση πέρα από το API. Τα δεδομένα που συλλέγονται μπορούν να εξαχθούν σε οποιοδήποτε εργαλείο ανάλυσής τους/backend.	Σύνολο από APIs, SDKs, βιβλιοθήκες, πράκτορες και συλλέκτες για την ανεξάρτητη από παρόχους, ενοποιημένη παρακολούθηση εφαρμογών με συλλογή traces και μετρικών. Η φορητότητα που προσφέρεται, επιτρέπει την εξαγωγή όλων των δεδομένων μέσω του Συλλέκτη σε διάφορα εργαλεία ανάλυσης και οπτικοποίησης τους (Jaeger, Prometheus, Zipkin).

3.3 Υποστήριξη κατανεμημένης παρακολούθησης από εργαλεία ενορχήστρωσης και παρακολούθησης

Όπως αναφέρθηκε και νωρίτερα, οι κατανεμημένες εφαρμογές συνηθίζεται να στήνονται σε ενορχηστρωτές όπως το Kubernetes. Έτσι η παρακολούθηση της υγείας μια εφαρμογής, μετατρέπεται και σε παρακολούθηση του cluster (συστάδας), των κόμβων (nodes) και των pods στα οποία εκτελούνται τα containers που την συγκροτούν.
Μέσα από το API του Kubernetes δίνονται πληροφορίες για την κατάσταση των αντικειμένων (pods, services, deployments), το πλήθος και τον επιθυμητό αριθμό αντιγράφων τους. Μέσα από την 'kube-state-metrics' υπηρεσία [32], η οποία όταν εκτελεστεί στο cluster επικοινωνεί με τον διακομιστή του Kubernetes API, οι παραπάνω μετρικές επιπέδου cluster εκφράζονται σε μορφοποίηση συμβατή με το Prometheus, ώστε να μπορούν να συλλεχθούν από αυτό.
Το Prometheus [33] χρησιμοποιείται συχνά στην κατανεμημένη παρακολούθηση εφαρμογών, ως ένα ανοιχτού κώδικα σύστημα που ανήκει στο CNCF, για την καταγραφή και αποθήκευση μετρικών με στόχο την παρουσίαση και ανάλυσή τους. Εκτός από μετρικές σχετικές με το Kubernetes cluster, το Prometheus παράγει και συλλέγει μετρικές που αφορούν στην εφαρμογή. Μέσω της χρήσης κατάλληλων βιβλιοθηκών του, η κάθε μικροϋπηρεσία προγραμματίζεται και ρυθμίζεται κατάλληλα ώστε να εκθέτει μετρικές σε ένα συγκεκριμένο HTTP endpoint (συνήθως στο μονοπάτι "`/metrics`"). Το Prometheus συλλέγει τις μετρικές που εκτίθενται μέσω αυτού του endpoint και στη συνέχεια μπορεί να τις παρέχει είτε μέσω API είτε γραφικά.
Το Prometheus χρησιμοποιεί ειδικού τύπου βάση δεδομένων για αποθήκευση χρονοσειρών (time-series database), η οποία αποθηκεύει και δέχεται δεδομένα οργανωμένα σε τιμές μέσα σε μια περίοδο του χρόνου. Κατά αυτόν τον τρόπο αποθηκεύονται και οι μετρικές που εκθέτουν οι εφαρμογές. Από το API και μέσω μιας ειδικής γλώσσας για ερωτήσεις πάνω σε αυτά τα δεδομένα, την PROMQL, υπάρχει η δυνατότητα ομαδοποίησης και συγκέντρωσης αυτών των χρονοσειρών.
Το Prometheus παρέχει 4 είδη μετρικών [35] με τις οποίες μπορούμε να συλλέξουμε χρήσιμες πληροφορίες:

1. Αθροιστή (Counter): για τιμές που μόνο αυξάνονται και δεν μειώνονται ποτέ, για παράδειγμα τα συνολικά αιτήματα που δέχεται μια εφαρμογή.
2. Μετρητή (Gauge): για τιμές που μπορούν και να αυξηθούν και να μειωθούν, για παράδειγμα η χρήση της μνήμης.
3. Ιστόγραμμα (Histogram): εκφράζει τη συχνότητα που οι τιμές μιας μεταβλητής που παρατηρείται ανήκουν σε κάποια ορισμένα εύρη τιμών. Για παράδειγμα, αντί να αποθηκεύεται η διάρκεια του κάθε αιτήματος, υπολογίζεται κατά προσέγγιση αποθηκεύοντας τη συχνότητα των αιτημάτων που ανήκουν σε εύρη διαρκειών που έχουν οριστεί εκ των προτέρων.
4. Περίληψη (Summary): παρόμοια μετρική με τα ιστογράμματα που προτιμάται όταν απαιτείται ακριβής υπολογισμός quantiles.

Μια χρήσιμη συνάρτηση της PromQL που θα χρησιμοποιηθεί παρακάτω είναι η «rate()», η οποία υπολογίζει το ρυθμό αλλαγής της χρονοσειράς που δίνεται ως παράμετρος. Η συνάρτηση αυτή χρησιμοποιείται μόνο με αθροιστές (counters).
Με την κατάλληλη παραμετροποίηση, μαζί με τις χρονοσειρές μετρικών μπορούν να συλλεχθούν και να αποθηκευτούν exemplars, τα οποία εμφανίζονται ως στίγματα μαζί με

την γραφική αναπαράσταση των μετρικών. Οι βιβλιοθήκες (client libraries) του Prometheus που υποστηρίζουν τα exemplars είναι :

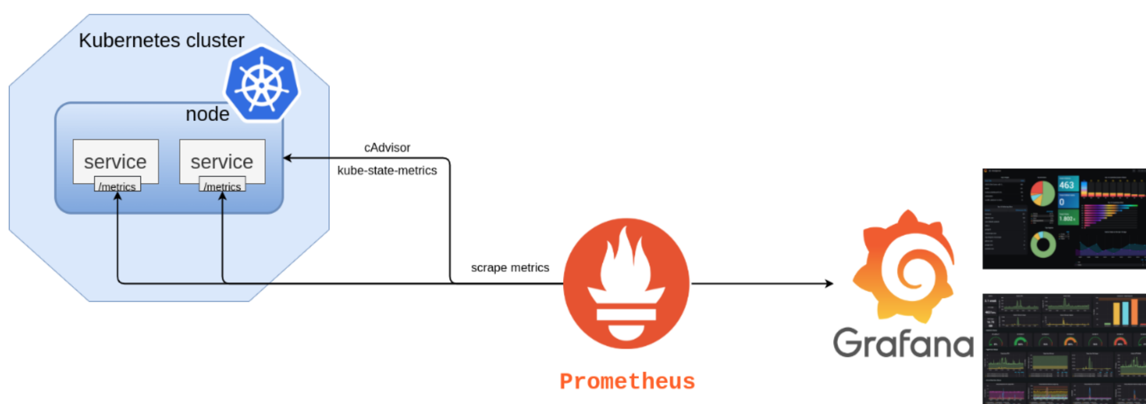
- Go client library
- Java client library

Τα exemplars υποστηρίζονται μόνο στις μετρικές Counter και Histogram του Prometheus. Το Prometheus μπορεί συλλέγει μετρικές για την κατάσταση του cluster, των κόμβων, των Pods και των containers, οι οποίες παράγονται από τον εκάστοτε ενορχηστρωτή. Χρήσιμες μετρικές για την κατανάλωση πόρων cpu και μνήμης από τα containers των Pods καταγράφονται από τον πράκτορα «cAdvisor» του Kubernetes και εκτίθενται σε μορφοποίηση συμβατή με το Prometheus, ώστε να συγκεντρωθούν από αυτό [Πίνακας 3].

Πίνακας 3: Περιγραφή μετρικών που παράγονται από Kubernetes για cpu και μνήμη

Όνομα μετρικής	Είδος μετρικής	Λειτουργία
container_cpu_usage_seconds_total	Αθροιστής (counter)	Συνολικός χρόνος σε δευτερόλεπτα που έχει χρησιμοποιηθεί η cpu του Pod από κάθε container που περιέχει
container_memory_usage_bytes	Μετρητής (Gauge)	Μετρητής χρήσης μνήμης του Pod σε byte από το κάθε container που περιέχει

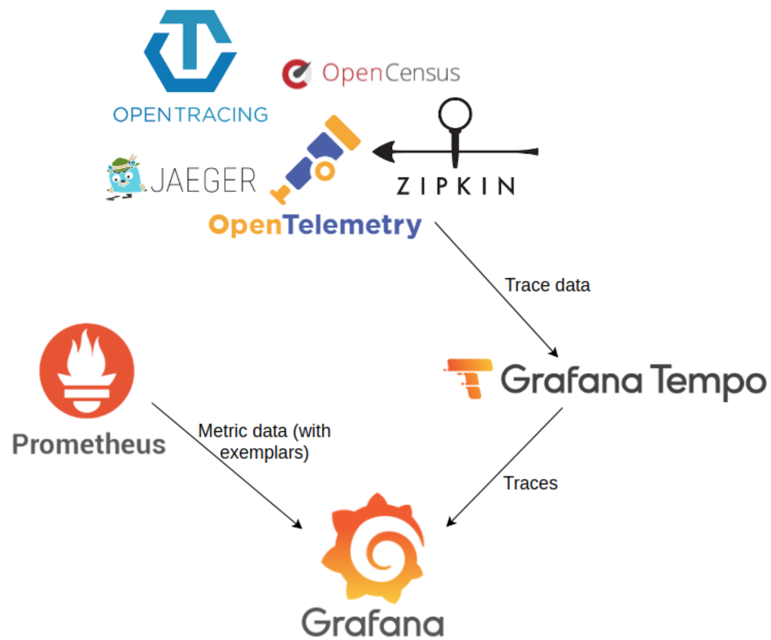
Για καλύτερη γραφική αναπαράσταση των μετρικών που συλλέγει το Prometheus χρησιμοποιείται το Grafana, το οποίο μπορεί να ομαδοποιήσει διάφορες μετρικές και να δημιουργήσει Dashboards [34] [Εικόνα 13].



Εικόνα 13: Συλλογή δεδομένων από Prometheus και αναπαράσταση σε Grafana

Το Grafana Tempo [Εικόνα 8], που αναπτύχθηκε από το Grafana Labs το 2020, είναι ένα ανοιχτού κώδικα σύστημα αποθήκευσης δεδομένων ιχνηλάτησης (traces) διαθέσιμων για ερωτήσεις (queries), τα οποία μπορούν να προέρχονται από οποιοδήποτε πρωτόκολλο παρακολούθησης και ιχνηλάτησης που αναφέρθηκε παραπάνω (Jaeger/OpenTracing, Zipkin, OpenCensus, OpenTelemetry)[49]. Το Grafana υποστηρίζει εγγενώς την οπτικοποίηση των δεδομένων ιχνηλάτησης (traces) του Grafana Tempo, καθώς και

ερωτήσεις (queries) προς αυτά. Συνεπώς μέσα από το Grafana, οι μετρικές με exemplars, που διατίθενται από το Prometheus, συνδέονται με τα αντίστοιχα traces από το Grafana Tempo και έτσι επιτρέπεται η μετάβαση από την τιμή μιας μετρικής στο trace του exemplar με το οποίο εκδόθηκε [Εικόνα 14].



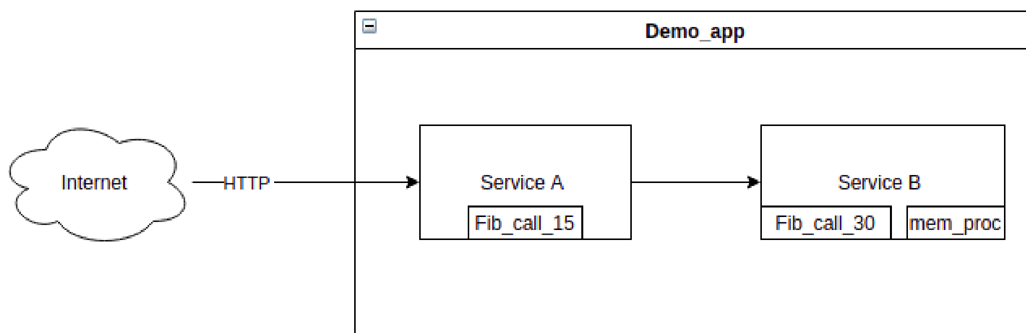
Εικόνα 14: Grafana για αναπαράσταση και σύνδεση δεδομένων από Prometheus και Grafana Tempo με χρήση exemplars

Κεφάλαιο 4: Προτεινόμενη Αρχιτεκτονική

4.1 Περιγραφή της προτεινόμενης λύσης

4.1.1 Πιλοτική υλοποίηση

Στο πλαίσιο αυτής της εργασίας, έγινε μια πιλοτική υλοποίηση με στόχο την ενοποίηση μετρικών και δεδομένων παρατηρησιμότητας σε μια κατανεμημένη εφαρμογή υπολογιστικού νέφους. Ακολουθώντας την κατανεμημένη αρχιτεκτονική των cloud-native εφαρμογών, η εφαρμογή που αναπτύχθηκε για τις δοκιμές αποτελείται από 2 μικροϋπηρεσίες, το Service A και το Service B, όπου η κάθε μικροϋπηρεσία επιτελεί μια ξεχωριστή λειτουργία [Εικόνα 15].



Εικόνα 15: Δομή πιλοτικής κατανεμημένης εφαρμογής

Το Service A δέχεται αιτήματα στο `http://localhost:8080/` και εκτελεί μια γρήγορη και υπολογιστικά ελαφριά λειτουργία, τον υπολογισμό του 15ου αριθμού της ακολουθίας Fibonacci. Στη συνέχεια καλεί με σύγχρονο http αίτημα το Service B, το οποίο με τη σειρά του καλεί εσωτερικά 2 διαδικασίες. Αρχικά εκτελεί έναν υπολογισμό Fibonacci του 30ου αριθμού της ακολουθίας, ο οποίος είναι αρκετά απαιτητικός σε επεξεργαστική ισχύ, και έπειτα μια διαδικασία προσθήκης χαρακτήρων σε αρχείο, η οποία καταναλώνει όλο και περισσότερη μνήμη καθώς μεγαλώνει το μέγεθος του αρχείου κάθε φορά που καλείται. Όταν ολοκληρωθούν όλες οι διαδικασίες, επιστρέφεται ένα μήνυμα επιτυχίας στον χρήστη που έκανε το αρχικό αίτημα στην εφαρμογή.

4.1.2 Κατανεμημένη παρακολούθηση εφαρμογής

Για την κατανεμημένη παρακολούθηση και ιχνηλάτηση της εφαρμογής επιλέχθηκε να χρησιμοποιηθεί το εργαλείο Zipkin με αποθήκευση των trace δεδομένων εσωτερικά στη μνήμη του (in-memory), ενώ για τη λήψη και συλλογή μετρικών το εργαλείο Prometheus. Για την αξιοποίηση των exemplars, επιλέχθηκε βιβλιοθήκη του Prometheus σε γλώσσα που

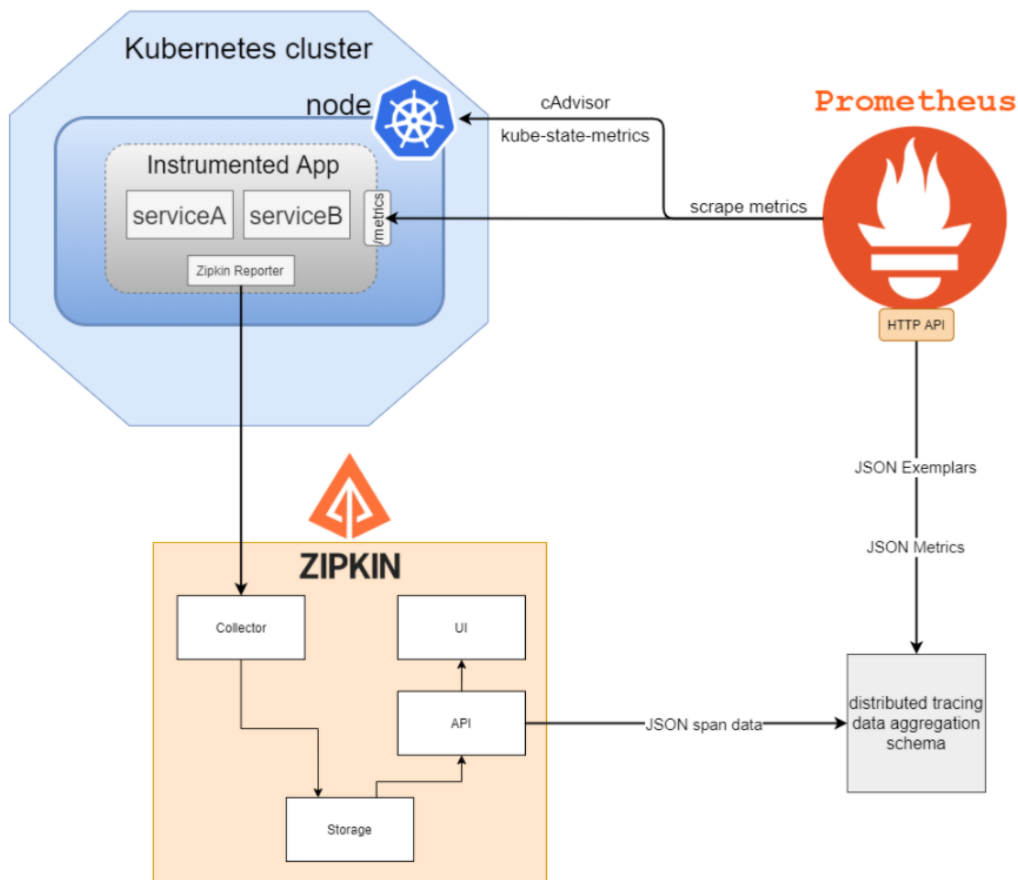
τα υποστηρίζει και συγκεκριμένα σε Golang. Συνεπώς όλη η πιλοτική εφαρμογή αναπτύχθηκε σε Golang.
Με την κατάλληλη βιβλιοθήκη του zipkin σε Golang για παρακολούθηση της εφαρμογής, παράγονται traces τα οποία μεταφέρονται για ανάλυση και αποθήκευση σε αυτό. Οι μετρικές που έχουν παραχθεί με χρήση κατάλληλων βιβλιοθηκών του Prometheus, εκτίθενται στο κατάλληλο `/metrics` endpoint της εφαρμογής μαζί με exemplars (`http://localhost:8080/metrics`), ώστε να συλλεχθούν και να ομαδοποιηθούν από το Prometheus. Μαζί με τις μετρικές που συλλέγονται από προεπιλογή, στο πλαίσιο της παρακολούθησης της εφαρμογής, δημιουργήθηκαν μετρικές ειδικού σκοπού [Πίνακας 4] :

