ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ
ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ
ΥΠΟΛΟΓΙΣΤΩΝ

Υλοποίηση και μελέτη συμπεριφοράς ενός RESTful API σε διαφορετικά οικοσυστήματα και αρχιτεκτονικές νέφους

Διπλωματική Εργασία

Δημήτριος Γ. Κομνηνός

Επιβλέπων: Βασίλειος Βεσκούκης
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Υλοποίηση και μελέτη συμπεριφοράς ενός RESTful API σε διαφορετικά οικοσυστήματα και αρχιτεκτονικές νέφους

Διπλωματική Εργασία

Δημήτριος Γ. Κομνηνός

Επιβλέπων: Βασίλειος Βεσκούκης
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 5η Ιουλίου

…….	…….	…….
Β. Βεσκούκης	Δ. Φωτάκης	Α. Παγουρτζής
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούλιος 2022

.....
Δημήτριος Γ. Κομνηνός
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Περίληψη

Το αντικείμενο της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και η μελέτη συμπεριφοράς ενός RESTful API σε διαφορετικά οικοσυστήματα και αρχιτεκτονικές νέφους.

Στο **πρώτο κεφάλαιο**, εξηγούνται οι βασικές έννοιες και ορολογίες του Cloud Computing και Serverless Computing, με απώτερο σκοπό την καλύτερη κατανόηση των επομένων κεφαλαίων. Επίσης αναφέρονται και τα σημαντικά πλεονεκτήματα της εκάστοτε τεχνολογίας.

Στο **δεύτερο κεφάλαιο**, μελετάμε μεμονωμένα όλες τις υπολογιστικές υπηρεσίες που έχουμε σκοπό να χρησιμοποιήσουμε. Σκοπός αυτού του κεφαλαίου είναι να εξηγήσουμε τα ιδιαίτερα χαρακτηριστικά της εκάστοτε υπηρεσίας και να εφαρμόσουμε καλές πρακτικές έτσι ώστε το τελικό πείραμα να είναι αξιόπιστο.

Στο **τρίτο κεφάλαιο**, μελετάμε μεμονωμένα όλες τις βάσεις δεδομένων που έχουμε σκοπό να χρησιμοποιήσουμε. Σκοπός αυτού του κεφαλαίου είναι να εξηγήσουμε τα ιδιαίτερα χαρακτηριστικά και πλεονεκτήματα της εκάστοτε βάσης, να εισάγουμε τα δεδομένα και να εφαρμόσουμε καλές πρακτικές για αξιόπιστο τελικό πείραμα.

Στο **τέταρτο κεφάλαιο**, ενώνουμε όλες τις υπηρεσίες και τα προϊόντα που μελετήθηκαν στα προηγούμενα κεφάλαια δημιουργώντας διάφορους deployment συνδυασμούς. Σκοπός μας είναι προσομοιώσουμε κίνηση πολλών χρηστών και να μελετήσουμε την συμπεριφορά του εκάστοτε deployment χρησιμοποιώντας συγκεκριμένες μετρικές.

Στο **πέμπτο κεφάλαιο**, αναφέρονται τα συμπεράσματα που προκύπτουν από την παρούσα διπλωματική εργασία και προτείνονται επόμενα βήματα για την συνέχιση της παρούσας ερευνητικής εργασίας.

Λέξεις Κλειδιά

Cloud, Cloud Computing, Τεχνολογίες νέφους, Serverless Computing, APIs, Deployment, Amazon Web Services, Google Cloud Platform, AWS DynamoDB, Cloud Firestore, AWS Lambda, AWS API Gateway, Google Cloud Run, GCP Cloud Functions, Flask, Zappa, Monolithic, Polylithic, Performance Testing

Abstract

The objective of this diploma thesis is the study of the behavior of a RESTful API in different ecosystems and cloud architectures.

The **first chapter** explains the basic concepts and terminologies of Cloud Computing and Serverless Computing, with a view to a better understanding of subsequent chapters, and the significant advantages of each technology are also mentioned.

In the **second chapter**, we study individually all the computing services that we intend to use. The purpose of this chapter is to explain the specific characteristics of each service and to apply best practices so that the final experiment is reliable.

In the **third chapter**, we study individually all the databases that we intend to use. The purpose of this chapter is to explain the specific characteristics and advantages of each database, bulk-write import and to apply best practices for a reliable final experiment.

In the **fourth chapter**, we bring together all the services and products studied in the previous chapters by creating various deployment combinations. Our aim is to simulate multi-user traffic and study the behavior of each deployment using specific metrics.

In the **fifth chapter**, the conclusions that emerge from this diploma thesis are mentioned and the next steps for the continuation of this research paper are proposed.

Keywords

Cloud, Cloud Computing, Serverless Computing, APIs, Deployment, Amazon Web Services, Google Cloud Platform, AWS DynamoDB, Cloud Firestore, AWS Lambda, AWS API Gateway, Google Cloud Run, GCP Cloud Functions, Flask, Zappa, Monolithic, Polylithic, Performance Testing

Πίνακας Περιεχομένων

Κεφάλαιο 1 Εισαγωγή

Σκοπός της παρούσας διπλωματικής εργασίας είναι να μελετήσουμε βασικές έννοιες και τεχνικές των Cloud Services, να αναλύσουμε τα ιδιαίτερα χαρακτηριστικά των προϊόντων που θα χρησιμοποιήσουμε και να μελετήσουμε την συμπεριφορά τους εφαρμόζοντας καλές πρακτικές.

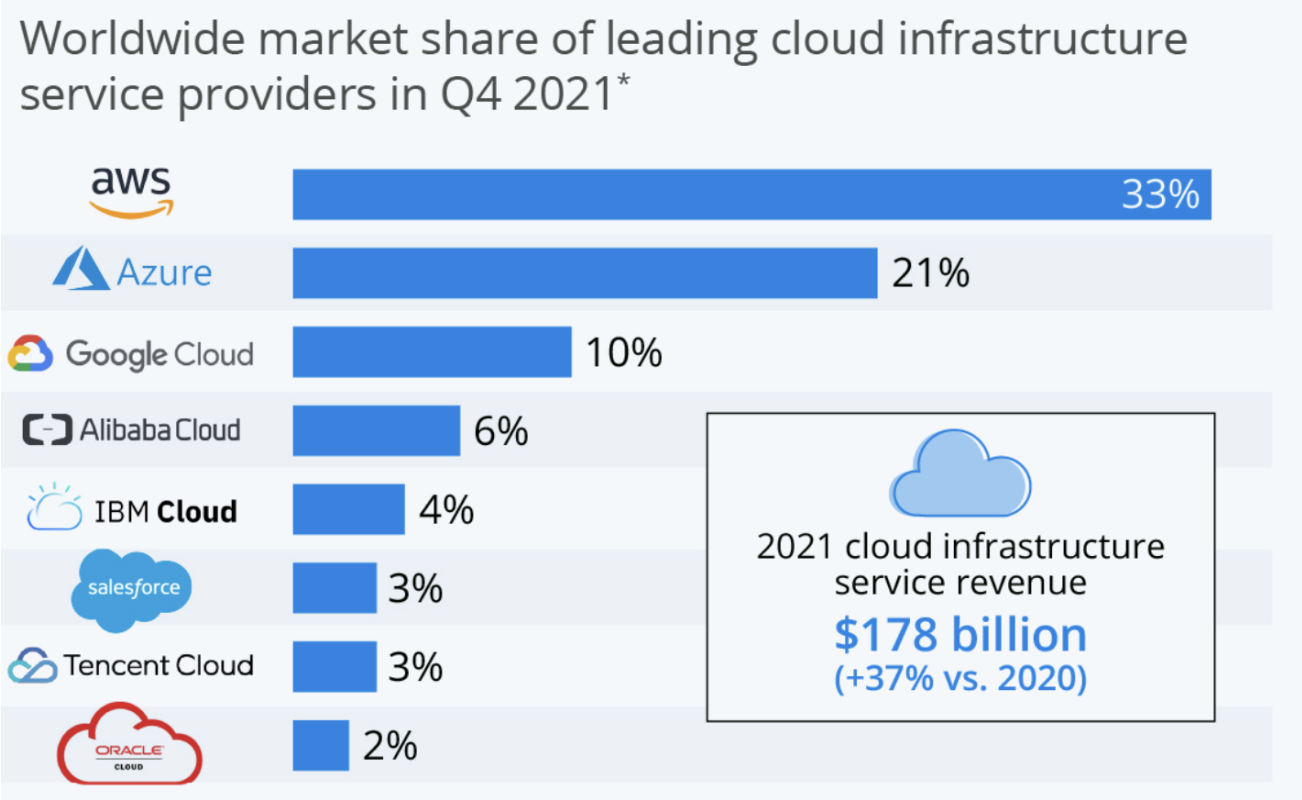
Αφού μελετήσουμε θεωρητικά όλες τις υπηρεσίες που μας ενδιαφέρουν, θα προχωρήσουμε σε κανονικά deployments, θα προσομοιώσουμε traffic spikes και θα αξιολογήσουμε όλα τα σενάρια χρησιμοποιώντας ειδικές μετρικές.

1.1 Τι είναι το Cloud Computing;

Cloud Computing ή αλλιώς *Web Services* ή αλλιώς *Cloud Services* είναι η παροχή διαφόρων computing services – δηλαδή Servers, Storage, Databases, Networking, Advanced analytics & Monitoring – over the Internet (“Cloud”).

Τα πιο γνωστά Web Services Platforms είναι:

1. ***AWS*** (Amazon Web Services)
2. ***GCP*** (Google Cloud Platform)
3. ***Microsoft Azure***



Εικόνα 1: Cloud providers market share

Στη παρούσα εργασία θα μελετήσουμε κυρίως προϊόντα του ***AWS*** και του ***GCP***.

1.1.1 Τα πλεονεκτήματα του Cloud Computing

Τα σημαντικότερα πλεονεκτήματα του Cloud είναι τα εξής:
(Azure, 2022)

- ***Κόστος***

  Εξαλείφεται, πλέον,
  - η χρηματική δαπάνη για την αγορά hardware & software,
  - η δημιουργία και λειτουργία επιτόπιων κέντρων δεδομένων — τα rack των διακομιστών,
  - η ηλεκτρική ενέργεια όλο το εικοσιτετράωρο για τροφοδοσία και ψύξη και
  - η ανάγκη από ειδικούς πληροφορικής για τη διαχείριση και συντήρηση της υποδομής.

- ***Απόδοση***

  Οι μεγαλύτερες υπηρεσίες λειτουργούν σε ένα παγκόσμιο δίκτυο ασφαλών κέντρων δεδομένων, τα οποία αναβαθμίζονται τακτικά στην τελευταία γενιά γρήγορου και αποτελεσματικού υπολογιστικού υλικού. Αυτό προσφέρει πολλά πλεονεκτήματα σε σχέση με ένα μόνο εταιρικό κέντρο δεδομένων, συμπεριλαμβανομένης της μειωμένης καθυστέρησης δικτύου για εφαρμογές και μεγαλύτερες οικονομίες κλίμακας.

- ***Αξιοπιστία***

  Όλα σχεδόν τα Web Services δημιουργούν αντίγραφα ασφαλείας δεδομένων για την ανάκτηση από καταστροφές. Καθιστούν την επιχειρηματική συνέχεια ευκολότερη και λιγότερο δαπανηρή, επειδή τα δεδομένα μπορούν να αντικατοπτρίζονται σε πολλαπλά redundant sites στο δίκτυο του παρόχου cloud.

- ***Παγκόσμια Κλίμακα***

  Τα οφέλη των υπηρεσιών υπολογιστικού νέφους περιλαμβάνουν τη δυνατότητα ελαστικής κλίμακας. Αυτό σημαίνει την παροχή της σωστής ποσότητας πόρων —για παράδειγμα, περισσότερη ή λιγότερη υπολογιστική ισχύ, αποθήκευση, εύρος ζώνης— ακριβώς όταν χρειάζονται και από τη σωστή γεωγραφική τοποθεσία.

- ***Παραγωγικότητα***

  Τα local data centers απαιτούν συνήθως πολλές ενέργειες όπως racking and stacking —ρύθμιση hardware, επιδιόρθωση λογισμικού και άλλες χρονοβόρες εργασίες διαχείρισης IT. Το cloud computing αφαιρεί την ανάγκη για πολλές από αυτές τις εργασίες, έτσι ώστε οι ομάδες IT να μπορούν να αφιερώνουν χρόνο στην επίτευξη πιο σημαντικών επιχειρηματικών στόχων. Με λίγα λόγια εξαλείφονται όλες οι DevOps διαδικασίες.

- ***Ασφάλεια***

  Πολλοί πάροχοι cloud προσφέρουν ένα ευρύ σύνολο πολιτικών, τεχνολογιών και ελέγχων που ενισχύουν την συνολική ασφάλεια, συμβάλλοντας στην προστασία των δεδομένων, των εφαρμογών και της υποδομής από πιθανές απειλές.

1.2 Τι είναι το Serverless Computing;

(Cloudflare, What is serverless, 2022)
(Cloudflare, Why use serverless, 2022)

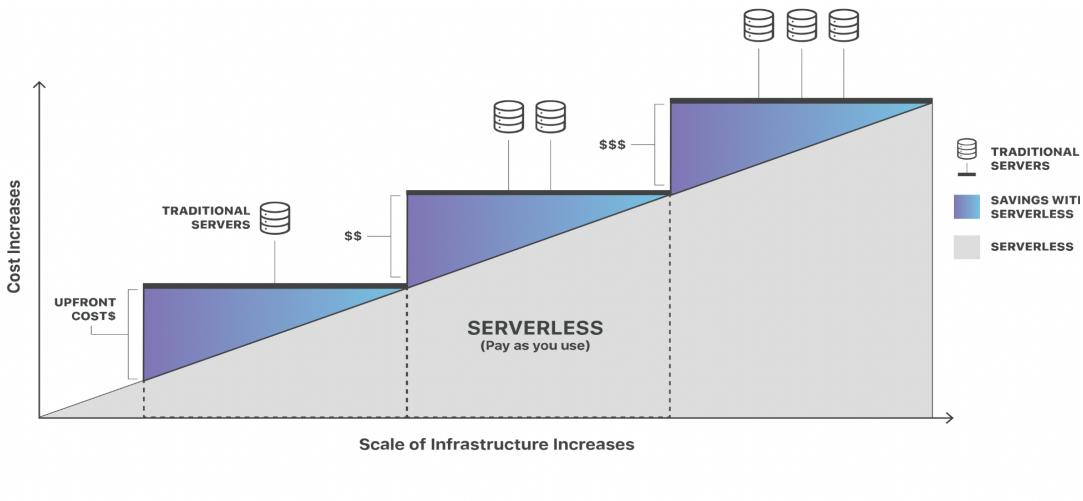
Το Serverless Computing είναι μία τεχνική παροχής Back-End services αλλά με την εξής σημαντική διαφορά: λειτουργεί on an *as-used basis* ή αλλιώς *on-demand*, που σημαίνει πως ο χρήστης πληρώνει μόνο για αυτά που χρησιμοποιεί και όταν τα χρησιμοποιεί.

Αυτή η αρχιτεκτονική έρχεται να αντικαταστήσει την παραδοσιακή τεχνική που αποτελείται κατά κύριο λόγο από dedicated servers που λειτουργούν συνεχώς 24/7 και “ακούνε” για client requests χωρίς να υπάρχει κάποιο traffic.
Αξίζει, βέβαια, να σημειώσουμε πως ενώ το όνομα “serverless” παραπέμπει σε κάτι που δεν χρησιμοποιεί servers, στην πραγματικότητα χρησιμοποιούνται κανονικά physical servers, όμως οι developers δεν χρειάζεται να ασχοληθούν με τίποτα παραπάνω πέρα από τον κώδικά τους.

Επίσης το Serverless Computing θεμελιώθηκε εξ αρχής πάνω στο Cloud καθώς βασίζεται πάνω σε principles όπως Scalability, Performance, Availability, Concurrency, Portability, Security κ.α. που θα αναλύσουμε στην συνέχεια.
Η serverless τοπική ανάπτυξη θεωρείται anti-pattern και δεν χρησιμοποιείται (McCumskey, 2021).

1.2.1 Τα πλεονεκτήματα του Serverless Computing

- ***Αποδοτικότητα κόστους***



Εικόνα 2: Serverless cost efficiency

Το Serverless computing επιτρέπει στους developers να ακολουθήσουν ένα “pay-as-you-go” σύστημα και να πληρώνουν τις υπηρεσίες που μόνο χρησιμοποιούν και όποτε τις χρησιμοποιούν. Ο κώδικας εκτελείται μόνο όταν χρειάζεται και αυξάνεται αυτόματα ανάλογα με τις ανάγκες. Το provisioning είναι δυναμικό, ακριβές και real-time. Συγκεκριμένα, μερικοί πάροχοι αναλύουν τις χρεώσεις τους σε προσαυξήσεις των 100 χιλιοστών του δευτερολέπτου. Σε αντίθεση με μία παραδοσιακή αρχιτεκτονική αποτελούμενοι από dedicated servers, που οι developers πρέπει να εκτιμήσουν το πιθανό traffic και να αγοράσουν τους απαραίτητους πόρους χρησιμοποιώντας τους ή όχι.

- ***Simplified Scalability***

  Οι Cloud Providers χειρίζονται εξ ολοκλήρου την κλιμάκωση on-demand με βάση το εκάστοτε traffic χρησιμοποιώντας τεχνολογίες όπως Kubernetes. Οι developers με ελάχιστο configuration μπορούν να χειριστούν auto-scaling λειτουργίες αλλάζοντας παραμέτρους όπως concurrency, provision κ.α.

  Serverless εφαρμογές μπορούν να διαχειριστούν ένα ασυνήθιστο και ξαφνικό υψηλό αριθμό από requests το ίδιο γρήγορα και αποδοτικά όπως θα χειριζόντουσαν ένα request από έναν μόνο χρήστη. Μια παραδοσιακή αρχιτεκτονική με fixed resources δεν θα μπορούσε να ανταπεξέλθει σε τέτοιες περιπτώσεις (spikes).

1.2.2 Serverless Cloud Products

Κάθε Cloud Provider έχει τα δικά του Serverless προϊόντα σε διάφορους τομείς όπως Databases, Lambda Functions, API Gateways, Stateless Containers κ.α.

Εμείς θα κάνουμε deploy κάποια back-end services χρησιμοποιώντας τα εξής, με διάφορους συνδυασμούς:

Amazon Web Services:
1. AWS Lambda – FaaS
2. AWS API Gateway
3. AWS DynamoDB – NoSQL Database

Google Cloud:
1. GCP Cloud Functions – FaaS
2. GCP Cloud Run – Stateless Containers
3. GCP API Gateway
4. Cloud Firestore (GCP) – NoSQL Database

Παρακάτω φαίνεται ένας συγκεντρωτικός πίνακας με τα πειράματα που θα διεξαχθούν στο κεφάλαιο 4, δείχνοντας ακριβώς τις υπηρεσίες που θα χρησιμοποιήσουμε στο κάθε πείραμα.

Scenario	Cloud Firestore	AWS DynamoDB	AWS Lambda	AWS API Gateway	Cloud Functions	Cloud Run	Cloud Run (concurrency)
4.4.1	•					•	4
4.4.2	•				•		
4.4.3	•		•	•			
4.4.4		•	•	•			
4.4.5		•				•	80
4.4.6		•				•	4

Κεφάλαιο 2 Αυτόνομη λειτουργία των υπολογιστικών υπηρεσιών - Καλές πρακτικές

2.1 Initial Testing Setup

Σε πρώτο στάδιο θα μελετηθούν τα services μεμονωμένα χωρίς κάποιο integration με κάποια βάση δεδομένων έτσι ώστε να ξεκαθαρίσουμε τις θεμελιώδεις λειτουργίες τους, να αναλύσουμε καλές πρακτικές και να εξάγουμε κάποια συμπεράσματα σύμφωνα με συγκεκριμένες μετρικές, οι οποίες είναι ενσωματωμένες στα συστήματα αυτά.

Οπότε σκοπός μας είναι να φτιάξουμε κάποια ψεύτικα testing APIs.
Στην αρχή χρησιμοποιήσαμε time.sleep() για το testing αλλά παρατηρήσαμε πως είναι κακή πρακτική, καθώς δεν καταναλώνει CPU time αλλά περιμένει κάποιο INTERRUPT από τον kernel του συστήματος. Στην ουσία γίνεται handle από το λειτουργικό και δεν είναι αξιόπιστη τεχνική. Σε αυτό το σημείο να προσθέσουμε πως οι Cloud Providers χρεώνουν με βάση το CPU time που χρησιμοποιεί το εκάστοτε function / service, οπότε και από άποψη pricing measuring θα ήταν μία λάθος επιλογή.

Συνεπώς καταλήξαμε με dummy for-loops που κάνουν dummy υπολογισμούς. Φτιάχτηκαν, λοιπόν, 2 APIs το SoftSleep και το HardSleep.

```python
class SoftSleep(Resource):

    @configuration.measure_time
    def get(self, sleep_id):

        # Initialize global times keeper
        configuration.times = {}

        a = 0
        for i in range(1000):
            for j in range(10000):
                a += i * j

        return {
            'times': configuration.times,
            'msg': sleep_id
        }
```

Εικόνα 3: Python code of SoftSleep function

```python
class HardSleep(Resource):

    @configuration.measure_time
    def get(self, sleep_id):

        # Initialize global times keeper
        configuration.times = {}

        a = 0
        for i in range(3000):
            for j in range(10000):
                a += i * j

        return {
            'times': configuration.times,
            'msg': sleep_id
        }
```

Εικόνα 4: Python code of HardSleep function

Εδώ κάνουμε 2 παρατηρήσεις:

1. ***sleep_id***: είναι ξεκάθαρο πως το sleep_id δεν χρησιμοποιείται πουθενά, αλλά με αυτήν την τεχνική εξασφαλίζουμε πως το service δεν θα κάνει κάποιο URL Caching, καθώς κάθε request θα έχει διαφορετικό URL. Προφανώς, μπορούμε να ελέγξουμε το caching και από τις ίδιες τις πλατφόρμες (AWS, GCP) αλλά εξαλείφουμε και οποιοδήποτε caching έχει γίνει σε επίπεδο δικτύου, έτσι ώστε να είμαστε σίγουροι πως οι χρόνοι στο testing είναι αντιπροσωπευτικοί.

2. ***measure_time wrapper***: Πριν χρησιμοποιηθεί κάποιο testing suite, το testing έγινε locally και για να εξαλείψουμε τον παράγοντα κακού δικτύου φτιάχτηκε ένας wrapper που “κολλάει” σε κάθε function έτσι ώστε να ξέρουμε ακριβώς τον χρόνο που έτρεξε το function στο εκάστοτε Cloud Service.

```python
def measure_time(func):
    def wrap(*args, **kwargs):
        start = time.time()
        result = func(*args, **kwargs)
        end = time.time()

        times[func.__name__] = end - start

        return result

    return wrap
```

Εικόνα 5: Python code of measure_time wrapper

Συνεπώς το response στο endpoint *base_url/soft_sleep/hello* είναι κάπως έτσι:


```
“times”: {
        “get”: 0.835730
},
“msg”: “hello”
```


Εικόνα 6: Response of base_url/soft_sleep/hello

2.2 Google Cloud Run

(Google, Cloud Run, 2022)
Το Cloud Run είναι μία εντελώς αυτο-διαχειριζόμενη πλατφόρμα που μας επιτρέπει να κάνουμε deploy containers τα οποία μπορούν να ενεργοποιηθούν είτε μέσα από requests είτε μέσα από events.

Αυτή η λογική, λοιπόν, μας θυμίζει την Serverless νοοτροπία όπου το Cloud (δηλαδή το GCP) χειρίζεται όλο το Hardware Infrastructure και όλες τις DevOps διαδικασίες με σκοπό εμείς να επικεντρωθούμε στον κώδικα.

2.2.1 Ιδιαίτερα πλεονεκτήματα του Cloud Run

Το Serverless από μόνο του έχει ιδιαίτερα χαρακτηριστικά, το Cloud Run, όμως, έχει κάποια χαρακτηριστικά που το ξεχωρίζουν από τα υπόλοιπα προϊόντα.

- ***Portability***

  Στην ουσία όταν θέλουμε να φτιάξουμε ένα Cloud Run Service φτιάχνουμε ένα container.
  Ένα container έχει την δυνατότητα να γίνει deploy σε οποιοδήποτε περιβάλλον είτε αυτό είναι Kubernetes/GKE, είτε Cloud Run (serverless and GKE), είτε VM, οπουδήποτε.

  Από την άλλη, όταν γράφουμε για ένα Cloud Function, ή για οποιοδήποτε FaaS προϊόν, υποθέτουμε ότι:
  - Υπάρχει κάποιος Web-Server για να κάνει route τα requests μας μέσα στον κώδικα,
  - Γνωρίζουμε το περιβάλλον στο οποίο τρέχει ο κώδικάς μας
  - Το περιβάλλον πρέπει να υποστηρίζει την γλώσσα στην οποία γράφουμε

  Και αρκετοί άλλοι περιορισμοί.

  Χρησιμοποιώντας το Cloud Run καταφέρνουμε να μειώνουμε το “rework” και το “refactoring” κάθε φορά που θέλουμε να αλλάξουμε περιβάλλον και το κυριότερο είναι

ότι είμαστε λιγότερο “εγκλωβισμένοι” στο Google Cloud, καθώς ανά πάσα στιγμή παίρνουμε το container και το κάνουμε deploy σε άλλο provider.

- ***Testability***

  Σε μεγάλα και περίπλοκα projects ή σε projects όπου το production level είναι πολύ critical μία πολύ σημαντική παράμετρος είναι το testability. Προφανώς μία εφαρμογή πριν βγει στο production level περνάει από ένα staging level όμως και πάλι δεν μπορούμε να κάνουμε πλήρες local testing.
  Επίσης, δεν αναφερόμαστε σε unit tests αλλά σε end-to-end tests ή αλλιώς “integration” tests.

  Φτιάχνοντας, λοιπόν, containers μπορούμε να κάνουμε “docker run” και να δούμε ακριβώς πως λειτουργεί το container και πως αλληλεπιδρά με διάφορα events.
  Σε αντίθεση με Cloud Function, ο κώδικας που γράφουμε απλά χειρίζεται το request που έχει ήδη γίνει forward από τον Web-Server του περιβάλλοντος εκτέλεσης.

2.2.2 Use Cloud Run Build

Ακολουθώντας τα επίσημα έγγραφα της Google:
(Google, Deploy a Python service to Cloud Run, 2022)

Όταν χρησιμοποιούμε το gcloud CLI και τρέχουμε την εντολή gcloud run deploy,
που είναι ίση με :

1. gcloud builds submit --tag [IMAGE] /path/to/sourcecode &&
2. gcloud run deploy [SERVICENAME] --image [IMAGE]

τότε το CLI αναλαμβάνει:
(Άμα είναι η πρώτη φορά που το χρησιμοποιούμε θα φτιαχτεί το Container Repository ανάλογα με το region που επιλέγουμε να αποθηκευτεί το container)

1. Να ανεβάσει τον κώδικα (source code files) στο Cloud Build, όπου εκεί χτίζεται το image πάνω στο Cloud.
2. Να δημιουργήσει το Cloud Run Service το οποίο χρησιμοποιεί το image που δημιουργήθηκε προηγουμένως.
3. Να δρομολογήσει όλο το traffic (και το 100%) στο νέο revision που δημιουργήθηκε. Αυτό χρησιμεύει αν έχουμε πολλά versions ενός Cloud Run Service. Χρησιμοποιείται εσωτερικός Load Balancer για να κάνει handle το οποιοδήποτε traffic.

2.2.3 Build Docker Locally & Upload it

Ακολουθώντας το επίσημο documentation της Google:
(Google, Deploy a Python service to Cloud Run, 2022)

Σημείωση: Με κόκκινα γράμματα φαίνονται οι εντολές που γράφτηκαν στην πραγματικότητα, με απώτερο σκοπό να γίνει

1. Πηγαίνουμε αρχικά στον φάκελο με τον source code μας:

   docker build -t [DOCKER-TAG] .
   ```
   $ docker build -t cloud-run-test .
   ```

   Αυτή η εντολή ψάχνει ένα Dockerfile στο directory στο οποίο εκτελείται (εδώ είναι το current directory καθώς βάζουμε “τελεία”) και φτιάχνει ένα container με TAG που του δίνουμε εμείς.
   Μιας και έχουμε local το container μπορούμε:
   - να το δούμε τρέχοντας “docker images”
   - να το τρέξουμε γράφοντας “docker run …”
   - να το δούμε ότι τρέχει με την εντολή “docker ps”
   - να δούμε τα logs του με “docker logs <docker-tag>”

2. Πρέπει να ενεργοποιήσουμε το API που μας επιτρέπει να ανεβάζουμε δικά μας container στο GCP Container Registry. Συνδέεται στην ουσία με το dockerhub και για αυτό όπως θα δούμε στην συνέχεια μπορούμε να χρησιμοποιούμε εντολές όπως docker push και docker pull.

   ```
   $ gcloud services enable containerregistry.googleapis.com
   ```

3. Θα προσθέσουμε ένα ακόμα Tag στο docker image μας με βάση το registry name, χρησιμοποιώντας την εξής εντολή:

   docker tag [SOURCE_IMAGE] [HOSTNAME]/[PROJECT-ID]/[IMAGE]:[TAG]
   ```
   $ docker tag cloud-run-test:latest eu.gcr.io/cloud-engineering-974ea/cloud-run-test:1.1
   ```

4. Τέλος, κάνουμε push το image στο GCP container registry

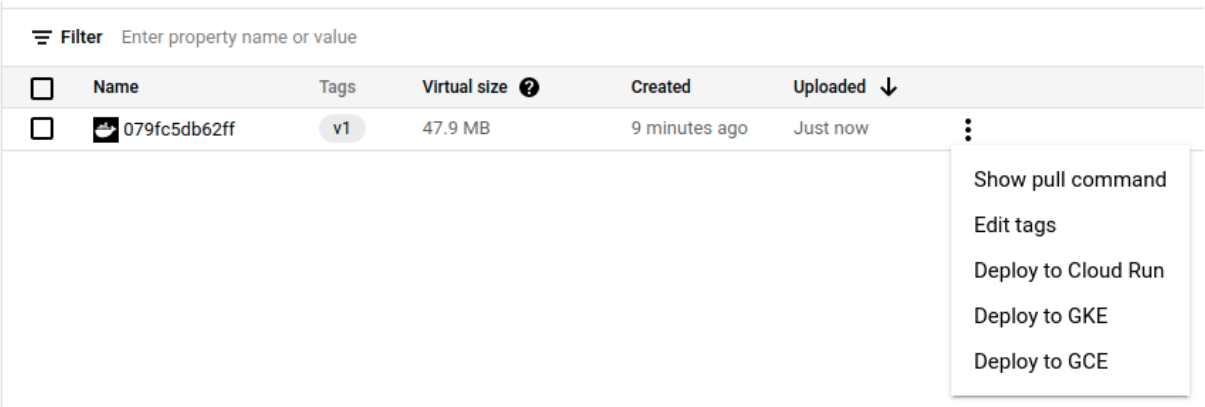
   docker push [HOSTNAME]/[PROJECT-ID]/[IMAGE]:[TAG]
   ```
   $ docker push eu.gcr.io/cloud-engineering-974ea/cloud-run-test:1.1
   ```

Μόλις ανεβάσαμε το Image μας στο Container Registry και μπορούμε να δημιουργήσουμε τώρα το Cloud Run Service μας.

Ουσιαστικά υπάρχουν 2 τρόποι να γίνει αυτό, είτε μέσα από το Console είτε μέσα από το CLI.

- ***Web Console***

  Είτε από το Cloud Run “Create a Service” είτε και από το ίδιο το directory του container που ανεβάσαμε στο Container Registry.



Εικόνα 7: Container registry console

- ***CLI***

  Η μοναδική προϋπόθεση για να μπορέσουμε να χρησιμοποιήσουμε το CLI είναι να έχουμε φτιάξει Google Cloud λογαριασμό. Για να μπορέσουμε να το εγκαταστήσουμε αρκεί να ακολουθήσουμε τις επίσημες οδηγίες (Google, Install the gcloud CLI, 2022)

  Χρησιμοποιούμε, λοιπόν, την εξής εντολή:
  gcloud run deploy <cloud-run-service-name>
  --image [HOSTNAME]/[PROJECT-ID]/[IMAGE]:[TAG]

```
$ gcloud run deploy cloud-run-test
      -concurrency=20
      -cpu=1
      -memory=1Gi
      -min-instances=1
      -max-instances=100
      -allow-unauthenticated -image=eu.gcr.io/cloud-engineering-974ea/cloud-
      run-test:1.1
```

  Εδώ προσθέτουμε κι άλλες σημαντικές παραμέτρους για το Cloud Service που θα μελετηθούν στην συνέχεια όπως CPU, Memory, Concurrency κ.α.

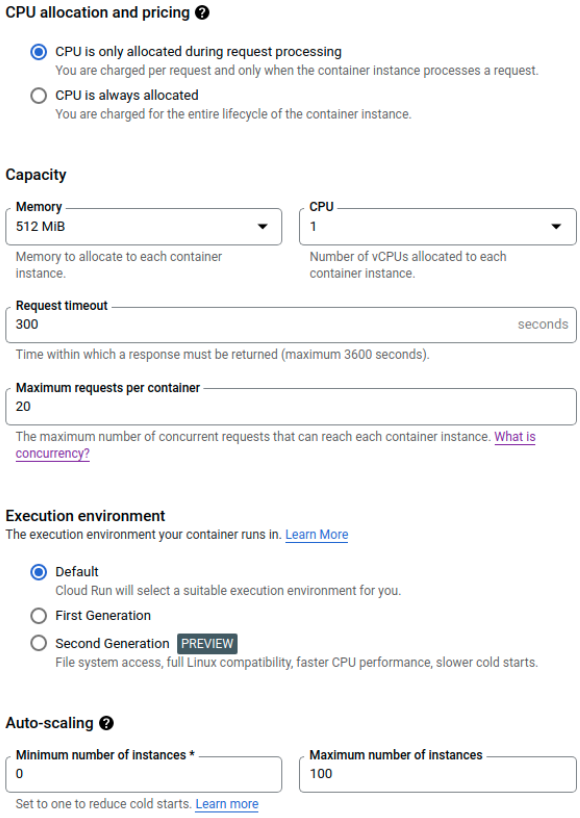
2.2.4 Update

Στην ουσία άμα θέλουμε να ενημερώσουμε ένα Cloud Run Service απλά ξανακάνουμε deploy ένα image (ακολουθώντας όποιον τρόπο θέλουμε) και απλά ορίζουμε ως όνομα το Service που θέλουμε να κάνουμε update.
Αυτομάτως, λοιπόν, δημιουργείται ένα καινούριο revision που του ανατίθεται 100% του traffic. Το παλιό revision δεν χάνεται, μπορούμε ανά πάσα στιγμή να του δώσουμε traffic.

		Name	Traffic	Deployed	Revision URLs (tags)	Actions
○	✓	cloud-run-test-00010-xaz	100% (to latest)	17 hours ago		⋮
○	✓	cloud-run-test-00009-ged	0%	17 hours ago		⋮
◉	✓	cloud-run-test-00008-zav	0%	2 days ago	+	⋮
○	✓	cloud-run-test-00007-cav	0%	2 days ago		⋮
○	✓	cloud-run-test-00006-nin	0%	2 days ago		⋮
○	✓	cloud-run-test-00005-mur	0%	2 days ago		⋮
○	✓	cloud-run-test-00004-cix	0%	2 days ago		⋮
○	✓	cloud-run-test-00003-beq	0%	2 days ago		⋮
○	✓	cloud-run-test-00002-jir	0%	2 days ago		⋮
○	✓	cloud-run-test-00001-keq	0%	2 days ago	+	⋮

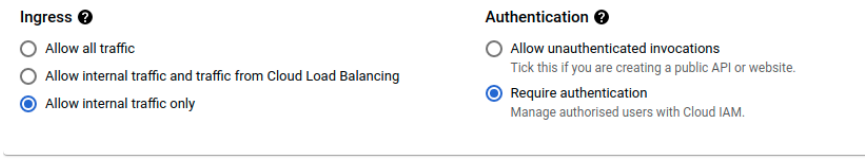
Εικόνα 8: Cloud Run Service revisions

Επίσης δημιουργείται καινούριο revision όταν γίνεται οποιαδήποτε αλλαγή στις παραμέτρους.



Εικόνα 9: Update revision

Ή αλλάζοντας τον τρόπο με τον οποίο γίνεται trigger το service.



Εικόνα 10: Revision traffic

2.2.5 Παράμετροι

Ένας cloud engineer σε πρώτη φάση, ανάλογα με το εκάστοτε use case, καλείται να επιλέξει τον σωστό cloud provider και το σωστό stack από επιμέρους cloud products. Στην συνέχεια, όμως, οφείλει να τα παραμετροποιήσει με τέτοιο τρόπο έτσι ώστε να υπάρξει μία ισορροπία μεταξύ κόστους, ταχύτητας, διαθεσιμότητας, επεκτασιμότητας κ.ο.κ.

Σε αυτό το σημείο, λοιπόν, θα μελετήσουμε όλες τις παραμέτρους που επηρεάζουν την γενικότερη επίδοση του Cloud Run και τι αλλαγές επιφέρει η κάθε μία.

1. ***Max Instances***

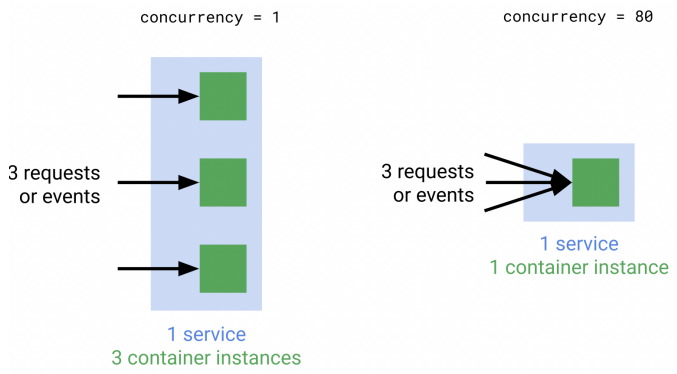
   Αυτή η παράμετρος επηρεάζει την κλιμακωσιμότητα και την ικανότητα του συστήματος να ανταπεξέρχεται σε κάποιο traffic spike. Αν ήταν, ας πούμε, η τιμή ίση με 1 δεν θα μπορούσαμε να διαχειριστούμε πολλά ταυτόχρονα requests, καθώς θα αποτύγχαναν λόγω timeout μέχρι να εξυπηρετηθούν.

2. ***Minimum instances***

   Αυτή η παράμετρος επηρεάζει τα λεγόμενα Cold Starts και άρα την απόδοση. Ως Cold Start ορίζουμε την κατάσταση που έρχεται ένα request ενώ το σύστημα πριν ήταν σε πλήρη αδράνεια. Αυτά τα πρώτα requests έχουν αυξημένο latency καθώς πρέπει να δημιουργηθούν τα πρώτα active instances, οπότε για αυτό ορίζουμε κάποια minimum instances, παραπάνω από 0, έτσι ώστε να υπάρχουν κάποια idle containers για να απαντήσουν μέχρι το σύστημα να “σηκώσει” κι άλλα active containers.

3. ***Concurrency***

   Αυτή η παράμετρος, προφανώς, καθορίζει το Concurrency και επιτρέπει σε ένα container να χειριστεί μέχρι και παραπάνω από 1 request και εκείνο είναι μετά υπεύθυνο για το πως θα τα χειριστεί. Στην δική μας περίπτωση το Flask υποστηρίζει multi-threading. Αν βάζαμε ίσο με 1 τότε στην πραγματικότητα θα ήταν σαν να είχαμε Google Cloud Functions.



Εικόνα 11: Cloud Run concurrency

Αυτή η παράμετρος, όμως, όπως θα δούμε, επηρεάζει και την απόδοση και το κόστος. Δηλαδή, άμα έχουμε concurrency=1 και max-instances=100 τότε για να διαχειριστούμε 100 ταυτόχρονα requests θα χρειαστούμε 100 διαφορετικά containers που αυτό σημαίνει αυξημένο κόστος. Ενώ άμα είχαμε concurrency=20, ναι μεν θα χρειαζόμασταν μόνο 5 containers, αλλά θα μειωνόταν η απόδοση καθώς ένα container με συγκεκριμένους πόρους θα έπρεπε να διαχειριστεί περισσότερα requests.

4. ***Memory***

Αυτή η παράμετρος επηρεάζει προφανώς την απόδοση, την ανθεκτικότητα (Resilience) και την multi-threading ικανότητα των containers. Έστω, δηλαδή, ότι βάζουμε concurrency=100 και threads=10, τότε ένα container θα προσπαθήσει να φτιάξει όσα processes χρειάζονται με 10 threads το καθένα. Εάν, έχουμε βάλει memory=512Mb, ανά container, και το κάθε request έχει ανάγκη από μεγάλη μνήμη στο runtime τότε κινδυνεύουμε να υπάρξουν αρκετά 5XX errors εξαιτίας μη επαρκούς μνήμης.

5. ***CPU***

Εδώ ορίζουμε τα CPU cores επιλέγοντας 1,2 ή 4. Αυτή η παράμετρος επηρεάζει την απόδοση, όταν έχουμε υψηλό Concurrency, και το κόστος.
Δηλαδή, άμα έχουμε concurrency=1 τότε 1 CPU core είναι αρκετός.
Αυτή η παράμετρος έχει πρακτική αξία όταν συνδυάζεται σωστά και με τους workers.

6. ***Workers***

Ας σκεφτούμε ότι ο αριθμός των workers καθορίζει τα πόσα processes μπορεί να γεννήσει ένα container. Αυτή η παράμετρος ορίζεται από το Dockerfile και επηρεάζει ττην απόδοση όταν θέλουμε να έχουμε υψηλό concurrency. Συνεπώς μία καλή πρακτική είναι ο αριθμός των CPU Cores να ταυτίζεται με τον αριθμό των workers.

7. ***Threads***

Και αυτή η παράμετρος καθορίζεται από το Dockerfile και ορίζει τα πόσα threads μπορεί να γεννήσει το εκάστοτε process ενός Container.

Το Cloud Run, λοιπόν, πέρα από την νοοτροπία του Stateless Containers το οποίο από μόνο του προσδίδει αρκετή ελευθερία, μέσα από αυτές τις παραμέτρους μας επιτρέπει να διαχειριστούμε όπως εμείς θέλουμε το Scalability και το Concurrency ανάλογα με τις ανάγκες μας ως προς το κόστος και την γενική απόδοση.
Π.χ. Αν θέσουμε:

- workers=1, threads=1, concurrency=1:

τότε προσομοιώνουμε τα Cloud Functions, έχουμε υψηλό Performance αλλά αυξάνεται και το κόστος.

- workers=4, threads=1, concurrency=4, cpu=4:

και έτσι τότε προσομοιώνουμε τα Cloud Functions, έχοντας πάλι έχουμε υψηλό Performance αλλά και υψηλό κόστος. Μπορεί έτσι να μειώνονται τα συνολικά active containers, όμως κάθε container κοστίζει περισσότερο καθώς έχει μεγαλύτερη CPU και Memory.

