Αριστοτέλειο Πανεπιστήμιο Θεσσαλονίκης
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών &
Μηχανικών Υπολογιστών
Τομέας Ηλεκτρονικής και Υπολογιστών
Εργαστήριο Επεξεργασίας Πληροφορίας
και Υπολογισμών (IPL)

Διπλωματική Εργασία

Σχεδίαση και Ανάπτυξη Συστήματος Ενεργής Παρακολούθησης Διαδικτυακών Εφαρμογών

Εκπόνηση:
Σεντονάς Σταύρος
ΑΕΜ: 9386

Επίβλεψη:
Καθ. Συμεωνίδης Ανδρέας
Δρ. Παπαμιχαήλ Μιχαήλ
Υπ. Δρ. Καρανικιώτης Θωμάς

Θεσσαλονίκη, Μάρτιος 2024

Περίληψη

Η συνεχόμενη ψηφιοποίηση της καθημερινότητας έχει οδηγήσει σε μία πληθώρα διαδικτυακών εφαρμογών που αναπτύσσονται στο πλαίσιο αυτής. Τα δεδομένα αυτά, καθιστούν επιτακτική την ανάγκη ύπαρξης συστημάτων που θα ελέγχουν την εύρυθμη λειτουργία τέτοιων εφαρμογών λογισμικού, που λειτουργούν ως υπηρεσίες (SaaS - Software as a Service).

Ο έλεγχος μπορεί να γίνει σε διάφορα επίπεδα. Από ελέγχους μονάδων (Unit Tests), στο πλαίσιο του κύκλου ανάπτυξης του λογισμικού (continuous integration, continuous deployment cycle), μέχρι την Παρακολούθηση Δικτύου (Network Monitoring), στην πραγματική λειτουργία του συστήματος.

Ενώ υπάρχουν αρκετά εργαλεία για τον έλεγχο εφαρμογών, κατά τη φάση ανάπτυξης του λογισμικού, δεν υπάρχουν αντίστοιχα εύχρηστα και ικανά εργαλεία για έλεγχο στη φάση λειτουργίας αυτού. Η παρούσα, λοιπόν, διπλωματική εστιάζει στην ανάπτυξη ενός συστήματος Έξυπνης Παρακολούθησης και κατ' επέκταση εφαρμογής που θα παρέχει τη δυνατότητα στους χρήστες της να παρακολουθούν εύκολα την ομαλή λειτουργία των διαδικτυακών σελιδών τους, είτε αυτά είναι εφαρμογές, είτε απλά στατικές σελίδες, να κάνουν αναγνώριση και εντοπισμό μη φυσιολογικής λειτουργίας, καθώς και να τροποποιούν αυτόματα, τους διαθέσιμους στα υπολογιστικά συστήματα, πόρους σε πραγματικό χρόνο.

Το σύστημα στηρίζεται στη βασική μέθοδο εντοπισμού διαθεσιμότητας μίας ιστοσελίδας, γνωστή και ως ping. Κάνοντας ping μπορούμε να πάρουμε χρήσιμη πληροφορία σχετικά με το αν το υπό μελέτη σύστημα μπορεί να αποκριθεί ορθά στα αιτήματα που δέχεται, και σχετικά με το χρόνο που χρειάστηκε προκειμένου να απαντήσει. Συνεχίζοντας την λογική πορεία ενός τέτοιου συστήματος μπορούμε ακόμα στο μήνυμα που στέλνουμε να έχουμε πληροφορία που θα επηρεάζει την απάντηση που θα περιμέναμε να δούμε, έχοντας έτσι έναν ακόμα μηχανισμό για την αναγνώριση και αποφυγή πιθανών bugs, ή λαθών κατά τη διαδικασία ανάπτυξης λογισμικού ως υπηρεσία.

Επιπλέον αναλύοντας την ακολουθία των χρονισμών, που παράγεται από τις συνεχείς κλήσεις στο υπό μελέτη σύστημα, μπορούμε να αντλήσουμε σημαντικά στοιχεία για την εξέλιξη της στο χρόνο και έτσι να εντοπίσουμε μη φυσιολογικά σημεία πάνω σε αυτή. Τέλος αν το σύστημα που μελετάμε αποτελεί docker container, προσφέρεται η δυνατότητα δυναμικής τροποποίησεις των διαθέσιμων πόρων, προκειμένου να ανταποκρίνεται και να λειτουργεί με βέλτιστο τρόπο ακόμα και υπό συνθήκες υψηλού φορτίου.

Title

Development of an Active Monitoring System for Web Applications

Abstract

The continuous digitization of daily life, has led to a plethora of internet applications developed within this framework. These data make it imperative, to have systems that will control the smooth operation of such software applications, which function as services (SaaS - Software as a Service).

Control checks can be exerted at various levels, from unit tests within the software development cycle (continuous integration, continuous deployment cycle) to Network Monitoring during the actual system operation.

While there are several tools for application control during the software development phase, there are not similarly many user-friendly and capable tools for control during the operational phase. Therefore, this thesis focuses on developing an Intelligent Monitoring system and, by extension, an application that will allow users to easily monitor the smooth operation of their internet pages, whether they are applications or simple static pages, to recognize and detect abnormal operation and to automatically modify the available computational resources in real-time.

The system relies on the basic method of detecting the availability of a website, known as ping. By pinging, we obtain useful information about whether the system under study can respond correctly to the requests it receives and the time it takes to respond. Continuing the logical progression of such system, we can even modify the message we send with predetermined data, to check the response of the system and verify response, thus having an additional mechanism of identifying and avoiding potential bugs or errors during the software's development process.

Furthermore, by analyzing the sequence of timestamps generated by continuous requests to the system under study, we can extract significant information about its evolution over time, and this way identify abnormal behaviours. Finally, if the system we study is built as a Docker container, we can even dynamically modify the available resources, in order to operate optimally even under high load conditions.

Sentonas Stavros
Electrical & Computer Engineering Department,
Aristotle University of Thessaloniki, Greece

March 2024

Περιεχόμενα

Κατάλογος Σχημάτων

Κατάλογος Πινάκων

Ακρωνύμια Εγγράφου

Παρακάτω παρατίθενται ορισμένα από τα πιο συχνά χρησιμοποιούμενα ακρωνύμια της παρούσας διπλωματικής εργασίας:

RUM ⟶ Real User Monitoring
API ⟶ Application Programming Interface
SaaS ⟶ Software as a Service
HTTP ⟶ Hypertext Transfer Protocol
CRUD ⟶ Create, Read, Update, Delee
GCS ⟶ Google Cloud Storage
CPU ⟶ Central Processing Unit
RAM ⟶ Random Access Memory
RL ⟶ Reinforcement Learning
CNN ⟶ Convolutional Neural Network
LSTM ⟶ Long Short-Term Memory

1
Εισαγωγή

Τα τελευταία χρόνια, ο κλάδος του Διαδικτύου προσεγγίζει ένα μεγαλύτερο κομμάτι ανθρώπων, τόσο από τη μεριά του καταναλωτή όσο και από τη μεριά του παραγωγού. Όσο αφορά τον καταναλωτή οι δυνατότητες που του προσφέρονται μπορούν να διακριθούν στους εξής τομείς:

- Επικοινωνία: το διαδίκτυο παρέχει τη δυνατότητα άμεσης επικοινωνίας μεταξύ μεγάλων αποστάσεων, που δεν περιορίζεται μόνο στο ακουστικό ερέθισμα, αλλά επιτρέπει και την μετάδοση οπτικο-ακουστικής πληροφορίας

- Πρόσβαση Πληροφορίας: ίσως το σημεντικότερο αγαθό που προσφέρει το διαδίκτυο είναι η πληθώρα πληροφορίας που στεγάζει. Μηχανές Αναζήτηση (search engines), Online Βάσεις Δεδομένων (online databases), και άλλου είδους εφαρμογών εκπαιδευτικού χαρακτήρα που δίνουν πρόσβαση σε άτομα που το επιθυμούν, να κάνουν έρευνα

- Ποιότητα ζωής: σε αυτή την κατηγορία περιλαμβάνονται όλες εκείνες οι υπηρεσίες που διευκολύνουν την καθημερινότητα των χρηστών. Online αγορές (eshops) που γλιτώνουν την αναμονή σε ουρές ή ακόμα επιτρέπουν την εύκολη αγορά προϊόντων από απομακρυσμένες περιοχές του πλανήτη, ψυχαγωγία και πρόσβαση σε υπηρεσίες που επιταχύνουν ενέργειες που υπό άλλες περιπτώσεις θα ήταν χρονοβόρες (online banking, πληρωμή λογαριασμών, κρατήσεις ξενοδοχείων/εισητηρίων)

Από τη μεριά του παραγωγού, τα μέσα που υπάρχουν για την ανάπτυξη τέτοιων εφαρμογών/υπηρεσιών/συστημάτων μέρα με τη μέρα αυξάνονται. Η ραγδαία εξέλιξη στον χώρο των cloud υποδομών, καθιστά ευκολότερη και επισπεύδει τόσο την δημιουργία διαδικτυακών εφαρμογών, και σελιδών σε ένα γενικότερο πλαίσιο, όσο και την μεγέθυνση και αύξηση αυτών (scale up). Μάλιστα η επιλογή κατάλληλου παρόχου τέτοιων υπηρεσιών αποτελεί ένα αρκετά σημεντικό αντικείμενο μελέτης

[1]. Πέρα από τον οικονομικό παράγοντα θα πρέπει να προσμετρηθούν οι παροχές, τα πλεονεκτήματα αλλά και η αποδοτικότητα που κάθε ένας προσφέρει.

Βλέποντας λοιπόν το πόσο συνυφασμένη είναι η ζωή του σύγχρονου ανθρώπου με το δίκτυο αλλά και τις δυνατότητες και τα μέσα που έχει ο καθένας για να αναπτύξει εφαρμογές σε αυτό, καθίσταται επιτακτική η ανάγκη ύπαρξης μηχανισμών που θα αναγνωρίζουν σφάλματα (bugs), θα επιβλέπουν την ορθή λειτουργία των υπό μελέτη συστημάτων καθόλη τη διάρκεια ζωής τους και θα διαθέτουν τη δυνατότητα αυτόματης εφαρμογής κατάλληλων ρυθμίσεων για τη βελτιστοποίησή τους

1.1 Περιγραφη του Προβληματος

Η Παρακολούθηση (Monitoring) ενός συστήματος που "ζει" στο χώρο του διαδικτύου μπορεί να γίνει κυρίως με δύο τρόπους:

- **Ενεργή Παρακολούθηση (Active Monitoring)**: έχει περισσότερο προγνωστικό και προληπτικό χαρακτήρα. Συχνά αναφέρεται και ως **Συνθετική παρακολούθηση (Synthetic Monitoring)**, λόγω της φύσης των ενεργειών της. Ουσιαστικά δημιουργεί πλασματικά api calls και όχι πραγματικά δεδομένα χρηστών προκειμένου να ελεγχθεί η απόκριση του υπό μελέτη συστήματος. Η συχνότητα αποστολής των συνθετικών αιτημάτων συνήθως ρυθμίζεται από το χρήστη.

- **Παθητική Παρακολούθηση (Passive Monitoring)**: παρέχει μία πιο πλήρη εικόνα σχετικά με πως χρησιμοποιούνται οι πόροι του δικτύου καταγράφοντας, αποθηκεύοντας και αναλύοντας τα δεδομένα του χρήστη. Για αυτό πολλές φορές αναφέρεται στη βιβλιογραφία ως **Παρακολούθηση Πραγματικών Χρηστών (Real User Monitoring - RUM)**. Έτσι μπορεί κανείς να εντοπίσει τις τάσεις χρήσης του δικτύου για τη βελτίωση και βελτιστοποίησή του συστήματος.

Και οι δύο μέθοδοι στηρίζονται στη συλλογή δεδομένων, είτε εντός είτε εκτός του συστήματος που μελετάμε. Πιο συγκεκριμένα μπορούμε να δούμε τα χαρατηριστικά των δύο μεθόδων στον πίνακα 1.1.

Πιο συγκεκριμένα όσον αφορά την Ενεργή Παρακολούθηση, οι βασικοί λόγοι που κρίνεται επιτακτική η ανάγκη χρήσης τέτοιων υπηρεσιών όπως αναφέρεται και στα [2], [3] είναι οι εξής:

- Βελτίωση προβλημάτων που σχετίζονται με την απόδοση του συστήματος πρωτού τα βιώσουν οι πραγματικοί χρήστες του συστήματος
- Ύπαρξη κάποιας μονάδας αξιολόγησης της απόδοσης του
- Αξιολόγηση του συστήματος υπό μεγαλύτερο φορτίο
- Διασφάλιση της Συμφωνίας Επιπέδου Υπηρεσιών (Service Level Agreement - SLA), μεταξύ του παρόχου υπηρεσιών και των χρηστών

Πίνακας 1.1: Χαρακτηριστικά Ενεργής και Παθητικής Παρακολούθησης

Ενεργή Παρακολούθση (Active Monitoring)	**Παθητική Παρακολούθηση (Passive Monitoring)**
• Στηρίζεται σε συνθετικά API calls	• Αναλύει δεδομένα πραγματικών χρηστών
• Παράγει δεδομένα για συγκεκριμένες πτυχές του δικτύου	• Πλήρης εικόνα της απόδοσης του συστήματος
• Μπορεί να μετρήσει την κίνηση εντός και εκτός του δικτύου	• Μετράει κίνηση μόνο εντός του δικτύου
• Μπορεί να εντοπίσει προβλήματα πριν ακόμα μπορέσουν να τα εντοπίσουν οι χρήστες	• Εντοπίζει προβλήματα που εμφανίζονται εκείνη τη στιγμή

- Παρέχει χρήσιμα δεδομένα ακόμα και σε καινούργια συστήματα που ακόμα μπορεί να μην έχουν χρήστες

Σε ορισμένες περιπτώσεις μάλιστα, μπορούν να συνδυαστούν μέθοδοι Ενεργής και Παθητικής Παρακολούθησης, για να έχουμε μία πιο πλήρη εικόνα του υπό μελέτη συστήματος και να μπορούμε να κάνουμε αλλαγές αυτόματα πάνω στο ίδιο, για τη βελτίωση της απόδοσής του.

1.2 Σκοπος - Συνεισφορα της Διπλωματικης Εργασιας

Η παρούσα διπλωματική εργασία μελετά τη χρήση σύγχρονων τεχνολογιών για τη δημιουργία ενός συστήματος που κατά κύριο λόγο θα αποτελεί έναν μηχανισμό Ενεργής Παρακολούθησης (Active Monitoring) σε συνδυασμό με συγκεκριμένες λειτουργίες Παθητικής Παρακολούθησης (Passive Monitoring), καθώς και μία SaaS (Software as a Service) εφαρμογή που θα παρουσιάζει μέσα από διαγράμματα τα αποτελέσματα της ανάλυσης που κάνει το προαναφερθέν σύστημα. Αξίζει να αναφερθεί ότι για λόγους ''γενίκευσης'' το σύστημα Παθητικής Παρακολούθησης θα αποτελεί μία ξεχωριστή οντότητα που θα συλλέγει δεδομένα μόνο από εφαρμογές που χρησιμοποιούν τεχνολογία Docker για την ανάπτυξή/λειτουργία τους.

Στη συνέχεια θα εξεταστεί ο τρόπος δημιουργίας και υλοποίησης του συστήματος που δημιουργήσαμε, καθώς και τα αποτελέσματα και οι μετρήσεις που παρήχθησαν από τα πειράματα που διενεργήθηκαν.

1.3 Διαρθρωση της Αναφορας

Η διάρθρωση της παρούσας διπλωματικής εργασίας είναι η εξής:

- **Κεφάλαιο 2**: Περιγράφονται τα βασικά εργαλεία και θεωρητικά στοιχεία στα οποία βασίστηκαν οι υλοποιήσεις
- **Κεφάλαιο 3** Αναφορά συστημάτων που ήδη χρησιμοποιούνται και παράθεση διαφορών με την υλοποίησή μας
- **Κεφάλαιο 4** Περιγραφή της υλοποίησης και λειτουργίας του συστήματος
- **Κεφάλαιο 5** Παρουσιάζονται τα αποτελέσματα μετρήσεων που έγιναν καθώς και εικόνες της γραφικής διεπαφής που υλοποιήσαμε.
- **Κεφάλαιο 6** Προτείνονται θέματα για μελλοντική μελέτη, αλλαγές και επεκτάσεις.

2
Θεωρητικό Υπόβαθρο

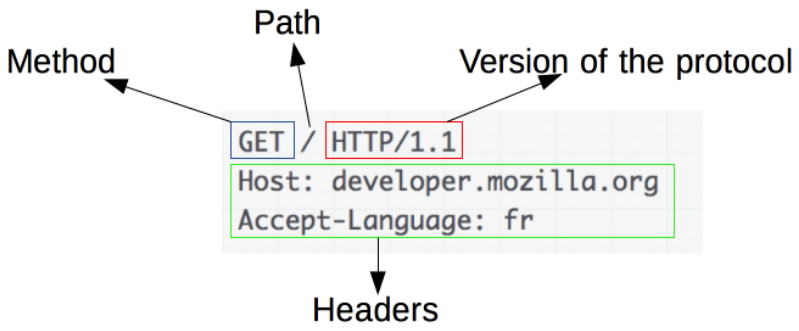
Στο κεφάλαιο αυτό θα παρουσιαστούν εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του Συστήματος Ενεργής Παρακολούθησης, καθώς και έννοιες και τεχνολογίες που αξιοποιήθηκαν για το σκοπό αυτό.

2.1 Hypertext Transfer Protocol

Το πρωτόκολλο επικοινωνίας HTTP (Hypertext Transfer Protocol) αποτελεί το πιο διαδεδομένο και ευρέως γνωστό πρωτόκολλο στο χώρο του διαδικτύου. Αναπτύχθηκε από τους Tim Berners-Lee και την ομάδα του το 1990 και από τότε έχει περάσει πολλές αλλαγές προκειμένου να μπορεί να ανταπεξέλθει στις ολοένα και συνεχώς αυξανόμενες ανάγκες του σήμερα.

Αποτελεί τη βάση κάθε μετάδοσης πληροφορίας στο διαδίκτυο. Στηρίζεται στην επικοινωνία δύο υπολογιστών, ενός που κάνει τα αιτήματα (client) και ενός που απαντά σε αυτά (server). Στο τέλος της επικοινωνίας στην μεριά του παραλήπτη θα υπάρχει ανακατασκευασμένο ένα ολοκληρωμένο αρχείο, από τα διάφορα υποαρχεία που μαζεύτηκαν, που μπορεί να είναι αρχεία ήχου, εικόνας, video. Τα αιτήματα αυτού που ξεκινάει την επικοινωνία ονομάζονται requests, ενώ οι απαντήσεις του αποστολέα responses.

Η βασική δομή ενός http αιτήματος, η οποία φαίνεται και στο σχήμα 2.1, περιληπτικά περιλαμβάνει τη μέθοδο (method) του αιτήματος, που περιγράφει τη βασική λειτουργία του, το μονοπάτι (path) στο οποίο θα επικοινωνήσει με τον server, την έκδοση του πρωτοκόλλου που θα χρησιμοποιηθεί και τέλος headers προκειμένου να κρίνει ο server αν πρέπει να απαντήσει ή όχι πίσω στον client



Σχήμα 2.1: Βασική Δομή ενός αιτήματος http

2.1.1 Μέθοδοι

Πιο συγκεκριμένα οι βασικές μέθοδοι που παρέχει το http και οι συνήθεις λειτουργίες τους είναι οι εξής:

- **GET**: παίρνει πληροφορία από τον server
- **POST**: υποβάλλει πληροφορία, προκαλώντας αλλαγές στον τρόπο λειτουργίας του server. Σχετίζεται συχνά με τη δημιουργία πληροφορίας που προηγουμένως δεν υπήρχε
- **PUT**: όπως και πριν στέλνει πληροφορία στον παραλήπτη υπολογιστή, αλλά αυτή τη φορά επηρεάζει πόρους που ήδη υπήρχαν στο σύστημα. Σχετίζεται συχνά με την τροποποίηση ήδη υπάρχουσας πληροφορίας
- **DELETE**: διαγράφει από το σύστημα του server το συγκεκριμένο πόρο.

Αξίζει να σημειωθεί ότι πέρα από τις τέσσερις αυτές βασικές μεθόδους υπάρχουν και άλλες όπως είναι η **PATCH** που αποτελεί ειδική περίπτωση της PUT, η **HEAD** που αποτελεί ειδική περίπτωση της GET, καθώς και άλλες που σχετίζονται με τη σύνδεση μεταξύ server και client. Αυτές είναι οι **CONNECT**, **OPTIONS** και **TRACE**. Καθώς όμως, οι υπόλοιπες αυτές οι μέθοδοι, δεν χρησιμοποιούνται τόσο συχνά στην πράξη, δεν θα αναλυθούν περαιτέρω.

2.1.2 Εκδόσεις HTTP

Η πρώτη έκδοση του HTTP, παρόλλο που δεν είχε κάποια συγκεκριμένο τίτλο, εκ των υστέρων ονομάστηκε HTTP/0.9. Αποτελεί την πιο απλή έκδοση του πρωτοκόλλου. Δεν υποστηρίζονταν headers και κωδικοί κατάστασης (status codes). Εξυπηρετούσε μόνο GET αιτήματα και η μοναδική απάντηση που μπορούσε να επιστρέψει ήταν hypertext αρχεία. Κάθε φορά που ο server ανταποκρινόταν και έστελνε απάντηση, η επικοινωνία με τον client έκλεινε κατευθείαν.

Στη συνέχεια και με την ανάπτυξη του διαδικτύου προστέθηκαν και άλλες λειτουργίες. Πέριξ του 1996, με τη επόμενη έκδοση του πρωτοκόλλου (HTTP/1.0)

τα αιτήματα πλέον συνοδεύονταν από headers, μεταπληροφορία σχετικά με τη κατάσταση του αιτήματος, τον τύπο της πληροφορίας που περιμένουμε να έρθει (stylesheets, media, hypertext) καθώς και την έκδοση του HTTP που χρησιμοποιήθηκε στη συγκεκριμένη επικοινωνία. Επιπλέον πέρα από τη GET μέθοδο υπάρχει η δυνατότητα για POST και PUT, δημιουργία και τροποποίηση πληροφορίας δηλαδή.

Στη συνέχεια το HTTP/1.1 προσπαθεί να βελτιώσει τις ήδη υπαρχουσες δυνατότητες κάνοντας την επικοινωνία μεταξύ server και client πιο αποδοτική. Αντί να κλείνει η επικοινωνία μετά από κάθε μήνυμα, η σύνδεση παραμένει ανοιχτή γλιτώνοντας έτσι μία σταθερή καθυστέρηση που υπήρχε σε κάθε αίτημα

Φτάνοντας στο σήμερα, μιλάμε για το HTTP/2.0 [4]. Αξιοποιώντας το πρωτόκολλο Speedy (SPDY) που αναπτύχθηκε κάποια χρόνια πριν την κυκλοφορία του, και κτίζοντας πάνω σε αυτό, κατάφερε να μειώσει τους χρόνους επικοινωνίας server-client. Μερικοί από τους τρόπους που επιτυγχάνεται αυτό είναι η μετατροπή του http από text πρωτόκολλο, σε δυαδικό (binary protocoll), επιτρέποντας έτσι χρήση καλύτερων και αποδοτικότερων τεχνικών επικοινωνίας. Επιπλέον συμπιέζει τους headers (header compression) καθώς αποτελούν πληροφορία που επαναλαμβάνεται όταν τα αιτήματα στον server είναι συνεχή. Ο server ακόμα, αποκτά έναν μηχανισμό (server-push) που του επιτρέπει να προωθεί πληροφορία στον client (στην cache του client συγκεκριμένα), που δεν έχει ζητήσει ακόμα, αλλά βάση αυτού που αιτήται, μάλλον θα ζητήσει εντός της ιδίας συνεδρίας.

Τέλος, πρέπει να αναφερθούμε στην τελευταία, αν και όχι ακόμα ευρέως διαδεδομένη, έκδοση HTTP/3.0. Η βασική διαφορά με τους πρωκατόχους του είναι ότι αλλάζει το πρωτόκολλο επικοινωνίας που χρησιμοποιεί όλα αυτά τα χρόνια, από TCP (Transfer Communication Protocoll) σε έναν συνδυασμό UDP (User Datagram Protocoll) και QUIC (Quick UDP Internet Connections), μίας νέας τεχνολογίας που λύνει το πρόβλημα και βελτιστοποιεί τόσο το πρόβλημα της ασφάλειας των επικοινωνιών (TLS handshakes), όσο και της απώλειας πληροφορίας που μπορεί να υπήρχε λόγω UDP, πρωτοκόλλου που είναι γνωστό για την ταχύτερη απόδοσή του σε σχέση με το TCP, αλλά και το γεγονός ότι είναι πιο επιρρεπές σε σφάλματα. Η νέα αυτή έκδοση από τα αποτελέσματα της [5], φαίνεται να έχει ήδη καλύτερους χρόνους σε σχέση με τις παλαιότερες εκδόσεις και ήδη το 28% του διαδικτύου αξιοποιεί τις δυνατότητές του.