Πίνακας 4: Δημιουργία ειδικών μετρικών εφαρμογής με το Prometheus

Όνομα μετρικής	**Είδος μετρικής**	**Λειτουργία**	**Exemplar**
request_latency_seconds_total	Αθροιστής (counter)	Αθροίζει τις διάρκειες εξυπηρέτησης των αιτημάτων που στάλθηκαν στην εφαρμογή	Ναι
request_latency	Μετρητής (gauge)	Κρατάει τη διάρκεια εξυπηρέτησης του κάθε αιτήματος	Δεν υποστηρίζεται
http_requests_total	Αθροιστής (counter)	Αθροίζει τον αριθμό των αιτημάτων που λαμβάνει η εφαρμογή	Όχι
errors_500_total	Αθροιστής (counter)	Αθροίζει τις διάρκειες εξυπηρέτησης των αιτημάτων που εμφάνισαν κωδικό κατάστασης 500, κατά την κλήση προς το Service b. Υπάρχουν 2 περιπτώσεις σφάλματος: 1. το pod του Service b έχει καταρρεύσει και είναι εκτός λειτουργίας οπότε επιστρέφεται κωδικός 500 με μήνυμα `"Get "http://serviceb:8080/": dial tcp 10.97.14.101:8080: connect: connection`	Ναι

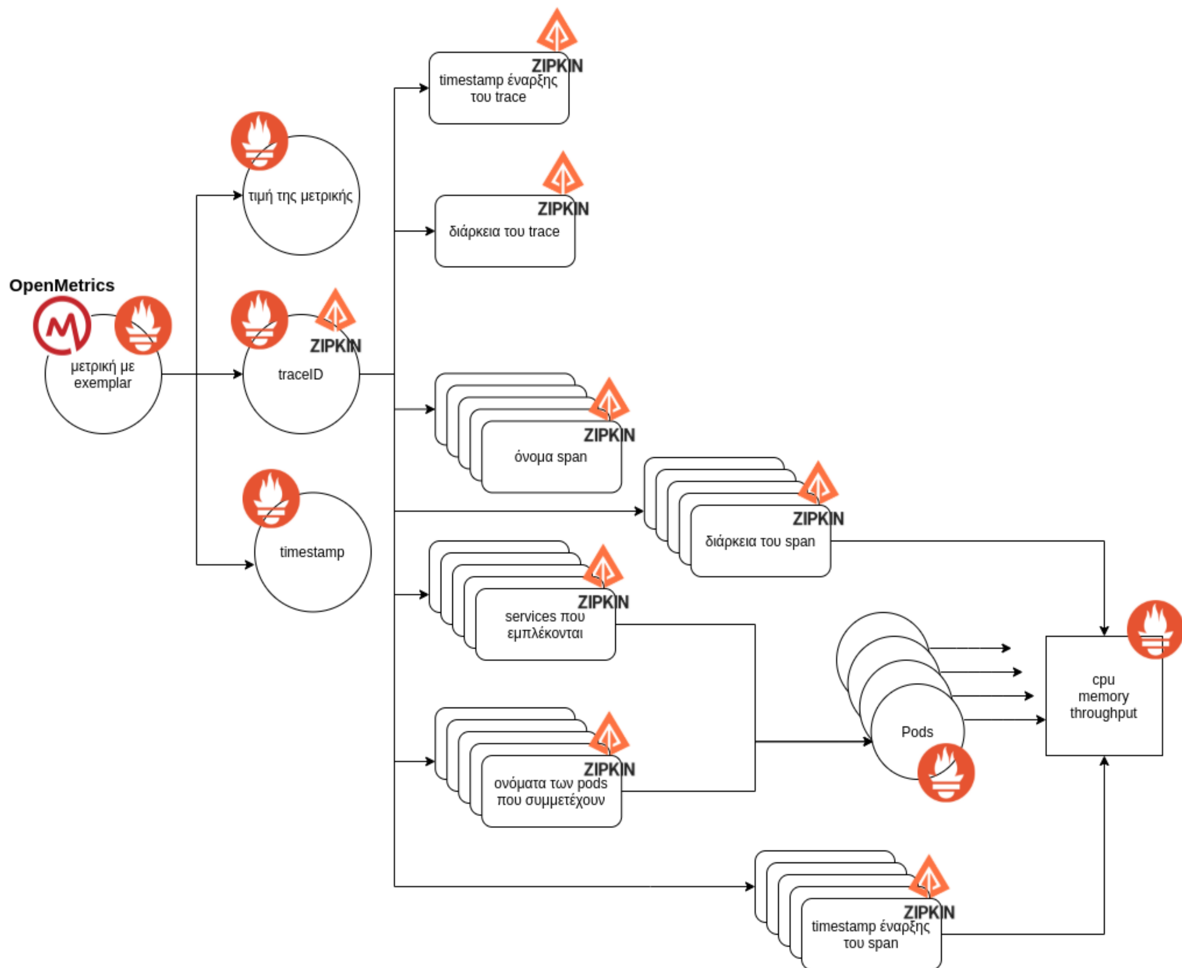
		`refused"` 2. δημιουργείται υπέρβαση του χρόνου αναμονής για απόκριση του Service b (timeout) λόγω υπερφόρτωσης και επιστρέφεται κωδικός 500 με μήνυμα `"Get "http://serviceb:8080/": net/http: request canceled (Client.Timeout exceeded while awaiting headers)"`	

Η πληροφορία που διαθέτουμε είναι τα δεδομένα ιχνηλάτησης (spans-traces) από το zipkin και οι μετρικές, με ή χωρίς exemplars, που συλλέγει το Prometheus από την εφαρμογή και το Kubernetes. Συλλέγοντας τα διαθέσιμα δεδομένα από τα αντίστοιχα APIs των εργαλείων σε JSON μορφή, μπορούμε να τα συγκεντρώσουμε σε ένα ενιαίο σχήμα [Εικόνα 16].



Εικόνα 16: Διάγραμμα συλλογής και ομαδοποίησης δεδομένων της εφαρμογής

Για την ομαδοποίηση των δεδομένων, συλλέγουμε τα exemplars τα οποία συνοδεύουν κάποιες ειδικά επιλεγμένες μετρικές από το Prometheus (“request_latency_seconds_total”, “errors_500_total”). Με βάση την τιμή του TraceID από το κάθε exemplar, μπορούμε να αναζητήσουμε τις πληροφορίες που είναι διαθέσιμες μέσω Zipkin για το trace στο οποίο αντιστοιχεί και τα spans τα οποία περιέχει. Για την χρονική διάρκεια που εξελίσσεται το κάθε span και trace, μέσω του Prometheus συγκεντρώνουμε μετρικές των Pods που φιλοξενούν τις μικροϋπηρεσίες από τις οποίες δημιουργήθηκαν τα spans [Εικόνα 17].

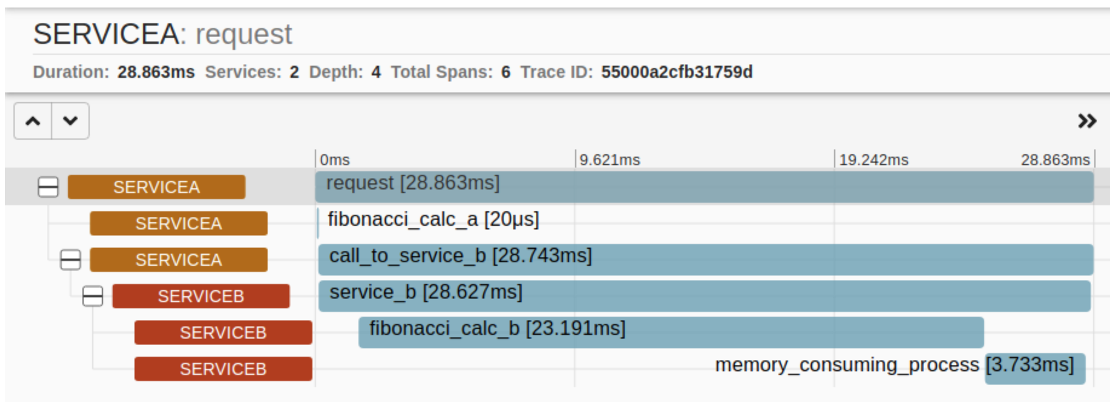


Εικόνα 17: Ομαδοποίηση δεδομένων ιχνηλάτησης και μετρικών

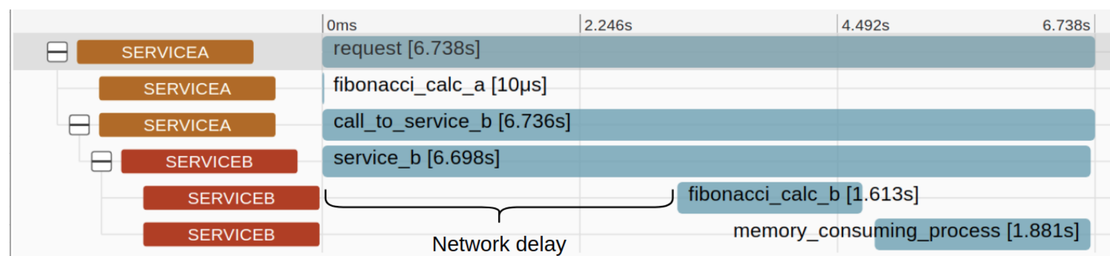
4.2 Αναπαράσταση δεδομένων

4.2.1 Zipkin

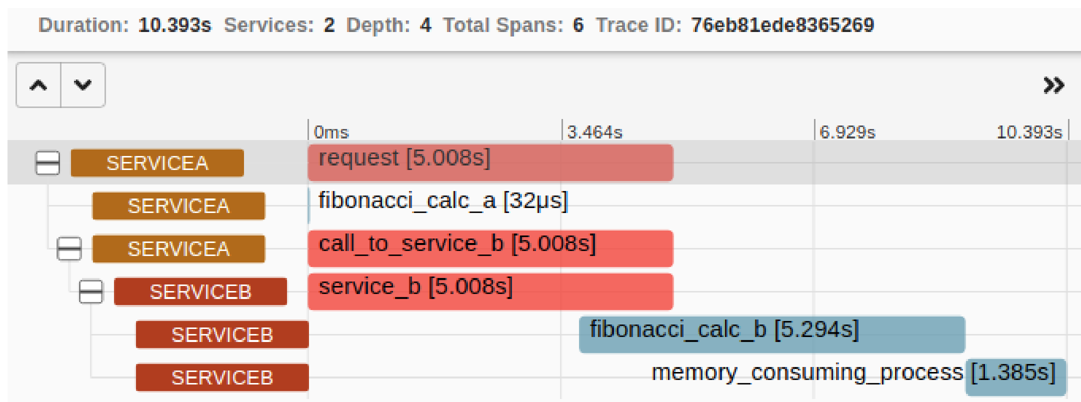
Για την παρακολούθηση της εφαρμογής, χρησιμοποιώντας τη βιβλιοθήκη του Zipkin, ορίζεται από την κάθε μικροϋπηρεσία να δημιουργούνται spans για κάθε ξεχωριστή λειτουργία που εκτελείται σε αυτή. Κάθε αίτημα που δέχεται η εφαρμογή αντιστοιχεί σε ένα trace το οποίο έχει 6 spans. Μέσα από το Zipkin UI παρουσιάζεται η δομή του κάθε trace, πληροφορίες για αυτό και τα spans του, αλλά και οι εξαρτήσεις των μικροϋπηρεσιών [Εικόνα 18-21].



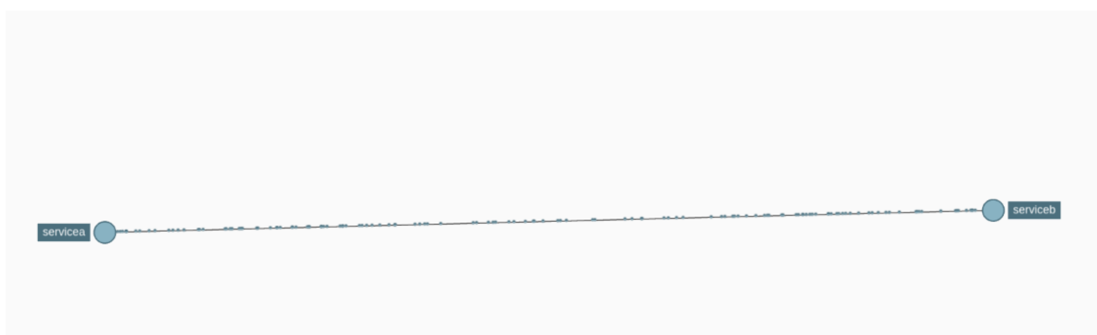
Εικόνα 18: Αναπαράσταση trace της πιλοτικής εφαρμογής στο Zipkin UI



Εικόνα 19: Αναπαράσταση trace της πιλοτική εφαρμογής με υψηλότερη καθυστέρηση δικτύου



Εικόνα 20: Αναπαράσταση trace με σφάλμα υπέρβασης χρονικού ορίου (timeout error) της πιλοτικής εφαρμογής στο Zipkin UI



Εικόνα 21: Διάγραμμα εξαρτήσεων μεταξύ των μικροϋπηρεσιών με ροή των ιχνηλατημένων αιτημάτων στο Zipkin UI

Για το κάθε span σημειώνεται η υπηρεσία στην οποία εκτελείται (SERVICE_A ή SERVICE_B), το όνομα της διαδικασίας που περιγράφει και η χρονική του διάρκεια με ακρίβεια μικροδευτερολέπτων [Εικόνα 18-20].
Το πρώτο span είναι το root span με όνομα “request” και αναπαριστά όλο το αίτημα από τον χρήστη προς την εφαρμογή μέχρι την απάντηση που θα του επιστραφεί. Συνεπώς η διάρκειά του ταυτίζεται με όλη τη διάρκεια του trace. Λόγω της επικοινωνίας των μικροϋπηρεσιών με σύγχρονα http αιτήματα, το root span τελειώνει αφού ολοκληρωθούν όλα τα spans.
Το πρώτο εμφωλευμένο span είναι το 1ο παιδί-span, με όνομα “fibonacci_calc_a” και αναπαριστά τον υπολογισμό που εκτελεί το service A για τον 15ο αριθμό της ακολουθίας Fibonacci.
Μετά το πέρας αυτού του span, καλείται το 2ο παιδί - span του “request”- span, το οποίο αναπαριστά την κλήση του service A προς το service B, με όνομα “call_to_service_b”. Αυτό το span διαρκεί από τη χρονική στιγμή που στάλθηκε το αίτημα από το ένα service στο άλλο μέχρι και την επιστροφή του, μετά την ολοκλήρωση του service b. Σαν παιδιά του έχει όλες τα spans που προέρχονται από το service b.
Συγκεκριμένα το “service_b” span, αναπαριστά όλες τις διαδικασίες που πραγματοποιήθηκαν στο service b από τη στιγμή που έλαβε το αίτημα από το service a, γι’αυτό έχει 2 παιδιά-spans. Το 1ο είναι το span “fibonacci_calc_b”, που αντιπροσωπεύει τον υπολογισμό του 30ου αριθμού της ακολουθίας Fibonacci και ακολουθεί το span “memory_consuming_process”, της διαδικασίας ανάγνωσης και εγγραφής σε αρχείο.

Μέσα από το zipkin API μπορούμε να λάβουμε την πληροφορία [13]:

- Για όλες τις μικροϋπηρεσίες που σχετίζονται τα spans:

  ```
  http://localhost:9411/api/v2/services
  ```

  Η πληροφορία επιστρέφεται σε μορφή λίστας:

  ```
  [ "service_a", "service_b" ]
  ```

- Για τις σχέσεις μεταξύ των μικροϋπηρεσιών:

  ```
  http://localhost:9411/api/v2/dependencies?endTs=<end_span_timestamp>&lookback=<span_duration>
  ```

  Για την εύρεση των μικροϋπηρεσιών που συμμετείχαν στη διάρκεια ενός span, η αναζήτηση πραγματοποιείται από τη χρονική στιγμή “endTs” = <χρονική στιγμή λήξης του span> και πίσω “lookback” = <διάρκεια του span> με βάση τη διάρκεια του span. Η πληροφορία επιστρέφεται σε αυτή τη μορφή:

```
[{
     "parent":    "service_a",
     "child":     "service_b",
     "callCount":  34,
     "errorCount": (opt)
  }]
```

Όπου “parent”: το όνομα της μικροϋπηρεσίας που πραγματοποιεί την κλήση (client) προς άλλη μικροϋπηρεσία, “child”: το όνομα της μικροϋπηρεσίας που δέχεται την κλήση (server), “callcount”: οι συνολικές καταγεγραμμένες κλήσεις από τη μια μικροϋπηρεσία σε άλλη και “errorCount”: οι συνολικές κλήσεις που καταγράφηκαν με σφάλμα (προεραιτικό).