2.2.6 Πειράματα & Συμπεράσματα

Εδώ, λοιπόν, θα μελετήσουμε την συμπεριφορά του γενικότερου συστήματος δοκιμάζοντας διάφορους συνδυασμούς των παραπάνω παραμέτρων. Θα χρησιμοποιήσουμε μόνο 2 endpoints για το testing. Το *testing/soft_sleep* και το *testing/hard_sleep*, τα οποία όπως έχουμε εξηγήσει κάνουν dummy υπολογισμούς για να καταναλώνουν όντως CPU time και να μην γίνεται απλό sleep. Έχοντας βάλει, λοιπόν, measure_time wrapper στα endpoints μπορούμε να γνωρίζουμε ακριβώς το CPU time που καταναλώνουν. Συγκεκριμένα:

- soft_sleep: 0.899 seconds
- hard_sleep: 2.580 seconds

Έχοντας αυτά τα 2 endpoints, λοιπόν, θα φτιάξουμε προσομοιώσεις από traffic spikes, με διάρκεια 2-3 λεπτών, αποτελούμενα από 100+ χρήστες, και θα παρατηρούμε τις απαραίτητες μετρικές.

Παρακάτω, λοιπόν, ακολουθεί ο πίνακας με τους διαφορετικούς συνδυασμούς των παραμέτρων που έχουμε σκοπό να δοκιμάσουμε. Σε κάθε νέο σενάριο φαίνεται με κόκκινα η παράμετρος που αλλάζει και στην συνέχεια θα εξηγήσουμε:

1) γιατί αλλάζουμε την εκάστοτε παράμετρο και
2) πως περιμένουμε να επηρεαστεί το Cloud Run Service.

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	***1000***	30
2.2.6.4	***2***	1	***2***	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	***80***	1	1000	30
2.2.6.6	2	***4***	2	1024	80	1	1000	30
2.2.6.7	***4***	4	***4***	***2048***	80	1	1000	30
2.2.6.8	4	***1***	4	2048	***40***	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	***90***

Εικόνα 12: Πίνακας παραμέτρων Cloud Run

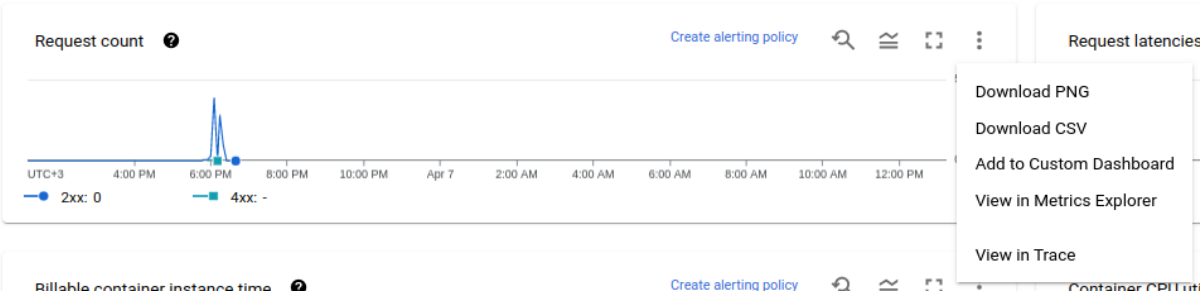
2.2.6.1 Εργαλεία

Πριν ξεκινήσουμε, να αναφέρουμε τα εργαλεία που θα χρησιμοποιήσουμε για τα tests και για το monitoring.

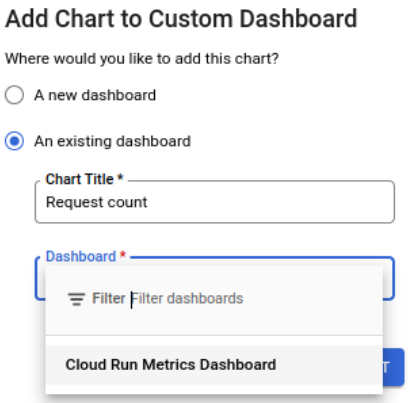
Για το testing θα χρησιμοποιήσουμε το ***Loadium***. Είναι cloud based load testing εργαλείο για performance tests χρησιμοποιώντας open-source tools, όπως JMeter, Gatling κ.α. (Loadium, 2022)

Για το monitoring θα χρησιμοποιήσουμε το ***GCP Monitoring***. Αυτό το προϊόν μας επιτρέπει να φτιάχνουμε δικά μας dashboards από διάφορες μετρικές πάνω σε όσα services χρησιμοποιούμε στο GCP. Εμείς, όμως, θέλουμε συγκεκριμένες μετρικές του Cloud Run οπότε τα βήματα είναι τα εξής:

1. Φτιάχνουμε στην αρχή ένα κενό dashboard.
2. Κατευθυνόμαστε στην σελίδα του Cloud Run Service που μας ενδιαφέρει και επιλέγουμε το tab “Metrics”.
3. Στην συνέχεια επιλέγουμε το διάγραμμα που μας ενδιαφέρει και το προσθέτουμε στο Dashboard μας

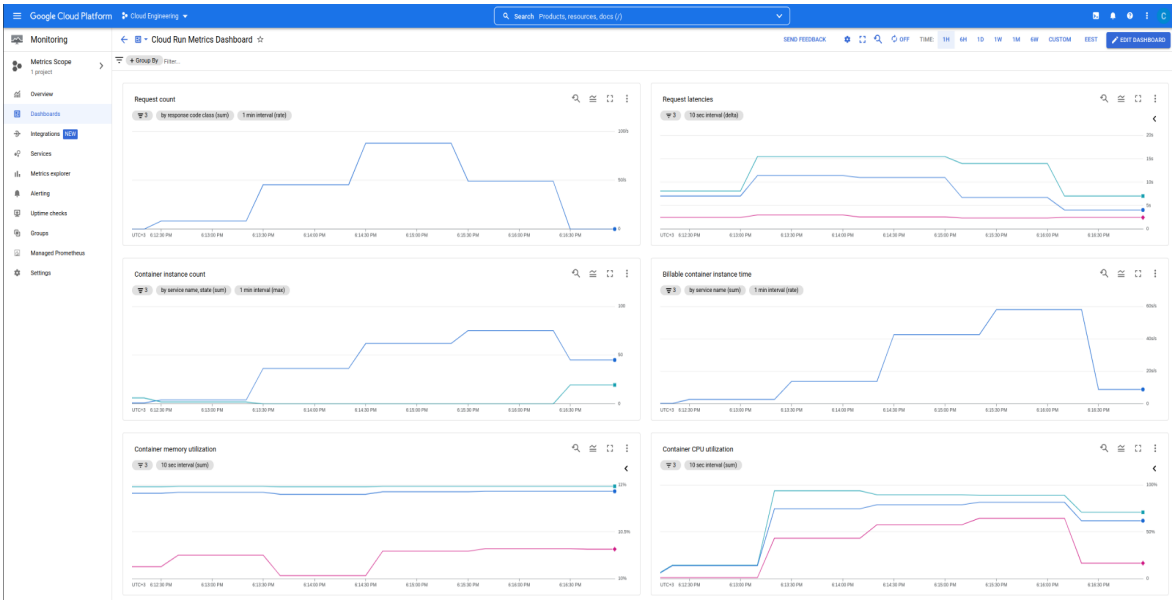


Εικόνα 13: Request count chart



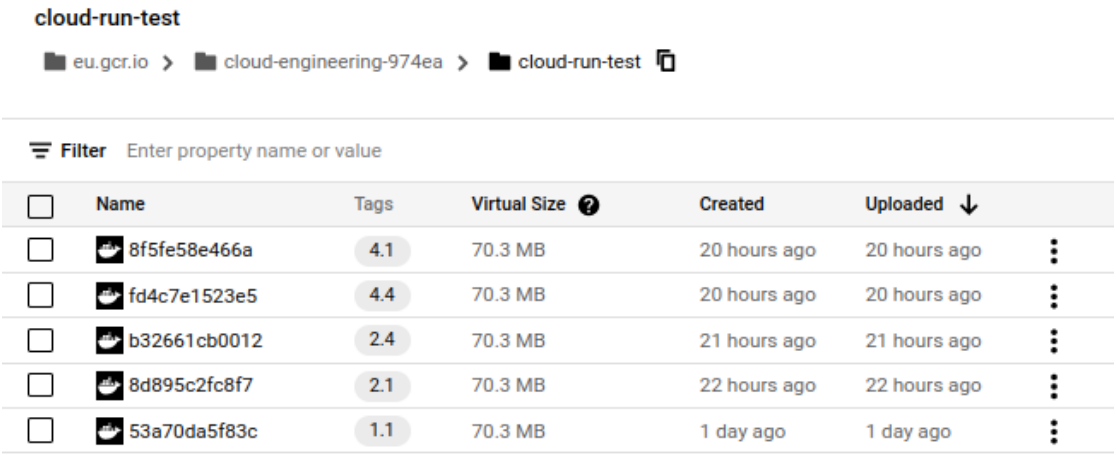
Εικόνα 14: Add chart to custom dashboard

4. Το τελικό custom dashboard με όνομα “Cloud Run Metrics Dashboard” φαίνεται κάπως έτσι.



Εικόνα 15: Cloud Run Metrics Dashboard

Παρακάτω για λόγους ευκολίας θα ορίζουμε τα διάφορα cases ανάλογα με τις παραμέτρους, δηλαδή το W 1, T 1, CPU 1, M 512, CONC 20, MIN 1, MAX 100 σημαίνει ότι έχουμε ορίσει workers=1, threads=1, cpu=1 core, memory=512Mb, concurrency=20, min-instances=1 και max-instances=100. Όπως αναφέραμε και προηγουμένως, οι παράμετροι workers και threads ορίζονται στο Dockerfile οπότε θα πρέπει να δημιουργηθούν και τα αντίστοιχα images. Συγκεκριμένα:



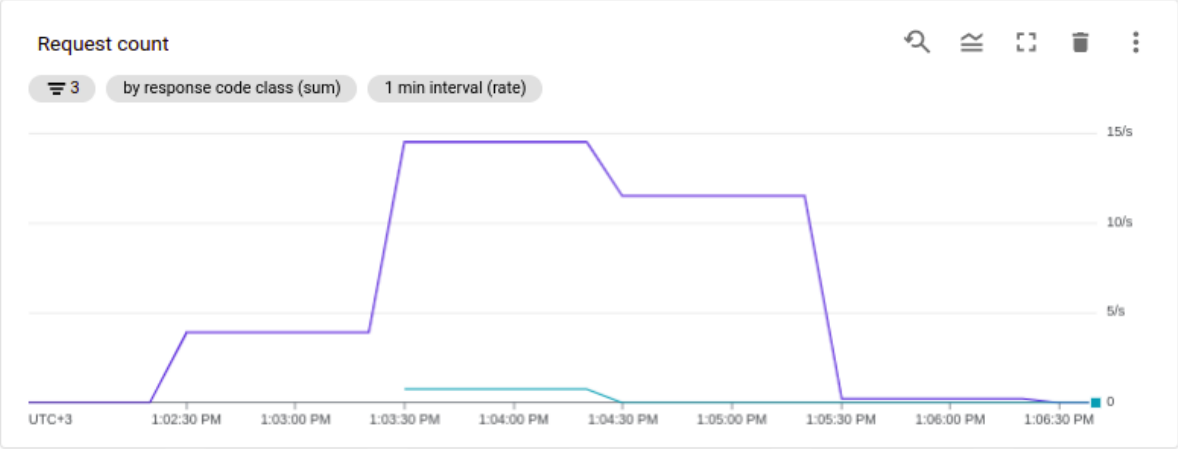
Εικόνα 16: Containers versioning

Εδώ βλέπουμε το GCP Container Registry με τα διάφορα images που φτιάξαμε και για να τα ξεχωρίζουμε ορίσαμε το Tag με βάση το workers και το threads. Δηλαδή το image με tag 4.1 σημαίνει workers=4 και threads=1 κ.ο.κ.

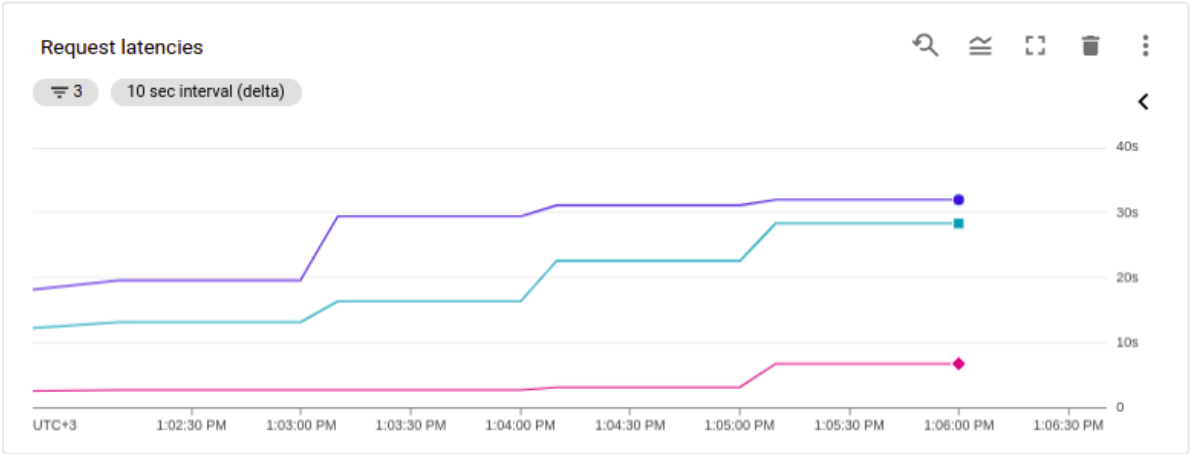
2.2.6.2 W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

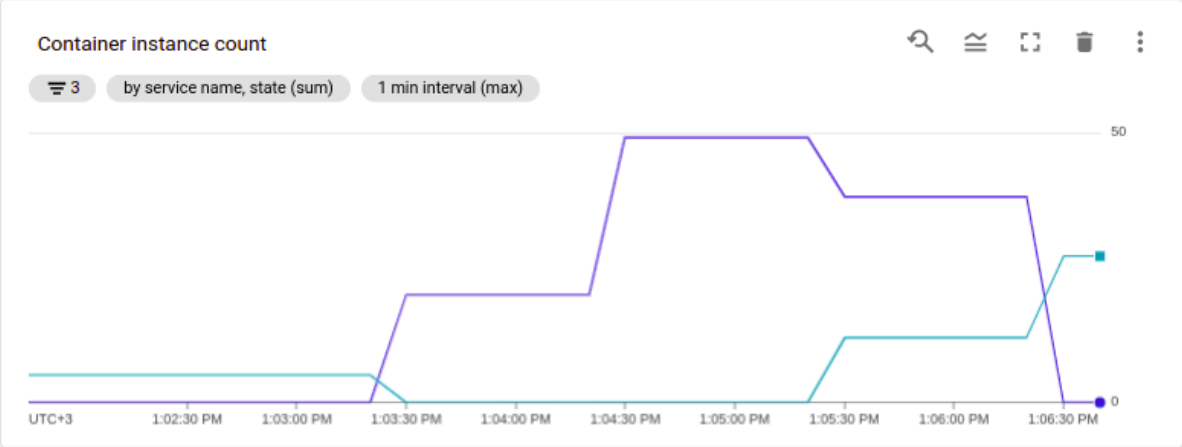
Το Dashboard από το GCP Monitoring:



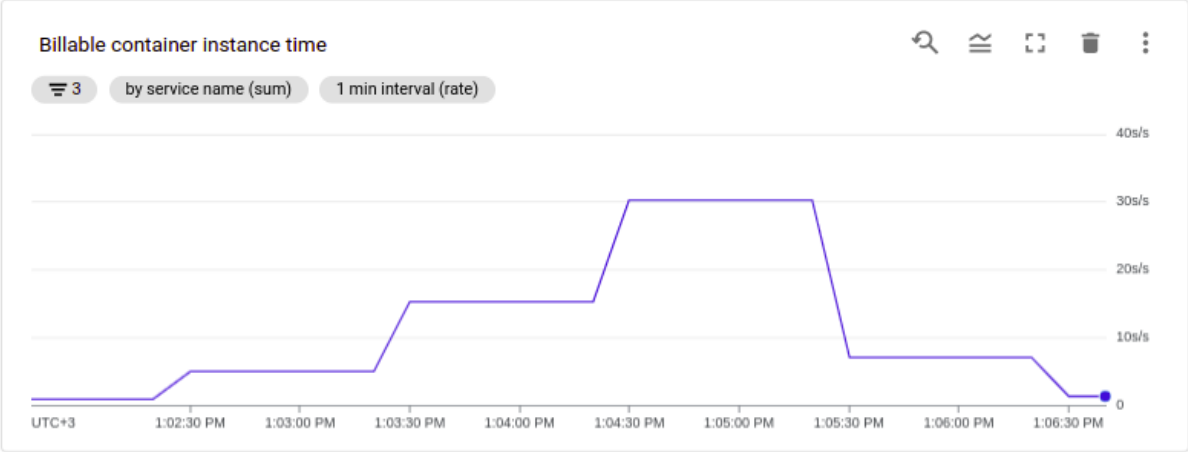
Εικόνα 17: Request count - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100



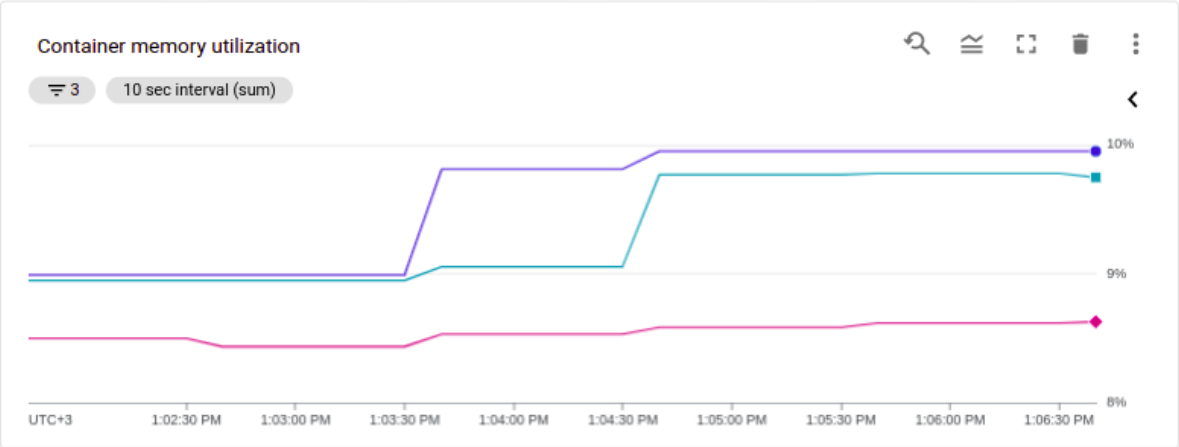
Εικόνα 18: Request latencies- W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100



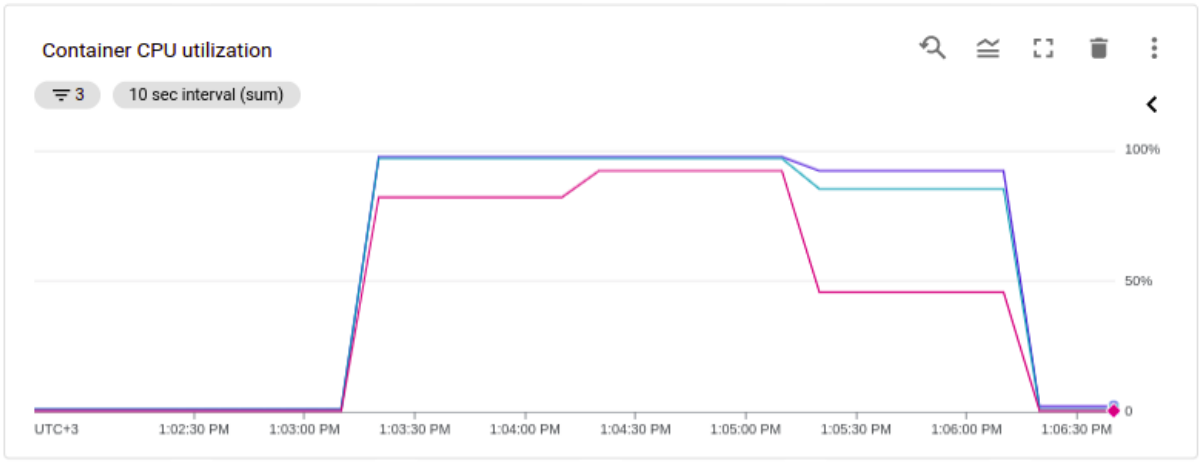
Εικόνα 19: Container instance count - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100



Εικόνα 20: Billable container instance time - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

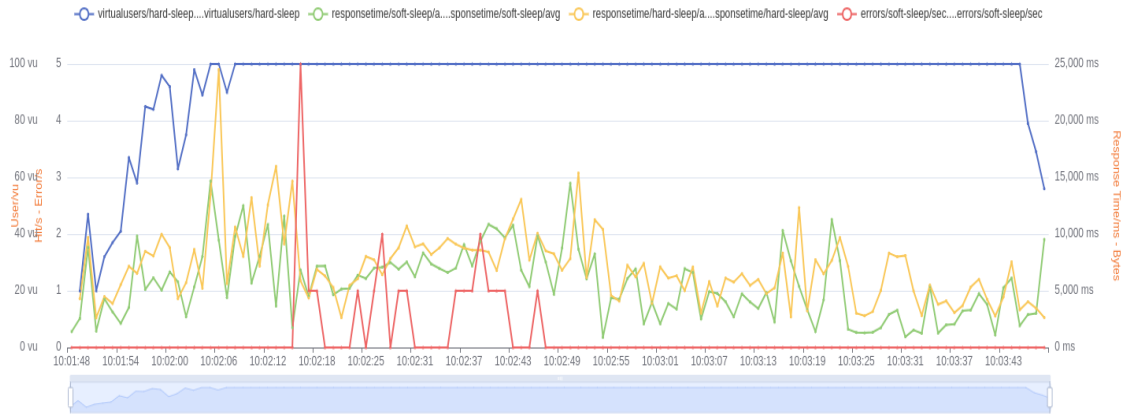


Εικόνα 21: Container memory utilization- W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100



Εικόνα 22: Container CPU utilization - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

To Report από το Loadium:



Εικόνα 23: Loadium average response graph - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

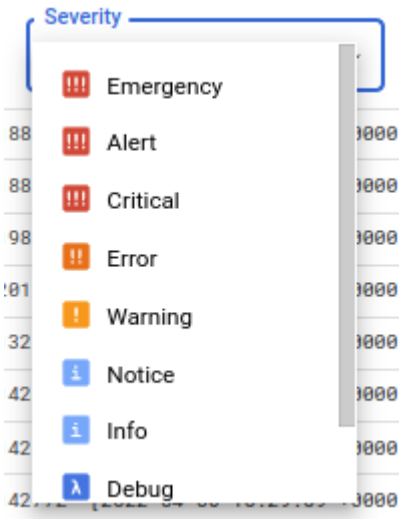
Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	942	6.83	5.13
"hard-sleep"	905	6.67	6.77

Εικόνα 24: Loadium total metrics - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	920
soft-sleep	429	Too Many Requests	22
hard-sleep	200	OK	884
hard-sleep	429	Too Many Requests	21

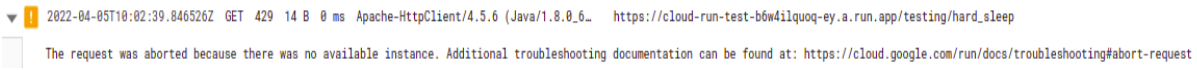
Εικόνα 25: Loadium response codes- W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

Αρχικά η πρώτη παρατήρηση είναι πως έχουμε κάποια errors με τίτλο “Too many requests”. Το Cloud Run διαθέτει ειδικό tab για Logs για κάθε διαφορετικό service το οποίο μας επιτρέπει να φιλτράρουμε όλα τα logs που παράγονται ανάλογα με το Severity:



Εικόνα 26: Cloud Run logs severity filter - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 100

Οπότε είναι πολύ εύκολο να εντοπίσουμε γρήγορα το πρόβλημα που είναι το εξής:

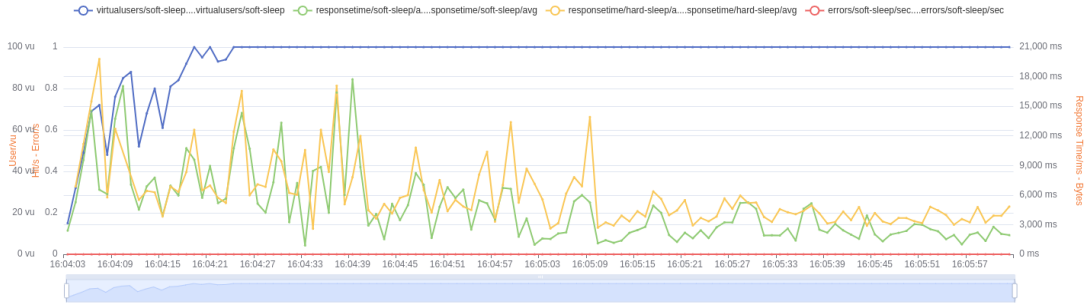


Συνεπώς, για να λύσουμε αυτό το πρόβλημα αρκεί να αυξήσουμε τα max-instances.

2.2.6.3 W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	***1000***	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Το Report από το Loadium:



Εικόνα 27: Loadium average response graph - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

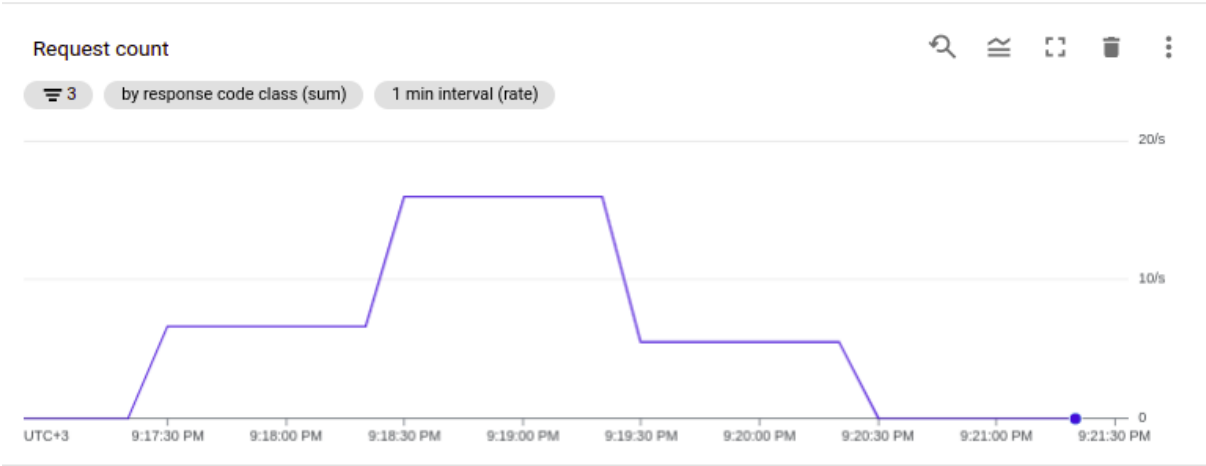
Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	1210	8.89	3.81
"hard-sleep"	1170	8.80	5.21

Εικόνα 28: Loadium total metrics - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

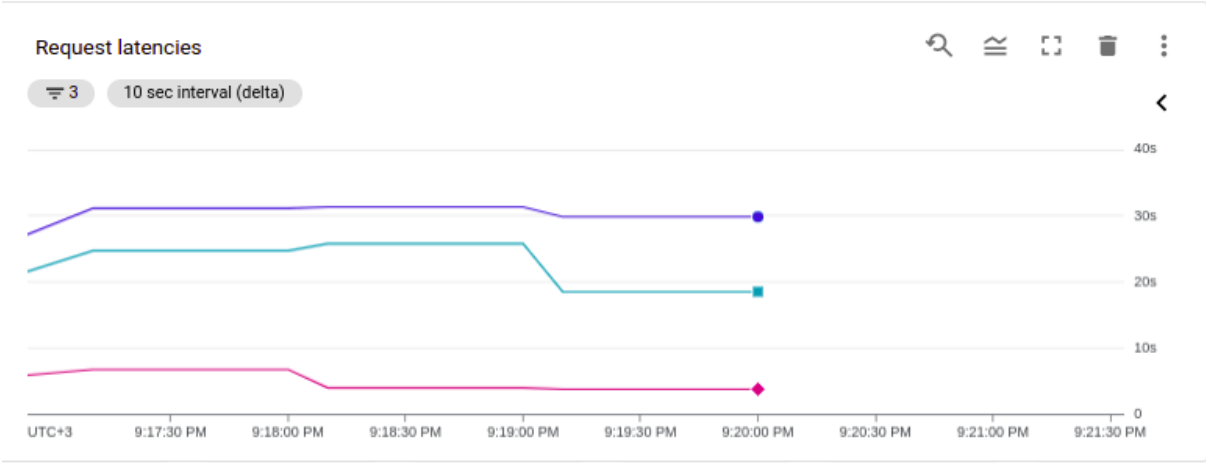
Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	1210
hard-sleep	200	OK	1170

Εικόνα 29: Loadium response codes - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

Το Dashboard από το GCP Monitoring:

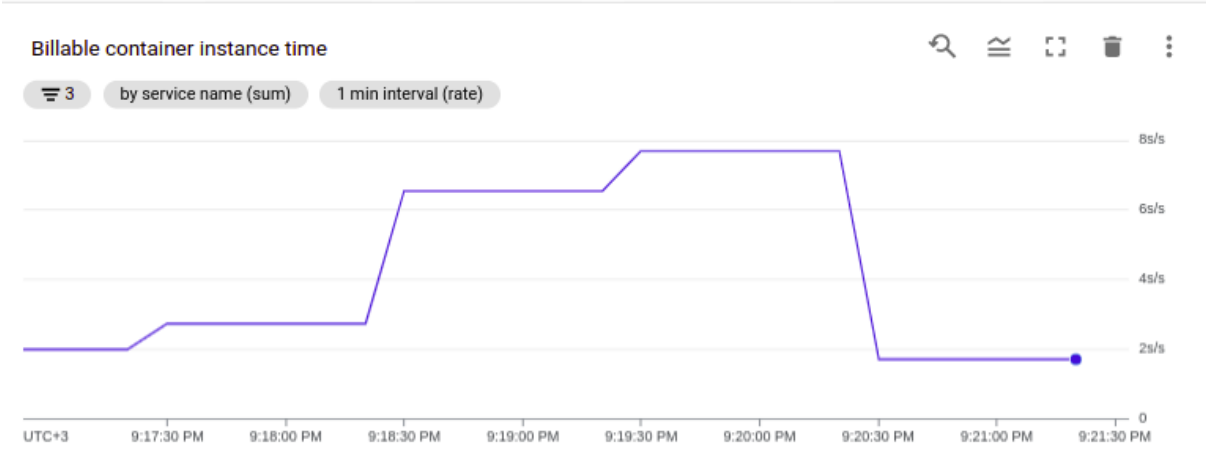


Εικόνα 30: Request count - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

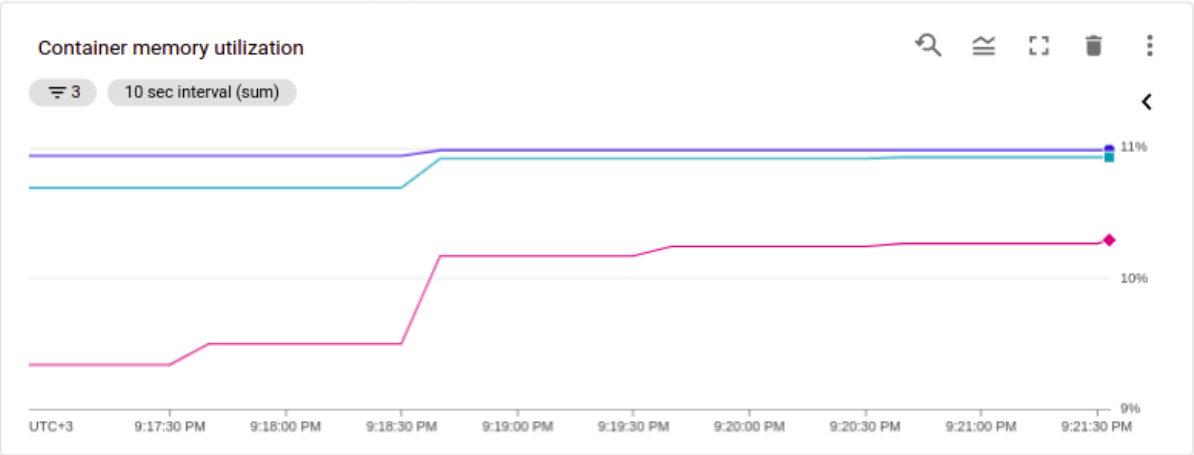


Εικόνα 31: Request latencies - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

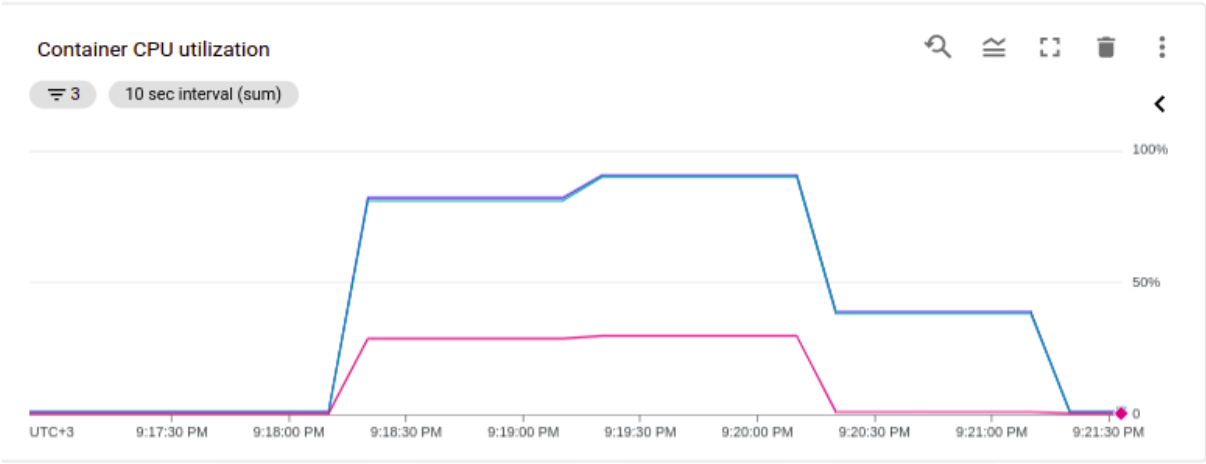
Εικόνα 32: Container instance count - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 33: Billable container instance time - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 34: Container memory utilization - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 35: Container CPU utilization - W 1, T 1, CPU 1, M 1024, CONC 20, MIN 1, MAX 1000

Παρατηρήσεις:

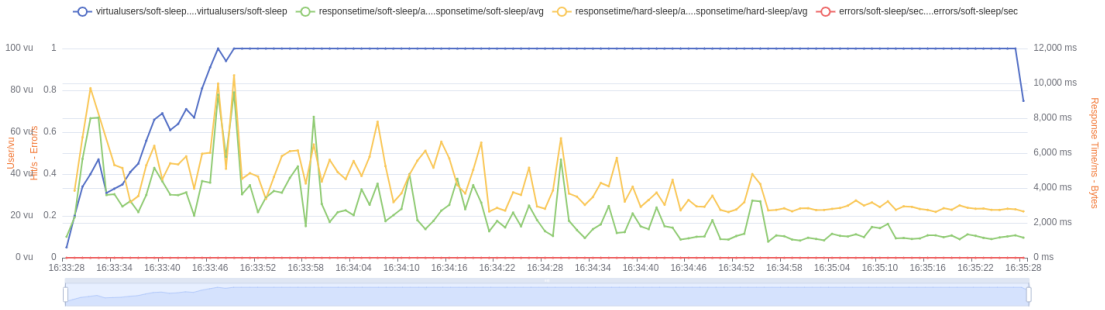
- Όπως περιμέναμε, τα errors εξαλείφθηκαν οπότε τώρα ας προσπαθήσουμε να αυξήσουμε το performance μετρώντας το Average Response Time / Sec που φαίνεται στο Loadium report.

2.2.6.4 W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Πρώτη προσπάθεια γίνεται αυξάνοντας τα CPU cores και όπως αναφέραμε σε multi-core environments καλό είναι να αυξάνουμε αναλόγως και τους workers.