2.1.3 Κωδικοί Κατάστασης

Οι κωδικοί κατάστασεις (status codes) αποτελούν μέρος της απάντησης του server. Επιτρέπουν στον χρήστη να καταλάβει με μία ματιά αν το αίτημα που έχει κάνει έχει επιστρέψει σωστά, ή έχει γίνει κάποιο λάθος στη μεριά του server. Υπάρχουν πέντε μεγαλύτερες κατηγορίες που στεγάζουν όλες τις υποπεριπτώσεις αυτών. Πιο συγκεκριμένα:

- **Εύρος 100-199**: Υποδηλώνουν ενημερωτική απάντηση σχετικά με τη λειτουργία του server
- **Εύρος 200-299**: Επιτυχή αιτήματα.

- **Εύρος 300-399**: Υποδηλώνουν την ανακατεύθυνση του μηνύματος του client. Συνήθως συνοδεύονται από το νέο url στο οποίο πρέπει να αποσταλλεί το αίτημα
- **Εύρος 400-499**: Ανεπιτυχές αίτημα, που οφείλεται στον client. Ένα σύνηθες παράδειγμα είναι η αίτηση πρόσβασης σε προστατευόμενους πόρους χωρίς κάποιου είδους αυθεντικοποίηση, ή χωρίς τα σωστά στοιχεία για αυθεντικοποίηση
- **Εύρος 500-599**: Ανεπιτυχές αίτημα, που οφείλεται στον server.

2.2 API

Ένα ακόμα πολύ σημαντικό συστατικό του διαδικτύου αποτελούν οι Διεπαφές Εφαρμογών Προγραμμάτων (Application Programming Interfaces - APIs). Πρόκειται για το σύνολο των ορισμών, κανόνων και πρωτοκόλλων για τη δημιουργία και ενσωμάτωση μίας εφαρμογής. Ουσιαστικά λειτουργεί ως το συμβόλαιο μεταξύ ενός συστήματος παροχής υπηρεσίας και του χρήστη του συστήματος αυτού, καθορίζοντας την απαραίτητη πληροφορία που απαιτεί για να λειουργήσει σωστά ο server αλλά και αντίστροφά, καθορίζοντας την απαραίτητη πληροφορία που απαιτεί ο client στην απάντηση που θα του επιστραφεί.

Είναι εμφανές λοιπόν ότι σε κάθε περίπτωση, αν θέλει κανείς να αλληλεπιδράσει με κάποιο σύστημα είτε για να αντλήσει πληροφορία, είτε για να στείλει πληροφορία (αποθήκευση ή τροποποίηση ήδη υπάρχουσας) θα πρέπει να υπάρχουν κανόνες που ορίζουν και καθιστούν πιο εύκολη και απλή την επικοινωνία αυτή.

Η έννοια του api δεν περιορίζεται φυσικά μόνο στο πλαίσιο του διαδικτύου, αλλά σε όλων των ειδών εφαρμογές που υπάρχει επικοινωνία ενός κεντρικού σημείου (server) με κάποιον χρήστη (client) που θέλει να κάνει χρήση των υπηρεσιών που αυτό προσφέρει.

Όσων αφορά τη διαθεσιμότητα και ασφάλεια των APIs, μπορούμε να διακρίνουμε τις εξής κατηγορίες:

- **Ανοιχτά (Open)**: Σε αυτού του τύπου διεπαφών λογισμικού, έχει ελεύθερη πρόσβαση ο καθένας, χωρίς να απαιτείται κάποια παραπάνω πληροφορία που αφορά την αυθεντικοποίηση ή ταυτοποίηση του χρήστη. Επειδή ακριβώς η πληροφορία που παρέχεται (ανοιχτά) είναι τεράστια, χρησιμοποιούνται πολύ συχνά σε έρευνες, όπως η [6] που αξιοποιεί ελεύθερα προσβάσιμους πόρους για να αναλύσει τα πιο διαδεδομένα APIs.
- **Δημόσια (Public)**: Μοιάζουν πολύ με τα **Ανοιχτού Τύπου API**, με τη μόνη διαφορά, τον περιορισμό πρόσβασης σε ορισμένα σημεία που απαιτούν κλειδιά προκειμένουν να γίνει, σε αντίθεση με πριν, αυθεντικοποίηση και ταυτοποίηση.
- **Ιδιωτικά (Private)**: Αφορούν διεπαφές λογισμικού που αναπτύσσονται και χρησιμοποιούνται μόνο εντός ενός κλειστού πλαισίου, όπως θα μπορούσε να είναι μία επιχείρηση ή κάποιο πανεπιστήμιο που παρέχει ορισμένες υπηρεσίες

μόνο εντός του χώρου του. Πολλές φορές στη βιβλιογραφία αναφέρονται και ως **Εσωτερικά (Internal APIs).**

- **Εταιρικά (Partner)**: Είναι περιορισμένα στο πλήθος χρηστών που έχουν πρόσβαση σε αυτό. Χρησιμοποιούνται για επικοινωνία μεταξύ συστημάτων εταιριών/επιχειρήσεων για την ανάπτυξη και αξιοποίηση εφαρμογών και υπηρεσιών. Συνήθως η ασφάλεια σε όλες αυτές τις αλληλεπιδράσεις είναι αρκετά πιο αυστηρή.

- **Σύνθετα (Composite)**: Συνδυάζουν περισσότερα από ένα API αιτημάτα σε ένα, κάνοντας έτσι την επικοινωνία πιο αποδοτική (κερδίζοντας χρόνο από πολλά διαδοχικά APIs).

Αξίζει να σημειωθεί σε αυτό το σημείο ότι ο τρόπος δημιουργίας, η δομή, καθώς και τα πρωτόκολλα που χρησιμοποιούν τα διαφόρα APIs που υπάρχουν στο διαδίκτυο δεν είναι πάντα κοινά. Υπάρχουν κάποιες γνωστές αρχιτεκτονικές και πρωτόκολλα που πέρα από τη δομή των συστημάτων, παρέχουν και κανόνες "Καλύτερων Πρακτικών" (Best Practices). Μία από τις πιο γνωστές και πλέον διαδεδομένες αποτελεί η REST api αρχιτεκτόνική (Representational State Transfer), τα χαρακτηριστικά της οποίας θα μελετήσουμε στη συνέχεια.

2.2.1 RESTful API

Πρόκειται για ένα τύπο αρχιτεκτονικής λογισμικού για APIs, που αποτελείται από κατευθυντήριες γραμμές και βέλτιστες πρακτικές για τη δημιουργία επεκτάσιμων εφαρμογών στο διαδίκτυο. Προτάθηκε το 2000, από τον Thomas Fielding [7], σαν ένας τρόπος για να κατευθύνει την ανάπτυξη εφαρμογών προκειμένου να υπάρχει ένα κοινός τρόπος "γραφής", ώστε να υπάρξη εξέλιξη στον τομέα αυτό ακόμα πιο γρήγορα και οργανωμένα. Το δομικό συστατικό του είναι το πρωτόκολλο HTTP που χρησιμοποιεί για να πραγματοποιεί κλήσεις μεταξύ συστημάτων.

Όπως προαναφέρθηκε ο τρόπος λειτουργίας στην απλούστερη έκδοσή του ξεκινάει με ένα αίτημα κάποιου χρήστη σε ένα σύστημα της επιλογής του, παρέχοντας την απαραίτητη πληροφορία προκειμένου ο server να μπορέσει να αναταποκριθεί επιτυχώς. Έπειτα, και αφού το σύστημα διεκπεραιώσει όλες τις εσωτερικές λειτουργίες του, απαντά μεταφέροντας πίσω στον χρήστη την επιθυμητή πληροφορία και ενημερώνοντας τον σχετικά με την κατάσταση του αιτήματος του (σε περίπτωση που κάτι πήγε λάθος ο χρήστης θα πρέπει να ενημερώνεται αναλόγως).

Ο βασικός τρόπος αλληλεπίδρασης με τους πόρους του συστήματος θα πρέπει να γίνεται μέσα από τις τέσσερις βασικές μεθόδους του HTTP (**GET**, **POST**, **PUT**, **DELETE**), χωρίς αυτό να απαγορεύει τη χρήση των υπόλοιπων. Κάθε σύστημα θα πρέπει τυπικά να υποστηρίζει CRUD λειτουργίες (Create, Read, Update, Delete). Αυτές σχετίζονται με την ύπαρξη μεθόδων εντός του συστήματος που θα επιτρέπουν τη δημιουργία δεδομένων και την τροποποίηση, ανάγνωση, και διαγραφή ήδη υπάρχουσας πληροφορίας.

Για να χαρακτηριστεί ένα API ώς RESTful θα πρέπει να ικανοποιεί τα παρακάτω, ενώ στο σχήμα 2.2 μπορούμε να τα δούμε σε ένα γενικότερο πλαίσιο:

- **Client-Server**: Τα δύο αυτά υπολογιστικά συστήματα θα πρέπει να είναι ανεξάρτητα. Πρακτικά αυτό σημαίνει ότι ο client θα ασχοληθεί αποκλειστικά και μόνο με το κομμάτι παρουσιάσης της πληροφορίας στον χρήστη, μέσα από διεπαφές γραφικού χαρακτήρα, ενώ ο server θα εκτελεί μόνο λειτουργίες που αφορούν την δημιουργία, τροποποίηση και διαγραφή των πόρων του συστήματος. Με αυτό τον διαχωρισμό πλέον ο client είναι πιο ελαφρής, καθώς διαχειρίζεται μόνο πληροφορία που του έρχεται έτοιμη από το σύστημα και ο server μπορεί να κάνει πιο εύκολα scaling.

- **Cacheable**: ο client θα πρέπει να αποθηκεύει προσωρινά (cache) τις απαντήσεις που λαμβάνει από τον server για την αποφυγή συνεχών επαναλαμβανόμενων αιτημάτων, βελτιώνοντας έτσι την απόδοση του συστήματος.

- **Stateless**: Η πληροφορία της κατάστασης του συστήματος δεν θα πρέπει να αποθηκεύεται, αλλά κάθε αίτημα θα πρέπει να περιέχει την απαραίτητη πληροφορία για να εκτελεστεί από τη μεριά του server. Η πληροφορία αυτή μπορεί να αποτελεί μέρος του ίδιου του url, του body, των headers ή του query.

- **Πολυεπίπεδο Σύστημα (Layered System)**: Τα υπολογιστικά συστήματα που συμμετέχουν στη διαδικασία αυτή δεν θα πρέπει να γνωρίζουν αν είναι άμεσα συνδεδεμένα μεταξύ τους ή υπάρχουν ενδιάμεσοι κόμβοι που παρεμβάλλονται. Το σύστημα θα πρέπει να λειτουργεί δίχως να έχει επίγνωση των γειτόνων του, περιμένοντας απλά την κατάλληλη πληροφορία για να λειτουργήσει, τόσο από τη μεριά του server όσο και από τη μεριά του client.

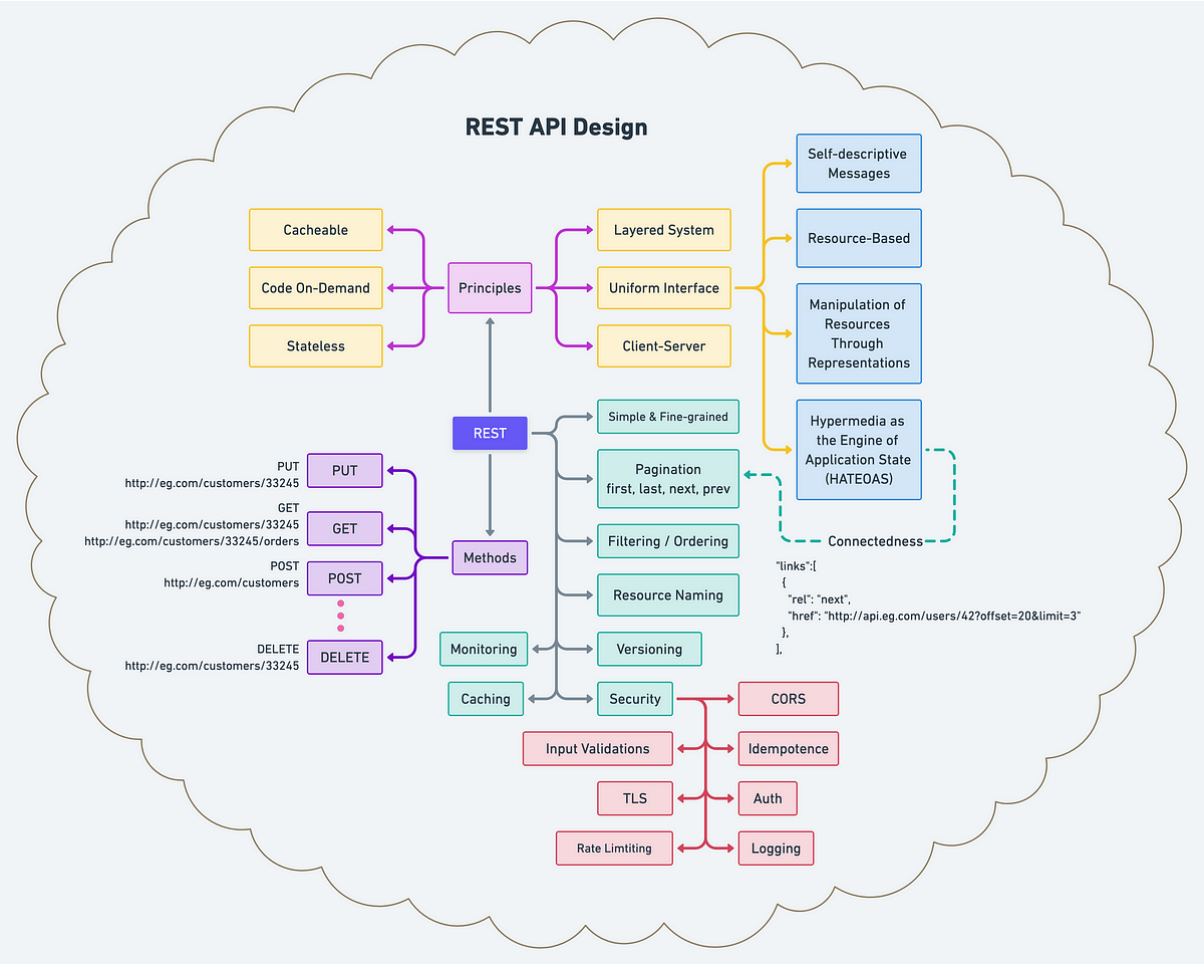
2.3 LSTM

Το Μοντέλο Long Short-Term Memory (LSTM) αποτελεί ένα είδος αναδρομικών νευρωνικών δικτύων, σχεδιασμένο για την αντιμετώπιση προβλημάτων που συνδέονται με την ανάλυση και την πρόβλεψη χρονοσειρών. Το LSTM model διαθέτει μια αναδρομική δομή που του επιτρέπει να διατηρεί και να χρησιμοποιεί πληροφορίες από προηγούμενες χρονικές στιγμές. Αυτό είναι κρίσιμο για την αντιμετώπιση μακροπρόθεσμων εξαρτήσεων στα δεδομένα.

Το LSTM έχει μια αναδρομική δομή, που σημαίνει ότι μπορεί να αποθηκεύει πληροφορίες από προηγούμενες χρονικές στιγμές. Κάθε μονάδα LSTM διαθέτει τρεις βασικές πύλες για τον έλεγχο της ροής των πληροφοριών:

1. Πύλη Εισόδου (Input Gate): Η πύλη εισόδου καθορίζει ποιες νέες πληροφορίες θα αποθηκευτούν στην κυτταρική μνήμη. Είναι υπεύθυνη για την επιλογή του ποσοστού πληροφοριών που θα εισέλθουν στον κυτταρικό χώρο. Η διαδικασία αυτή συμβάλλει στη δημιουργία νέων μνημονικών αποτυπωμάτων με βάση τα νέα δεδομένα που εισέρχονται.

2. Πύλη Εξόδου (Output Gate): Η πύλη λήθης είναι υπεύθυνη για την απόφαση, του ποιες πληροφορίες θα ”ξεχαστούν” ή θα διαγραφούν από την κυτταρική μνήμη. Αυτή η λειτουργία βοηθά στο να αποφεύγεται η συσσώρευση ανεπιθύμητων πληροφοριών που δεν είναι πλέον σημαντικές για την πρόβλεψη.



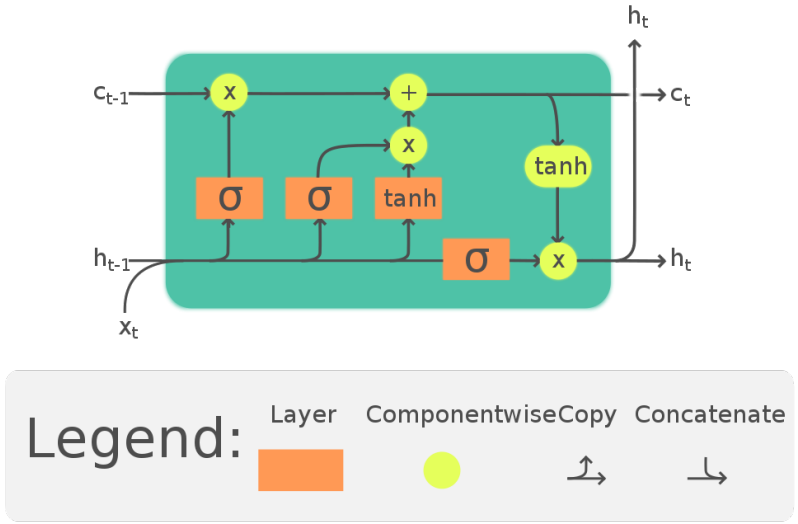
Σχήμα 2.2: Αρχές και Καλές Πρακτικές Σχεδίασης REST API

3. Πύλη Λήθης (Forget Gate): Η πύλη εξόδου αποφασίζει ποια πληροφορία από την κυτταρική μνήμη θα εξαχθεί ως τελική έξοδος της μονάδας LSTM. Ελέγχει το ποσοστό της μνημονικής πληροφορίας που θα συνεισφέρει στο τελικό αποτέλεσμα. Αυτή η πύλη καθορίζει την τελική έξοδο της LSTM μονάδας.

Η διαχείριση αυτών των πυλών επιτρέπει στο LSTM να διατηρεί και να χρησιμοποιεί μακροπρόθεσμες εξαρτήσεις στα δεδομένα, καθιστώντας το ιδιαίτερα αποτελεσματικό στη χρονοσειριακή ανάλυση και πρόβλεψη.

Κάθε κύτταρο του LSTM περιέχει μνήμη. Η μνήμη κυττάρου είναι υπεύθυνη για την αποθήκευση και τη διατήρηση των πληροφοριών. Αυτή η δομή επιτρέπει στο μοντέλο να αντιλαμβάνεται και να ανταποκρίνεται σε μακροπρόθεσμες συναρτήσεις στα δεδομένα. Με τον τρόπο αυτό, το LSTM model εκπαιδεύεται να αναγνωρίζει προτεραιότητες, τάσεις και μοτίβα στα δεδομένα χρονοσειρών, παρέχοντας ένα πολύ ισχυρό εργαλείο για την πρόβλεψη μελλοντικών τιμών και την ανίχνευση ανωμαλιών.

Η εφαρμογή του LSTM model στην παρακολούθηση συστημάτων προσφέρει ποικίλα οφέλη που απορρέουν από την εξελιγμένη του λειτουργία. Η αρχιτεκτονική του LSTM model επιτρέπει την ακριβή πρόβλεψη των επιπέδων φόρτου του συστήματος. Αυτό είναι ιδιαίτερα σημαντικό για την πρόληψη υπερφόρτωσης και την



Σχήμα 2.3: Σχηματική απεικόνιση Long-Short Term Memory cell [8]

αποτελεσματική διαχείριση των πόρων. Ένα από τα ισχυρά σημεία του LSTM model είναι η ικανότητά του να προσαρμόζεται αυτόματα σε νέες συνθήκες, διατηρώντας την απόδοσή του σε ποικίλες μεταβαλλόμενες συνθήκες. Χάρη στην αναδρομική δομή, τις πύλες και τη διαχείριση μνήμης κάθε cell, το LSTM model αποτελεί ιδανική επιλογή για παρακολούθηση χρονοσειρών. Με τον τρόπο αυτό, το LSTM model χρησιμοποιείται σε πρόβλεψη σεισμών, την παρακολούθηση κίνησης οχημάτων εκτός δρόμου, την πρόβλεψη κίνησης τροφοδοσίας σε συστήματα ηλεκτρικής ενέργειας, στην πρόβλεψη κυκλοφοριακής συμφόρησης αλλά και στην πρόβλεψη κατανάλωσης ενέργειας σε κτίρια.

2.4 Τεχνολογιες που χρησιμοποιηθηκαν

Στην τελευταία υποενότητα αυτού του κεφαλαίου θα μιλήσουμε για τα εργαλεία που χρησιμοποιήθηκαν για την υλοποίηση του συστήματος Αυτόματης Παρακολούθησης που υλοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας.

2.4.1 Node.js

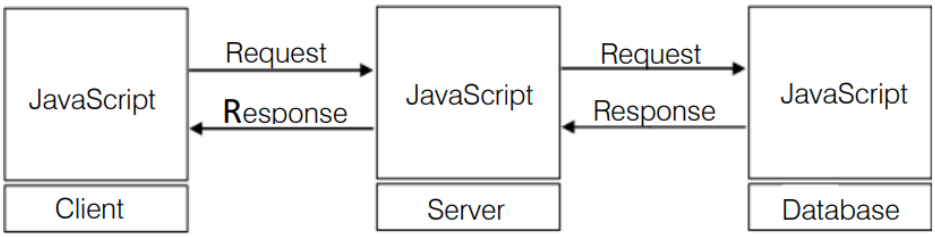
Είναι ένα περιβάλλον εκτέλεσης (runtime) της προγραμματιστικής γλώσσας JavaScript. Πληθώρα μηχανικών λογισμικού το χρησιμοποιούνε σήμερα, για τη γρήγορη και, σχετικά με άλλα εργαλεία, εύκολη ανάπτυξη διαδικτυακών εφαρμογών. Η Node.js μάλιστα διαθέτει το δικό της package manager (NPM - Node Package Manager), στον οποίο διαρκώς προστίθενται καινούργιες βιβλιοθήκες από χρήστες για χρήστες.

Είναι σχεδιασμένο για να μπορεί να χτίζει κλιμακούμενες (scalable) εφαρμογές, καθώς η αρχιτεκτονική του, επιτρέπει τη σύνδεση πολλών εξωτερικών συστημά-

των/χρηστών και την εξυπηρέτηση αυτών ταυτόχρονα. Κάθε φορά που γίνεται κάποια σύνδεση στην εφαρμογή εκτελείται μία callback συνάρτηση προκειμένου να μπορέσει να απαντήσει πίσω. Όσο το σύστημα δεν δέχεται αιτήματα "κοιμάται" και περιμένει το επόμενο που θα έρθει.

Όλα όσα προαναφέρθηκαν την καθιστούν κατάλληλη για τη δημιουργία συστημάτων server. Αξίζει να σημειωθεί μάλιστα ότι για το σκοπό αυτό υπάρχουν πλήθος βιβλιοθηκών που παρέχουν έτοιμες συναρτήσεις και διεπαφές που κάνουν την διαδικασία ανάπτυξης ακόμα πιο εύκολη και γρήγορη. Πολλές φορές μάλιστα, κάποιες βιβλιοθήκες πέρα από βοηθητικές συναρτήσεις και υπορουτίνες, επηρεάζουν τον τρόπο συγγραφής κώδικα, μέσα από APIs που παρέχουν. Αυτές χαρακτηρίζονται ως frameworks, και επιταχύνουν τόσο τη διαδικασία ελέγχου (testing), όσο και τη διαδικασία ανάπτυξης (development) του λογισμικού. Μερικά από τα πιο γνωστά backend framerworks είναι τα: Express, Jest, Koa, Socket.io, Meteor, Loopback

Τα παραπάνω, σε συνδυασμό με το ότι η JavaScript είναι μία ευρέως διαδεδομένη και πολυχρησιμοποιούμενη γλώσσα προγραμματισμού στο διαδίκτυο, δίνει την ευκαιρία σε developers να αναπτύξουν μία εφαρμογή πλήρως (frontend και backend) με τη χρήση ενός κοινού εργαλείου. Ένα τυπικό παράδειγμα εφαρμογής που μπορεί να αναπτυχθεί με τον τρόπο αυτό φαίνεται στο σχήμα 2.4.



Σχήμα 2.4: Σχεδίαση συστήματος με χρήση Node.js [9]

2.4.2 PM2

Η διαχείρηση διεργασιών (processes) που τρέχουν σε ένα υπολογιστικό σύστημα μπορεί να είναι δύσκολη και αρκετά περίπλοκη για κάποιον που δεν διαθέτει μεγάλη πείρα σε αυτό τον τομέα. Το PM2, ή αλλιώς Process Manager 2, αποτελεί ένα έργο λογισμικού ανοιχτού κώδικα (open source project), που κάνει την παραπάνω διαδικασία πιο απλή και κατανοητή, κερδίζοντας χρόνο για τον developer που μπορεί να τον αξιοποιήσει για την ανάπτυξη εφαρμογών.

Με την έννοια διαχείρηση διεργασιών, αναφερόμαστε σε όλες εκείνες τις ενέργειες που σχετίζονται με τον (πρόωρο ή όχι) τερματισμό και την παρακολούθηση ήδη υπάρχοντων διεργασιών (σχήμα 2.5), αλλά και τη δημιουργία καινούργιων. Προγράμματα όπως το pm2 προσφέρουν ακόμα δυνατότητες όπως είναι η αυτόματη επανεκκίνηση διεργασιών σε περίπτωση σφάλματος αποτρέποντας έτσι την ύπαρξη downtime των εφαρμογών που τρέχουμε.