- Για τα spans μέσα από αναζήτηση με βάση ένα traceID:

  `http://localhost:9411/api/v2/trace/<traceID>`

  Η πληροφορία επιστρέφεται σε μια λίστα με 7 spans σε JSON μορφή ακολουθώντας το SpanModel που περιγράφηκε στο Κεφάλαιο 2 (βλ.Παράρτημα):


```
[
{<"request"_span>},

{<"fibonacci_calc_a"_span>},

{<"call_to_service_b"_span>},

{"traceId":"63780b7667ab8719",
"parentId":"2c3536d941d51841",
"id":"07927da43830608a",
"kind":"CLIENT",
"name":"http/get",
"timestamp":1632482002668857,
"duration":115224,
"localEndpoint":{
       "serviceName":"servicea",
       "ipv4":"10.104.207.109",
       "port":8080},
"remoteEndpoint":{
       "serviceName":"serviceb",
       "ipv4":"10.110.63.151",
       "port":8080},
"tags":{
       "http.method":"GET",
       "http.path":"/",
       "http.response.size":"2",
       "servicea":"servicea-f89cb5f99-nqq9l"}
},

{"traceId":"63780b7667ab8719",
"parentId":"2c3536d941d51841",
"id":"07927da43830608a",
"kind":"SERVER",
"name":"service_b",
"timestamp":1632482002748481,
"duration":29456,
"localEndpoint":{
       "serviceName":"serviceb",
       "ipv4":"10.110.63.151",
```

```
        "port":8080},
"remoteEndpoint":{
        "ipv4":"172.17.0.5",
        "port":40372},
"tags":{
        "http.method":"GET",
        "http.path":"/",
        "serviceb":"serviceb-7cf797dc95-m9vk9"},
"shared":true
},

{<"fibonacci_calc_b"_span>},

{<"memory_consuming_process_span">}
]
```

Στο κάθε span, προσθέτουμε ως tag την πληροφορία της μικροϋπηρεσίας (“servicea” ή “serviceb”) στην οποία ανήκει με το αντίστοιχο όνομα του pod στο οποίο φιλοξενείται.

```
"tags":{
        "serviceb":"serviceb-7cf797dc95-m9vk9"
            or
        "servicea":"servicea-f89cb5f99-nqq9l"
}]
```

Στο root span προσθέτουμε το tag `"status"` με τον κωδικό κατάστασης `“200 OK”` και το tag `"response"` με την απάντηση της κλήσης προς την μικροϋπηρεσία b όταν αυτή είναι επιτυχημένη.

```
"tags":{
        "http.method":"GET",
        "http.path":"/",
        "http.status_code":"500",
        "response":"OK",
        "servicea":"servicea-f89cb5f99-nqq9l",
        "status":"200OK"
}
```

Αν η κλήση αυτή επέστρεψε με κάποιο σφάλμα τα tags του root span ακολουθούν αυτή τη δομή:

```
"tags":{
        "error":"500",
        "error_message":"Get \"http://serviceb:8080/\": net/http: request canceled (Client.Timeout exceeded while awaiting headers)",
        "http.method":"GET","http.path":"/",
        "http.status_code":"500",
```

```
        "servicea":"servicea-7c48df944f-lrx6j"
}
```

Σε περίπτωση σφάλματος υπέρβασης χρονικού ορίου (timeout) στο span `"call_to_service_b"` προστίθεται το tag :

`"error": "Get \"http://serviceb:8080/\": net/http: request canceled (Client.Timeout exceeded while awaiting headers)"`

Και στο span `"http/get"` προστίθεται το tag :

`"error": “net/http: request canceled”`

Ενδιαφέρον παρουσιάζει το πώς εκφράζεται η επικοινωνία μεταξύ των μικροϋπηρεσιών. Ρυθμίζοντας και προγραμματίζοντας την εφαρμογή κατάλληλα για ιχνηλάτηση, ορίζουμε το αίτημα προς το serviceb να καταγράφεται από ένα span με όνομα `"call_to_service_b"`, το οποίο έχει `"localEndpoint":"servicea"`, `"remoteEndpoint":"serviceb"` και πέρας τη χρονική στιγμή που έλαβε το service a την απάντηση. Ο διαχειριστής εξυπηρέτησης αιτημάτων (handler) του service b ξεκινά ένα span με τη λήψη του αιτήματος, το οποίο ονομάζεται `"service_b"` και διαρκεί μέχρι να ολοκληρωθούν όλες οι διαδικασίες και να επιστραφεί απάντηση στο service a. Το αναγνωριστικό (id) αυτού του `"service_b"` - span στα json δεδομένα εμφανίζεται σε 2 spans. Συγκεκριμένα το ένα span έχει `"name":"http/get"`, `"kind":"CLIENT"` με `"localEndpoint":"servicea"` και `"remoteEndpoint":"serviceb"`, ξεκινώντας τη στιγμή που στάλθηκε το αίτημα προς την μικροϋπηρεσία b. Το 2ο span με το ίδιο αναγνωριστικό (id) είναι αυτό με `"name":"service_b"`, `"kind":"SERVER"` , `"localEndpoint":"servicea"` και tag `"shared":true`. Ξεκινάει τη χρονική στιγμή που το serviceb έλαβε το αίτημα και διαρκεί μέχρι να επιστραφεί απάντηση στο servicea.

4.2.2 Prometheus

Το API του Prometheus [36] επιστρέφει απαντήσεις σε JSON μορφή, με τα δεδομένα που ζητήθηκαν στο πεδίο “data”, αν το αίτημα ήταν επιτυχημένο (“status”: “success”).

```
{
 "status":"success"|"error",
 "data":<data>,

 // Only set if status is "error". The data field may still hold
 // additional data.
 "errorType":"<string>",
 "error":"<string>",

 // Only if there were warnings while executing the request.
 // There will still be data in the data field.
 "warnings":["<string>"]
}
```

Με το endpoint `/api/v1/query_range` μπορούμε να πάρουμε αποθηκευμένα δεδομένα χρονοσειρών που αφορούν μια συγκεκριμένη χρονική διάρκεια. Η απάντηση που επιστρέφεται έχει την ακόλουθη JSON δομή :

```
{
       "status":"success",
       "data":{
             "resultType":"matrix",
             "result":[
                         {
                           "metric": { "<label_name>": "<label_value>", ... },
                           "values": [ [ <unix_time>, "<metric_value>" ], ... ]
                         },
                         ...
             ]
       }
}
```

Στο πεδίο “values” επιστρέφονται σε λίστα τα ζευγάρια [<χρονικής στιγμής>,<τιμής μετρικής>] ([<unix_time>, <metric_vaue>]) για το εύρος <χρονική στιγμή έναρξης> έως <χρονική στιγμή λήξης> ανά <χρονικό βήμα> ακρίβειας όπως αυτά ορίζονται στις παραμέτρους της ερώτησης:

```
http://localhost:9090/api/v1/query_range?query=<ονομα_μετρικής>&start=<χρονική_
στιγμή_έναρξης>&end=<χρονική_στιγμή_λήξης>&step=<χρονικό_βήμα>
```

Στην πιλοτική εφαρμογή το χρησιμοποιούμε για να συλλέξουμε μετρικές ανά χρονικό βήμα (step) στο χρονικό διάστημα εκτέλεσης ενός span ([start=χρονική_στιγμή_έναρξης_span, end=χρονική_στιγμή_λήξης_span]). Συγκεκριμένα τις μετρικές που αφορούν :

- Το ρυθμό λήψης αιτημάτων:
  ```
  rate(http_requests_total[10s])
  ```
- Τη χρήση cpu από το κάθε pod στο οποίο εκτελείται το εκάστοτε span:
  ```
  rate(container_cpu_usage_seconds{container=~"<service>",id"/docker.*"}[1
  m])
  ```
- Τη χρήση μνήμης από το κάθε pod στο οποίο εκτελείται το εκάστοτε span:
  ```
  container_memory_working_set_bytes{container=~"<service>",id"/docker.*"}
  ```

Στο Prometheus, τα χρονικά δεδομένα (timestamps) αποθηκεύονται με ακρίβεια χιλιοστών του δευτερολέπτου (millisec) - σε αντίθεση με το Zipkin που τα αποθηκεύει με ακρίβεια μικροδευτερολέπτων (microsec).

Η μέγιστη ακρίβεια στο ερώτημα `query_range` είναι 11000 ζευγάρια για την κάθε μετρική. Δεδομένου ότι ισχύει:

$$πλήθος_ζευγαριών[<χρονικής_στιγμής>,<τιμής_μετρικής>] = \frac{διάρκεια_span}{χρονικό_βήμα}$$

επιλέγουμε την ελάχιστη τιμή του χρονικού βήματος (step = 1 ms, 10 ms, 100 ms, …) που επιτρέπει τη μέγιστη δυνατή ακρίβεια εντός του ορίου των 11000 ζευγαριών.

Με το endpoint `/api/v1/query_exemplars` μπορούμε να πάρουμε μια λίστα από exemplars που έχουν συλλεχθεί μαζί με μια μετρική. Μπορεί επίσης να ορισθεί συγκεκριμένο χρονικό εύρος αναζήτησης με παραμέτρους `start` και `end` :

```
http://localhost:9090/api/v1/query_exemplars?query=<metric_name_with_exemplars>
&start=<begin_timestamp>&end=<end_timestamp>
```

Η απάντηση που επιστρέφεται έχει την ακόλουθη JSON δομή:

```
{
      "status":"success",
      "data":[
              {
                "seriesLabels": {
                     "__name__": <metric_name_with_exemplars>,
                     "instance": "localhost:8080",
                     "io_kompose_service":<service_name>,
                     "job": "kubernetes-service-endpoints",
                     "kubernetes_name": "<service_name>,
                     "kubernetes_namespace": <namespace>
                 },
                "exemplars": [
                               {
                                 "labels": {
                                      "traceID": <traceID>
                                  },
                                 "value": <metric_value>,
                                 "timestamp": <timestamp in seconds>,
                               },
                              ...
                             ]
              },
              ...
            ]
}
```

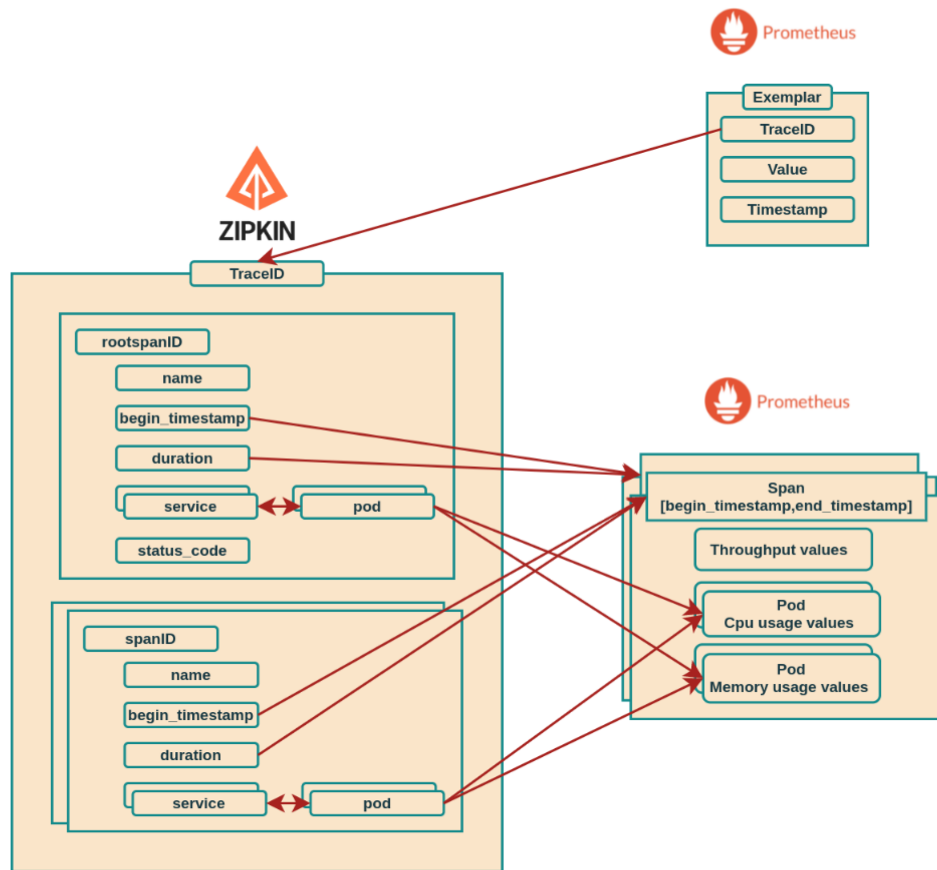
Στην πιλοτική εφαρμογή, οι μετρικές με exemplars οι οποίες συλλέγονται είναι :

- ‘request_latency_seconds_total’
- ‘errors_500_total’

4.3 Ενοποίηση δεδομένων

Τα exemplars που συλλέγουμε από την εφαρμογή συνοδεύουν τις μετρικές "request_latency_seconds_total" και "erros_500_total" που αθροίζουν το χρόνο απόκρισης συστήματος σε ένα αίτημα, επιτυχές ή με σφάλμα, αντίστοιχα. Συνεπώς η τιμή "value" του κάθε exemplar είναι ο χρόνος που χρειάστηκε η εφαρμογή για να εξυπηρετήσει το αντίστοιχο αίτημα. Το "timestamp" του exemplar είναι η χρονική στιγμή σύλληψής του, η οποία στην πιλοτική εφαρμογή ταυτίζεται με τη χρονική στιγμή λήξης του trace. Το σχήμα αναπαράστασης της ενοποιημένης πληροφορίας [Εικόνα 22] θα ακολουθεί την παρακάτω JSON δομή :

```
[{
      "traceID": <exemplar_trace_id>,
      "timestamp_end(sec)": <exemplar_timestamp>,
      "latency(sec)": <exemplar_value>,
      "status_code": "200 OK" or "500",
      "root_span": {
              "span_id": <span_id>,
              "name": <span_name>,
              "timestamp(micro)": <begin_timestamp>,
              "span_duration(micro)": <span_duration>,
              "throughput_avg": <average_throughput>,
              "pods_metrics_avg": [
                    { "pod": <service_name>,
                      "cpu": <cpu_usage_rate>,
                      "memory": <memory_usage>
                    },
                    ...
              ]
      },
      "spans": [
              {"span_id": <span_id>,
               "name":<span_name>,
               "timestamp(micro)":<begin_timestamp>,
               "span_duration(micro)": <span_duration>,
               "throughput_avg": <average_throughput>
               "pods_metrics_avg": [
                    { "pod": <service_name>,
                      "cpu": <cpu_usage_rate>,
                      "memory": <memory_usage>
                    },
                    ...
              ]
  },
  …
]
```



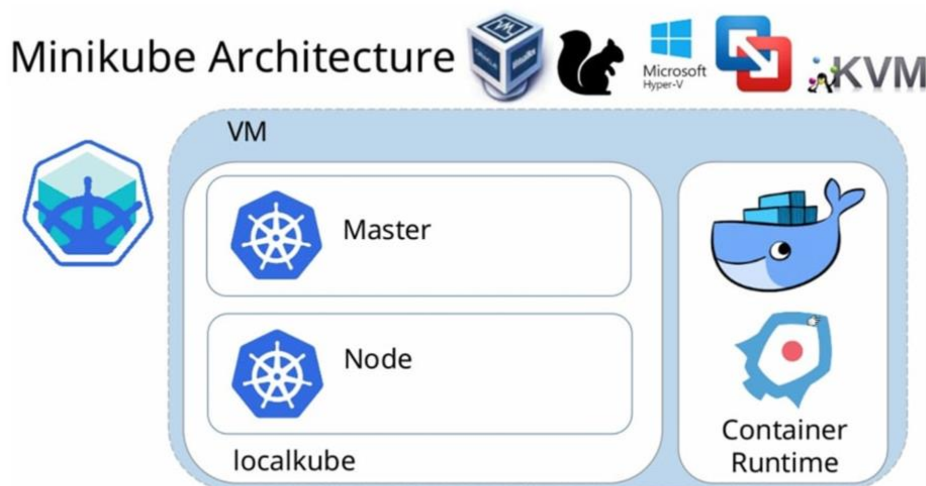
Εικόνα 22: Ενοποίηση δεδομένων από διαφορετικές πηγές

Τα "traceID", "timestamp_end(sec)", "latency(sec)" αναπαριστούν την πληροφορία που συλλέγουμε από τα exemplars.
Με βάση αυτό το traceID αναζητούμε στο zipkin τα spans από τα οποία αποτελείται το trace. Κάθε trace περιλαμβάνει το root span ("root_span"), που στην περίπτωσή μας αναπαριστά όλο το αίτημα και έχει διάρκεια ίση με όλο το trace, και τα επιμέρους spans ("spans"), τα οποία συγκεντρώνονται σε μια λίστα. Για το κάθε span η πληροφορία που συλλέγεται από το zipkin είναι το αναγνωριστικό του ("span_id"), το όνομά του ("name"), η χρονική στιγμή έναρξής του ("timestamp(micro)") και η διάρκειά του ("span_duration(micro)"). Γνωρίζοντας τη χρονική στιγμή έναρξης και τη διάρκεια ενός span, μπορούμε να υπολογίσουμε και τη χρονική στιγμή λήξης του.
Έτσι ορίζοντας στο Prometheus ένα ορισμένο χρονικό εύρος, συλλέγουμε τις τιμές μετρικών για αυτό το διάστημα και υπολογίζουμε το μέσο όρο τους ως αντιπροσωπευτική τιμή. Συγκεκριμένα, επισυνάπτουμε ως πληροφορία στο span το μέσο ρυθμό αιτημάτων που δέχεται η εφαρμογή ("throughput_avg") καθώς αυτό εξελίσσεται. Επίσης από το Zipkin, ανάλογα με τα tags "localEndpoint", "remoteEndpoint", "<service_name>": <pod_name> και τις σχέσεις μεταξύ των services (`/api/v2/dependencies`), προσδιορίζουμε ποια services συμμετείχαν στη διαδικασία που περιγράφει το span και σε ποια pod συγκεκριμένα. Με βάση αυτά αναζητούμε στο Prometheus, για το χρονικό διάστημα που εκτυλίσσεται το span, τις τιμές των cpu και memory των pod αυτών και υπολογίζουμε τη μέση τιμή τους ("pods_metrics_avg": "cpu", "pods_metrics_avg": "memory").
Τέλος από το root span και τα tags του "status" και "error" προσδιορίζουμε το αίτημα ως επιτυχές (status 200 OK) ή με κάποιο σφάλμα (error 500) και το αναγράφουμε ως πληροφορία για όλο το trace στο "status_code".

Κεφάλαιο 5: Υποδομή δοκιμών και αποτελέσματα

5.1 Υποδομή υπολογιστικού νέφους

Η πιλοτική εφαρμογή δοκιμάστηκε σε εγκατάσταση της πλατφόρμας Kubernetes τοπικά σε Linux, έκδοσης Ubuntu 20.04.2 LTS, χρησιμοποιώντας το Minikube. Το Minikube είναι μια έκδοση του Kubernetes για τοπική εκτέλεση σε υπολογιστή, που λειτουργεί ως μια εικονική μηχανή (VM), μέσω Virtual Box [37,38]. Με το minikube δημιουργείται ένα cluster με ένα κόμβο (single-node cluster) σε ρόλο master και worker, με προεγκατεστημένο περιβάλλον για docker ώστε να μπορούν να τρέχουν containers μέσα σε αυτό [Εικόνα 23,24].



Εικόνα 23: Αρχιτεκτονική και Δομικά στοιχεία του minikube[10]

```
😄  minikube v1.14.2 on Ubuntu 20.04
✨  Using the docker driver based on existing profile
👍  Starting control plane node minikube in cluster minikube
🔄  Restarting existing docker container for "minikube" ...
🐳  Preparing Kubernetes v1.19.2 on Docker 19.03.8 ...
🔎  Verifying Kubernetes components...
🌟  Enabled addons: default-storageclass, storage-provisioner, metrics-server,
 dashboard
🏄  Done! kubectl is now configured to use "minikube" by default
```

Εικόνα 24: Εκκίνηση minikube

Κάθε μικροϋπηρεσία της εφαρμογής εκτελείται σε ένα docker container μέσα στο δικό της Pod. Δυνητικά μπορούν να εκτελούνται πολλά Pods ταυτόχρονα, με το καθένα τα τρέχει την ίδια docker εικόνα που αντιπροσωπεύει την μικροϋπηρεσία.
Η εκτέλεση στο kubernetes γίνεται μέσω:

- Deployment (Εκτελεστή), το οποίο περιγράφεται στο αρχείο <deployment>.yaml και αναλαμβάνει τη δημιουργία και τη διαχείριση των Pods στα οποία θα τρέχει η κάθε μικροϋπηρεσία, με δυνατότητα κλιμάκωσης και διαρκή έλεγχο της υγείας τους [39] .

[10] https://dev.to/elkhatibomar/030-kubernetes-minikube-4ofa

- Service (υπηρεσία), η οποία περιγράφεται σε αρχείο <service>.yaml, επιτρέποντας στα Pods και συνεπώς στις μικροϋπηρεσίες να επικοινωνούν μεταξύ τους και με εξωτερικές πηγές [40].