Το Report από το Loadium:



Εικόνα 36: Loadium average response graph - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000

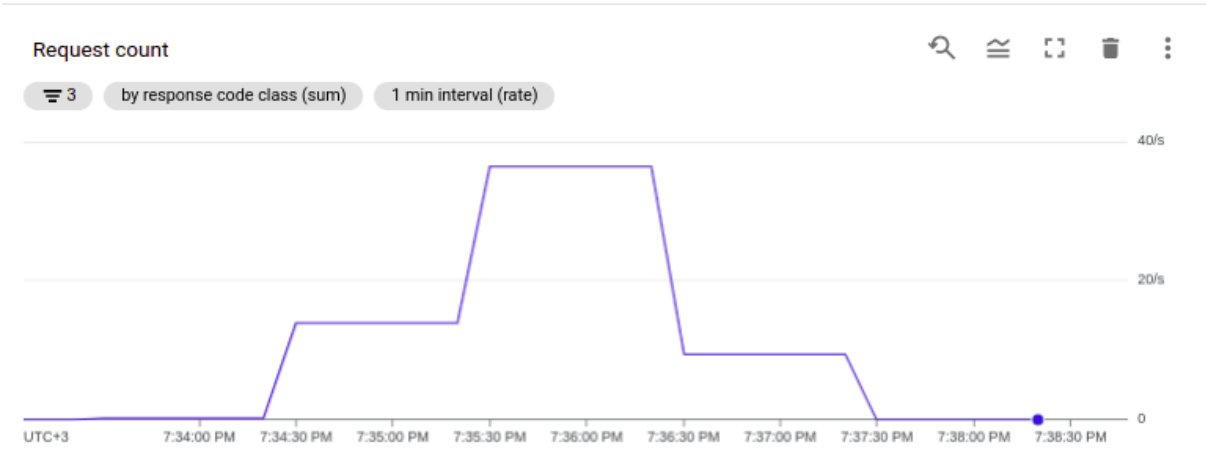
Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	1803	14.76	2.12
"hard-sleep"	1777	14.52	3.73

Εικόνα 37: Loadium total metrics - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000

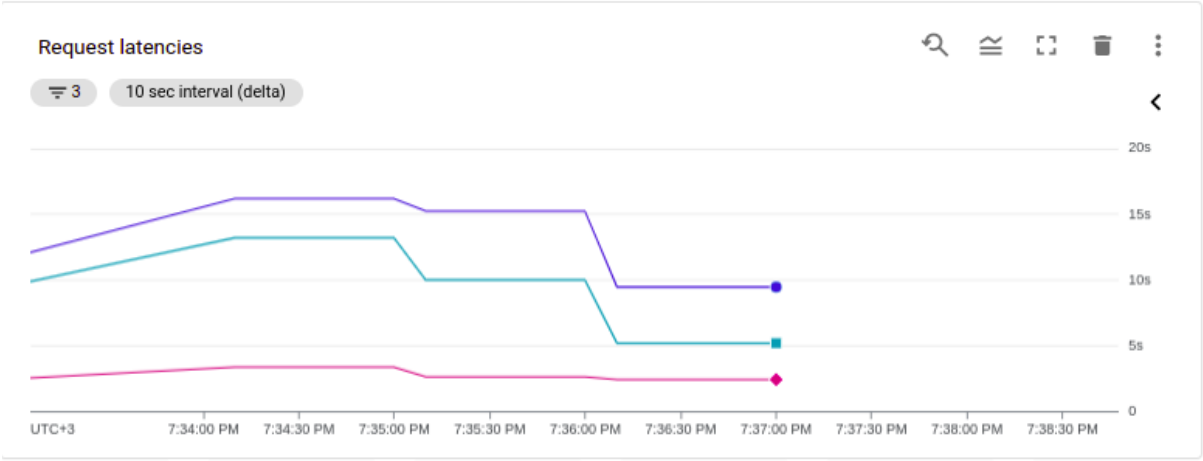
Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	1803
hard-sleep	200	OK	1777

Εικόνα 38: Loadium response codes - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000

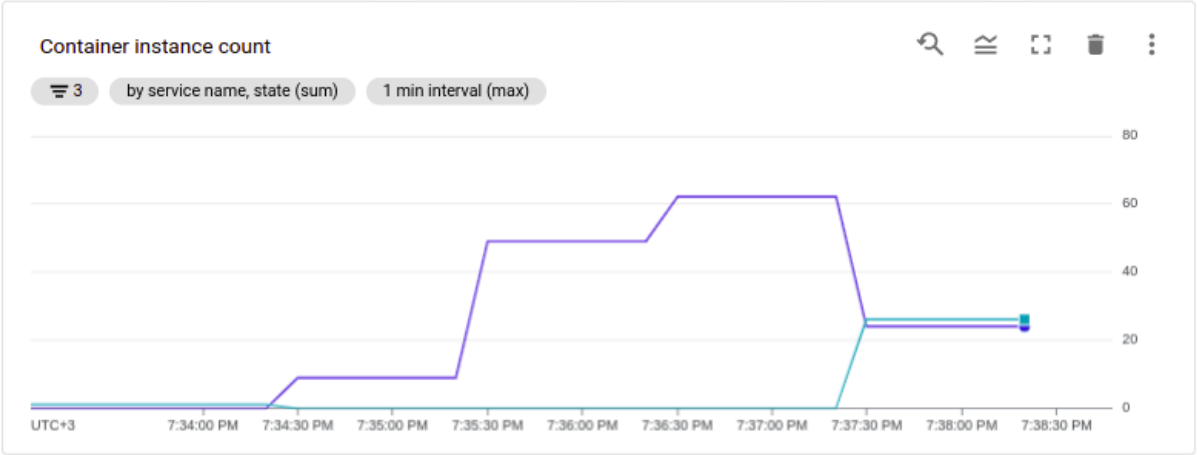
Το Dashboard από το GCP Monitoring:



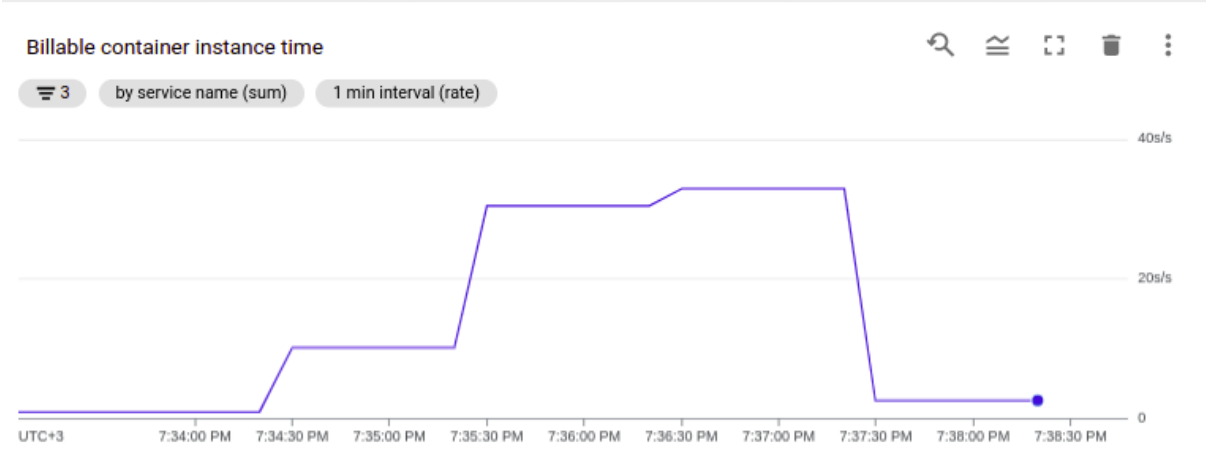
Εικόνα 39: Request count - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000



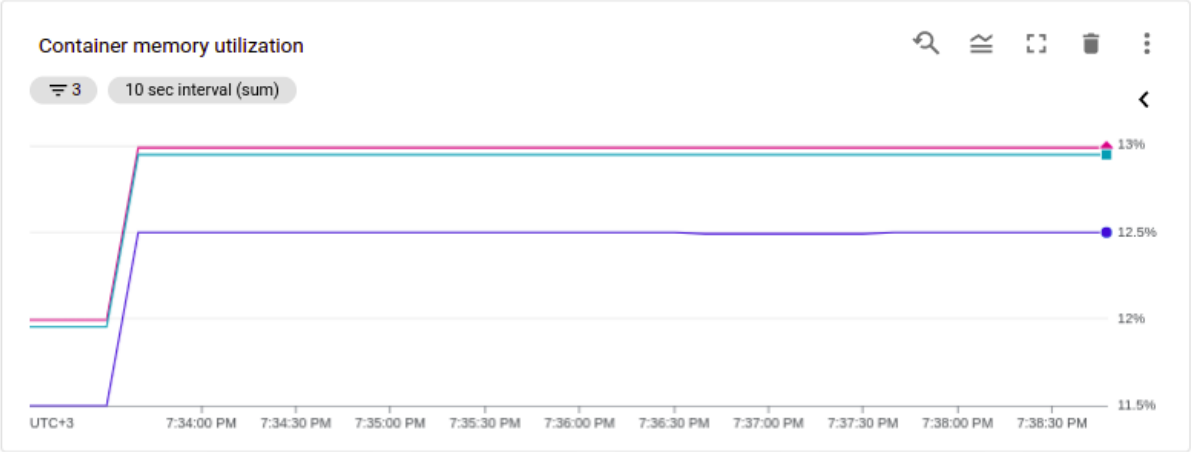
Εικόνα 40: Request latencies - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000



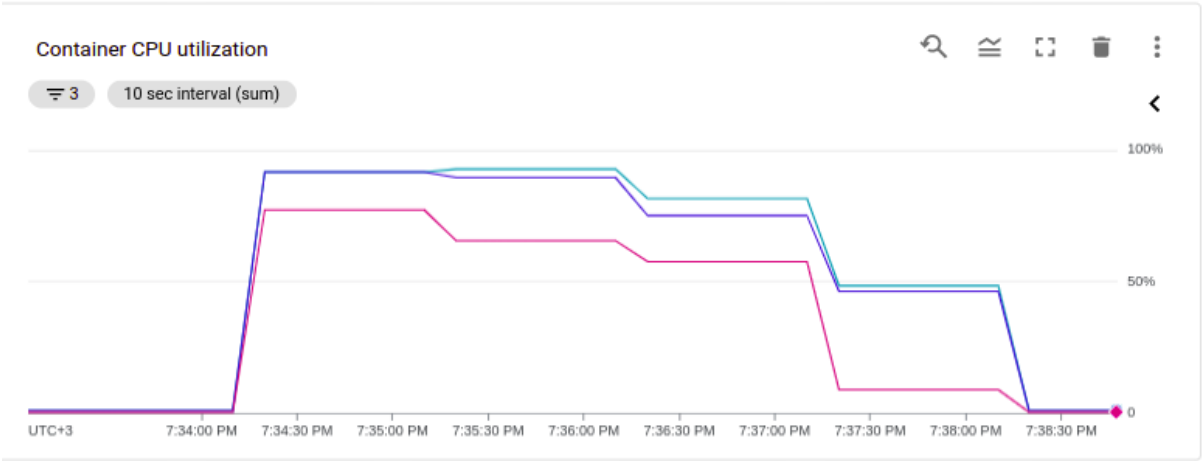
Εικόνα 41: Container instance count - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 42: Billable container instance time - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 43: Container memory utilization - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000



Εικόνα 44: Container CPU utilization - W 2, T 1, CPU 2, M 1024, CONC 20, MIN 1, MAX 1000

Παρατηρήσεις:

- Συνεχίζουμε να έχουμε μηδενικά errors.
- Το CPU utilization μειώθηκε, ενώ στις προηγούμενες περιπτώσεις ήταν οριακά 100%.
- Μειώθηκαν τα active instances καθώς ένα container με 2 workers επεξεργάζεται πιο εύκολα πολλαπλά requests σε σχέση με εκείνο που έχει 1 worker. Σε αυτό το σημείο να πούμε πως το Cloud Run κατά την διάρκεια εισερχόμενων requests γνωρίζει ακριβώς την κατάσταση (CPU, RAM κ.ο.κ.) όλων των active instances οπότε εκείνο με βάση αυτά αποφασίζει εάν θα φτιάξει νέα instances ή θα στείλει παραπάνω από 1 request στο ίδιο instance, σεβόμενο πάντα την παράμετρο concurrency. Οπότε ναι μεν το concurrency να έμεινε σταθερό στο 20 αλλά ο αριθμός των instances επηρεάζεται και από άλλες παραμέτρους.
- Παρατηρούμε ότι αυξήθηκαν τα total hits και το Request Per Second πλησίασε το 40. Αυτό συνέβη εξαιτίας του Loadium, καθώς εμείς βάλαμε ως περιορισμό το test να

διαρκεί max 2 λεπτά, οπότε εφόσον το σύστημα απέκτησε μεγαλύτερο throughput τότε και το Loadium μπόρεσε να κάνει περισσότερα requests μέσα σε 2 λεπτά.

- Μειώθηκε το average time response. Βελτιώθηκε το performance.

2.2.6.5 W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	***80***	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Εδώ θα αυξήσουμε το concurrency από 20 σε 80.

Το Report από το Loadium:

Εικόνα 45: Loadium average response graph - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

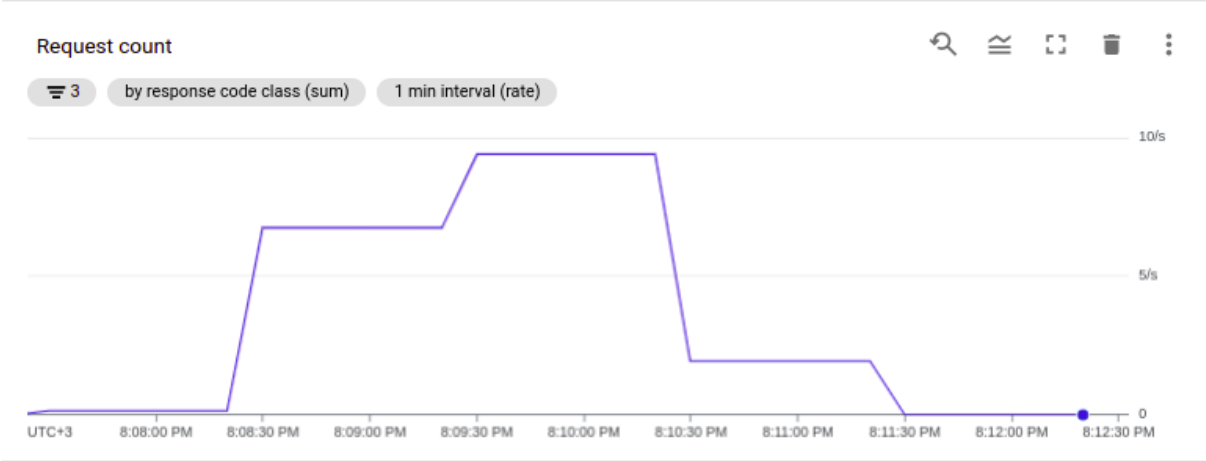
Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	563	3.58	9.98
"hard-sleep"	530	3.41	11.35

Εικόνα 46: Loadium total metrics - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

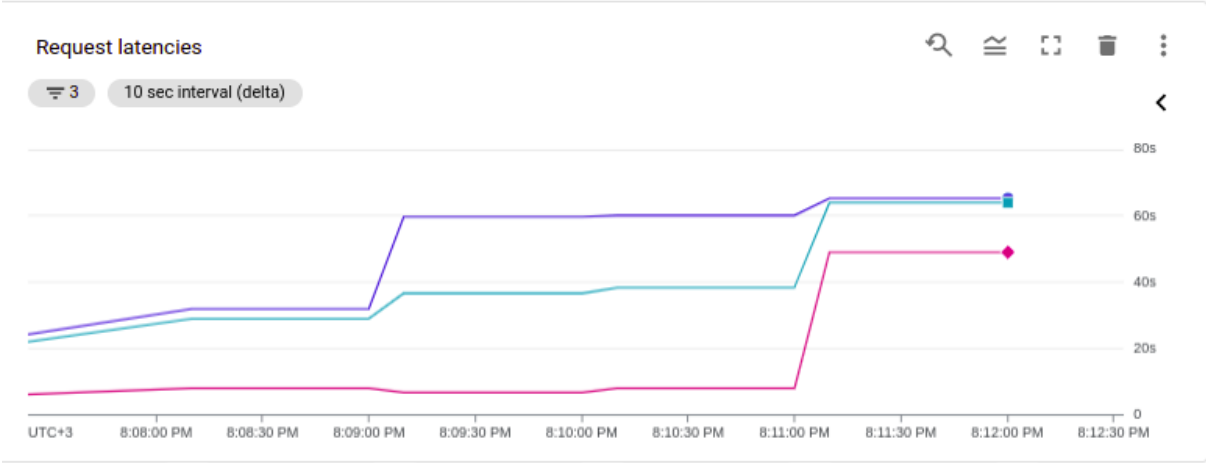
Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	563
hard-sleep	200	OK	530

Εικόνα 47: Loadium response codes - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

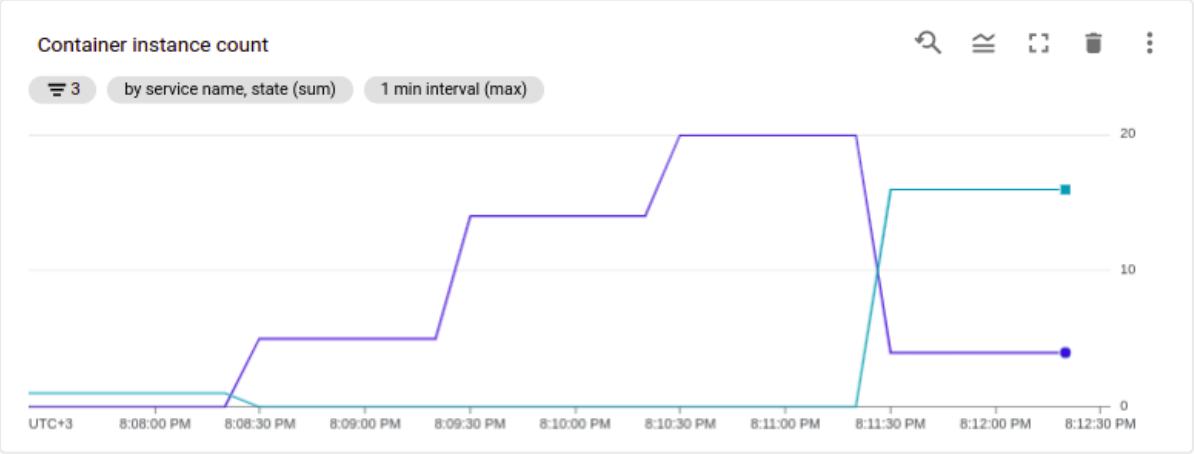
Το Dashboard από το GCP Monitoring:



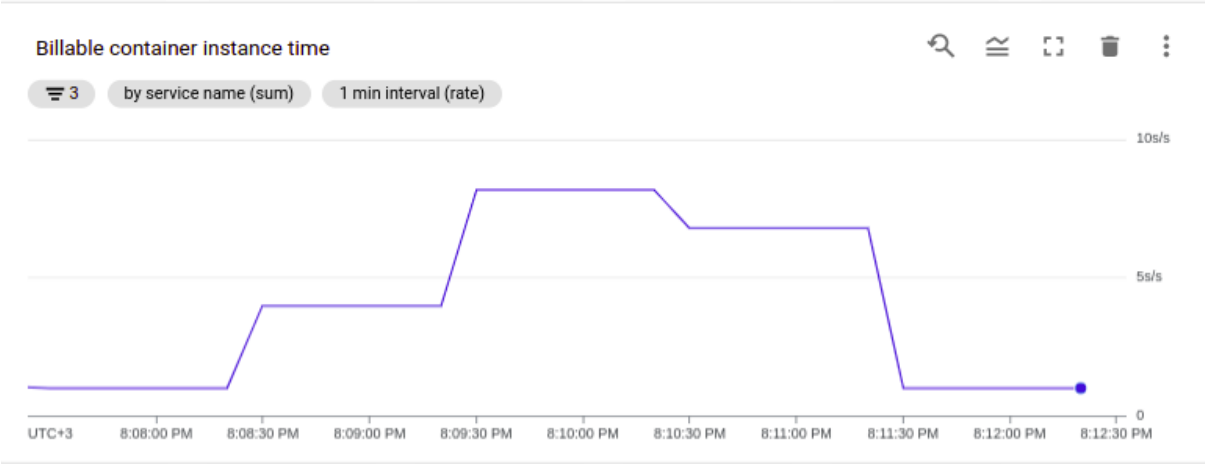
Εικόνα 48: Request count - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 49: Request latencies - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

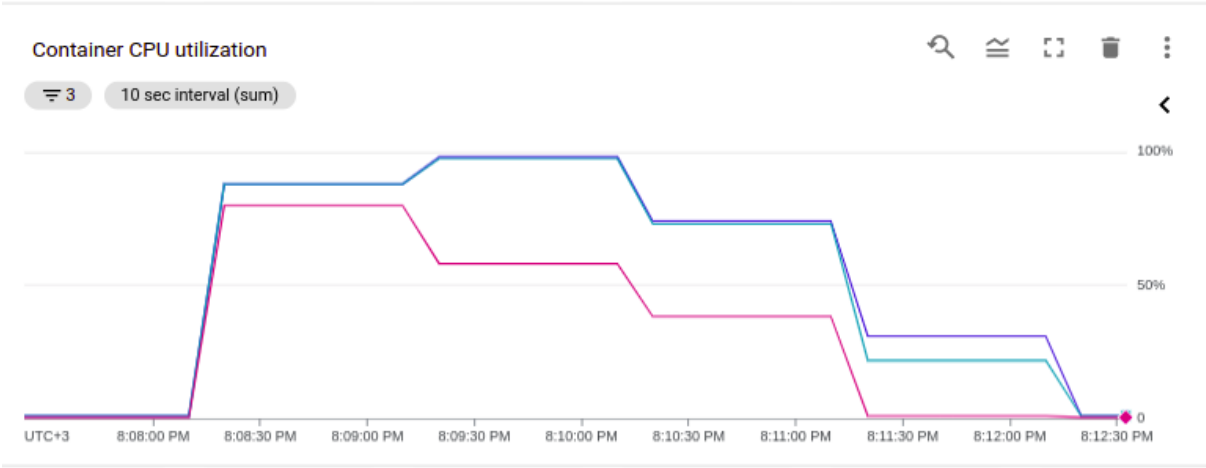


Εικόνα 50: Container instance count - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 51: Billable container instance time - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Εικόνα 52: Container memory utilization - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 53: Container CPU utilization - W 2, T 1, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Παρατηρήσεις:

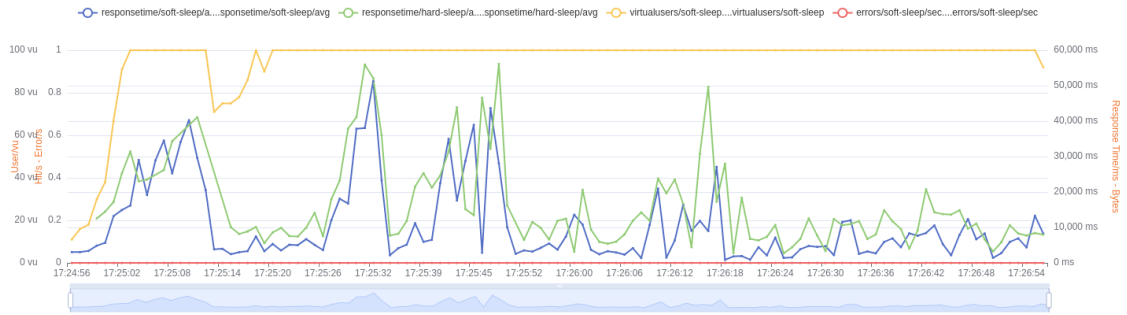
- Τα active instances μειώθηκαν. Συγκεκριμένα υπό-τετραπλασιάστηκαν, όπως ήταν αναμενόμενο καθώς αυξήθηκε το concurrency από 20 σε 80.
- Επίσης αυξήθηκε το request latency καθώς το ένα container διαχειρίζεται περισσότερα requests από ότι πριν.

2.2.6.6 W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	***4***	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Εδώ θα αυξήσουμε τα threads από 1 σε 4.

Το Report από το Loadium:



Εικόνα 54: Loadium average response graph - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	464	3.34	10.21
"hard-sleep"	430	3.11	15.18

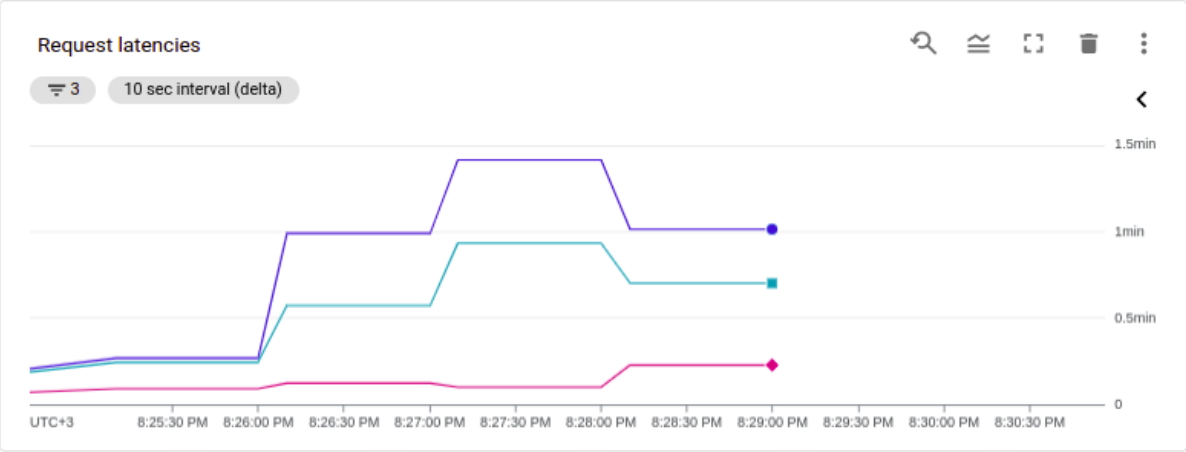
Εικόνα 55: Loadium total metrics - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Label	Response Code	Response Code Message	Number of Respons
soft-sleep	200	OK	464
hard-sleep	200	OK	430

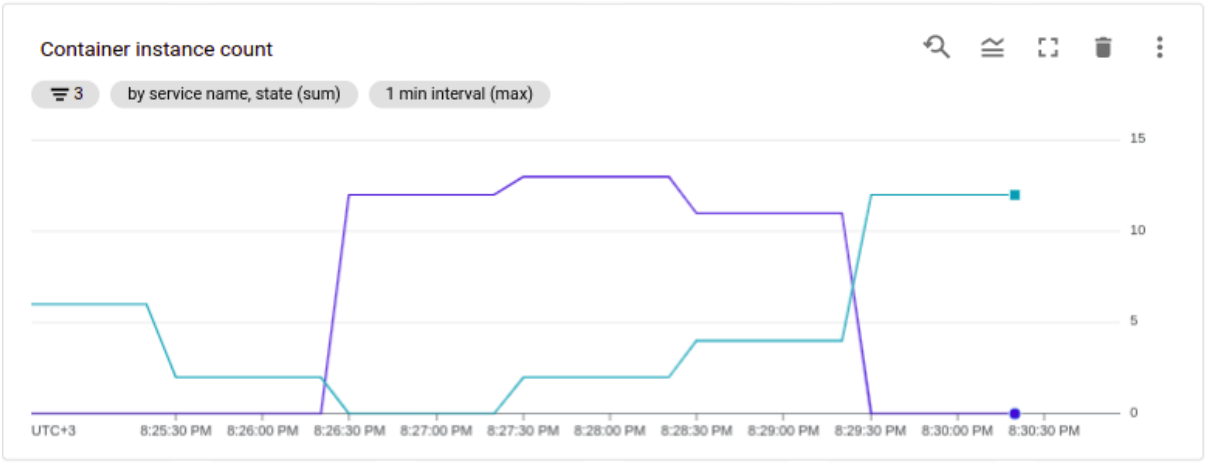
Εικόνα 56: Loadium response codes - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Το Dashboard από το GCP Monitoring:

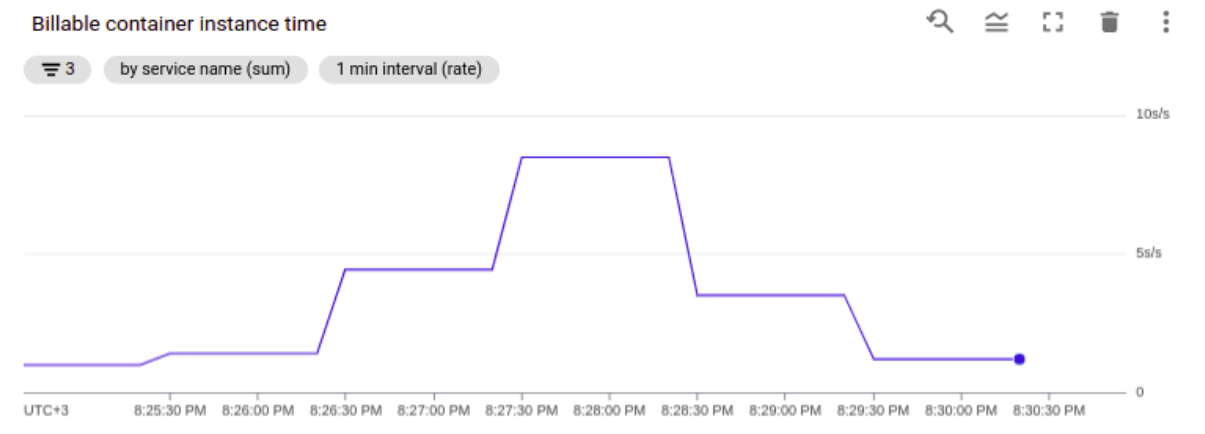
Εικόνα 57: Request count - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



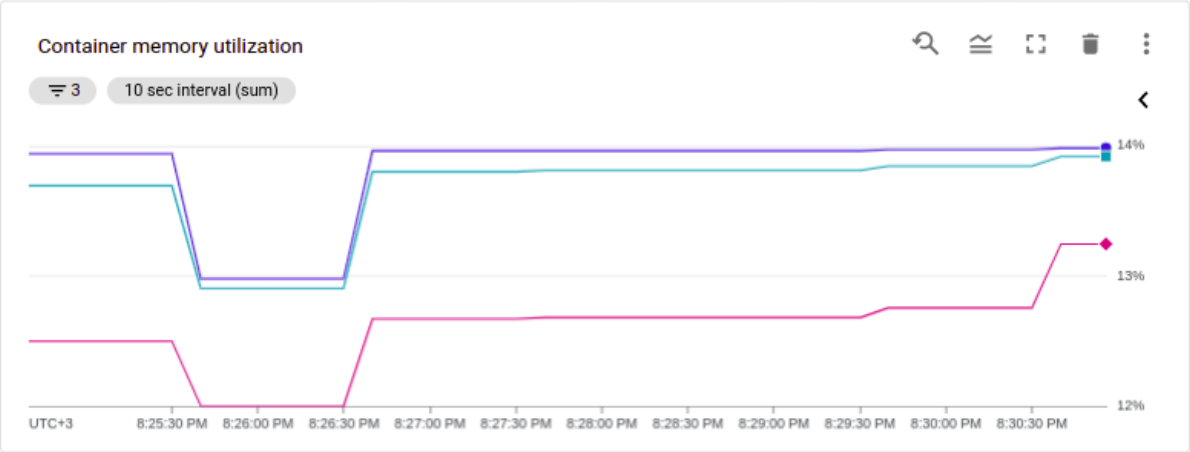
Εικόνα 58: Request latencies - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



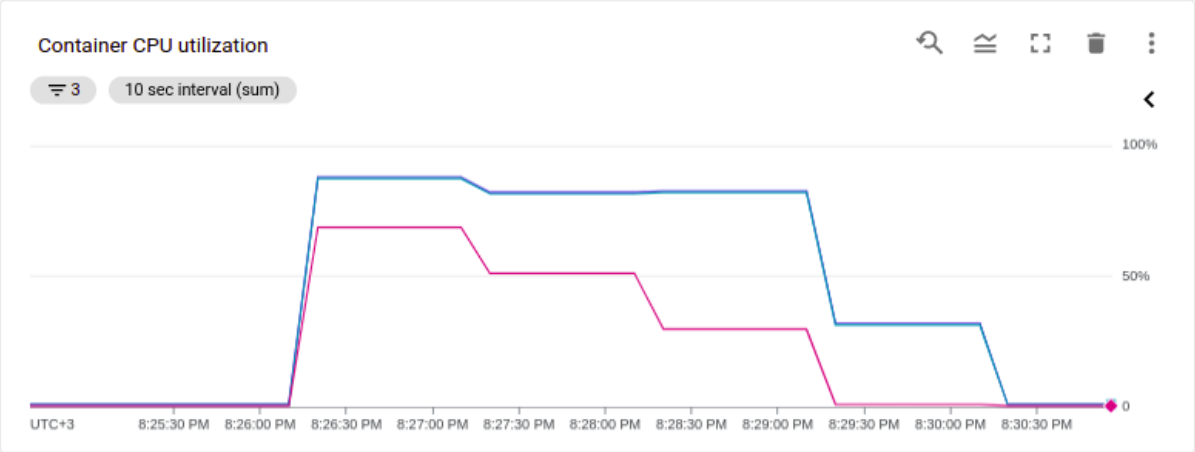
Εικόνα 59: Container instance count - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 60: Billable container instance time - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 61: Container memory utilization - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000



Εικόνα 62: Container CPU utilization - W 2, T 4, CPU 2, M 1024, CONC 80, MIN 1, MAX 1000

Αρχικά για να λειτουργήσει το multi-threading θα πρέπει να το υποστηρίζει η εφαρμογή μας και στην δική μας περίπτωση ισχύει:

```
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=int(os.environ.get("PORT")), threaded=True)
```

Εικόνα 63: Flask multi-threading

καθώς η παράμετρος threaded του Flask είναι True.

Παρατηρήσεις:

- Μειώθηκαν περισσότερο τα instances, όπως και ήταν αναμενόμενο.
- Μειώθηκε το Performance, καθώς αυξήθηκε το request latency για τους ίδιους λόγους και με πριν.

Προσοχή: Το multi-threading πρέπει να υποστηρίζεται και από την σωστή RAM, καθώς κινδυνεύει το σύστημα από 5XX λόγω μη επαρκούς run-time μνήμης. Για να μπορέσω να

αναπαράξω αυτό το error μείωσα την μνήμη από 1024 σε 512 και είχαμε το αναμενόμενο αποτέλεσμα με το εξής error:

Memory limit of 512M exceeded with 525M used. Consider increasing the memory limit

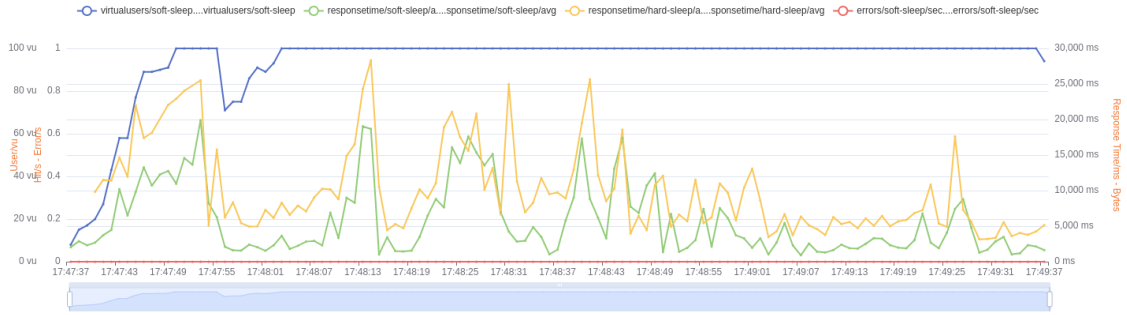
Επίσης μέσα από εδώ (είτε μέσα από το διάγραμμα του Memory Utilization) μπορούμε να εκτιμήσουμε την RAM που χρειάζεται το container μας, όπως εδώ 525 Mb, και να ορίσουμε την ελάχιστη δυνατή για να έχουμε cost-efficiency.

2.2.6.7 W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	***4***	4	***4***	***2048***	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Εδώ θα αυξήσουμε την CPU από 2 σε 4. Οπότε όπως έχουμε πει θα αυξήσουμε και τους workers και αναγκαστικά και την μνήμη, καθώς το σύστημα δεν επιτρέπει 4 CPU cores και 1 Gb RAM.

Το Report από το Loadium:



Εικόνα 64: Loadium average response graph - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	855	6.41	5.04
"hard-sleep"	815	6.19	7.99

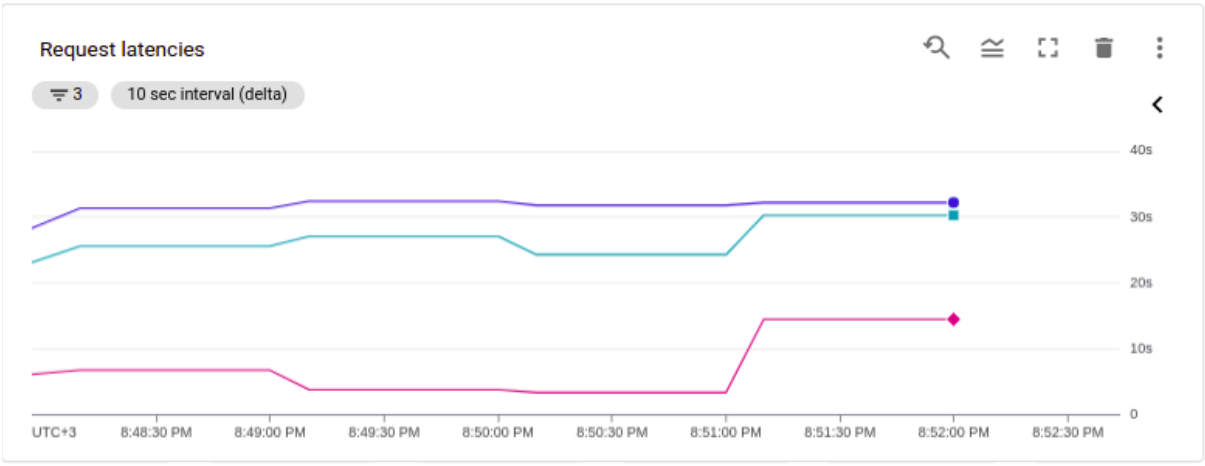
Εικόνα 65: Loadium total metrics - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	855
hard-sleep	200	OK	815

Εικόνα 66: Loadium response codes - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

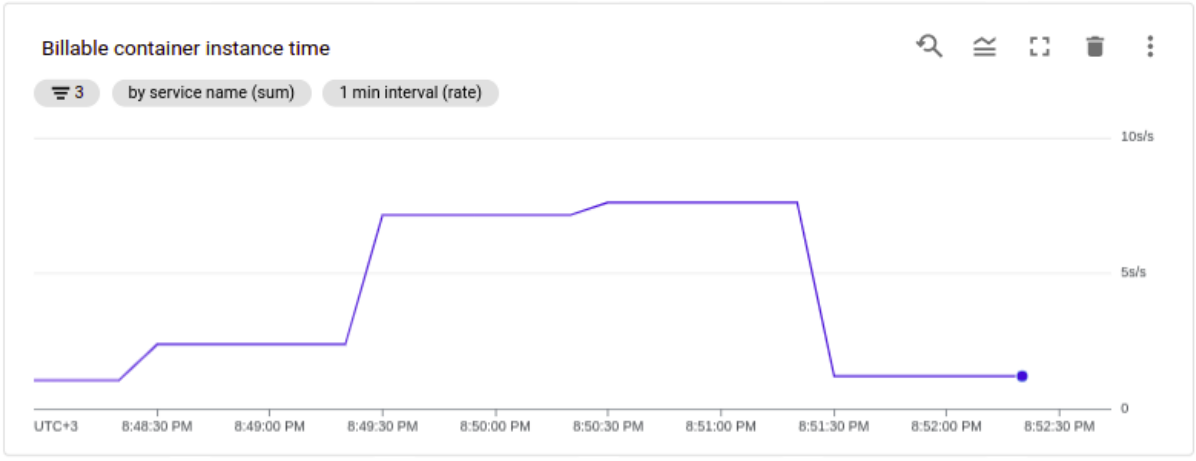
Το Dashboard από το GCP Monitoring:

Εικόνα 67: Request count - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

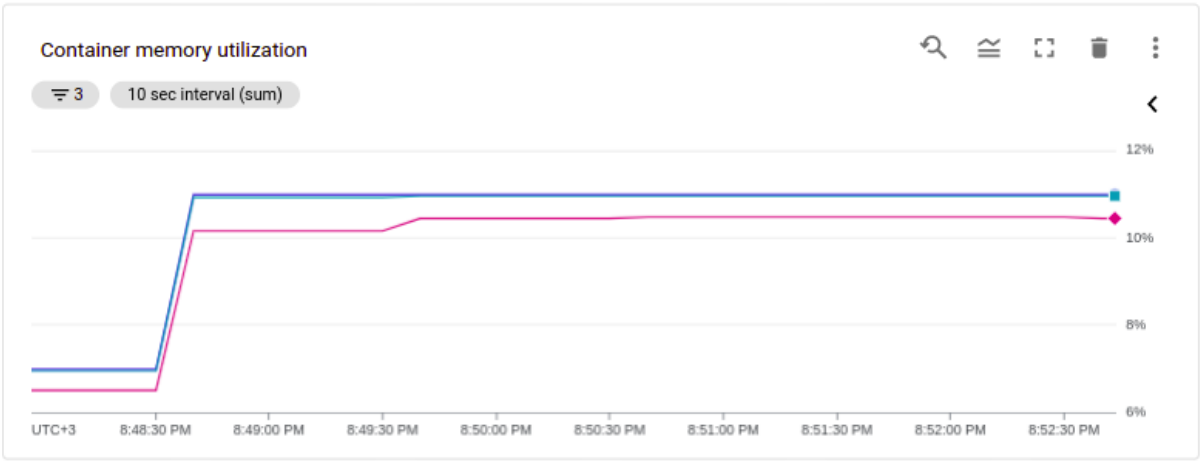


Εικόνα 68: Request latencies - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Εικόνα 69: Container instance count - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000



Εικόνα 70: Billable container instance count - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000



Εικόνα 71: Container memory utilization - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Εικόνα 72: Container CPU utilization - W 4, T 4, CPU 4, M 2048, CONC 80, MIN 1, MAX 1000

Παρατηρήσεις:

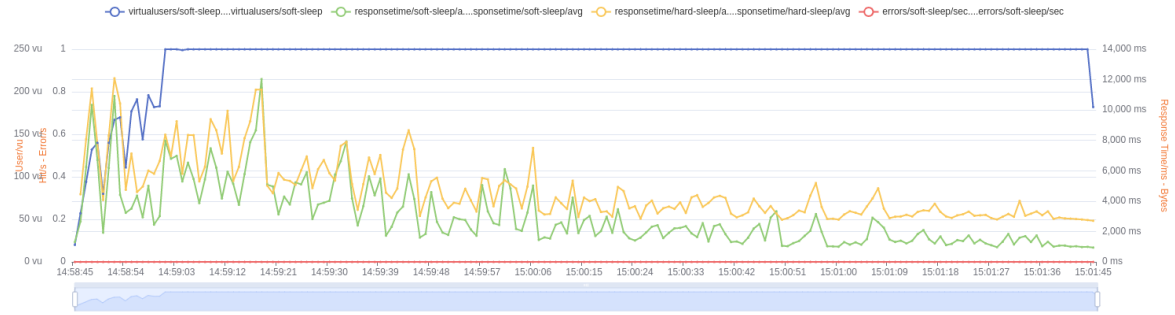
- Τα instances έμειναν περίπου ίδια καθώς δεν άλλαξε το concurrency ή το threads
- Μειώθηκε το CPU utilization όπως συμβαίνει κάθε φορά που αυξάνουμε τα cores.
- Βελτιώθηκε το performance.

2.2.6.8 W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	***1***	4	2048	***40***	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	90

Θα προσπαθήσουμε τώρα να βελτιώσουμε κι άλλο την επίδοση του συστήματος μειώνοντας το threads από 4 σε 1 και το concurrency από 80 σε 40. Επίσης από 100 total users θα προσθέσουμε 250 users στο simulation για να ελέγξουμε το scalability του συστήματος.

Το Report από το Loadium:



Εικόνα 73: Loadium average response graph - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	6230	33.67	2.44
"hard-sleep"	6154	33.02	4.08

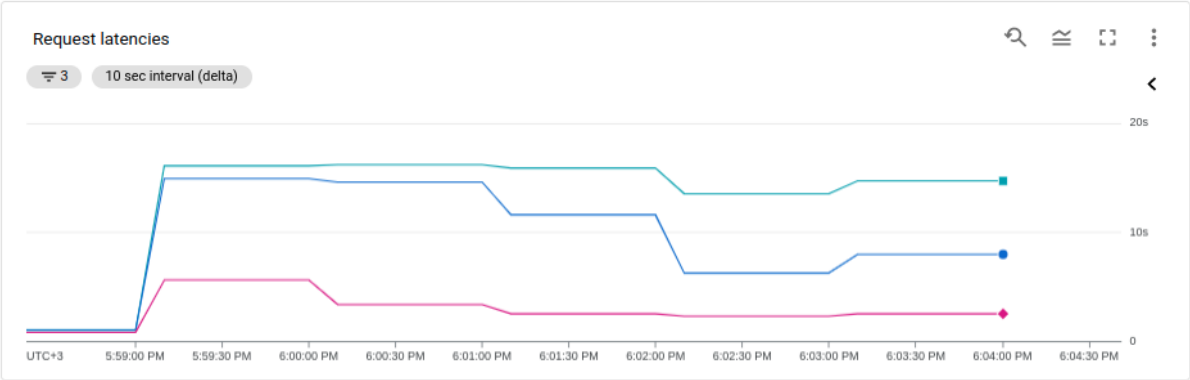
Εικόνα 74: Loadium total metrics - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	6230
hard-sleep	200	OK	6154

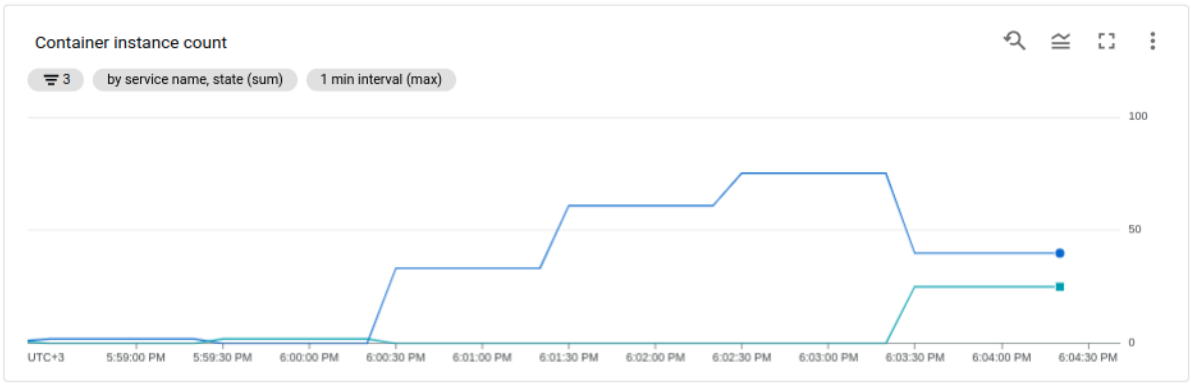
Εικόνα 75: Loadium response codes - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Το Dashboard από το GCP Monitoring:

Εικόνα 76: Request count - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

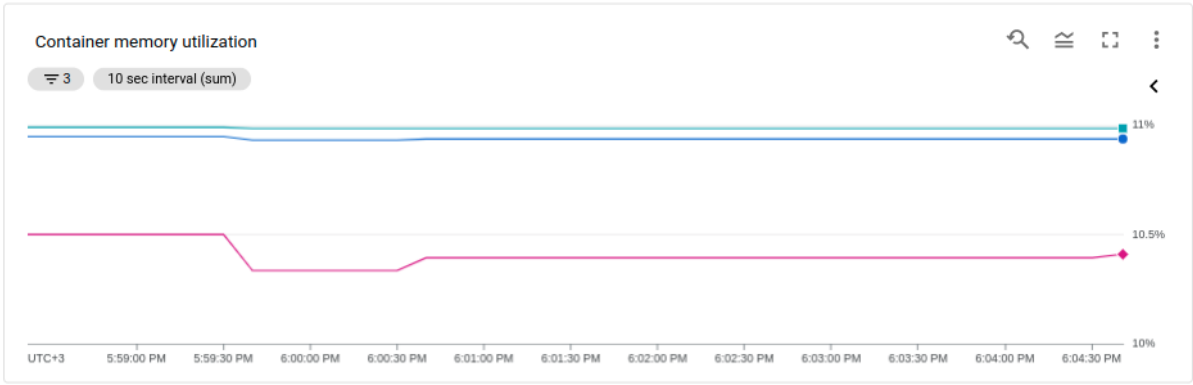


Εικόνα 77: Request latencies - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

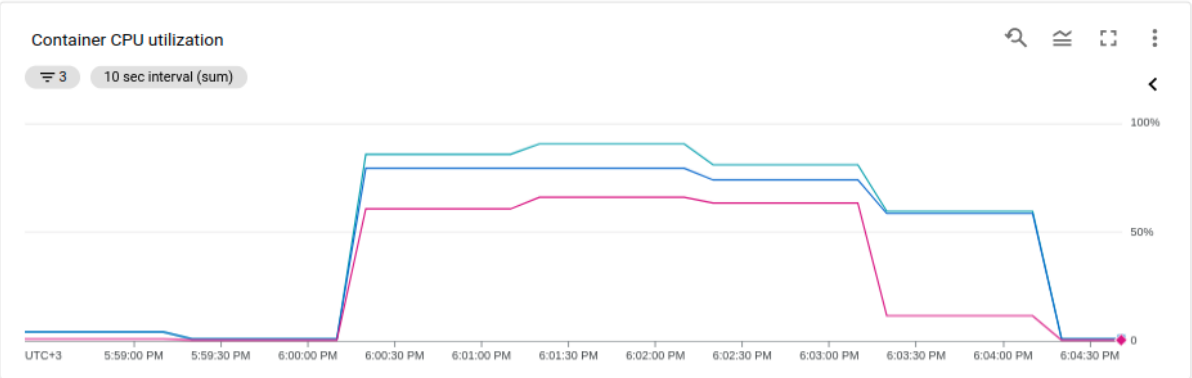


Εικόνα 78: Container instance count - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Εικόνα 79: Billable container instance time - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000



Εικόνα 80: Container memory utilization - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000



Εικόνα 81: Container CPU utilization - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Παρατηρήσεις:

- Το σύστημα μας έχει την καλύτερη επίδοση με αυτές τις παραμέτρους σε σχέση με όλες τις υπόλοιπες περιπτώσεις. Πλησιάσαμε το 100 RPS άρα αυξήθηκε το throughput και καταφέραμε να εξυπηρετήσουμε περίπου τα δεκαπλάσια requests στον ίδιο χρόνο έχοντας περισσότερους χρήστες.