Ένα από τα πιο χρήσιμα εργαλεία που παρέχει το PM2 είναι η λειτουργία cluster mode. Αν και δεν αναφέρθηκε πιο πάνω, ο συγκεκριμένος διαχειριστής διεργασιών

εξειδικεύεται σε Node.js εφαρμογές και στη διαχείρηση αυτών. Εξ ορισμού, εφαρμογές γραμμένες σε JavaScript τρέχουν σε ένα νήμα (thread) στο υπολογιστικό σύστημα που εκτελούνται. Χρησιμοποιώντας όμως το cluster mode, μπορούμε να ξεκινήσουμε πολλαπλές διεργασίες που θα λειτουργούν ταυτόχρονα και θα μοιράζουν το φόρτο κατάλληλα (load-balancing) ώστε το τελικό σύστημα να έχει καλύτερη απόδοση και να μπορεί να εξυπηρετεί περισσότερο κόσμο.

id	name	namespace	version	mode	pid	uptime	↺	status	cpu	mem	user	watching
0	lychte-server	default	N/A	fork	10432	66s	0	online	0%	66.4mb	stavs	disabled
1	phiphisus-worker	default	N/A	fork	33936	66s	0	online	0%	68.9mb	stavs	disabled
2	sisiphus-worker	default	N/A	fork	34312	66s	0	online	0%	68.7mb	stavs	disabled

Σχήμα 2.5: Παράδειγμα Παρακολούθησης διεργασιών με τη χρήση του PM2

2.4.3 MongoDB

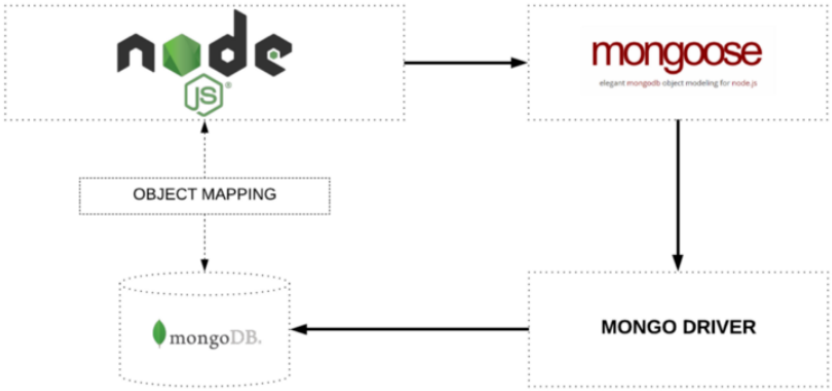
Η MongoDB αποτελεί μία Μη Σχεσιακή Βάση Δεδομένων (Non Relational Database), που χρησιμοποιεί documents, για να αποθηκεύσει πληροφορία, αντί στήλες και γραμμές όπως θα είχαμε σε κλασσικές SQL βάσεις δεδομένων. Είναι σχεδιασμένη για αποθήκευση δεδομένων μεγάλης κλίμακας και παράλληλη επεξεργασία δεδομένων μοιρασμένα σε έναν μεγάλο αριθμό από servers.

Τα documents που αναφέρθηκαν αποτελούν τον ακρογωνιαίο λίθο της Mongo, καθώς αποτελούν τη βασική μονάδα της αποθηκευμένης πληροφορίας. Μορφοποιούνται ως BJSON (Binary JSON) και παρέχουν πληθώρα τύπων δεδομένων που μπορούν να αποθηκευτούν (string, integer, double, boolean, array, object, date, timestamp, null, binary). Κάθε βάση mongo μπορεί να περιέχει μία ή περισσότερες συλλογές (collections), τα δεδομένα των οποίων πρέπει να υπακούουν στο ίδιο σχήμα (schema). Επειδή όμως η βάση αυτή παρέχει δυνατότητες δυναμικού σχήματος (dynamic schema) μπορούμε να αποθηκεύουμε πληροφορία και να προσθέτουμε πεδία (fields) που δεν είχαμε ορίσει από την αρχή δημιουργίας του συστήματος.

Μερικοί από τους λόγους που την επιλέξαμε είναι οι εξής:

- **Document Oriented Storage**: η αποθήκευση και διαχείριση των δεδομένων (στο πλαίσιο της εφαρμογής) είναι εύκολη καθώς στηρίζεται σε δεδομένα σε μορφή JSON.
- **Ευρετήρια (Indexes)**: Υπάρχει η δυνατότητα, κατά τη διαδικασία του στησίματος της βάσης (αλλά και μετέπειτα), να οριστούν ευρετήρια σε πεδία που χρησιμοποιούνται συχνά σε queries, προκειμένου να βελτιωθεί η απόδοση του συστήματος στο σύνολο. Όσο πιο γρήγορη είναι βάση, τόσο καλύτερη θα είναι η ανταπόκριση του συστήματος.
- **Ομοιοτυπία (Replication) και μεγάλη Διαθεσιμότητα**: Δημιουργώντας πολλαπλά αντίτυπα των αποθηκευμένων δεδομένων σε περισσότερους από έναν servers, μπορούμε να είμαστε σίγουροι ότι θα έχουμε πρόσβαση στην πληροφορία ακόμα και αν η κίνηση (data traffic) της εφαρμογής αυξηθεί.

- **Αυτόματο sharding**: Διασπώντας την αποθηκευμένη πληροφορία και έχοντας τμήματα αυτής σε διαφορετικούς server μας δίνεται η δυνατότητα να κάνουμε οριζόντιο scaling πολύ πιο εύκολα.
- **Εύκολη ενσωμάτωση**: Η mongo διαθέτει βιβλιοθήκες στο περιβάλλον του node.js που καθιστούν τη χρήση και το στήσιμό της πολύ απλά. Πέρα από τη mongo, την επίσημη βιβλιοθήκη που υπάρχει, διατίθεται και η mongoose που αποτελεί μία Object Data Modelling (ODM) βιβλιοθήκη για τη Mongo και τη nodejs (σχήμα 2.6)



Σχήμα 2.6: Η mongoose λειτουργεί ώς ένα abstract layer μεταξύ της Node και των driver της mongo προκειμένου η επικοινωνία μεταξύ των δύο να γίνεται πιο εύκολα

2.4.4 Google Cloud Storage

Πέρα από την κλασική βάση δεδομένων που προαναφέρθηκε θέλουμε να αποθηκεύουμε ιστορικά δεδομένα, δεδομένα δηλαδή που δεν θα εμφανίζονται άμεσα στον χρήστη, αλλά θα χρησιμοποιούνται για τον υπολογισμό μετρικών και στατιστικά σημαντικών αποτελεσμάτων, στο σύνολο όλης της μέχρι τώρα αποθηκευμένης πληροφορίας.

Σε αυτό λοιπόν το σημείο έρχεται το Google Cloud Storage (GCS), το οποίο παρέχει μεταξύ άλλων, δυνατότητες Αποθήκευσης Αρχείων (File Storage). Τα δεδομένα αυτά δεν χρειάζεται να υπακούουν σε κάποιο σχήμα, ενώ παράλληλα ο τρόπος αρχειοθέτησης των δεδομένων ταυτίζεται με ένα κλασικό σύστημα αρχείων ενός υπολογιστικού συστήματος. Διαθέτει paths και τύπους αρχείων όπως ακριβώς και οι υπολογιστές και όλες οι συσκευές που χρησιμοποιούμε. Για να διαβάσει κανείς πληροφορία, χρειάζεται να ξέρει μόνο το μονοπάτι που οδηγεί στο επιθυμητό αρχείο, καθώς και τη μορφή αυτού. Οι υποστηριζόμενοι τύποι αρχείων μέχρι τώρα είναι οι εξής:

- Binary
- Flat
- JSON
- Avro
- Parquet

2.4.5 Rust Language

Η Rust αναδεικνύεται ως μια σύγχρονη γλώσσα προγραμματισμού που έχει κερδίσει δημοτικότητα για την ασφαλή και αποτελεσματική ανάπτυξη συστημάτων. Η Rust παρέχει ένα πλαίσιο που ενσωματώνει σημαντικές έννοιες για τη δημιουργία αξιόπιστου και αποτελεσματικού κώδικα.

Παρακάτω αναπτύσσονται οι βασικές έννοιες της Rust:

- Ownership: Η Rust χρησιμοποιεί ένα σύστημα ιδιοκτησίας για τη διαχείριση της μνήμης. Αυτό σημαίνει ότι κάθε κομμάτι δεδομένων έχει έναν ''ιδιοκτήτη'', που είναι υπεύθυνος για την απελευθέρωση της μνήμης όταν δεν χρειάζεται πλέον [10].
- Borrowing: Μηχανισμός κατά τον οποία τμήματα δεδομένων δανείζονται στιγμιαία σε άλλο σημειο του κώδικα χωρίς να μεταφέρεται το ownership [11]. Με τον τρόπο αυτό, μειώνεται η ανάγκη παρουσίας πολλών αντιγράφων των δεδομένων σε διαφορετικά σημεία του κώδικα. Ο δανεισμός δεδομένων πραγματοποιείται σε 2 σημεία. Τα mutable borrows αναφέρονται σε πρόσβαση με δικαιώματα read και write. Τα immutable αναφέρονται σε πρόσβαση με δικαιώματα read.

- Lifetimes: Οι διάρκειες ζωής καθορίζουν το χρονικό διάστημα κατά το οποίο είναι έγκυρες οι αναφορές (references) των δεδομένων [10]. Αυτό βοηθά στην αποφυγή dangling references (αναφορές προς μνήμη που έχει ήδη απελευθερωθεί) και εξασφαλίζει τη σωστή χρήση των δεδομένων από τις αναφορές.

Η Rust υποστηρίζει τη συναρτησιακή προγραμματιστική παραδειγματική, που περιλαμβάνει υψηλή τάξη συναρτήσεων, κλεισίματα (closures), και μονάδες (monads).

- Closures (Κλεισίματα): Τα κλεισίματα στη Rust είναι σαν ανώνυμες συναρτήσεις που μπορούν να αποθηκευτούν και να περάσουν ως παράμετροι σε άλλες συναρτήσεις. Χρησιμοποιούνται για να δημιουργούνται ευέλικτες λογικές και να αποτρέπουν την ανάγκη για κώδικα πολλαπλών γραμμών.

- Pattern Matching: Η Rust χρησιμοποιεί το pattern matching για να αντιστοιχίσει τα δεδομένα σε διάφορα πρότυπα, γεγονός που είναι κοινό στη συναρτησιακή προγραμματιστική

- Iterators (Επαναληπτές): Οι επαναληπτές στη Rust επιτρέπουν τη διατριβή μέσα σε συλλογές δεδομένων με στυλ συναρτησιακού προγραμματισμού.

Η προγραμματιστική γλώσσα Rust προσφέρει μια πληθώρα πλεονεκτημάτων:

- Ασφάλεια Μνήμης: Το σύστημα ιδιοκτησίας (ownership) και ο δανεισμός (borrowing) επιτρέπουν στη Rust να αποφεύγει σχεδόν πλήρως τα σφάλματα μνήμης όπως οι ανεξέλεγκτοι δείκτες και η διαρροή μνήμης. [11].

- Παράλληλος Προγραμματισμός: Ο παράλληλος προγραμματισμός είναι σχετικά εύκολος στη Rust λόγω του συστήματος ιδιοκτησίας και της ενσωματωμένης υποστήριξης για πολλά νήματα.

- Χαμηλό Κόστος Αφαιρετικότητας: Η Rust προσφέρει high level abstractions χωρίς να προσθέτει τον επιπλέον χρόνο εκτέλεσης που συνήθως συνοδεύει τις γλώσσες προγραμματισμού με αφαιρετική τυπολογία.Αυτό επιτυγχάνεται με τη χρήση των λειτουργιών:

  - Zero-cost abstractions: Τα μοτίβα type safety, generics και pattern matching δεν προσθέτουν επιπλέον χρόνο.
  - No garbage collection: Αυτό επιτυγχάνεται μέσω του συστήματος borrowing και lifetime.

- Performance: Η Rust σχεδιάστηκε για να είναι αποτελεσματική, προσφέροντας χαμηλή ανάλυση και εκτέλεση κοντά στον κώδικα γραμμένο σε C και C++.

- Κοινότητα και Εργαλεία: Η Rust έχει μια ενεργή κοινότητα και υποστηρίζεται από ένα ισχυρό οικοσύστημα βιβλιοθηκών και εργαλείων ανάπτυξης.

2.4.6 Python

Η Python είναι μια υψηλού επιπέδου γλώσσα προγραμματισμού γνωστή για την απλότητα και την αναγνωσιμότητά της [12]. Έχει σχεδιαστεί με σκοπό την ευκολία στη κατανόηση και στη συγγραφή, δίνοντας έμφαση στην αναγνωσιμότητα του κώδικα μέσω της χρήσης indentation αντί για αγκύλες ή λέξεις-κλειδιά. Η αυτόματη διαχείριση μνήμης της Python ενισχύουν την παραγωγικότητα των προγραμματιστών επιτρέποντας πιο γρήγορους κύκλους ανάπτυξης.

Η απήχηση της Python στην επιστήμη των δεδομένων υποστηρίζεται από το εκτεταμένο εύρος εξειδικευμένων βιβλιοθηκών και frameworks. Κυρίως το NumPy ξεχωρίζει για τον αποτελεσματικό χειρισμό αριθμητικών υπολογισμών, παρέχοντας τη βάση για προηγμένες μαθηματικές πράξεις και χειρισμούς πινάκων [13]. Συνδυαστικά, η pandas διευκολύνει το χειρισμό και την ανάλυση δεδομένων, προσφέροντας δομές δεδομένων υψηλού επιπέδου. Οι δυνατότητες οπτικοποίησης των δεδομένων ενισχύονται από τα εργαλεία Matplotlib και Seaborn, παρέχοντας τη δυνατότητα για τη δημιουργία περίπλοκων αναπαραστάσεων των δεδομένων [13]. Αυτά τα εργαλεία είναι καθοριστικά για τη μετάδοση σύνθετων εννοιών και ιδεών. Επιπλέον, η ενσωμάτωση της Python με το Jupyter, ένα διαδραστικό υπολογιστικό περιβάλλον, ενθαρρύνει μια συστηματική και συνεργατική προσέγγιση στην εξερεύνηση και ανάλυση δεδομένων.

Συγκεκριμένα, στο πεδίο της μηχανικής μάθησης, το scikit-learn είναι μια βιβλιοθήκη για την εφαρμογή αλγορίθμων και στατιστικών μοντέλων. Η σύνταξη της Python είναι συνοπτική στην έκφραση περίπλοκων εννοιών μηχανικής μάθησης, προωθώντας μια βελτιωμένη διαδικασία συλλογιστικής πορείας και ανάπτυξης κώδικα. Επιπλέον, μέσα από την ένταξη της βαθιάς μάθησης δημιουργήθηκαν εργαλεία, όπως το TensorFlow και το PyTorch. Αυτά τα frameworks επιτρέπουν στους ερευνητές να εφαρμόζουν και να πειραματίζονται με αρχιτεκτονικές νευρωνικών δικτύων για δραστηριότητες όπως η ταξινόμηση εικόνων αλλά και η επεξεργασία φυσικής γλώσσας.

Η ευελιξία της Python ενισχύεται από τη διαλειτουργικότητά της με βάσεις δεδομένων, web API και άλλες πηγές δεδομένων. Αυτό το χαρακτηριστικό είναι ιδιαίτερα ωφέλιμο στην επιστήμη δεδομένων, όπου είναι αναγκαίος ο συνδυασμός εργαλείων. Με αυτόν τον τρόπο, επιτρέπει απρόσκοπτες διαδικασίες εξαγωγής, μετασχηματισμού και φόρτωσης δεδομένων (ETL), διευκολύνοντας την ενσωμάτωση διαφορετικών ροών δεδομένων. Η υποστήριξη του RESTful API ενισχύει περαιτέρω τις δυνατότητές της στην πρόσβαση και το χειρισμό δεδομένων από υπηρεσίες web.

Ο συνεργατικός χαρακτήρας της κοινότητας αποτελεί ένα επιπλέον ωφέλιμο χαρακτηριστικό, το οποίο παρέχει λύσεις, αλλά και ενισχύει ένα περιβάλλον που ευνοεί την ανταλλαγή γνώσης. Επομένως, διασφαλίζονται γρήγορες ενημερώσεις, διορθώσεις σφαλμάτων και εξελίσσονται γοργά τα εργαλεία της επιστήμης δεδομένων για την αντιμετώπιση των αναδυόμενων προκλήσεων.

Tensorflow

Το TensorFlow, μια βιβλιοθήκη μηχανικής μάθησης ανοιχτού κώδικα, αντιπροσωπεύει τον ακρογωνιαίο λίθο στη σύγχρονη έρευνα της επιστήμης δεδομένων και της τεχνητής νοημοσύνης. Κεντρικό στοιχείο του σχεδιασμού του είναι η έννοια των

υπολογιστικών γραφημάτων, όπου οι λειτουργίες είναι κόμβοι και η ροή δεδομένων αναπαρίσταται μέσω κατευθυνόμενων ακμών. Αυτό το παράδειγμα διευκολύνει αποτελεσματικά τον παράλληλο υπολογισμό, ένα κρίσιμο πλεονέκτημα στις περίπλοκες αρχιτεκτονικές νευρωνικών δικτύων οι οποίες επικρατούν στη σύγχρονη μηχανική μάθηση.

Η βιβλιοθήκη ασχολείται με τον χειρισμό τανυστών, δηλαδή πολυδιάστατων πινάκων που χρησιμεύουν ως θεμελιώδη δομή δεδομένων. Η ικανότητα του TensorFlow να ενσωματώνεται απρόσκοπτα με το Keras, ένα API νευρωνικών δικτύων υψηλού επιπέδου, παρέχει μια φιλική προς το χρήστη διεπαφή για την κατασκευή και την εκπαίδευση πολύπλοκων μοντέλων, μειώνοντας έτσι το κόστος υλοποίησης [14].

Η εξέλιξη του TensorFlow στην έκδοση 2.0 εισήγαγε την αυτόματη εκτέλεση, μια αλλαγή από την αρχική προσέγγιση του στατικού γραφήματος. Η αυτόματη εκτέλεση επιτρέπει την άμεση αξιολόγηση των λειτουργιών, βελτιώνοντας τη διαδραστικότητα του κώδικα και απλοποιώντας τη διαδικασία εντοπισμού σφαλμάτων. Επιπλέον, υπάρχει υποστήριξη για κατανεμημένους υπολογιστές, δίνοντας τη δυνατότητα στους χρήστες να αξιοποιήσουν την υπολογιστική ισχύ πολλών CPUs ή GPUs για την εκπαίδευση μοντέλων μεγάλης κλίμακας. Τέλος, κατά την ανάπτυξη, το TensorFlow ενσωματώνει την αρχιτεκτονική του μοντέλου και τα βάρη με τρόπο ανεξάρτητο από την πλατφόρμα.

Ενισχύεται η διαφάνεια μέσα από το TensorBoard, μια εργαλειοθήκη οπτικοποίησης που διευκολύνει την παρακολούθηση και την απεικόνιση μετρήσεων σε πραγματικό χρόνο κατά τη διάρκεια της εκπαίδευσης των μοντέλων. Τέλος, έχουν δημιουργηθεί διάφορα ακόμη περιβάλλοντα, όπως το TensorFlow Lite για κινητές συσκευές και ενσωματωμένες συσκευές και το TensorFlow.js για τη μηχανική εκμάθηση μέσω προγραμμάτων περιήγησης [15].

2.4.7 Apache Kafka Queues

Το Apache Kafka αποτελεί ένα κατανεμημένο σύστημα ανταλλαγής μηνυμάτων που έχει σχεδιαστεί για να αντιμετωπίζει μεγάλο όγκο δεδομένων και να εξασφαλίζει την αντοχή και την απόδοση σε περιβάλλοντα πραγματικού χρόνου [16].

Τα κύρια χαρακτηριστικά της δομής του συστήματος Kafka είναι:

- Θέματα (Topics): Τα δεδομένα οργανώνονται σε θέματα, κάθε ένα εκ των οποίων αντιπροσωπεύει μια κατηγορία. Τα μηνύματα στέλνονται σε ένα ειδικό θέμα και διανέμονται σε όλους τους καταναλωτές.

- Καταναλωτές και Παραγωγοί: Οι παραγωγοί είναι υπεύθυνοι για την αποστολή μηνυμάτων σε ένα θέμα, ενώ οι καταναλωτές λαμβάνουν μηνύματα από τα θέματα και επεξεργάζονται τα δεδομένα.

- Διαμοιρασμός (Partitioning): Τα θέματα διαιρούνται σε κομμάτια, τα partitions. Αυτό επιτρέπει την κατανεμημένη αποθήκευση και επεξεργασία των δεδομένων.

Σχετικά με την λειτουργία του Kafka, σημειώνονται τα παρακάτω στοιχεία λειτουργικότητας:

- Θέματα (Topics): Αποστολή Δεδομένων (Producer): Ο παραγωγός αποστέλλει μηνύματα σε ένα ή περισσότερα θέματα. Τα μηνύματα είναι ανεξάρτητα και αποθηκεύονται σε partitions.
- Διανομή Δεδομένων (Brokers): Οι Kafka Brokers αναλαμβάνουν τη διαχείριση των partitions, διανέμοντας τα μηνύματα στους καταναλωτές. Κάθε partition μπορεί να έχει αντίγραφα για αντοχή σε αποτυχία.
- Λήψη Δεδομένων (Consumer): Οι καταναλωτές εγγράφονται σε ένα ή περισσότερα θέματα και λαμβάνουν τα μηνύματα. Κάθε καταναλωτής είναι υπεύθυνος για το δικό του partition.
- Συγκράτηση και Διατήρηση Δεδομένων (Retention): Τα μηνύματα διατηρούνται για καθορισμένο χρονικό διάστημα ή μέχρι να φτάσουν σε ένα συγκεκριμένο μέγεθος, εξασφαλίζοντας τη συγκράτηση δεδομένων.

Τα μηνύματα αποθηκεύονται με συγκράτηση, διασφαλίζοντας ότι δεν θα χαθούν ακόμη και σε περίπτωση αποτυχίας. Το Kafka είναι εύκολα κλιμακούμενο, διαχειρίζεται υψηλό όγκο δεδομένων και μπορεί να επεκταθεί οριζόντια. Υποστηρίζει τη συμπίεση δεδομένων, βελτιώνοντας την απόδοση και μειώνοντας την κατανάλωση εύρους ζώνης.

Συνολικά, το Apache Kafka αποτελεί ισχυρή λύση για τη διαχείριση ροών δεδομένων σε περιβάλλοντα υψηλής κλίμακας και αξιοπιστίας. Η αρχιτεκτονική του επιτρέπει τη διαχείριση μεγάλων όγκων δεδομένων σε πραγματικό χρόνο [17]. Το Kafka προσφέρει δυνατότητ

α αποθήκευσης αντιγράφων για εξασφάλιση αντοχής σε αποτυχία. Τα δεδομένα αποθηκεύονται κατανεμημένα σε Brokers, εξασφαλίζοντας ανθεκτικότητα και δυνατότητα εύκολης οριζόντιας επέκτασης.

Το Kafka υποστηρίζει τρία επίπεδα σημασιολογίας: ακριβής, τουλάχιστον ακριβής μία φορά, και ακριβής μία φορά με διατήρηση σειράς. Αυτό επιτρέπει στους παραγωγούς και τους καταναλωτές να επιλέξουν το επίπεδο συνοχής που ταιριάζει με τις απαιτήσεις τους. Επίσης, προσφέρει τα Σημεία Ελέγχου (Checkpoints). Οι καταναλωτές μπορούν να διατηρούν σημεία ελέγχου για το πού έχουν διαβάσει μηνύματα από κάθε partition, εξασφαλίζοντας ότι δεν χάνουν δεδομένα κατά τη διαδικασία επεξεργασίας. Σχετικά με τη διαχείριση απόδοσης, προσφέρεται η Υποστήριξη Συνόλων Ερωτήσεων (Query Sets). Το Kafka Streams API επιτρέπει τη δημιουργία εφαρμογών επεξεργασίας ροών δεδομένων μέσω συνόλων ερωτήσεων, που επιτρέπουν την αναγνώριση των αναγκών επεξεργασίας και ανάκτησης δεδομένων. Τέλος, χρήσιμο εργαλείο αποτελεί η Παρακολούθηση Επίδοσης. Το Kafka παρέχει εργαλεία παρακολούθησης και διαχείρισης, όπως το Kafka Manager, που επιτρέπουν στους διαχειριστές να παρακολουθούν την απόδοση του συστήματος και να λαμβάνουν μέτρα για τυχόν προβλήματα.

2.4.8 Docker

Το Docker είναι μια πλατφόρμα ανοιχτού κώδικα που αυτοματοποιεί τη διαδικασία ανάπτυξης, διαχείρισης και εκτέλεσης εφαρμογών σε ελαφριά, φορητά containers. Τα containers παρέχουν ένα συνεχές και απομονωμένο περιβάλλον για την εγκατάσταση εφαρμογών και dependencies, διασφαλίζοντας ότι εκτελούνται με συνέπεια σε διάφορα υπολογιστικά περιβάλλοντα.