Πέρα από την εφαρμογή σε containers, στον κόμβο του Kubernetes θα στηθεί και το Prometheus [41] ώστε να μπορούμε να το χρησιμοποιήσουμε για συλλογή μετρικών. Ορίζονται τα ακόλουθα:

1. (Προεραιτικά) Ορίζουμε namespace (χώρο ονομάτων) για τα στοιχεία (components) που θα αφορούν την παρακολούθηση της εφαρμογής
2. ClusterRole, με άδειες για πρόσβαση στις διαθέσιμες μετρικές από τα Pods
3. Config Map (χάρτη παραμετροποίησης), με παραμετροποιήσεις για τη συλλογή μετρικών, εντοπισμό των υπηρεσιών που τις παρέχουν αλλά και την αποθήκευσή τους
4. Τον εκτελεστή (deployment) που θα δημιουργήσει ένα Pod για να στήσει το Prometheus με βάση την docker εικόνα `prom/prometheus:v2.28.0`, η οποία υποστηρίζει τα exemplars. Για την ενεργοποίηση της δυνατότητας συλλογής και αποθήκευσης exemplars χρησιμοποιούμε την παράμετρο (flag) `--enable-feature=exemplar-storage`
5. Το Prometheus διατίθεται ως υπηρεσία, μέσω του service, στην πόρτα 9090

Το Zipkin στήνεται επίσης στο Kubernetes ώστε να χρησιμοποιηθεί ως το εργαλείο κατανεμημένης παρακολούθησης της εφαρμογής. Τα trace δεδομένα που θα συλλεχθούν αποθηκεύονται στην προσωρινή μνήμη (in memory) στο Pod του Zipkin.

1. Μέσω του εκτελεστή (deployment) για το Zipkin, δημιουργείται ένα Pod με ένα container που εκτελεί την `openzipkin/zipkin` εικόνα του
2. Το Prometheus διατίθεται ως υπηρεσία, μέσω του service, στην πόρτα 9411

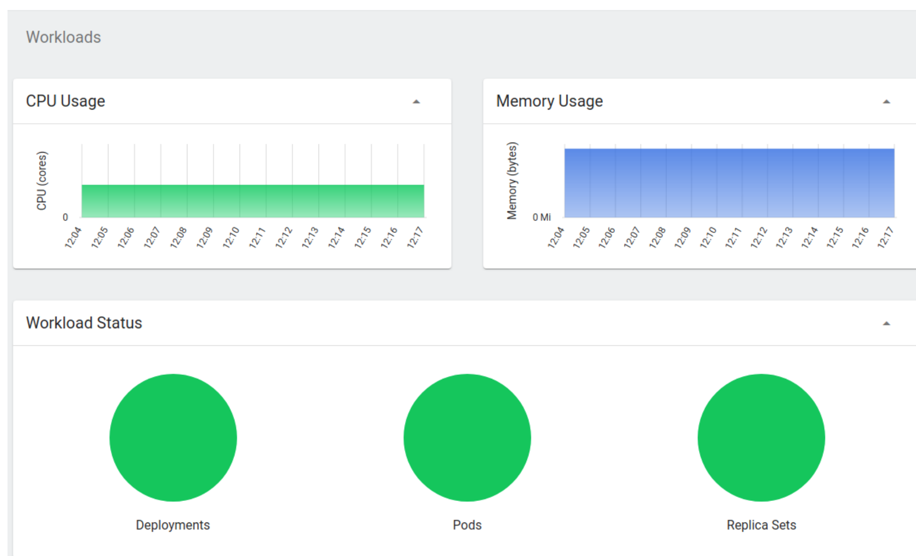
Στο κόμβο του cluster που έχει στηθεί μέσω minikube υπάρχουν συνολικά τα ακόλουθα Pods:

1. Ένα Pod για την μικροϋπηρεσία A με όνομα “servicea”, με 1 container που τρέχει την εικόνα `“christinamaria/service_a:latest”`
2. Ένα Pod για την μικροϋπηρεσία B με όνομα “serviceb”, με 1 container που τρέχει την εικόνα `“christinamaria/service_b:latest”`
3. Ένα Pod για το Zipkin με όνομα “zipkin”, με 1 container που τρέχει την εικόνα του zipkin `“openzipkin/zipkin:latest”`
4. Ένα Pod για τον Prometheus με όνομα “prometheus-deployment” , με 1 container που τρέχει την εικόνα του Prometheus που υποστηρίζει και exemplars `"prom/prometheus:v2.28.0"`
5. Ένα Pod για kube-state-metrics, με 1 container που τρέχει την εικόνα `"k8s.gcr.io/kube-state-metrics/kube-state-metrics:v2.2.0"`
6. Ένα Pod για metrics-server, για έκθεση μετρικών (cpu, memory) στο API, με 1 container που τρέχει την εικόνα `"k8s.gcr.io/metrics-server-amd64:v0.2.1"`

Για τη διαχείριση του kubernetes cluster από τερματικό χρησιμοποιείται η εντολή `kubectl` [42], ενώ για πρόσβαση στο Dashboard του Kubernetes cluster [Εικόνα 24-28], χρησιμοποιείται η εντολή :

```
minikube dashboard
```

Στο default namespace παρουσιάζεται μια γενική κατάσταση της υγείας των deployments, pods και αντιγράφων τους που εκτελούνται, καθώς και γραφήματα για τη συνολική χρήση cpu και μνήμης του κόμβου από τα pods που φιλοξενεί [Εικόνα 25].



Εικόνα 25: Dashboard του minikube με ένδειξη υγιούς κατάστασης των deployments, pods και replica sets και χρήσης των μετρικών CPU Usage και Memory Usage στο default namespace

Η κατάσταση των deployments που έχουν εκτελεστεί σε αυτό το namespace φαίνεται αναλυτικότερα στον πίνακα "Deployments". Για τα "zipkin", "servicea" και "serviceb" δίνονται πληροφορίες για την εικόνα που χρησιμοποιεί το κάθε ένα για την εκτέλεση των containers και το πλήθος των ενεργών τους pods (1/1) [Εικόνα 26] .

Deployments

Name	Labels	Pods	Created ↑	Images
zipkin	io.kompose.service: zipkin	1 / 1	28 minutes ago	openzipkin/zipkin:latest
serviceb	io.kompose.service: serviceb	1 / 1	28 minutes ago	christinamaria/service_b:latest
servicea	io.kompose.service: servicea	1 / 1	28 minutes ago	christinamaria/service_a:latest

1 – 3 of 3

Εικόνα 26: Dashboard του minikube με τα 3 deployments στο default namespace

Στον πίνακα Pods παρουσιάζονται πληροφορίες για τα pods, τον κόμβο στον οποίο ανήκουν (node: minikube), την κατάσταση λειτουργία τους (status: running), το πλήθος των επανεκκινήσεών τους (restarts) αλλά και τη χρήση τους σε υπολογιστικούς πόρους (cpu usage, memory usage) [Εικόνα 27].

Pods

	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
✔	zipkin-6b99d6d78d-7whwh	io.kompose.service: zipkin pod-template-hash: 6b99d6d78d	minikube	Running	0	4.00m	167.50Mi	28 minutes ago
✔	servicea-f89cb5f99-nqq9l	io.kompose.service: servicea pod-template-hash: f89cb5f99	minikube	Running	0	4.00m	14.47Mi	28 minutes ago
✔	serviceb-7cf797dc95-m9vk9	io.kompose.service: serviceb pod-template-hash: 7cf797dc95	minikube	Running	0	3.00m	2.33Mi	28 minutes ago

1 – 3 of 3

Εικόνα 27: Dashboard του minikube με 3 pods που εκτελούνται στο default namespace

Ο ρόλος των ReplicaSets, τα οποία ελέγχονται από το αντίστοιχο deployment, είναι να διασφαλίζουν σταθερό το ζητούμενο πλήθος από Pods ([Εικόνα 28] Pods: 1/1) οποιαδήποτε χρονική στιγμή [43]. Ο εντοπισμός των pods για τα οποία είναι υπεύθυνο το κάθε ReplicaSet γίνεται με την χρήση των ετικετών (labels) που αυτά φέρουν.

Replica Sets

	Name	Labels	Pods	Created ↑	Images
✔	zipkin-6b99d6d78d	io.kompose.service: zipkin pod-template-hash: 6b99d6d78d	1 / 1	29 minutes ago	openzipkin/zipkin:latest
✔	servicea-f89cb5f99	io.kompose.service: servicea pod-template-hash: f89cb5f99	1 / 1	29 minutes ago	christinamaria/service_a:latest
✔	serviceb-7cf797dc95	io.kompose.service: serviceb pod-template-hash: 7cf797dc95	1 / 1	29 minutes ago	christinamaria/service_b:latest

1 – 3 of 3

Εικόνα 28: Dashboard του minikube με τα 3 Replica Sets του default namespace

Το κάθε Pod, κατά τη δημιουργία του, αποκτά μια IP διεύθυνση, η οποία είναι γνωστή μόνο εσωτερικά του cluster (Cluster IP). Για τη δικτύωση του Pod χρησιμοποιείται το Service [Εικόνα 29], το οποίο αντιστοιχίζεται με το εσωτερικό τελικό σημείο επικοινωνίας (internal endpoint) που εκθέτει το κάθε Pod (όνομα - πόρτα).

Discovery and Load Balancing

Services

	Name	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created ↑
✔	zipkin	io.kompose.service: zipkin	10.99.44.83	zipkin:9411 TCP zipkin:0 TCP	-	29 minutes ago
✔	serviceb	io.kompose.service: serviceb	10.110.63.151	serviceb:8080 TCP serviceb:0 TCP	-	29 minutes ago
✔	servicea	io.kompose.service: servicea	10.104.207.109	servicea:8080 TCP servicea:0 TCP	-	29 minutes ago
✔	kubernetes	component: apiserver provider: kubernetes	10.96.0.1	kubernetes:443 TCP kubernetes:0 TCP	-	11 hours ago

1 – 4 of 4

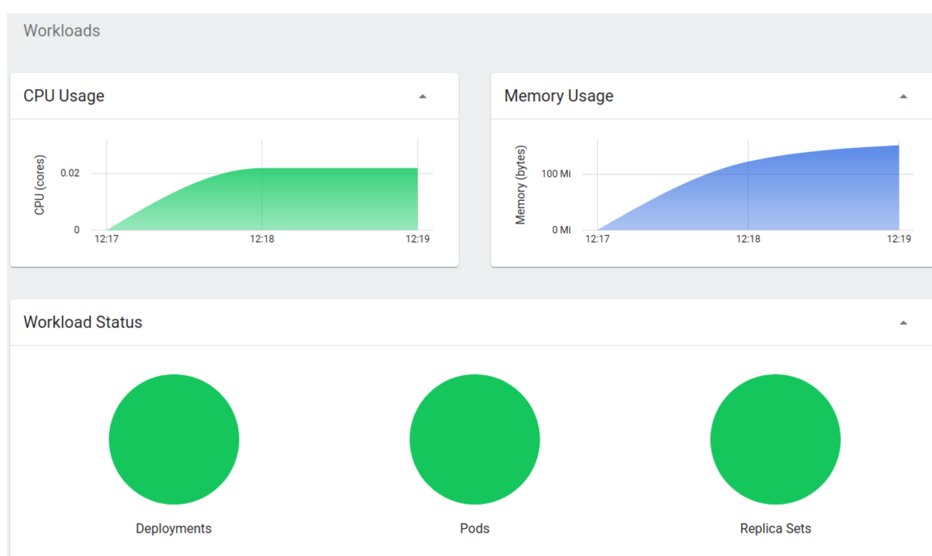
Εικόνα 29: Dashboard του minikube με service για το κάθε deployment και το kubernetes api

Με την επιλογή `port-forward` της `kubectl` εντολής η πόρτα στην οποία ακούει εσωτερικά στο cluster το κάθε Pod αντιστοιχίζεται σε μια πόρτα του τοπικού μηχανήματος όπου εκτελείται το minikube (localhost:port) [44].

```
kubectl port-forward <όνομα του service του pod>  <πόρτα του τοπικού μηχανήματος>:<πόρτα στην οποία δέχεται κίνηση το pod>
```

- Με την εντολή `kubectl port-forward svc/servicea 8080:8080` το service a και συνεπώς η εφαρμογή είναι διαθέσιμη στο `http://localhost:8080/`
- Με την εντολή `kubectl port-forward svc/zipkin 9411:9411` έχουμε πρόσβαση από το `http://localhost:9411/` στο web ui του zipkin
- Με την εντολή `kubectl port-forward -n monitoring svc/prometheus 9090`:9090 έχουμε πρόσβαση από το `http://localhost:9090/` στο web ui του prometheus

Αντίστοιχα στο monitoring namespace, όπου έχει στηθεί το Prometheus, λαμβάνουμε μια γενική εικόνα για την κατάσταση των Deployments με τα Pods τους και τη χρήση υπολογιστικών πόρων (cpu,μνήμη) [Εικόνα 30]. Στη συνέχεια παρουσιάζονται αναλυτικότερα τα Deployments με τα αντίστοιχα Pods, Replica Sets και Services [Εικόνα 31, 32], καθώς και οι χάρτες παραμετροποίησης (Config Maps) [Εικόνα 33] που χρειάστηκαν για την εκτέλεση του Prometheus.



Εικόνα 30: Dashboard του minikube με ένδειξη υγιούς κατάστασης των deployments, pods και replica sets και χρήσης των μετρικών CPU usage και Memory Usage στο monitoring namespace

Deployments

	Name	Labels	Pods	Created ↑	Images
✓	prometheus-deployment	app: prometheus-server	1 / 1	5 minutes ago	prom/prometheus:v2.28.0

1 – 1 of 1

Pods

	Name	Labels	Node	Status	Restarts	CPU Usage (cores)	Memory Usage (bytes)	Created ↑
✓	prometheus-deployment-775bd7bf9-6dwqt	app: prometheus-server pod-template-hash: 775bd7bf9	minikube	Running	0	23.00m	194.34Mi	5 minutes ago

1 – 1 of 1

Replica Sets

	Name	Labels	Pods	Created ↑	Images
✓	prometheus-deployment-775bd7bf9	app: prometheus-server pod-template-hash: 775bd7bf9	1 / 1	5 minutes ago	prom/prometheus:v2.28.0

1 – 1 of 1

Εικόνα 31: Dashboard του minikube με πληροφορίες για το Deployment, Pod και Replica Set του Prometheus στο monitoring namespace

Discovery and Load Balancing

Services

	Name	Labels	Cluster IP	Internal Endpoints	External Endpoints	Created ↑
✓	prometheus	app: prometheus-server	10.97.4.26	prometheus.monitor TCP prometheus.monitor TCP	-	5 minutes ago

1 – 1 of 1

Εικόνα 32: Dashboard του minikube με πληροφορίες για το Service του Prometheus στο monitoring namespace

Config and Storage

Config Maps

Name	Labels	Created ↑
prometheus-server-conf	name: prometheus-server-conf	5 minutes ago

1 – 1 of 1

Εικόνα 33: Dashboard του minikube με πληροφορίες για το χάρτη παραμετροποίησης (Config Map) του Prometheus στο monitoring namespace

5.2 Φορτίο Δοκιμών

Για την παρακολούθηση της συμπεριφοράς της εφαρμογής υπό διάφορες συνθήκες πραγματοποιήθηκαν φορτία δοκιμών με το ανοιχτού κώδικα εργαλείο “Vegeta” [45]. Με το Vegeta στέλνονται HTTP αιτήματα στην εφαρμογή με σταθερό ρυθμό για συγκεκριμένο χρονικό διάστημα, όπως αυτά ορίζονται ως παράμετροι. Τα αποτελέσματα της δοκιμής μπορούν να ληφθούν από το Vegeta είτε σε μορφή json είτε σε μορφή αρχείου txt.

Για το φορτίο δοκιμής τα αιτήματα στέλνονται στο http://localhost:8080/ όπου ακούει η εφαρμογή (`"GET http://localhost:8080/"`). Η διάρκεια της δοκιμής ορίζεται στην παράμετρο `-duration= <διάρκεια>` και ο ρυθμός αποστολής αιτημάτων στην παράμετρο `-rate=<αιτήματα/δευτερόλεπτο>`. Τέλος το είδος της αναφοράς για τα αποτελέσματα από τη δοκιμή ορίζεται στο `vegeta report --type = <text / json>`.

```
echo "GET http://localhost:8080/" | vegeta attack -
duration=<duration><s/ms/m> -rate=<rate> | vegeta report --
type=<text/json>
```

Πραγματοποιείται φορτίο δοκιμής συνολικής διάρκειας 5 λεπτών στο `http://localhost:8080/` όπου ακούει η εφαρμογή , με τους εξής ρυθμούς :

- 10 αιτήματα/δευτερόλεπτο για 60 δευτερόλεπτα
- 20 αιτήματα/δευτερόλεπτο για 60 δευτερόλεπτα
- 30 αιτήματα/δευτερόλεπτο για 60 δευτερόλεπτα
- 20 αιτήματα/δευτερόλεπτο για 60 δευτερόλεπτα
- 10 αιτήματα/δευτερόλεπτο για 60 δευτερόλεπτα

Η αναφορά από το vegeta :

```
Requests      [total, rate, throughput]         600, 10.02, 10.01
Duration      [total, attack, wait]             59.917s, 59.9s, 17.576ms
Latencies     [min, mean, 50, 90, 95, 99, max]  8.578ms, 29.175ms, 30.124ms,
31.781ms, 32.296ms, 40.873ms, 265.07ms
Bytes In      [total, mean]                     21600, 36.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:600
Error Set:
Requests      [total, rate, throughput]         1200, 20.02, 20.01
Duration      [total, attack, wait]             59.977s, 59.95s, 27.375ms
Latencies     [min, mean, 50, 90, 95, 99, max]  8.359ms, 24.008ms, 26.279ms,
31.534ms, 32.377ms, 35.196ms, 86.772ms
Bytes In      [total, mean]                     43200, 36.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:1200
Error Set:
Requests      [total, rate, throughput]         1800, 30.02, 7.45
Duration      [total, attack, wait]             1m5s, 59.967s, 5.115s
Latencies     [min, mean, 50, 90, 95, 99, max]  8.497ms, 4.125s, 5.062s, 5.178s,
5.268s, 5.753s, 6.102s
Bytes In      [total, mean]                     153857, 85.48
Bytes Out     [total, mean]                     0, 0.00
```

```
Success       [ratio]                           26.94%
Status Codes  [code:count]                      200:485  500:1315
Error Set:
500 Internal Server Error
Requests      [total, rate, throughput]         1200, 20.02, 12.99
Duration      [total, attack, wait]             59.966s, 59.95s, 16.185ms
Latencies     [min, mean, 50, 90, 95, 99, max]  8.388ms, 2.544s, 2.494s, 5.218s,
5.388s, 5.656s, 5.92s
Bytes In      [total, mean]                     71342, 59.45
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           64.92%
Status Codes  [code:count]                      200:779  500:421
Error Set:
500 Internal Server Error
Requests      [total, rate, throughput]         600, 10.02, 10.01
Duration      [total, attack, wait]             59.93s, 59.9s, 30.304ms
Latencies     [min, mean, 50, 90, 95, 99, max]  8.556ms, 29.071ms, 30.657ms,
32.773ms, 33.875ms, 37.75ms, 131.949ms
Bytes In      [total, mean]                     21600, 36.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:600
Error Set:
```