- Απόδειξη του scalability του συστήματος καθώς χωρίς καμία παραπάνω ενέργεια από την πλευρά του developer εξυπηρετήθηκαν κανονικά 250 users

- Memory & CPU utilization είναι σε καλά επίπεδα.

- Βέβαια έχουμε αρκετά active instances, καθώς μειώσαμε και το threads και το concurrency.

2.2.6.9 W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000

Scenario	Workers	Threads	CPU (cores)	Memory (Mb)	Concurrency (requests)	Min-Instances	Max-Instances	Ramp-Up (seconds)
2.2.6.2	1	1	1	1024	20	1	100	30
2.2.6.3	1	1	1	1024	20	1	1000	30
2.2.6.4	2	1	2	1024	20	1	1000	30
2.2.6.5	2	1	2	1024	80	1	1000	30
2.2.6.6	2	4	2	1024	80	1	1000	30
2.2.6.7	4	4	4	2048	80	1	1000	30
2.2.6.8	4	1	4	2048	40	1	1000	30
2.2.6.9	4	1	4	2048	40	1	1000	***90***

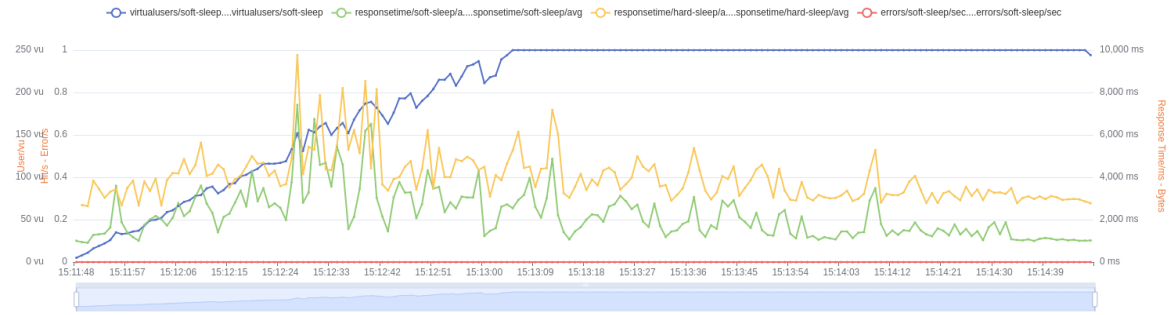
Παρατηρούμε πως το average response time επηρεάζεται πολύ από τα πρώτα δευτερόλεπτα του simulation καθώς εισέρχονται όλοι οι χρήστες μέσα σε 30 δευτερόλεπτα ενώ το σύστημα είναι προηγουμένως σε πλήρη αδράνεια. Οπότε υπάρχουν μερικά πρώτα requests με πολύ μεγάλο latency (τα λεγόμενα Cold Starts) που αυξάνουν τον μέσο όρο. Για να λυθεί αυτό το πρόβλημα υπάρχουν 2 τρόποι:

- Αυξάνουμε το minimum-instances για να μειώσουμε τα Cold Starts.

- Αυξάνουμε το Ramp-Up time του simulation, δηλαδή να μπουν όλοι οι χρήστες σε 90 seconds και όχι σε 30, έτσι ώστε να προλάβει να “ζεσταθεί” το σύστημα.

Επιλέγοντας την δεύτερη επιλογή έχουμε τα εξής αποτελέσματα.

Το Report από το Loadium:



Εικόνα 82: Loadium average response graph - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"soft-sleep"	5743	31.66	1.98
"hard-sleep"	5663	31.00	3.77

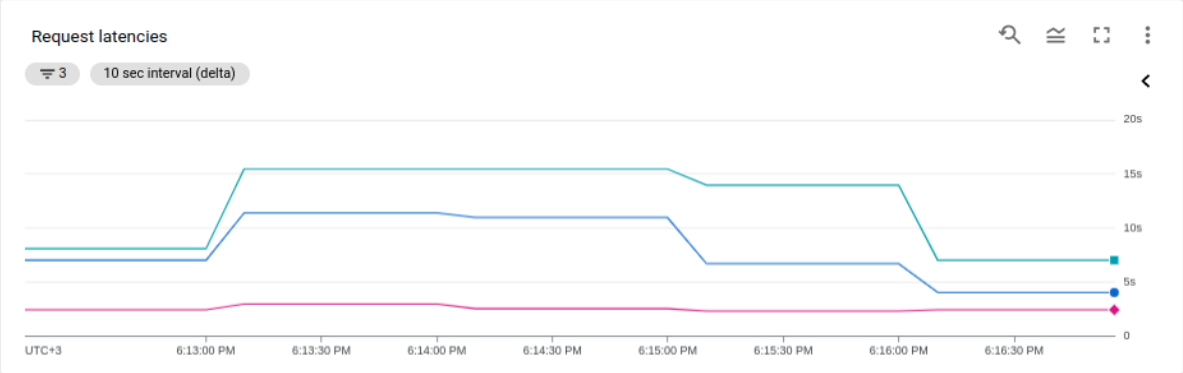
Εικόνα 83: Loadium total metrics - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

Label	Response Code	Response Code Message	Number of Response
soft-sleep	200	OK	5743
hard-sleep	200	OK	5663

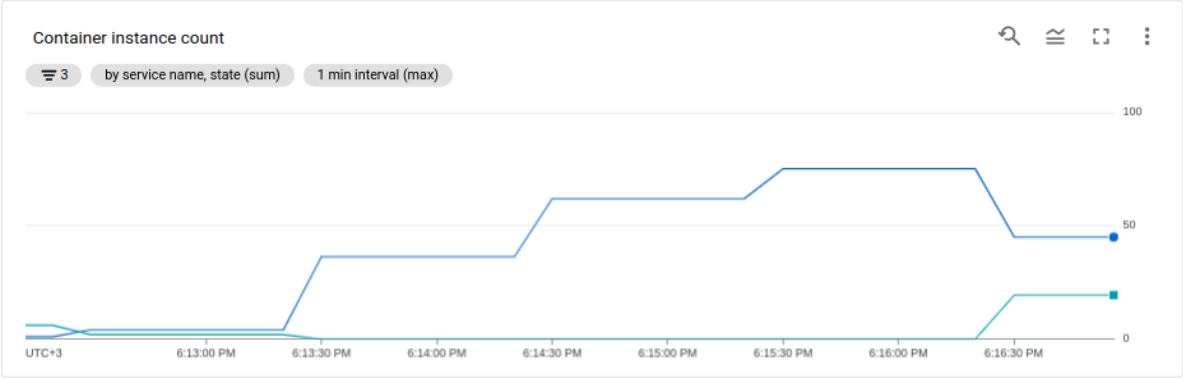
Εικόνα 84: Loadium response codes - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

Το Dashboard από το GCP Monitoring:

Εικόνα 85: Request count - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

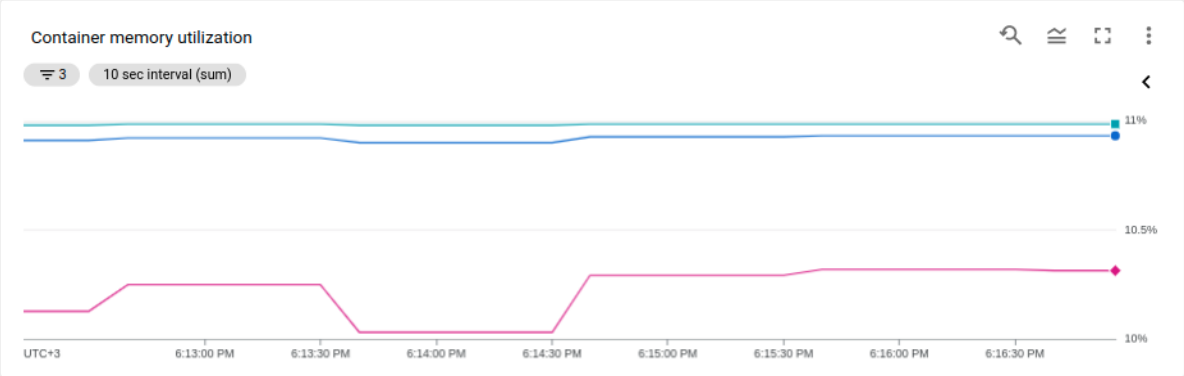


Εικόνα 86: Request latencies - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

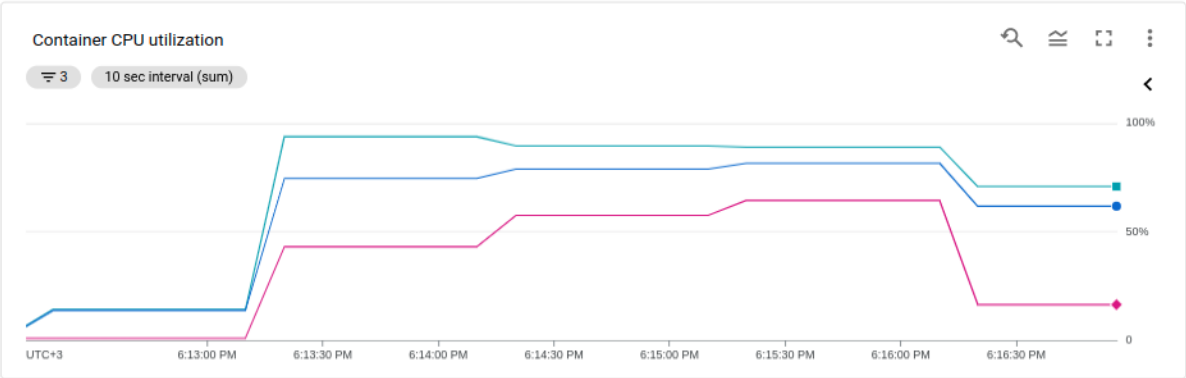


Εικόνα 87: Container instance count - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

Εικόνα 88: Billable container instance time - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)



Εικόνα 89: Container memory utilization - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)



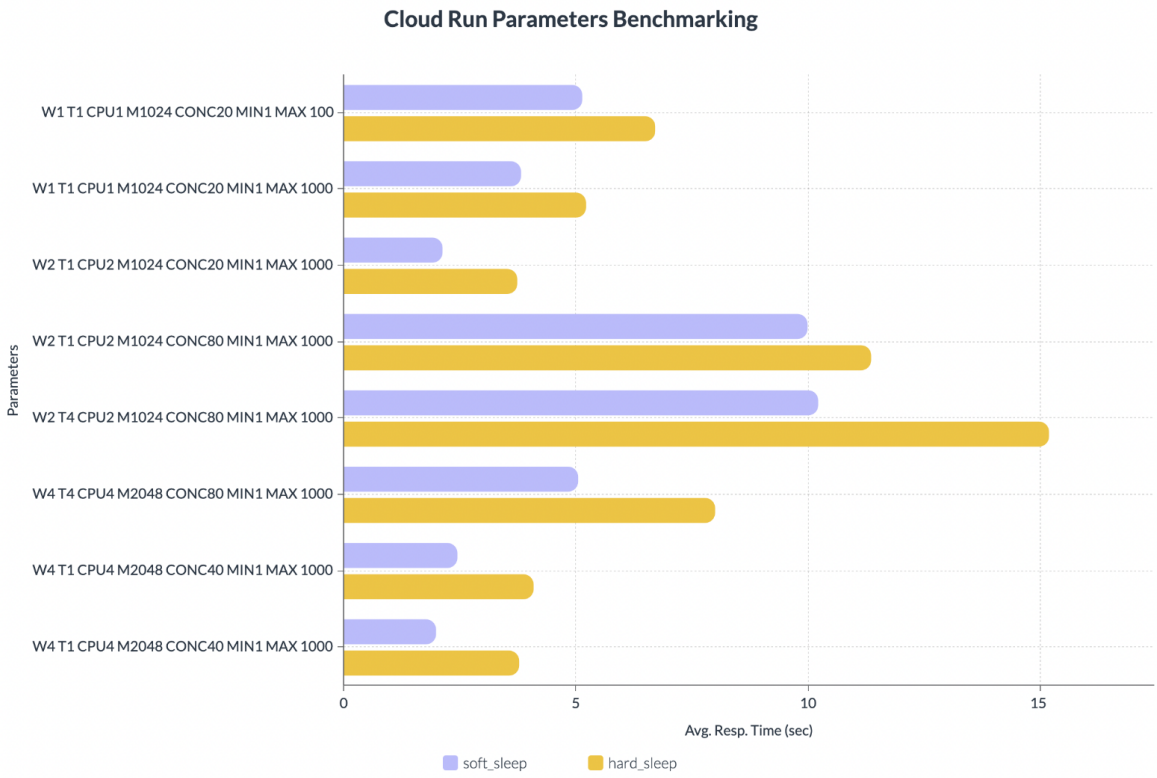
Εικόνα 90: Container CPU utilization - W 4, T 1, CPU 4, M 2048, CONC 40, MIN 1, MAX 1000 (Ramp-Up)

Παρατηρήσεις:

- Όντως όπως βλέπουμε το average response time μειώθηκε. Βέβαια πάλι δεν είναι πλήρως αντιπροσωπευτικό, καθώς παρατηρούμε ότι μόλις εισέλθουν όλοι οι χρήστες το σύστημα σταθεροποιείται και οι χρόνοι για το soft_sleep είναι ~1 sec και για το hard_sleep είναι ~2.8 sec, αριθμοί που πλησιάζουν localhost επιδόσεις.

- Ένα ακόμα optimization για την βελτίωση του performance είναι να αλλάξουμε το region από us-central-1 να το φέρουμε σε eu-west-3.

2.2.6.10 Γενικές παρατηρήσεις



Εικόνα 91: Cloud Run parameters benchmarking graph

Όπως βλέπουμε, λοιπόν, οι επιδόσεις του Cloud Run επηρεάζονται σημαντικά από τις παραπάνω παραμέτρους αλλά δεν υπάρχει κάποιο συγκεκριμένο best-practice. Υπάρχει ένα και μοναδικό trade-off μεταξύ Performance και Cost.
Είναι αρκετά ξεκάθαρο πως όσο περισσότερα active containers γεννιούνται με μεγάλη RAM και πολλά CPU Cores τόσο πιο κοντά φτάνουμε το Cloud Run στις μέγιστες του επιδόσεις, όμως έτσι αυξάνεται δραματικά και το κόστος. Σκοπός μας, λοιπόν, είναι να παραμετροποιήσουμε το εκάστοτε service ανάλογα με business case της εφαρμογής για να πετύχουμε το μέγιστο δυνατό Performance με το ελάχιστο δυνατό Cost.

Αυτή η ελευθερία στις παραμέτρους είναι πολύ σημαντικό χαρακτηριστικό του Cloud Run και από άποψη κόστους, καθώς σε FaaS προϊόντα η μοναδική παράμετρος που ίσως να μας σώσει από μεγάλα κόστη σε απρόβλεπτα spikes είναι το maximum-requests, η οποία όμως επηρεάζει το Resilience του συστήματος.

Μία επίσης σημαντική παρατήρηση είναι πως τα περισσότερα κόστη κατά την διάρκεια το testing προήλθαν από Idle Min-Instance CPU & Memory Allocation. Συνεπώς οφείλουμε να είμαστε πολύ προσεκτικοί με την παράμετρο των min-instances σε περιπτώσεις που δεν θέλουμε το service μας να είναι 24/7 διαθέσιμο.

2.2.3 GCP Cloud Functions

(Google, Cloud Functions, 2022)
Το Google Cloud Functions (GCF) είναι ένα FaaS προϊόν της Google το οποίο ανήκει κι αυτό στην Serverless νοοτροπία και συμμερίζεται όλα τα χαρακτηριστικά αυτής της αρχιτεκτονικής όπως Pay-Per-Use, Scalability κ.ο.κ. Είναι event-driven πλατφόρμα για lightweight εφαρμογές που υποστηρίζουν μεμονωμένα services και μπορούν να ενεργοποιηθούν είτε μέσω HTTP είτε μέσω background events.
Το Cloud Function υποστηρίζει συγκεκριμένες γλώσσες (και versions) όπως JavaScript, Python, Go κ.α. Επίσης ο κώδικας πρέπει να “πακεταριστεί” ως function - και για αυτόν τον λόγο το περιβάλλον υποστηρίζει συγκεκριμένα Frameworks όπως π.χ. για Python μόνο Flask και Django - και η Google μετά αναλαμβάνει το deployment και το execution του κώδικα όποτε χρειάζεται.

2.3.1 Cloud Functions vs Cloud Run

Το GCF, όπως κάθε FaaS product, μπορεί να διαχειριστεί μόνο ένα request τη φορά σε κάθε instance, οπότε είναι καλή επιλογή μόνο όταν το case είναι single-purpose.
Σε αντίθεση με το Cloud Run, που όπως είδαμε προηγουμένως, υποστηρίζει πολλαπλά concurrent requests σε ένα container instance γεγονός που μας γλιτώνει από χρόνο και κυρίως κόστος. Συνεπώς το Cloud Run είναι πιο efficient και πιο scalable με βάση την ζήτηση.

Επίσης, όπως έχουμε αναφέρει, το Cloud Function περιορίζει τον developer όσον αφορά το portability και το testability, καθώς έχει δικό του Web-Server και περιβάλλον με συγκεκριμένα limitations.

Όμως, ένα πλεονέκτημα του GCF έναντι των υπολοίπων είναι πως έχει άμεση συνδεσιμότητα με Triggers (χρησιμοποιώντας το Pub/Sub) που ενώνουν όλα τα Google Cloud προϊόντα. Π.χ. μπορεί να ενεργοποιηθεί ένα Cloud Function μόλις προστεθεί ένα καινούριο object σε ένα Bucket, ή μόλις το Billing Budget ξεπεράσει ένα συγκεκριμένο ποσό, ή όταν κάνει reboot ένα virtual machine. Χάρης τα triggers τα Cloud Functions μπορούν να λειτουργήσουν ως orchestrator και αποτελούν ένα πολύ χρήσιμο εργαλείο για διάφορους αυτοματισμούς.

Το Cloud Run, προς το παρόν, δεν έχει αυτήν την δυνατότητα.

2.3.2 Cloud Functions vs AWS Lambda

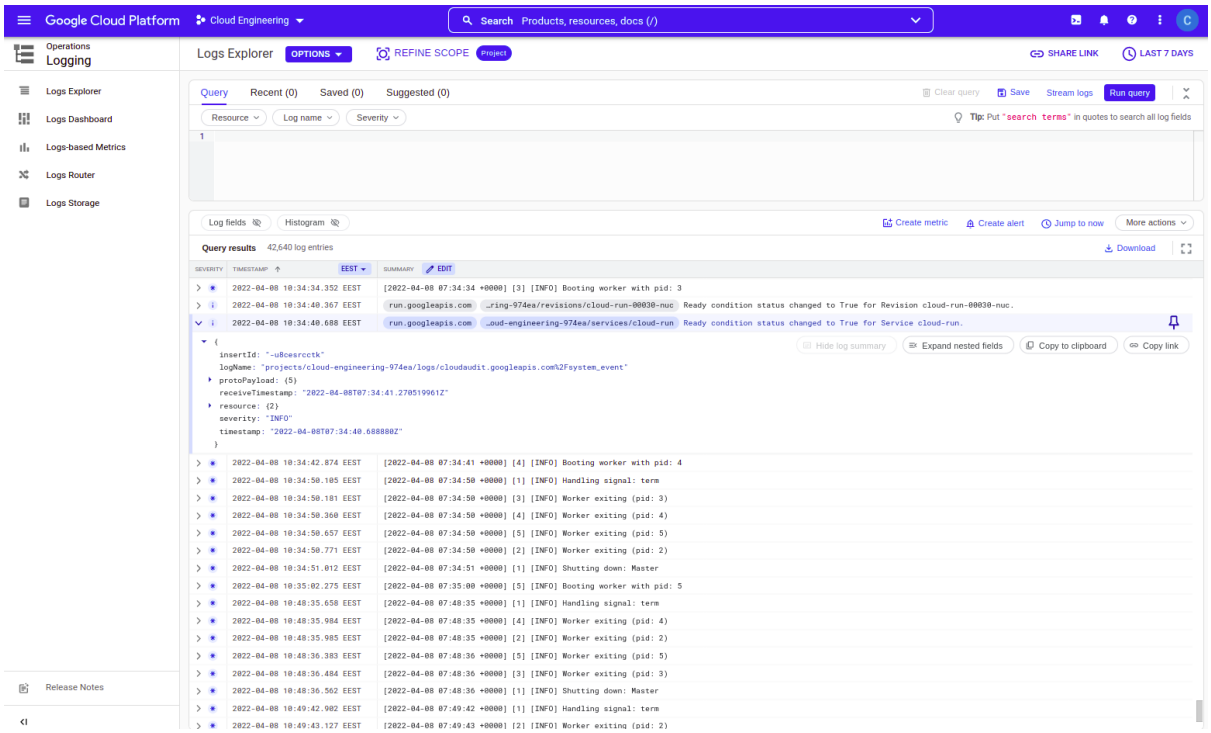
(Yongfook, 2020)
Ένα από τα πιο γνωστά FaaS είναι το και εκείνο της AWS. Τα benchmarks δείχνουν μία σημαντική υπεροχή του AWS Lambda έναντι του GCF στο επίπεδο του performance, όμως ούτε και εδώ υπάρχει ξεκάθαρη επιλογή.
Οπότε όταν έρχεται η στιγμή να επιλέξουμε μεταξύ 2 “ίδιων” προϊόντων, ως προς functionality, έρχονται στην επιφάνεια κάποια άλλα κριτήρια πέρα από το performance, τα οποία είναι τα εξής:

- ***Cost***

  Αυτή η παράμετρος επηρεάζεται κυρίως από τον ίδιο τον Cloud Provider. Πιο συγκεκριμένα η AWS έχει φθηνότερα computation services (CPU allocation, cost per request, storage costs κ.ο.κ.) όμως χρεώνει περισσότερα τα monitoring tools όπως το CloudWatch και το X-Ray. Η Google από την άλλη έχει ακριβότερα services όμως πολύ φθηνότερα monitoring & logging εργαλεία.

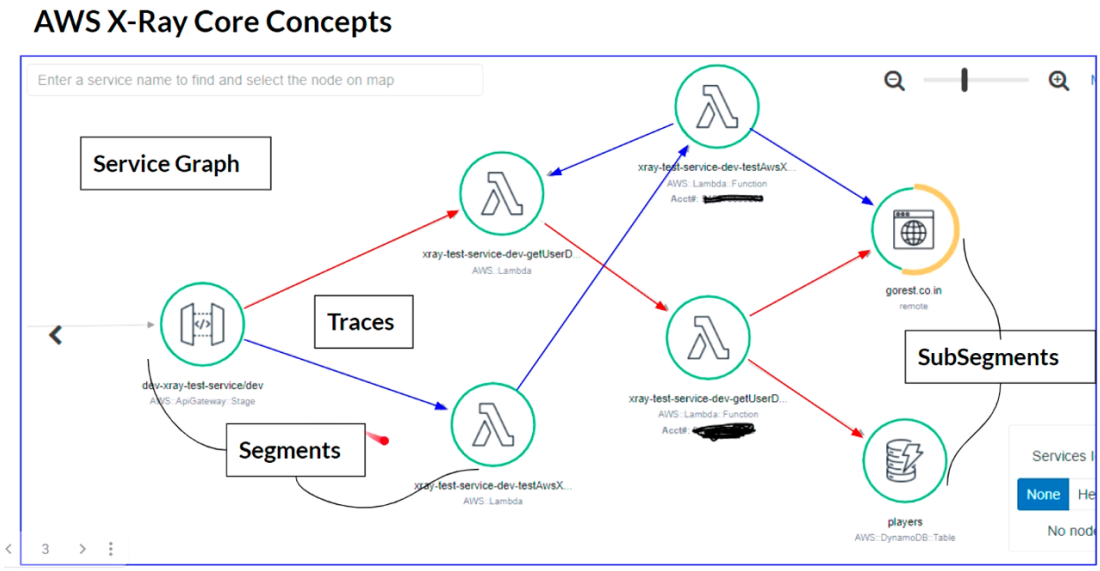
- ***Monitoring***

  Το Google Cloud έχει μακράν πολύ καλύτερο User Experience σε όλα της τα προϊόντα και κατ' επέκταση παρέχει πιο απλά και εύκολα στην χρήση monitoring εργαλεία.



Εικόνα 92: Google Cloud logger

Το AWS από την άλλη έχει πιο πολύπλοκα συστήματα παρακολούθησης των services, όμως έχει περισσότερες δυνατότητες και παρέχει μεγαλύτερο περιθώριο για customization και customized metrics. Επίσης το X-Ray, ένα Distributed Tracing System που χρησιμοποιείται για monitoring και troubleshooting, είναι κάτι που δεν έχει το GCP. Το X-Ray συλλέγει δεδομένα από διάφορα AWS Services και φτιάχνει τον λεγόμενο X-Ray χάρτη που βοηθάει τους developers να εξάγουν πολύ σημαντικά συμπεράσματα για το εκάστοτε application, όπως ποιο service αποτελεί το bottleneck για το όλο σύστημα, ποια endpoints έχουν περισσότερο traffic, ποιο service έχει τα περισσότερα errors κ.α.



Εικόνα 93: AWS X-Ray example

- ***Learning Curve***

  Το Learning Curve του AWS Lambda είναι σημαντικά μεγαλύτερο από εκείνο του Google Cloud Function.

- ***Deployment Workflows***

  Στο Google Cloud Function τα βήματα για το deployment είναι πολύ λιγότερα και πιο απλά σε σχέση με το AWS Lambda, όμως για το AWS Lambda υπάρχουν διάφορα 3rd party εργαλεία που διευκολύνουν και απλοποιούν αυτήν την διαδικασία όπως το Zappa για Python και το Serverless.com για Node.js.

Δεδομένου, λοιπόν, πως το AWS Lambda είναι ευρέως γνωστό και αποδεκτό ότι είναι αρκετά πιο γρήγορο από το Google Cloud Functions θα μελετήσουμε στην συνέχεια την συμπεριφορά και τις παραμέτρους μόνο του AWS Lambda.

2.4 AWS Lambda

(AWS, AWS Lambda, 2022)
Το Lambda είναι όπως αναφέραμε ένα FaaS προϊόν, δηλαδή ένα compute service που επιτρέπει να εκτελούμε κώδικα χωρίς παροχή ή διαχείριση των physical servers. Το Lambda εκτελεί τον κώδικά μας σε μια υπολογιστική υποδομή υψηλής διαθεσιμότητας και εκτελεί όλη τη διαχείριση των υπολογιστικών πόρων, συμπεριλαμβανομένης και της συντήρησης διακομιστή και λειτουργικού συστήματος.
Παρέχει capacity provisioning, automatic scalability, code monitoring and logging. Με το Lambda, μπορούμε να εκτελέσουμε κώδικα για σχεδόν κάθε τύπο εφαρμογής ή υπηρεσία. Το μόνο που χρειάζεται να κάνουμε είναι να γράψουμε τον κωδικό μας σε μία από τις γλώσσες που υποστηρίζει το Lambda.

2.4.1 Zappa

Πριν ξεκινήσουμε αξίζει να αναφέρουμε το Zappa. Πρόκειται για ένα πολύ χρήσιμο εργαλείο που διευκολύνει τα deployments σε AWS Lambda χρησιμοποιώντας Python (είτε Flask είτε Django).

(Zappa, 2022), (Kazarinoff, 2020)

Στην ουσία ο developer καλείται να γράψει μόνο τον κώδικα, στην συγκεκριμένη περίπτωση ένα Flask Application, και το Zappa αναλαμβάνει:

1. να φτιαξει τα απαραίτητα yaml αρχεία,
2. να κάνει zip τον κώδικα και να το ανεβάσει σε ένα Storage Bucket (AWS S3 στην προκειμένη),
3. να φτιάξει ένα Lambda Function με τα απαραίτητα permissions,
4. να συνδέσει το API Gateway με το Lambda και να κάνει generate ένα public URL που θα κάνει trigger την συνάρτηση.
5. Και, τέλος, φτιάχνει ένα Event που καλεί την συνάρτηση κάθε 4 λεπτά έτσι ώστε να την κρατάει “ζεστή” με σκοπό να αντιμετωπιστούν τα λεγόμενα “Cold Starts”, για οποία θα μιλήσουμε στην πορεία.

Σε επόμενο βήμα το Zappa:

1. με την εντολή ”zappa tail” σου επιτρέπει να δεις τα logs που παράγονται από την Lambda,
2. έχει σύστημα διαχείρισης των release channels, π.χ.:

```
{
    "dev": {
        "app_function": "app.app",
        "aws_region": "eu-west-3",
        "profile_name": "softlab",
        "project_name": "softlab-cloud-engineering",
        "runtime": "python3.8",
        "s3_bucket": "softlab-cloudeng-energy-apis-bucket",
        "excluded": ["venv", "reqs.txt", "fill_database.py"],
        "slim_handler": true,
        "xray_tracing": false,
        "cloudwatch_log_level": "OFF",
        "cloudwatch_data_trace": false,
        "cloudwatch_metrics_enabled": false,
        "memory_size": 2048

    }
}
```

Εικόνα 94: Example of zappa configuration

Εδώ βλέπουμε ένα απλό configuration json αρχείο που παραμετροποιεί το Lambda function που θα δημιουργήσει το Zappa. Όπως παρατηρούμε, υπάρχει ένα object με όνομα dev, με τον ίδιο τρόπο μπορώ να φτιάξω και ένα με όνομα prod ή stage. Συνεπώς με τις εντολές zappa update (ή deploy) dev/prod/stage μπορώ να διαχειρίζομαι τα διάφορα channels.

Αυτά, λοιπόν, είναι τα βασικά που μπορεί να κάνει το Zappa. Σε πιο προχωρημένο στάδιο μπορούμε:

- να φτιάξουμε δικούς μας Authorizers που μπαίνουν σε API Gateway επίπεδο και όχι μέσα στον κώδικα μας,
- να βάλουμε environmental variables σε Cloud επίπεδο (ή Secret Keys) για να μην έχουμε hardcoded sensitive κωδικούς ή διάφορα κλειδιά.

και άλλες αυτοματοποιήσεις που καταγράφονται στα αντίστοιχο Documentation.

2.4.2 Slow Performance, γιατι;

Στην αρχή έγινε η εξής παρατήρηση: Τα (dummy) APIs έτρεχαν πιο γρήγορα (~1.5s) σε local deployment και πιο αργά σε AWS (~4.2s). Ακολουθήθηκε το εξής Debugging:

1. Ανεβάσαμε τον κώδικα σε ένα Google Cloud VM και παρατηρήσαμε ότι έτρεχε πιο γρήγορα από το local, όπως και ήταν αναμενόμενο, άρα επιβεβαιώσαμε πως δεν γινόταν κάτι περίεργο στο local deployment.

2. Μελετήσαμε τα configuration settings και είδαμε πως μπορεί να αυξηθεί η RAM που διατίθεται στην Lambda. Σύμφωνα με την AWS, όσο αυξάνεται η RAM αυξάνεται και το CPU power και κατ' επέκταση το κόστος.

3. Στην αρχή, λοιπόν, είχαμε θέσει αυτήν την παράμετρο ίση με 512MB και διαπιστώσαμε πως δεν είναι αρκετή για το virtual environment του κώδικά μας, δηλαδή packages & modules. Πράγμα που σημαίνει ότι σε run-time η RAM δεν είναι αρκετή και το instance που τρέχει το Lambda θα έπρεπε να ανατρέξει σε σκληρό δίσκο και κατ' επέκταση να είναι πολύ αργό.

4. Αυξήσαμε την RAM σε 10GB, που είναι το μέγιστο, και το response time έπεσε στο 0.9s από τα 4.2s.

5. Μειώσαμε σε 1GB και ο χρόνος πήγε στο 2s

6. Καταλήξαμε πως τα 2GB, καθώς είχαμε τους ίδιους χρόνους με την επιλογή των 10GB, είναι η βέλτιστη RAM έτσι ώστε να έχουμε και μικρότερο pricing.

2.4.3 Παράλληλα Requests

(Rapes, 2019), (AWS, AWS service quotas , 2022)
Όπως έχουμε ήδη αναφέρει μία πολύ σημαντική παράμετρος για το scalability του συστήματος είναι η συμπεριφορά του σε συνθήκες με πολλά παράλληλα requests.

Μέχρι στιγμής, λοιπόν, το testing γινόταν με σειριακά requests, οπότε πρέπει να μελετηθεί η συμπεριφορά του συστήματος και σε συνθήκες με πολλά παράλληλα requests.

2.2.3.1 AWS Lambda Concurrency

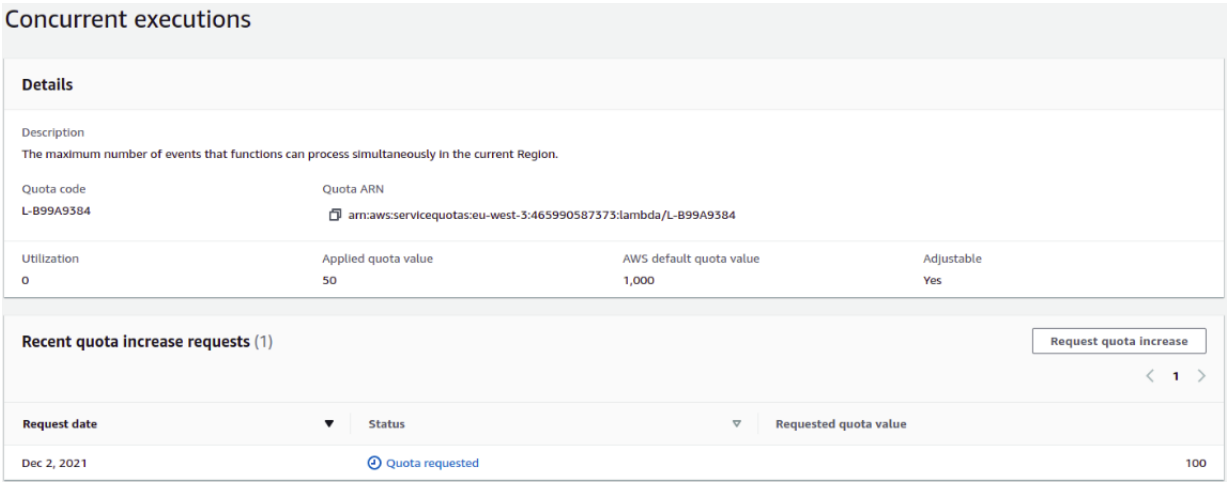
Concurrency is the number of requests that your function is serving at any given time. When your function is invoked, Lambda allocates an instance of it to process the event. When the function code finishes running, it can handle another request. If the function is invoked again while a request is still being processed, another instance is allocated, which increases the function's concurrency. The total concurrency for all the functions in your account is subject to a per-region quota.

By default, μόλις γίνει deploy ένα Lambda Service, το AWS δίνει 3 workers (VMs ή αλλιώς Instances) που είναι dedicated στην συνάρτηση. Τί γίνεται, όμως, όταν έρθουν ταυτόχρονα 1000 requests;

2.2.3.2 Increase Request

Συνολικά ένας AWS λογαριασμός έχει max quota 1000 concurrent requests σε ένα common pool per region. Δηλαδή ένας λογαριασμός μπορεί να έχει 1000 στο Παρίσι (eu-west-3), 1000 στο Μιλάνο (eu-south-1) κ.ο.κ.

Όμως, by default, το max quota για concurrent Lambda Executions είναι 50. Για να αυξηθεί, λοιπόν, πρέπει να γίνει ένα Increase Request που περνάει από το AWS Support και χρειάζονται περίπου 1-2 μέρες μέχρι να εγκριθεί. Δεν χρειάζεται κάποια αιτιολόγηση του αιτήματος. Στην παρακάτω εικόνα φαίνεται το αίτημα αύξησης από τα 50 στα 100.



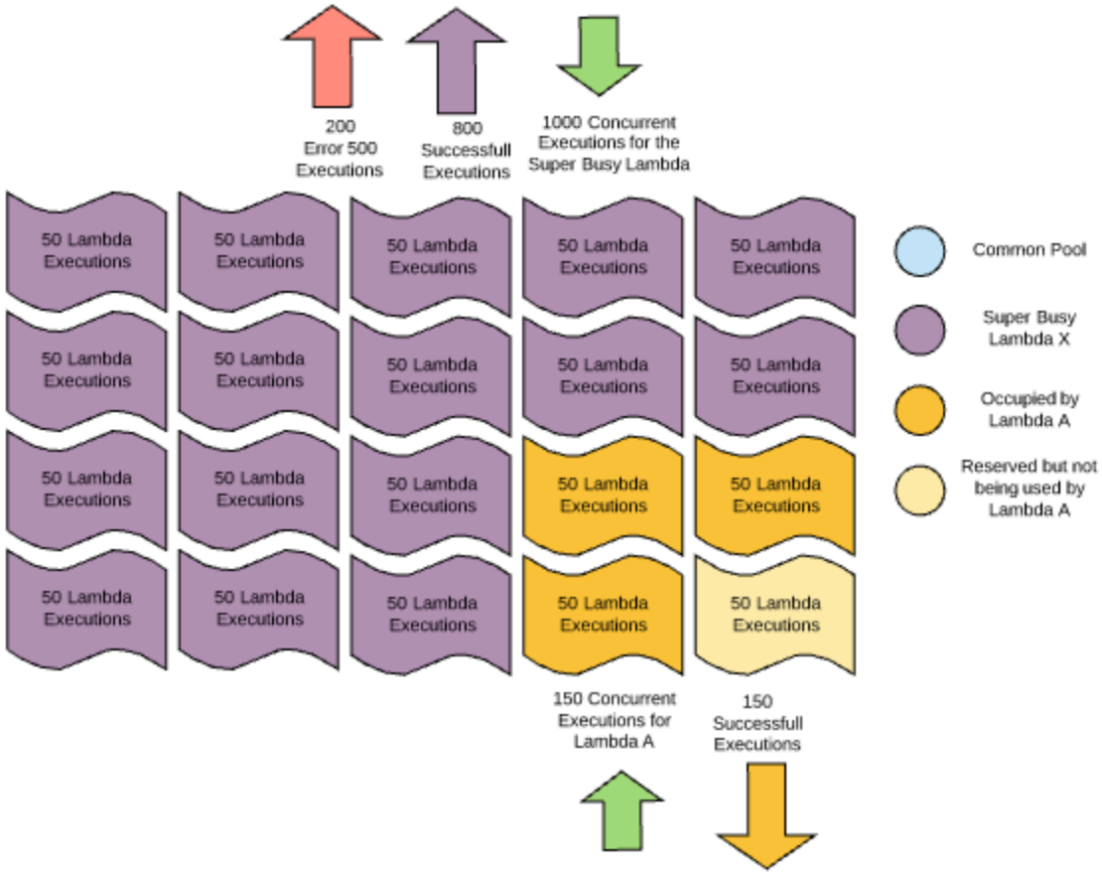
Εικόνα 95: AWS concurrency increase request

2.2.3.3 Reserve Concurrency

Άρα, λοιπόν, υπάρχει ένα common pool από requests.
Πως γίνεται ο διαμοιρασμός αυτών των requests όμως;

Μπορεί στο σύστημα μας να έχουμε πολλά lambda, όμως μερικές από αυτές να είναι πιο σημαντικές και το “Internal Server Error” να μην είναι αποδεκτό.

Οπότε στην ουσία θέλουμε να δώσουμε κάποια προτεραιότητα σε αυτές τις Lambda.
Εδώ έρχεται κάτι που λέγεται Reserve Concurrency, που στην ουσία “δεσμεύει” συγκεκριμένο quota για συγκεκριμένες Lambda. Έτσι, λοιπόν, εξασφαλίζεται ότι οι πιο “σημαντικές” Lambda θα έχουν πάντα τα requests που χρειάζονται από το pool.



Εικόνα 96: AWS reserve concurrency

2.2.3.5 Provisioned Concurrency

(AWS, Managing Lambda provisioned concurrency , 2022)
Αν, όμως, έχουμε λάθος εκτιμήσει πως 100 quota είναι αρκετά και έρθουν ξαφνικά πολλά παραπάνω requests, τί γίνεται;
Δεν μπορεί ο developer να αλλάζει συνεχώς αυτό το όριο και ούτε μπορεί να βάλει μία μεγάλη τιμή καθώς συνολικά έχουμε 1000.
Εδώ έρχεται κάτι που λέγεται Provisioned Concurrency.

Business case από το documentation της AWS.

- Έστω ότι διαχειριζόμαστε ένα Delivery Application και περιμένουμε ένα traffic spike στις 20:00. Θέλουμε η εφαρμογή να είναι responsive ακόμα και όταν το demand κάνει γρήγορο scale-up & scale-down, οπότε ενεργοποιούμε το Provisioned Concurrency και το ορίζουμε ίσο με π.χ. 100 για την χρονική περίοδο που επιθυμούμε.

- Για την ακρίβεια, το Provisioned Concurrency απλά αρχικοποιεί και έχει έτοιμα συγκεκριμένα execution environments έτσι ώστε να μπορέσουμε να ανταποκριθούμε αμέσως στα επερχόμενα Lambda invocations.

- Προσοχή, το Provisioned Concurrency κοστίζει και έχει δικό του pricing.