Στον πυρήνα του, το Docker χρησιμοποιεί μια αρχιτεκτονική του μοντέλου πελάτη - διακομιστή, όπου ο Docker Daemon διαχειρίζεται containers στο παρασκήνιο και ο Docker Client χρησιμεύει ως διεπαφή. Τα Docker images, τα οποία αποτελούν δομικά στοιχεία των containers, δεν επιτρέπουν την επεξεργασία αλλά μόνο την ανάγνωση και ενσωματώνουν εφαρμογές, βιβλιοθήκες και dependencies, επιτρέποντας την αποτελεσματική κοινή χρήση και διανομή των στοιχείων σε διάφορα περιβάλλοντα [18]. Αυτά τα images κατασκευάζονται χρησιμοποιώντας Dockerfiles. Το Dockerfile είναι ένα αρχείο κειμένου που περιέχει οδηγίες για τη δημιουργία ενός Docker image. Καθορίζει το base image, τον κωδικό εφαρμογής, τα dependencies και τη διαμόρφωση.

Τα containers εκτελούν μεμονωμένες διεργασίες, μοιράζονται το kernel του λειτουργικού συστήματος, αλλά διατηρούν ξεχωριστά συστήματα αρχείων και διεργασίες [19]. Επιπλέον, υπάρχει το Docker Hub, το οποίο χρησιμεύει ως κοινό αποθετήριο για πρόσβαση σε προκατεσκευασμένα Docker images. Άλλο ένα σημαντικό χαρακτηριστικό είναι τα Docker volumes, τα οποία χρησιμοποιούνται για τη διατήρηση δεδομένων εκτός του συστήματος αρχείων των containers. Με αυτόν τρόπο, επιτρέπεται η κοινή χρήση δεδομένων μεταξύ containers, τη διατήρηση δεδομένων στις επανεκκινήσεις και τον αποτελεσματικό χειρισμό μεγάλων συνόλων δεδομένων.

Το Docker, επιπρόσθετα, χρησιμοποιεί εργαλεία διαχείρισης όπως το Docker Compose για τον καθορισμό και την εκτέλεση εφαρμογών πολλαπλών containers και το Kubernetes ή το Docker Swarm για τη διαχείριση και την επεκτασιμότητα των εφαρμογών σε περιβάλλοντα παραγωγής.

Τέλος, λόγω και της δομής του εργαλείου, το Docker ενσωματώνει διάφορα χαρακτηριστικά ασφαλείας, όπως την απομόνωση των containers με χρήση namespaces, περιορισμό των πόρων και τη δυνατότητα σάρωσης των images για τρωτά σημεία χρησιμοποιώντας εργαλεία όπως το Clair ή το Trivy.

Το Docker μπορεί να χρησιμοποιηθεί αποτελεσματικά στην παρακολούθηση φόρτου για να παρέχει πληροφορίες σχετικά με τη χρήση των πόρων, το χρόνο λειτουργίας και την απόκριση των εφαρμογών σε containers [20]. Επιτρέπει στους χρήστες να αναπτύσσουν προσαρμοσμένα σενάρια παρακολούθησης εντός των containers. Αυτά τα σενάρια μπορούν να συλλέγουν μετρήσεις για συγκεκριμένες εφαρμογές, δεδομένα καταγραφής ή να ενσωματωθούν με εξωτερικά συστήματα παρακολούθησης [20]. Παράλληλα, υποστηρίζει health checks, τα οποία μπορούν να χρησιμοποιηθούν για την επαλήθευση της υγείας ενός container εκτελώντας συγκεκριμένες εντολές ή ελέγχοντας την κατάσταση ορισμένων σημείων.

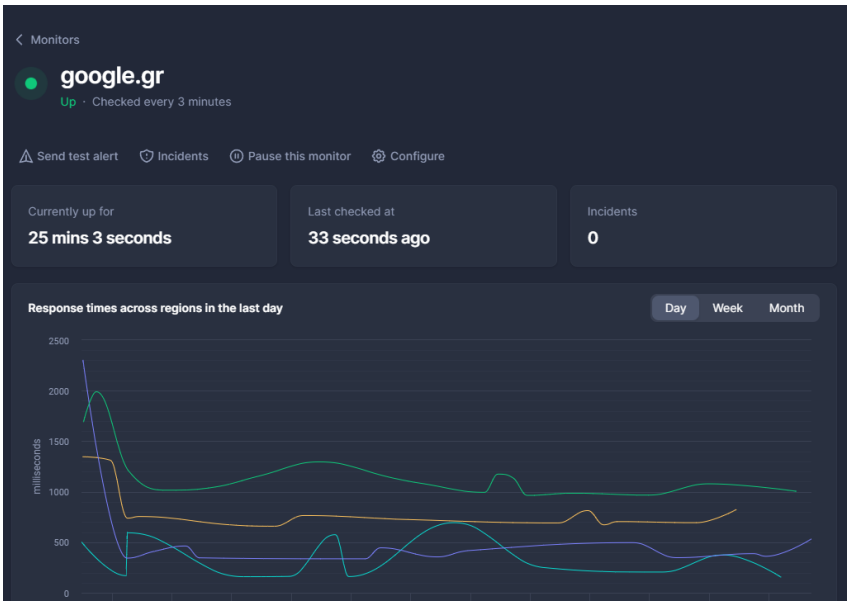
3
Βιβλιογραφική Αναζήτηση Τεχνολογιών Αιχμής

Πριν αναλυθεί η υλοποίηση του συστήματος που δημιουργήσαμε για την Ενεργή Παρακολούθηση Εφαρμογών ως Υπηρεσίες (SaaS) και ιστοσελιδών που εδρεύουν στο διαδίκτυο, θα πασουσιάσουμε τεχνολογίες που έχουν χτιστεί ήδη για τον σκοπό αυτό και θα δείξουμε τους τομείς στους οποίους διαφέρει το δικό μας σύστημα.

Εφαρμογές τέτοιου τύπου έχουν αναπτυχθεί κυρίως από εταιρίες, αλλά υπάρχουν πολλά open source projects μικρότερου βεληνεκούς που επιτυγχάνουν τον ίδιο στόχο. Στη συνέχεια θα αναφερθούμε κυρίως στα πιο γνωστά και διαδεδομένα εργαλεία, παρουσιάζοντας τα δυνατά τους σημεία και περιγράφοντας τις λειτουργίες που παρέχουν.

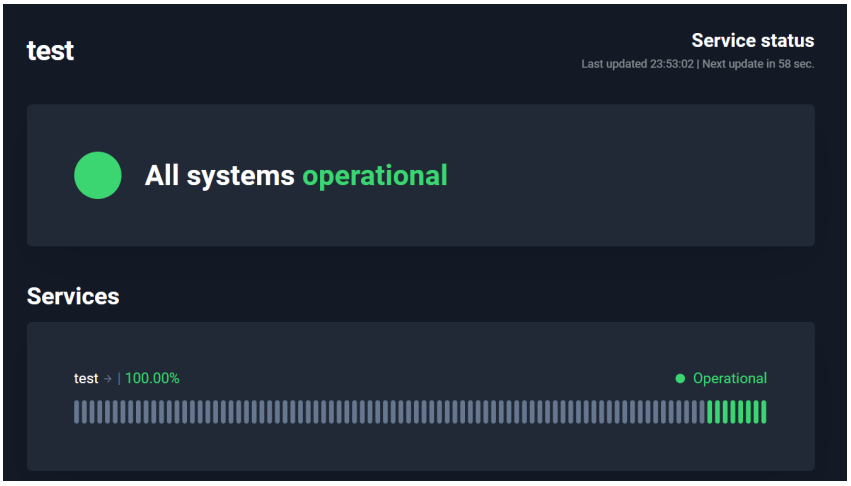
- **Better Uptime**: Προσφέρει εύκολη ενσωμάτωση των ιστοσελιδών που θέλει κανείς να παρακολουθήσει. Λειτουργεί κάνοντας ping κάθε τριάντα δευτερόλεπτα στο url που ορίζει ο χρήστης και παρουσιάζει τα παραγόμενα δεδομένα σε ευπαρουσίαστα διαγράμματα (σχήμα 3.1). Ένα από τα μεγάλα πλεονεκτήματα που έχει αφορά τη δυνατότητα για πολλαπλά ping από διαφορετικές περιοχές του κόσμου (Ευρώπη, Ασία, Βόρεια Αμερική, Αυστραλία), ώστε οι χρήστες να διαθέτουν μία πιο πλήρη εποπτεία του υπό μελέτη συστήματος/ιστοσελίδας. Αξίζει να σημειωθεί ότι παρέχει και μηχανισμούς ενημέρωσης για να ειδοποιεί το χρήστη σε περίπτωση μη απόκρισης τους συστήματος, μέσα από mail, εφαρμογές chatting και τηλεφωνικών κλήσεων.

- **Uptime Robot**: Επιτρέπει, πέρα από την επιλογή του url, και την επιλογή παραπάνων παραμέτρων που επηρεάζουν την απάντηση που θα επιστρέψει το υπό μελέτη σύστημα. Οι παράμετροι αυτοί σχετίζονται με τους headers του μηνύματος που αποστέλλεται, και πιο συγκεκριμένα, με αυτούς που αφορούν την αυθεντικοποίηση του χρήστη (στην προκειμένη περίπτωση του συστήματος παρακολούθησης). Πέρα από αυτά μπορεί να καθορίσει την επιθυμητή



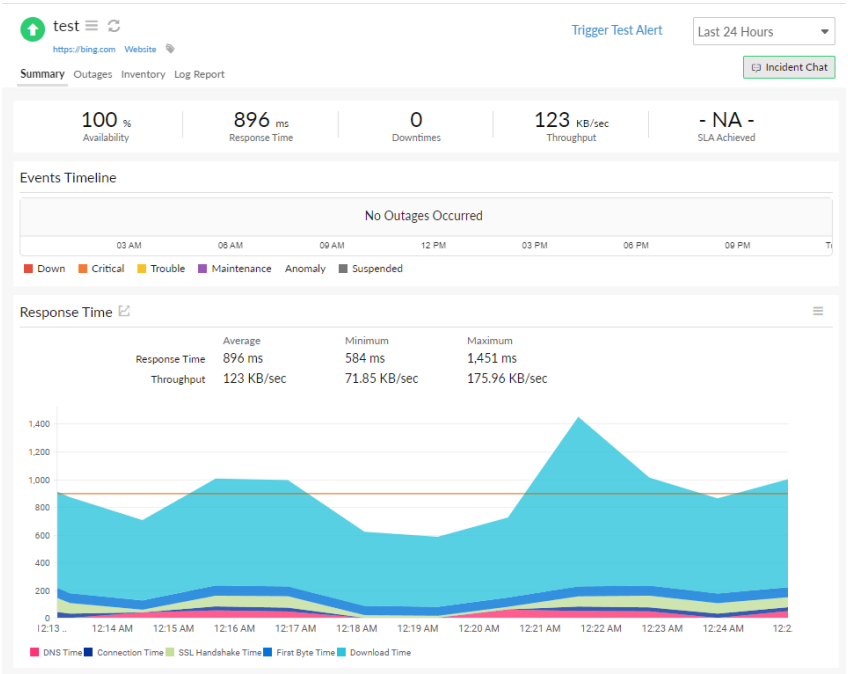
Σχήμα 3.1: Παράδειγμα χρήσης του εργαλείου Better Uptime

http κατάσταση της απόκρισης της ιστόσελίδας και το χρόνο που θα παρεμβάλλεται μεταξύ διαδοχικών αιτημάτων (pings). Τέλος, διαθέτει κάποια βασικά διαγράμματα που σχετίζονται με το αν η απόκριση του συστήματος είναι ορθή ή όχι (σχήμα 3.2).



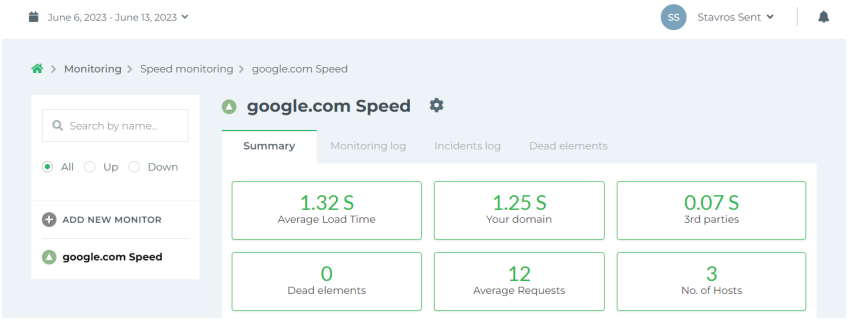
Σχήμα 3.2: Παράδειγμα χρήσης του εργαλείου Uptime Robot

- **Site24x7**: Διαθέτει μετρικές, που αφορούν τη μέγιστη/ελάχιστη τιμή του χρόνου απόκρισης του συστήματος, καθώς και τη μέση τιμή του, ενώ παράλληλα δίνει μία εικόνα του throughput του συστήματος. Δεν λείπει φυσικά και ένα διάγραμμα απόκρισης χρόνου (σχήμα 3.3) που καθιστα τα δεδομένα που συλλέγονται πιο εύκολα στην κατανόηση και οπτικοποίηση.
- **Uptimia**: Πέρα από τα κλασικού τύπου http αιτήματα, μπορεί να κάνει ελέγχους παρακολούθησης (uptime monitoring) σε DNS, UDP, TCP και email με



Σχήμα 3.3: Παράδειγμα χρήσης του εργαλείου Site24x7

απόσταση έως και τριάντα δευτερολέπτων (μεταξύ αιτημάτων). Αξίζει, να σημειωθεί, ότι η συγκεκριμένη εφαρμογή έχει δυνατότες και παθητικής παρακολούθησης (RUM). Αρχικά επιλέγεται το site το οποίο ο χρήστης θέλει να παρακολουθήσει, καθώς και τα δεδομένα για το οποία επιθυμεί να ενημερώνεται ή να παρακολουθεί. Αυτά σχετίζονται, κυρίως με σφάλματα ή καταστάσεις στις οποίες βρίσκεται το σύστημα και μπορεί να δηλώνουν κάποιο πρόβλημα. Καταστάσεις όπως είναι η μειωμένη απόδοση του υπό μελέτη συστήματος ή η απότομη πτώση του πλήθους των χρηστών μίας σελίδας. Για να επιτύχει τέτοιας μορφής ελέγχους, παράγει (ανάλογα με τον τύπο των ελέγχων που επιλέγουμε) ένα script γραμμένο σε JavaScript που τοποθείται στην αρχή της ιστοσελίδας την οποία θέλουμε να παρακολουθήσουμε. Με αυτό τον τρόπο δίνεται η δυνατότητα συλλογής δεδομένων πραγματικών χρηστών.

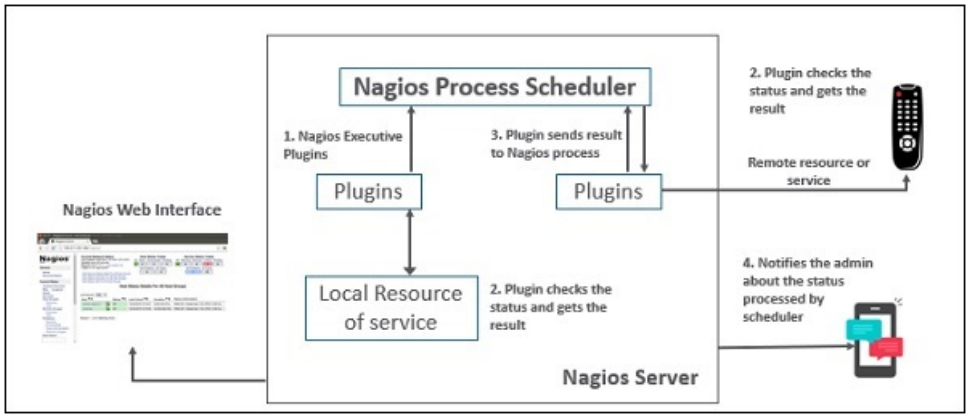


Σχήμα 3.4: Παράδειγμα χρήσης του εργαλείου Uptimia

Παραπάνω αναφέρθηκαν μερικά μόνο κάποια από τα εργαλεία που υπάρχουν σήμερα για την Ενεργη Παρακολούθηση του Χρόνου και Απόκρισης Ιστοτόπων και Διαδικτυακών Εφαρμογών. Σε αυτά θα πρέπει να προστεθούν πληθώρα εφαρμογών όπως τα: **StatusCake**, **SemaText**, **Uptrends**, **Dotcom-monitor**, **Updown**, **Datadog Synthetics**. Οι δυνατότες που προσφέρουν ως επί το πλείστον μπορούν περιγραφούν πλήρως από όσα αναπτύξαμε προηγουμένως, για αυτό το λόγω δεν θα αναφερθούμε περαιτέρω.

Πρέπει σε αυτό το σημείο όμως, να τονίσουμε ότι όσα προαναφέρθηκαν αποτελούν προϊόντα εταιριών. Αυτό όμως δεν σταματάει την ανάπτυξη open source projects που υλοποιήθηκαν από χρήστες είτε ως προσωπικά projects, είτε ως projects μίας μεγαλύτερης ομάδας από developers. Έτσι και στο πλαίσιο της Ενεργής Παρακολούθησης μερικά από τα πιο γνωστά και δημοφιλή μεταξύ developers αποθετήρια αποτελούν τα:

- Upptime[1]: Αποτελεί μία ενδιαφέρουσα προσέγγιση στο πρόβλημα της διαχείρησης των schedulers που θα πρέπει να έχει το σύστημα για να κάνει αιτήματα ανά ένα συγκεκριμένο και σταθερό χρονικό διάστημα. Προκειμένου να επιτύχει κάτι τέτοιο αξιοποιεί τις δυνατότητες του GitHub (cloud-based υπηρεσία αποθήκευσης git αποθετηρίων) και των actions που αυτό σου επιτρέπει να εκτελείς κάθε πέντε λεπτά. Έτσι λοιπόν ανά πέντε λεπτά (ελάχιστος χρόνος ελέγχου απόκρισης) τρέχει αυτοματοποιημένα και μέσω του GitHub (ανεξάρτητα από το σύστημα αυτό) μία διαδικασία που κάνει αιτήματα στην σελίδα που ο χρήστης ορίζει. Φυσικά τα αποτελέσματα αυτά αποθηκεύονται και ανά έξι ώρες παράγονται διαγράμματα που μπορείς να τα δεις μέσα από μία σελίδα που παράγεται αυτόματα.



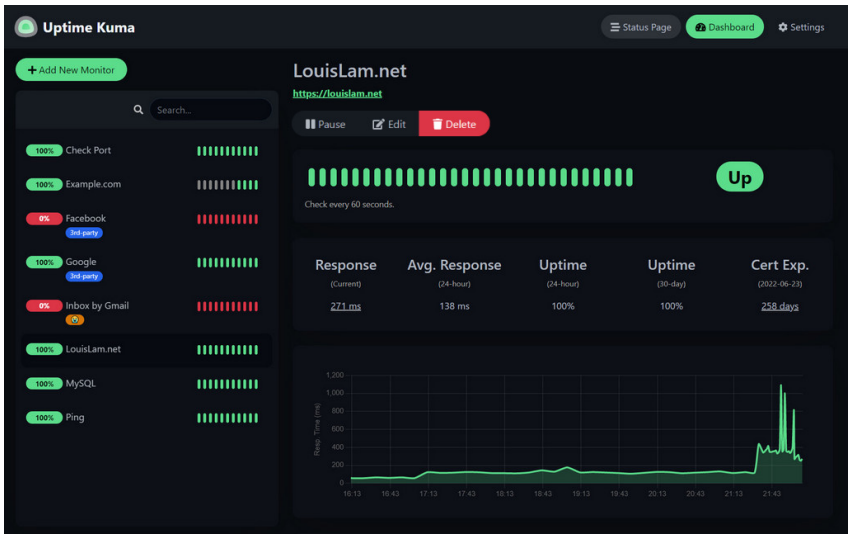
Σχήμα 3.5: Αρχιτεκτόνική Συστήματος Nagios

- Nagios[2]: Είναι ένα πρόγραμμα γραμμένο στη γλώσσα προγραμματισμού C. Πέρα από το backend διαθέτει και Γραφικό Περιβάλλον Χρήστη (Graphical User Interface - GUI). Αποτελείται από ένα σύστημα ενός server (nagios server), ο οποίος λειτουργεί σαν ένας scheduler που στέλνει σήματα για να ξεκινήσει

[1]αποθετήριο κώδικα Upptime: *https://github.com/upptime/upptime*
[2]αποθετήριο κώδικα Nagios: *https://github.com/NagiosEnterprises/nagioscore*

την εκτέλεση plugins στα απομακρυσμένα συστήματα που θέλουμε να παρακολουθήσουμε. Μόλις τα plugins δεχτούν απάντηση επιστρέφουν στον server ο οποίος προωθεί την πληροφορία στο GUI για να τη δούνε και οι χρήστες. Δεν χρησιμοποιείται κάποια βάση δεδομένων, καθώς η πληροφορία που μαζεύεται αποθηκεύεται μόνο σε logs στο σύστημα που τρέχει την υπηρεσία αυτή. Την αρχιτεκτονική του συστήματος μπορούμε δούμε στο σχήμα 3.5.

- Kuma Uptime[3]: Αποτελεί μία εύκολή, στη χρήση και στήσιμο, self-hosted εφαρμογή γραμμένη σε JavaScript για Ενεργή Παρακολούθηση. Για την αποστολή των αιτημάτων σε απομακρυσμένες (στο διαδίκτυο) σελίδες χρησιμοποιεί **child_proccesses**, ένα api δήλαδή που περιλαμβάνει η Node.js για τη δημιουργία διεργασιών (processes) εντός άλλων διεργασιών. Η εφαρμογή περιλαμβάνει backend και frontend, στο οποίο εμφανίζονται τα αποτελέσματα του monitoring. Αξίζει να σημειωθεί ότι τα δεδομένα αποθηκεύονται σε βάση SQLite, ένα προσωρινό σύνολο δεδομένων που υφίσταται μόνο στο πλαίσιο εκτέλεσης μίας εφαρμογής. Αποτελεί μία server-less βάση δεδομένων και είναι άρρηκτα συνδεδεμένη με την εφαρμογή στην οποία υπάρχει.



Σχήμα 3.6: Παράδειγμα χρήσης του εργαλείου Kuma Uptime

Κάθε σύστημα που αναφέρθηκε έχει πλεονεκτήματα αλλά και μειονεκτήματα σε σχέση με άλλα. Αυτό που θα θέλαμε να διαθέτει ιδανικά ένα σύστημα Παρακολούθησης, βάσει όλων αυτών που είδαμε μέχρι τώρα, είναι τα εξής:

1. Σταθερό και Αξιόπιστο σύστημα διαχείρησης προγραμματισμού (scheduling system) που θα καθορίζει το πότε πρέπει να ξεκινάνε τα αιτήματα προς τα εξωτερικά υπό παρακολούθηση συστήματα.

2. Κάποια μορφή βάσης δεδομένων στην οποία θα αποθηκεύουμε τα δεδομένα που συλλέγουμε.

[3]αποθετήριο κώδικα Kuma Uptime: *https://github.com/louislam/uptime-kuma*

3. Χρήση ιστορικών δεδομένων για τον υπολογισμό μετρικών στατιστικής φύσης σε βάθος χρόνου.

4. Προβολή των δεδομένων σε ευπαρουσίαστα και εύπεπτα διαγράμματα που θα βοηθούν το χρήστη να αντιλαμβάνεται γρήγορα την κατάσταση των συστημάτων του

5. Διάθεση τρόπων ειδοποίησης του χρήστη σε περίπτωση που κάποιο από τα συστήματα δεν ανταποκρίνεται

6. Δυνατότητα για οριζόντια κλιμάκωση (**horizontal scaling**)

7. Ελαχιστοποίηση downtime του συστήματος

8. Πλήρης παραμετροποίηση του μηνύματος που αποστέλλεται (κατάλληλη ρύθμιση headers, body και query)

9. Ρύθμιση του χρόνου μεταξύ διαδοχικών μηνυμάτων

Πολλές απο τις εφαρμογές που είδαμε καλύπτουν σε κάποιο βαθμό μερικά από τα παραπάνω, αλλά καμία δεν τα καλύπτει όλα ταυτόχρονα. Τα εργαλεία που αναπτύχθηκαν σε enterprise επίπεδο έχουν δυνατότητες κλιμάκωσης αλλά στα περισσότερα (αν όχι σε όλα) μπορείς να δεις πληροφορία μέχρι ένα συγκεκριμένο χρονικό διάστημα πίσω στο χρόνο, κρύβοντας δεδομένα που είτε δεν αποθηκεύουν πλέον είτε δεν μπορούν να αντλήσουν αρκετά γρήγορα. Απο την άλλη οι περισσότερες open source εφαρμογές που αναφέραμε δεν μπορούν να κάνουν τόσο εύκολα scaling καθώς είναι φτιαγμένες να δουλεύουν για περιορισμένο πλήθος χρηστών.

Στο πλαίσιο της τρέχουσας διπλωματικής καλούμαστε να δημιουργήσουμε ένα τέτοιο σύστημα Ενεργής Παρακολούθησης που θα καλύπτει τα παραπάνω. Παράλληλα θα έχει δυνατότητες επικοινωνίας με το ίδιο το μηχάνημα στο οποίο τρέχει το υπό μελέτη σύστημα προκειμένου να λαμβάνει διαγνωστικά της λειτουργίας του (κατανάλωση CPU και υπολογιστικής μνήμης RAM) και να λαμβάνει αποφάσεις, ανάλογα με την απόκριση του συστήματος, για τη διανομή πόρων αυτού.

