

ΙΟΝΙΟ ΠΑΝΕΠΙΣΤΗΜΙΟ  
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ



Πτυχιακή Εργασία

# Κατανεμημένη εκτέλεση κώδικα ελέγχου εφαρμογών (test suites) σε περιβάλλον Kubernetes

**Νικόλαος Γκαρλής**  
**Επιβλέπων: Μιχαήλ Στεφανιδάκης**

21 Ιουνίου 2023

## Επιβλέπων

**Μιχαήλ Στεφανιδάκης**, Αναπληρωτής Καθηγητής,  
Τμήμα Πληροφορικής

## Τριμελής Επιτροπή

**Μιχαήλ Στεφανιδάκης**, Αναπληρωτής Καθηγητής,  
Τμήμα Πληροφορικής

**Κωνσταντίνος Οικονόμου**, Καθηγητής,  
Τμήμα Πληροφορικής

**Δημήτριος Ρίγγας**, Εργαστηριακό Διδακτικό Προσωπικό,  
Τμήμα Πληροφορικής

Η παρούσα πτυχιακή εργασία εκπονήθηκε στα πλαίσια του προπτυχιακού προγράμματος του Τμήματος Πληροφορικής του Ιονίου Πανεπιστημίου.

# Ευχαριστίες

Θέλω να εκφράσω την ειλικρινή μου ευγνωμοσύνη προς τους γονείς μου, που με στήριξαν ανεπιφύλακτα κατά τη διάρκεια των σπουδών μου. Χωρίς την συνεχή στήριξή τους, αυτή η διαδρομή δεν θα ήταν δυνατή.

# Περίληψη

Τα `microservices` είναι πλέον ένας δημοφιλής τρόπος να σχεδιάσει κανείς το σύστημά του. Σε μεγάλα αποθετήρια κώδικα μπορεί να βρεί κανείς εφαρμογές οι οποίες αποτελούνται από μια ή παραπάνω βάσεις δεδομένων, πολλές φορές παραπάνω από ένα REST API και μερικές φορές ακόμα και παραπάνω από μια Frontend εφαρμογή. Οι οργανισμοί επιλέγουν να επαληθεύσουν την ποιότητα αυτών των `microservices` μέσω μιας τεχνικής αυτοματοποιημένων test (integration tests). Αυτή η τεχνική όμως παρουσιάζει προβλήματα χρόνου, πόρων καθώς και ποιότητας των αποτελεσμάτων.

Αυτή η πτυχιακή προτείνει τον σχεδιασμό ενός συστήματος το οποίο θα επιχειρήσει να αντιμετωπίσει τα παραπάνω προβλήματα μέσω κατανεμημένης εκτέλεσης των tests σε περιβάλλον Kubernetes. Το σύστημα δοκιμάστηκε με τρία διαφορετικά πειράματα χρησιμοποιώντας test suites (κώδικα ελέγχου εφαρμογών) ενός δημοφιλούς αποθετηρίου κώδικα. Τα αποτελέσματα υποδεικνύουν βελτίωση του χρόνου εκτέλεσης υπό συγκεκριμένες συνθήκες.

# Abstract

Microservices have emerged as a preferred method for system design. In large code repositories, applications often incorporate one or more databases, usually exceeding a single REST API, and in certain cases, comprising more than one frontend application. Organizations ensure the quality of these microservices through a technique of automated testing (integration tests). However, this approach poses challenges concerning time, resources, and reliability.

In response to these challenges, this thesis suggests the creation of a system designed to address these issues by executing said tests in a distributed manner within a Kubernetes environment. This proposed system underwent testing with three distinct experiments using test suites from a widely-used code repository. The outcomes highlight a boost in the execution speed under particular conditions.

# Περιεχόμενα

|                                                                                               |           |
|-----------------------------------------------------------------------------------------------|-----------|
| Επιβλέπων.....                                                                                | 2         |
| Τριμελής Επιτροπή.....                                                                        | 2         |
| <b>Ευχαριστίες.....</b>                                                                       | <b>3</b>  |
| <b>Περίληψη.....</b>                                                                          | <b>4</b>  |
| <b>Abstract.....</b>                                                                          | <b>5</b>  |
| <b>Περιεχόμενα.....</b>                                                                       | <b>6</b>  |
| <b>Εισαγωγή.....</b>                                                                          | <b>8</b>  |
| <b>Κεφάλαιο 1</b>                                                                             |           |
| <b>Θεωρία.....</b>                                                                            | <b>10</b> |
| 1.1 Αυτοματοποιημένος έλεγχος test.....                                                       | 10        |
| 1.1.1 Unit tests.....                                                                         | 10        |
| 1.1.2 Integration tests.....                                                                  | 10        |
| 1.1.3 End-to-end tests.....                                                                   | 10        |
| 1.2 Συνεχής Ενσωμάτωση και Συνεχής Παράδοση (CI/CD).....                                      | 11        |
| 1.3 Virtualization.....                                                                       | 11        |
| 1.3.1 Hardware Virtualization.....                                                            | 11        |
| 1.3.2 Operating System Level Virtualization.....                                              | 12        |
| 1.4 Kubernetes.....                                                                           | 13        |
| 1.4.1 Kubernetes API.....                                                                     | 14        |
| 1.5.1 Load Balancing.....                                                                     | 15        |
| 1.6 Parallel Computing.....                                                                   | 16        |
| 1.6.1 Granularity.....                                                                        | 16        |
| 1.6 Distributed Computing.....                                                                | 17        |
| <b>Κεφάλαιο 2</b>                                                                             |           |
| <b>Τεχνική υλοποίηση του συστήματος καταναμημένης εκτέλεσης κώδικα ελέγχου εφαρμογών.....</b> | <b>18</b> |
| 2.1 Αρχιτεκτονική του συστήματος.....                                                         | 18        |
| 2.2 Worker.....                                                                               | 19        |
| 2.2.1 Δέσμευση διεργασίας.....                                                                | 20        |
| 2.2.2 Αποδέσμευση διεργασίας.....                                                             | 21        |
| 2.2.3 Διάγραμμα ροής.....                                                                     | 22        |
| 2.2.4 Docker container.....                                                                   | 23        |
| 2.3 Controller.....                                                                           | 26        |
| 2.4 Client.....                                                                               | 29        |
| 2.5 Database.....                                                                             | 30        |
| 2.6 Storage.....                                                                              | 31        |
| 2.7 Container registry.....                                                                   | 31        |
| 2.8 Kubernetes cluster.....                                                                   | 31        |
| 2.8.1 Google Cloud.....                                                                       | 31        |
| 2.8.2 Terraform.....                                                                          | 31        |
| 2.8.3 Helm.....                                                                               | 31        |

|                                             |           |
|---------------------------------------------|-----------|
| 2.9 Πειραματικό περιβάλλον.....             | 32        |
| 2.9.1 Σύνταξη setup.sh.....                 | 32        |
| 2.9.2 Σύνταξη job.sh.....                   | 33        |
| 2.9.3 Σύνταξη Dockerfile.....               | 34        |
| <b>Κεφάλαιο 3</b>                           |           |
| <b>Αξιολόγηση των αποτελεσμάτων.....</b>    | <b>36</b> |
| 3.1 Σειριακή εκτέλεση (Coarse-Grained)..... | 36        |
| 3.2 Σειριακή εκτέλεση (Fine-Grained).....   | 37        |
| 3.3 Παράλληλη εκτέλεση (Fine-Grained).....  | 38        |
| 3.4 Συμπέρασμα.....                         | 43        |
| <b>Κεφάλαιο 4</b>                           |           |
| <b>Μελλοντικές επεκτάσεις.....</b>          | <b>44</b> |
| 4.1 Διανομή κοινών εργασιών.....            | 44        |
| 4.2 Load Balancing Optimization.....        | 44        |
| 4.3 Caching Optimization.....               | 45        |
| 4.4 Stream των logs κατά την εκτέλεση.....  | 45        |
| <b>Γλωσσάρι.....</b>                        | <b>46</b> |
| <b>Βιβλιογραφία.....</b>                    | <b>49</b> |