Τι γίνεται, όμως, όταν και πάλι δεν μπορείς να προβλέψεις το σωστό time frame του πιθανού traffic spike; Εδώ έρχεται το Application Auto Scaling που διαχειρίζεται εντελώς αυτόματα το provisioned concurrency ανάλογα με κάποιες μετρικές.
(AWS, What is Application Auto Scaling? , 2022)

Γιατί να θέλουμε να περιορίσουμε όμως τα executions; Γιατί να μην θέσουμε σε όλα τα regions το max quota και να είμαστε σίγουροι; Υπάρχουν διάφοροι λόγοι:

1. Cost ή Security. Μπορεί να μην θέλουμε κάποιος να κάνει κατά λάθος ή κακόβουλα μεγάλο αριθμό αιτημάτων στο σύστημά μας.

2. Performance, καθώς επιβάλλουμε λογικά batch sizes.

3. Scalability, αντιστοίχιση του throughput με τους σωστούς πόρους.

4. Off Switch, στην ουσία μπορούμε, μειώνοντας το concurrency, να εξασφαλίσουμε ότι δεν θα ρέει κίνηση στα συστήματα μας και θα είναι σαν να τα απενεργοποιούμε.

2.2.4 Testing

Έχοντας, λοιπόν, μελετήσει όλες τις παραμέτρους που επηρεάζουν το concurrency ήρθε η ώρα να επιβεβαιώσουμε όλες τις θεωρητικές υποθέσεις.

Έστω ότι έχουμε θέσει ως max concurrency σε μία Lambda ίσο με 50. Τι θα συμβεί όταν έρθουν 100 παράλληλα requests;

Φτιάξαμε ένα script που, χρησιμοποιώντας fork, γεννάει 50 παράλληλα requests. Όλες οι απαντήσεις ήρθαν ταυτόχρονα και στον ίδιο περίπου χρόνο. Μόλις, όμως, βάλαμε 51 τότε ήρθε μία απάντηση με Internal Server Error. Οπότε έτσι επιβεβαιώθηκε η συμπεριφορά του concurrent invocation.

Όμως, αυτή δεν είναι “σωστή” συμπεριφορά. Κανονικά θα έπρεπε να εκτελεστεί και το 51ο request απλά με μία καθυστέρηση καθώς θα περίμενε να απελευθερωθεί κάποιος από τους 50 workers.

Ακολουθήσαμε τα εξής βήματα για debugging:

1. Στο script προσθέσαμε ένα time.sleep(0.01), δηλαδή 10ms, ανάμεσα σε κάθε os.fork()

2. Οπότε καλέσαμε 60 requests και ενώ περιμέναμε να δούμε 10 Internal Server Error messages ήρθαν μόνο 3 και τα υπόλοιπα 7 είχαν απλά μεγαλύτερο response time.

3. Δοκιμάσαμε, λοιπόν, 100 παράλληλα requests με time.sleep(0.1), δηλαδή 100ms, και ήρθαν μόνο 2 Internal Server Errors και υπήρχαν μόνο 20 απαντήσεις με αυξημένο response time.

Σε αυτό το σημείο αντιληφθήκαμε πως υπάρχει ένας μηχανισμός άμυνας των DDoS Attacks, καθώς η συμπεριφορά αλλάζει ανάλογα με το timestamp των requests.

4. Για να επιβεβαιώσουμε, λοιπόν, αυτόν τον μηχανισμό ανεβάσαμε τον κώδικα σε Google Cloud VM έτσι ώστε να έχει διαφορετική IP από το local. Καλέσαμε 50 requests από local και 50 από το VM με time.sleep(0), δηλαδή χωρίς κάποιο ενδιάμεσο χρόνο.

Άρα συνολικά έχουμε 100 ταυτόχρονα requests με την μόνη διαφορά πως έρχονται από διαφορετική IP.
Δεν υπήρξε κανένα Internal Server Error και αυξημένο response time υπήρξε μόνο σε 10 responses.

2.2.5 Συμπεράσματα

Το Lambda σύστημα του AWS:

1. Μπορεί να διαχειριστεί μέχρι 50 requests (ή όσα ορίσεις) που έχουν ίδιο timestamp και ίδια IP.

2. Είναι “αυστηρό” για αριθμό requests που ξεπερνούν το επιτρεπτό όριο και έχουν το ίδιο timestamp και IP.

3. Όταν υπάρχει χρονική διαφορά ~10ms μεταξύ των requests το σύστημα γίνεται πιο ελαστικό αλλά και πάλι υπάρχει κίνδυνος για Internal Server Error.

4. Μπορεί να διαχειριστεί σχεδόν απεριόριστα requests από διαφορετική IP και με διαφορά κάποια ms, με average response time περίπου το ίδιο.

Κεφάλαιο 3 Μελέτη των βάσεων δεδομένων σε αυτόνομη διάταξη - Καλές πρακτικές

Για να μπορέσουμε να προσομοιώσουμε την λειτουργία μιας ολόκληρης εφαρμογής δεν θα μπορούσαμε να παραλείψουμε το κομμάτι της Βάσης Δεδομένων.
Θα μελετήσουμε 2 από τα πιο γνωστά NoSQL Databases Cloud Products.

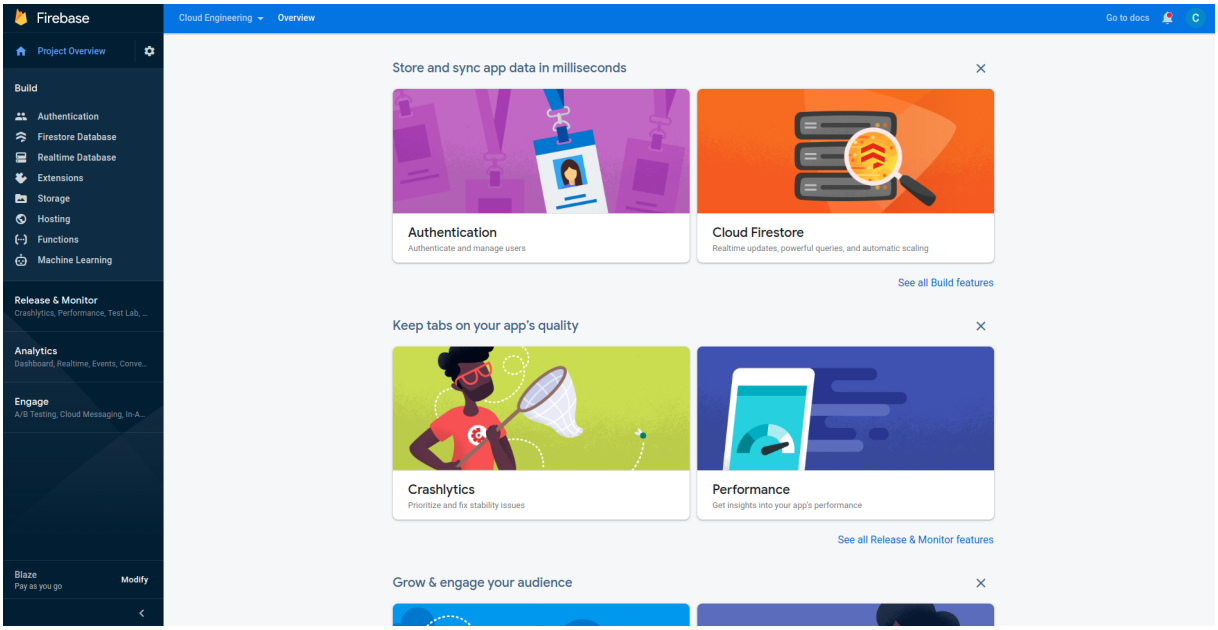
1. Cloud Firestore (GCP) – NoSQL Database
2. AWS DynamoDB – NoSQL Database

2.3.1 Cloud Firestore

Το Firestore είναι μια βάση δεδομένων NoSQL που έχει δημιουργηθεί για αυτόματη κλιμάκωση, υψηλή απόδοση και ευκολία στην ανάπτυξη εφαρμογών. Ενώ η διεπαφή Firestore έχει πολλά κοινά χαρακτηριστικά με τις παραδοσιακές βάσεις δεδομένων, ως βάση δεδομένων NoSQL διαφέρει από αυτές στον τρόπο που περιγράφει τις σχέσεις μεταξύ των αντικειμένων δεδομένων.

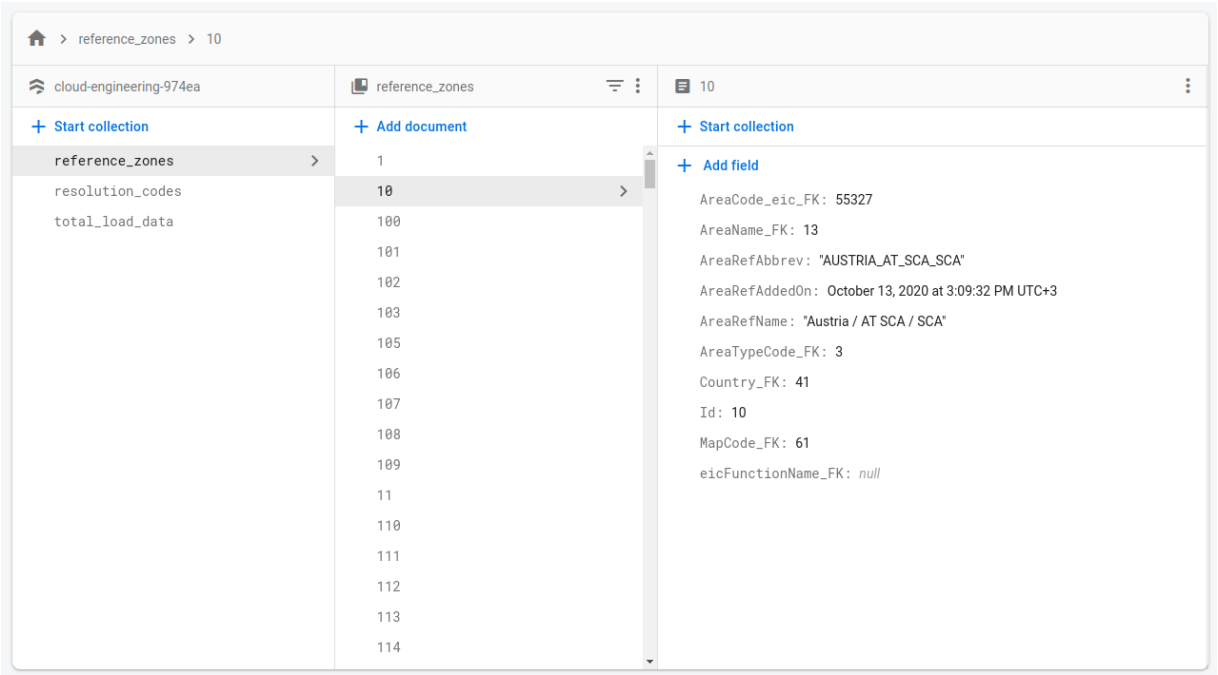
3.1.1 Firebase

Μία Firestore Βάση Δεδομένων μπορεί να δημιουργηθεί από το ίδιο το GCP Console αλλά δημιουργείται και σε ένα Firebase Project.
Αρκετοί νομίζουν πως το Firebase είναι κάτι διαφορετικό από το GCP, στην πραγματικότητα όμως είναι το ίδιο καθώς το Firebase χρησιμοποιεί Google Cloud products.
Για την ακρίβεια το Firebase είναι ένα οικοσύστημα που προσφέρει διάφορα εργαλεία και προϊόντα για να μπορέσει να υλοποιηθεί και γίνει deploy μία εφαρμογή είτε είναι web-app είτε mobile-app.



Εικόνα 97: Firebase project overview

Προσφέρει Authentication, Firestore Database, Hosting, Storage, Functions, Monitoring Tools και πολλά άλλα.
Συνεπώς, όταν φτιάχνουμε ένα Firebase Project φτιάχνεται και ένα Firestore Database. Η οποία στο Firebase φαίνεται κάπως έτσι:



Εικόνα 98: Cloud Firestore collection overview

3.1.2 Bulk Write

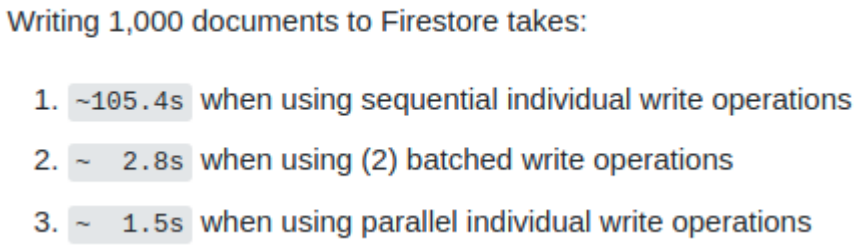
Πριν ξεκινήσουμε, λοιπόν, να αλληλεπιδρούμε με το Firestore πρέπει να το γεμίσουμε με δεδομένα. Το Firestore ναι μεν είναι NoSQL και λειτουργεί με μία JSON αρχιτεκτονική αλλά δεν υποστηρίζει κάποια “Import from File” λειτουργία. Άρα δεν υπάρχει κάποια προφανής διαδικασία για Bulk Write.

Στην δική μας περίπτωση, έχουμε ένα Dataset από 10.000.000 Documents (~4 GB) και στην προσπάθεια μας για Bulk Data Entry αντιμετωπίσαμε 2 προβλήματα:

1. Μεγάλο JSON αρχείο δεν μπορεί να διαβαστεί με τον κλασικό τρόπο της Python που φέρνει στην μνήμη όλο το αρχείο και φτιάχνει ένα ολικό μεγάλο json object.
   Η λύση σε αυτό το πρόβλημα:
   - Πρέπει να διαβάζουμε line by line έτσι ώστε να μην χρειαστεί να φορτώσουμε όλο το αρχείο.
   - Η κλασσικές μέθοδοι με getline(), streamline() etc. δεν λειτουργούν καθώς απαιτούν ειδικό format του αρχείου. Η λύση είναι μια ειδική βιβλιοθήκη ijson που σου επιτρέπει να διαβάζεις όχι line by line, αλλά στην ουσία json by json.

2. Το να βάζεις στο Firestore ένα-ένα τα documents παίρνει πολύ χρόνο (10M documents estimated time ~ 11 μέρες). ***Λύση***:

Batches. Στην ουσία συνδυάζεις μέχρι και 500 operations (μόνο set(), update(), delete()) μέσα σε ένα API Request. (Estimated time ~ 7 ώρες). Το κόστος παραμένει το ίδιο. Κάθε operation μετράει κανονικά.



Εικόνα 99: Cloud Firestore bulk write times

(Puffelen, 2019), (Google, Best practices, 2022)

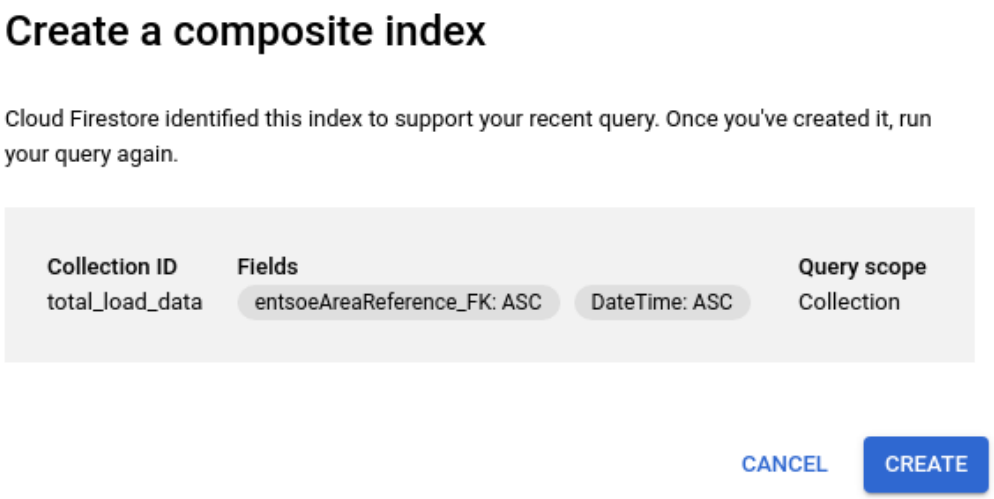
3.1.3 Indexes

Το Firestore είναι γνωστό για τις υψηλές επιδόσεις του από άποψη Performance, Availability, Scalability, Concurrency.

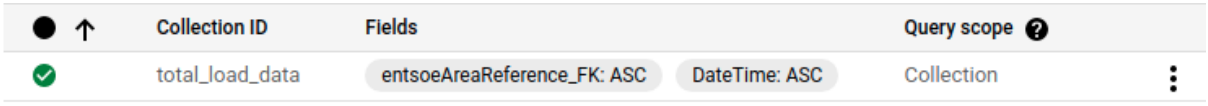
- Αξίζει να σημειωθεί ότι το Firestore λειτουργεί με ειδικό replica-set που σημαίνει ότι τα δεδομένα μας είναι αποθηκευμένα σε πολλά instances -και ας φαίνεται ότι είναι μόνο σε ένα- και για αυτόν λόγο έχουμε υψηλό Availability και Scalability από την πρώτη κιόλας μέρα. Το Firestore με τον ίδιο τρόπο που μπορεί να υποστηρίξει 1 read/sec μπορεί και 100.000 reads/sec χωρίς καμία παραπάνω ρύθμιση από τον developer.

- Άμα δεν υπάρχει κάποιο index για ένα query οι περισσότερες βάσεις δεδομένων ψάχνουν όλα τους δεδομένα στοιχείο ανά στοιχείο, μία πολύ αργή διαδικασία που επιβραδύνεται ακόμα περισσότερο καθώς μεγαλώνει η βάση δεδομένων. Το Cloud Firestore εγγυάται υψηλή απόδοση χρησιμοποιώντας indexes για όλα τα queries. Ως αποτέλεσμα, το παράδοξο του Firestore, η απόδοση ενός query εξαρτάται από το μέγεθος του συνόλου αποτελεσμάτων και όχι από τον αριθμό των στοιχείων στη βάση δεδομένων. Δηλαδή ένα query που επιστρέφει 50KB δεδομένα ως απάντηση θα χρειαστεί τον ίδιο χρόνο διεκπεραίωσης είτε έχουμε 100 documents στην βάση μας είτε έχουμε 100 εκατομμύρια
  (Google, Index types in Cloud Firestore, 2022)

Όπως έχουμε αναφέρει, το Google Cloud φημίζεται για το πόσο user friendly και για το πόσο διευκολύνει τις διαδικασίες για τους developers. Μία ακόμη απόδειξη είναι το Firestore καθώς αντιλαμβάνεται από μόνο του ποια indexes χρειάζεται να δημιουργηθούν ανάλογα με τα queries που γίνονται, και τα φτιάχνει μόνο του.
Η διαδικασία φαίνεται κάπως έτσι:



Εικόνα 100: Cloud Firestore create composite index



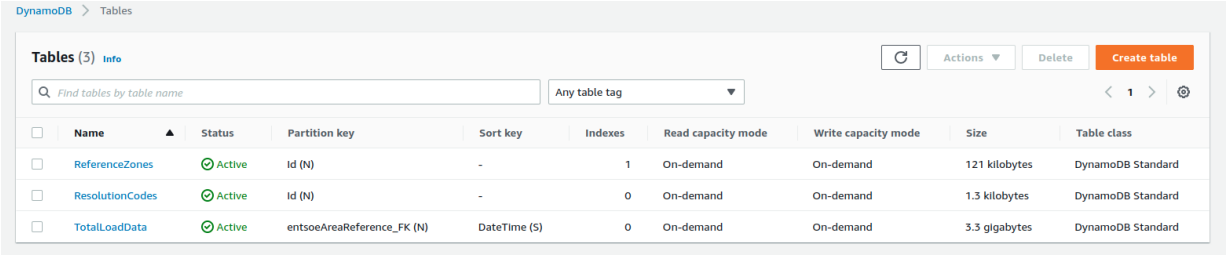
Εικόνα 101: Cloud Firestore composite index is created

Στην ουσία ένα composite-index αποθηκεύει ένα ταξινομημένο mapping όλων των documents ενός collection βάση των πεδίων που χρειαζόμαστε στα queries μας. Για αυτόν τον λόγο τα indexes κοστίζουν, καθώς χρειάζονται αποθηκευτικό χώρο. Υπάρχουν best practices για να μειωθούν αυτά τα κόστη.
(Google, Reducing index costs with map fields, 2022)

3.2 AWS DynamoDB

Η Amazon DynamoDB είναι μία πλήρως αυτοματοποιημένη, serverless, key-value NoSQL βάση δεδομένων σχεδιασμένη για απαιτητικές high-performance εφαρμογές κάθε κλίμακας. Η DynamoDB παρέχει built-in security, continuous backups, automated multi-Region replication, in-memory caching, and data export tools.
Λέγεται DynamoDB γιατί έχει την αρχιτεκτονική της MongoDB αλλά με δυναμικό Scalability.

Μία εικόνα για το πως φαίνεται το Dashboard της DynamoDB.

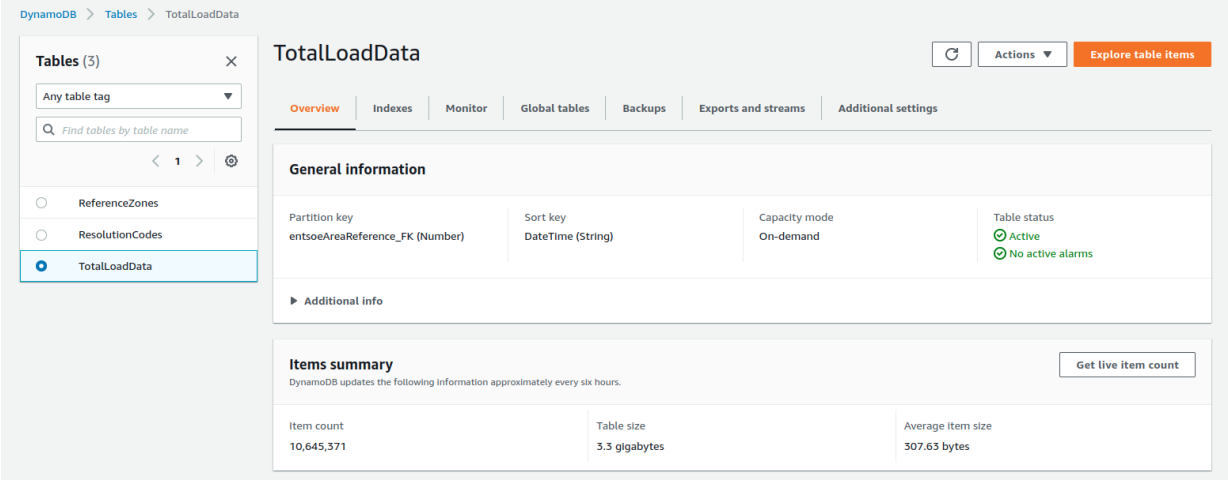


Εικόνα 102: DynamoDB collection overview

Εδώ παρατηρούμε πως αντί για collections, ονομάζονται tables αλλά αυτό δεν πρέπει να μας παραπέμπει σε SQL. Σε επόμενο στάδιο βλέπουμε το overview ενός πίνακα.

Με το explore table items μπορούμε:

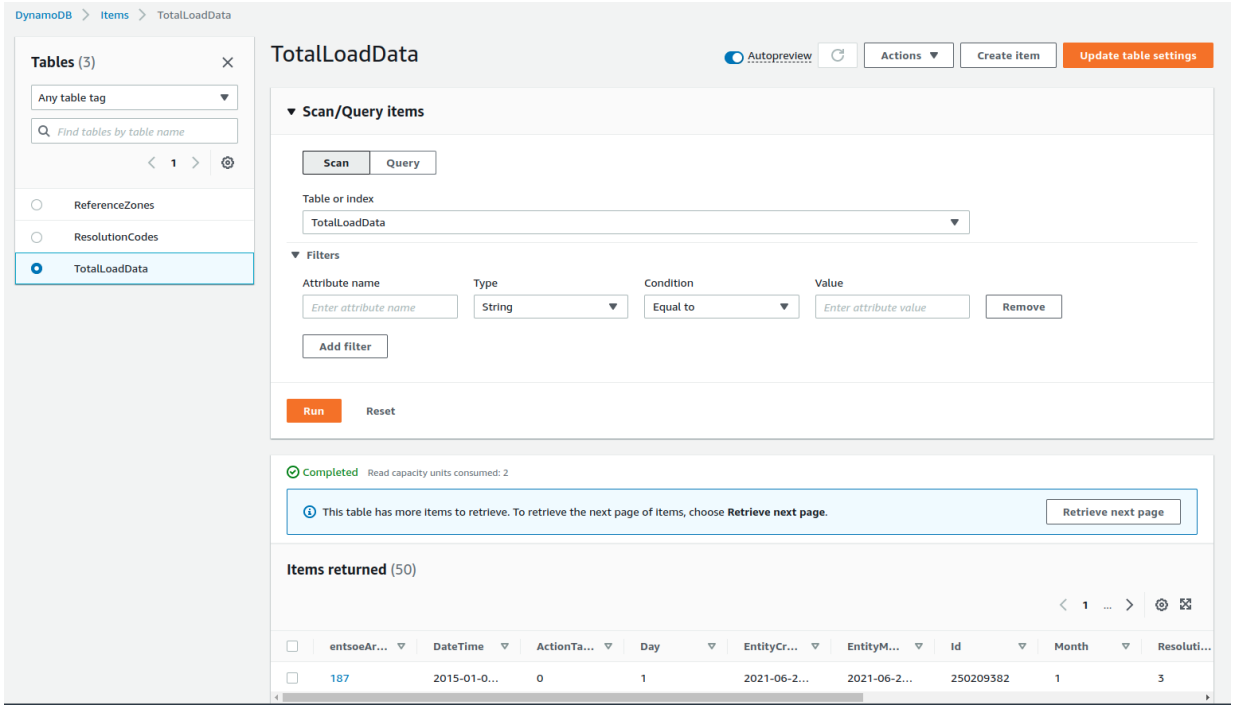
- να δούμε τα items μας μέσα από την κονσόλα,



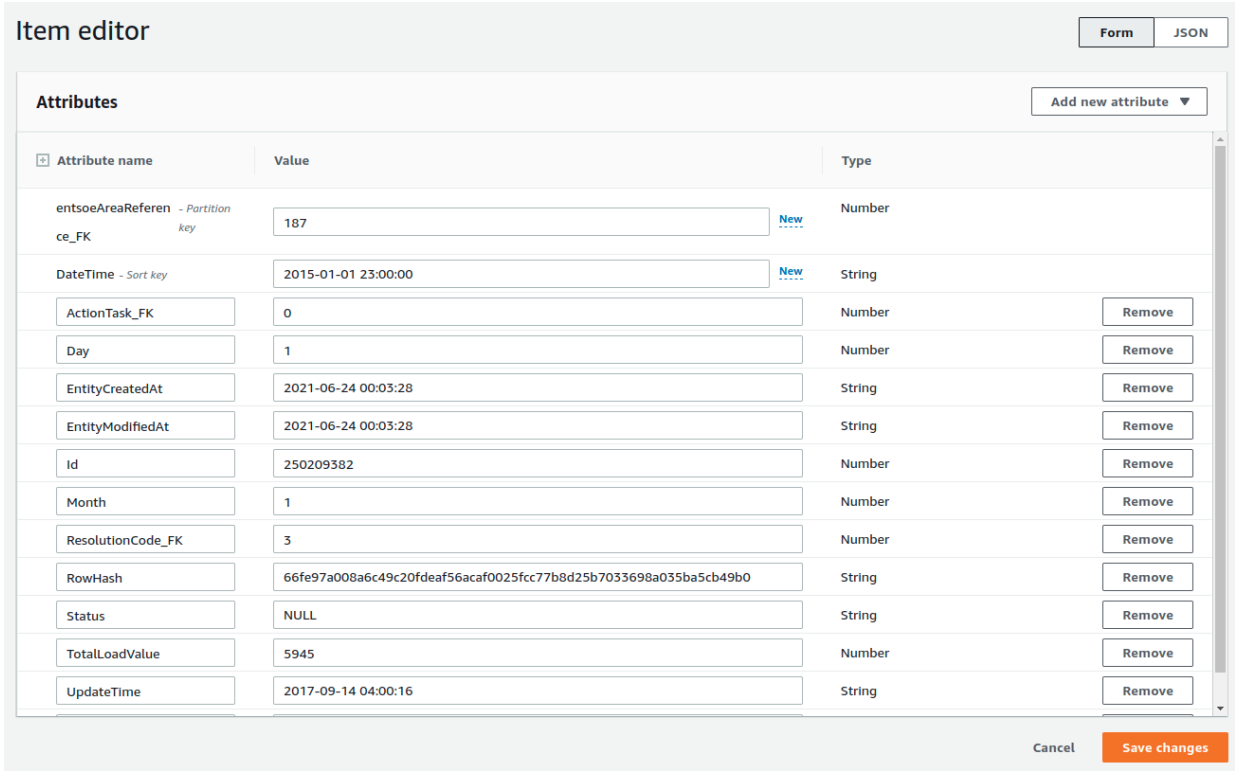
Εικόνα 103: DynamoDB explore table items

- να προσθέσουμε ένα καινούριο item,
- να τρέξουμε scans με πολλά φίλτρα πάνω στα attributes των δεδομένων μας,
- και να τρέξουμε queries για να ελέγξουμε την ορθότητα τους πριν τα ενσωματώσουμε στον κώδικά μας.

Όλες αυτές τις λειτουργίες τις έχει και το Firestore Console, πέρα από το Scan που το έχει το μόνο το DynamoDB.

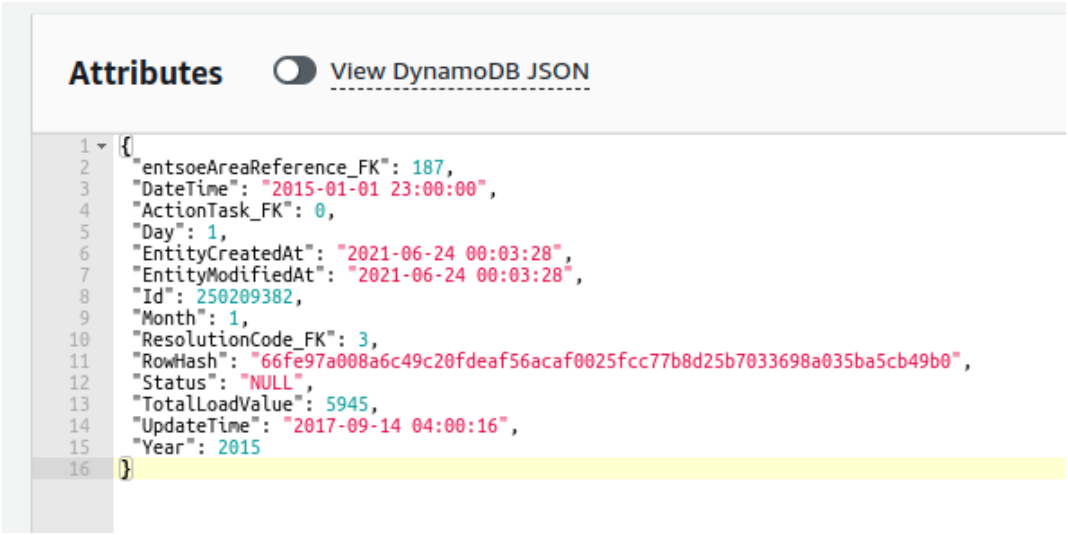


Εικόνα 104: DynamoDB table overview



Εικόνα 105: DynamoDB item editor

Επίσης υπάρχει και ο Item Editor που έχει 2 επιλογές οπτικοποίησης είτε Form είτε JSON. Είτε κανονικό JSON, είτε DynamoDB JSON που στην ουσία είναι το ίδιο JSON αλλά με τα attributes του DynamoDB.



Εικόνα 106: DynamoDB view simple JSON

Εικόνα 107: View as DynamoDB JSON

3.2.1 Firestore vs DynamoDB

Οι βασικές διαφορές μεταξύ δύο NOSQL βάσεων δεδομένων, πέρα από το Performance, Scalability, Availability κ.ο.κ., έγκειται στις δυνατότητες και στις ελευθερίες που παρέχουν τα Queries τους στους developers.
Η DynamoDB φημίζεται ότι έχει υψηλότερες επιδόσεις από το Firestore, όμως αυτό είναι κάτι που θα το διαπιστώσουμε στην συνέχεια. Εδώ θα αναλύσουμε τις βασικές διαφορές σε Query επίπεδο.

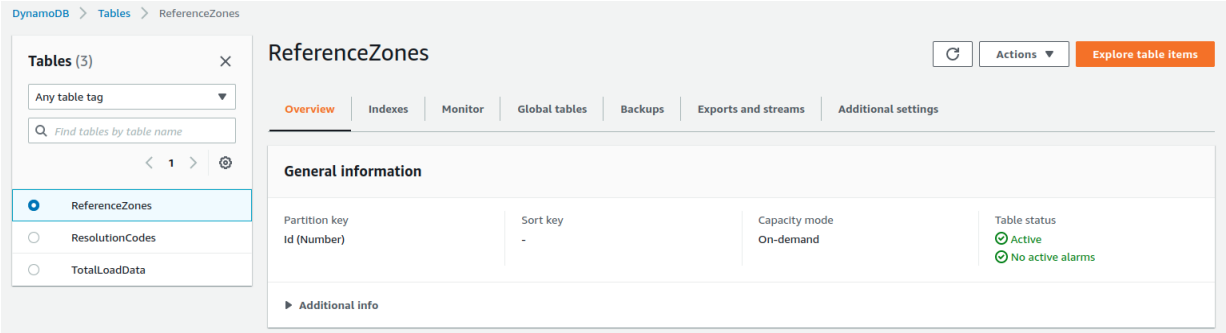
1. Το DynamoDB έχει Scans, ενώ το Firestore όχι. Η διαφορά μεταξύ Scan και Query είναι πως το Scan "σκανάρει" ολόκληρο το table ψάχνοντας για στοιχεία που πληρούν τα κριτήρια, ενώ το Query κάνει ένα ειδικό lookup σε επιλεγμένο partition με βάση το primary ή το secondary partition/hash key.

Και εδώ έρχεται μία από τις σημαντικότερες διαφορές του DynamoDB και του Firestore. Στο DynamoDB κατά την διάρκεια ενός item καλούμαστε να επιλέξουμε ποιο θα είναι το Partition Key και ποιο το Sort Key. Αυτά τα δύο συμβάλλουν στην δημιουργία του τελικού Hash Key και μόνο αυτά τα δύο μπορούμε να χρησιμοποιούμε για τα Queries μας. Αυτό συμβαίνει για να ξέρει το σύστημα εξ αρχής ποια indexes να δημιουργήσει. Σε αντίθεση με το Firestore που δημιουργεί, εξ αρχής για κάθε νέο document, single indexes για όλα τα πεδία και απλά μετέπειτα σου δίνει την δυνατότητα δημιουργίας πιο σύνθετων indexes, των λεγόμενων composite indexes όπως έχουμε προαναφέρει.

Αξίζει να σημειωθεί πως χάρης αυτό το χαρακτηριστικό η DynamoDB καταφέρνει να έχει ομοιόμορφη κατανομή φορτίου.
(AWS, Best Practices for Designing and Architecting with DynamoDB, 2022)

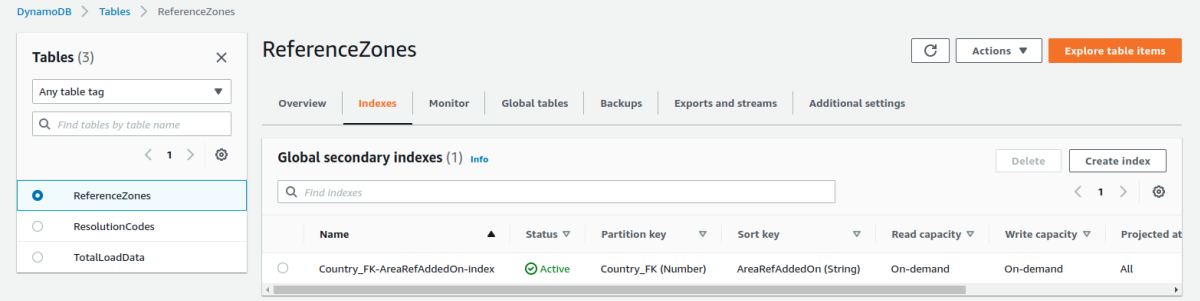
Αυτό το χαρακτηριστικό προσθέτει ένα παραπάνω αρχιτεκτονικό βάρος στον developer καθώς πρέπει να σκεφτεί εξ αρχής ποια θα είναι τα πιθανά queries που θα χρειάζεται η εκάστοτε εφαρμογή με απώτερο σκοπό να επιλέξει σωστά το Partition και Sort Key.
Αυτό δεν σημαίνει όμως, βέβαια, ότι δεν μπορούν να δημιουργηθούν καινούρια indexes βασισμένα σε διαφορετικά attributes. Υπάρχουν τα λεγόμενα Global Secondary Indexes που μας επιτρέπουν να εκτελούμε queries με attributes που δεν αποτελούν το primary ή sort key.

Π.χ. εδώ βλέπουμε ένα Table με Primary key το πεδίο με όνομα ID.



Εικόνα 108: DynamoDB primary key

Αλλά έχουμε φτιάξει και ένα καινούριο Index με βάση το Country_FK και το AreaRefAddedOn.



Εικόνα 109: DynamoDB global secondary indexes

Προφανώς η δημιουργία και η χρήση των Global Secondary Indexes κοστίζει για τους ίδιους λόγους που κοστίζουν και τα indexes του Firestore.

2. Στην DynamoDB το update ή το delete υπό συνθήκη δεν απαιτεί read στην αρχή. Σε αντίθεση με το Firestore, όπου αν θέλουμε να διαγράψουμε ένα document με συγκεκριμένα χαρακτηριστικά θα πρέπει να κάνουμε ένα read στην αρχή για να βρούμε το ID του και μετά να το διαγράψουμε. Αυτό έχει ως αποτέλεσμα στην DynamoDB να χρειαζόμαστε μόνο ένα operation, ενώ στο Firestore θέλουμε δύο.

3. Στην DynamoDB μπορούμε να αλλάξουμε τα αντικείμενα μιας λίστας ενός Item σε ένα single operation, σε αντίθεση με το Firestore που πρέπει να διαβάσουμε το αντικείμενο και μετά να το ξανακάνουμε update ολόκληρο το document έχοντας αλλάξει την λίστα.

Για περισσότερες διαφορές μεταξύ των Queries του DynamoDB και του Firestore:

- DynamoDB
  (AWS, Boto3 documentation, 2022)

- Firestore
  (Google, Perform simple and compound queries in Cloud Firestore, 2022)

3.2.2 Bulk Write

Όπως και στο Firestore έτσι και εδώ έπρεπε να γεμίσουμε την βάση με δεδομένα.
Στο DynamoDB υπάρχουν 2 τρόποι:

1. AWS Data Pipeline
   (AWS, Importing Data From Amazon S3 to DynamoDB , 2022)

   Εάν μπορούμε να ανεβάσουμε τα δεδομένα μας στο AWS S3 (Simple Storage Service) - κάτι που είναι πολύ πιο γρήγορο και πιο φθηνό από το να βάλουμε τα δεδομένα μας κατευθείαν στο DynamoDB - μπορούμε μετά να χρησιμοποιήσουμε το AWS Data Pipeline για να κάνουμε export τα δεδομένα στο DynamoDB. Στην ουσία το Data Pipeline αυτοματοποιεί την διαδικασία
   - δημιουργίας ενός Amazon EMR Cluster
   - και εξαγωγής των δεδομένων από το S3 στο DynamoDB σε μορφή παράλληλων BatchWriteItem requests.

Συνεπώς, εάν χρησιμοποιήσουμε το Data Pipeline δεν θα χρειάζεται να γράψουμε εμείς οι ίδιοι τον κώδικα για parallel batch transfer.

2. Batches
   (Hiltch, 2022)

   Έχουν ακριβώς την ίδια λογική με τα Batches του Firestore, δηλαδή πολλά single operations σε ένα μόνο request. Γλιτώνουμε bandwidth όμως το κόστος παραμένει το ίδιο.

Κεφάλαιο 4 Τελική συγκριτική αξιολόγηση

Ένα απλό Back-End μιας οποιασδήποτε εφαρμογής χρειάζεται τουλάχιστον ένα execution environment για τον κώδικα μας και μία βάση δεδομένων. Στην δική μας περίπτωση αναλύσαμε μεμονωμένα τα εξής Cloud Products:

- Execution Environments:
    1. AWS Lambda – FaaS
    2. GCP Cloud Functions – FaaS
    3. GCP Cloud Run – Stateless Containers

- Databases:
    1. AWS DynamoDB – NoSQL Database
    2. Cloud Firestore (GCP) – NoSQL Database

Σκοπός αυτής της προεργασίας ήταν να διερευνήσουμε τα best-practices έτσι ώστε να καταφέρουμε να κάνουμε ένα αξιόπιστο τελικό benchmarking. Οπότε σαν επόμενο βήμα είναι να ενώσουμε τα παραπάνω μεταξύ τους και να προσομοιώσουμε καταστάσεις και traffic spikes κανονικών εφαρμογών με APIs και Database. Τα deployments που δοκιμάστηκαν είναι τα εξής:

1. Cloud Firestore - AWS Lambda (monolithic)
2. Cloud Firestore - GCP Cloud Run (monolithic)
3. Cloud Firestore - GCP Cloud Functions (monolithic)
4. Cloud Firestore - GCP Cloud Functions (polylithic)
5. AWS DynamoDB - AWS Lambda (monolithic)
6. AWS DynamoDB - GCP Cloud Run (monolithic)

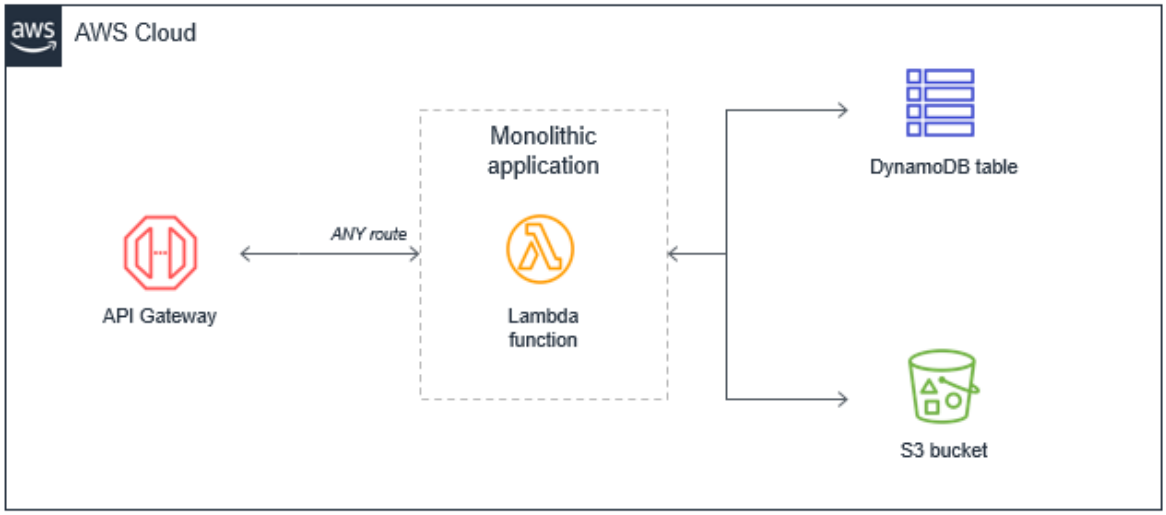
Εδώ βρίσκεται ο πηγαίος κώδικας για κάθε περίπτωση:
[https://github.com/Cloud-Engineering-Softlab-Project]

Κάθε repository περιέχει ένα Flask Application αλλά και τα απαραίτητα configuration αρχεία που χρειάζονται για να συνδεθούμε στην εκάστοτε βάση δεδομένων. Στις περιπτώσεις που έχουμε επιλέξει AWS θα δείτε και ένα zappa_settings.json αρχείο του οποίου ο ρόλος είναι να παραμετροποιήσει το zappa. (Το Zappa όπως έχουμε αναφέρει είναι ένα εργαλείο που διευκολύνει το deployment σε AWS Lambda).