Όσο αφορά το τελευταίο κομμάτι, στη βιβλιογραφία, έχουμε βρει ανάλογες προσπάθειες για την βέλτιστη διανομή των υπολογιστικών πόρων εφαρμογών/ιστοσελιδών που έχουν δημιουργηθεί με τη χρήση τεχνολογίας docker, υπό μορφή containers. Μερικές από αυτές είδαμε στα [21], [22] όπου χρησιμοποιούνται machine learning μέθοδοι και αλγόριθμοι προκειμένου να γίνουν προβλέψεις για τη λειτουργία του συστήματος που μελετάμε και να τροποποιηθούν όπως κάθε φορά πρέπει οι αντίστοιχοι πόροι αυτού. Οι αλλαγές γίνονται προμειμένου να εξασφαλιστεί η Quality of Service (QoS) απαίτηση σχετικα με το μέγιστο latency end-to-end, που δεν πρέπει να ξεπερνάει το 99ο εκατοστημόριο (99 percentile end-to-end latency) [23] [24]. Πιο συγκεκριμένα στον Sinan [22] χρησιμοποιείται Reinforcement Learning (RL) που βασίζεται σε Markov Decision Process. Ανά συγκεκριμένενες χρονικές στιγμές λαμβάνει δεδομένα από το ίδιο το docker container που ελέγχει και αποφασίζει για το πως θα διανείμει τη CPU κάθε φορά. Στο Sinan [21] αξιοποιεί Convolutional Neural Networks (CNN) για να κάνει προβλέψεις, με δεδομένα από το διαχειριστικό του Docker, τις οποίες στη συνέχεια βάζει σαν είσοδο σε Boosted Tress. Αυτά λαμβάνουν την απόφαση για το πως θα γίνει η αναδιανομή των πόρων στα υπο μελέτη

Docker containers. Οι πόροι που τροποποιεί είναι η CPU και η μνήμη RAM. Προκειμένου να λειτουργήσουν τα Boosted Trees θα πρέπει να γίνει ένα pre-training με πραγματικά δεδομένα στο υπό μελέτη σύστημα για όσες το δυνατό περισσότερες περιπτώσεις στις οποίες αυτό μπορεί να βρεθεί (διαφορετικοί συνδυασμοί CPU, μνήμης RAM, latency, traffic).

Αξίζει να αναφερθεί ότι στο πλαίσιο βελτίωσης της λειτουργίας docker containers, πέραν της κατακόρυφης κλιμάκωσης (vertical scaling) υπάρχουν μέθοδοι οριζόντιας κλιμάκωσης (horizontal scaling) όπως αναφέρεται και στο [25], όπου προσπαθούν να βρουν κατάλληλο μηχάνημα κάθε φορά, για να δημιουργηθούν τα καινούργια services που θέλουν να προσθέσουν ή να αναβαθμίσουν τα ήδη υπάρχοντα.

Στη δική μας υλοποίηση το dynamic resource allocation βασίζεται όπως θα δούμε και στη συνέχεια στην αναγνώριση ανωμαλιών στη χρονοσειρά αποκρίσεων του υπό μελέτη συστήματος. Για την αναγνώριση ανωμαλιών σε πραγματικά δεδομένα, σε πραγματικό χρόνο, υπάρχουν αλγόριθμοι όπως αυτοί που αναφέρονται στα [26], [27] και [28]. Τα τελευταία δύο βασίζονται στο RePAD [26] και αποτελούν αναβαθμισμένες εκδόσεις αυτού. Ο τρόπος λειτουργίας παραμένει κατά βάση ίδιος. Γίνεται εκπαίδευση ενός Long Short-Term Memory (LSTM) μοντέλου με τις προηγούμενες n μετρήσεις και κάθε φορά προβλέπεται η τιμή με βάση το προηγούμενο μοντέλο που υπολογίστηκε. Στη συνέχεια υπολογίζεται ένας δείκτης σχετικού λάθους και ένα όριο (threshold). Όταν το σχετικό λάθος ειναι μεγαλύτερο του threshold έχουμε εντοπισμό ανώμαλου σημείου. Στην απλή εκδοχή του RePAD για τον υπολογισμό του threshold χρησιμοποιούνται όλα τα προηγούμενα υπολογισμένα σχετικά λάθοι, κάτι το οποίο σε βάθος χρόνου θα καθιστά τον εντοπισμό αργό και σε ορισμένες περιπτώσεις μη λειτουργικό. Το πρόβλημα αυτό λύνεται στο RePAD2 [27], που το πλήθος των προηγούμενων σχετικών λαθών που συνυπολογίζονται είναι πεπερασμένο. Για να καλύψει τα κενά που αυτό δημιουργεί γίνονται περαιτέρω αλλαγές όπως ο επαναϋπολογισμός των μετρικών (σχετικό λάθος, threshold, προβλεπόμενη τιμή).

Πίνακας 3.1: Σύγκριση Εργαλείων Ενεργής Παρακολούθησης

	Τρέχουσα Διπλωματική	**Better Uptime**	**Uptime Robot**	**Site24x7**	**Uptimia**	**Kuma**
Σταθερότητα	✓	✓	✓	✓	✓	✓
Βάση Δεδομένων	✓	✓	✓	✓	✓	
Ιστορικά Δεδομένα	✓	✓				
Διαγράμματα	✓	✓		✓	✓	
Ειδοποιήσεις	✓	✓	✓	✓	✓	✓
Ελαχιστοποίηση Downtime	✓	✓	✓	✓	✓	✓
Παραμετροποίηση μηνυμάτων	✓	✓	✓	✓	✓	
Ρύθμιση Χρόνου μεταξύ μηνυμάτων	✓	✓	✓	✓	✓	✓
Δυναμική Ρύθμιση Πόρων	✓					
Αναγνώριση Ανωμαλιών	✓					

4
Υλοποιήσεις

Στο κεφάλαιο αυτό θα αναφερθούμε στο σύστημα που υλοποιήσαμε προκειμένου να πετύχουμε έναν συνδυασμό Ενεργής-Παθητικής Παρακολούθησης με δυνατότητες αυτόματης διευθέτησης πόρων, με σκοπό τη βελτιστοποίηση των υπό μελέτη συστημάτων. Οι βασικές λειτουργίες του συστήματός μας παρατίθενται παρακάτω:

Functional Requirements:

- Το σύστημα θα πρέπει να μπορεί να δέχεται URLs καθώς και άλλες πληροφορίες που κρίνονται απαραίτητες για να μπορεί να τα παρακολουθεί, ανά χρονικά διαστήματα που ορίζει ο χρήστης.
- Ο χρήστης θα πρέπει να μπορεί να εισάγει κάποια βασικά στοιχεία για τον έλεγχο που θέλει να προσθέσει (url, χρόνο μεταξύ διαδοχικών ελέγχων) καθώς και περαιτέρω στοιχεία που αφορούν το ίδιο το αίτημα που θα πραγματοποιήσει (προσθήκη authorization headers, εκτιμώμενη απάντηση από το σύστημα για validation)
- Το σύστημα θα πρέπει να διαθέτει λειτουργίες scheduler. Να εκτελεί δηλαδή συγκεκριμένες ρουτίνες ανά συγκεκριμένες χρονικές περιόδους.
- Το σύστημα θα πρέπει να μπορεί να συλλέγει μεγάλο όγκο δεδομένων και να κρατάει στατιστικά στοιχεία πάνω σε αυτά.
- Το σύστημα θα πρέπει να διαθέτει έναν ''έξυπνο'' μηχανισμό αναγνώρισης ανωμαλιών στους χρόνους αποκρίσεων που συλλέγει κατά τη διάρκεια λειτουργίας του.
- Το σύστημα θα πρέπει να μπορεί να συλλέγει δεδομένα κατανάλωσης πόρων από το ίδιο το μηχάνημα που μελετά απομακρυσμένα, εφόσον αυτό έχει μορφή docker container.

- Το σύστημα θα πρέπει να μπορεί να τροποποιεί τους διαθέσιμους πόρους του μηχανήματος που ελέγχεται, εφόσον αυτό έχει μορφή docker container.

Non-Functional Requirements:

- Το scheduling σύστημα θα πρέπει να εκτελεί της ρουτίνες του εντός ορισμένου χρονικού πλαισίου και να μην αποκλίνει από αυτό

- Ο αλγόριθμος αναγνώρισης ανωμαλιών θα πρέπει να μην είναι υπολογιστικά ”βαρύς”, ώστε να εξυπηρετούνται παράλληλα περισσοτεροι από ένας χρήστες

Γίνεται εμφανές ότι το παραπάνω σύστημα σπάει σε περισσότερα από ένα υποσυστήματα:

- Έναν node express server που στεγάζει το API του συστήματός μας, μοιράζει τις απαραίτητες λειτουργίες στα υπόλοιπα υποσυστήματα και οργανώνει τη ροή της πληροφορίας.

- Έναν scheduler, βασική λειτουργία του οποίου είναι να κάνει ping ανά ορισμένα χρονικά διαστήματα URLs (τα οποία είναι αποθηκευμένα σε μία βάση δεδομένων). Πέραν αυτού διαθέτει και άλλες ρουτίνες οι λειτουργίες των οποίων θα αναλυθούν στη συνέχεια.

- Αν το υπό μελέτη σύστημα είναι φτιαγμένο με τεχνολογία Docker τότε προστίθεται ένα ακόμα υποσύστημα σε μορφή docker container, που τρέχει στο ίδιο σύστημα με αυτό που μελετάμε και στέλνει logs για την κατανάλωση πόρων των containers που ελέγχουμε. Το υποσύστημα αυτό τροποποιεί δυναμικά τους διαθέσιμους προς τα containers πόρους ανάλογα με τις εντολές που δέχεται απο το σύστημα που θα αναφερθεί στη συνέχεια. Επειδή το υποσύστημα αυτό λειτουργεί σαν μία διάβαση πεζών αυξάνοντας και μειώνοντας τους πόρους προκειμένου να εχουμε λιγότερο συνοστισμό στο εξής θα αναφέρεται ως **Krosswalk**

- Τέλος ένα υποσύστημα που διαβάζει τους χρόνους αποκρίσεων που μαζεύουν οι schedulers κατα τη λειτουργία τους, και κάνει αναγνώριση ανωμαλιών πάνω στα καινούργια δεδομένα που λαμβάνει κάθε φορά. Το υποσύστημα αυτό στέλνει εντολές στο Krosswalk, κάθε φορά που κρίνει ότι υπάρχει κάποια μη φυσιολογική τιμή στη χρονοσειρά που αναλύει. Το σύστημα αυτό στο εξής θα αναφέρεται ως **Oracle**.

Πλέον και χάριν συντομίας, στο υπόλοιπο κομμάτι της διπλωματικής αυτής εργασίας, θα αναφερόμαστε στο παραπάνω σύστημα, με το όνομα **Lychte** (/licht/). Η ονομασία αυτή προκύπτει από τα αρχικά του ”**L**ightweight **Y**et **C**onfigura-ble **H**TTP **T**raffic **E**xpert”, που περιγράφει την λειτουργία και μερικά μόνο από τα πολλά χαρακτηριστικά του συστήματος μας.

4.1 Server

Αποτελεί τον ”εγκέφαλο” του Lychte. Επικοινωνεί άμεσα και έμμεσα με όλα τα υπόλοιπα υποσυστήματα. Οργανώνει τη ροή της πληροφορίας και διαθέτει api μέσω του οποίου μπορούμε να έχουμε πρόσβαση σε όλη την αποθηκευμένη πληροφορία, εφόσον έχουμε τα κατάλληλα κλειδιά authentication και authorization. Πέρα των προαναφερθέντων υποσυστημάτων επικοινωνεί με μία βάση δεδομένων NoSQL και πιο συγκεκριμένα με μια MongoDB. Η βάση αυτή επιλέχθηκε λόγω της ικανότητάς της να αποθηκεύει μεγάλο όγκο δεδομένων, ο τύπος σχήματος (schema) των οποίων δεν χρειάζεται να είναι αυστηρά προκαθορισμένος. Επιπλέον οι δυνατότητες που παρέχει για sharding καθώς και για εύκολη και γρήγορη κλιμάκωση την καθιστούν ιδανική για το σύστημα μας, που συνεχώς αποθηκεύει νέα δεδομένα στη βάση.

Συνεχίζοντας στο πλαίσιο της βάσης δεδομένων έχουμε ορίσει ορισμένες οντότητες (collections) για την πιο εύκολη αποθήκευση και οργάνωση της πληροφορίας:

- **Api** (σχήμα 4.1): περιέχει αποθηκευμένες πληροφορίες για τον έλεγχο που θέλουμε να υλοποιήσουμε. Σε αυτές περιλαμβάνονται:
  - Μοναδικό ID που περιγράφει μονοσήμαντα το συγκεκριμένο API
  - Τίτλος
  - URL
  - Η μέθοδος του url που θέλουμε να ελέγξουμε (GET, POST, PUT, DELETE, PATCH)
  - Ο τύπος της αναμενόμενης απάντησης (JSON, STREAM, TEXT)
  - Ο χρόνος μεταξύ των διαδοχικών ελέγχων
  - Query παράμετροι που θα στείλουμε μαζί με το url
  - Body που θα στείλουμε υπό μορφή json στις περιπτώσεις των POST, PUT, DELETE, PATCH αιτημάτων
  - Headers
  - Status Code: Το ορίζει ο χρήστης προκειμένου να μπορεί να γίνει έλεγχος αν το αίτημα επιστρέφει αυτό που κανονικά θα έπρεπε
  - Response body υπό μορφή json: Ομοίως ορίζεται από τον χρήστη για τον λόγο που προαναφέρθηκε
- **Response** (σχήμα 4.2): περιέχει αποθηκευμένες πληροφορίες από τα αιτήματα που κάνουμε στα αποθηκευμένα στη βάση apis. Σε αυτά αποθηκεύουμε:
  - Το id από το api για το οποίο έχει πραγματοποιηθεί το αίτημα
  - Ένα πεδίο hasError που περιγράφει το αν το αναμενόμενο (αυτό που έχει ορίσει ο χρήστης) response διαφέρει από το συγκεκριμένο.
  - Ένα πεδίο hasFailed που δηλώνει σφάλμα κατά την προσπάθεια εκτέλεσης του αιτήματος που οφείλεται σε αποτυχία του Lychte.

– Οι χρονισμοί του αιτήματος. Ο χρόνος δηλαδή που χρειάστηκε για να πραγματοποιηθεί το αίτημα, καθώς και όλοι οι χρόνοι στις διάφορες φάσεις εκτέλεσής του (σύνδεση σε DNS server, secure σύνδεση, χρόνος αποστολής πρώτου byte)

– Ένα πεδίο που έχει όλη την απάντηση του εξωτερικού συστήματος

– Το status code τη στιγμή που πραγματοποιήθηκε το αίτημα

Σχήμα 4.1: Παράδειγμα μίας τυπικής εγγραφής API.

4.2 Scheduler

Κύρια λειτουργία του συστήματος που υλοποιήσαμε είναι η αυτόματη παρακολούθηση του χρόνου λειτουργίας και απόκρισης ενός ιστότοπου/εφαρμογής. Για να γίνει αυτό θέλουμε ένα υποσύστημα που θα εκτελεί συνέχεια συγκεκριμένα κομμάτια κώδικα σε όσο το δυνατό πιο ακριβής χρονικές στιγμές.

Έχοντας ως ένα κοινό σημείο αναφοράς, τη Mongo βάση, μπορούμε να φτιάξουμε ”έξυπνους” schedulers. Κάθε scheduler θα είναι υπεύθυνος για περισσότερα από ένα apis, στα οποία θα κάνει requests, προκειμένου να δει αν είναι εν λειτουργεία και να ελέγξει την ορθότητα της επιστρεφόμενης απάντησης σε περίπτωση που ο χρήστης έχει ορίσει αναμενόμενη απάντηση. Πλέον και για την πιο οργανωμένη λειτουργία των schedulers προστίθεται ένα ακόμα collection στη βάση. Αυτό ονομάζεται Job (σχήμα 4.3 και περιέχει τα εξής πεδία:

- Name: η ονομασία του job που εκτελείται. Όπως θα δούμε και στη συνέχεια κάθε scheduler θα έχει παραπάνω από έναν τύπο jobs. Η βασική κατηγορία

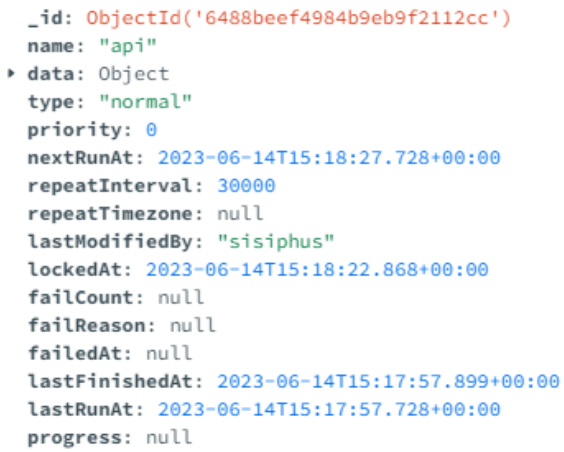
Σχήμα 4.2: Παράδειγμα εγγραφών Responses σε αίτημα που πραγματοποιήθηκε επιτυχώς και σε αίτημα που απέτυχε λόγω σφάλματος στο σύστημα που μελετάμε.

που θα εμφανίζεται κατά μεγαλύτερο βαθμό είναι η ”api” και περιγράφει την εκτέλεση ενός αιτήματος σε ένα url και τον έλεγχο ορθότητάς του.

- Data: πληροφορία που είναι αποθηκευμένη στο collection api
- repeatInterval: ρυθμός επανάληψης του συγκεκριμένου job (σε ms)
- nextRutAt: ο χρόνος σε ms που θα πρέπει να ξανατρέξει το job
- lockedAt: πότε ξεκίνησε να εκτελείται (κλείδωσε) το συγκεκριμένο job
- Επιπλέον πεδία για την εύκολη αναγνώριση σφαλμάτων (debugging) όπως το (lastFinishedAt, lastRunAt)

Στο πλαίσιο του Lychte μπορούν να υπάρχουν παραπάνω από ένας scheduler που εκτελούν jobs τα οποία ορίζει κάθε φορά ο server. Προκειμένου να έχουμε παραπάνω από έναν scheduler δίνουμε σε κάθε έναν, ένα μοναδικό όνομα. Κάθε scheduler στη συνέχεια πηγαίνει και διαβάζει jobs που είναι αποθηκευμένα στο collection ”όνομα-jobs”, με αποτέλεσμα να έχεις καταμερισμό των jobs σε περισσότερους από έναν schedulers. Κάθε api πλέον αντιστοιχίζεται μονοσήμαντα σε έναν μόνο scheduler το όνομα του οποίου αποθηκεύεται και στο api που κρατάμε στη βάση κατά τη δημιουργία του. Ο scheduler αποφασίζεται από τον server ανάλογα με το πόσα apis εξυπηρετεί κάθε scheduler εκείνη την χρονική στιγμή.

Η λειτουργία των scheduler περιγράφεται στη συνέχεια. Αρχικά κάθε scheduler ψάχνει κάθε δέκα δευτερόλεπτα στη βάση ποια αιτήματα πρέπει να εκτελεστούν. Πιο συγκεκριμένα ελέγχει ποια jobs στο συγκεκριμένο collection εκτελούνται και πόσα από αυτά θα πρέπει θα πρέπει να εκτελεστούν. Αυτό μπορούμε να το γνωρίζουμε από πριν καθώς κάθε φορά που ξεκινάει ένα job αποθηκεύουμε το χρόνο



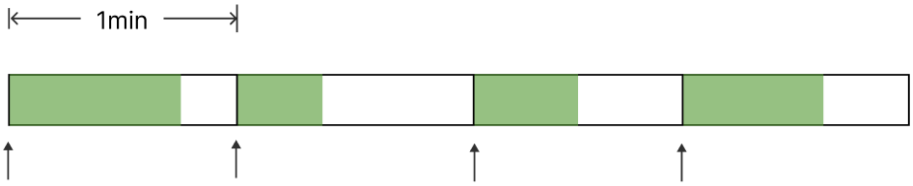
Σχήμα 4.3: Παράδειγμα εγγραφής ενός **Job** στη βάση. Το πεδίο *lockedAt* έχει τιμή διάφορη του null, όταν το job ήδη εκτελείται. Το πεδίο *nextRunAt* υπολογίζεται από το άθροισμα του χρόνου πλήρωσης της προηγούμενης εκτέλεσής του (*lastFinishedAt*) και του interval. Στο πεδίο *data* έχει αποθηκευμένη όλη την πληροφορία που χρειάζεται για να εκτελέσει το αίτημα για το οποίο δημιουργήθηκε το συγκεκριμένο job. Βάσει των πεδίων αυτών κυρίως κρίνεται το αν ο κεντρικός scheduler θα ξεκινήσει τη διεργασία

που ξεκίνησε και υπολογίζουμε το πότε θα πρέπει να ξαναεκτελεστεί (start + repeatInterval). Όλα αυτά γίνονται μόνο εφόσον το job δεν είναι locked (έχει ξεκινήσει αλλά δεν έχει τελειώσει ακόμα). Έτσι αναγνωρίζουμε τις εξής περιπτώσεις χρονισμών:

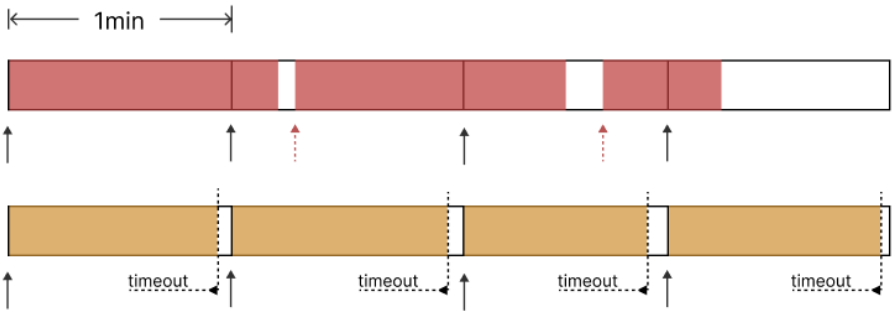
- Επιτυχής Εκτέλεση εντός χρονικού διαστήματος (σχήμα 4.4). Κάθε φορά που έρχεται η στιγμή να ξαναεκτελεστεί το αίτημα έχει ήδη προλάβει να τελειώσει το προηγούμενο
- Καθυστερημένη Εκτέλεση αιτήματος (σχήμα 4.5): Αν κάποιο αίτημα αργήσει να εκτελεστεί, σε χρόνο μεγαλύτερο από το χρόνο που θα πρέπει να ξαναγίνει το αίτημα τότε βάζουμε timeout προκειμένου να λήξουμε πρόωρα το αίτημα και να μην υπάρξουν περαιτέρω καθυστερήσεις στα επόμενα αιτήματα που πρέπει να γίνουν εντός ορισμένου χρονικού διαστήματος

Όλα αυτά αφορούν αποκλειστικά τη λειτουργία του backend του συστήματος. Πέρα από αυτά θα πρέπει να υπάρχει μία γραφική διεπαφή, μέσω της οποίας, κάθε χρήστης θα μπορεί να δει τα δεδομένα που παράγονται από τη χρήση του συστήματός.

Το σύστημα όμως, όπως είναι τώρα, δεν είναι σε θέση να μπορεί να δείχνει επαρκώς γρήγορα ιστορικά δεδομένα. Αν υποθέσουμε ότι έχουμε ένα Api που θέλουμε να ελέγχουμε κάθε λεπτό, στο τέλος της ημέρας θα έχουν μαζευτεί από αυτό και μόνο 1.440 εγγραφές. Αν αυτά συνεχίζουν να μαζεύονται, τότε ακόμα και τα πιο απλά queries στο μοντέλο των αποκρίσεων θα καθυστερούν, καθιστώντας έτσι την εφαρμογή αργή. Για την αποφυγή συσσώρευσης περιττής, μετά από ορισμένο



Σχήμα 4.4: Κύκλος Ζωής εκτέλεσης ενός επαναλαμβανόμενου Αιτήματος με ρυθμό επανάληψης ενός λεπτού



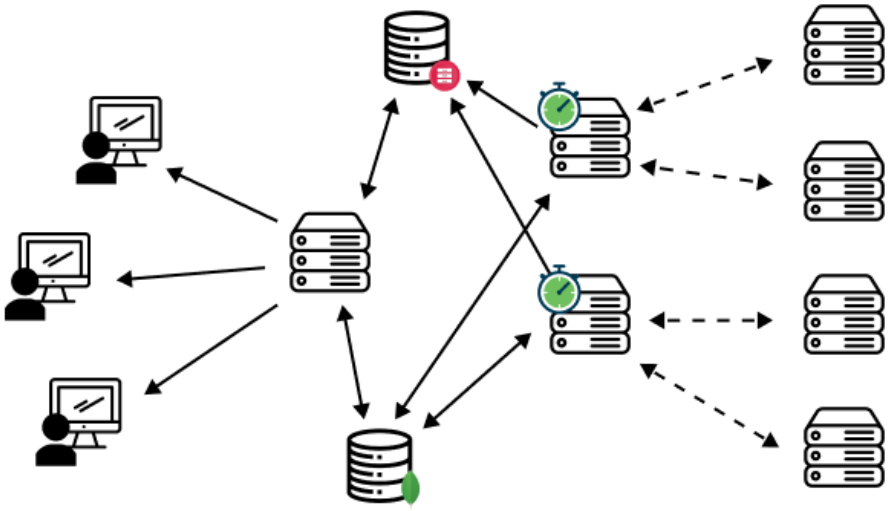
Σχήμα 4.5: Σύγκριση μη χρήσης ή χρήσης timeouts στον κύκλο ζωής εκτέλεσης ενός αργού επαναλαμβανόμενου αιτήματος. Στην πρώτη περίπτωση το επόμενο αίτημα ξεκινάει μέσα σε χρόνο δέκα δευτερολέπτων από την στιγμή που εκτελέστηκε, με αποτέλεσμα οι χρονισμοί να μην είναι πάντα σταθεροί

χρονικό διάστημα, πληροφορίας, έχουμε βάλει επιπρόσθετα σε κάθε scheduler ένα ακόμα job που αφορά τον καθαρισμό της βάσης από εγγραφές που προέρχονται από το καθένα. Η διεργασία αυτή τρέχει μία φορά την εβδομάδα. Κάθε φορά που εκτελείται ψάχνει τα Apis ανάλογα με τον scheduler στον οποίο τρέχει. Έπειτα μαζεύει όλες τις αποκρίσεις που σχετίζονται με το κάθε Api ξεχωριστά και ξεκινάει τους υπολογισμούς. Τα δεδομένα που αντλούνται στο πλαίσιο αυτής της διεργασίας, στο τέλος, διαγράφονται από τη βάση, ώστε ολα τα αιτήματα στη βάση να εκτελούνται πιο γρήγορα και αποδοτικά. Σβήνοντας όμως πληροφορία, χάνουμε την ιστορικότητα των δεδομένων μας, με αποτέλεσμα να χάνουν και οι χρήστες μία πιο γενική εικόνα των ελέγχων που έχουν πραγματοποιηθεί. Στο σημείο αυτό εισάγεται μία άλλης μορφής βάση δεδομένων, η Google Cloud Storage.