# Εισαγωγή

Η αρχιτεκτονική των *microservices* είναι ένα πλαίσιο όπου μια εφαρμογή χωρίζεται σε μικρότερες υπηρεσίες. Η κάθε μία από αυτές τις υπηρεσίες τρέχει συνήθως ως ξεχωριστή διεργασία και διαχειρίζεται τη δική της βάση δεδομένων. Υπάρχουν πολλά πλεονεκτήματα και μειονεκτήματα αυτής της προσέγγισης. Μερικά από τα πλεονεκτήματα είναι:

- Η δυνατότητα *scaling* μεμονωμένων κομματιών μιας εφαρμογής μέσα από την χρήση πολλών αντιγράφων του ίδιου κώδικα σε διαφορετικά μηχανήματα στο *cloud*.
- Η εύκολη διαχείριση του κώδικα από μικρές ομάδες.
- Η γρήγορη ανάπτυξη του κάθε μεμονωμένου μέρους της εφαρμογής.

Ένα από τα κύρια μειονεκτήματα όμως είναι τα προβλήματα που παρουσιάζονται στον αυτοματοποιημένο κώδικα ελέγχου (*test suites*) στα οποία θα επικεντρωθεί αυτή η πτυχιακή εργασία.

Καθώς η αρχιτεκτονική των *microservices* διανέμει τις εφαρμογές, οι προγραμματιστές δεν μπορούν να ελέγξουν ολόκληρο το σύστημα από τα δικά τους μηχανήματα. Έτσι καταφεύγουν σε τεχνικές απομίμησης ολόκληρου του συστήματος μέσω *containers* η οποία είναι μια σύνθετη διαδικασία και ακριβή σε υπολογιστικούς πόρους.

Αυτή η προσέγγιση παρουσιάζει δύο προβλήματα στο πλαίσιο του αυτοματοποιημένου ελέγχου μιας εφαρμογής. Αυτά είναι η αργή εκτέλεση του κώδικα ελέγχου όπως και τα ανακριβή αποτελέσματα του κώδικα ελέγχου. Ένας από τους παράγοντες που συμβάλλουν στην καθυστέρηση της εκτέλεσης του κώδικα ελέγχου είναι η ενεργή χρήση του δικτύου καθώς η επικοινωνία μεταξύ των εφαρμογών βασίζεται σε αυτό. Κάθε έλεγχος μπορεί να κάνει μία ή παραπάνω κλήσεις στην ελεγχόμενη εφαρμογή η οποία με τη σειρά της μπορεί να κάνει μία ή παραπάνω κλήσεις σε βάσεις δεδομένων.

Ένας άλλος λόγος για την αργή εκτέλεση είναι η σειριακή εκτέλεση των διάφορων ελέγχων. Οι προγραμματιστές επιλέγουν να εκτελούν τους ελέγχους ενσωμάτωσης (*integration tests*) σειριακά καθώς αυτό δίνει μια επιπλέον εγγύηση ότι τα αποτελέσματα του ελέγχου θα είναι ακριβή. Διαφορετικά μπορεί να παρουσιαστεί το πρόβλημα του “noisy neighbor” διότι οι διάφορες διεργασίες του λειτουργικού συστήματος ανταγωνίζονται για υπολογιστικούς πόρους μεταξύ τους, κάτι που μπορεί να προκαλέσει αστάθεια στα αποτελέσματα του ελέγχου. Τα *tests* από τα οποία δεν προκύπτουν πάντα τα ίδια αποτελέσματα ονομάζονται “flaky tests”.<sup>[1]</sup>

Για αυτούς τους λόγους προτείνεται ο σχεδιασμός ενός συστήματος το οποίο θα έχει ως στόχο την μείωση του χρόνου εκτέλεσης των *tests* μέσω της παράλληλης εκτέλεσης τους αλλά διατηρώντας επίσης την ίδια σταθερότητα με τη σειριακή εκτέλεση των

*tests*. Το σύστημα αυτό θα κατανέμει τα αυτοματοποιημένα *tests* σε μεμονωμένους *Docker containers* οι οποίοι θα εκτελούν τα *test* παράλληλα σε έναν *Kubernetes*

cluster. Τέλος, θα διευκολύνει ακόμα την διαδικασία οριζόντιου scaling κατά την εκτέλεση των ελέγχων.

# Κεφάλαιο 1

## Θεωρία

Προτού η τεχνική υλοποίηση τεθεί υπό ανάλυση, είναι απαραίτητο για τον αναγνώστη να εξοικειωθεί πρώτα με μία σειρά κύριων όρων και βασικών εννοιών. Αυτοί οι όροι και οι έννοιες είναι άμεσα συνδεδεμένοι με την παρούσα πτυχιακή εργασία και θα αποτελέσουν τη βάση για την πιο βαθιά κατανόηση και εκτίμηση των αποτελεσμάτων που θα παρουσιαστούν στη συνέχεια.

### 1.1 Αυτοματοποιημένος έλεγχος test

Στον τομέα του ελέγχου λογισμικού, η αυτοματοποίηση των tests αναφέρεται στη χρήση λογισμικού ανεξάρτητου από το λογισμικό που δοκιμάζεται, με σκοπό τον έλεγχο της εκτέλεσης των tests και τη σύγκριση των πραγματικών αποτελεσμάτων με τα προβλεπόμενα αποτελέσματα. Συχνά, τα αυτοματοποιημένα tests οργανώνονται σε test suites, τα οποία είναι συλλογές από συνδυασμένα test cases που ελέγχουν συγκεκριμένες λειτουργικότητες ή απαιτήσεις του λογισμικού. Τα test suites βοηθούν στην αποτελεσματική και συστηματική εκτέλεση των tests, εξασφαλίζοντας τη συνέπεια και την ακρίβεια των αποτελεσμάτων.

Η αυτοματοποίηση των tests μπορεί να εκτελέσει χωρίς επιτήρηση ορισμένες επαναλαμβανόμενες, αλλά απαραίτητες εργασίες σε μία ήδη καθορισμένη διαδικασία δοκιμής, ή να πραγματοποιήσει επιπλέον δοκιμές που θα ήταν δύσκολο να γίνουν χειροκίνητα. Επιπλέον, είναι κρίσιμη για τη συνεχή δοκιμή και τη συνεχή παράδοση (CI/CD).<sup>[2]</sup>

Τα αυτοματοποιημένα tests χωρίζονται σε τρεις κατηγορίες.

#### 1.1.1 Unit tests

Αυτά είναι tests που ελέγχουν ανεξάρτητα τα μεμονωμένα στοιχεία ή μονάδες κώδικα. Είναι γρήγορα, αξιόπιστα και απομονώνουν τις αποτυχίες σε μικρές μονάδες κώδικα.

#### 1.1.2 Integration tests

Αυτά τα tests ελέγχουν πώς λειτουργούν μαζί διάφορες μονάδες κώδικα. Παρόλο που οι μεμονωμένες μονάδες μπορεί να λειτουργούν σωστά αυτόνομα, τα tests ενσωμάτωσης εξασφαλίζουν ότι λειτουργούν αποτελεσματικά μαζί.

#### 1.1.3 End-to-end tests

Αυτά τα tests ελέγχουν το σύστημα συνολικά, προσομοιώνοντας πραγματικά σενάρια χρήσης. Αυτά τα tests είναι πιο αργά και περίπλοκα.