Όπως φαίνεται και από τα αποτελέσματα του Vegeta υπάρχουν αιτήματα με κωδικό κατάστασης (Status Code) 200 τα οποία ολοκληρώθηκαν με επιτυχία και άλλα τα οποία εμφάνισαν κωδικό κατάστασης “500: Internal Server Error”. Σε αυτά τα αιτήματα δημιουργήθηκε σφάλμα:

```
Get "http://serviceb:8080/": net/http: request canceled (Client.Timeout
exceeded while awaiting headers)
```

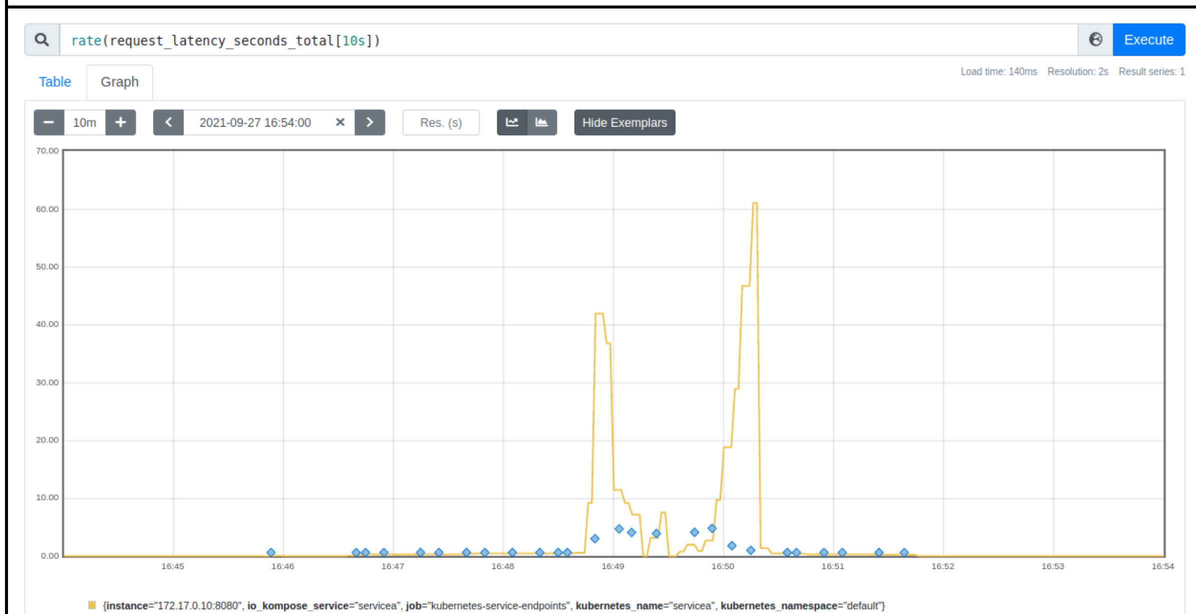
δηλαδή υπέρβαση χρονικού ορίου (timeout) κατά την κλήση από την μικροϋπηρεσία a στην μικροϋπηρεσία b. Το χρονικό όριο αναμονής λήψης απάντησης έχει οριστεί στα 5 δευτερόλεπτα.

5.3 Αποτελέσματα

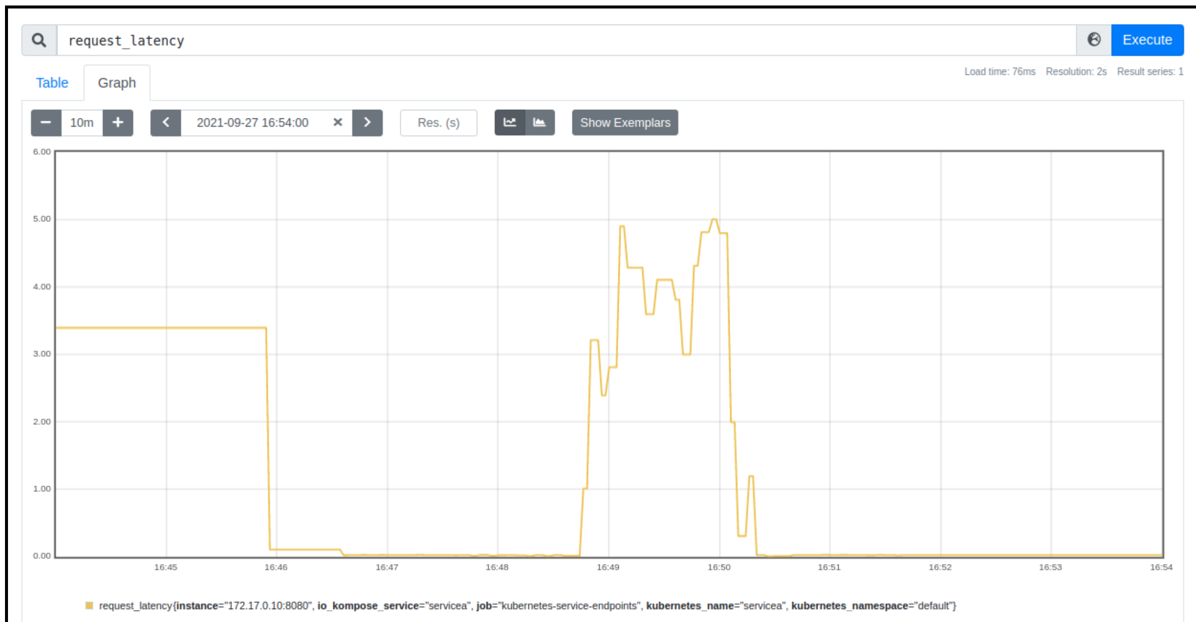
5.3.1 Αποτελέσματα εργαλείων παρακολούθησης

Από το WEB UI του Prometheus μπορούμε να λάβουμε τα παρακάτω γραφήματα με βάση τη δοκιμή φορτίου που πραγματοποιήσαμε.

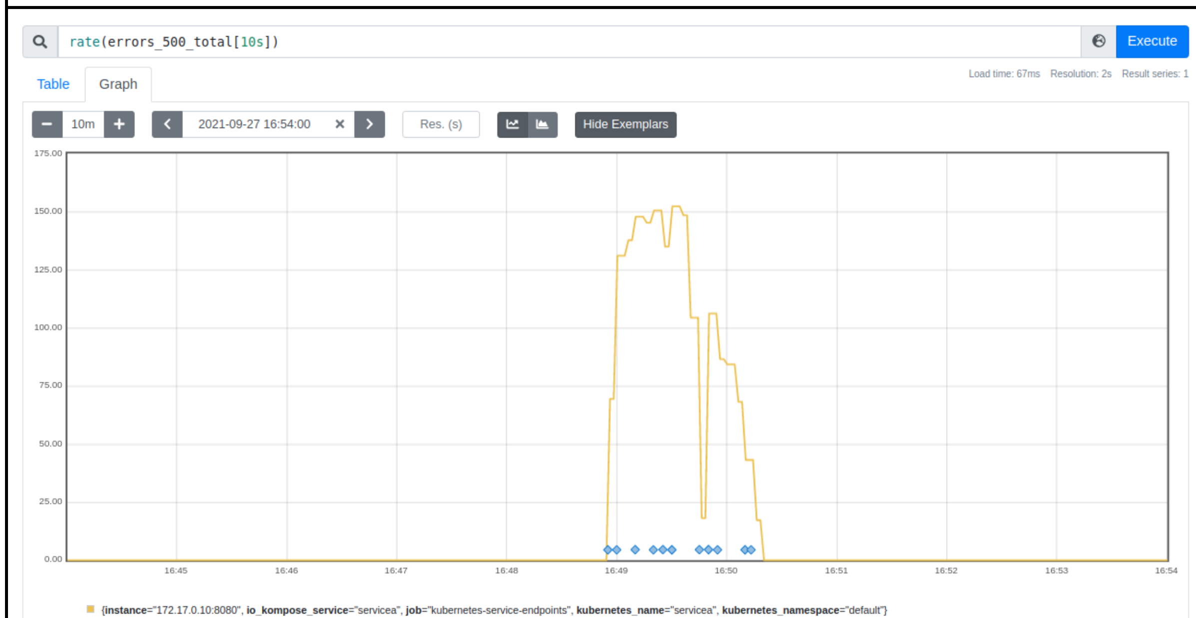
Εικόνα 34: Prometheus γράφημα για ρυθμό λήψης αιτημάτων από την εφαρμογή



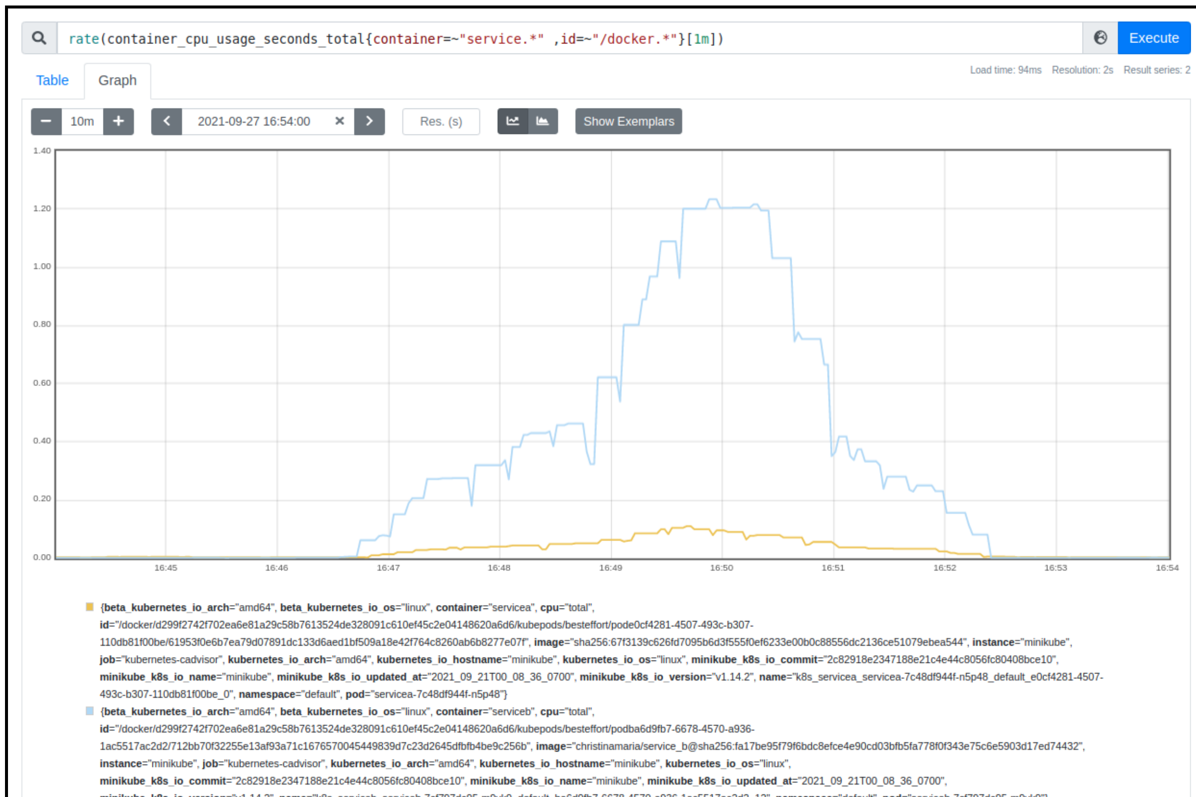
Εικόνα 35: Prometheus γράφημα για ρυθμό μεταβολής στις διάρκειες εξυπηρέτησης των αιτημάτων



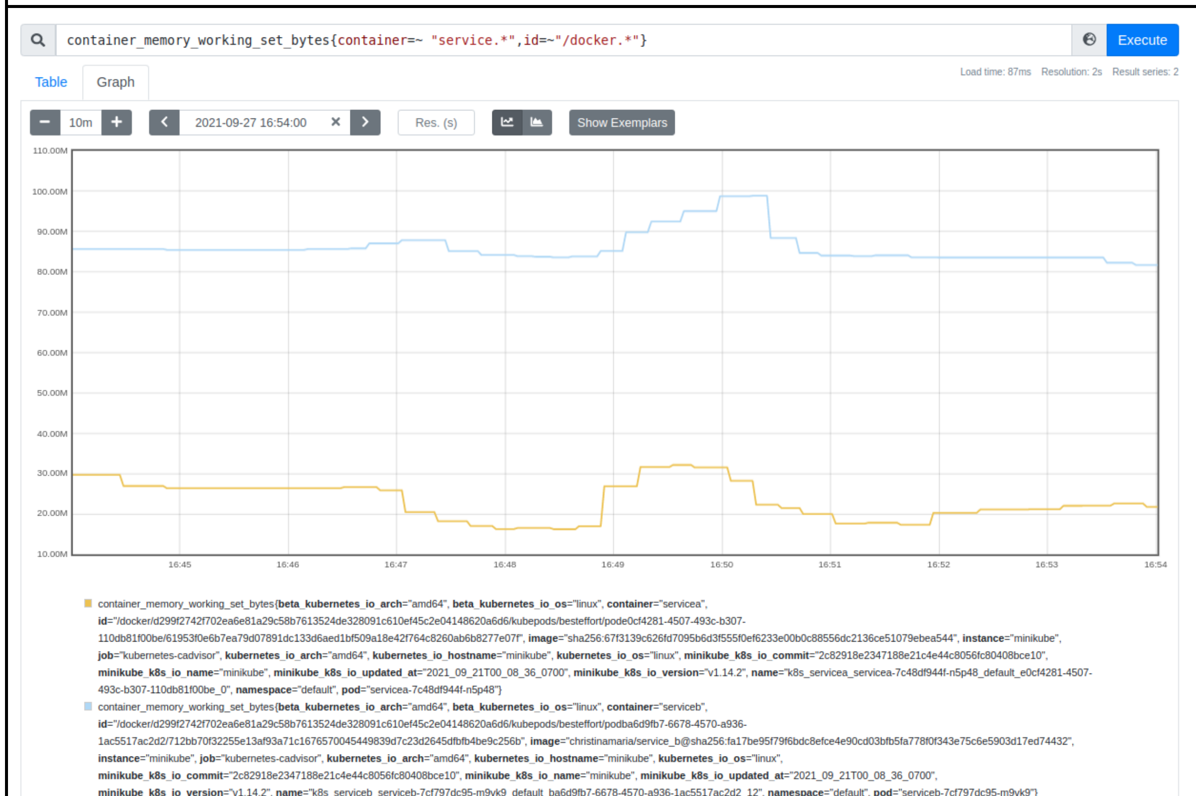
Εικόνα 36: Prometheus γράφημα με τις διάρκειες εξυπηρέτησης των αιτημάτων



Εικόνα 37: Prometheus γράφημα για ρυθμό μεταβολής στις διάρκειες εξυπηρέτησης αιτημάτων που εμφάνισαν σφάλμα υπέρβασης χρονικού ορίου (timeout) με κωδικό 500



Εικόνα 38: Prometheus γράφημα με συνολική χρήση της cpu από το κάθε pod



Εικόνα 39: Prometheus γράφημα με συνολική χρήση της μνήμης σε bytes από το κάθε pod

Στο γράφημα στην [Εικόνα 34], με την PromQL ερώτηση "rate(http_requests_total[10s])", απεικονίζεται ο ρυθμός λήψης αιτημάτων που δέχεται η εφαρμογή, δηλαδή το throughput, κατά τη διάρκεια του φορτίου δοκιμής. Πράγματι συμφωνεί με το ρυθμό και τη διάρκεια που θέσαμε στο Vegeta. Στο γράφημα στην [Εικόνα 35], με την PromQL ερώτηση "rate(request_latency_seconds_total[10s])", απεικονίζεται ο ρυθμός μεταβολής των διαρκειών εξυπηρέτησης των αιτημάτων από την εφαρμογή. Οι τιμές αυτής της μετρικής συνοδεύονται και από exemplars τα οποία αναπαρίστανται με σημεία πάνω στο γράφημα και έχουν ως τιμή την πραγματική διάρκεια του trace το οποίο αντιπροσωπεύουν. Για το κάθε exemplar παρέχεται από το UI του Prometheus το traceID του, η τιμή του και διάφορες ετικέτες που σχετίζονται με τη συλλογή του μαζί με την μετρική [Εικόνα 40]. Οι 2 αιχμές του γραφήματος, ερμηνεύονται ως απότομη αύξηση στη διάρκεια εξυπηρέτησης αιτημάτων (διάρκεια των αντίστοιχων traces) και παρουσιάζονται όταν το σύστημα φορτώνεται με πολλά αιτήματα (ρυθμός 30 αιτήματα/δευτερόλεπτο που ακολουθείται από ρυθμό 20 αιτήματα/δευτερόλεπτο). Αυτή η παρατήρηση επαληθεύεται και από τις ακριβείς διάρκειες εξυπηρέτησης του κάθε αιτήματος που δέχθηκε το σύστημα [Εικόνα 36 - με τη γραφική αναπαράσταση της μετρικής "request_latency" τύπου gauge], οι οποίες εμφανίζουν τις υψηλότερες τιμές τους την ίδια περίοδο υπερφόρτωσης του συστήματος. Τότε μάλιστα παρουσιάζονται και αιτήματα που εμφάνισαν σφάλμα υπέρβασης χρονικού ορίου (timeout error), όπως φαίνεται στο γράφημα "rate(errors_500_total[10s])" [Εικόνα 37 - ρυθμός μεταβολής της χρονικής διάρκειας των αιτημάτων που εμφάνισαν σφάλμα με κωδικό 500]. Τα exemplars που έχουν συλλεχθεί μαζί με την μετρική "errors_500_total" [Εικόνα 41], έχουν ως τιμές τους τις διάρκειες εξυπηρέτησης των αιτημάτων με σφάλμα που εκπροσωπούν. Στην [Εικόνα 38] με "rate(container_cpu_usage_seconds_total{container=~"service.*",id=~"/docker.*"}[1m])" παρουσιάζεται η συνολική χρήση της cpu από το κάθε Pod στο οποίο εκτελείται και η αντίστοιχη μικροϋπηρεσία. Παρατηρείται ότι η χρήση της cpu και στα 2 Pods κατά την εκτέλεση του φορτίου δοκιμής αυξάνεται ανάλογα με το πλήθος και το ρυθμό αιτημάτων που δέχεται η εφαρμογή. Το Pod του service b καταναλώνει, όπως ήταν αναμενόμενο, πολύ περισσότερη υπολογιστική ισχύ από το Pod του service a, καθώς πραγματοποιεί έναν πιο απαιτητικό Fibonacci υπολογισμό. Η χρήση της μνήμης τους, από την άλλη, παρουσιάζει αύξηση κυρίως την περίοδο που το σύστημα φορτώνεται με τα περισσότερα αιτήματα (περίοδος με ρυθμό 30 αιτήματα/δευτερόλεπτο που ακολουθείται από ρυθμό 20 αιτήματα/δευτερόλεπτο) [Εικόνα 39 - "container_memory_working_set_bytes{container=~"service.*", id=~"/docker.*"}"]. Η μνήμη που καταναλώνει το service b είναι, όπως αναμένεται λόγω των απαιτητικών διεργασιών του (memory_consuming_process), αρκετά περισσότερη από αυτή που χρειάζεται το service a.