4.1 Monolithic vs Polylithic

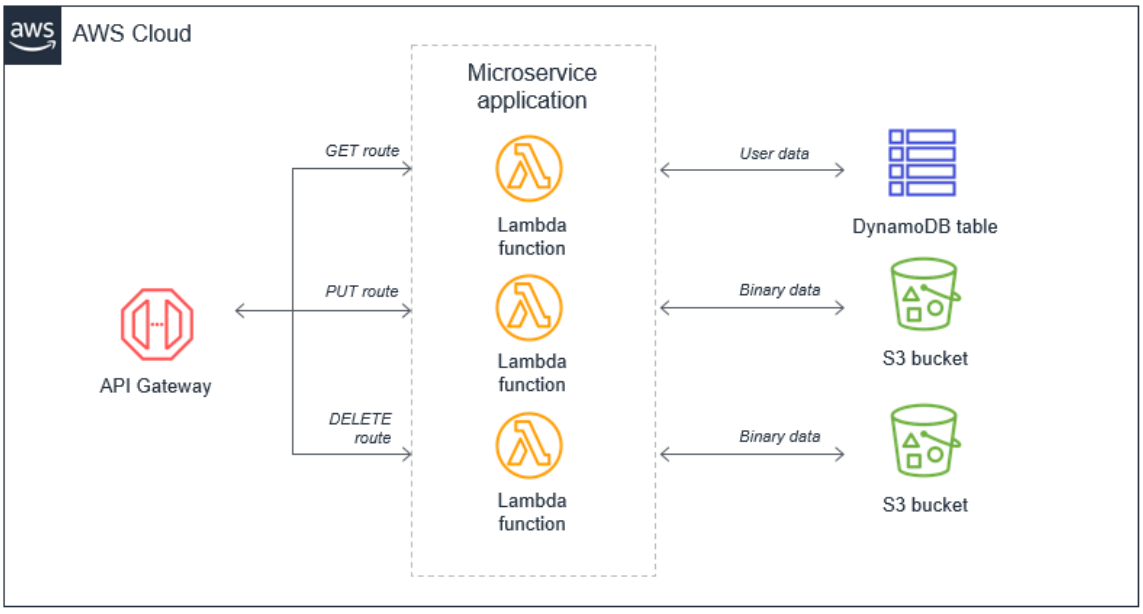
Πριν προχωρήσουμε αξίζει να σχολιάσουμε τις διαφορές μεταξύ το μονολιθικου και του πολυλιθικού μοντέλου.

Μονολιθικό μοντέλο είναι όταν πακετάρουμε όλα μας τα endpoints μέσα σε μία Lambda, ή μέσα σε ένα Cloud Run Service ή μέσα σε ένα Cloud Function. Αυτό σημαίνει ότι υπάρχει ένα Base-URL για όλα μας τα APIs και αυτό σημαίνει ότι μία συνάρτηση ή ένα container διαχειρίζεται όλο το logic και όλα τα triggers.
Π.χ. για AWS Lambda



Εικόνα 110: Monolithic application

Πολυλιθικό μοντέλο είναι όταν χωρίζουμε τα endpoints μας σε διαφορετικές συναρτήσεις ή containers ακολουθώντας πάντα μια συγκεκριμένη στρατηγική. Δηλαδή έστω ότι φτιάχνουμε το Back-End για το LinkedIn, θα χωρίζαμε τα APIs που αφορούν τους mobile-users, τους web-users, τους clients, τους recruiters κ.ο.κ. Και άμα θέλαμε να το σπάσουμε ακόμα περισσότερο θα μπορούσαμε να χωρίσουμε τα APIs σε αυτά που είναι υπεύθυνα για το προφίλ ενός χρήστη, αυτά για την Homepage, αυτά για το Login κ.ο.κ.
Π.χ. για AWS Lambda



Εικόνα 111: Polylithic application

Όταν έχουμε λίγα endpoints με κοινή λογική, κοινά μοντέλα και κοινά triggers θα μπορούσαμε να ακολουθήσουμε το μονολιθικό μοντέλο, αλλά εν γένει δεν είναι πολύ καλή πρακτική διότι:

- Package size: Για την Python, για παράδειγμα, αν έχουμε όλα τα πακέτα που χρειάζεται το Back-End μέσα σε ένα service τότε κινδυνεύει το περιβάλλον, που χρειάζεται για να τρέξει ο κώδικας μας, να είναι πολύ μεγάλο και να χρειάζεται

παραπάνω RAM από το κανονικό. Αυτό έχει ως αποτέλεσμα είτε να αυξήσουμε την RAM, που σημαίνει παραπάνω κόστος, είτε να είναι αργό το service μας καθώς θα χρειάζεται να ανατρέχουμε στον σκληρό δίσκο σε κάθε request.

- Harder to upgrade: Σε production περιβάλλον τα upgrades μπορεί να προκαλέσουν σημαντικά προβλήματα. Όταν έχουμε, λοιπόν, όλο μας το σύστημα σε ένα service και απλά θέλουμε να αλλάξουμε κάτι σε ένα απλό endpoint, τότε βάζουμε κάθε φορά σε κίνδυνο όλο μας το σύστημα καθώς θα πρέπει να γίνουν deploy ξανά όλα μας τα endpoints.

- Harder to maintain: Είναι δύσκολο όταν υπάρχουν πολλοί developers να δουλεύουν σε ένα μονολιθικό code repository και είναι ακόμα πιο δύσκολο να γραφτούν testing scripts.

- Harder to reuse code: Τυπικά είναι αρκετά δύσκολο να διαχωρίσουμε reusable κώδικα από μονολιθικά projects.

- Harder to test: Όπως είπαμε και προηγουμένως είναι δύσκολο να τεστάρουμε καθώς όσο μεγαλώνει το project τόσο μεγαλώνει και ο κώδικας και είναι ακόμα πιο δύσκολο να ελέγξεις όλους τους πιθανούς input και entry point συνδυασμούς. Γενικά είναι ευκολότερο το testing σε πολυλιθικά μοντέλα καθώς υπάρχει και η παράμετρος του isolation.

  (AWS, The Lambda monolith , 2022)

Παρατηρούμε ότι παραπάνω δεν αναφέρθηκε το επιχείρημα περί “single point of failure” και δεν τέθηκε το ερώτημα “Τι θα γίνει εάν στο μονολιθικό μοντέλο πέσει το σύστημα; Θα καταρρεύσει όλη η εφαρμογή;” Η απάντηση είναι πως ναι, αλλά στο Serverless Cloud δεν είναι αυτό το νόημα του πολυλιθικού μοντέλου.
Κανονικά οι Cloud Providers υπόσχονται ότι δεν θα πέσει ποτέ το σύστημα μας και ότι πάντα θα υπάρχει υψηλό Availability και Scalability. Ο μόνος τρόπος για να πέσει το σύστημα είναι να έχει κάνει λάθος ο developer, οπότε το πολυλιθικό μοντέλο δεν υπόσχεται υψηλότερο Availability ούτε ελαχιστοποίηση του ρίσκου, απλά υπόσχεται καλύτερες συνθήκες coding, updating, testing, maintaining, monitoring, debugging. Αυτά θα πρέπει να μας ενδιαφέρουν μόνο στην Serverless αρχιτεκτονική.

Στην δική μας περίπτωση, λοιπόν, εφόσον είχαμε μόνο 2 βασικά endpoints που αλληλοεπιδρούν με την βάση και εφόσον χρησιμοποιούν κοινά packages το πολυλιθικό μοντέλο θα ήταν overkill.

4.2 Monolithic Cloud Function

Η μόνη περίπτωση στην οποία εφαρμόσαμε και τα δύο μοντέλα είναι στο Cloud Functions. Αυτό έγινε καθαρά για πειραματικούς σκοπούς καθώς το Cloud Function είναι φτιαγμένο για να διαχειρίζεται μόνο μία συνάρτηση οπότε μελετήθηκε το πως γίνεται ένα Cloud Function να διαχειριστεί ένα ολόκληρο σύστημα. Κάτι τέτοιο προφανώς και δεν είναι best practice.

4.3 APIs

Τα δεδομένα που έχουμε εισάγει στην βάση είναι προέρχονται από το ENTSO-E, πιο συγκεκριμένα:

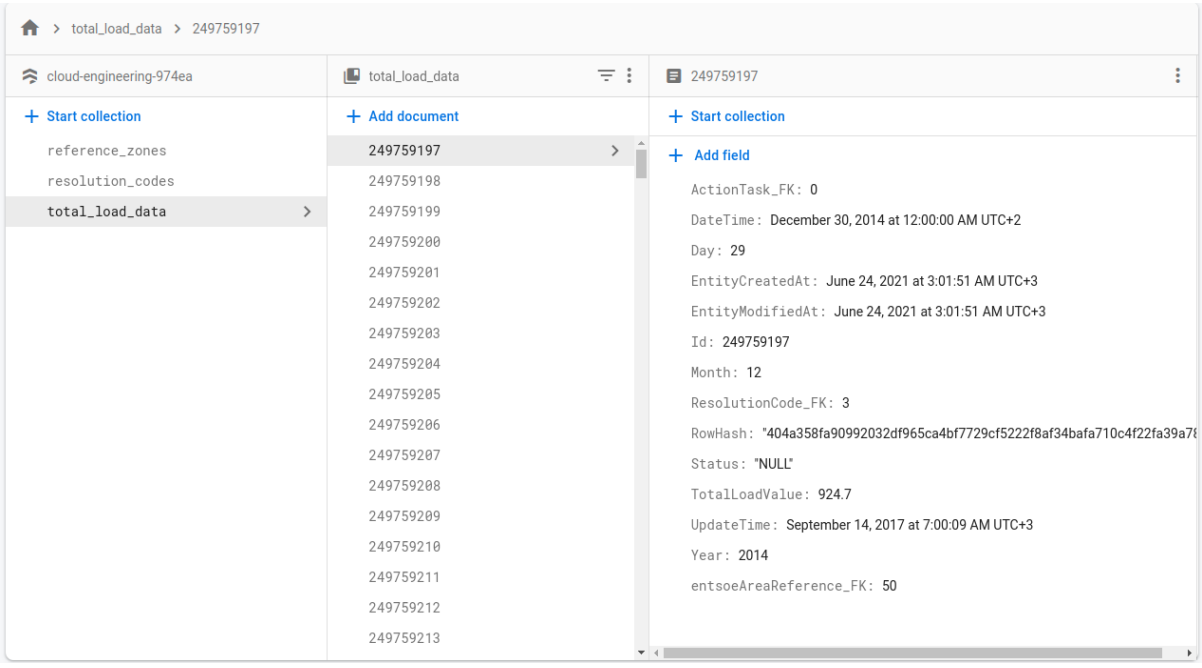
Το ENTSO-E, το Ευρωπαϊκό Δίκτυο Διαχειριστών Συστημάτων Μεταφοράς Ηλεκτρικής Ενέργειας, είναι η ένωση για τη συνεργασία των ευρωπαϊκών διαχειριστών συστημάτων μεταφοράς (ΔΣΜ). Οι 39 ΔΣΜ μέλη που εκπροσωπούν 35 χώρες είναι υπεύθυνοι για την ασφαλή και συντονισμένη λειτουργία του συστήματος ηλεκτρικής ενέργειας της Ευρώπης, του μεγαλύτερου διασυνδεδεμένου ηλεκτρικού δικτύου στον κόσμο. Εκτός από τον βασικό, ιστορικό ρόλο του στην τεχνική συνεργασία, το ENTSO-E είναι επίσης η κοινή φωνή των ΔΣΜ. Το ENTSO-E και τα μέλη του, ως ευρωπαϊκή κοινότητα ΔΣΜ, εκπληρώνουν μια κοινή αποστολή: Διασφάλιση της ασφάλειας του διασυνδεδεμένου συστήματος ηλεκτρικής ενέργειας σε όλα τα χρονικά διαστήματα σε πανευρωπαϊκό επίπεδο και τη βέλτιστη λειτουργία και ανάπτυξη των ευρωπαϊκών αγορών διασυνδεδεμένης ηλεκτρικής ενέργειας, επιτρέποντας ταυτόχρονα την ενοποίηση της ηλεκτρικής ενέργειας που παράγεται από ανανεώσιμες πηγές ενέργειας και των αναδυόμενων τεχνολογιών Συγκεκριμένα χρησιμοποιήσαμε το dataset Actual Total Load [6.1.A] που ορίζεται ως το συνολικό φορτίο που είναι ίσο με το άθροισμα της ισχύς που δημιουργείται από τα εργοστάσια παραγωγής και των δύο δικτύων από τα οποία συνάγονται.

(ENTSOE, entsoe.eu, 2022)
(ENTSOE, ENTSO-E Transparency Platform, 2022)

Πριν ξεκινήσουμε ας αναλύσουμε τα endpoints και την σχέση που έχουν με τα δεδομένα:

4.3.1 Get Simple Energy Data

Αυτό το Endpoint γυρνάει documents από το Collection “total_load_data”:



Εικόνα 112: total_load_data collection

Query Parameters:

- *zone_code* (*int*): είναι το αντίστοιχο ResolutionCode_FK στα attributes.
- *date_from* (*string*): είναι το αντίστοιχο DateTime στα attributes.
- *duration* (*int*): αναφέρεται σε ημέρες.

Logic:

Ζητάμε για δεδομένα που έχουν ResolutionCode_FK ίδιο με το zone_code και το DateTime τους ανήκει στο range date_from + duration.

Successful Response:

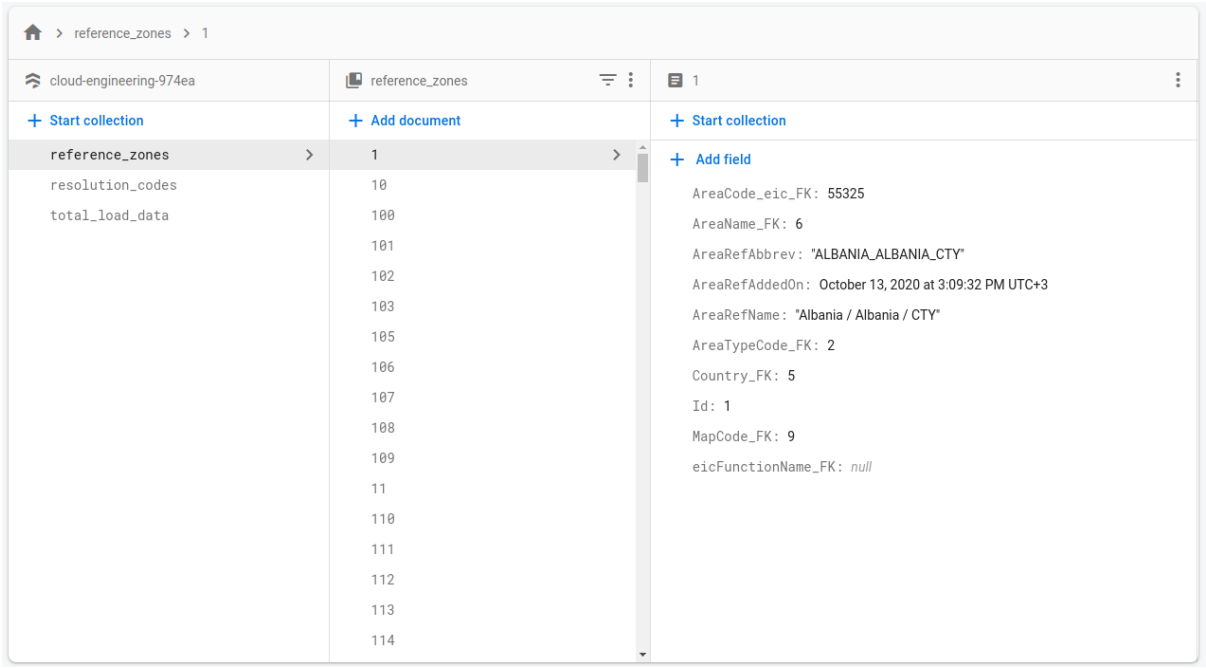

```
{
    "times": {
        "query_energy_data": 0.18474340438842773,
        "get": 0.1852884292602539
    },
    "parameters": {
        "zone_code": "3",
        "date_from": "20-12-2014",
        "duration": "20"
    },
    "len_of_data": 172,
    "data": [
        {
            "entsoeAreaReference_FK": 3,
            "ResolutionCode_FK": 1,
            "TotalLoadValue": 674.0,
            "DateTime": "2015-01-04 23:00:00+00:00"
        },
        {…
        },
        {…
        },
        {…
        },
        {
```


Εικόνα 113: Get Simple Energy Data Successful response

4.3.2 Get Advanced Energy Data

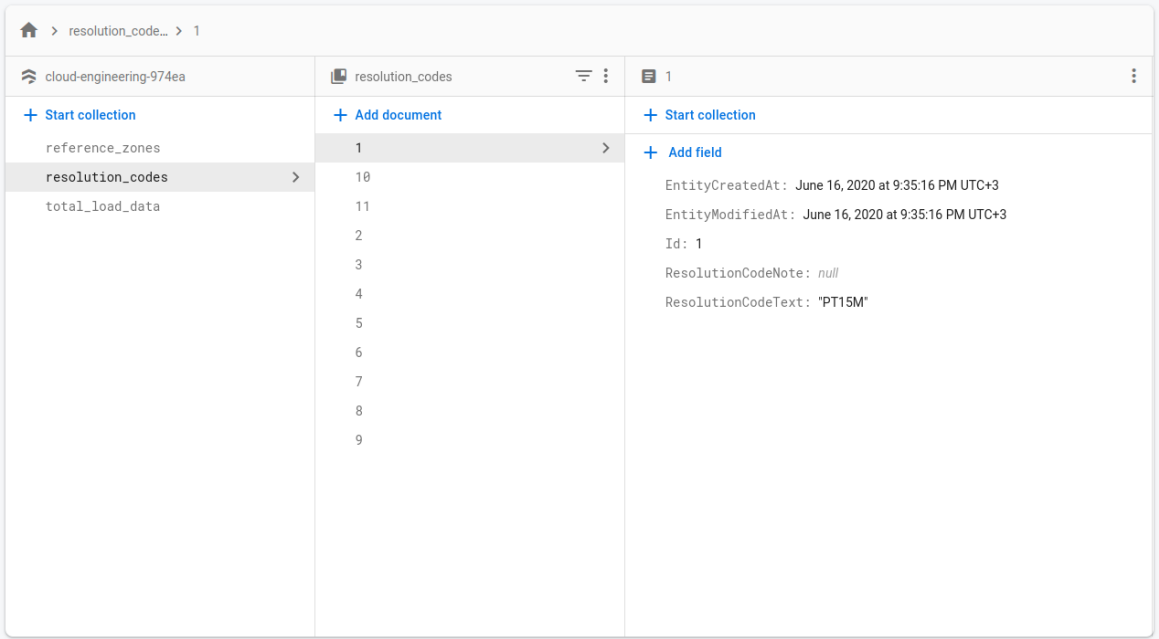
Εδώ έχουμε υλοποιήσει μία πιο σύνθετη μορφή του Get Simple Energy Data καθώς εκτελούμε κάτι σαν το Join που έχουν οι SQL βάσεις δεδομένων. Θυμίζουμε πως μια από τις μεγαλύτερες διαφορές μεταξύ SQL και NoSQL είναι πως η NoSQL δεν υποστηρίζει το Join και πρέπει οι προγραμματιστές στην σχεδίαση του Database Structure να το λάβουν αυτό υπόψη για να δώσουν τα κατάλληλα IDs στα documents. Κάτι τέτοιο κάναμε και εμείς εδώ: Παρατηρούμε πως υπάρχουν κι άλλα δύο collections.

- “reference_zones”



Εικόνα 114: reference_zones collection

- και “resolution codes”



Εικόνα 115: resolution_codes collection

Σε ένα total_data_load document υπάρχουν τα πεδία:

- entsoeAreaReference_FK που αντιστοιχεί στο ID των reference_zones documents.
- ResolutionCode_FK που αντιστοιχεί στο ID των resolution_codes documents.

Οπότε, λοιπόν, άμα θέλουμε μέσα σε ένα total_data_load document να εντάξουμε και τις πληροφορίες του εκάστοτε reference_zone ή του resolution_code ξέρουμε κατευθείαν το object path μέσα στην βάση και δεν χρειάζεται να εκτελέσουμε queries για να βρούμε το αντίστοιχο document.

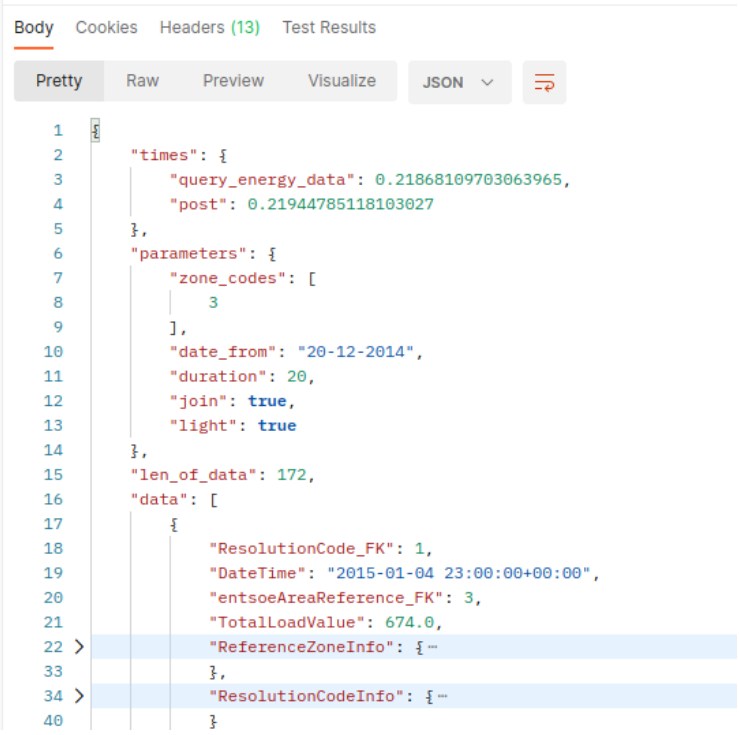
Body Parameters:

- *zone_codes (Array of int):* είναι το αντίστοιχο ResolutionCode_FK στα attributes.
- *date_from (string):* είναι το αντίστοιχο DateTime στα attributes.
- *duration (int):* αναφέρεται σε ημέρες.
- *join (bool):* εάν True εκτελεί την Join λειτουργία που αναφέραμε.
- *light (bool):* εάν True τότε γυρνάει όλο το total_data_load object.

Logic:

Ζητάμε για δεδομένα που έχουν ResolutionCode_FK ίδιο με το zone_code και το DateTime τους ανήκει στο range date_from + duration.

Successful Response:

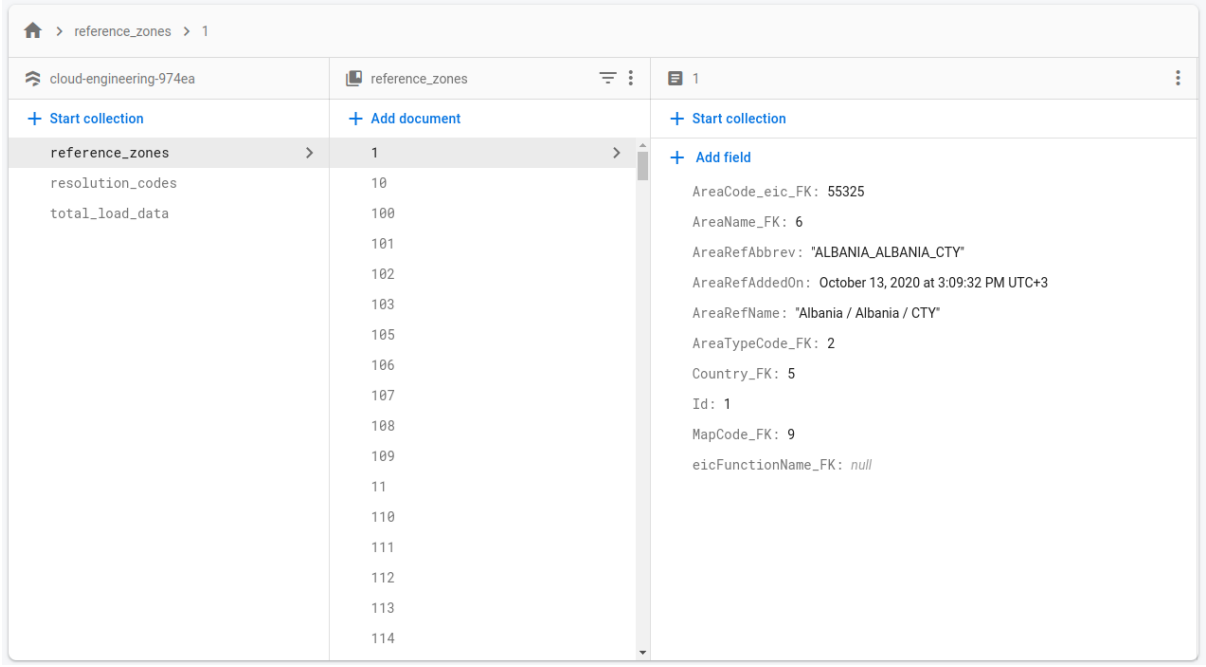


Εικόνα 116: Get Advanced Energy Data Successful response

Παρατηρούμε ότι υπάρχει 2 παραπάνω πεδία ReferenceZoneInfo και ResolutionCodeInfo, καθώς το πεδίο join είναι true.

4.3.3 Get Reference Zones

Αυτό το Endpoint γυρνάει documents από το Collection “reference_zones”:



Εικόνα 117: reference_zones collection

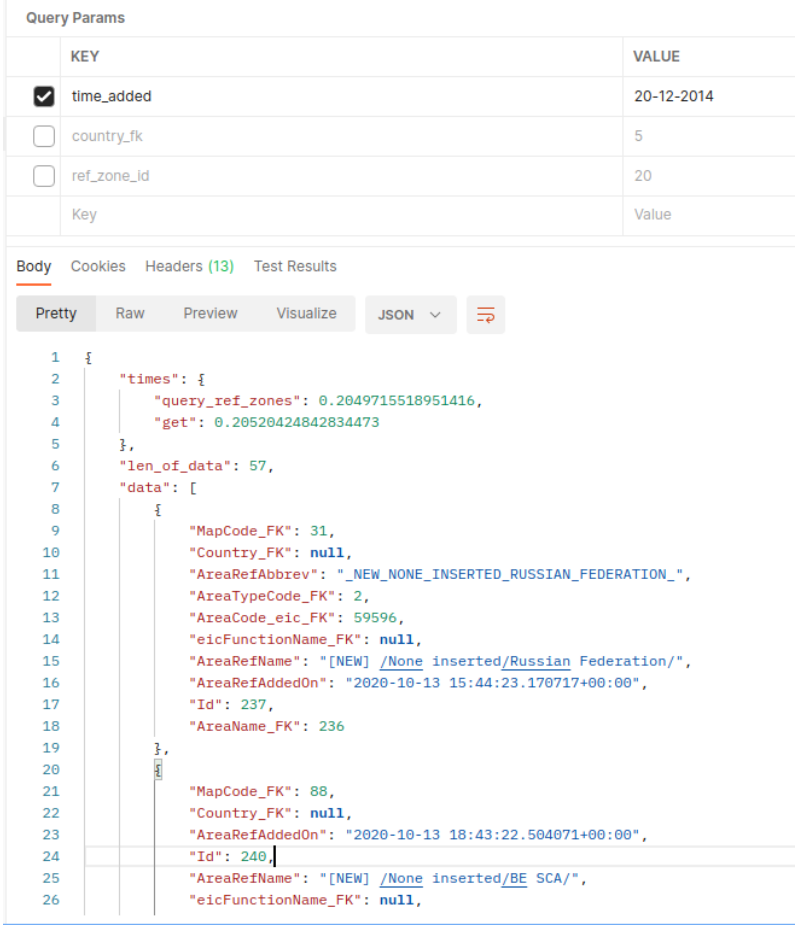
Query Parameters:

- *time_added (string):* είναι το αντίστοιχο AreaRefAddedOn στα attributes.
- *country_fk (int):* είναι το αντίστοιχο Country_FK στα attributes.
- *ref_zone_id (int):* αναφέρεται στο ίδιο το ID του document.

Logic:

1. Εάν έχει δοθεί ref_zone_id γυρνάμε το document με αυτό το id.
2. Εάν έχει δοθεί country_fk γυρνάμε όλα τα documents που έχουν αυτό το country_fk.
3. Εάν δοθεί time_added γυρνάμε όλα όσα δημιουργήθηκαν από το time_added και μετά.

Successful Response:



Εικόνα 118: Get Reference Zones Successful Response

4.3.4 Create Dummy Reference Zone

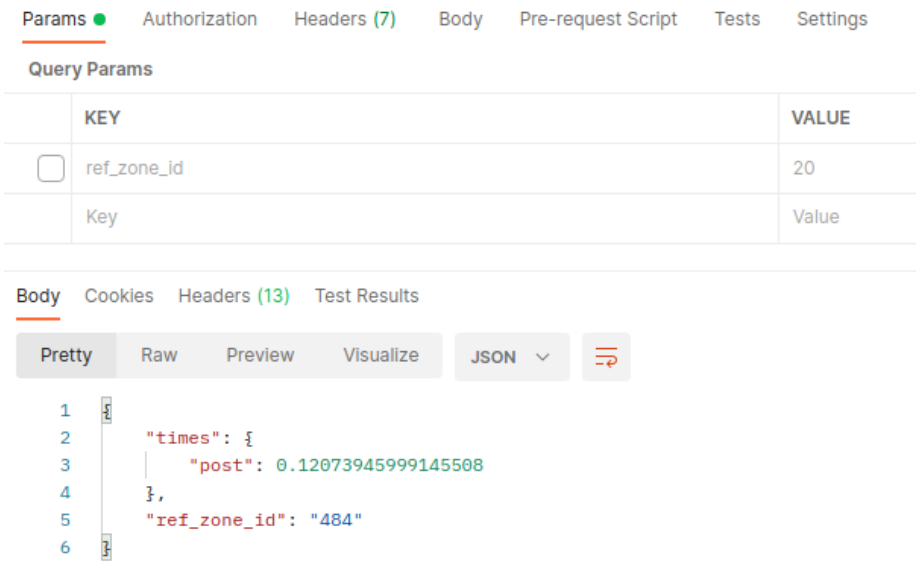
Query Parameters:

- *ref_zone_id (int):* είναι optional, και αν δοθεί δημιουργεί ένα dummy reference_zone document, εάν όχι το ID του document είναι τυχαίο.

Logic:

Δημιουργεί ένα reference_zone document.

Successful Response:



Εικόνα 119: Create Dummy Reference Zone Successful Response

Unsuccessful Response:

Εικόνα 120: Create Dummy Reference Zone Unsuccessful Response

Εδώ όπως βλέπουμε η παράμετρος ref_zone_id δεν έχει επιλεχθεί οπότε το endpoint προσπάθησε να δημιουργήσει ένα document με τυχαίο ID. Όμως, υπήρχε ήδη document με ID=734 και για αυτό επέστρεψε statusCode=400. Οπότε, λοιπόν, όταν θα κάνουμε stress tests θα δούμε αρκετά 4XX errors τα οποία θα οφείλονται σε τέτοιες καταστάσεις.

4.3.5 Remove Reference Zone

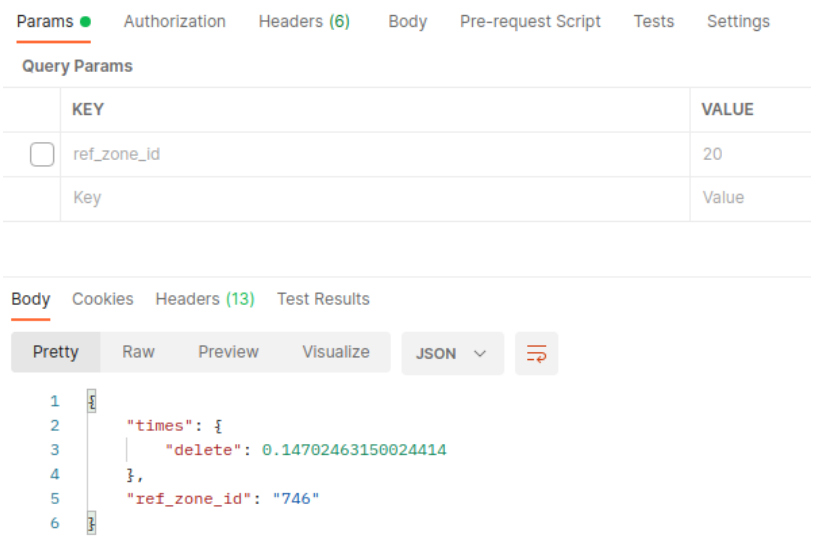
Query Parameters:

- ref_zone_id (int): είναι optional, και αν δοθεί διαγράφει το συγκεκριμένο reference_zone document, εάν όχι διαγράφει ένα τυχαίο.

Logic:

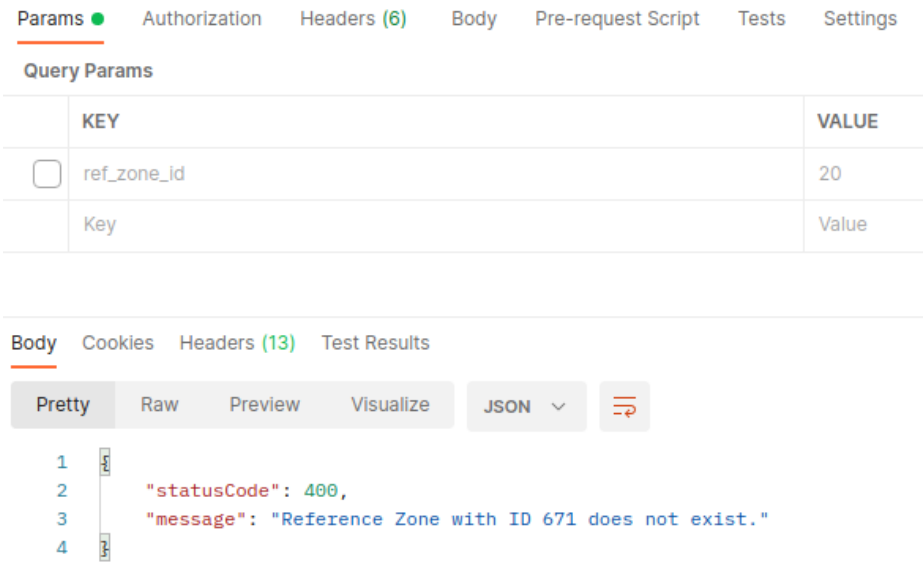
Διαγράφει ένα reference_zone document.

Successful Response:



Εικόνα 121: Remove Reference Zone Successful Response

Unsuccessful Response:



Εικόνα 122: Remove Reference Zone Unsuccessful Response

Εδώ όπως βλέπουμε η παράμετρος ref_zone_id δεν έχει επιλεχθεί οπότε το endpoint προσπάθησε να διαγράψει ένα document με τυχαίο ID. Όμως, δεν υπήρχε document με ID=671 και για αυτό επέστρεψε statusCode=400. Οπότε, λοιπόν, όταν θα κάνουμε stress tests θα δούμε αρκετά 4XX errors τα οποία θα οφείλονται σε τέτοιες καταστάσεις.

4.4 Πειράματα

Αν και στα μεμονωμένα tests παρατηρήσαμε πως το AWS Lambda είναι το πιο γρήγορο οφείλουμε να ελέγξουμε περιπτώσεις όπου τα εργαλεία που χρησιμοποιούμε ανήκουν στον ίδιο Cloud Provider. Μπορεί το Cloud Run να συνεργάζεται πιο αποδοτικά με το Firestore από ότι το AWS Lambda.

Για τα stress tests χρησιμοποιούμε ξανά το Loadium:

Εικόνα 123: Loadium tests overview

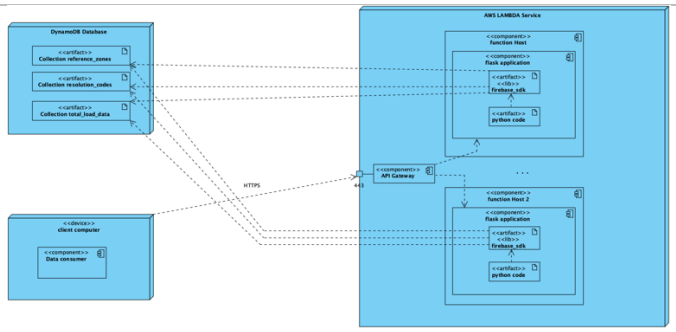
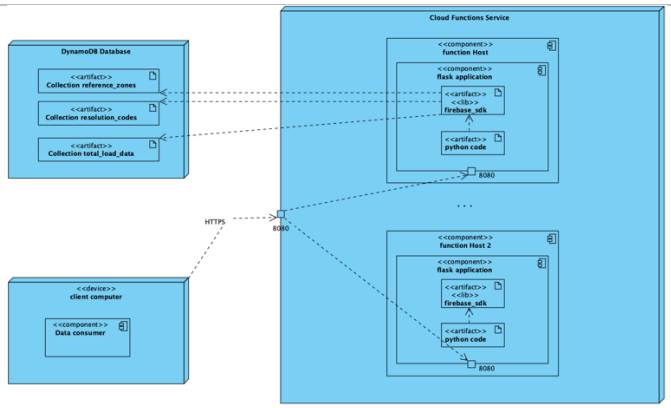
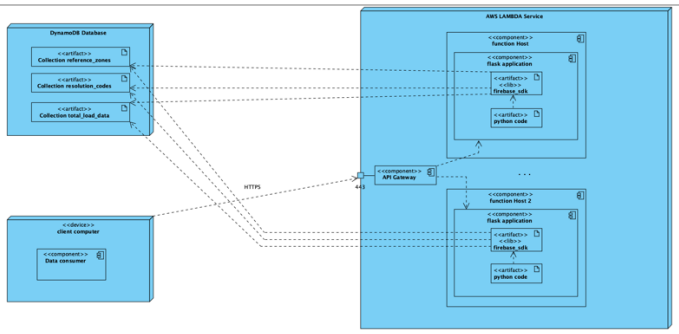
το οποίο με τυχαίο τρόπο θα κάνει τα παραπάνω requests με απώτερο σκοπό να προσομοιώσουμε την κίνηση 250 χρηστών σε διάρκεια 2 λεπτών που σημαίνει περίπου 4000 hits κατά μέσο όρο ανά endpoint.

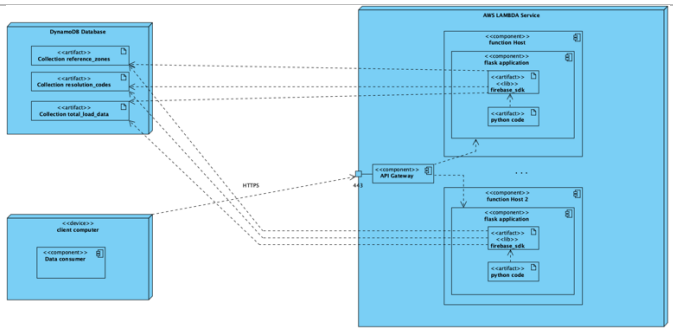
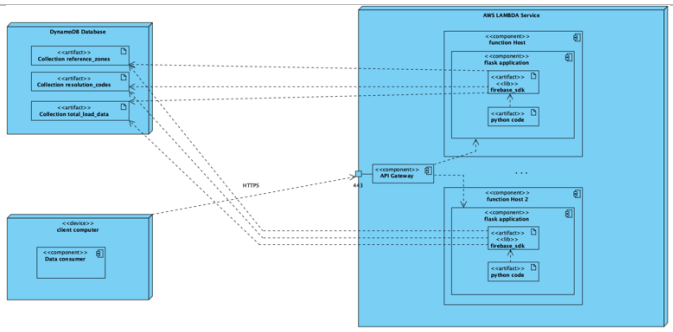
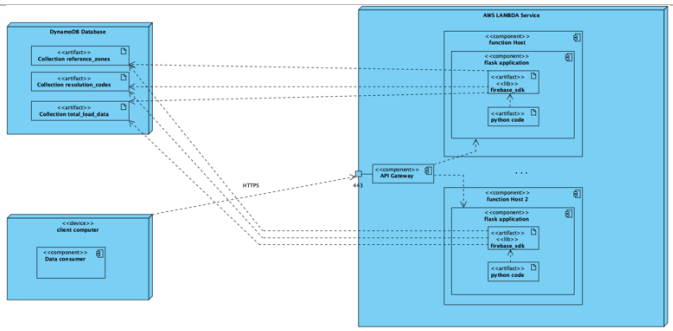
Παρακάτω φαίνεται ένας συγκεντρωτικός πίνακας των deployments που θα δοκιμάσουμε. Για κάθε σενάριο έχουμε σημειώσει τι ακριβώς έχουμε χρησιμοποιήσει. Για παράδειγμα:

- Στο πρώτο σενάριο για βάση δεδομένων θα χρησιμοποιήσουμε το Cloud Firestore και για τα APIs θα χρησιμοποιήσουμε το Cloud Run έχοντας ορίσει max-concurrency ίσο με 4.
- Στο τέταρτο σενάριο για βάση δεδομένων έχουμε το AWS DynamoDB και για τα APIs χρησιμοποιούμε AWS Lamba. Αξίζει να σημειώσουμε, βέβαια, ότι εφόσον χρησιμοποιούμε AWS Lambda πρέπει να εντάξουμε στο σύστημα μας και ένα API Gateway καθώς το AWS Lambda, σε αντίθεση με το Cloud Run και Cloud Functions, δεν δημιουργεί public URL. Περισσότερες λεπτομέρειες στο εκάστοτε UML Deployment Diagram.