Η πτυχιακή αυτή θα επικεντρωθεί στα integration tests καθώς τα πειράματα που εκτελέστηκαν χρησιμοποίησαν αυτού του είδους τα tests.

## 1.2 Συνεχής Ενσωμάτωση και Συνεχής Παράδοση (CI/CD)

Συνεχής Ενσωμάτωση και Συνεχής Παράδοση, (CI/CD) είναι μια πρακτική ανάπτυξης λογισμικού που στοχεύει στην αυτοματοποίηση και την αποτελεσματικότητα της κατασκευής, ενσωμάτωσης (integration) και παράδοσης (deployment) λογισμικού.

Συνεχής Ενσωμάτωση (CI) είναι η διαδικασία ενσωμάτωσης των αλλαγών κώδικα από πολλούς προγραμματιστές σε ένα κοινό αποθετήριο συχνά, συνήθως αρκετές φορές την ημέρα. Ο στόχος είναι η ανίχνευση και η αντιμετώπιση τυχόν προβλημάτων ενσωμάτωσης έγκαιρα, μέσω αυτοματοποιημένων διαδικασιών κατασκευής και δοκιμής του κώδικα.<sup>[3]</sup>

## 1.3 Virtualization

Ο παραδοσιακός τρόπος για έναν οργανισμό να εκτελέσει τις εφαρμογές του είναι μέσω φυσικών servers. Αυτή η προσέγγιση όμως, μπορεί να παρουσιάζει προβλήματα καθώς δεν υπάρχει απλός τρόπος να καθοριστούν όρια των πόρων για τις εφαρμογές αυτές. Αν για παράδειγμα υπάρχουν πολλές εφαρμογές που λειτουργούν στον ίδιο φυσικό server μπορεί να υπάρξει το πρόβλημα του “noisy neighbor”<sup>[4]</sup> όπου μία εφαρμογή καταλαμβάνει τους περισσότερους πόρους και επηρεάζει την εκτέλεση των άλλων. Μια λύση για το πρόβλημα αυτό θα ήταν οι εφαρμογές να μοιραστούν σε περισσότερους φυσικούς servers. Αυτή η λύση βέβαια μπορεί να είναι ακριβή διότι ένα μεγάλο ποσοστό των πόρων μένει ανεκμετάλλετο. Έτσι δημιουργείται ένα ζήτημα κατανομής πόρων. Αυτό είναι το πρόβλημα που προσπαθεί να λύσει το virtualization. Υπάρχουν διάφορα ήδη virtualization αλλά η πτυχιακή αυτή θα επικεντρωθεί σε δύο, το hardware virtualization και το operating system level virtualization.

### 1.3.1 Hardware Virtualization

Το hardware virtualization είναι η δημιουργία ενός εικονικού μηχανήματος που λειτουργεί ως ένας πραγματικός υπολογιστής με ένα λειτουργικό σύστημα. Το λογισμικό που εκτελείται σε αυτά τα εικονικά μηχανήματα απομονώνεται από τους υποκείμενους φυσικούς υπολογιστικούς πόρους.<sup>[5]</sup> Για παράδειγμα, ένας υπολογιστής που εκτελεί το Ubuntu Linux μπορεί να φιλοξενεί ένα εικονικό μηχανήμα που μοιάζει με έναν υπολογιστή με το λειτουργικό σύστημα Microsoft Windows. Τα εικονικά συστήματα που δημιουργούνται ονομάζονται “guest” ενώ τα συστήματα που τα δημιουργούν ονομάζονται “host”. Τα “host” μηχανήματα δημιουργούν εικονικά συστήματα μέσω ενός λογισμικού που ονομάζεται Hypervisor. Στο παρακάτω διάγραμμα φαίνονται 3 εικονικά συστήματα (VMs) τα οποία φιλοξενούνται από το ίδιο μηχανήμα. Το κάθε εικονικό σύστημα έχει το δικό του λειτουργικό σύστημα.

|            |          |          |
|------------|----------|----------|
| VM 1       | VM 2     | VM 3     |
| App A      | App B    | App C    |
| Guest OS   | Guest OS | Guest OS |
| Hypervisor |          |          |
| Hardware   |          |          |

Διάγραμμα 1.1

### 1.3.2 Operating System Level Virtualization

Η εικονικοποίηση σε επίπεδο λειτουργικού συστήματος, γνωστή και ως containerization, αναφέρεται σε μια δυνατότητα του πυρήνα ενός λειτουργικού συστήματος, με την οποία επιτρέπεται η ύπαρξη πολλαπλών απομονωμένων περιβαλλόντων χρήστη. Όπως τα VMs το κάθε container έχει το δικό του filesystem, καθορισμένους πόρους CPU και μνήμης. Τα προγράμματα που εκτελούνται μέσα σε ένα container μπορούν να δουν μόνο το περιεχόμενο του container και τις συσκευές που έχουν αντιστοιχιστεί στο container. Ωστόσο η κύρια διαφορά μεταξύ των containers και των VMs είναι ότι τα πρώτα μοιράζονται το ίδιο λειτουργικό σύστημα, κάτι που τα κάνει πιο ελαφρά στο σύστημα. Αυτοί είναι μερικοί από τους λόγους που τα containers είναι πολύ δημοφιλή. Ένα από τα πιο δημοφιλή λογισμικά containerization είναι το Docker. Στο διάγραμμα βλέπουμε 3 containers τα οποία λειτουργούν στο ίδιο λειτουργικό σύστημα.<sup>[6]</sup>

|               |             |             |
|---------------|-------------|-------------|
| Container 1   | Container 2 | Container 3 |
| App A         | App A       | App A       |
| Docker Engine |             |             |
| Host OS       |             |             |

Διάγραμμα 1.2

Το Docker είναι ένα λογισμικό που χρησιμοποιεί virtualization σε επίπεδο λειτουργικού συστήματος για την παράδοση λογισμικού σε containers.<sup>[7]</sup> Χρησιμοποιείται για την αυτοματοποίηση της ανάπτυξης εφαρμογών σε ελαφρά συστήματα, έτσι ώστε οι εφαρμογές να λειτουργούν αποδοτικά σε διάφορα περιβάλλοντα τα οποία είναι απομονωμένα.

Το Docker επιτρέπει τη συσκευασία (packaging) μιας εφαρμογής και των dependencies της σε ένα εικονικό container που μπορεί να εκτελεστεί σε διάφορα λειτουργικά συστήματα, όπως Linux, Windows και macOS. Έτσι επιτρέπει την

ανάπτυξη της εφαρμογής σε διάφορα μηχανήματα, χωρίς να ανησυχεί κανείς για προβλήματα συμβατότητας.<sup>[8]</sup> Μέσω του Docker, οι εφαρμογές μπορούν να εκτελούνται σταθερά σε διάφορες τοποθεσίες. Αυτό επιτρέπει στους οργανισμούς να υιοθετήσουν αρχιτεκτονικές αποκεντρωμένου υπολογισμού, κατακεντρωμένου υπολογισμού και cloud υπολογισμού.

## 1.4 Kubernetes

Το Kubernetes είναι ένα περιβάλλον διαχείρισης containers, που αυτοματοποιεί την ανάπτυξη, το scaling (κλιμάκωση) και τη διαχείριση εφαρμογών που είναι εγκατεστημένες σε clusters.<sup>[9]</sup> Επιτρέπει στους χρήστες να διαχειρίζονται και να υποστηρίζουν πολλαπλές εκδόσεις της ίδιας εφαρμογής σε διαφορετικά περιβάλλοντα, εξασφαλίζοντας ταυτόχρονα υψηλό επίπεδο διαθεσιμότητας και αποτελεσματικής χρήσης των πόρων.