Πριν σβήσουμε δεδομένα που κρίνουμε ότι δεν είναι απαραίτητο να υπάρχουν στη βάση, αλλά έχουν αξία για ιστορική ανάλυση, τα μεταφέρουμε στο Cloud Storage της Google, υπό μορφή αρχείων. Με τον τρόπο αυτό κερδίζουμε χώρο, στη κεντρική Mongo βάση, στη λειτουργία της οποίας στηρίζεται όλο το σύστημα, και γενικά απόδοση καθώς έχει καλύτερες ταχύτητες στο σύνολο (σχήμα 4.6).

Στη συνέχεια θα αναφερθούμε πιο συγκεκριμένα στς βήματα που ακολουθούμε προκειμένου να καθαρίσουμε τη βάση:

1. Συλλογή όλων των Responses που υπάρχουν αποθηκευμένα στη βάση, μέχρι μία μέρα πριν την εκτέλεση της διεργασίας καθαρισμού. Αυτό γίνεται προ-



Σχήμα 4.6: Διάγραμμα λειτουργίας Server-Schedulers.

κειμένου να σιγουρευτούμε ότι υπάρχει πάντα πληροφορία στη βάση για την προηγούμενη ημέρα που μπορούμε να δείξουμε.

2. Δημιουργία δύο διαφορετικών πινάκων, ενός που αποθηκεύει τους χρόνους διάρκειας των αιτημάτων και ενός που περιέχει Boolean μεταβλητές σχετικά με το αν τα αιτήματα γίνονται επιτυχώς ή όχι.

3. Στη συνέχεια, για κάθε μέρα στην οποία έχουμε εγγραφές Responses, υπολογίζουμε και αποθηκεύουμε στατιστικά σχετικά με τη μέση τιμή, διάμεσο και τυπική απόκλιση διάρκειας των αιτημάτων.

4. Στο τέλος της διαδικασίας αποθηκεύονται τρία αρχεία για κάθε Api. Ένα στο οποίο περιέχονται τα responses όπως ακριβώς ήταν αποθηκευμένα στη mongodb (σε αρχείο της μορφής "**../apiId/raw-responses/MM-DD-YYYY.json**"), ένα που περιέχει τους πίνακες που περιγράψαμε στο δεύτερο βήμα ("**../apiId/-raw-values/MM-DD-YYYY.avro**") και τέλος, ένα αρχείο που περιέχει κάποια χρήσιμα στατιστικά για κάθε μέρα που εντοπίστηκε, όπως περιγράφεται στο βήμα τρία ("**../apiId/statistics.avro**")

Επειδή τα αρχεία που περιέχουν στατιστικά για κάθε μέρα χρησιμοποιούνται αρκετά συχνά από τη γραφική διεπαφή του Lychte για την παρουσίαση ιστορικών δεδομένων του υπό μελέτη Api, κρίναμε απαραίτητη τη χρήση ενός διαφορετικού τύπου αρχείου σε σχέση με τον κλασσικό και ευρέως διαδεδομένο στο διαδίκτυο τύπο JSON, το AVRO, για να κερδίσουμε ως προς τον αποθηκευτικό χώρο αλλά και την ταχύτητα ανάγνωσης. Αξίζει να σημειωθεί ότι ένα αρχείο avro σε σχέση με ένα αρχείο json που περιέχουν ακριβώς την ίδια πληροφορία, για τα στατιστικά στα οποία αναφερόμαστε, είναι 70% περίπου μικρότερο σε μέγεθος (json αρχείο 250Mb, avro αρχείο 75Mb) Ο λόγος που μπορούμε να χρησιμοποιήσουμε avro αρχεία είναι

ότι η μορφή της αποθηκευμένης πληροφορίας παραμένει ίδια και δεν αλλάζει. Πιο συγκεκριμένα, τα schemas που χρησιμοποιούνται στα αρχεία "**raw-values/MM-DD-YYYY.avro**" και "**statistics.avro**" φαίνονται στο σχήμα 4.7.

```
const statisticsType = Type.forSchema({
    type: "array",
    name: "statistics",
    items: [
        {
            type: "record",
            fields: [
                {
                    name: "date",
                    type: "string",
                },
                {
                    name: "durationMean",
                    type: "double",
                },
                {
                    name: "durationStd",
                    type: "double",
                },
                {
                    name: "durationMedian",
                    type: "double",
                },
                {
                    name: "durationMin",
                    type: "double",
                },
                {
                    name: "durationMax",
                    type: "double",
                },
                {
                    name: "durationQuartiles",
                    type: [
                        "null",
                        {
                            name: "nestedDurationQuartiles",
                            type: "record",
                            fields: [
                                {
                                    name: "q1",
                                    type: "double",
                                },
                                {
                                    name: "q3",
                                    type: "double",
                                },
                            ],
                        },
                    ],
                },
                {
                    name: "successRate",
                    type: "double",
                },
            ],
        },
    ],
});
```

```
const rawValuesType = Type.forSchema({
    type: "record",
    name: "rawValues",
    fields: [
        {
            name: "durations",
            type: [
                "null",
                {
                    type: "array",
                    items: [
                        { type: "long" },
                    ],
                },
            ],
        },
        {
            name: "successes",
            type: [
                "null",
                {
                    type: "array",
                    items: [
                        { type: "boolean" },
                    ],
                },
            ],
        },
    ],
});
```

Σχήμα 4.7: Schemas πληροφορίας που αποθηκεύουμε σε avro αρχεία. Το πρώτο αφορά τα statistics που αποθηκεύονται και αποτελείται από ένα array από objects, τα fields των οποίων σχετίζονται σε μετρικές στατιστικής φύσης. Το δεύτερο schema χαρακτηρίζει τα raw-values που αποθηκεύουμε και αποτελείται από ένα object με δύο arrays

4.3 Krosswalk

Το τρίτο αυτό υποσύστημα μπορεί να υπάρχει εφόσον ο ιστότοπος/εφαρμογή που θέλουμε να ελέγξουμε έχει δημιουργηθεί με εργαλεία docker. Αποτελεί δηλαδή ένα docker container. Το ίδιο έχει πάλι τη μορφή docker container και είναι υπεύθυνο για την συλλογή logs των πόρων του container που θέλουμε να μελετήσουμε και την αποστολή τους σε μία ουρά kafka προκειμένου να έχουμε persistent δεδομένα από τη λειτουργία του. Το krosswalk αποτελεί container που θα πρέπει ο ίδιος ο χρήστης να εγκαταστήσει και να εκτελέσει στο δικό του docker, ώστε να βρίσκεται ''δίπλα'' σε αυτό που θέλει να ελέγχετε.

Το krosswalk αποτελεί ένα container, που περιέχει ένα εκτελέσιμο αρχείο, το οποίο προήλθε από το ''χτίσιμο'' κώδικα γραμμένου σε Rust. Η γλώσσα πραγραμματισμού Rust επιλέχθηκε κατά κύριο λόγο για την ασφάλεια μνήμης, τις δυνατότητες παράλληλου προγραμματισμού, την αποδοτικότητα που προσφέρει καθώς και τη δυνατότητα δημιουργίας εκτελέσιμων αρχείων (αρχείων που εκτελεί ο υπολογιστής και δεν μπορούν να διαβαστούν από τον άνθρωπο). Αξίζει να σημειωθεί ότι για να κάνουμε έλεγχο ενός συγκεκριμένου container θα πρέπει να γνωρίζουμε το container id του docker container. Αυτό το γνωρίζει μόνο ο χρήστης και το προσθέτει σαν μεταβλητή περιβάλλοντος στο dockerfile του container που περιέχει το krosswalk.

Η βασική λειτουργία του είναι να εκτελεί την εντολή ***''docker stats cntrId''*** κάθε n (=10) δευτερόλεπτα και να υπολογίζει το ποσοστό χρήσης CPU και RAM του container σε σχέση πάντα με τους διαθέσιμους πόρους που του προσφέρουμε. Αν π.χ. παρέχουμε 0.5cpu και η εκτέλεση της παραπάνω εντολής μας λέει ότι κάνουμε χρήση του 50% της cpu, τότε καταλαβαίνουμε αμέσως ότι κάνουμε 100% χρήση της διαθέσιμης cpu, καθώς το docker δίνει το ποσοστό κατανάλωσης όχι ανάλογα με τους διαθέσιμους πόρους αλλά με ποσοστό χρησης της φυσικής cpu του μηχανήματος που τρέχει το container (κάθε 100% αντιστοιχεί σε 1 physical core). Κάτι παρόμοιο εφαρμόζεται για τον υπολογισμό του ποσοστού χρήσης της μνήμης (RAM). Αυτά στη συνέχεια αποστέλλονται σαν events σε μία ουρά kafka στο topic usages. Εδώ αξίζει να σημειωθεί ότι κάθε φορά που στέλνουμε ένα μήνυμα βάζουμε σαν κλειδί το id του api που αντιστοιχεί στο συγκεκριμένο έλεγχο προκειμένου να μπορούμε να ξεχωρίζουμε ποια usages ανήκουν σε ποια apis. Πέρα από λειτουργίες Kafka Producer θα πρέπει να διαθέτει και λειτουργίες Consumer προκειμένου να τροποποιεί τα resources ανάλογα με τα μηνύματα που θα δέχεται από το επόμενο κατά σειρά υποσύστημα Oracle.

Έτσι στην αρχή λειτουργίας του Krosswalk ξεκινάμε δύο threads, ένα που εκτελεί την παραπάνω διεργασία και ένα που διαβάζει κάθε n (=10) δευτερόλεπτα μηνύματα από τον topic updates, μόνο εφόσον αυτά έχουν σαν κλειδί το id του api που ελέγχουμε. Ανάλογα με το μήνυμα που δέχεται κάθε φορά εκτελεί την εντολή ***''docker update –cpus x –memory y cntrId''*** αυξάνοντας/μειώνοντας τη cpu κατά 0.5 πυρήνες και τη μνήμη κατά 0.5gb.

4.4 Oracle

Αποτελεί το τέταρτο και τελευταίο υποσύστημα του Lychte. Είναι υπεύθυνο για αναγνώριση ανωμαλιών καθώς και την λήψη αποφάσεων σχετικά με το αν θα πρέπει να γίνει αύξηση ή μείωση των διαθεσίμων πόρων, εφόσον έχει εγκατασταθεί το Krosswalk, στο υπό μελέτη σύστημα.

Όπως προαναφέραμε, πριν γίνει οποιαδήποτε τροποποίηση στο σύστημα που μελετάμε κάθε φορά, θα πρέπει πρώτα να βρούμε σημεία στη χρονοσειρά των αποκρίσεων ενός συστήματος που δηλώνουν απόκλιση από το φυσιολογικό. Για να πετύχουμε κάτι τέτοιο χρησιμοποιήσαμε τον αλγόριθμο RePAD2 [27]. Τα βήματα του αλγορίθμου μπορούμε να τα δούμε και στο σχήμα 4.8. Ουσιαστικά εκπαιδεύουμε ένα LSTM με τις τελευταίες τρεις τιμές μάθε φορά της προαναφερθείσας χρονοσειράς και κάνουμε προβλέψεις για την τιμή της τωρινής. Στη συνέχεια υπολογίζουμε ένα δείκτη σφάλματος που συνυπολογίζει τη σχετική διαφορά των τελευταίων τρία κατα σειρά τιμών από τις προβλεπόμενες. Αν αυτό ξεπερνάει ένα threshold (η τιμή του οποίου υπολογίζεται εκ νέου σε κάθε επανάληψη) τότε θεωρούμε ότι υπάρχει εν δυνάμει ανώμαλο σημείο στη χρονοσειρά. Στην περίπτωση που αυτό υπάρξει δημιουργούμε ένα καινούργιο μοντέλο LSTM το οποίο εκπαιδεύεται στα τελευταία τρία σημεία και υπολογίζονται εκ νέου οι μεταβλητές που χρειαζόμαστε (νέα προβλεπόμενη τιμή, νέα τιμή threshold, νέα τιμή σχετικού σφάλματος).

Από την παραπάνω λειτουργία παρατηρούμε γρήγορα ότι ένα από τα πλεονεκτήματα του αλγορίθμου είναι το γεγονός ότι δε χρειάζεται μεγάλο αριθμό training data, καθώς για να ξεκινήσει τη λειτουργία του χρειάζεται τρία σημεία κάθε φορά. Στην πραγματικότητα ο εντοπισμός ανώμαλων σημείων γίνεται από τη στιγμή που έχουμε 7 και παραπάνω ενδείξεις καθώς μέχρι το έβδομο σημείο γίνονται οι πρώτοι ακόμα υπολογισμοί των σχετικών σφαλμάτων που θα χρειαστούν σε επόμενα βήματα. Ένα ακόμα πλεονέκτημα είναι ότι τα δεδομένα που του δίνουμε στο στάδιο της εκπαίδευσης δεν έχουν labels. Αναφερόμαστε δηλαδή σε αμιγώς unsupervised learning. Τέλος η «επανεκπαίδευση» δεν γίνεται σε κάθε επανάληψη αλλά μόνο όταν κρίνεται ότι υπάρχει σημείο που αποκλίνει από τη φυσιολογική τάση της χρονοσειράς. Αξίζει να αναφερθεί ότι η διαδικασία δημιουργίας καινούργιου LSTM ειναι πολύ γρήγορη (0.2 δευτερόλεπτα κατά μέσο όρο) με αποτέλεσμα οι προβλέψεις που κάνουμε να πλησιάζουν συμπεριφορά real time (ακόμα και στην περίπτωση που πρέπει να ξαναφτιάξουμε καινούργιο μοντέλο).

Ο παραπάνω αλγόριθμος, στο πλαίσιο αυτής της διπλωματικής αναφοράς, υλοποιήθηκε με χρήση της προγραμματιστικής γλώσσας python και κατά κύριο λόγω με βιβλιοθήκες αυτής όπως οι pandas και numpy για την αποθήκευση/τροποποίηση δεδομένων και η tensorflow για τη δημιουργία, διαχείρηση, αποθήκευση machine learning μοντέλων. Η αποθήκευση των δεδομένων που χρειάζονται σε μελλοντικά στάδια του αλγορίθμου όπως είναι ο δείκτης σχετικού σφάλματος και οι προβλέψεις που κάνουμε σε κάθε βήμα αποθηκεύονται στη βάση mongo, στο αντίστοιχο collection responses κάθε φορά, τροποποιώντας έτσι ελάχιστα το schema που είχαμε ορίσει στην αρχή του κεφαλαίου. Το python script έχει έναν mini scheduler που ξεκινάει τον επαναληπτικό αλγόριθμο. Αυτός καλείτε κάθε m (=10 δευτερόλεπτα) για κάθε api που θέλουμε να κάνουμε real time anomaly detection).

Αφού γίνει η αναγνώριση «σφάλματος» στο υπό μελέτη σύστημα θα πρέπει να

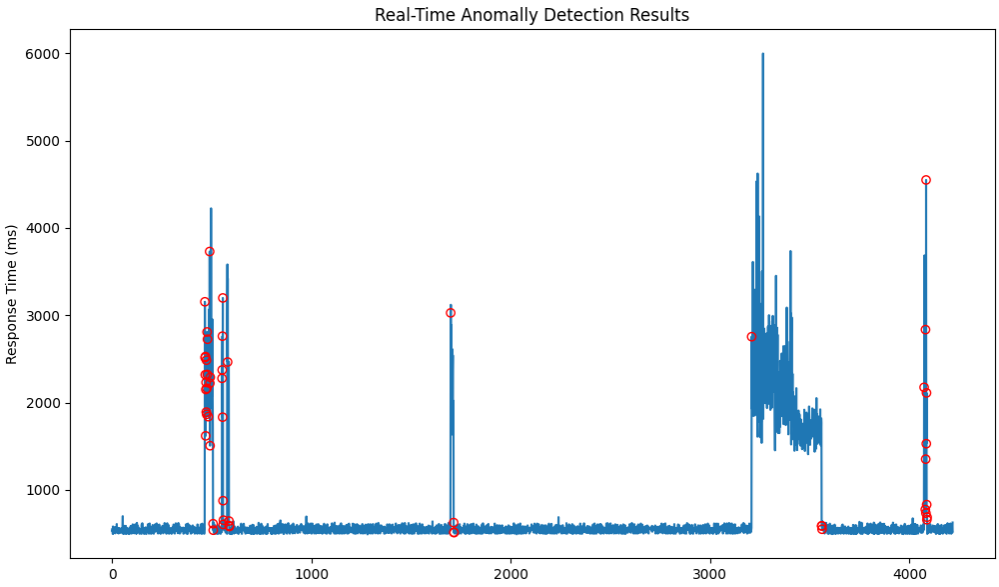
Input: Data points in the target time series
Output: Anomaly notifications
Procedure:

1: Let T be the current time point and T starts from 0; Let $flag^*$ be True;
2: **While** time has advanced {
3: Collect data point D_T;
4: **if** $T \geq 2$ and $T < 5$ {
5: Train an LSTM model by taking D_{T-2}, D_{T-1}, and D_T as the training data;
6: Let M^* be the resulting LSTM model and use M^* to predict $\widehat{D_{T+1}}$;}
7: **else if** $T \geq 5$ and $T < 7$ {
8: Calculate $AARE_T^*$ based on Equation 5;
9: Train an LSTM model by taking D_{T-2}, D_{T-1}, and D_T as the training data;
10: Let M^* be the resulting LSTM model and use M^* to predict $\widehat{D_{T+1}}$;}
11: **else if** $T \geq 7$ and $flag^*$==True {
12: **if** $T \neq 7$ { Use M^* to predict $\widehat{D_T}$;}
13: Calculate $AARE_T^*$ based on Equation 5;
14: Calculate Thd^* based on Equation 6;
15: **if** $AARE_T^* \leq Thd^*$ { D_T is not considered as an anomaly;}
16: **else**{
17: Train an LSTM model with D_{T-3}, D_{T-2}, and D_{T-1};
18: Use the newly trained LSTM model to re-predict $\widehat{D_T}$;
19: Re-calculate $AARE_T^*$ using Equation 5;
20: Re-calculate Thd^* based on Equation 6;
21: **if** $AARE_T^* \leq Thd^*$ {
22: D_T is not considered as an anomaly;
23: Replace M^* with the new LSTM model from line 17;
24: Let $flag^*$ be True;}
25: **else** {
26: D_T is reported as an anomaly immediately;
27: Let $flag^*$ be False;}}}
28: **else if** $T \geq 7$ and $flag^*$==False {
29: Train an LSTM model with D_{T-3}, D_{T-2}, and D_{T-1};
30: Use the newly trained LSTM model to predict $\widehat{D_T}$;
31: Calculate $AARE_T^*$ based on Equation 5;
32: Calculate Thd^* based on Equation 6;
33: **if** $AARE_T^* \leq Thd^*$ {
34: D_T is not considered as an anomaly;
35: Replace M^* with the new LSTM model from line 29;
36: Let $flag^*$ be True;}
37: **else** {
38: D_T is reported as an anomaly immediately; Let $flag^*$ be False;}}}

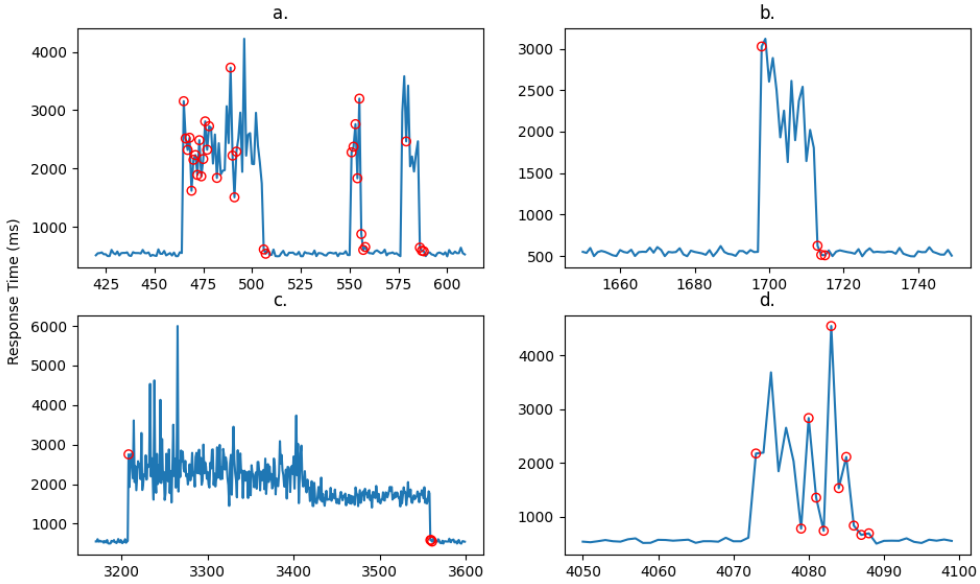
Σχήμα 4.8: Ανάλυση κώδικα REPAD2

γίνεται και διόρθωση αυτού. Σε αυτό το σημείο έρχεται το Krosswalk και η Kafka ουρά στην οποία κρατάμε δεδομένα για τη λειτουργία του συστήματος (εφόσον όπως έχουμε ήδη αναφέρει αποτελεί docker container και έχει εγκατασταθεί το Krosswalk σε ένα container εντός του υπό μελέτη συστήματος). Συνεπώς μετά την αναγνώριση λάθους το python script ξεκινάει να λειτουργεί σαν Kafka Consumer. Διαβάζει events από το topic usages μέχρι να βρει κάποιο που προέρχεται από το api που αναλύει εκείνη τη στιγμή. Αν εντός k δευτερολέπτων δε λάβει απάντηση συνεχίζει κανονικά τη λειτουργία του σε μια επόμενη επανάληψη. Αν λάβει κάποιο μήνυμα τότε ανάλογα με το ποσοστό χρήσης της μνήμης και της CPU στέλνει ανάλογα μηνύματα για την αύξηση ή μείωση του καθενός ξεχωριστά. Πιο συγκεκριμένα αν το ποσοστό χρήσης ξεπερνά το 90% τότε αυξάνει τους πόρους και αν είναι κάτω του 10%, τους μειώνει κατά μια μονάδα κάθε φορά (0.5 πυρήνες για τη cpu και 0.5gb για τη μνήμη). Για να μεταφερθεί η πληροφορία τροποποίησης των πόρων στο Krosswalk, το Oracle στέλνει ένα event στο topic updates με συγκεκριμένες κάθε φορά εντολές.

Προκειμένου να δούμε καλύτερα τη λειτουργία του αλγορίθμου RePAD2 και να μελετήσουμε την αποτελεσματικότητα του στο δικό μας σύστημα έχουμε πάρει πραγματικά δεδομένα απο χρονικές αποκρίσεις ενός συστήματος και τον εφαρμόσαμε πάνω σε αυτά. Οι αποκρίσεις φαίνονται στο σχήμα 4.9. Με κόκκινα κυκλάκια σηματοδοτούμε τα σημεία στα οποία η εφαρμογή του αλγορίθμου εντοπίζει ανώμαλα σημεία. Για να γίνουν πιο εμφανή τα σημεία που εντοπίστηκαν, στο σχήμα 4.10 μπορούμε να τα δούμε σε μεγέθυνση. Αξίζει να σημειωθεί ότι και στις έξι περιπτώσεις που έχουμε ανώμαλα σημεία, τέτοια δηλαδή που αποκλίνουν από τους φυσιολογικούς χρονισμούς που παρατηρούνται, ο εντοπισμός γίνεται τόσο στην αρχή της αύξησης του φόρτου του υπο μελέτη συστήματος όσο και στη μείωση αυτού. Στις πρώτες τρεις μάλιστα περιπτώσεις αύξησης του φορτίου βλέπουμε ότι παρατηρούνται περισσότερα από ένα συνεχόμενα σημεία που αναγνωρίζονται ως ανώμαλα. Κάτι το οποίο σε επόμενες περιπτώσεις δε συμβαίνει αφού ο εντοπισμός γίνεται στην αρχή και στο τέλος της τροποποίησης του φορτίου. Αυτό γίνεται διότι ο τρόπος που υπολογίζεται το όριο απόφασης (decision threshold) εμπεριέχει όλες τις προηγούμενες τιμές σχετικού λάθους. Με αποτέλεσμα όσο περισσότερα δεδομένα υπάρχουν τόσο πιο ανθεκτικός να είναι σε μικρότερες μεταβολές.