Scenario	Cloud Firestore	AWS DynamoDB	AWS Lambda	AWS API Gateway	Cloud Functions	Cloud Run	Cloud Run (concurrency)
4.4.1	•					•	4
4.4.2	•				•		
4.4.3	•		•	•			
4.4.4		•	•	•			
4.4.5		•				•	80
4.4.6		•				•	4

Εικόνα 124: Συγκεντρωτικός πίνακας όλων των deployment scenarios

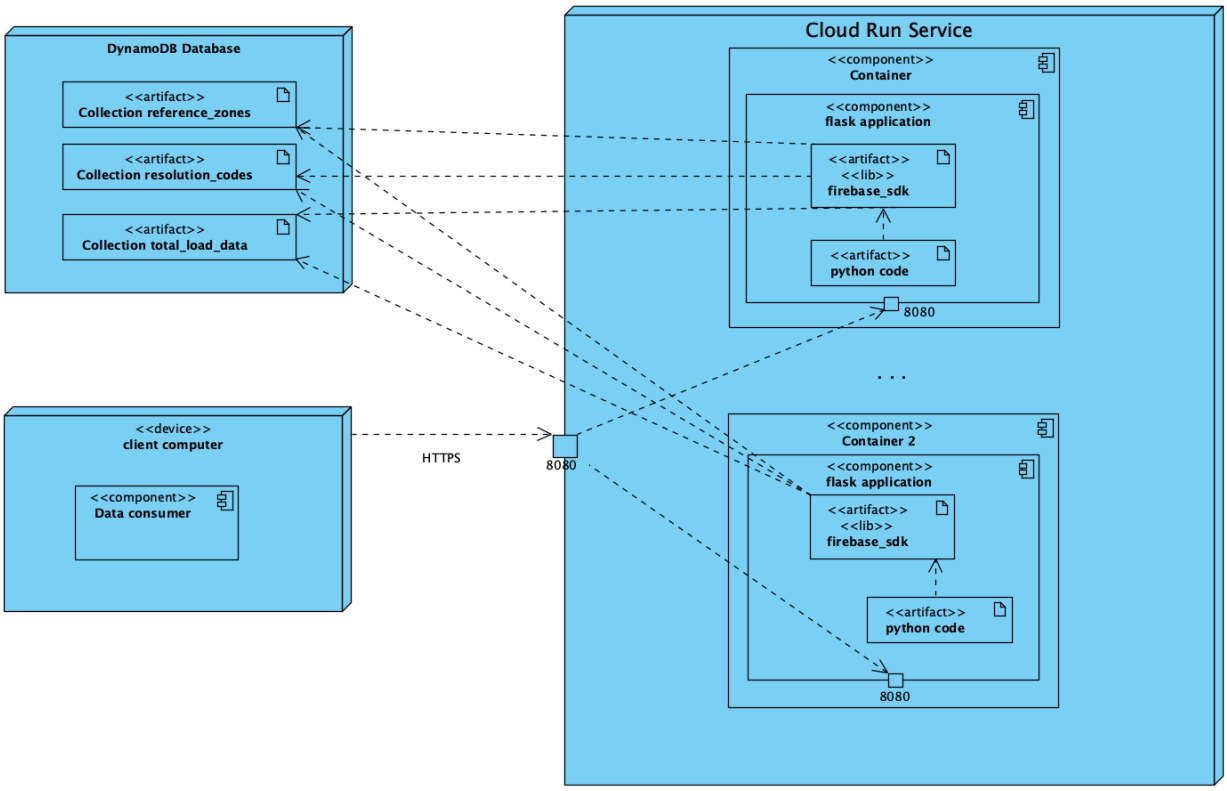
Scenario	UML Deployment Diagram
4.4.1	
4.4.2	
4.4.3	

4.4.4	
4.4.5	
4.4.6	

Εικόνα 125: Συγκεντρωτικός πίνακας όλων των UML Deployment Diagrams

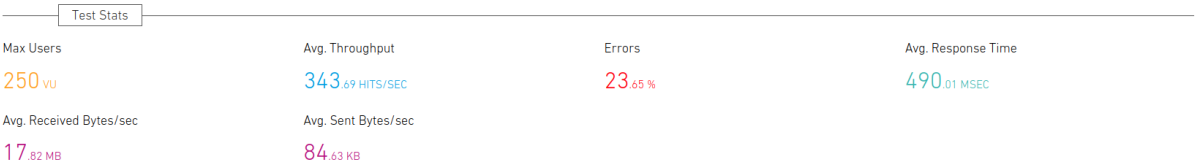
4.4.1 Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4

Τα δεδομένα είναι σε Cloud Firestore και τα APIs είναι deployed σε Google Cloud Run, έχοντας ορίσει το max-Concurrency ίσο με 4, ως ένα container ακολουθώντας το μονολιθικό μοντέλο. Το αντίστοιχο repository: [https://github.com/Cloud-Engineering-Softlab-Project/Firebase-CloudRun-single]

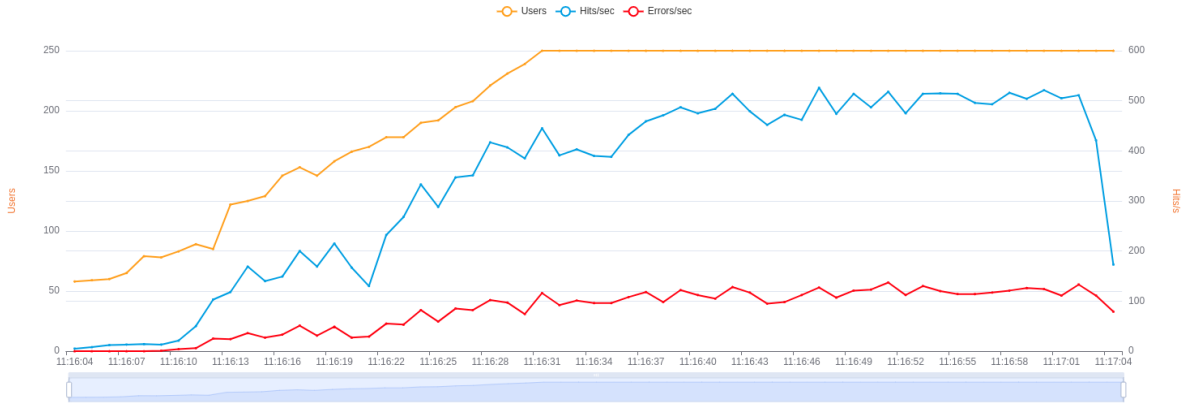


Εικόνα 126: UML Deployment Diagram of "Cloud Firestore - Cloud Run"

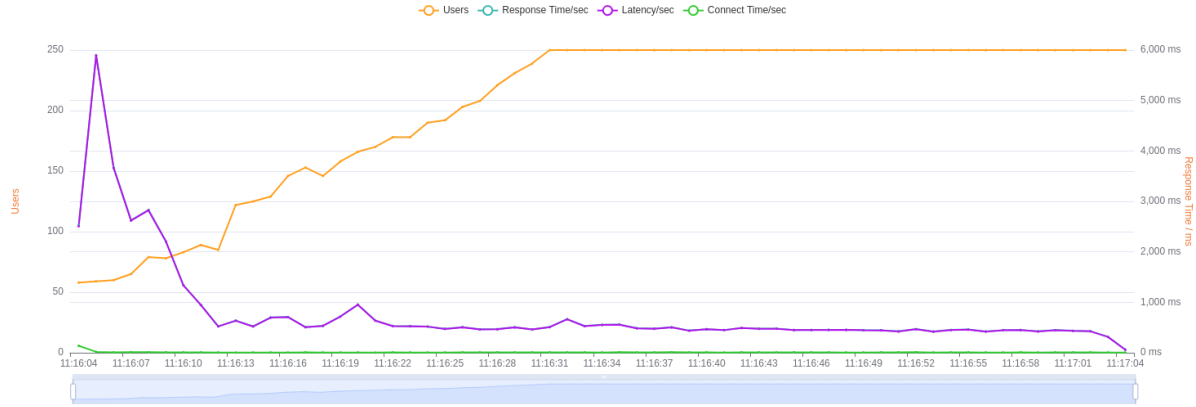
4.4.1.1 Διαγράμματα



Εικόνα 127: Test stats of “Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4”



Εικόνα 128: Hits & Errors of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"



Εικόνα 129: Timeline of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
20605	490.01 MSEC	6385.00 MSEC	30.00 MSEC
Percentage Error	**Total Throughput**	**Avg. Connect Time**	**Avg. Latency**
23.65 %	357.50 RPS	7.39 MSEC	486.46 MSEC
Total Error Hits			
4873			

Εικόνα 130: Summary of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	4221	70.41	1.20
"energy-data-simple"	4113	70.63	0.94
"get-ref-zone"	4104	71.44	0.10
"add-ref-zone"	4091	71.40	0.09
"delete-ref-zone"	4076	73.63	0.09

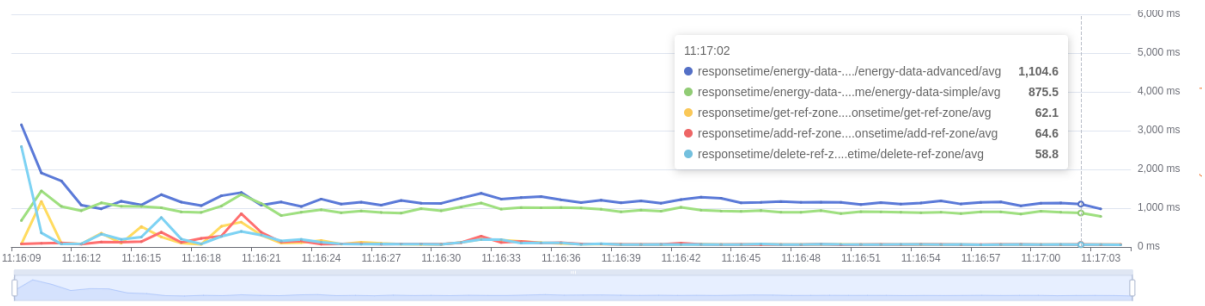
Εικόνα 131: APIs metrics (1) of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	6.38	0.79	0
"energy-data-simple"	5.19	0.63	0
"get-ref-zone"	5.50	0.03	0
"add-ref-zone"	4.17	0.03	2435
"delete-ref-zone"	5.56	0.03	2438

Εικόνα 132: APIs metrics (2) of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	4221
energy-data-simple	200	OK	4113
get-ref-zone	200	OK	4104
add-ref-zone	400	Bad Request	2435
add-ref-zone	200	OK	1656
delete-ref-zone	400	Bad Request	2438
delete-ref-zone	200	OK	1638

Εικόνα 133: Response codes of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"



Εικόνα 134: Average response time graph of "Cloud Firestore - Cloud Run (monolithic) – max-concurrency=4"

4.4.1.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 490 ms
- Average Throughput = 343 hits/sec
- Total Hits = 20605

Παρατηρήσεις:

- Έχουν υπάρξει 4873 Total Error Hits και αυτά οφείλονται στα 2 endpoints που αναφέραμε και προηγουμένως. Συγκεκριμένα είχαμε:
    - 2435 errors στο add_ref_zone και
    - 2438 errors στο delete_ref_zone

- Στην αρχή του test έχουμε αυξημένο response time σε όλα τα endpoints, όπως βλέπουμε και στο τελευταίο διάγραμμα, αλλά κυρίως φαίνεται και στο δεύτερο διάγραμμα με την μωβ γραμμή. Αυτό συμβαίνει λόγω των cold starts και για αυτόν τον λόγο μετά από λίγο όλες οι μετρικές σταθεροποιούνται

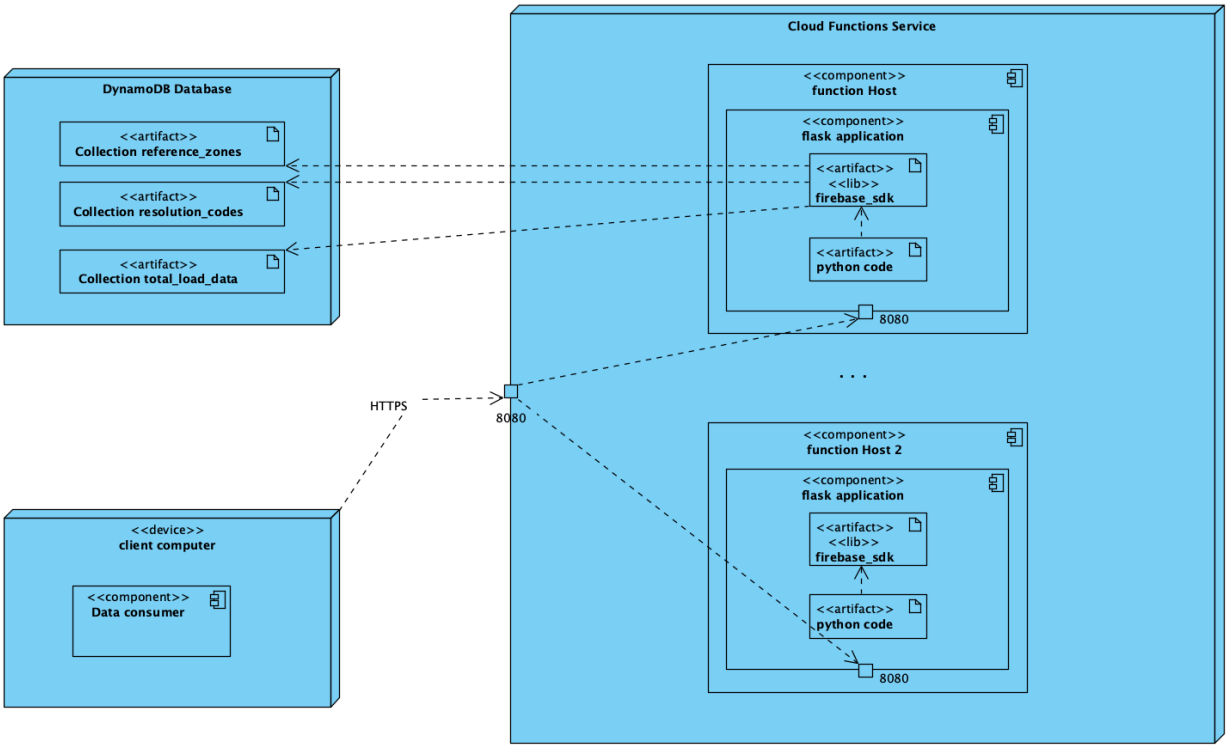
- Επιβεβαιώνεται το παράδοξο του Firestore από την σταθερή διαφορά μεταξύ Simple & Advanced Energy Data. Αν και τα 2 endpoints καταλήγουν να εκτελούν queries στο ίδιο collection με τα ίδια indexes και με τον ίδιο όγκο δεδομένων, είναι ξεκάθαρο πως το Simple Energy Data είναι σταθερά πιο γρήγορο από το Advanced Energy Data, και αυτό συμβαίνει γιατί το πρώτο επιστρέφει λιγότερα δεδομένα από ότι το δεύτερο.

Αυτή η παρατήρηση επιβεβαιώνει πως στο Firestore, χάρη τα indexes, δεν παίζει ρόλο ο όγκος των αποθηκευμένων δεδομένων αλλά ο όγκος των δεδομένων που επιστρέφουν τα Queries.

- Για τον ίδιο λόγο, τα endpoints που αναφέρονται στα References Zones είναι πολύ πιο γρήγορα.

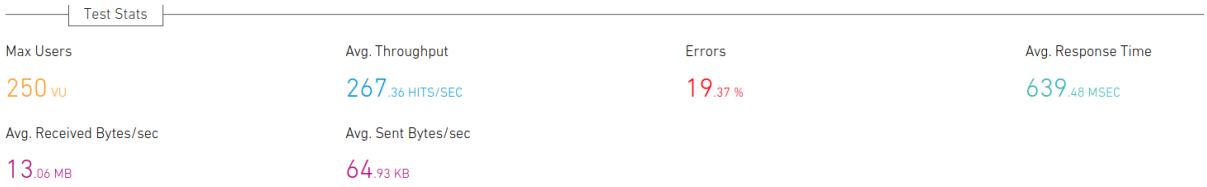
4.4.2 Cloud Firestore - Cloud Functions (polylithic)

Τα δεδομένα είναι στο Cloud Firestore και τα APIs είναι deployed σε Google Cloud Functions ακολουθώντας το πολυλιθικό μοντέλο. Το αντίστοιχο repository: [https://github.com/Cloud-Engineering-Softlab-Project/Firebase-CloudFunctions-poly]

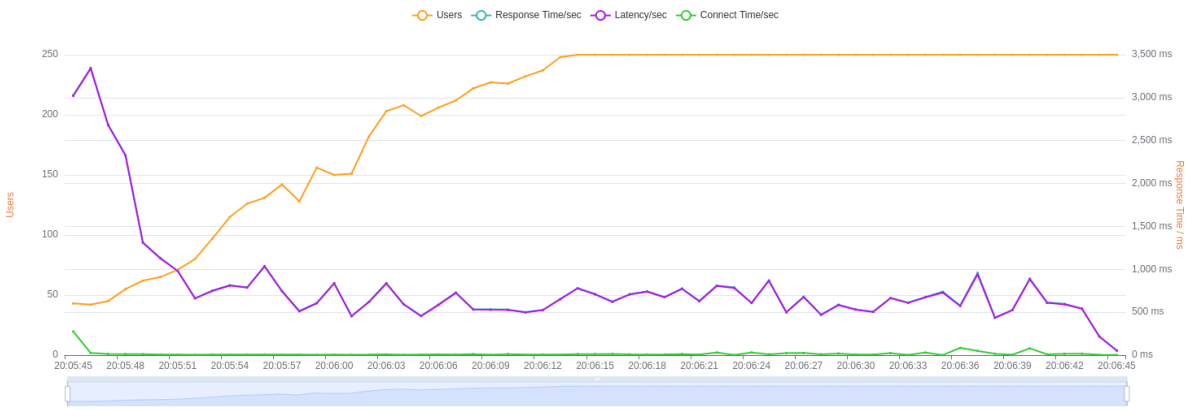


Εικόνα 135: UML Deployment Diagram of "Cloud Firestore - Cloud Functions"

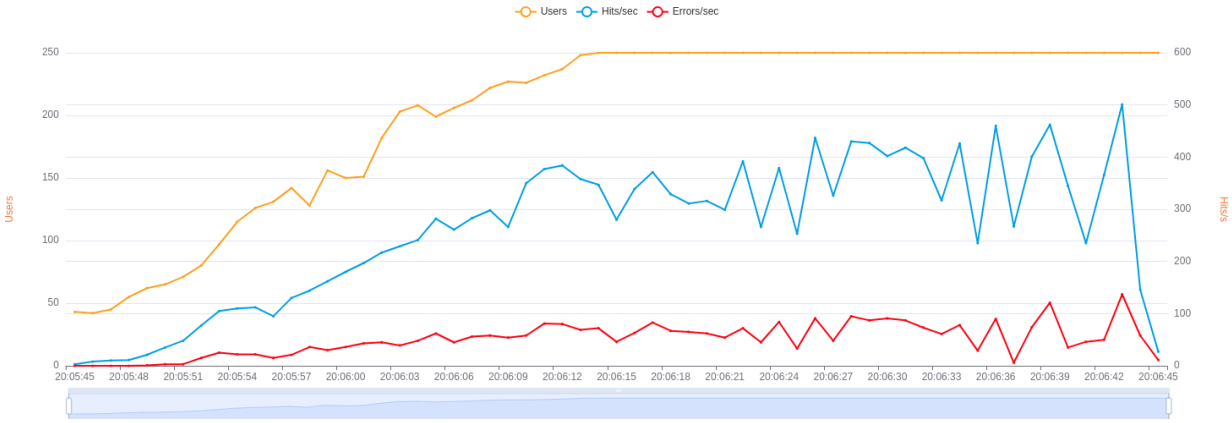
4.4.2.1 Διαγράμματα



Εικόνα 136: Test stats of "Cloud Firestore - Cloud Functions (polylithic)"



Εικόνα 137: Timeline of "Cloud Firestore - Cloud Functions (polylithic)"



Εικόνα 138: Hits & Errors of "Cloud Firestore - Cloud Functions (polylithic)"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
15902	639.48 MSEC	7066.00 MSEC	30.00 MSEC
Percentage Error	**Total Throughput**	**Avg. Connect Time**	**Avg. Latency**
19.37 %	278.83 RPS	16.20 MSEC	635.49 MSEC
Total Error Hits			
3081			

Εικόνα 139: Summary of "Cloud Firestore - Cloud Functions (polylithic)"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	3273	55.05	1.24
"energy-data-simple"	3202	55.27	0.95
"get-ref-zone"	3192	56.90	0.12
"add-ref-zone"	3164	56.61	0.29
"delete-ref-zone"	3071	55.00	0.57

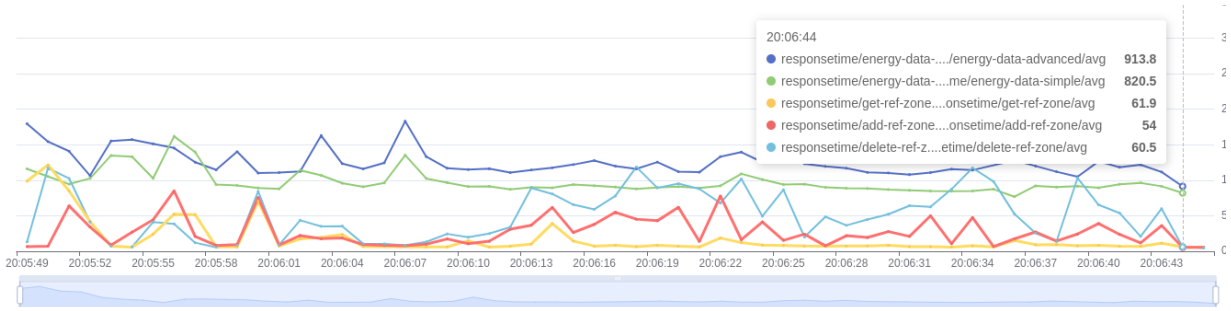
Εικόνα 140: APIs metrics (1) of "Cloud Firestore - Cloud Functions (polylithic)"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	7.07	0.85	0
"energy-data-simple"	5.44	0.65	0
"get-ref-zone"	4.38	0.04	0
"add-ref-zone"	4.20	0.03	2149
"delete-ref-zone"	4.13	0.03	932

Εικόνα 141: APIs metrics (2) of "Cloud Firestore - Cloud Functions (polylithic)"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	3273
energy-data-simple	200	OK	3202
get-ref-zone	200	OK	3192
add-ref-zone	400	Bad Request	2149
add-ref-zone	200	OK	1015
delete-ref-zone	200	OK	2139
delete-ref-zone	400	Bad Request	932

Εικόνα 142: Response codes of "Cloud Firestore - Cloud Functions (polylithic)"



Εικόνα 143: Average response time graph of "Cloud Firestore - Cloud Functions (polylithic)"

4.4.2.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 639 ms
- Average Throughput = 267 hits/sec
- Total Hits = 15902

Παρατηρήσεις:

- Παρατηρούμε ότι τα Total Hits μειώθηκαν. Αυτό συμβαίνει γιατί το test έχει αυστηρή διάρκεια 2 λεπτά και άμα το σύστημα έχει χαμηλό Throughput θα έχει και λιγότερα Total Hits.

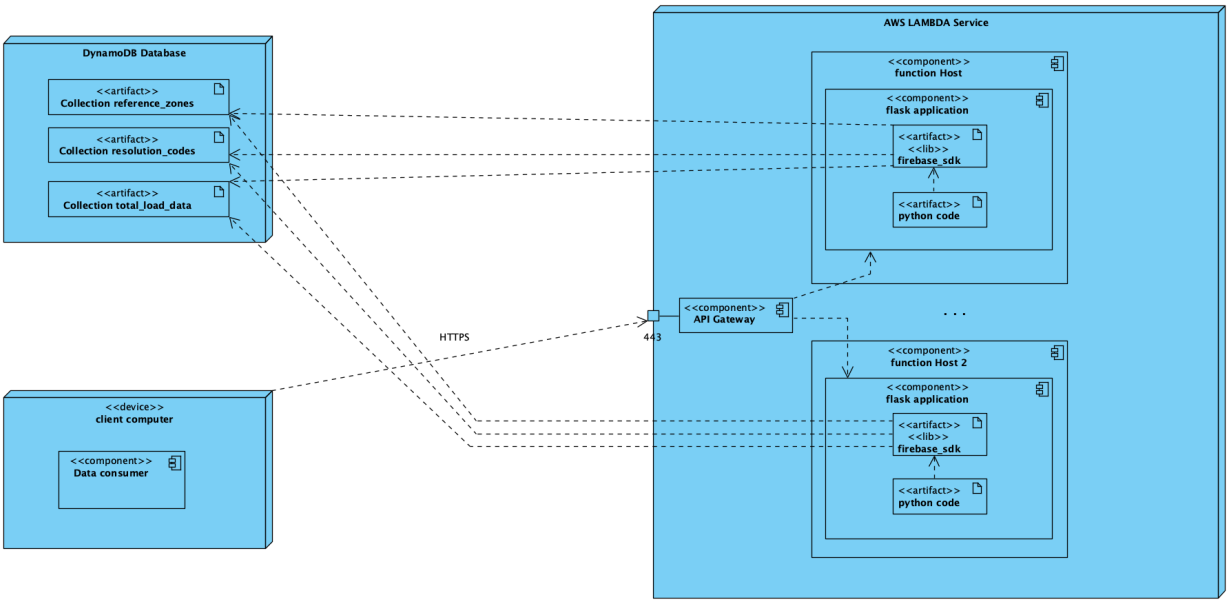
- Το σύστημα είναι διακριτά πιο αργό από το προηγούμενο. Αυτό το περιμέναμε γιατί στα μεμονωμένα tests το Cloud Run ήταν πολύ πιο γρήγορο από τα Cloud Functions και τώρα επιβεβαιώνεται για άλλη μια φορά καθώς η βάση δεδομένων είναι ακριβώς η ίδια με πριν.

- Παρατηρούμε ότι δεν υπάρχει η σταθερότητα στους χρόνους που υπήρχε στο προηγούμενο test. Στο προηγούμενο test όλα τα endpoints, μετά τα cold starts, είχαν σταθεροποιηθεί ενώ αυτό δεν ισχύει τώρα όπως βλέπουμε στο τελευταίο διάγραμμα. Αυτό συμβαίνει εξαιτίας του πολυλιθκού μοντέλου.

- Τα errors είναι στα ίδια endpoints και με πριν για τον ίδιο ακριβώς λόγο.

4.4.3 Cloud Firestore - AWS Lambda (monolithic)

Τα δεδομένα είναι στο Cloud Firestore και τα APIs είναι deployed σε AWS Lambda ως μία Lambda ακολουθώντας το μονολιθικό μοντέλο. Το αντίστοιχο repository: [https://github.com/Cloud-Engineering-Softlab-Project/Firebase-AWS-single]



Εικόνα 144: UML Deployment Diagram of "Cloud Firestore - AWS Lambda"

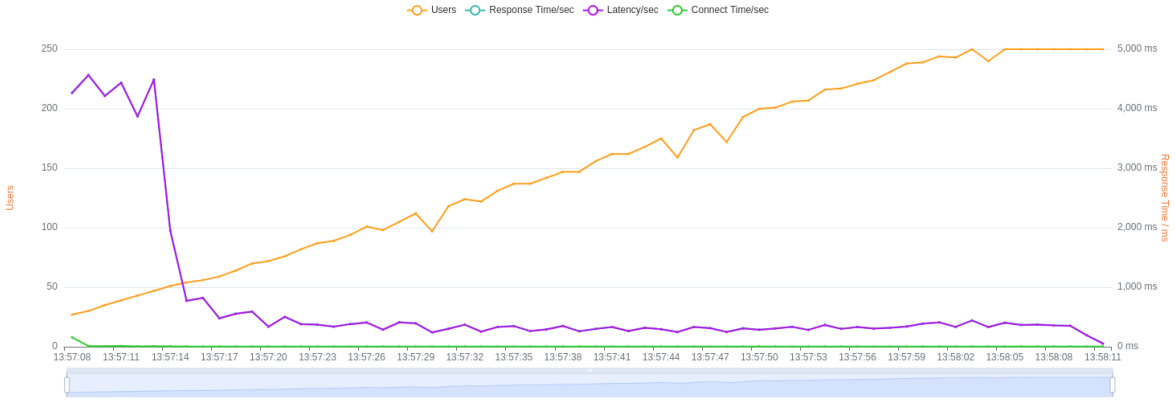
4.4.3.1 Διαγράμματα

Test Stats

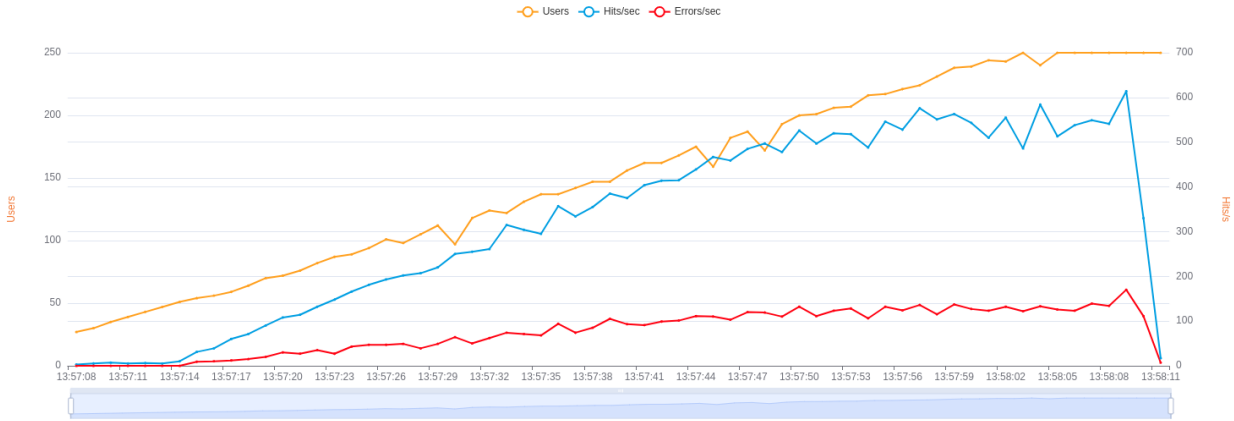
Max Users	Avg. Throughput	Errors	Avg. Response Time
250 VU	329.58 HITS/SEC	24.00 %	339.24 MSEC

Avg. Received Bytes/sec	Avg. Sent Bytes/sec
17.11 MB	80.53 KB

Εικόνα 145: Test stats of "Cloud Firestore - AWS Lambda (monolithic)"



Εικόνα 146: Timeline of "Cloud Firestore - AWS Lambda (monolithic)"



Εικόνα 147: Hits & Errors of "Cloud Firestore - AWS Lambda (monolithic)"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
20640	339.24 MSEC	6098.00 MSEC	28.00 MSEC

Percentage Error	Total Throughput	Avg. Connect Time	Avg. Latency
24.00 %	349.61 RPS	3.03 MSEC	337.40 MSEC

Total Error Hits
4954

Εικόνα 148: Summary of "Cloud Firestore - AWS Lambda (monolithic)"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	6.10	0.57	0
"energy-data-simple"	5.70	0.46	0
"get-ref-zone"	4.91	0.03	0
"add-ref-zone"	5.08	0.03	2502
"delete-ref-zone"	5.08	0.03	2452

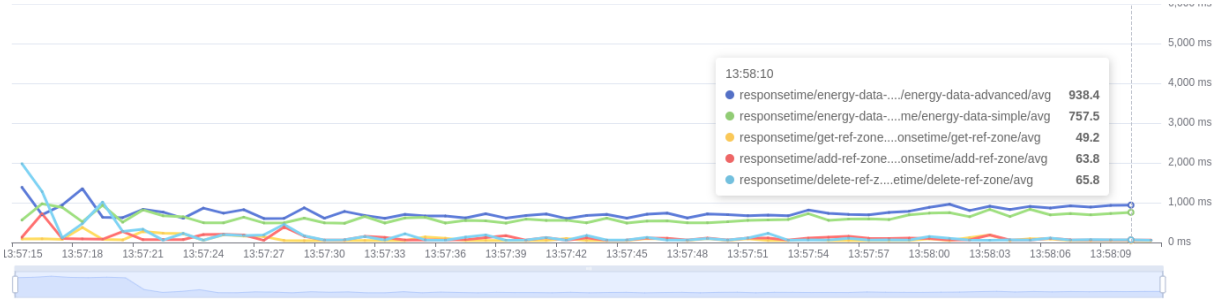
Εικόνα 149: APIs metrics (1) of "Cloud Firestore - AWS Lambda (monolithic)"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	4232	67.58	0.78
"energy-data-simple"	4128	67.01	0.62
"get-ref-zone"	4111	69.08	0.08
"add-ref-zone"	4090	72.95	0.10
"delete-ref-zone"	4079	73.00	0.11

Εικόνα 150: APIs metrics (2) of "Cloud Firestore - AWS Lambda (monolithic)"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	4232
energy-data-simple	200	OK	4128
get-ref-zone	200	OK	4111
add-ref-zone	400	Bad Request	2502
add-ref-zone	200	OK	1588
delete-ref-zone	400	Bad Request	2452
delete-ref-zone	200	OK	1627

Εικόνα 151: Response codes of "Cloud Firestore - AWS Lambda (monolithic)"



Εικόνα 152: Average response time graph of "Cloud Firestore - AWS Lambda (monolithic)"

4.4.3.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 339 ms
- Average Throughput = 329 hits/sec
- Total Hits = 20640

Παρατηρήσεις:

- Το σύστημα είναι πολύ πιο γρήγορο καθώς έχει Average Response Time περίπου το μισό σε σχέση με το προηγούμενο. Κάτι τέτοιο το περιμέναμε, καθώς το AWS Lambda είχε αποδειχθεί πολύ πιο γρήγορο και από το Cloud Run και από το Cloud Functions.

- Παρατηρούμε πολύ μεγάλη σταθερότητα κάτι που οφείλεται και στο μονολιθικό μοντέλο αλλά και στις υψηλές αποδόσεις του Lambda. Επίσης, τα cold starts διαρκούν λιγότερα και έχουν μικρότερα Latencies.

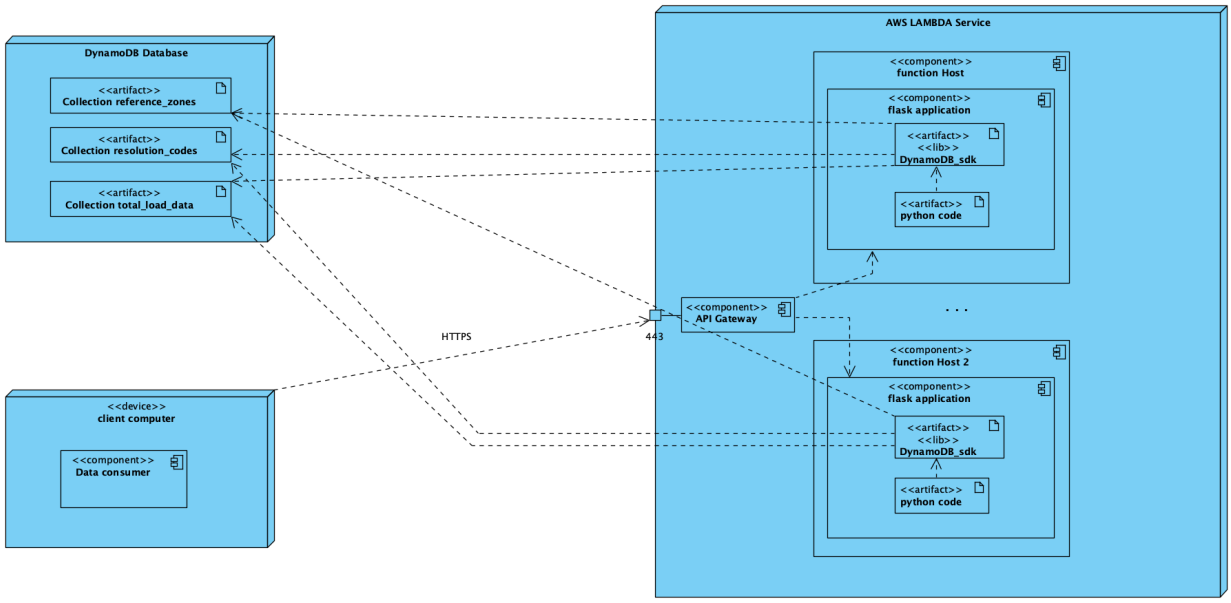
- Μεγαλύτερο Throughput ισοδυναμεί και σε περισσότερα Total Hits.

- Τα errors παραμένουν στα ίδια επίπεδα, οι διαφορές μεταξύ των endpoints παραμένουν και εδώ, αλλά σε πολύ μικρότερη κλίμακα.

- Είναι ξεκάθαρο, λοιπόν, πως σε αυτό το σύστημα το Firestore αποτελεί bottleneck καθώς τα endpoints έχουν χωριστεί σε 2 βασικές ομάδες, αυτά που αφορούν το collection total_load_data και σε αυτά που αφορούν το collection reference_zones.

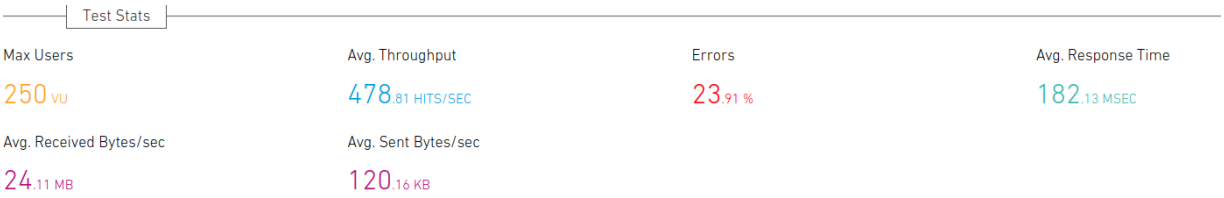
4.4.4 AWS DynamoDB - AWS Lambda (monolithic)

Τα δεδομένα είναι σε AWS DynamoDB και τα APIs είναι deployed σε AWS Lambda ως μία Lambda ακολουθώντας το μονολιθικό μοντέλο. Το αντίστοιχο repository: [https://github.com/Cloud-Engineering-Softlab-Project/DynamoDB-AWS-single]

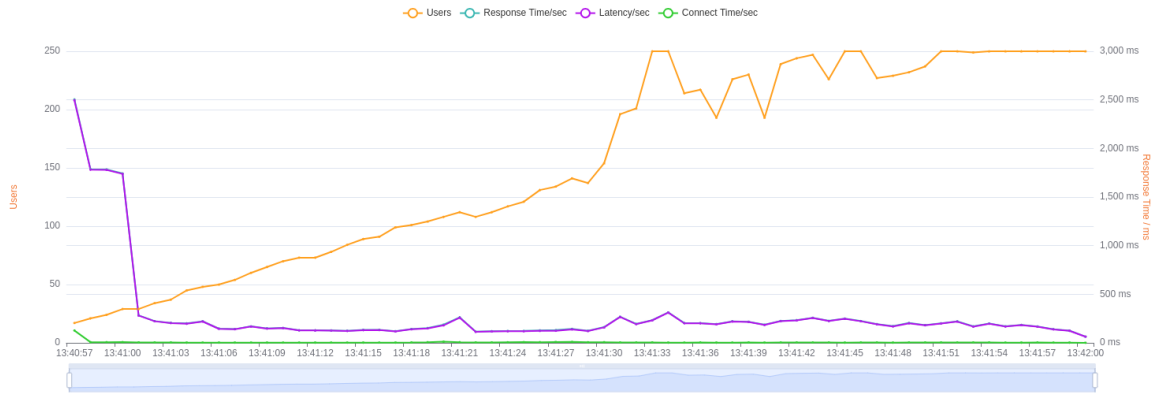


Εικόνα 153: UML Deployment Diagram of "AWS Lambda - AWS Lambda"

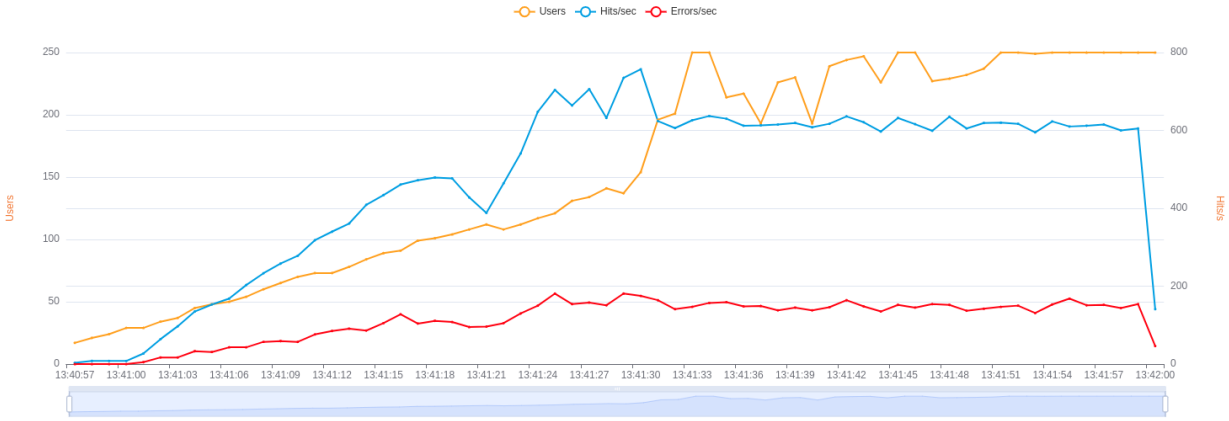
4.4.4.1 Διαγράμματα



Εικόνα 154: Test stats of "AWS DynamoDB - AWS Lambda (monolithic)"



Εικόνα 155: Timeline of "AWS DynamoDB - AWS Lambda (monolithic)"



Εικόνα 156: Hits & Errors of "AWS DynamoDB - AWS Lambda (monolithic)"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
30066	182.13 MSEC	25.94 SEC	20.00 MSEC
Percentage Error	**Total Throughput**	**Avg. Connect Time**	**Avg. Latency**
23.91 %	498.07 RPS	4.45 MSEC	179.32 MSEC
Total Error Hits			
7189			

Εικόνα 157: Summary of "AWS DynamoDB - AWS Lambda (monolithic)"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	6052	96.40	0.51
"energy-data-simple"	6022	96.90	0.23
"get-ref-zone"	6009	100.37	0.05
"add-ref-zone"	5998	102.19	0.06
"delete-ref-zone"	5985	102.20	0.06

Εικόνα 158: APIs metrics (1) of "AWS DynamoDB - AWS Lambda (monolithic)"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	25.94	0.23	2
"energy-data-simple"	3.39	0.18	0
"get-ref-zone"	3.17	0.02	0
"add-ref-zone"	3.12	0.02	3592
"delete-ref-zone"	3.22	0.02	3595

Εικόνα 159: APIs metrics (2) of "AWS DynamoDB - AWS Lambda (monolithic)"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	6050
energy-data-advanced	500	Internal Server Error	2
energy-data-simple	200	OK	6022
get-ref-zone	200	OK	6009
add-ref-zone	400	Bad Request	3592
add-ref-zone	200	OK	2406
delete-ref-zone	400	Bad Request	3595
delete-ref-zone	200	OK	2390

Εικόνα 160: Response codes of "AWS DynamoDB - AWS Lambda (monolithic)"

Εικόνα 161: Average response time graph of "AWS DynamoDB - AWS Lambda (monolithic)"

4.4.4.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 182 ms
- Average Throughput = 478 hits/sec
- Total Hits = 30066

Παρατηρήσεις:

- Ο συγκεκριμένος συνδυασμός είναι ο πιο γρήγορος που έχουμε δει μέχρι στιγμής φτάνοντας να είναι περίπου 2 φορές πιο γρήγορος από τον προηγούμενο. Δεδομένου, λοιπόν, πως το AWS Lambda από μόνο του είναι πιο γρήγορο από τα υπόλοιπα, περιμένουμε αυτός ο συνδυασμός να καταλήξει να είναι ο πιο αποδοτικός από όλους.