Είναι ένα σύστημα σχεδιασμένο για να ανταποκρίνεται δυναμικά στις μεταβολές της ζήτησης, ρυθμίζοντας αυτόματα τον αριθμό των εργαζομένων containers που εκτελούν μια εφαρμογή. Αυτή η ρύθμιση βασίζεται στο φορτίο εργασίας και μπορεί να προσαρμοστεί για να αυξήσει ή να μειώσει τον αριθμό των containers που λειτουργούν.

Αυτό σημαίνει ότι το Kubernetes είναι σε θέση να διατηρεί σταθερή την απόδοση της εφαρμογής, ακόμα και σε περιόδους μεγάλης ζήτησης, επιτρέποντας το εύκολο scaling. Επιπλέον, όταν η ζήτηση μειώνεται, το Kubernetes είναι σε θέση να μειώσει τον αριθμό των λειτουργούντων containers, διατηρώντας έτσι την αποδοτικότητα και την οικονομία των πόρων.

Επομένως, το Kubernetes είναι ιδιαίτερα χρήσιμο για εφαρμογές που εμπεριέχουν μεγάλη, αλλά και απρόβλεπτη, ζήτηση. Είτε πρόκειται για μια περίοδο εξαιρετικά υψηλής ζήτησης λόγω μιας συγκεκριμένης εκδήλωσης, είτε για απρόβλεπτες διακυμάνσεις της ζήτησης, το Kubernetes μπορεί να προσαρμοστεί και να διασφαλίσει την αποδοτική λειτουργία των εφαρμογών.

Το Kubernetes προσφέρει πολλές λειτουργίες που το κάνουν ξεχωριστό, μερικές από τα οποίες είναι:

- **Αυτοματοποίηση:** Το Kubernetes αυτοματοποιεί τη διαδικασία της δημιουργίας και διαχείρισης των containers, ανταποκρινόμενο δυναμικά στις αλλαγές της ζήτησης.
- **Scaling:** Το Kubernetes επιτρέπει το εύκολο scaling των εφαρμογών, ρυθμίζοντας τον αριθμό των containers που εκτελούν μια εφαρμογή.
- **Φορητότητα:** Επιτρέπει την εκτέλεση εφαρμογών σε διάφορα περιβάλλοντα (φυσικούς υπολογιστές, εικονικές μηχανές και πλατφόρμες cloud).
- **Υψηλή Διαθεσιμότητα:** Το Kubernetes διασφαλίζει την λειτουργία των εφαρμογών με rolling updates (σταδιακές ενημερώσεις χωρίς διακοπή λειτουργίας) και σε περίπτωση σφάλματος μέσω των επανεκκινήσεων.

- **Διαχείριση Πόρων:** Το Kubernetes διασφαλίζει την αποδοτική χρήση των πόρων, διαχειριζόμενο την κατανομή της CPU, της μνήμης και του δικτύου μεταξύ των containers.<sup>[9]</sup>

### 1.4.1 Kubernetes API

Το Kubernetes API είναι η βασική διεπαφή με το control plane.<sup>[10]</sup> Οι περισσότερες λειτουργίες του Kubernetes μπορούν να διαχειριστούν ως κλήσεις RESTful API. Οι οντότητες του συστήματος ονομάζονται Kubernetes Resources. Τα resources αντιπροσωπεύουν το state (κατάσταση) του συστήματος και μπορούν να δημιουργηθούν, αναγνωστούν, αναβαθμιστούν ή διαγραφούν μέσω HTTP κλήσεων. Κάποια από αυτά είναι:

- **Pods**  
Τα Pods αποτελούνται από ένα ή παραπάνω containers τα οποία τρέχουν σε ένα κοινό περιβάλλον. Τα containers ενός Pod έχουν το ίδιο δίκτυο και τοπική διεύθυνση IP έτσι ώστε να είναι προσπελάσιμα μεταξύ τους. Πέρα από το δίκτυο, μπορούν ακόμα να μοιράζονται αποθηκευτικό χώρο. Επιπλέον μπορούν να έχουν προκαθορισμένα όρια σε υπολογιστικούς πόρους.
- **Services**  
Τα Services είναι μια αφηρημένη έννοια που ελέγχει την πρόσβαση σε Pods. Για παράδειγμα μπορούν να εκθέσουν ένα Pod σε μία εσωτερική IP διεύθυνση στον cluster. Είναι ανεξάρτητες οντότητες από τα Pods άρα δεν διαγράφονται όταν ένα Pod αντικαθίσταται.
- **Ingress**  
Το Ingress παρέχει δρομολόγηση από εξωτερικές διευθύνσεις προς τα Services (για παράδειγμα HTTP). Είναι προσπελάσιμα μέσω των εξωτερικών διευθύνσεων IP του cluster.
- **Jobs**  
Τα Jobs είναι μια παραλλαγή των Pods. Το κύριο χαρακτηριστικό τους είναι πως εκτελούνται μέχρι να ολοκληρωθεί μια καθορισμένη εργασία σε αντίθεση με τα απλά Pods τα οποία εκτελούν μακροχρόνιες υπηρεσίες. Πιο αναλυτικά τα Jobs δημιουργούν Pods τα οποία διαγράφουν όταν λάβουν το κατάλληλο exit code. Προσφέρουν επίσης μερικές χρήσιμες λειτουργίες όπως παραλληλοποίηση. Αν ο χρήστης για παράδειγμα ζητήσει τη δημιουργία ενός Job με τον αριθμό παραλληλοποίησης ίσο με δύο τότε το Kubernetes θα δημιουργήσει δύο Pods με τα ίδια containers και θα τα διαγράψει όταν λάβει το κατάλληλο exit code από το καθένα.

## 1.5.1 Load Balancing

Το Load Balancing (ισορροπία φόρτου) είναι μια τεχνική που χρησιμοποιείται για την ομοιόμορφη κατανομή των εργασιών σε servers ή άλλους υπολογιστικούς πόρους. Οι στόχοι είναι

- Η βελτιστοποίηση της χρήσης των πόρων
- Η μεγιστοποίηση της ροής δεδομένων
- Η ελαχιστοποίηση του χρόνου απάντησης
- Η αποφυγή της υπερφόρτωσης οποιουδήποτε μεμονωμένου πόρου <sup>[11]</sup>

Οι load balancing αλγόριθμοι διακρίνονται σε δύο κατηγορίες

- **Στατικοί αλγόριθμοι**

Όταν ένας αλγόριθμος load balancing δεν εξετάζει την κατάσταση του συστήματος για την κατανομή των διεργασιών, τότε θεωρείται στατικός. Ένα παράδειγμα ενός στατικού αλγορίθμου είναι ο Round Robin. Ο αλγόριθμος αυτός δίνει ένα ίσο, προκαθορισμένο διάστημα χρόνου σε κάθε διεργασία και τις εκτελεί σειριακά. Αυτή η τεχνική είναι επίσης γνωστή ως time slicing (διαίρεση χρόνου), επειδή σε κάθε διεργασία δίνεται ένα διάστημα χρόνου πριν γίνει προληπτική διακοπή της.<sup>[12]</sup> Ο αλγόριθμος του Weighted Round Robin και το IP Hash αποτελούν επίσης δύο εξαιρετικά δημοφιλείς μεθόδους στην κατηγορία αυτή. Το πλεονέκτημα των στατικών αλγορίθμων είναι ότι είναι εύκολοι στην εγκατάσταση και εξαιρετικά αποδοτικοί στην περίπτωση σχετικά τακτικών εργασιών (όπως η επεξεργασία αιτημάτων HTTP από μια ιστοσελίδα). Ωστόσο, υπάρχει κάποια στατιστική διακύμανση στην ανάθεση εργασιών που μπορεί να οδηγήσει στην υπερφόρτωση κάποιων υπολογιστικών μονάδων.<sup>[11]</sup>