Selected exemplar labels:

traceID: 126d07523ae54571

Associated series labels:

__name__: request_latency_seconds_total
instance: 172.17.0.3:8080
io_kompose_service: servicea
job: kubernetes-service-endpoints
kubernetes_name: servicea
kubernetes_namespace: default

Εικόνα 40: Πληροφορίες για το exemplar της μετρικής "request_latency_seconds_total" από το UI του Prometheus

Selected exemplar labels:

traceID: 4665f6b6e1fabf9b

Associated series labels:

__name__: errors_500_total
instance: 172.17.0.10:8080
io_kompose_service: servicea
job: kubernetes-service-endpoints
kubernetes_name: servicea
kubernetes_namespace: default

Εικόνα 41: Πληροφορίες για το exemplar της μετρικής "errors_500_total" από το UI του Prometheus

5.3.2 Ανάλυση δεδομένων

Στο αρχείο *data_fusion.py*[11], συλλέγουμε όλα τα exemplars μέσω Prometheus API που έχουν δημιουργηθεί είτε από επιτυχημένα αιτήματα, μέσω της μετρικής *request_latency_seconds_total*, είτε από αιτήματα στα οποία προέκυψε σφάλμα, μέσω της μετρικής *errors_500_total* . Στη συνέχεια οι πληροφορίες για το κάθε exemplar και trace με τα span του ομαδοποιούνται στην JSON μορφοποίηση που έχει προταθεί παραπάνω και παρουσιάζονται σε πίνακα όπως φαίνεται στην [Εικόνα 42].

	root_span.span_id	traceID	root_span.name	status_code	root_span.timestamp(micro)	root_span.span_duration(micro)	root_span.throughput_avg	pod_x	cpu_x	memory_x	pod_y	cpu_y	memory_y
0	7868fd01c73ef483	7868fd01c73ef483	request	200 OK	1632761194847876	25675.0	0.000000	servicea	0.003704	26660864.0	serviceb	0.004946	85827584.0
1	798e6e5f74e7e5c7	798e6e5f74e7e5c7	request	200 OK	1632761199747729	27395.0	3.200000	servicea	0.003646	26660864.0	serviceb	0.006119	85827584.0
2	4f27f681757c8320	4f27f681757c8320	request	200 OK	1632761204847987	29257.0	9.889241	servicea	0.003646	26660864.0	serviceb	0.062319	87068672.0
3	31a1b8916fd0e90e	31a1b8916fd0e90e	request	200 OK	1632761209847913	27719.0	10.113269	servicea	0.010683	25874432.0	serviceb	0.062319	87068672.0
4	270a66e56ad8f7da	270a66e56ad8f7da	request	200 OK	1632761214847710	29492.0	10.000000	servicea	0.010683	25874432.0	serviceb	0.076470	87068672.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
72	2ced169cc978cddf	2ced169cc978cddf	request	200 OK	1632761479886020	25616.0	10.000000	servicea	0.028287	17616896.0	serviceb	0.333564	83914752.0
73	59fc149f9bad9b63	59fc149f9bad9b63	request	200 OK	1632761484886120	29724.0	10.000000	servicea	0.033798	17829888.0	serviceb	0.319978	84103168.0
74	3e962e6e342b8822	3e962e6e342b8822	request	200 OK	1632761489886126	27040.0	10.000000	servicea	0.033798	17829888.0	serviceb	0.280851	84103168.0
75	72954e1d17d1323c	72954e1d17d1323c	request	200 OK	1632761494886532	24920.0	10.000000	servicea	0.033018	17829888.0	serviceb	0.280851	84103168.0
76	55f8e487585c07e2	55f8e487585c07e2	request	200 OK	1632761498686211	28237.0	10.000000	servicea	0.032855	17313792.0	serviceb	0.280851	84103168.0

77 rows × 13 columns

Εικόνα 42: Πίνακας με πληροφορίες για το root span

Στην πρώτη στήλη του πίνακα είναι το αναγνωριστικό του root span (root_span.span_id) το οποίο ταυτίζεται με το αναγνωριστικό του trace (trace_id). Στις υπόλοιπες στήλες συγκεντρώνονται οι πληροφορίες που είναι διαθέσιμες από το σχήμα ενοποίησης για το root span, δηλαδή το όνομα του span "root_span.name", ο κωδικός κατάστασης που επέστρεψε το αίτημα προς την εφαρμογή "status_code", η χρονική στιγμή έναρξης του span και άρα του trace "root_span.timestamp(micro)", η χρονική του διάρκεια "root_span.span_duration(micro)" και ο μέσος ρυθμός λήψης αιτημάτων (μέσο throughput) της εφαρμογής για εκείνη την περίοδο που αυτό εκτελέστηκε "root_span.throughput_avg". Η κάθε στήλη Pod ("pod_x", "pod_y") αναφέρεται σε μια από τις μικροϋπηρεσίες από τις οποίες εκτελέστηκε το root_span και αντίστοιχα οι στήλες "cpu" και "memory" στη μέση

[11] https://gitlab.com/netmode/distributed-tracing-k8s/-/blob/master/data_fusion.py

cpu και μνήμη που κατανάλωσε σε αυτό το Pod κατά την εκτέλεσή του. Κάθε root_span, το οποίο περιγράφει συνολικά όλο το αίτημα, περνά και από τις 2 μικροϋπηρεσίες της εφαρμογής.
Συνολικά συλλέγονται 77 exemplars από τα 5400 αιτήματα που στάλθηκαν στην εφαρμογή.

Πίνακες ανά κατηγορία span με πληροφορίες για το κάθε span

Οι πίνακες ανά κατηγορία span (“fibonacci_calc_a” [Εικόνα 43], “fibonacci_calc_b” [Εικόνα 43], “memory_consuming_process” [Εικόνα 45]) περιέχουν την πληροφορία που συλλέγεται από το σχήμα ενοποίησης για το κάθε span, δηλαδή το αναγνωριστικό του (“spans.span_id”), το αναγνωριστικό του trace στο οποίο ανήκει (“traceID”) με τον κωδικό κατάστασής του (“status_code”), το όνομά του (“spans.name”), τη χρονική στιγμή έναρξής του (“spans.timestamp(micro)”) σε UNIX μορφή με ακρίβεια μικροδευτερολέπτων, τη χρονική του διάρκεια (“spans.span_duration(micro)”) σε μικροδευτερόλεπτα, το μέσο ρυθμό λήψης αιτημάτων (throughput) της εφαρμογής (“spans.throughput_avg”) καθώς αυτό εκτελείται, το Pod στο οποίο εκτυλίσσεται (“pod”) και αντίστοιχα τη μέση χρήση σε cpu (“cpu”) και μνήμη (“memory”) σε αυτό κατά τη διάρκειά του.

	spans.span_id	traceID	status_code	spans.name	spans.timestamp(micro)	spans.span_duration(micro)	spans.throughput_avg	pod	cpu	memory
0	3bb21632552e0910	7868fd01c73ef483	200 OK	fibonacci_calc_a	1632761194847926	21.0	0.000000	servicea	0.003704	26660864.0
8	4a85d3a75581de34	798e6e5f74e7e5c7	200 OK	fibonacci_calc_a	1632761199747783	20.0	3.200000	servicea	0.003646	26660864.0
16	6e9f70562282d652	4f27f681757c8320	200 OK	fibonacci_calc_a	1632761204848046	26.0	9.889241	servicea	0.003646	26660864.0
24	63dd0d0e1110edea	31a1b8916fd0e90e	200 OK	fibonacci_calc_a	1632761209847988	23.0	10.113269	servicea	0.010683	25874432.0
32	54959e4be4b01b30	270a66e56ad8f7da	200 OK	fibonacci_calc_a	1632761214847769	40.0	10.000000	servicea	0.010683	25874432.0
...	...	...	...	...	...	...	...	...	...	...
576	0a67d6304e55af45	2ced169cc978cddf	200 OK	fibonacci_calc_a	1632761479886074	30.0	10.000000	servicea	0.028294	17616896.0
584	0cadd22b5a1344c3	59fc149f9bad9b63	200 OK	fibonacci_calc_a	1632761484886168	20.0	10.000000	servicea	0.033798	17829888.0
592	1804ec59c0dd3618	3e962e6e342b8822	200 OK	fibonacci_calc_a	1632761489886171	21.0	10.000000	servicea	0.033798	17829888.0
600	53c4092d1d18fb71	72954e1d17d1323c	200 OK	fibonacci_calc_a	1632761494886576	20.0	10.000000	servicea	0.033018	17829888.0
608	7f3d003ecd448a7d	55f8e487585c07e2	200 OK	fibonacci_calc_a	1632761498686278	20.0	10.000000	servicea	0.032855	17313792.0

77 rows × 10 columns

Εικόνα 43: Πίνακας για τα spans "fibonacci_calc_a" που αφορούν τον υπολογισμό fibonacci στο servicea

	spans.span_id	traceID	status_code	spans.name	spans.timestamp(micro)	spans.span_duration(micro)	spans.throughput_avg	pod	cpu	memory
6	289acd3f50c1d524	7868fd01c73ef483	200 OK	fibonacci_calc_b	1632761194848484	19907.0	0.000000	serviceb	0.004946	85827584.0
14	7253cb29a776eff3	798e6e5f74e7e5c7	200 OK	fibonacci_calc_b	1632761199748462	21257.0	3.200000	serviceb	0.006119	85827584.0
22	30dd80ec8113db8d	4f27f681757c8320	200 OK	fibonacci_calc_b	1632761204848699	22677.0	9.889241	serviceb	0.062319	87068672.0
30	6614f91260b39dc7	31a1b8916fd0e90e	200 OK	fibonacci_calc_b	1632761209848619	21846.0	10.113269	serviceb	0.062319	87068672.0
38	7535f2ade3912467	270a66e56ad8f7da	200 OK	fibonacci_calc_b	1632761214848370	22664.0	10.000000	serviceb	0.076466	87068672.0
...	...	...	...	...	...	...	...	...	...	...
582	0efe2400d5140592	2ced169cc978cddf	200 OK	fibonacci_calc_b	1632761479886684	21669.0	10.000000	serviceb	0.333564	83914752.0
590	6034325b442ed144	59fc149f9bad9b63	200 OK	fibonacci_calc_b	1632761484886771	23505.0	10.000000	serviceb	0.319978	84103168.0
598	472217b28abc6096	3e962e6e342b8822	200 OK	fibonacci_calc_b	1632761489886671	22609.0	10.000000	serviceb	0.280851	84103168.0
606	3413d45bbeb64791	72954e1d17d1323c	200 OK	fibonacci_calc_b	1632761494887213	21100.0	10.000000	serviceb	0.280851	84103168.0
614	44c07ccf1469afdb	55f8e487585c07e2	200 OK	fibonacci_calc_b	1632761498686813	22358.0	10.000000	serviceb	0.280851	84103168.0

77 rows × 10 columns

Εικόνα 44: Πίνακας για τα spans "fibonacci_calc_b" που αφορούν τον υπολογισμό fibonacci στο serviceb

	spans.span_id	traceID	status_code	spans.name	spans.timestamp(micro)	spans.span_duration(micro)	spans.throughput_avg	pod	cpu	memory
7	08652ee1c0bd1512	472626de7a6deda7	200 OK	memory_consuming_process	1632217171763487	2349.0	0.14684	serviceb	0.002399	3239936.0
15	3ff5bf512bb0791d	2862d55fc2941f06	200 OK	memory_consuming_process	1632217269408169	3222.0	0.00000	serviceb	0.002402	4976640.0
23	37298b1ccf446a92	4c8fad1b19dd768e	200 OK	memory_consuming_process	1632217274410198	3541.0	1.00000	serviceb	0.021516	5718016.0
31	2b17835695b3a50b	19b8b03b3121d7d8	200 OK	memory_consuming_process	1632217279406756	5367.0	10.00000	serviceb	0.021516	5718016.0
39	6f2b5f5cd9725444	43343a84ecfb4822	200 OK	memory_consuming_process	1632217284408075	3438.0	10.00000	serviceb	0.021516	5718016.0
...	...	...	...	...	...	...	...	...	...	...
463	6a6e99d237879748	021e87f1c74b3311	200 OK	memory_consuming_process	1632217549328808	5360.0	10.00000	serviceb	0.348355	10309632.0
471	543879f438260560	3c810e8f36938894	200 OK	memory_consuming_process	1632217554327438	4760.0	10.00000	serviceb	0.227338	10153984.0
479	7a3e9375ca70bc64	0e6ad77f21e7e40a	200 OK	memory_consuming_process	1632217559328488	4361.0	10.00000	serviceb	0.279649	10153984.0
487	5aaab07df4008032	0ba8df125b909f0d	200 OK	memory_consuming_process	1632217564328656	5319.0	10.00000	serviceb	0.279649	10153984.0
495	45a8d6801e6c4ffd	440159790b8e9a67	200 OK	memory_consuming_process	1632217569029384	5263.0	10.00000	serviceb	0.279526	10203136.0

62 rows × 10 columns

Εικόνα 45: Πίνακας για τα spans "memory_consuming_process" που αφορούν την απαιτητική σε μνήμη διαδικασία

Για τον υπολογισμό της διάρκειας της κλήσης από το service a στο service b , αφαιρούμε τη χρονική στιγμή έναρξης του span "service_b" (χρονική στιγμή λήψης του αιτήματος που στάλθηκε από το service a) από τη χρονική στιγμή έναρξης του span "call_to_service_b" (χρονική στιγμή αποστολής αιτήματος προς service b). Έτσι ο πίνακας που προκύπτει [Εικόνα 46] περιέχει το αναγνωριστικό του trace "traceID" στο οποίο ανήκει η κλήση από το service a στο service b, τη χρονική στιγμή έναρξής της "timestamp(micro)", η οποία είναι η χρονική στιγμή έναρξης του span "call_to_service_b" και τη διάρκειά της "call_duration(micro)", η οποία υπολογίζεται όπως αναφέρθηκε προηγουμένως.

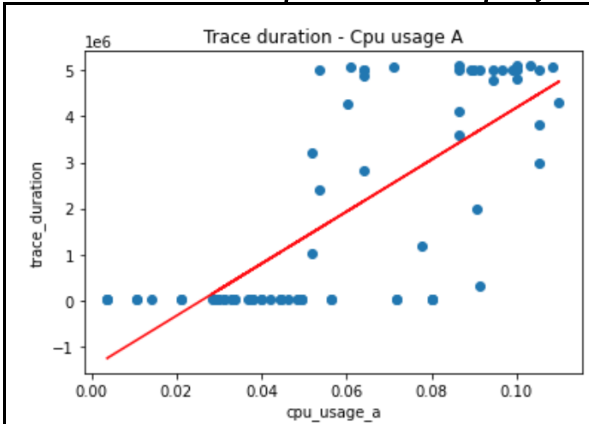
	traceID	timestamp(micro)	call_duration(micro)
0	7868fd01c73ef483	1632761194847977	456
1	798e6e5f74e7e5c7	1632761199747827	572
2	4f27f681757c8320	1632761204848102	539
3	31a1b8916fd0e90e	1632761209848036	481
4	270a66e56ad8f7da	1632761214847856	467
...	...	...	...
72	2ced169cc978cddf	1632761479886134	500
73	59fc149f9bad9b63	1632761484886211	510
74	3e962e6e342b8822	1632761489886214	407
75	72954e1d17d1323c	1632761494886657	507
76	55f8e487585c07e2	1632761498686334	431

77 rows × 3 columns

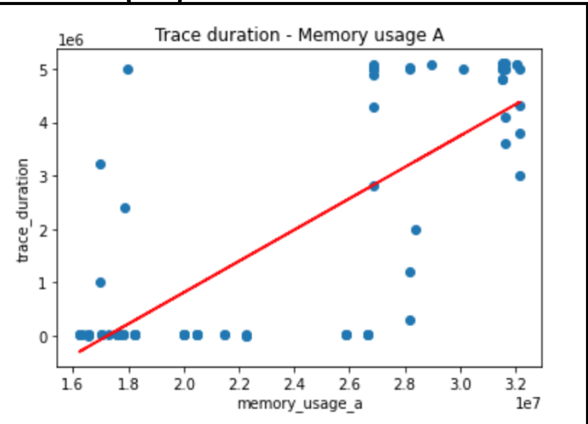
Εικόνα 46: Πίνακας για την κλήση από το servicea στο serviceb

Για τις διάφορες μεταβλητές που έχουμε συλλέξει μπορούμε να αναζητήσουμε τη σχέση τους και την αλληλεξάρτησή τους. Για την εύρεση σχέσεων αιτιότητας (causality) χρησιμοποιείται γραμμική παλινδρόμηση (linear regression) και συγκεκριμένα η μέθοδος ελαχίστων τετραγώνων (least squares). Όταν υπάρχει σχέση αιτιότητας που εκφράζεται ως γραμμική συσχέτιση, χρησιμοποιείται ο συντελεστής γραμμική συσχέτισης Pearson για να μετρηθεί ο βαθμός της συσχέτισης (correlation).