Σχήμα 4.9: Χρονοσειρά αποκρίσεων ενός συστήματος που προήλθε απο τη χρήση του συστήματος Lychte. Με κόκκινο φαίνονται τα ανώμαλα σημεία που εντοπίστηκαν απο τη χρήση του RePAD2.



Σχήμα 4.10: Εντοπισμένα σημεία ανωμαλιών σε μεγέθυνση.

5 Πειράματα και Επίδειξη Γραφικής Διεπαφής

Στο κεφάλαιο αυτό θα μελετήσουμε την απόδοση του συστήματος Lychte που υλοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας, μέσα από την παρουσίαση δύο ανεξάρτητων πειραμάτων. Τέλος, θα παρουσιαστούν εικόνες από τη γραφική διεπαφή, την εφαρμογή δηλαδή που αναπτύξαμε.

5.1 Έλεγχος ευρωστιας του Lychte

Προκειμένου να παρακολουθήσουμε τις ανάγκες και να δοκιμάσουμε τα όρια του Lychte καναμε μία σειρά διαφορετικών μετρήσεων. Η παράμετρος που αλλάζει σε κάθε μέτρηση είναι ο αριθμός των Apis/Jobs που τρέχει παράλληλα ένας worker. Έτσι σε κάθε διαφορετική μέτρηση κάναμε n καταχωρήσεις Apis και Jobs αντίστοιχα, όπου n ο αρθμός των αιτημάτων που θέλουμε να ελέγχουμε. Στο σημείο αυτό πρέπει να ανεφερθεί ότι οι μετρήσεις έγιναν σε μηχάνημα με τα τεχνικά χαρακτηριστικά που φαίνονται στον πίνακα 5.1

Πίνακας 5.1: Τεχνικά χαρακτηριστικά Υπολογιστικού Συστήματος

Operating System	WindowsOS
Processor	AMD Ryzen 7 5800H (3,2Ghz)
RAM	16GB

5.1.1 Περιγραφή

Προκειμένου να κάνουμε τις μετρήσεις χρειαζόμαστε κάποια urls τα οποία θα πρέπει να καλούμε ανά τακτά χρονικά διαστήματα, αυτά που κανονικά ορίζει ο χρήστης κατά τη διαδικασία δημιουργίας ενός Api. Για να υπάρχει μία ποικιλία στο πλήθος των ιστοσελιδών και διαδικτυακών apis που παρακολουθούμε και να μελετήσουμε τη λειτουργία του Lychte σε μία κατάσταση όσο το δυνατό πιο κοντά σε πραγματικά σενάρια, δημιουργήσαμε script που γράφει με τυχαίο τρόπο n Apis και Jobs, στα αντίστοιχα collections της βάσης, έχοντας μία προκαθορισμένη λίστα urls, μαζί με στοιχεία που ίσως χρειάζονται για αυθεντικοποίηση (tokens). Μαλιστα, για να ελέγξουμε τη σταθερότητα του συστήματος ως προς το χρόνο μεταξύ των διαδοχικών αιτημάτων ενός Api για διάφορους χρονισμούς, περά από την τυχαιότητα επιλογής του συνδέσμου στον οποίο θα κάνουμε κλήσεις επιλέγεται με τυχαίο τρόπο και ο χρόνος μεταξύ των αιτημάτων (το interval κάθε Api/Job). Επειδή σκοπός των μετρήσεων που κάναμε είναι το stress test του συστήματος, οι χρόνοι που επιλέχθηκαν είναι ένας μεταξύ των τριάντα δευτερολέπτων, ενός, δύο, ή τριών λεπτών.

Πιο συγκεκριμένα τα σενάρια που δοκιμάστηκαν είναι τα εξής:

- $n = 100$ apis
- $n = 500$ apis
- $n = 1.000$ apis
- $n = 5.000$ apis

Κάθε μία από τις παραπάνω διαφορετικές μετρήσεις διήρκησε δώδεκα ώρες. Σε αυτό το μήκος χρόνου, μετρήθηκαν η μέση χρήση/κατανάλωση CPU και RAM καθώς και η μέση απόκλιση του χρόνου μεταξύ των διάφορων διαδοχικών αιτημάτων από τον αναμενόμενο, βάσει του interval κάθε Api, χρόνο που θα έπρεπε να ξαναεκτελεστεί.

5.1.2 Αποτελέσματα

Στον πίνακα 5.2 μπορούμε να δούμε τα αποτελέσματα των μετρήσεών μας.

Παρατηρούμε ότι στα πρώτα τρία σενάρια η χρήση υπολογιστικών πόρων του συστήματος δεν είναι σημαντική, αλλά και η μέση απόκλιση του χρόνου επανεκτέλεσης των Apis είναι πολύ μικρή. Μάλιστα παίρνει αρνητικές τιμές. Αυτό δεν οφείλεται σε κάποιο λάθος στον υπολογισμούς μας, αλλά στον τρόπο λειτουργίας του scheduler. Για να κρίνει, αν πρέπει να εκτελέσει κάποιο αίτημα, έχει ένα μικρό bias που του επιτρέπει να επιλέγει jobs που βρίσκονται πολύ κοντά στο χρόνο που πρέπει να ξανατρέξουν. Η πόλωση αυτή του συστήματος είναι της τάξη nano-δευτερολέπτων και έχει σκοπό να κερδίζει χρόνο για εξομαλύνει πιθανές καθυστερήσεις του συστήματος.

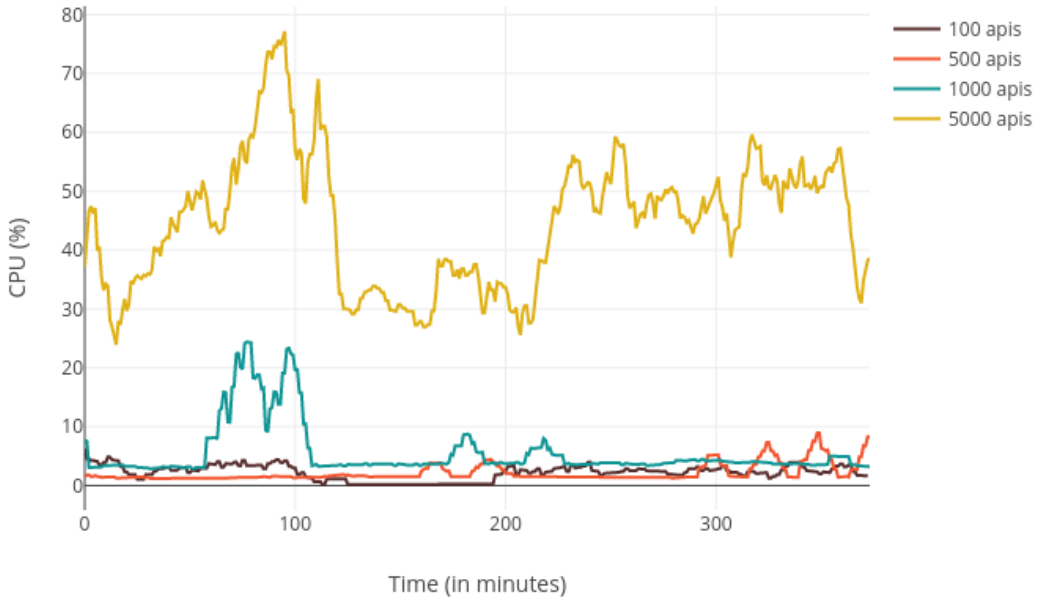
Στην τελευταία σειρά μετρήσεων, που αφορά τα 5.000 apis, που εκτελούνται ταυτόχρονα από έναν και μόνο worker, μπορούμε εύκολα να δούμε ότι το σύστημα συνεχίζει να ανταπεξέρχεται στις ανάγκες του ελέγχου και παρακολούθησης των

apis του (μέσος χρόνος απόκλισης μικρότερος του δευτερολέπτου). Αξίζει να σημειωθεί, όμως, όπως θα δούμε καλύτερα και στη συνέχεια ότι η κατανάλωση των υπολογιστικών πόρων είναι αρκετά μεγάλη, σε βαθμό μάλιστα που ιδανικά θα κάναμε load balancing με έναν από δύο τρόπους. Μεταφέροντας τα αιτημάτα που εκτελεί σε έναν ή περισσότερους άλλους workers, ή βάζοντας άλλους workers να διαβάζουν από το collection της υπό μελέτης διεργασίας.

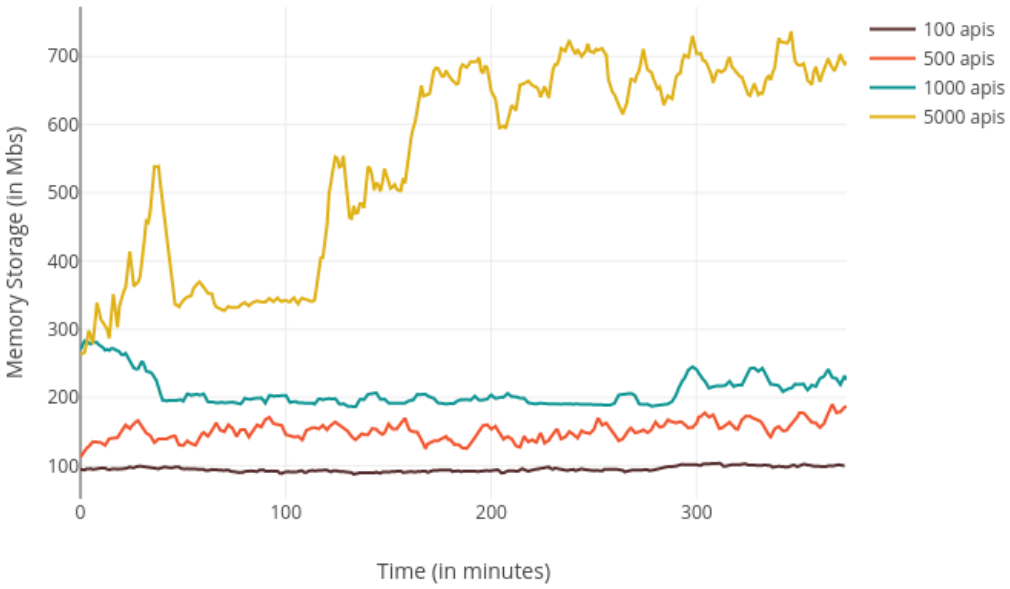
Πίνακας 5.2: Αποτελέσματα Μετρήσεων

	Μέση Απόκλιση	**Μέση Χρήση CPU**	**Μέση Χρήση RAM**
100 apis	-112,62ms	1,22%	102,54Mb
500 apis	-69,32ms	3,26%	171,1Mb
1.000 apis	-47,9ms	7,46%	223,13Mb
5.000 apis	664ms	56,36%	652,45Mb

Παρακάτω παρατίθενται διαγράμματα χρήσης cpu και ram αντίστοιχα για τις τελευταίες έξι ώρες λειτουργίας του Lychte στα τέσσερα σενάρια που αναφέραμε πιο πάνω. Λόγω της ευμετάβλητης φύσης των χαρακτηριστικών που μετράμε στο υπολογιστικό μας σύστημα, στα διαγράμματα παρουσιάζεται ο **Κινούμενος Μέσος Όρος** τους, με παράθυρο δέκα τιμών.



Σχήμα 5.1: Αποτελέσματα χρήσης της CPU.



Σχήμα 5.2: Αποτελέσματα χρήσης της RAM

5.2 Anomally Detection Test

Στο δεύτερο κατά σειρά πείραμα που θα παρουσιάσουμε, θα δούμε πιο αναλυτικά τις δυνατότητες του συστήματος μας να αναγνωρίζει σημεία ανωμαλιών σε μία συνεχόμενη σειρά αποκρίσεων ενός υπό μελέτη συστήματος.

5.2.1 Περιγραφή

Αρχικά κατασκευάσαμε έναν απλό node express server με τη χρήση Docker, και το αφήσαμε να τρέχει σε ένα container παρέχοντάς του ελάχιστους πόρους (0.5 CPUs και 512MB μνήμης). Διαθέτει ένα μόνο route, λειτουργία του οποίου είναι να υπολογίζει τον αριθμό Fibonacci του 3000 (ή όποιου άλλου αριθμού σταλεί στο αίτημα προς τον server). Έπειτα φτιάξαμε ένα api στο Lychte για την ενεργή παρακολούθηση του συγκεκριμένου route, στο οποίο δηλώσαμε ότι θέλουμε να έχει real time anomally detection.

Εδώ κάναμε δύο διαφορετικές μετρήσεις. Στην **πρώτη (α)** κλείσαμε την ουρά επικοινωνίας που είναι υπεύθυνη για την μεταφορά μηνυμάτων από το Oracle (script σε python που τρέχει τον αλγόριθμο RePAD2) στο Krosswalk που είναι υπεύθυνο για την δυναμική διαχείρηση των πόρων του υπό μελέτη συστήματος. Στη **δεύτερη (β)** κάναμε τις ίδιες μετρήσεις (όσο πιο κοντά μπορούσαμε από άποψη χρονισμών) επιτρέποντας στο σύστημα μας να λειτουργήσει πλήρως και να επηρεάζει τους πόρους όπως κάθε φορά κρίνει βέλτιστα.

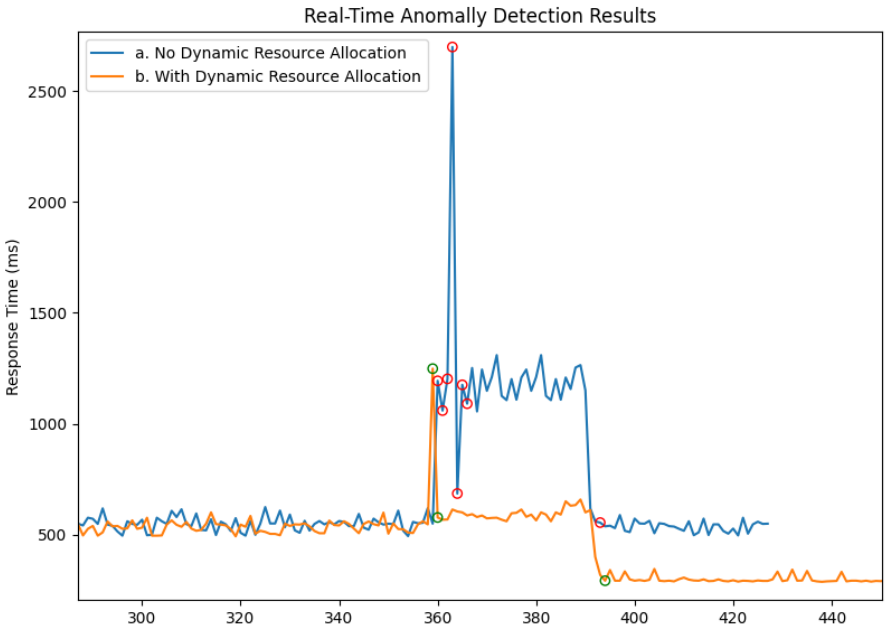
Για να δοκιμάσουμε τη λειτουργία των μηχανισμών που δημιουργήσαμε εισάγαμε, μετά από ορισμένο χρονικό διάστημα, πλασματικό φορτίο στο υπό μελέτη σύστημα, με τη χρήση του command line tool *stress-ng*. Πιο συγκεκριμένα γεμίσαμε 400mb στο σύστημα για να δούμε το πώς θα ανταπεξέλθει σε σχεδόν πλήρη λειτουργία του. Αξίζει να σημειωθεί ότι όταν το συγκεκριμένο εργαλείο τρέχει, χρησιμοποιεί ένα αρκετά μεγάλο μέρος της διαθέσιμης cpu για να επιτύχει τους στόχους που του βάζουμε. Το ίδιο ισχύει και για τη RAM καθώς όπως θα δούμε και στη συνέχεια, όταν πλέον έχει περισσότερη διαθέσιμη μνήμη, δεσμεύει ένα μεγαλύτερο μέρος αυτής.

5.2.2 Αποτελέσματα

Στα σχήματα σχήμα 5.3 και σχήμα 5.4 μπορούμε να δούμε τα αποτελέσματα μας. Και στις δύο περιπτώσεις παρατηρούμε ότι περίπου στην 370ή καταγραφή παρατηρείται απότομη άνοδος της χρήσης της μνήμης του υπολογιστικού συστήματος που μελετάμε, ενώ περίπου στην 400ή έχουμε πτώση αυτής. Μεταξύ των χρονικών περιόδων 370~400 είχαμε κάνει χρήση του εργαλείου *stress-ng* για την αύξηση της χρήσης της μνήμης συνθετικά. Αξίζει να σημειωθεί ότι και στα δύο πειράματα η απότομη αύξηση της μνήμης, επιφέρει και αντίστοιχο spike στην απόκριση του συστήματος. Και στις δύο περιπτώσεις έγινε επιτυχής αναγνώριση του anomaly που παρουσιάστηκε.

Όπως γίνεται φανερό όμως στη δεύτερη (β.) μέτρηση μετά την αρχική αύξηση της μνήμης ακολουθεί πτώση της στο 60%, πράγμα που σημαίνει ότι είχαμε δυναμική

αύξηση της, κατά 512mb προκειμένου να μπορέσει να ανταπεξέλθει στην αυξημένη λειτουργία. Παρατηρούμε αντίστοιχα ότι έχουμε σημαντική πτώση του χρόνου απόκρισης και αυτό φαίνεται και από τα σημείο που εντοπίζεται ως anomaly αμέσως μετά την αύξηση της. Αντιθέτως στη μέτρηση α. βλέπουμε ότι έχουμε σχεδόν πλήρωση των πόρων (90%) και κατά συνέπεια ένα σταθερά αυξημένο σύνολο αποκρίσεων. Παρόλα αυτά και στις δύο περιπτώσεις έχουμε ορθή αναγνώριση ενός ακόμα σημείου ως anomally και αυτό είναι στο πέρας της λειτουργίας του *stress-ng*. Στη μέτρηση β. μάλιστα ο χρόνος απόκρισης του συστήματος μετά τη χρονική στιγμή ~400 είναι μικρότερος σε σχέση με τη λειτουργία του πριν το stress-test και αυτό οφείλεται σε μεγάλο βαθμό στο γεγονός ότι από τη στιγμή ~370 και μετά έχει να διαθέσει περισσότερους πόρους για τον υπολογισμό της ακολουθίας Fibonacci, μία διαδικασία που έχει σχετικά μεγάλες απαιτήσεις από άποψη υπολογιστικής ισχύος.

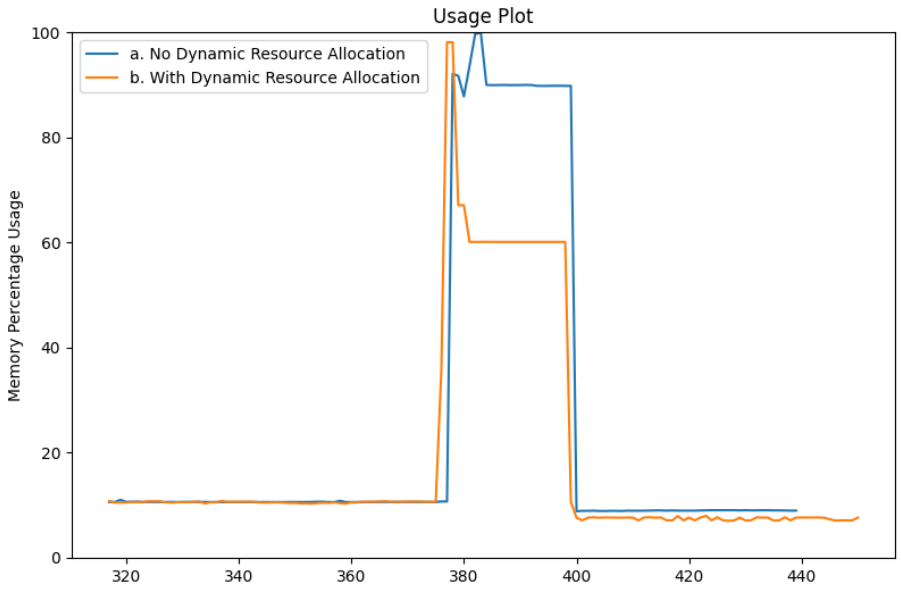


Σχήμα 5.3: Αποκρίσεις του Συστήματος α. χωρίς και β. με χωρίς τη χρήση δυναμικής διαχείρησης πόρων

5.3 Επίδειξη Γραφικης Διεπαφης

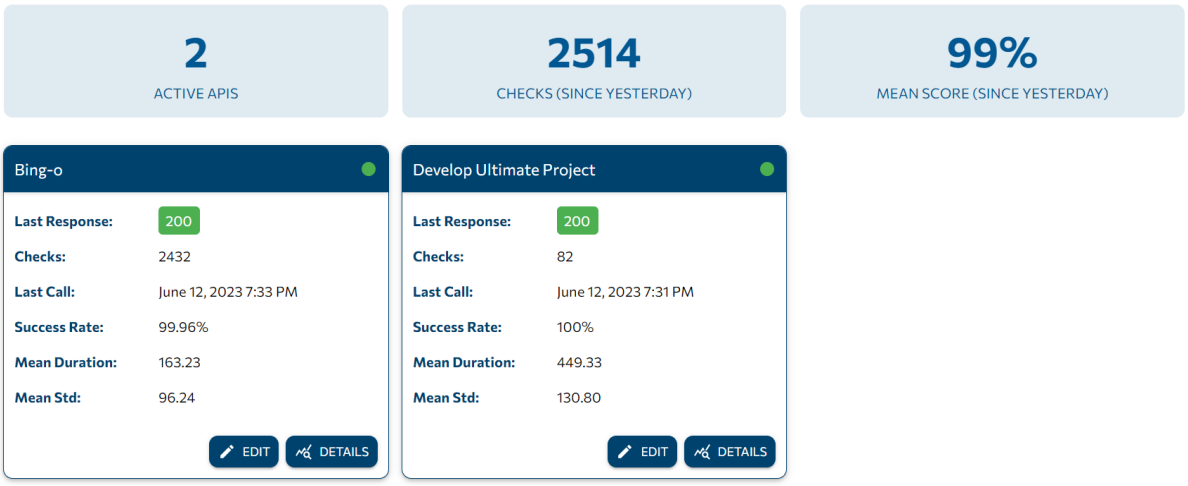
Στη συνέχεια παρατίθενται εικόνες από τη χρήση της γραφικής διεπαφής που υλοποιήσαμε, μέσω της οποίας οι χρήστες μπορούν να επικοινωνούν με το backend.

Στο σχήμα 5.5 μπορούμε να δούμε όλα τα Apis που έχουμε ενεργά, καθώς και κάποια στοιχεία σχετικά με αυτά. Η πληροφορία που δείχνουμε σε αυτήν την εικόνα, αφορά δεδομένα μέχρι μία ημέρα πριν. Αυτό γίνεται για να δώσουμε αρχικά μία γενική εικόνα των υπό μελέτη εξωτερικών συστημάτων, αλλά και να περιορίσουμε το πλήθος της πληροφορίας που θα πρέπει να ληφθεί υπόψη, ώστε να μπορεί να ανταποκρίνεται πιο γρήγορα και να είναι έτσι πιο αποδοτικά τα queries που



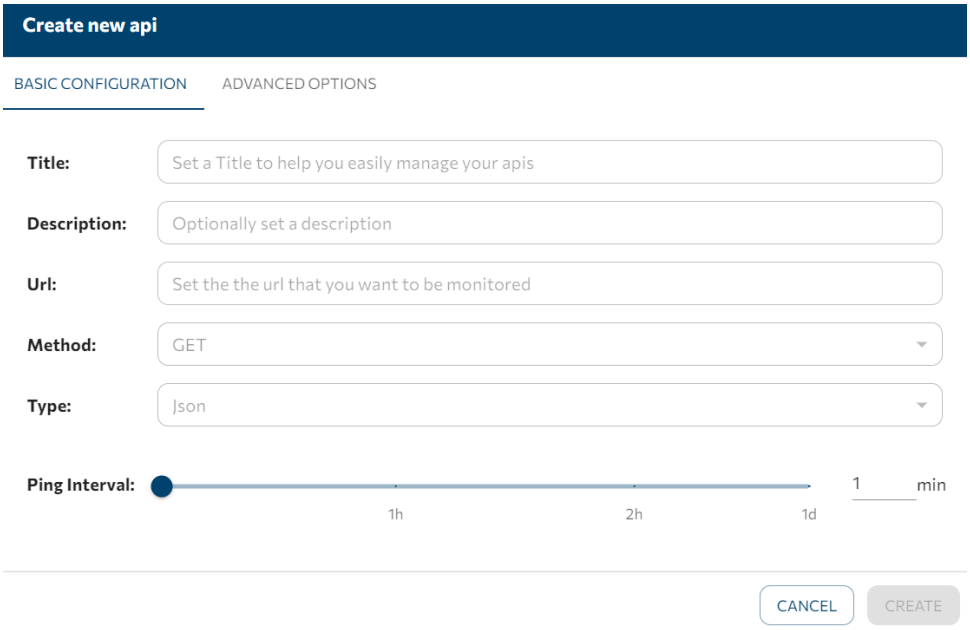
Σχήμα 5.4: Κατανάλωση RAM συστήματος α. χωρίς και β. με χωρίς τη χρήση δυναμικής διαχείρησης πόρων

κάνουμε στη βάση δεδομένων μας. Μερικά από τα στοιχεία που παρουσιάζονται αφορούν το πλήθος των ελέγχων που πραγματοποιήθηκαν εντός των τελευταίων εικοσιτεσσάρων ωρών, η μέση διάρκεια και η τυπική απόκλιση των αποκρίσεων των αιτημάτων που κάναμε, ο μέσος βαθμός επιτυχών αποκρίσεων και η κατάσταση του τελευταίου αποθηκευμένου ελέγχου.