- **Δυναμικοί αλγόριθμοι**

Ένας δυναμικός αλγόριθμος προσαρμόζεται δυναμικά, λαμβάνοντας υπόψη την τρέχουσα κατάσταση του συστήματος, όπως η χρήση της CPU ή της μνήμης.<sup>[11]</sup> Ένα παράδειγμα δυναμικού load balancing αλγορίθμου είναι ο αλγόριθμος Least connections. Ο αλγόριθμος δουλεύει κατανέμοντας τα requests στους servers με τις λιγότερες συνδέσεις. Μερικά ακόμα παραδείγματα δυναμικών αλγορίθμων είναι ο Weighted Least Connections και ο Weighted Response Time. Όπως και στον αλγόριθμο Least Connections, ο αλγόριθμος Weighted Least Connections κατανέμει τα requests στους servers με τις λιγότερες συνδέσεις λαμβάνοντας όμως υπόψη και ένα προκαθορισμένο βάρος για κάθε server. Απο την άλλη πλευρά, ο αλγόριθμος Weighted Response Time ρυθμίζει την κατανομή των requests λαμβάνοντας υπόψη τον μέσο χρόνο απόκρισης καθώς και το προκαθορισμένο βάρος για κάθε server.

## 1.6 Parallel Computing

Το parallel computing (παράλληλος υπολογισμός) είναι μια τεχνική όπου μια εργασία διαιρείται σε μικρότερα κομμάτια και εκτελείται ταυτόχρονα σε πολλούς επεξεργαστές στον ίδιο υπολογιστή.<sup>[13]</sup> Τα κομμάτια αυτά είναι ανεξάρτητα μεταξύ τους. Ο σκοπός της τεχνικής αυτής είναι να μειωθεί ο χρόνος εκτέλεσης μιας εργασίας.

### 1.6.1 Granularity

Στο πλαίσιο του parallel computing το granularity (επικοπή) αναφέρεται στην ποσότητα των υπολογισμών σε σχέση με την επικοινωνία, δηλαδή τον λόγο του χρόνου που δαπανάται για την εκτέλεση υπολογισμών προς τον χρόνο που δαπανάται για την επικοινωνία.<sup>[13]</sup>

$$G = \frac{T_e}{T_c}$$

Ανάλογα με τον φόρτο της εργασίας ο παράλληλος υπολογισμός συχνά κατηγοριοποιείται σε έναν από τους δύο τύπους.

- **Fine-Grained Parallelism (λεπτή επικοπή):**

Σε αυτό το είδος εκτέλεσης, το granularity, δηλαδή ο λόγος του χρόνου εκτέλεσης ως τον χρόνο επεξεργασίας είναι χαμηλός. Η εργασία χωρίζεται σε μικρά μέρη τα οποία στην συνέχεια εκτελούνται παράλληλα. Κάθε υπολογιστική μονάδα εκτελεί ένα μικρό μέρος υπολογισμού και στην συνέχεια προχωράει στο επόμενο. Αυτός ο τύπος παράλληλης επεξεργασίας απαιτεί υψηλό επικοινωνιακό φόρτο καθώς οι εργασίες πρέπει να συγχρονίζονται και να επικοινωνούν συχνά. Η Fine-grained εκτέλεση μπορεί να συνεισφέρει στην μεγιστοποίηση της λειτουργικότητας του συστήματος βελτιώνοντας ως συνέπεια το load balancing.<sup>[14]</sup>

- **Coarse-Grained Parallelism (χοντρή επικοπή):**

Σε αυτό το είδος εκτέλεσης, το granularity είναι υψηλό. Η εργασία διαιρείται σε πολύ μεγαλύτερα μέρη τα οποία απαιτούν λιγότερη επικοινωνία και συγχρονισμό. Αυτός ο τύπος παράλληλης επεξεργασίας είναι πιο εύκολος ως προς την διαχείριση, αλλά η παράλληλη επεξεργασία είναι λιγότερο ευέλικτη επειδή οι εργασίες είναι μεγαλύτερες. Στις περισσότερες περιπτώσεις ο χρόνος που δαπανάται για την επικοινωνία και τον συγχρονισμό είναι υψηλός σε σχέση με τον χρόνο εκτέλεσης όποτε είναι επωφελές να υπάρχει coarse granularity.<sup>[14]</sup>

Η ισορροπία του granularity είναι σημαντική για τον σχεδιασμό παράλληλων αλγορίθμων. Σε ένα ιδανικό σύστημα παράλληλου υπολογισμού η επιλεγμένη τιμή granularity μεγιστοποιεί τη χρήση των υπολογιστικών πόρων, ελαχιστοποιώντας τον χρόνο που δαπανάται για την επικοινωνία.

## 1.6 Distributed Computing

Το distributed computing (κατανεμημένος υπολογισμός) είναι υποσύνολο του parallel computing. Αναφέρεται στην υπολογιστική συνεργασία μεταξύ δύο ή περισσότερων συστημάτων που είναι συνδεδεμένα μεταξύ τους μέσω ενός δικτύου.<sup>[15]</sup>

Παρόμοια με το parallel computing οι μεγάλες εργασίες διαιρούνται σε πιο μικρά, διαχειρίσιμα τμήματα τα οποία όμως στη συνέχεια ανατίθενται και εκτελούνται από διαφορετικά συστήματα. Αυτά τα συστήματα μπορεί να λειτουργούν είτε αυτόνομα είτε υπό συντονισμό, παρέχοντας εν τέλει τη δυνατότητα για πιο αποτελεσματική και αποδοτική υπολογιστική εργασία.

Σε αυτήν τη διαδικασία, τα συστήματα ανταλλάσσουν και επεξεργάζονται πληροφορίες και υπολογισμούς συνεργαζόμενα για την επίλυση ενός κοινού προβλήματος. Υπάρχουν διάφοροι λόγοι για τους οποίους είναι επιθυμητή η χρήση του distributed computing: οι πρωτεύοντες είναι η βελτιστοποίηση της χρήσης των υπολογιστικών πόρων, η αύξηση της απόδοσης, η αντοχή σε σφάλματα (fault tolerance), καθώς και η δυνατότητα επέκτασης και κλιμάκωσης του συστήματος, ώστε να μπορεί να ανταποκρίνεται σε αυξανόμενες ανάγκες και απαιτήσεις.<sup>[16]</sup>