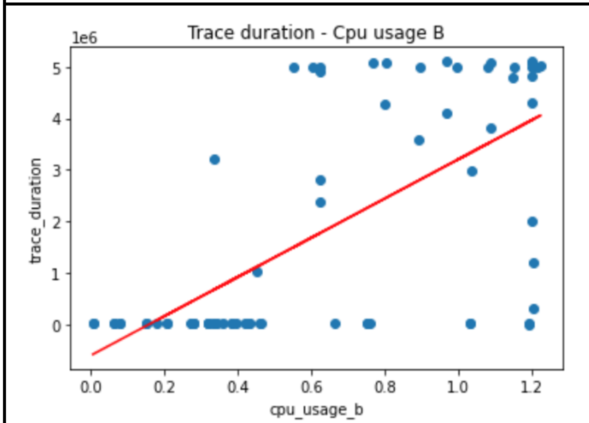
Διάρκεια trace - Μετρικές των Pod κατά τη διάρκεια του trace



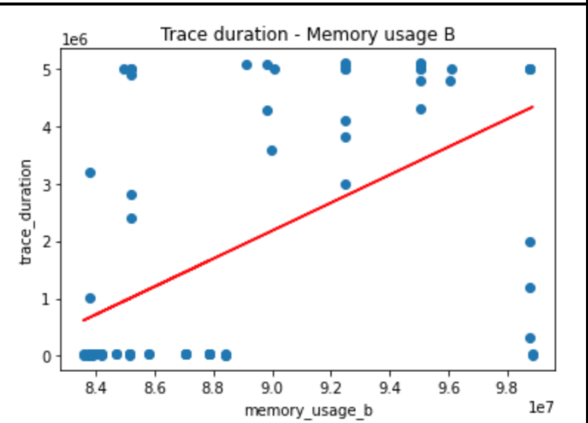
Εικόνα 47: Διάγραμμα σχέσης αιτιότητας διάρκειας trace και χρήσης cpu στο pod a καθώς αυτό εκτυλίσσεται



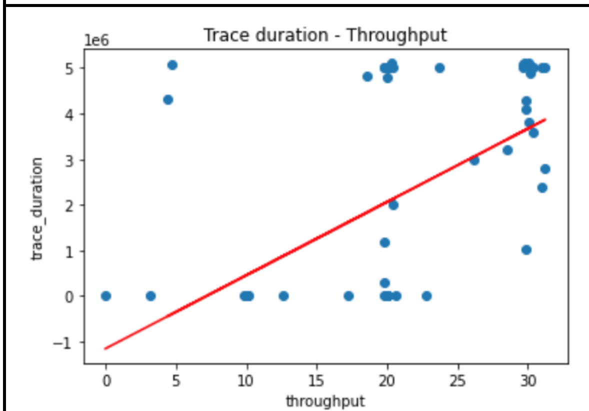
Εικόνα 48: Διάγραμμα σχέσης αιτιότητας διάρκειας trace και χρήσης μνήμης στο pod a καθώς αυτό εκτυλίσσεται



Εικόνα 49: Διάγραμμα σχέσης αιτιότητας διάρκειας trace και χρήσης cpu στο pod b καθώς αυτό εκτυλίσσεται



Εικόνα 50: Διάγραμμα σχέσης αιτιότητας διάρκειας trace και χρήσης μνήμης στο pod b καθώς αυτό εκτυλίσσεται



Εικόνα 51: Διάγραμμα σχέσης αιτιότητας διάρκειας trace και throughput της εφαρμογής καθώς αυτό εκτυλίσσεται

	trace_duration	cpu_a	memory_a	throughput_avg	cpu_b	memory_b
trace_duration	1.000000	0.745344	0.775972	0.591455	0.673020	0.552751
cpu_a	0.745344	1.000000	0.696999	0.527442	0.955886	0.746014
memory_a	0.775972	0.696999	1.000000	0.289419	0.686290	0.753314
throughput_avg	0.591455	0.527442	0.289419	1.000000	0.454091	0.177222
cpu_b	0.673020	0.955886	0.686290	0.454091	1.000000	0.832237
memory_b	0.552751	0.746014	0.753314	0.177222	0.832237	1.000000

Εικόνα 52: Πίνακας συσχέτισης των μεταβλητών διάρκειας trace (trace_duration) και μετρικών των pods ανά διάρκεια trace

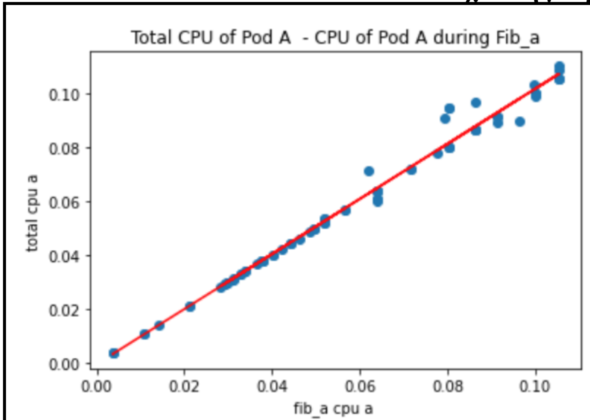
Από τα παραπάνω διαγράμματα [Εικόνα 47-51] παρουσιάζεται γραμμική συσχέτιση μεταξύ της διάρκειας των trace, άρα της διάρκειας εξυπηρέτησης των αιτημάτων, και της χρήσης των υπολογιστικών πόρων των Pod που φιλοξενούν τα service a και b. Συνεπώς τα αιτήματα με τη μεγαλύτερη διάρκεια εξυπηρέτησης καταναλώνουν και περισσότερους πόρους στα Pod. Από τους βαθμούς συσχέτισης στον πίνακα [Εικόνα 52] συμπεραίνουμε πως η κατανάλωση μνήμης και cpu στο Pod a είναι αυτή που επηρεάζει στο μεγαλύτερο βαθμό τη διάρκεια του trace. Υψηλή συσχέτιση εμφανίζεται επίσης και μεταξύ του ρυθμού λήψης αιτημάτων από την εφαρμογή (throughput) με τη διάρκεια των traces.

	trace_duration(micro)	fib_a_duration(micro)	call_duration(micro)	fib_b_duration(micro)	memory_cons_process_duration(micro)
trace_duration(micro)	1.000000	0.373800	0.697796	0.838538	0.761087
fib_a_duration(micro)	0.373800	1.000000	0.141801	0.305257	0.246003
call_duration(micro)	0.697796	0.141801	1.000000	0.622628	0.670692
fib_b_duration(micro)	0.838538	0.305257	0.622628	1.000000	0.658431
memory_cons_process_duration(micro)	0.761087	0.246003	0.670692	0.658431	1.000000

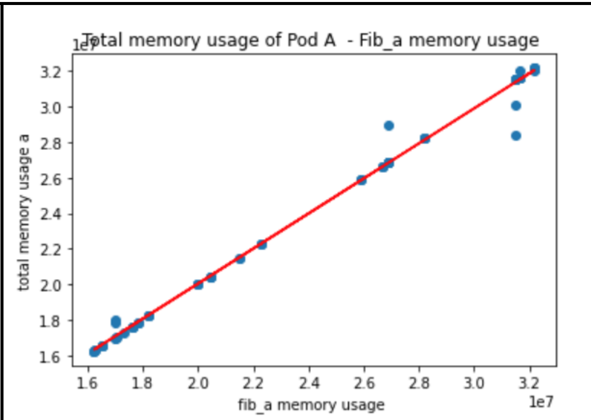
Εικόνα 53: Πίνακας συσχέτισης διάρκειας trace με επιμέρους διάρκειες των spans του (“fibonacci_calc_a”, “fibonacci_calc_b”, “memory_consuming_process”) και διάρκειας κλήσης από το service a στο service b (“call_duration”)

Οι επιμέρους διάρκειες των spans εξ ορισμού επηρεάζουν την συνολική διάρκεια του trace. Από τον πίνακα με τους βαθμούς συσχέτισης [Εικόνα 53] παρατηρούμε ότι η διάρκεια του span “fibonacci_calc_b” (“fib_b_duration(micro)”) καθορίζει στο μεγαλύτερο βαθμό τη διάρκεια του trace (“trace_duration(micro)”), καθώς είναι και η διαδικασία που απαιτεί περισσότερο χρόνο να εκτελεστεί λόγω του υπολογιστικού της φόρτου. Στη συνέχεια ακολουθούν με εξίσου υψηλούς βαθμούς συσχέτισης η διάρκεια εκτέλεσης της memory_consuming_process διαδικασίας (“process_duration(micro)”) και της κλήσης από το service a στο b (“call_duration(micro)”). Στο μικρότερο βαθμό επηρεάζει η διάρκεια του υπολογισμού fibonacci στο servicea αφού πραγματοποιείται πολύ γρήγορα.

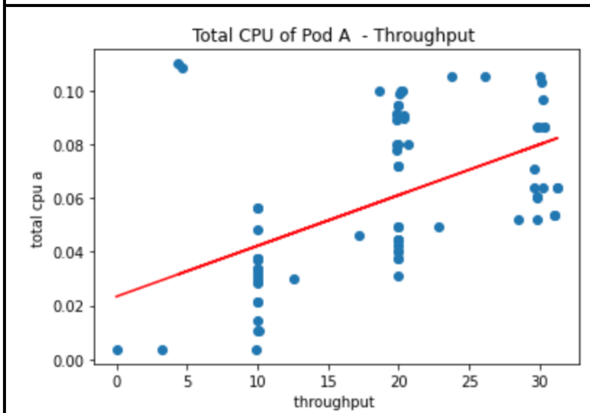
Συσχέτιση μετρικών του Pod A



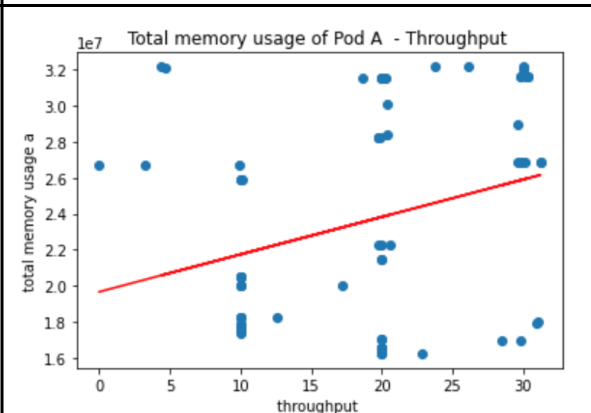
Εικόνα 54: Διάγραμμα σχέσης αιτιότητας χρήσης cpu στο pod a κατά τη διάρκεια ενός trace και κατά τον υπολογισμό fibonacci



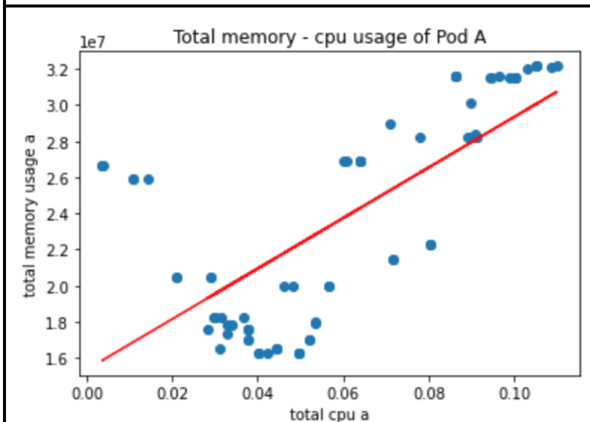
Εικόνα 55: Διάγραμμα σχέσης αιτιότητας χρήσης μνήμης στο pod a κατά τη διάρκεια ενός trace και κατά τον υπολογισμό fibonacci



Εικόνα 56: Διάγραμμα σχέσης αιτιότητας χρήσης cpu στο pod a και throughput



Εικόνα 57: Διάγραμμα σχέσης αιτιότητας χρήσης μνήμης στο pod a και throughput

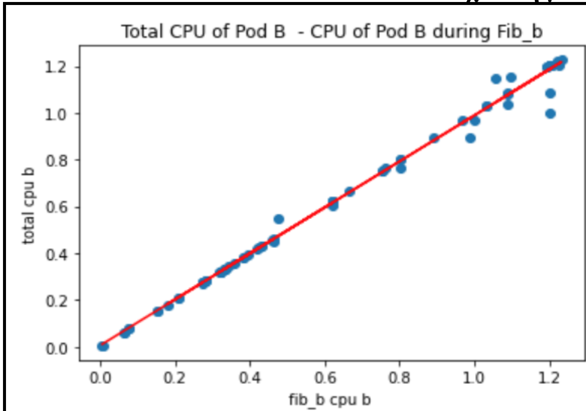


Εικόνα 58: Διάγραμμα σχέσης αιτιότητας χρήσης cpu και μνήμης στο pod a

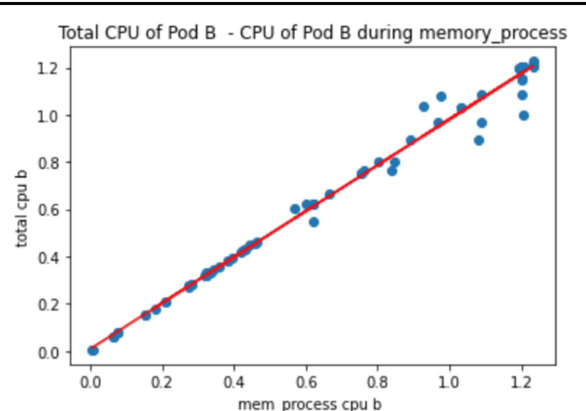
	throughput_avg	total_cpu_a	total_memory_a	fib_a_cpu	fib_a_memory
throughput_avg	1.000000	0.527442	0.289419	0.533113	0.274457
total_cpu_a	0.527442	1.000000	0.696999	0.994177	0.697831
total_memory_a	0.289419	0.696999	1.000000	0.684120	0.996739
fib_a_cpu	0.533113	0.994177	0.684120	1.000000	0.685009
fib_a_memory	0.274457	0.697831	0.996739	0.685009	1.000000

Εικόνα 59: Πίνακας συσχέτισης μερικών στο pod a

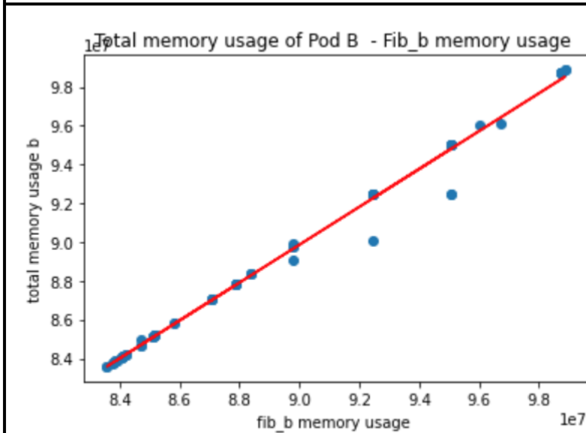
Συσχέτιση μετρικών του Pod B



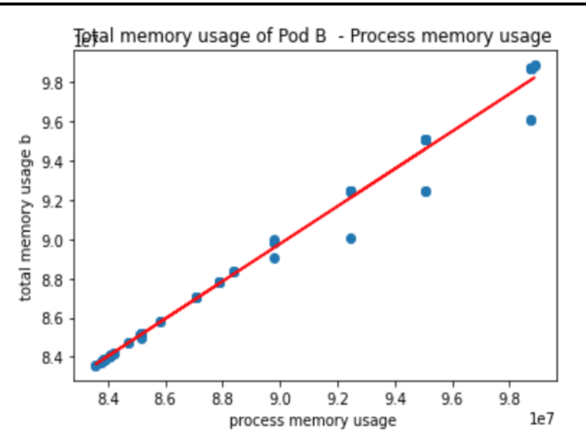
Εικόνα 60: Διάγραμμα σχέσης αιτιότητας χρήσης cpu στο pod b κατά τη διάρκεια ενός trace και κατά τον υπολογισμό fibonacci



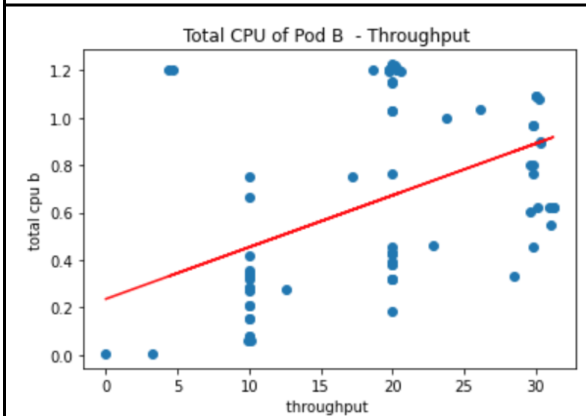
Εικόνα 61: Διάγραμμα σχέσης αιτιότητας χρήσης cpu στο pod b κατά τη διάρκεια ενός trace και κατά την εκτέλεση της memory_consuming_process



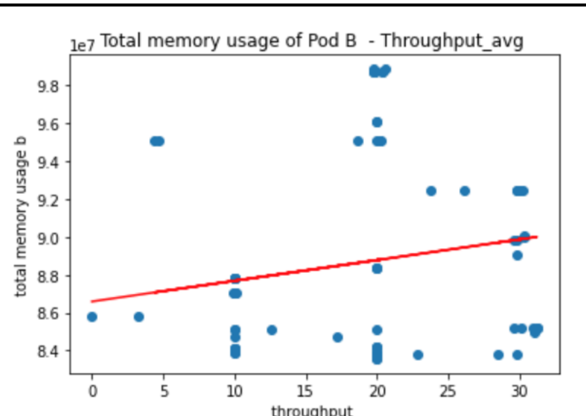
Εικόνα 62: Διάγραμμα σχέσης αιτιότητας χρήσης μνήμης στο pod b κατά τη διάρκεια ενός trace και κατά τον υπολογισμό fibonacci



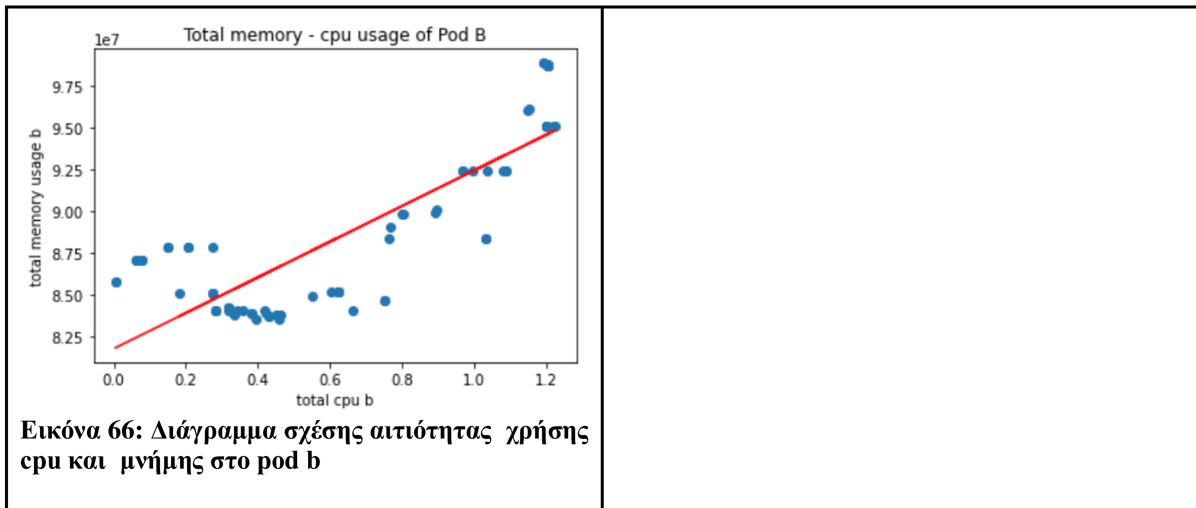
Εικόνα 63: Διάγραμμα σχέσης αιτιότητας χρήσης μνήμης στο pod b κατά τη διάρκεια ενός trace και κατά την εκτέλεση της memory_cosnuming_process



Εικόνα 64: Διάγραμμα σχέσης αιτιότητας χρήσης cpu στο pod b και throughput



Εικόνα 65: Διάγραμμα σχέσης αιτιότητας χρήσης μνήμης στο pod b και throughput



Εικόνα 66: Διάγραμμα σχέσης αιτιότητας χρήσης cpu και μνήμης στο pod b

	throughput_avg	total_cpu_b	total_memory_b	fib_b_cpu	fib_b_memory	process_cpu	process_memory
throughput_avg	1.000000	0.454091	0.177222	0.462311	0.196833	0.464264	0.196694
total_cpu_b	0.454091	1.000000	0.832237	0.996680	0.835971	0.994544	0.834964
total_memory_b	0.177222	0.832237	1.000000	0.826890	0.995412	0.827840	0.993053
fib_b_cpu	0.462311	0.996680	0.826890	1.000000	0.836760	0.995451	0.832941
fib_b_memory	0.196833	0.835971	0.995412	0.836760	1.000000	0.839002	0.997456
process_cpu	0.464264	0.994544	0.827840	0.995451	0.839002	1.000000	0.839048
process_memory	0.196694	0.834964	0.993053	0.832941	0.997456	0.839048	1.000000

Εικόνα 67: Πίνακας συσχέτισης μετρικών στο pod b

Η σχέση αιτιότητας μεταξύ της χρήσης υπολογιστικών πόρων ενός Pod και των απαιτήσεων στους αντίστοιχους πόρους από τις διεργασίες που εκτελούνται σε αυτά παρουσιάζεται στις [Εικόνες 54, 56] και [Εικόνες 60-63] για το service a και b αντίστοιχα. Από τους πίνακες με τους βαθμούς συσχέτισης [Εικόνα 59, 67] επιβεβαιώνεται ότι η κατανάλωση cpu και μνήμης από τις εσωτερικές διαδικασίες fibonacci και memory_cosnuming_process σχετίζεται απόλυτα με την συνολική κατανάλωση των πόρων αυτών στα Pod κατά την εξυπηρέτηση αιτημάτων. Ο ρυθμός με τον οποίο δέχεται αιτήματα η εφαρμογή (throughput) σχετίζεται, όπως φαίνεται στις [Εικόνες 56, 57] για το Pod a και [Εικόνες 64, 65] για το Pod b, με τη χρήση cpu και μνήμης στο κάθε Pod. Υψηλότερη βέβαια συσχέτιση παρατηρείται με την συνολική χρήση της cpu παρά της μνήμης, εξίσου και στα 2 Pods [Εικόνα 59, 67]. Τέλος για το κάθε Pod υπάρχει γραμμική εξάρτηση μεταξύ της χρήσης cpu και μνήμης [Εικόνα 58, 66], η οποία εμφανίζεται πιο έντονα στο δεύτερο pod [Εικόνα 59, 67], κυρίως λόγω της ύπαρξης διεργασιών απαιτητικών τόσο σε cpu όσο και μνήμη (fibonacci_calc_b και memory_consuming_process αντίστοιχα).