- Είναι ξεκάθαρο πως το DynamoDB είναι αρκετά πιο γρήγορο από το Firestore, καθώς το average response time όλων των endpoints μειώθηκε δραματικά πολύ σε σχέση με το προηγούμενο που ήταν το πιο γρήγορο του Firestore.

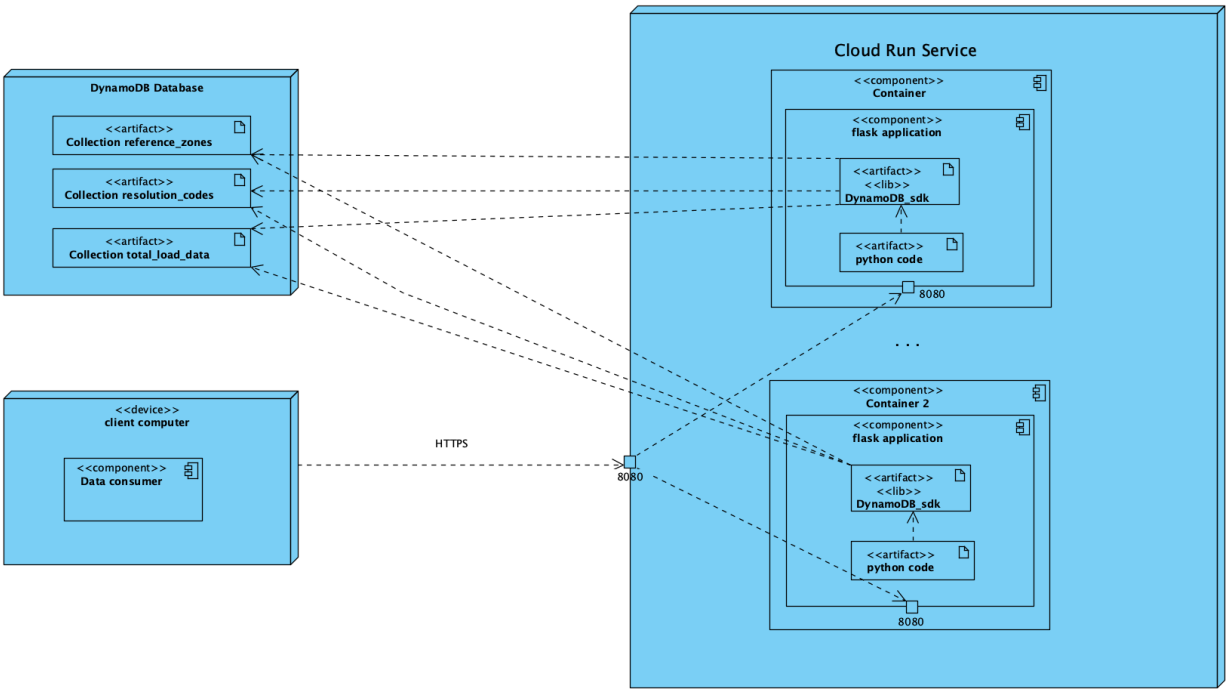
- Παρατηρούμε πολύ μεγάλη σταθερότητα και οριακά μηδενικά cold starts κάτι που δηλώνει πως και το Firestore συνέβαλε στα προηγούμενα cold starts.

- Τέλος, παρατηρούμε 2 internal server errors που δεν είχαμε ξαναδεί μέχρι στιγμής. Αυτό συμβαίνει, διότι το έχουμε ορίσει στις Lambda ως maximum concurrency ίσο με 100 και εδώ βλέπουμε για πρώτη φορά τα Requests Per Second να αγγίζουν το 100.

Αυτό συμβαίνει λόγω υψηλού Throughput του DynamoDB και κατ' επέκταση όλου του συστήματος, με αποτέλεσμα το Loadium να κάνει πολλά περισσότερα requests μέσα στα 2 λεπτά. Για αυτόν τον λόγο, έχουμε περίπου 30000 total hits, ενώ στις άλλες περιπτώσεις είχαμε περίπου 15000 - 20000.

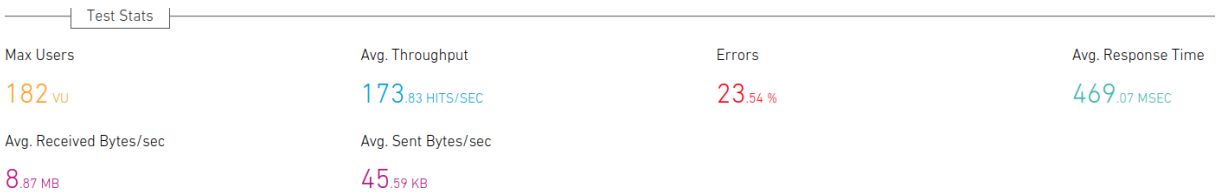
4.4.5 AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80

Τα δεδομένα είναι σε AWS DynamoDB και τα APIs είναι deployed σε Google Cloud Run, έχοντας ορίσει το max-concurrency ίσο με 80, ως ένα container ακολουθώντας το μονολιθικό μοντέλο. Το αντίστοιχο repository:
[https://github.com/Cloud-Engineering-Softlab-Project/DynamoDB-CloudRun-single]

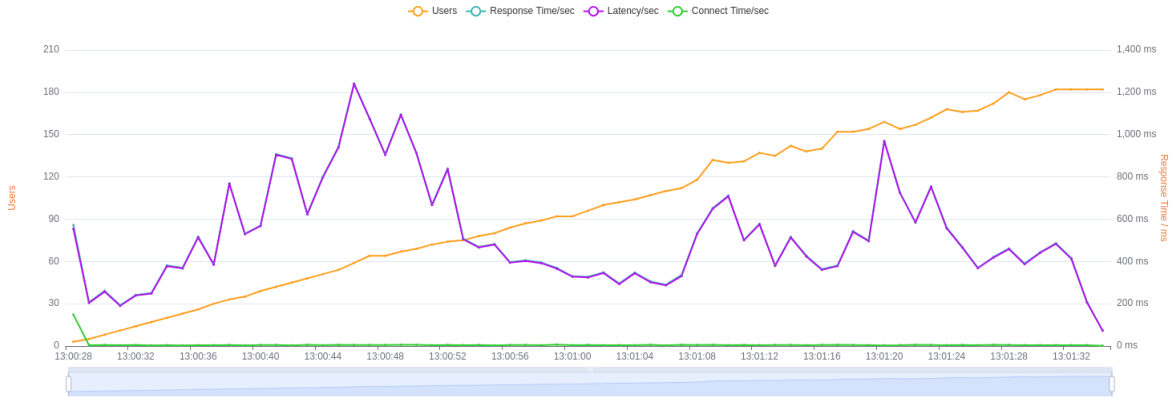


Εικόνα 162: UML Deployment Diagram of "AWS DynamoDB - Cloud Run"

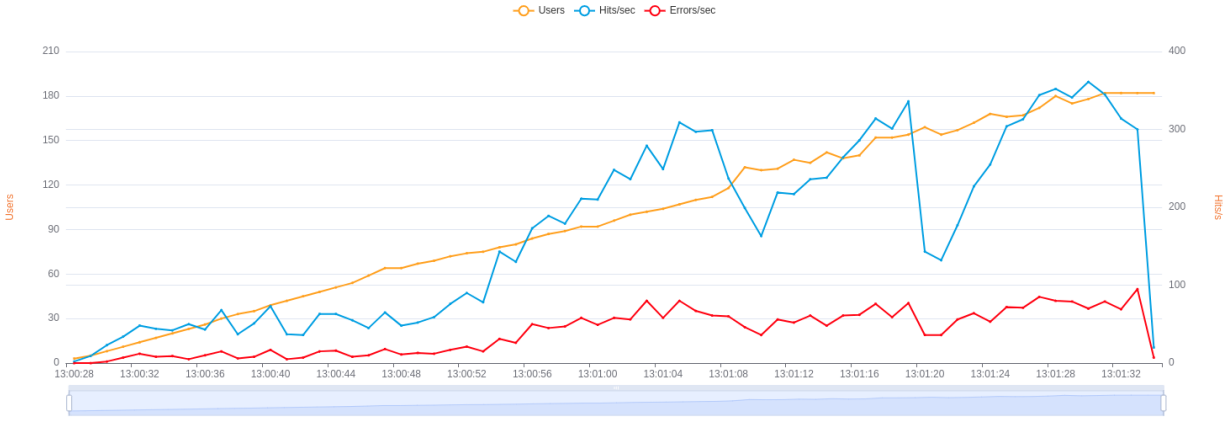
4.4.5.1 Διαγράμματα



Εικόνα 163: Test stats of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"



Εικόνα 164: Timeline of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"



Εικόνα 165: Hits & Errors of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
11381	469.07 MSEC	6407.00 MSEC	40.00 MSEC
Percentage Error	**Total Throughput**	**Avg. Connect Time**	**Avg. Latency**
23.54 %	176.16 RPS	4.65 MSEC	466.18 MSEC
Total Error Hits			
2679			

Εικόνα 166: Summary of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	2336	35.69	0.77
"energy-data-simple"	2291	35.39	0.64
"get-ref-zone"	2275	35.30	0.31
"add-ref-zone"	2257	35.13	0.31
"delete-ref-zone"	2222	34.65	0.30

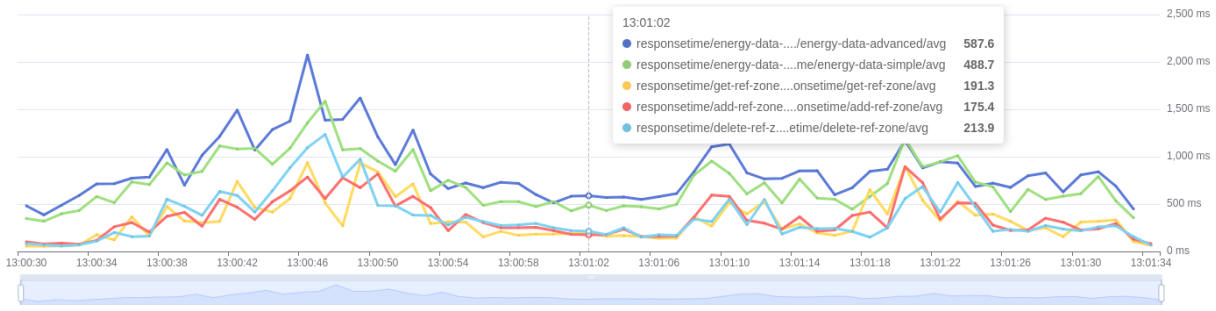
Εικόνα 167: APIs metrics (1) of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	6.41	0.27	0
"energy-data-simple"	5.58	0.21	0
"get-ref-zone"	5.61	0.04	0
"add-ref-zone"	6.00	0.04	1346
"delete-ref-zone"	3.83	0.04	1333

Εικόνα 168: APIs metrics (2) of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	2336
energy-data-simple	200	OK	2291
get-ref-zone	200	OK	2275
add-ref-zone	400	Bad Request	1346
add-ref-zone	200	OK	911
delete-ref-zone	400	Bad Request	1333
delete-ref-zone	200	OK	889

Εικόνα 169: Response codes of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"



Εικόνα 170: Average response time graph of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=80"

4.4.5.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 469 ms
- Average Throughput = 173 hits/sec
- Total Hits = 11381

Παρατηρήσεις:

- Εφόσον είδαμε πολύ υψηλές αποδόσεις από το DynamoDB τολμήσαμε να δοκιμάσουμε το Cloud Run Service με concurrency ίσο με 80 αντί για 4 που είχαμε ορίσει στο test με το Firestore. Από ότι φαίνεται αυτή η αλλαγή της παραμέτρου επηρέασε το σύστημα μας περισσότερο από ότι νομίζαμε. Καθώς πλέον έγινε bottleneck το Cloud Run και παρατηρούμε γενικά χαμηλότερες αποδόσεις σε σχέση με το Firestore - CloudRun (max-conc=80).

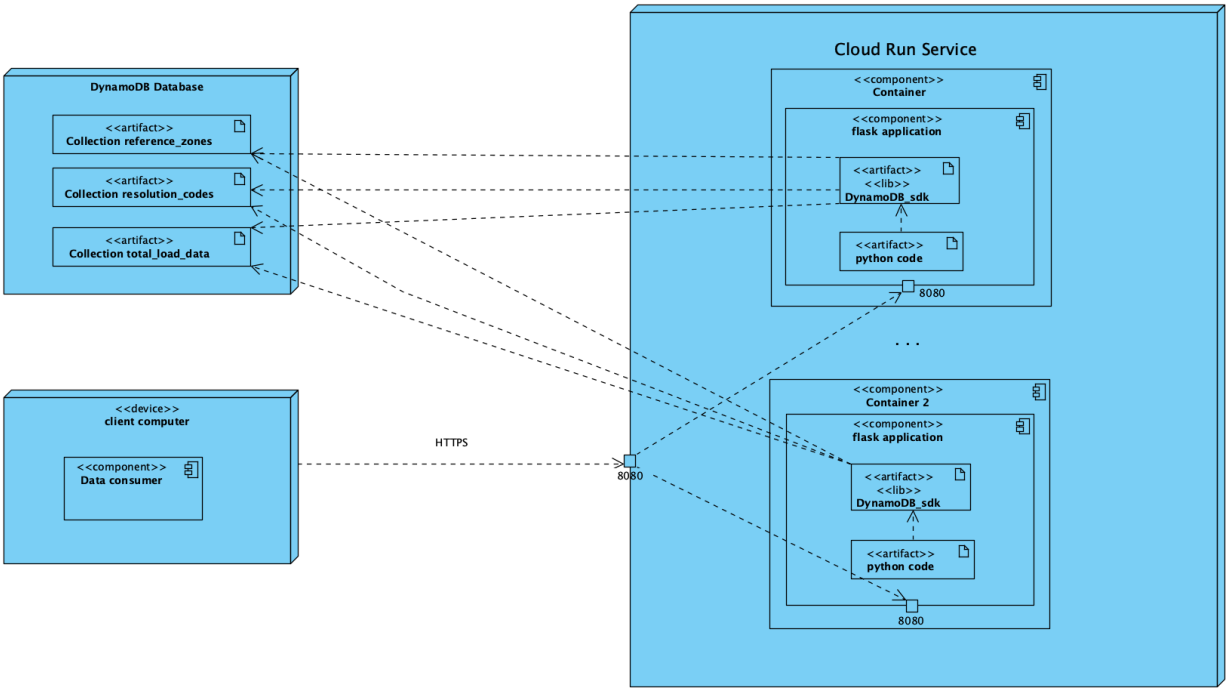
Γνωρίζαμε ότι αυξάνοντας το concurrency θα έπεφταν οι αποδόσεις του Cloud Run, αλλά θεωρήσαμε ότι θα αντισταθμιζόταν από το υψηλό throughput του DynamoDB. Κάτι τέτοιο τελικά δεν συνέβη καθώς το Average Throughput έπεσε στο 173 hits/sec ενώ Firestore - CloudRun (max-conc=80) είχε 343 hits/sec.

Κάτι τέτοιες παραμετροποιήσεις είναι αρκετά σημαντικές ως προς το cost efficiency. Δηλαδή ξέρουμε πως το max-concurrency στο Cloud Run επηρεάζει αρκετά το τελικό κόστος, οπότε θα ήταν πολύ σημαντικό να μπορούμε να το αυξήσουμε όπου μπορούμε.

- Παρατηρούμε μεγάλη αστάθεια. Γενικά τα endpoints φαίνεται να μην σταθεροποιήθηκαν ποτέ καθόλη την διάρκεια του test και αυτό οφείλεται στην παράμετρο του max concurrency.

4.4.6 AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4

Τα δεδομένα είναι σε AWS DynamoDB και τα APIs είναι deployed σε Google Cloud Run, έχοντας ορίσει το max-concurrency ίσο με 4, ως ένα container ακολουθώντας το μονολιθικό μοντέλο. Το αντίστοιχο repository: [https://github.com/Cloud-Engineering-Softlab-Project/DynamoDB-CloudRun-single]



Εικόνα 171: UML Deployment Diagram of "AWS DynamoDB - Cloud Run"

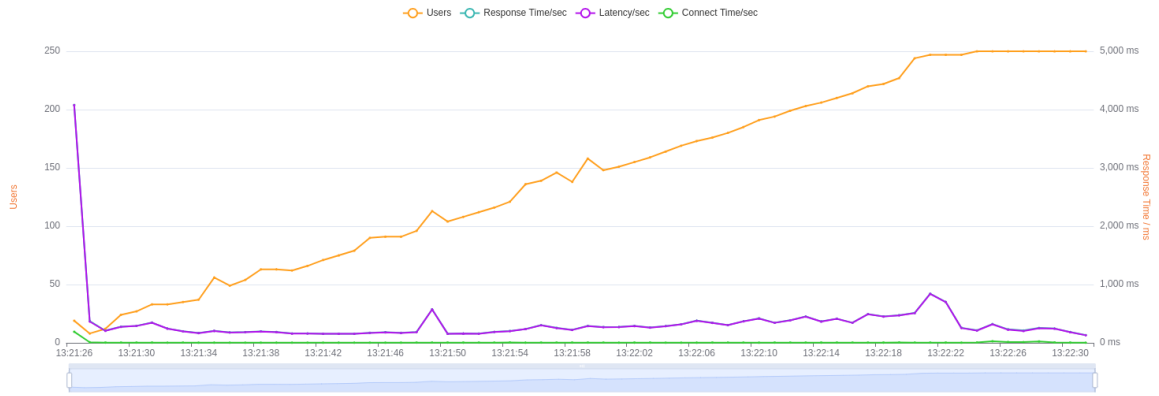
4.4.6.1 Διαγράμματα

Test Stats

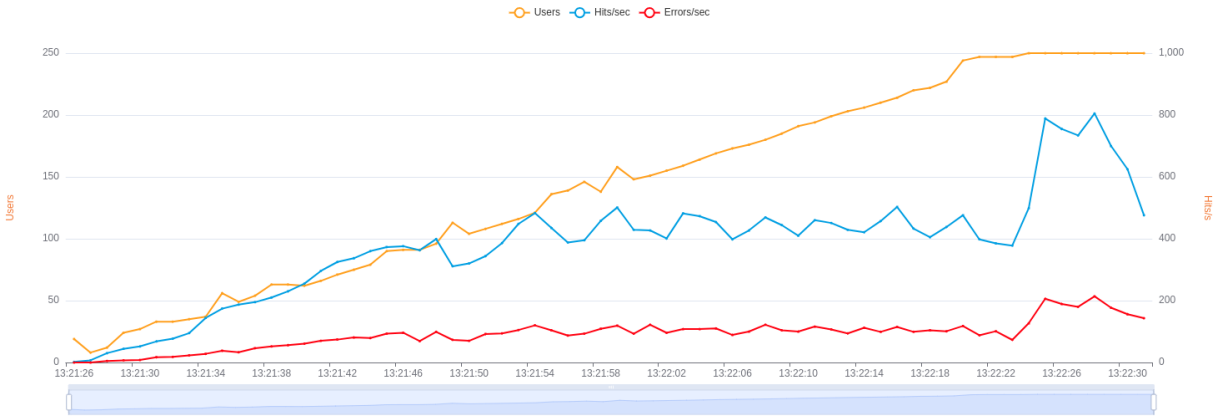
Max Users	Avg. Throughput	Errors	Avg. Response Time
250 VU	381.58 HITS/SEC	24.06 %	293.97 MSEC

Avg. Received Bytes/sec	Avg. Sent Bytes/sec
19.26 MB	99.89 KB

Εικόνα 172: Test stats of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"



Εικόνα 173: Timeline of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"



Εικόνα 174: Hits & Errors of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"

Summary Values of Session

Total Hits	Avg. Response Time	Max Response	Min Response
24899	293.97 MSEC	5126.00 MSEC	39.00 MSEC

Percentage Error	Total Throughput	Avg. Connect Time	Avg. Latency
24.06 %	387.81 RPS	6.04 MSEC	290.44 MSEC

Total Error Hits
5990

Εικόνα 175: Summary of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"

Label	Total Hits	Avg. Throughput/RPS	Avg. Resp. Time/Sec
"energy-data-advanced"	5031	77.10	0.51
"energy-data-simple"	4987	77.47	0.41
"get-ref-zone"	4973	77.81	0.18
"add-ref-zone"	4963	77.76	0.18
"delete-ref-zone"	4945	77.67	0.18

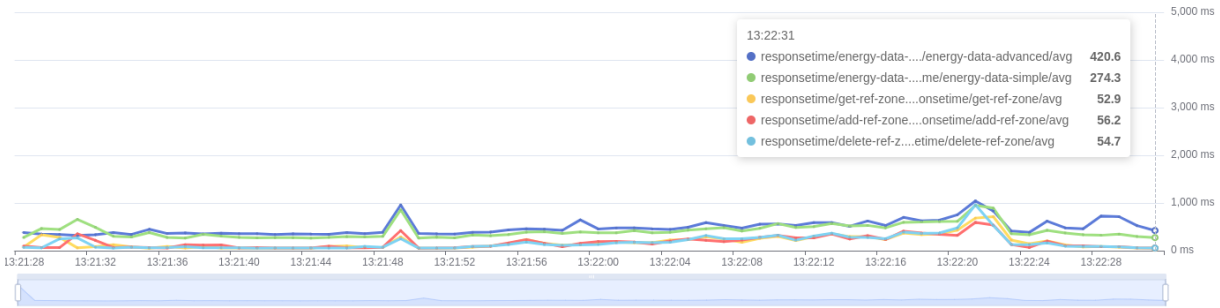
Εικόνα 176: APIs metrics (1) of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"

Label	Max Resp. Time/Sec	Min Resp. Time/Sec	Total Error Hits
"energy-data-advanced"	5.13	0.26	0
"energy-data-simple"	4.65	0.20	0
"get-ref-zone"	4.14	0.04	0
"add-ref-zone"	3.88	0.04	3007
"delete-ref-zone"	4.43	0.04	2983

Εικόνα 177: APIs metrics (2) of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"

Label	Response Code	Response Code Message	Number of Response
energy-data-advanced	200	OK	5031
energy-data-simple	200	OK	4987
get-ref-zone	200	OK	4973
add-ref-zone	400	Bad Request	3007
add-ref-zone	200	OK	1956
delete-ref-zone	400	Bad Request	2983
delete-ref-zone	200	OK	1962

Εικόνα 178: Response codes of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"



Εικόνα 179: Average response time graph of "AWS DynamoDB - Cloud Run (monolithic) – max-concurrency=4"

4.4.6.2 Σχόλια

Βασικές μετρικές:

- Test Duration = 2 min
- Average Response Time = 293 ms
- Average Throughput = 381 hits/sec
- Total Hits = 24899

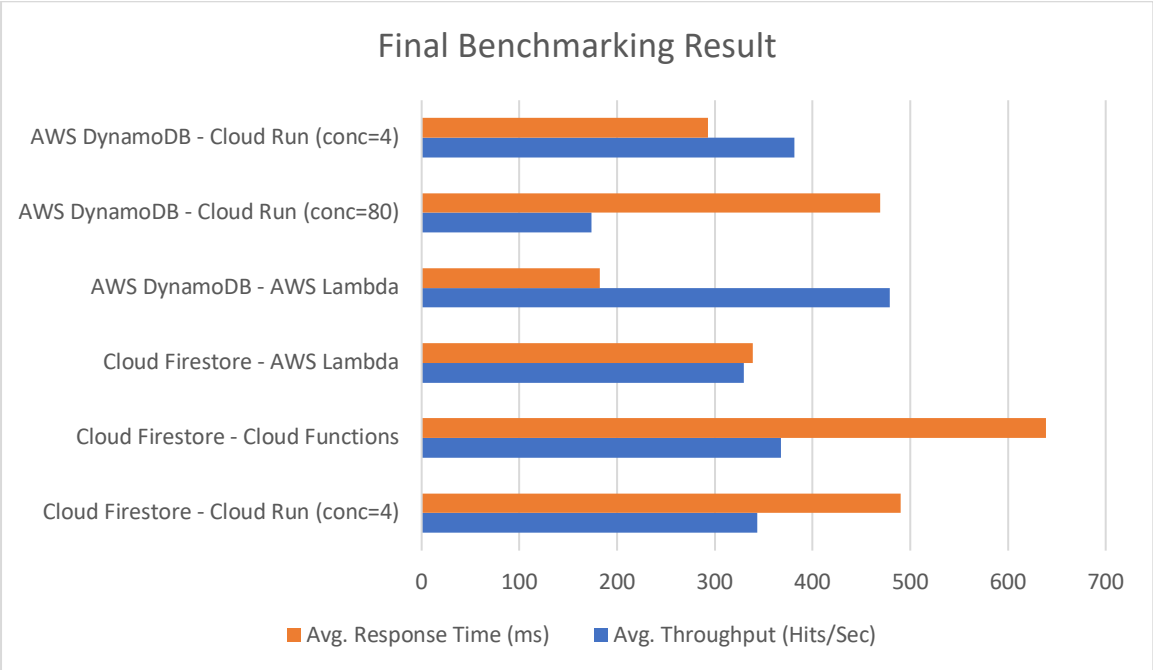
Παρατηρήσεις:

- Μειώσαμε το concurrency από 80 σε 4, οπότε τώρα το εκάστοτε container διαχειρίζεται το πολύ 4 requests ταυτόχρονα. Όπως ήταν αναμενόμενο, βελτιώθηκε το σύστημα και αυξήθηκε περίπου στο διπλάσιο το Throughput. Αυτό αποδεικνύει για ακόμη μία φορά πως το Cloud Run είναι το bottleneck του συστήματος όταν εμπλέκεται το DynamoDB.

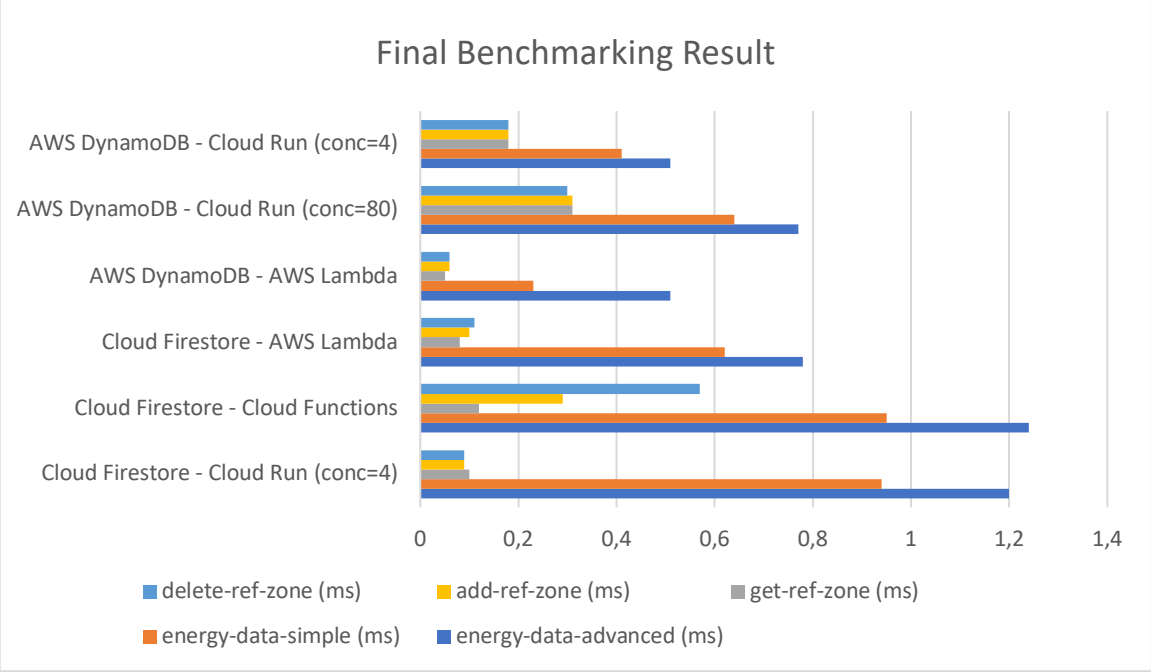
- Επίσης, επιβεβαιώνεται πως η αστάθεια στα endpoints οφειλόταν στο υψηλό concurrency που είχαμε θέσει, καθώς τώρα παρατηρούμε όλα τα endpoints να έχουν σταθεροποιηθεί.

- Τέλος, παρατηρούμε ότι δεν υπάρχουν cold starts και αυτό συμβαίνει διότι το προηγούμενο test το είχαμε κάνει ακριβώς πριν από αυτό και έτσι το Cloud Run Service είχε φροντίσει να δημιουργήσει κάποια Idle Containers που χρησίμευσαν εν τέλει σε αυτό το test.

4.5 Γενικές παρατηρήσεις



Εικόνα 180: General results or each deployment scenario



Εικόνα 181: Results of each deployment scenario (grouped by endpoint)

Σύμφωνα με τα παραπάνω διαγράμματα είναι ξεκάθαρο πως ο καλύτερος συνδυασμός από άποψη Performance είναι ο ***AWS DynamoDB – AWS Lambda***.
Αυτός ο συνδυασμός έχει το μεγαλύτερο Average Throughput, και κατ' επέκταση το μικρότερο Average Response Time (Εικόνα 180), κάτι που εξηγείται και στην Εικόνα 181 καθώς έχει το μικρότερο Average Response Time σε κάθε endpoint.

Αξίζει να σημειωθεί πως αμέσως επόμενος πιο γρήγορος συνδυασμός είναι το AWS DynamoDB – Cloud Run (conc=4), ο οποίος συμπεριλαμβάνει κι αυτός το AWS DynamoDB. Στο κεφάλαιο 3 δεν ελέγξαμε το throughput των βάσεων δεδομένων, αλλά τώρα συγκρίνοντας τα σενάρια 4.4.1 και 4.4.6 μπορούμε να αποφανθούμε πως ***το DynamoDB είναι πιο γρήγορο από το Cloud Firestore*** καθώς η μόνη διαφορά των 2 σεναρίων είναι η βάση δεδομένων.

Στο σενάριο 4.4.5 βλέπουμε για άλλη μία φορά πόσο πολύ επηρεάζει το σύστημα μας η παράμετρος του Cloud Run concurrency. Παρατηρούμε, λοιπόν, πως πέρα από αυτό το σενάριο ***το bottleneck είναι η βάση δεδομένων μας***. Αυτό είναι αναμενόμενο καθώς τα APIs δεν εκτελούν κάποια διεργασία με μεγάλο υπολογιστικό φόρτο, κυρίως εκτελούν queries στην εκάστοτε βάση, οπότε είναι λογικό το performance να επηρεάζεται ως επί το πλείστων από την βάση δεδομένων.

Επιβεβαιώνονται τα αποτελέσματα του κεφαλαίου 2. Αν και η βάση δεδομένων επηρεάζει κατά κύριο λόγο το performance του συστήματος παρατηρούμε την υπεροχή του AWS Lambda έναντι του Cloud Runσυγκρίνοντας τα σενάρια 4.4.4 και 4.4.6. Τέλος συγκρίνοντας τα σενάρια 4.4.1 και 4.4.2 και 4.4.3 επιβεβαιώνουμε για άλλη μια φορά την ιεραρχία των υπολογιστικών υπηρεσιών όσον αφορά το performance.

Συνεπώς, τα προϊόντα που μελετήσαμε, με βασικό κριτήριο το performance, ταξινομούνται ως εξής:

- Υπολογιστικές υπηρεσίες:
  - AWS Lambda
  - Google Cloud Run
  - Google Cloud Functions

- Βάσεις δεδομένων:
    - AWS DynamoDB
    - Google Cloud Firestore

Κεφάλαιο 5 Συμπεράσματα και μελλοντική εργασία

5.1 Συνεισφορά και πλεονεκτήματα

Το Cloud Engineering είναι ένας πολύ μεγάλος τεχνολογικός τομέας που γνωρίζει αρκετά μεγάλη αναγνώριση τα τελευταία χρόνια. Στην παρούσα διπλωματική το μελετήσαμε από την οπτική ενός χρήστη ή ενός καταναλωτή, όμως αξίζει να σημειωθεί πως στο παρασκήνιο χρησιμοποιούνται τεχνολογίες και αλγόριθμοι δικτύων, κατανεμημένων συστημάτων, λειτουργικών συστημάτων κ.α.

Μετά την ολοκλήρωση της εργασίας αυτής, έχουμε μελετήσει δύο από τις πιο γνωστές NoSQL βάσεις δεδομένων του Cloud και τρεις από τις πιο γνωστές serverless υπολογιστικές υπηρεσίες χρησιμοποιώντας δύο από τους μεγαλύτερους Cloud providers στον κόσμο: Amazon και Google. Έχοντας αναφέρει τα ιδιαίτερα χαρακτηριστικά, τα πλεονεκτήματα και τα μειονεκτήματα της εκάστοτε υπηρεσίας, ο αναγνώστης μπορεί να προσανατολιστεί πιο γρήγορα στην υπηρεσία που χρειάζεται με βάση τις ανάγκες του.

Συνεισφορά της παρούσας εργασίας εντοπίζεται επίσης στην υλοποίηση και στο deployment του εκάστοτε σεναρίου καθώς αντιμετωπίζονται προβλήματα που δεν αναφέρονται στα επίσημα documentations, ενώ ταυτόχρονα αναφέρονται τεχνικές αξιολόγησης με την χρήση διαφόρων μετρικών
όπως performance, concurrency, scalability, cost κ.α.

Στην συνέχεια, παραθέτουμε μια τελική σύγκριση των υπηρεσιών που μελετήθηκαν με βασικό κριτήριο το performance. Εξασφαλίσαμε την αξιοπιστία αυτής της σύγκρισης εφαρμόζοντας καλές πρακτικές ως προς την υλοποίηση αλλά και ως προς την αξιολόγηση των αποτελεσμάτων. Αξίζει, βέβαια, να σημειωθεί πως το performance δεν είναι το μοναδικό κριτήριο επιλογής ενός Cloud προϊόντος και πως πρέπει να ληφθούν υπόψιν κι άλλοι παράγοντες όπως και αναφέρονται στα ανάλογα κεφάλαια.

Τέλος η συγγραφή των προγραμμάτων σε Python αποτελεί χρήσιμο template για όσους θέλουν να υλοποιήσουν Back-End services με τα προϊόντα που μελετήθηκαν.

5.2 Μελλοντικές Προτάσεις

5.2.1 Εκτίμηση κόστους

Οι Serverless Cloud υπηρεσίες έχουν αρκετά θετικά χαρακτηριστικά, όπως έχουν αναφερθεί στο κεφάλαιο 1.2.1, όπως cost-efficiency, scalability, security κ.α.
Έχει αναφερθεί το cost-efficiency ως θετικό χαρακτηριστικό και όντως είναι, καθώς το κόστος υπολογίζεται με pay-per-use πολιτική, όμως αυτό κρύβει αρκετούς κινδύνους. Όταν, χρησιμοποιούμε dedicated servers ναι μεν δεν έχουμε αυτοματοποιημένο scalability και σίγουρα έχουμε περιορισμένους πόρους, όμως ξέρουμε ακριβώς πόσο πληρώνουμε.

Στην περίπτωση του Cloud, λοιπόν, αν και υπάρχουν pricing calculators όπως στην Amazon [https://calculator.aws/#/] είναι δύσκολο να εκτιμήσουμε το ακριβές τελικό κόστος. Αυτό συμβαίνει γιατί χρησιμοποιούνται διάφορες υπηρεσίες που μπορεί να μην έχουμε λάβει υπόψιν.

Π.χ. Έστω ότι χρησιμοποιούμε το stack AWS API Gateway – AWS Lambda – AWS DynamoDB. Στην πραγματικότητα χρειαζόμαστε και το AWS CloudWatch για να έχουμε logs, θέλουμε και το AWS X-Ray για καλύτερο health monitoring της εφαρμογής, μπορεί να χρειαστούμε secondary indexes στο DynamoDB και διάφορες άλλες υπηρεσίες ή παραμετροποιήσεις που κοστίζουν πολύ περισσότερο από τα πρώτα 3 βασικά προϊόντα.

Εν κατακλείδι, όλα είναι θέμα κλίμακας. Όταν χρειαζόμαστε έναν server να δουλεύει συνεχώς 24/7 τότε ναι, ο dedicated server είναι καλύτερη λύση από άποψη κόστους. Όμως αν το σύστημα μας δεν έχει ανάγκη για συνεχόμενη λειτουργία τότε η serverless αρχιτεκτονική είναι καλύτερη λύση.

Οπότε υπάρχει ανάγκη για ένα εργαλείο που:

- σε πρώτο στάδιο, να κάνει καλύτερη και πιο ακριβή εκτίμηση του συνολικού κόστους μιας serverless αρχιτεκτονικής και

σε δεύτερο στάδιο, λαμβάνοντας υπόψη τις ανάγκες της εκάστοτης εφαρμογής, να αποφανθεί εάν συμφέρει η serverless αρχιτεκτονική ή η full dedicated server αρχιτεκτονική.

Βιβλιογραφία

McCumskey, G. (2021, 06 07). *Why local development for serverless is an anti-pattern*. Retrieved from dev.to: https://dev.to/garethmcc/why-local-development-for-serverless-is-an-anti-pattern-1d9b

Azure. (2022, 01 01). *What is cloud computing*. Retrieved from azure.microsoft.com: https://azure.microsoft.com/en-us/resources/cloud-computing-dictionary/what-is-cloud-computing/#benefits

Cloudflare. (2022, 01 01). *What is serverless*. Retrieved from cloudflare.com: https://www.cloudflare.com/en-gb/learning/serverless/what-is-serverless/

Cloudflare. (2022, 01 01). *Why use serverless*. Retrieved from cloudflare.com: https://www.cloudflare.com/en-gb/learning/serverless/why-use-serverless/

Google. (2022, 01 01). *Cloud Run*. Retrieved from google.cloud.com: https://cloud.google.com/run

Google. (2022, 06 24). *Deploy a Python service to Cloud Run*. Retrieved from cloud.google.com: https://cloud.google.com/run/docs/quickstarts/build-and-deploy/deploy-python-service

Google. (2022, 06 24). *Install the gcloud CLI*. Retrieved from cloud.google.com: https://cloud.google.com/sdk/docs/install

Loadium. (2022, 06 24). *Loadium landing page*. Retrieved from loadium.com: https://loadium.com

Google. (2022, 06 24). *Cloud Functions*. Retrieved from cloud.google.com: https://cloud.google.com/functions

Yongfook, J. (2020, 08 01). *Google Cloud Functions vs. AWS Lambda Test Results*. Retrieved from bannerbear.com: https://www.bannerbear.com/blog/google-cloud-functions-vs-aws-lambda/

Zappa. (2022, 06 24). *Zappa repository*. Retrieved from github.com: https://github.com/zappa/Zappa#about

Kazarinoff, P. (2020, 01 14). *Deploy a Serverless Web App on AWS Lambda with Zappa*. Retrieved from pythonforundergradengineers.com: https://pythonforundergradengineers.com/deploy-serverless-web-app-aws-lambda-zappa.html

Rapes, K. (2019, 03 18). *The Everything Guide to Lambda Throttling, Reserved Concurrency, and Execution Limits*. Retrieved from itnext.io: https://itnext.io/the-everything-guide-to-lambda-throttling-reserved-concurrency-and-execution-limits-d64f144129e5

AWS. (2022, 06 24). *AWS service quotas* . Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html

AWS. (2022, 06 24). *Managing Lambda provisioned concurrency* . Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/lambda/latest/dg/provisioned-concurrency.html

AWS. (2022, 06 24). *What is Application Auto Scaling?* . Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/autoscaling/application/userguide/what-is-application-auto-scaling.html

AWS. (2022, 06 24). *AWS Lambda*. Retrieved from aws.amazon.com: https://aws.amazon.com/lambda/

Puffelen, F. V. (2019, 11 17). *Fastest way to write a lot of documents to Firestore*. Retrieved from stackoverflow.com: https://stackoverflow.com/questions/58897274/what-is-the-fastest-way-to-write-a-lot-of-documents-to-firestore?answertab=trending

Google. (2022, 06 24). *Best practices*. Retrieved from cloud.google.com: https://cloud.google.com/firestore/docs/best-practices

Google. (2022, 06 24). *Index types in Cloud Firestore*. Retrieved from cloud.google.com: https://firebase.google.com/docs/firestore/query-data/index-overview

Google. (2022, 06 24). *Reducing index costs with map fields*. Retrieved from cloud.google.com: https://cloud.google.com/firestore/docs/solutions/index-map-field

AWS. (2022, 06 24). *Best Practices for Designing and Architecting with DynamoDB*. Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/best-practices.html#GuidelinesForTables.UniformWorkload

AWS. (2022, 06 24). *Boto3 documentation*. Retrieved from boto3.amazonaws.com: https://boto3.amazonaws.com/v1/documentation/api/latest/reference/customizations/dynamodb.html#dynamodb-conditions

Google. (2022, 06 24). *Perform simple and compound queries in Cloud Firestore*. Retrieved from firebase.google.com: https://firebase.google.com/docs/firestore/query-data/queries

AWS. (2022, 06 24). *Importing Data From Amazon S3 to DynamoDB* . Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBPipeline.html#DataPipelineExportImport.Importing

Hiltch, O. (2022, 06 24). *DynamoDB Insert: Performance Basics in Python/Boto3* . Retrieved from medium.com: https://medium.com/skyline-ai/dynamodb-insert-performance-basics-in-python-boto3-5bc01919c79f

AWS. (2022, 06 24). *The Lambda monolith* . Retrieved from docs.aws.amazon.com: https://docs.aws.amazon.com/lambda/latest/operatorguide/monolith.html

ENTSOE. (2022, 06 24). *entsoe.eu*. Retrieved from ENTSO-E Mission Statement : https://www.entsoe.eu/about/inside-entsoe/objectives/

ENTSOE. (2022, 06 24). *ENTSO-E Transparency Platform*. Retrieved from transparency.entsoe.eu: https://transparency.entsoe.eu/content/static_content/Static%20content/knowledge%20base/knowledge%20base.html

Πίνακας Εικόνων

Στον παρακάτω σύνδεσμο βρίσκονται όλες οι φωτογραφίες με την ίδια αρίθμηση [https://drive.google.com/drive/folders/1MBkVgEWoh65bh9dRAMs67UdmFp5j1xQR?usp=sharing]