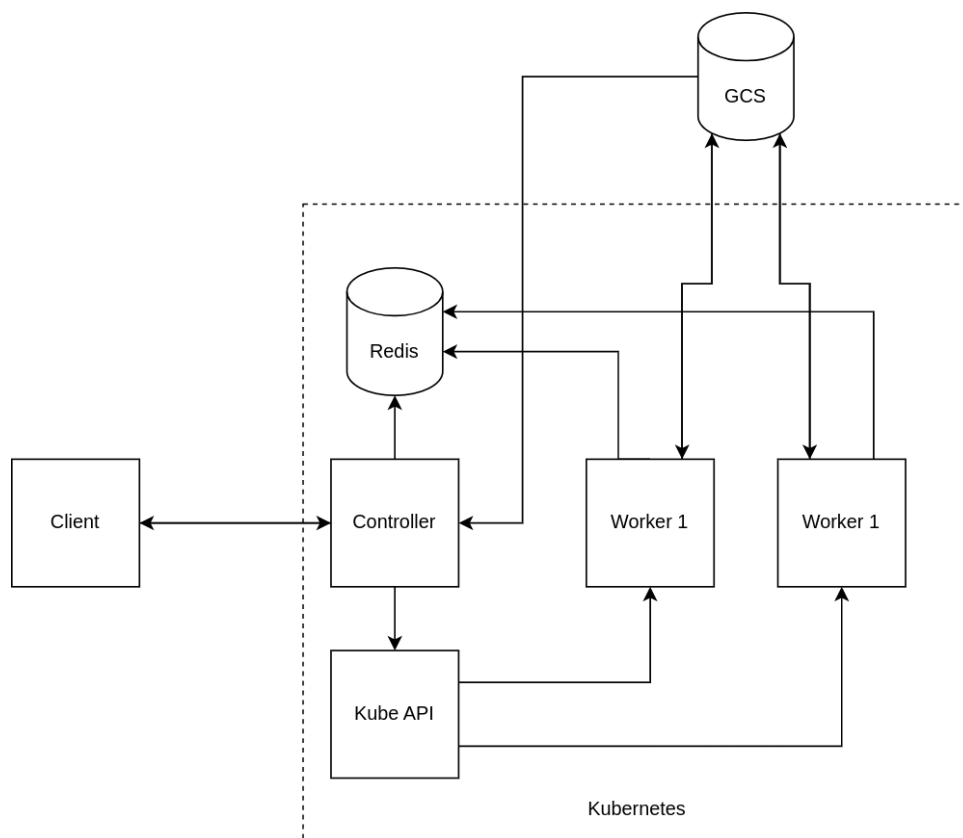
## Κεφάλαιο 2

# Τεχνική υλοποίηση του συστήματος κατανεμημένης εκτέλεσης κώδικα ελέγχου εφαρμογών

### 2.1 Αρχιτεκτονική του συστήματος

Το σύστημα αποτελείται από 5 μέρη. Στο διάγραμμα φαίνεται η βάση δεδομένων (Redis) όπου αποθηκεύονται τα δεδομένα της κάθε διεργασίας και ο controller ο οποίος διευθύνει την εκτέλεση των διεργασιών δημιουργώντας workers.

Οι workers εκτελούν τις υποδιεργασίες που είναι αποθηκευμένες στην βάση δεδομένων και αποθηκεύουν τα αποτελέσματα σε έναν bucket στο Google Cloud Storage. Ο controller προσπελαίνεται μέσω του client ο οποίος προσφέρει ένα γραφικό περιβάλλον με την κατάσταση των διεργασιών. Ο controller, οι workers και η βάση δεδομένων φιλοξενούνται στον Kubernetes cluster.



Διάγραμμα 2.1

## 2.2 Worker

Ο worker είναι γραμμένος με τη γλώσσα προγραμματισμού Go. Μια εργασία μπορεί να αποτελείται από έναν ή παραπάνω workers. Ο κάθε worker είναι υπεύθυνος για την αρχικοποίηση ενός περιβάλλοντος δοκιμής καθώς και την εκτέλεση εργασιών.

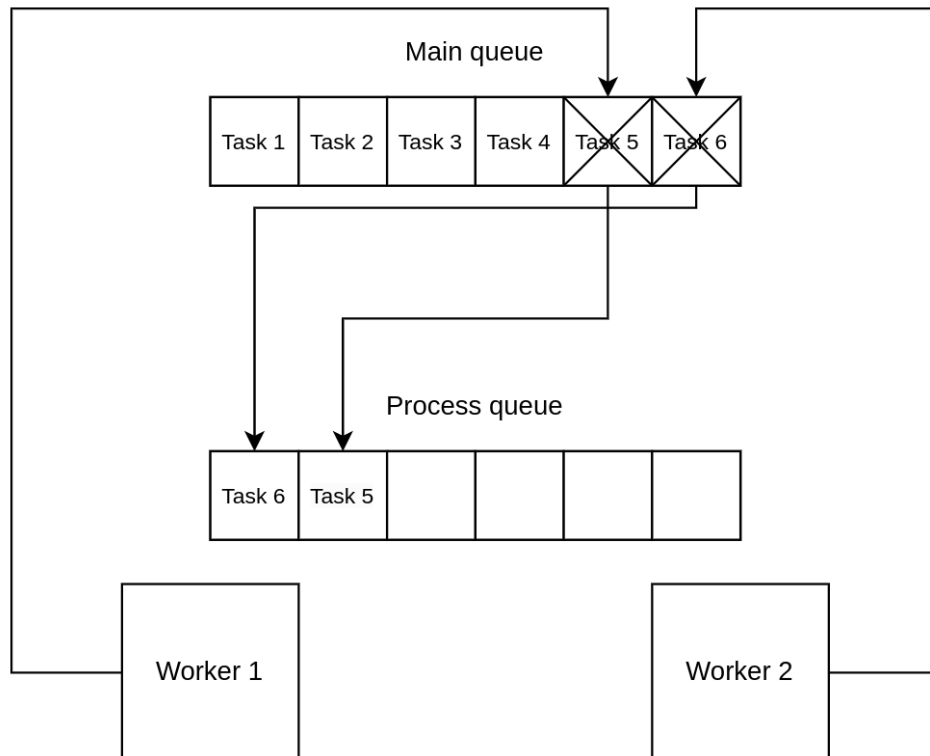
Πιο συγκεκριμένα οι workers εφαρμόζουν το μοντέλο του Fine-grained parallelism. Ολόκληρη η εργασία δηλαδή διαιρείται σε μικρότερα κομμάτια (διεργασίες) οι οποίες μοιράζονται στους workers χρησιμοποιώντας Load Balancing. Το Load Balancing είναι απαραίτητο για να μην μένει κάποιος worker σε αδράνεια σε περίπτωση που οι διεργασίες δεν έχουν μοιραστεί σωστά σύμφωνα με τον χρόνο που παίρνει η κάθε μια για την εκτέλεση. Επιτυγχάνεται μέσω της χρήσης κοινής μνήμης μεταξύ των workers. Η κοινή μνήμη που επιλέχθηκε να χρησιμοποιηθεί είναι η Βάση Δεδομένων Redis. Όπως αναφέρθηκε στην ενότητα 1.6.1, η Fine-Grained παράλληλη επεξεργασία αξιοποιείται καλύτερα σε αρχιτεκτονικές που υποστηρίζουν γρήγορη επικοινωνία για αυτόν τον λόγο η βάση δεδομένων λειτουργεί στο ίδιο περιβάλλον με τους workers, δηλαδή στον ίδιο Kubernetes cluster.

Χρησιμοποιώντας την βάση δεδομένων οι workers επικοινωνούν έμμεσα μεταξύ τους μέσω μιας ουράς διεργασιών η οποία δημιουργείται και αποθηκεύεται στην βάση δεδομένων από τον controller.

Η ουρά διεργασιών περιέχει όλες τις μοναδικές τιμές που χρειάζονται προκειμένου να εκτελεστούν όλες οι απαραίτητες διεργασίες, για παράδειγμα μια λίστα με όλα τα tests τα οποία πρέπει να εκτελεστούν. Η πρόκληση όσον αφορά την έλλειψη άμεσης επικοινωνίας μεταξύ των εργαζομένων είναι να διασφαλίσουμε ότι κανένας worker δεν εκτελεί μια διεργασία η οποία εκτελείται ήδη από έναν άλλον worker. Αυτή πρόκληση αντιμετωπίζεται χρησιμοποιώντας δύο ουρές. Η Main ουρά περιέχει όλες τις διεργασίες οι οποίες είναι έτοιμες να εκτελεστούν ενώ η Process ουρά περιέχει τις διεργασίες οι οποίες εκτελούνται από κάποιον worker.