Κεφάλαιο 6: Συμπεράσματα

Η κατανεμημένη παρακολούθηση (ιχνηλάτηση) παίζει καθοριστικό ρόλο και είναι αναγκαία προϋπόθεση για τη διασφάλιση της ομαλής λειτουργίας των cloud-native εφαρμογών. Η ορατότητα που παρέχει στην εσωτερική τους δομή και διασύνδεση, παρά την πολυπλοκότητά τους, επιτρέπει τον εντοπισμό σφαλμάτων με στόχο την αντιμετώπιση ή και αποτροπή τους. Όπως φάνηκε και από την πιλοτική εφαρμογή που αναπτύχθηκε, η προτεινόμενη αρχιτεκτονική κατανεμημένης παρακολούθησης με την κατάλληλη συλλογή και ενοποίηση των δεδομένων ιχνηλάτησης και μετρικών, μπορεί να εντοπίσει ποια/ποιες από τις μικροϋπηρεσίες είναι υπεύθυνες για την καθυστέρηση εξυπηρέτησης αιτημάτων των χρηστών, σε ποιες διεργασίες αφιερώνεται περισσότερος χρόνος και πώς επηρεάζεται η κατανάλωση υπολογιστικών πόρων των υποδομών.
Μέσω της ενασχόλησης με τεχνικές κατανεμημένης παρακολούθησης εφαρμογών υπολογιστικού νέφους στο πλαίσιο της εργασίας, προέκυψαν ανάγκες και ιδέες για πιθανές επεκτάσεις στο μέλλον. Κρίνεται αναγκαία η διενέργεια δοκιμών της προτεινόμενης αρχιτεκτονικής κατανεμημένης παρακολούθησης σε πραγματικό Kubernetes cluster, μεγαλύτερης κλίμακας από το minikube, με δοκιμές σε πολυπλοκότερες cloud-native εφαρμογές περισσότερων μικροϋπηρεσιών. Με αυτό τον τρόπο θα επιτευχθεί διευρυμένη ανάλυση των επιπτώσεων που μπορεί να έχουν οι καθυστερήσεις που παρατηρούνται στην επικοινωνία μεταξύ των μικροϋπηρεσιών και να βγουν πιο ασφαλή συμπεράσματα σχετικά με την αξία της συλλογής δεδομένων κατανεμημένης παρακολούθησης. Επιπλέον, προτείνεται να εξεταστούν οι επιδράσεις στην απόδοση υπό διαφορετικά φορτία δοκιμών με στόχο την εξέταση της συμπεριφοράς των εφαρμογών σε διάφορα σενάρια χρήσης. Τέλος, το προτεινόμενο σχήμα ενοποίησης των δεδομένων μπορεί να επανεξεταστεί και πιθανόν να προτυποποιηθεί με στόχο την υιοθέτησή του από εφαρμογές παρακολούθησης και ανάλυσης δεδομένων, άμεσα συνδεδεμένες με μηχανισμούς ενορχήστρωσης εφαρμογών υπολογιστικού νέφους.

Αποθετήριο Κώδικα

Ο κώδικας για την ανάπτυξη της πιλοτικής υλοποίησης καθώς και για την ανάλυση των δεδομένων που συλλέγονται με στόχο την ενοποίησή τους βρίσκεται στο αποθετήριο κώδικα: https://gitlab.com/netmode/distributed-tracing-k8s

Ο τρόπος εκτέλεσης της προτεινόμενης υλοποίησης περιγράφεται στο README αρχείο και η περιγραφή της πορείας της εργασίας και των ενεργειών που ακολουθήθηκαν υπάρχει στο Wiki του αποθετηρίου.

Εικόνα 68: Αποθετήριο κώδικα

Παράρτημα

Αναλυτική δομή δεδομένων ιχνηλατημένης πληροφορίας από το API του εργαλείου Zipkin για την πιλοτική εφαρμογή που αναπτύχθηκε.


```
[
{"traceId":"63780b7667ab8719",
"id":"63780b7667ab8719",
"kind":"SERVER",
"name":"request",
"timestamp":1632482002668318,
"duration":115881,
"localEndpoint":{
       "serviceName":"servicea",
       "ipv4":"10.104.207.109",
       "port":8080},
"remoteEndpoint":{
       "ipv4":"127.0.0.1",
       "port":36102},
"tags":{
       "http.method":"GET",
       "http.path":"/",
       "http.status_code":"500",
       "response":"OK",
       "servicea":"servicea-f89cb5f99-nqq9l",
       "status":"200OK"
},
{"traceId":"63780b7667ab8719",
"parentId":"63780b7667ab8719",
"id":"6888cb65fbb853cd",
"name":"fibonacci_calc_a",
"timestamp":1632482002668759,
"duration":26,
"localEndpoint":{
       "serviceName":"servicea",
       "Ipv4":"10.104.207.109",
       "Port":8080},
"tags":{
       "Fibonacci_Calc":"15",
       "servicea":"servicea-f89cb5f99-nqq9l"}
},

{"traceId":"63780b7667ab8719",
"parentId":"63780b7667ab8719",
"id":"2c3536d941d51841",
"name":"call_to_service_b",
"timestamp":1632482002668838,
"duration":115331,
"localEndpoint":{
       "serviceName":"servicea",
       "ipv4":"10.104.207.109",
```

```
        "port":8080},
"remoteEndpoint":{
        "serviceName":"serviceb",
        "ipv4":"10.110.63.151",
        "port":8080},
"tags":{
        "http.method":"GET",
        "http.path":"/",
        "http.response.size":"2",
        "servicea":"servicea-f89cb5f99-nqq9l"}
},

{"traceId":"63780b7667ab8719",
"parentId":"2c3536d941d51841",
"id":"07927da43830608a",
"kind":"CLIENT",
"name":"http/get",
"timestamp":1632482002668857,
"duration":115224,
"localEndpoint":{
        "serviceName":"servicea",
        "ipv4":"10.104.207.109",
        "port":8080},
"remoteEndpoint":{
        "serviceName":"serviceb",
        "ipv4":"10.110.63.151",
        "port":8080},
"tags":{
        "http.method":"GET",
        "http.path":"/",
        "http.response.size":"2",
        "servicea":"servicea-f89cb5f99-nqq9l"}
},

{"traceId":"63780b7667ab8719",
"parentId":"2c3536d941d51841",
"id":"07927da43830608a",
"kind":"SERVER",
"name":"service_b",
"timestamp":1632482002748481,
"duration":29456,
"localEndpoint":{
        "serviceName":"serviceb",
        "ipv4":"10.110.63.151",
        "port":8080},
"remoteEndpoint":{
        "ipv4":"172.17.0.5",
        "port":40372},
"tags":{
        "http.method":"GET",
        "http.path":"/",
        "serviceb":"serviceb-7cf797dc95-m9vk9"},
"shared":true
```

```
},

{"traceId":"63780b7667ab8719",
"parentId":"07927da43830608a",
"id":"24178c13c4aad2e4",
"name":"fibonacci_calc_b",
"timestamp":1632482002748765,
"duration":22136,
"localEndpoint":{
       "serviceName":"serviceb",
       "ipv4":"10.110.63.151",
       "port":8080},
"tags":{
       "Fibonacci_Calc":"30",
       "serviceb":"serviceb-7cf797dc95-m9vk9"}
},

{"traceId":"63780b7667ab8719",
"parentId":"07927da43830608a",
"id":"089b09a90bdc3124",
"name":"memory_consuming_process",
"timestamp":1632482002770930,
"duration":6986,
"localEndpoint":{
       "serviceName":"serviceb",
       "ipv4":"10.110.63.151",
       "port":8080},
"tags":{
       "serviceb":"serviceb-7cf797dc95-m9vk9"}
}]
```

Γλωσσάρι

agility	δυναμικότητα
API	προγραμματιστική διεπαφή
bottlenecks	συμφορήσεις
CI/CD	Συνεχής Ενσωμάτωση -Συνεχής Παράδοση
cloud	υπολογιστικό νέφος
cloud computing	προγραμματιζόμενες υποδομές υπολογιστικού νέφους
cloud-native εφαρμογές	εγγενείς εφαρμογές υπολογιστικού νέφους
cluster	συστάδα υπολογιστών
collector	συλλέκτης
container image	εικόνα του container
context span	πλαίσιο του span
DAG-Directed Acyclic Graph	κατευθυνόμενος άκυκλος γράφος
data centers	κέντρα φιλοξενίας υποδομής
dependencies	εξαρτήσεις κώδικα
deployment	εκτέλεση
DevOps	Ομάδες Διαχείρισης
distributed tracing	κατανεμημένη παρακολούθηση/ιχνηλάτηση
downtime	χρονική περίοδος που το σύστημα δεν είναι διαθέσιμο
end-to-end	από άκρο σε άκρο
framework	προγραμματιστικό πλαίσιο
headers	επικεφαλίδες
host	φιλοξενητής
labels	ετικέτες
latency	χρόνος εξυπηρέτησης αιτήματος
load balancing	εξισορρόπηση φορτίου
loose-coupling	χαλαρή σύζευξη

master-components	κύρια τμήματα master-slave αρχιτεκτονικής
microservices	μικροϋπηρεσίες
monitoring	παρακολούθηση με καταγραφή μετρικών
overhead	απαιτήσεις
portability	φορητότητα
query	ερώτηση (προς προγραμματιστική διεπαφή)
registry	αποθετήριο εφαρμογών
reporter	ανταποκριτή
resilience	ανθεκτικότητα
runtime environment	προγραμματιστικό περιβάλλον
servers	εξυπηρετητές-διακομιστές
service	υπηρεσία
standardization	προτυποποίηση
throughput	ρυθμός λήψης αιτημάτων
timestamp	χρονική στιγμή
timing data	χρονικά δεδομένα
trace	ίχνος
tracer	ιχνηλάτης
User Interface (UI)	διεπαφή χρηστών
virtual machine	εικονική μηχανή

Βιβλιογραφία

[1] B. Lutkevich and K. Goulart, “cloud-native application,” *TechTarget*, Jul. 26, 2021.

[2] “Defining cloud native,” *Microsoft Docs*. https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

[3] “Microservices communication: methods, types and styles,” *Chakray*, Mar. 09, 2020. https://www.chakray.com/microservices-communication-methods-types-and-styles/

[4] “Cloud Native Is Trending: What You Need to Know,” *SDxCentral*, Apr. 13, 2021. https://www.sdxcentral.com/podcast/7-layers/cloud-native-is-trending-what-you-need-to-know/2021/04/

[5] “What is a Container?,” *Docker*. https://www.docker.com/resources/what-container.

[6] “Production-Grade Container Orchestration,” *Kubernetes*. https://kubernetes.io/

[7] “Kubernetes Components,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/components/

[8] “What Is Distributed Tracing in Microservices?,” *Splunk*. https://www.splunk.com/en_us/data-insider/what-is-distributed-tracing.html

[9] A. Parker, D. Spoonhower, R. Isaacs, J. Mace, and B. Sigelman, *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O’Reilly Media, 2020.

[10] R. Whitmore, “Understand Distributed Tracing,” *LightStep*, Nov. 21, 2019.

[11]“Architecture,” *Jaeger: open source, end-to-end distributed tracing*. https://www.jaegertracing.io/docs/1.26/architecture/ (accessed Sep. 28, 2021).

[12] openzipkin, “zipkin-go/span.go at master · openzipkin/zipkin-go,” *GitHub*. https://github.com/openzipkin/zipkin-go/blob/master/model/span.go

[13] “Swagger UI.” https://zipkin.io/zipkin-api/.

[14] “Introduction to exemplars,” *Grafana*. https://grafana.com/docs/grafana/latest/basics/exemplars/.

[15] V. Behar, “Using Prometheus Exemplars to jump from metrics to traces in Grafana,” *Medium*, Mar. 08, 2021.

[16] OpenObservability, “OpenMetrics/OpenMetrics.md at main · OpenObservability/OpenMetrics,” *GitHub*. https://github.com/OpenObservability/OpenMetrics/blob/main/specification/OpenMetrics.md

[17] "Blog - What's New in Prometheus 2.26?," *Blog - What's New in Prometheus 2.26?* https://promlabs.com/blog/2021/04/01/whats-new-in-prometheus-2-26

[18] "Blog - What's New in Prometheus 2.28?," *Blog - What's New in Prometheus 2.28?* https://promlabs.com/blog/2021/06/21/whats-new-in-prometheus-2-28

[19] Prometheus, "HTTP API," *Prometheus*. https://prometheus.io/docs/prometheus/latest/querying/api/#querying-exemplars (accessed Sep. 29, 2021).

[20] V. Seifermann, "Application Performance Monitoring in Microservice-Based Systems," May 2017.

[21] "OpenZipkin · A distributed tracing system." https://zipkin.io/.

[22] "Cloud Native Computing Foundation announces Jaeger graduation," *Cloud Native Computing Foundation*, Oct. 31, 2019. https://www.cncf.io/announcements/2019/10/31/cloud-native-computing-foundation-announces-jaeger-graduation/

[23] openzipkin, "GitHub - openzipkin/zipkin: Zipkin is a distributed tracing system," *GitHub*. https://github.com/openzipkin/zipkin/

[24] jaegertracing, "GitHub - jaegertracing/jaeger: CNCF Jaeger, a Distributed Tracing Platform," *GitHub*. https://github.com/jaegertracing/jaeger

[25] D. Berman, "Zipkin vs Jaeger: Getting Started With Tracing," *Logz.io*, Jul. 12, 2018. https://logz.io/blog/zipkin-vs-jaeger/

[26] B. H. Sigelman *et al.*, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure," Google, Inc., 2010.

[27] "Spring Cloud Sleuth." https://spring.io/projects/spring-cloud-sleuth

[28] openzipkin, "GitHub - openzipkin/brave: Java distributed tracing implementation compatible with Zipkin backend services.," *GitHub*. https://github.com/openzipkin/brave

[29] "The OpenTracing project." https://opentracing.io/

[30] "OpenCensus." https://opencensus.io/

[31] "OpenTelemetry," *OpenTelemetry*. https://opentelemetry.io/

[32] kubernetes, "GitHub - kubernetes/kube-state-metrics: Add-on agent to generate and expose cluster-level metrics.," *GitHub*. https://github.com/kubernetes/kube-state-metrics

[33] Prometheus, "Overview," *Prometheus*. https://prometheus.io/docs/introduction/overview/

[34] "Grafana® Features," *Grafana Labs*. https://grafana.com/grafana/

[35] Prometheus, “Metric types,” *Prometheus*. https://prometheus.io/docs/concepts/metric_types/

[36] Prometheus, “HTTP API,” *Prometheus*. https://prometheus.io/docs/prometheus/latest/querying/api/

[37] “Welcome!,” *minikube*. https://minikube.sigs.k8s.io/docs/

[38] “Running Kubernetes locally on Linux with Minikube - now with Kubernetes 1.14 support,” *Kubernetes*, Mar. 28, 2019. https://kubernetes.io/blog/2019/03/28/running-kubernetes-locally-on-linux-with-minikube-now-with-kubernetes-1.14-support/

[39] “Deployments,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ .

[40] “Service,” *Kubernetes*. https://kubernetes.io/docs/concepts/services-networking/service/

[41] B. Wilson, “How To Setup Prometheus Monitoring On Kubernetes Cluster [Tutorial],” *devopscube*, Apr. 07, 2021. https://devopscube.com/setup-prometheus-monitoring-on-kubernetes/

[42] “Overview of kubectl,” *Kubernetes*. https://kubernetes.io/docs/reference/kubectl/overview/

[43] “ReplicaSet,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

[44] “kubectl Port-Forward - Kubernetes Port Forwarding Guide,” *phoenixNAP*, Feb. 25, 2020. https://phoenixnap.com/kb/kubectl-port-forward

[45] tsenart, “GitHub - tsenart/vegeta: HTTP load testing tool and library. It’s over 9000!,” *GitHub*. https://github.com/tsenart/vegeta

[46] “Kubernetes Components,” *Kubernetes*. https://kubernetes.io/docs/concepts/overview/components/

[47] “Microservices Advantages and Disadvantages: Everything You Need to Know,” *Solace*, Jun. 01, 2020. https://solace.com/blog/microservices-advantages-and-disadvantages/

[48] “Pods,” *Kubernetes*. https://kubernetes.io/docs/concepts/workloads/pods/

[49] “Grafana Tempo,” *Grafana Labs*. https://grafana.com/oss/tempo/