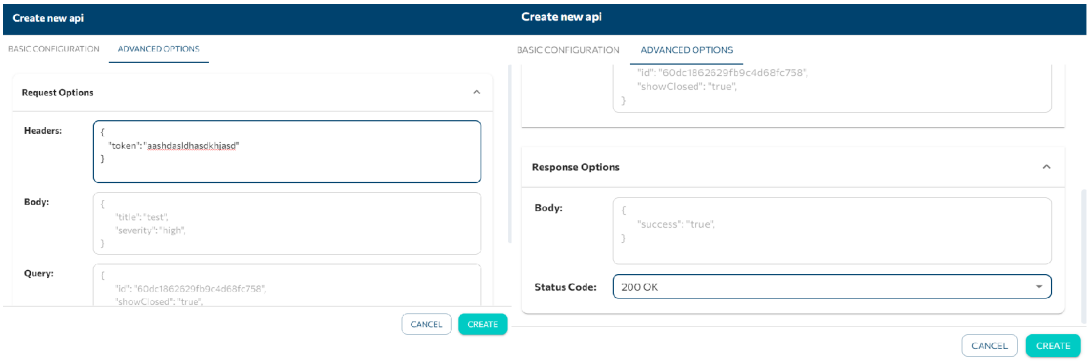


Σχήμα 5.5: Lychte Api Overview

Έπειτα στα σχήματα 5.6 και 5.7, φαίνονται τα στοιχεία που καλείται να συμπληρώσει η χρήστης για να φτιάξει κάποιο καινούργιο Api στο σύστημά μας και να ξεκινήσουν οι έλεγχοι.

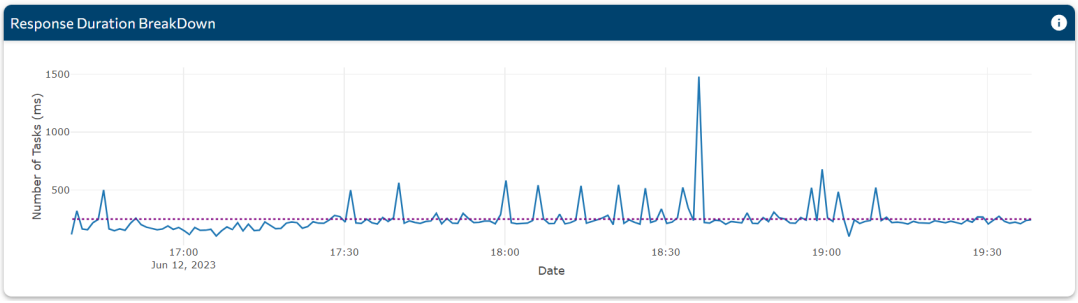


Σχήμα 5.6: Εικόνα Βασικών Στοιχείων προς συμπλήρωση από το χρήση

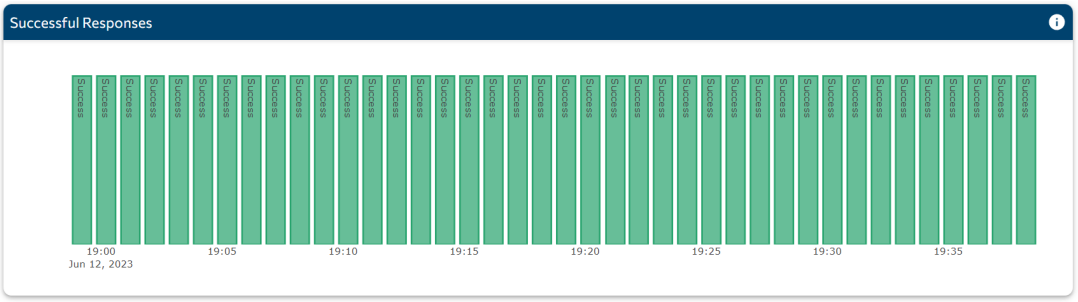


Σχήμα 5.7: Εικόνες Εισαγωγής ”Προηγμένων” στοιχείων για τη δημιουργία νέου ελέγχου

Στη συνέχεια, παρουσιάζονται διαγράμματα που μπορεί να δει ο χρήστης και σχετίζονται, όπως και πριν, με δεδομένα των τελευταίων εικοσιτεσσάρων ωρών. Τα διαγράμματα ανανεώνονται αυτόματα κάθε ένα λεπτό προκειμένου να δείχνουν πάντα την τελευταία έκδοση των δεδομένων που έχουμε αποθηκευμένα. Στο σχήμα 5.8 μπορούμε να δούμε το χρόνο που μεσολάβησε από τη στιγμή που γίνεται κάποιο αίτημα μέχρι να ανταποκριθεί το υπό μελέτη σύστημα, σε βάθος χρόνου ημέρας, μπορούμε, ωστόσο, να επιλέξουμε αν θελουμε να φιλτράρουμε τα δεδομένα στις τελευταίες τρεις, έξι και δώδεκα ώρες. Με μία διακεκομμένη γραμμή αναπαριστούμε το μέσο ώρο του χρόνου που μετρήσαμε. Τέλος, όσον αφορά, τα δεδομένα της τελευταίας μέρας υπάρχει και το ραβδόγραμμα που φαίνεται στο σχήμα 5.9, στο οποίο μπορούμε να δούμε την κατάσταση των τελευταίων πενήντα αποκρίσεων.



Σχήμα 5.8: Διάγραμμα Διάρκειας Αποκρίσεων

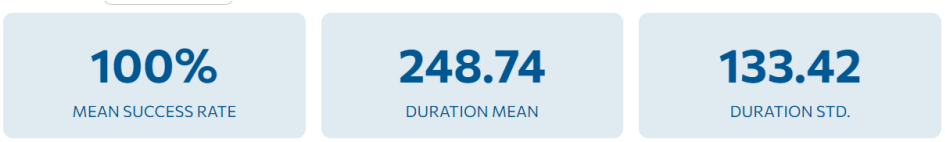


Σχήμα 5.9: Ραβδόγραμμα Κατάστασης Αποκρίσεων

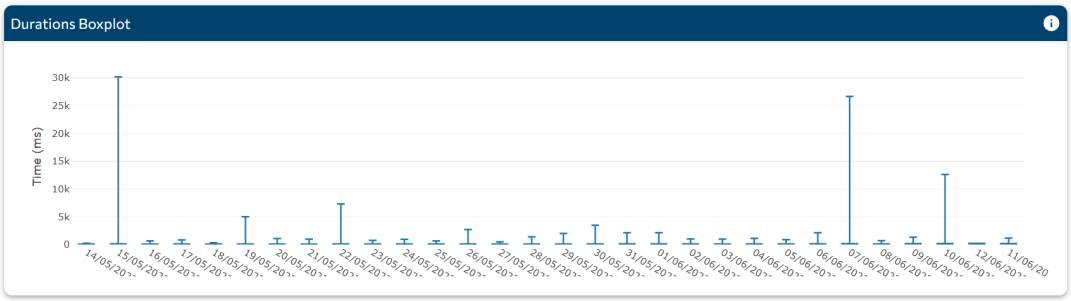
Τέλος, θα δείξουμε διαγράμματα και μετρικές που προκύπτουν από το σύνολο των δεδομένων που υπάρχουν τόσο στη Mongo βάση υπό μορφή documents, όσο και στη Google Cloud βάση, υπό τη μορφή αρχείων. Πιο συγκεκριμένα, αξιοποιούμε τα ήδη υπολογισμένα δεδομένα που υπάρχουν σε αρχεία τις μορφής "**/:apiId/statistics.avro**". Στα δεδομένα που αντλούμε από τα προαναφερθέντα αρχεία, ενσωματώνουμε όσα δεν έχουν αποθηκευτεί ακόμα εκεί και βρίσκονται στη MongoDB. Κάθε φορά που γίνονται οι υπολογισμοί αυτοί, και για να γλιτώσουμε τέτοιους συνεχείς υπολογισμούς, αποθηκεύουμε όσα στοιχεία χρειαζόμαστε σε μία νέα συλλογή δεδομένων στην κύρια βάση μας (ονομασία **Statistics**), που αποθηκεύει στατιστικά. Κάθε φορά που φορτώνει η συγκεκριμένη σελίδα, ελέγχει αν υπάρχει καταχώρηση για το συγκεκριμένο Api στο νέο αυτό collection εντός ενός χρονικού διαστήματος (τριάντα λεπτών). Αν υπάρχει, αντλεί τα δεδομένα από εκεί. Αλλιώς, κάνει τους

απαραίτητους υπολογισμούς και στο τέλος αποθηκεύει τα αποτελέσματα εκεί, με αποτέλεσμα τα επόμενα, χρονικά κοντινά, αιτήματα να ικανοποιούνται πιο γρήγορα.

Στα σχήματα 5.10 και 5.11 παρουσιάζονται μετρικές που υπολογίζονται στο σύνολο των ιστορικών δεδομένων και ένα διάγραμμα Θηκογραμμάτων της διάρκειας απόκρισης των ελέγχων που πραγματοποιήθηκαν, ομαδοποιημένα σε διαστήματα ημέρας.



Σχήμα 5.10: Μετρικές που αφορούν το σύνολο των δεδομένων (ιστορικά δεδομένα)



Σχήμα 5.11: Διάγραμμα Θηκογραμμάτων (Boxplots) της διάρκειας απόκρισης αιτημάτων για κάθε μέρα (ιστορικά δεδομένα)

6 Συμπεράσματα και Μελλοντικές Επεκτάσεις

6.1 Συμπερασματα

Συνοψίζοντας θα περιγράψουμε το τελικό σύστημα που υλοποιήσαμε καθώς και τα αποτελέσματα των μετρήσεων που κάναμε προκειμένου να ελέγξουμε την απόδοση και αξιοπιστία του συστήματός μας.

Η υλοποίηση, μπορεί να χωριστεί σε τέσσερα διακριτά υποσυστήματα. Έναν server που φιλοξενεί το api του συστήματός μας. Μέσω αυτού γίνεται η επικοινωνία με τα υπόλοιπα υποσυστήματα και η μεταφορά της αποθηκευμένης πληροφορίας εφόσον το εισερχόμενο αίτημα διαθέτει τα απαραίτητα κλειδιά.

Έναν ή περισσότερους schedulers που είναι υπεύθυνοι για τον έλεγχο αποθηκευμένων στη βάση URLs. Αυτοί είναι προγραμματισμένοι να τρέχουν με συγκεκριμένο ρυθμό αιτήματα, προς τα συστήματα που θέλουμε κάθε φορά, να ελέγχουμε. Πέραν αυτού, είναι υπεύθυνοι για τον καθαρισμό της βάσης δεδομένων (mongo) από περιττές πληροφορίες και τη μεταφορά τους σε file storage βάση δεδομένων (Google Cloud Storage).

Στη συνέχεια ακολουθεί το Krosswalk. Αυτό είναι υπεύθυνο για τη μεταφορά logs της λειτουργίας του container που ελέγχουμε και την τροποποίηση των πόρων του συστήματος ανάλογα με τις εντολές που λαμβάνει από το Oracle. Βασική συνθήκη είναι το υπό μελέτη σύστημα να είναι και αυτό docker container. Το Krosswalk επικοινωνεί με μια Kafka ουρά από την οποία διαβάζει events από το topic updates και στέλνει events στο topic usages.

Το τελευταίο υποσύστημα είναι το Oracle το οποίο είναι υπεύθυνο για την αναγνώριση ανωμαλιών στη χρονοσειρά των αποκρίσεων που μαζεύει το Lychte κατά τη λειτουργία του. Για τον εντοπισμό των σημείων χρησιμοποιούμε τον αλγόριθμο RePAD2. Μόλις βρούμε κάποιο σημείο που αποκλίνει από τη φυσιολογική λειτουργία του συστήματος, διαβάζουμε από την Kafka ουρά, και πιο συγκεκριμένα από το

topic usages το τελευταίο event που έχει έρθει για το API που ελέγχουμε. Ανάλογα με τις τιμές κατανάλωσης CPU και RAM που έχουμε, στέλνουμε event στο topic updates προκειμένου να τα τροποποιήσει αντίστοιχα το Krosswalk.

Για να ελέγξουμε την απόδοση και την αξιοπιστία του συστήματος, κάναμε κάποιες μετρήσεις. Αρχικά μελετήσαμε τη λειτουργία των schedulers που υλοποιήσαμε. Ξεκινώντας με ένα σχετικά μικρό πλήθος αιτημάτων που τρέχουν παράλληλα (100 στο πλήθος), και αυξάνοντας τα σταδιακά (φτάσαμε στα 5.000), είδαμε ότι η μέση απόκλιση του αναμενόμενου χρόνου εκτέλεσης των διάφορων αιτημάτων παραμένει χαμηλή, φτάνοντας μάλιστα σε ορισμένες περιπτώσεις σε αρνητικές τιμές. Όσον αφορά στην κατανάλωση πόρων του υπολογιστικού συστήματος, παρατηρούμε ότι αυξάνοντας το πλήθος των αιτημάτων που εκτελούνται παράλληλα από έναν scheduler, αυξάνονται, μετά από ένα ορισμένο σημείο, κατά πολύ οι απαιτήσεις του συστήματος. Αυτό μας οδηγεί στο συμπέρασμα, ότι πάντα θα πρέπει να υπάρχουν διαθέσιμοι schedulers στους οποίους θα μπορεί να γίνεται καταμερισμός των ενεργών αιτημάτων, ώστε να αποφεύγονται καταστάσεις υπερφόρτωσης του συστήματος.

Τέλος πραγματοποιήσαμε μετρήσεις για να ελέγξουμε τη λειτουργία της αυτόματης τροποποιήσεις των πόρων ενός συστήματος που αποτελεί docker container. Εδώ παρατηρήσαμε ότι ο εντοπισμός ανώμαλων σημείων γίνεται επιτυχώς. Στα σημεία που έχουμε spikes στη χρονοσειρά των αποκρίσεων βρίσκουμε ανώμαλα σημεία τα οποία εξομαλύνονται σε επόμενες χρονικές στιγμές που έχουν δωθεί περισσότεροι διαθέσιμοι πόροι στο υπό μελέτη σύστημα. Μάλιστα όταν ο φόρτος στο σύστημα πέφτει και κατά συνέπεια οι τιμές των λαμβανομένων, από το σύστημα, αποκρίσεων μειώνονται γίνεται πάλι εντοπισμός ανώμαλων σημείων. Αυτή τη φορά η κατανάλωση του συστήματος είναι τέτοια που δεν απαιτείται η περαιτέρω τροποποίηση του υπό μελέτη συστήματος.

Αξίζει να αναφερθεί ότι σε αντίθεση με άλλα συστήματα ([21], [22]) δεν απαιτείται μεγάλης διάρκειας offline training period, καθώς από την έβδομη εγγραφή και ύστερα ξεκινάει να λειτουργεί κανονικά ο αγλόριθμος εντοπισμού που χρησιμοποιήσαμε (RePAD2). Επίσης τα δεδομένα με τα οποία εκπαιδεύουμε τα μοντέλα μας δεν είναι labeled, καθιστώντας τα έτσι αμιγώς unsupervised και συνεπώς πιο απλά στην εφαρμογή τους.

6.2 Μελλοντικες Επεκτασεις

Όπως είδαμε, το σύστημα Lychte που υπολοποιήσαμε στο πλαίσιο της διπλωματικής αυτής εργασίας δουλεύει σε ένα αρκετά ικανοποιητικό επίπεδο, με τα αποτελέσματα των μετρήσεων που πραγματοποιήσαμε να το αποδεικνύουν.

Υπάρχει όμως ακόμα χώρος για βελτίωση. Όπως είδαμε, όταν το πλήθος των εξωτερικών συστημάτων που θέλουμε να ελέγχουμε αυξάνεται θα πρέπει να αυξάνεται αντίστοιχα και το πλήθος των schedulers που λειτουργούν προς την εξυπηρέτηση αυτών. Μία λύση στο πρόβλημα αυτό, και για την αποφυγή ύπαρξης καταστάσεων που ένας scheduler δυσλειτουργεί ή το υπολογιστικό σύστημα στο οποίο ζει δεν μπορεί να ικανοποιήσει τις ανάγκες του ως προς τους πόρους που απαιτεί, είναι η αυτόματη ενημέρωση κάποιου admin, που διαχειρίζεται το σύστημα, προκειμένου

να δημιουργήσει και άλλους schedulers. Είναι σημαντικό ακόμα να ερευνηθεί στο πλαίσιο αυτό το κομμάτι της κύριας βάσης δεδομένων που χρησιμοποιούμε. Η παρούσα υλοποίηση αξιοποιεί τις δυνατότητες της MongoDB. Υπάρχουν όμως πλήθος άλλων noSql βάσεων, όπως για παράδειγμα του Redis, η χρήση του οποίου θα μπορούσε να βελτιώσει την απόδοση και την ταχύτητα των queries στη βάση. Ακόμα, θα μπορούσε να ερευνηθεί το κατά πόσο θα είχε νόημα ο συνδυασμός μίας κύριας και μίας δευτερεύουσας βάσης (πέραν αυτής του Google Cloud Storage), που θα χρησιμοποιούνται από τις δύο διαφορετικές οντότητες του συστήματός μας, προκειμένου να διαμοιραστεί ο φόρτος των λειτουργιών που κατά βάση εκτελούν οι schedulers.

Ένας ακόμα τομέας στον οποίο υστερεί το σύστημα, είναι το περιορισμένο πλήθος των τύπων ελέγχου που μπορούμε να κάνουμε. Σε μελλοντική έκδοση, θα μπορούσαμε να ενσωματώσουμε αιτήματα ελέγχου PORTS, TCP συνδέσεων, αποστολής μηνυμάτων μέσω του πρωτοκόλλου επικοινωνίας MQTT, καθώς και ελέγχους συνδέσεων και αποστολής μηνυμάτων σε socket.io server. Επιπλέον, θα μπορούσαμε να επεκτείνουμε τα μέσα ενημέρωσης (σε περίπτωση προβλήματος) των χρηστών. Το τωρινό σύστημα παρέχει δυνατότητες ενημέρωσης μέσω mail, αλλά θα μπορούσαμε ακόμα να επιτρέψουμε τη σύνδεση άλλων μορφών επικοινωνίας, όπως είναι οι διάφορες messaging εφαρμογές (Slack, Discord).

Μία ακόμα επέκταση που θα μπορούσαμε να κάνουμε σε επόμενο βήμα μας είναι η προσαρμογή του Krosswalk προκειμένου να μπορεί να χρησιμοποιηθεί και από άλλα εργαλεία που προσφέρουν δυνατότητες containerisation εφαρμογών (Podman, Lxd, Containerd, Buildah). Μιας και πλέον ανάλογα εργαλεία έχουν όλο και μεγαλύτερη απήχηση, παρατηρείται συνεχής αύξηση και βελτίωση των λειτουργιών που αυτά παρέχουν.

Πέρα από τη βελτίωση του backend του συστήματος, θα μπορούσαμε ακόμα να προσθέσουμε παραπάνω διαγράμματα, αξιοποιώντας περαιτέρω τα αποθηκευμένα δεδομένα μας και δείχνοντας παραπάνω χρήσιμη πληροφορία στο χρήστη.

Βιβλιογραφία

[1] Santhosh S and Narayana Swamy Ramaiah. "*Cloud-Based Software Development Lifecycle: A Simplified Algorithm for Cloud Service Provider Evaluation with Metric Analysis*". Big Data Mining and Analytics, 6(2):127–138, june 2023.

[2] Sorin POPA. "*WEB Server monitoring*". Annals of University of Craiova - Economic Sciences Series, 2(36):710–715, may 2008.

[3] Siddhesh Vaidya and Prabhat Padhy. "*View towards Synthetic Monitoring using HTTP Archive*". International Journal of Engineering Research and Technology, august 2022.

[4] Daniel Stenberg. "*HTTP2 explained*". ACM SIGCOMM Computer Communication Review, 44:120–128, 07 2014.

[5] Martino Trevisan, Danilo Giordano, Idilio Drago, and Ali Safari Khatouni. "*Measuring HTTP/3: Adoption and Performance*", 02 2021.

[6] Andy Neumann, Nuno Laranjeiro, and Jorge Bernardino. "*An Analysis of Public REST Web Service APIs*". IEEE Transactions on Services Computing, 14:957–970, 07 2021.

[7] Roy Thomas Fielding and Richard N. Taylor. "*Architectural Styles and the Design of Network-Based Software Architectures*". PhD thesis, University of California, Irvine, 2000. AAI9980887.

[8] CC BY-SA 4.0 via Wikimedia Commons Guillaume Chevalier. "*Schematic of the Long-Short Term Memory cell, a component of recurrent neural networks*", 2021. URL https://commons.wikimedia.org/wiki/File:LSTM_Cell.svg.

[9] Hezbullah Shah and Tariq Soomro. "*Node.js Challenges in Implementation*". Global Journal of Computer Science and Technology, 17:72–83, 05 2017.

[10] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. "*RustBelt: Securing the Foundations of the Rust Programming Language*". Proc. ACM Program. Lang., 2(POPL), dec 2017.

[11] Nicholas D. Matsakis and Felix S. Klock. "*The Rust Language*". In "*Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*", HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450332170.

[12] Abhinav Nagpal and Goldie Gabrani. "*Python for Data Analytics, Scientific and Technical Applications*". In "*2019 Amity International Conference on Artificial Intelligence (AICAI)*", pages 140–145, 2019.

[13] Arun Kumar and Supriya.P. Panda. "*A Survey: How Python Pitches in IT-World*". In "*2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon)*", pages 248–251, 2019.

[14] Oliver Duerr and Beate Sick. 2020.

[15] Stanley Bileschi, Shanqing Cai, and Eric Nielsen. 2020.

[16] Han Wu, Zhihao Shang, and Katinka Wolter. "*TRAK: A Testing Tool for Studying the Reliability of Data Delivery in Apache Kafka*". In "*2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*", pages 394–397, 2019.

[17] Rishika Shree, Tanupriya Choudhury, Subhash Chand Gupta, and Praveen Kumar. "*KAFKA: The modern platform for data management and analysis in big data domain*". In "*2017 2nd International Conference on Telecommunication and Networks (TEL-NET)*", pages 1–5, 2017.

[18] Antonio Brogi, Davide Neri, and Jacopo Soldani. "*DockerFinder: Multi-attribute Search of Docker Images*". In "*2017 IEEE International Conference on Cloud Engineering (IC2E)*", pages 273–278, 2017.

[19] Lara Lorna Jiménez, Miguel Gómez Simón, Olov Schelén, Johan Kristiansson, Kåre Synnes, and Christer Åhlund. "*CoMA: Resource Monitoring of Docker Containers.*". In "*CLOSER*", pages 145–154, 2015.

[20] Suchit Dhakate and Anand Godbole. "*Distributed cloud monitoring using Docker as next generation container virtualization technology*". In "*2015 Annual IEEE India Conference (INDICON)*", pages 1–5, 2015.

[21] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina Delimitrou. "*Sinan: Data-Driven, QoS-Aware Cluster Management for Microservices*", 2021.

[22] Quintin Fettes, Avinash Karanth, Razvan Bunescu, Brandon Beckwith, and Sreenivas Subramoney. "*Reclaimer: A Reinforcement Learning Approach to Dynamic Resource Allocation for Cloud Microservices*", 04 2023.

[23] Christian Esposito, Aniello Castiglione, and Kim-Kwang Raymond Choo. "*Challenges in Delivering Software in the Cloud as Microservices*". IEEE Cloud Computing, 3(5):10–14, 2016.

[24] Zhiyuan Xu, Jian Tang, Jingsong Meng, Weiyi Zhang, Yanzhi Wang, Chi Harold Liu, and Dejun Yang. "*Experience-driven Networking: A Deep Reinforcement Learning based Approach*", 2018.

[25] Ye Tao, Xiaodong Wang, Xu Xiaowei, and Yinong Chen. "*Dynamic Resource Allocation Algorithm for Container-Based Service Computing*". pages 61–67, 03 2017.

[26] Ming-Chang Lee, Jia-Chun Lin, and Ernst Gran. "*RePAD: Real-Time Proactive Anomaly Detection for Time Series*", pages 1291–1302. 03 2020. ISBN 978-3-030-44040-4.

[27] Ming-Chang Lee and Jia-Chun Lin. "*RePAD2: Real-Time, Lightweight, and Adaptive Anomaly Detection for Open-Ended Time Series*", 2023.

[28] Ming-Chang Lee, Jia-Chun Lin, and Ernst Gunner Gan. "*ReRe: A Lightweight Real-Time Ready-to-Go Anomaly Detection Approach for Time Series*". In "*2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC)*". IEEE, July 2020.