### 2.2.1 Δέσμευση διεργασίας

Για να δεσμεύσει ένας worker μια διεργασία πρέπει να διαβάσει ένα στοιχείο από την Main ουρά και να το μετακινήσει στην Process ουρά. Είναι σημαντικό αυτές οι δύο διαδικασίες να είναι ατομικές έτσι ώστε σε περίπτωση που υπάρξει race condition μεταξύ των workers να αποφευχθούν τα deadlocks.<sup>[17]</sup> Στο παρακάτω διάγραμμα απεικονίζονται δύο workers. “Worker 1” και “Worker 2” οι οποίοι δεσμεύουν τις διεργασίες “Task 5” και “Task 6” αντίστοιχα.

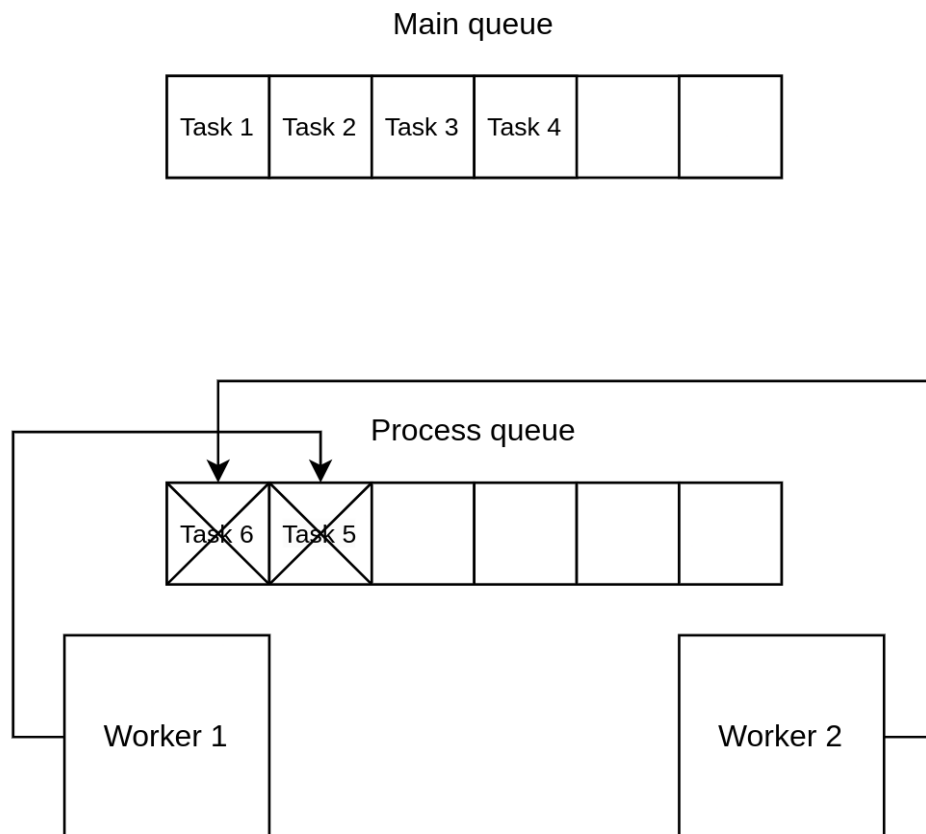


Διάγραμμα 2.2

### 2.2.2 Αποδέσμευση διεργασίας

Για να αποδεσμεύσει ένας worker μια διεργασία πρέπει απλά να τη διαγράψει από την Process ουρα όπως φαίνεται στο παρακάτω διάγραμμα.

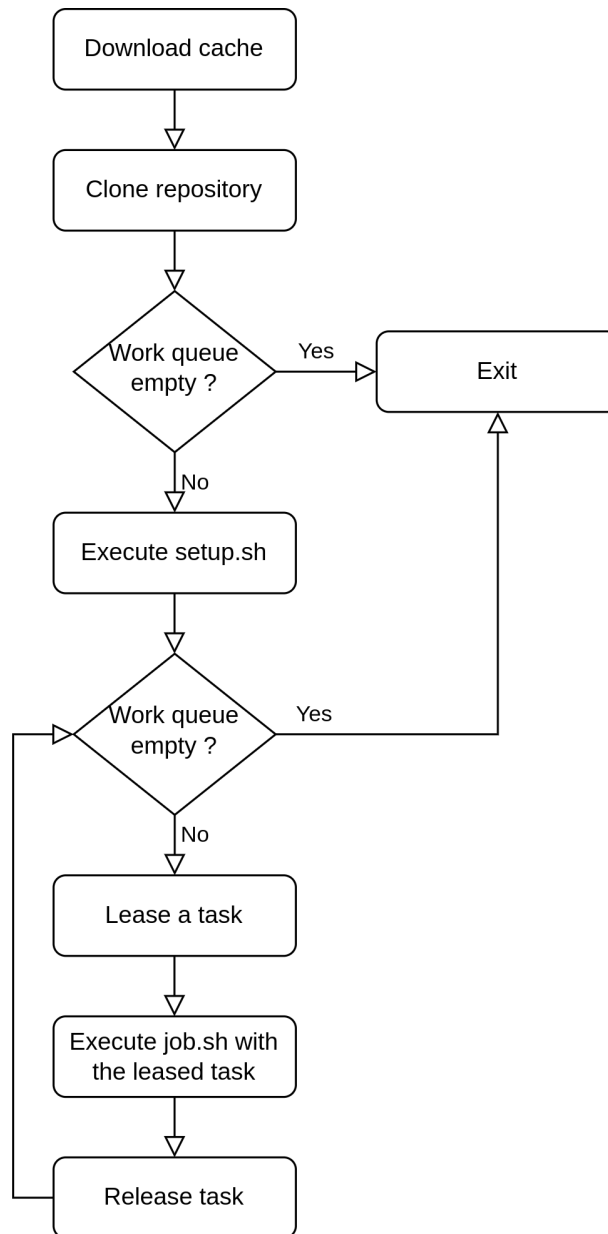
Μια εργασία θεωρείται ολοκληρωμένη όταν η Main αλλά και η Process ουρές έχουν αδειάσει.



Διάγραμμα 2.3

### 2.2.3 Διάγραμμα ροής

Στο παρακάτω διάγραμμα ροής βλέπουμε πιο αναλυτικά τα βήματα του worker.



Διάγραμμα 2.4

Ο κάθε worker ξεκινάει κατεβάζοντας το cache από προηγούμενες δοκιμές καθώς και το επιλεγμένο αποθετήριο.

Το cache παρέχεται από τον χρήστη και είναι αποθηκευμένο σε ένα Storage Bucket. Απαρτίζεται από τα εκτελέσιμα αρχεία της εφαρμογής, καθώς και τα dependencies,

τα οποία χρησιμοποιήθηκαν για τη διαδικασία του compile. Ο κύριος σκοπός της χρήσης του είναι να προσφέρει επιπλέον ταχύτητα στους workers, εξοικονομώντας τον χρόνο που θα χρειάζονταν για την επαναληπτική διαδικασία του compile της εφαρμογής και την επαναλαμβανόμενη λήψη των dependencies που χρησιμοποιούνται στην εφαρμογή αλλά και στα tests. Η χρήση του cache αποτελεί προαιρετική διαδικασία.

Έπειτα, ο worker αρχικοποιεί το περιβάλλον δοκιμής εκτελώντας το script "setup.sh" το οποίο εγκαθιστά τα dependencies των tests και κάνει compile τον κώδικα τους σύμφωνα με τις οδηγίες που έχει ορίσει ο χρήστης.

Στη συνέχεια εκτελεί το script "job.sh" μία ή περισσότερες φορές σειριακά. Το script αυτό περιέχει τις οδηγίες προς το σύστημα για το πως πρέπει να εκτελεστούν τα tests. Ο ελάχιστος αριθμός εκτελέσεων του "job.sh" από τον κάθε worker μπορεί να εκφραστεί από την συνάρτηση

$$Ne = \lfloor n / w \rfloor$$

Όπου  $n$  είναι ο αριθμός των διεργασιών στην ουρά και  $w$  είναι ο αριθμός των workers.

Σε κάθε εκτέλεση του script "job.sh" ο worker προσθέτει ένα environment variable (μεταβλητή περιβάλλοντος) με το όνομα "TASK". Η τιμές της μεταβλητής αυτής προέρχονται από την ουρά εργασιών.

#### 2.2.4 Docker container

Το παρακάτω απόσπασμα παρουσιάζει με ενδεικτικό τρόπο τμήμα του Dockerfile, το οποίο έχει ως αντικείμενο τη διαμόρφωση και τη συσκευασία του βασικού image του worker.

```
FROM golang:alpine AS build

RUN apk update && apk add --no-cache 'git=~2'

ENV GO111MODULE=on

WORKDIR $GOPATH/github.com/nickgarlis/thesis/

COPY go.mod go.sum ./
RUN go mod download

COPY pkg pkg
COPY worker/main.go worker/main.go

RUN CGO_ENABLED=0 GOOS=linux go build -o /go/worker worker/main.go
```

```
FROM ubuntu:22.04 AS base
```

```
RUN apt-get update -y \  
    && apt-get install -y software-properties-common \  
    && add-apt-repository -y ppa:git-core/ppa \  
    && apt-get update -y \  
    && apt-get install -y --no-install-recommends \  
    curl \  
    ca-certificates \  
    git \  
    git-lfs \  
    iptables \  
    jq \  
    software-properties-common \  
    sudo \  
    unzip \  
    zip \  
    && rm -rf /var/lib/apt/lists/*
```

```
RUN adduser --disabled-password --gecos "" --uid 1001 worker \  
    && groupadd docker --gid 121 \  
    && usermod -aG sudo worker \  
    && usermod -aG docker worker \  
    && echo "%sudo    ALL=(ALL:ALL) NOPASSWD:ALL" > /etc/sudoers \  
    && echo "Defaults env_keep += \"DEBIAN_FRONTEND\"" >>  
/etc/sudoers
```

```
RUN curl -fLo /usr/bin/dumb-init  
https://github.com/Yelp/dumb-init/releases/download/v1.2.5/dumb-init\_1.2.5\_x86\_64 \  
    && chmod +x /usr/bin/dumb-init
```

```
RUN set -vx; \  
    curl -fLo docker.tgz  
https://download.docker.com/linux/static/stable/x86\_64/docker-20.10.21.tgz \  
    && tar zxvf docker.tgz \  
    && install -o root -g root -m 755 docker/* /usr/bin/ \  
    && rm -rf docker docker.tgz
```

```
COPY worker/entrypoint-dind.sh worker/startup.sh worker/logger.sh  
worker/wait.sh /usr/bin/
```

```
RUN chmod +x /usr/bin/entrypoint-dind.sh /usr/bin/startup.sh

COPY worker/docker-shim.sh /usr/local/bin/docker

VOLUME /var/lib/docker

COPY --from=build /go/worker /usr/bin/worker
```

Για τη διαδικασία αυτή, χρησιμοποιήθηκε το multi-stage build, ένα πολύτιμο εργαλείο του Docker για τη βελτιστοποίηση της διαχείρισης των images.<sup>[18]</sup>

Το Dockerfile ξεκινά με το πρώτο στάδιο της διαδικασίας (build), όπου το Docker επεκτείνει το image "golang:alpine". Κατά την επέκταση αυτή εγκαθίστανται όλα τα απαραίτητα dependencies, δημιουργώντας έτσι ένα περιβάλλον στο οποίο ο κώδικας του worker μπορεί να διαμορφωθεί και να γίνει compile με αποτελεσματικότητα.

Στο δεύτερο στάδιο της διαδικασίας (base), το Docker επεκτείνει το image "ubuntu:22.04". Κατά την επέκταση αυτή, γίνεται εγκατάσταση σειράς απαραίτητων πακέτων, όπως το git, curl, sudo και άλλα. Στη συνέχεια, δημιουργείται ο χρήστης που θα είναι υπεύθυνος για την εκτέλεση των tests και προστίθεται σε μια ομάδα που του παρέχει τα δικαιώματα διαχείρισης του docker, το οποίο εγκαθίσταται αργότερα μαζί με το dumb-init.

Καταλήγει, αντιγράφοντας κάποια scripts που αποτελούν την εκκίνηση του worker και επιτρέπουν τη σύνδεση με το docker. Η διαδικασία κλείνει με την αντιγραφή του binary αρχείου από το πρώτο στάδιο (build), καταλήγοντας σε μια πλήρως λειτουργική εικόνα του worker, έτοιμη για χρήση.

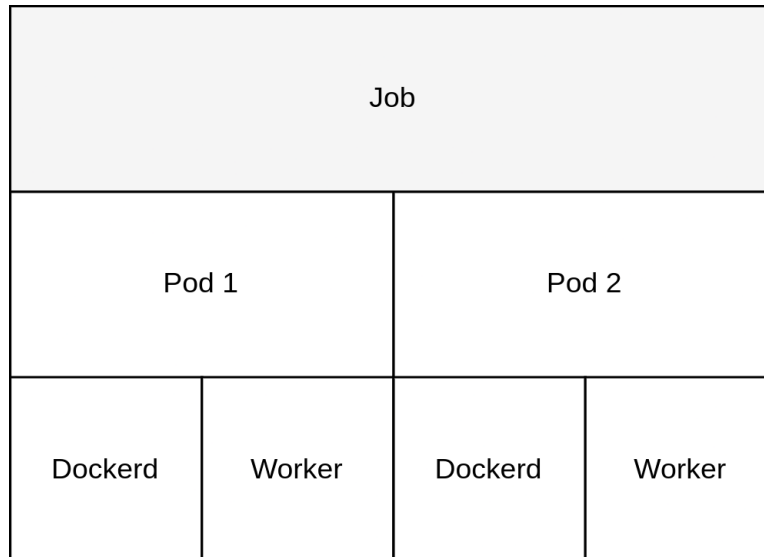
Μία από τις πρωταρχικές προκλήσεις που αντιμετωπίστηκαν κατά τη συσκευασία του "controller" μέσα σε ένα Docker container, αφορούσε την υλοποίηση της τεχνολογίας "Docker in Docker" (DinD).<sup>[19]</sup> Πολλά test frameworks επιτρέπουν την δημιουργία Docker containers, μιμούμενα ενδεικτικά διάφορες υπηρεσίες όπως βάσεις δεδομένων, που είναι απαραίτητες για την ολοκληρωμένη διεξαγωγή ενός test.

Ειδικότερα, ένα συγκεκριμένο test μπορεί να απαιτεί την προσομοίωση μιας βάσης δεδομένων, όπως την Postgres, για να δοκιμάσει την αποτελεσματική ενσωμάτωση (integration) της εφαρμογής με αυτήν. Επομένως, αποτελεί κρίσιμο παράγοντα ο κάθε worker να είναι σε θέση να δημιουργεί αυτόνομα τα δικά του child containers.

Ωστόσο, καθώς ο κάθε worker προσλαμβάνει τη μορφή ενός docker container, ανακύπτει η ανάγκη για την εφαρμογή μιας εξειδικευμένης λύσης, η οποία είναι γνωστή ως "Docker in Docker" (DinD). Αυτό σημαίνει ότι κάθε worker container πρέπει να συνοδεύεται από ένα Dockerd (docker:dind) container, με το οποίο επικοινωνεί μέσω του πρωτοκόλλου TCP.

Στο πλαίσιο του Kubernetes, οι workers βρίσκονται υπό την διαχείριση ενός Job το

οποίο προκαλεί τη δημιουργία Pods όπου το καθένα περιέχει έναν worker container και έναν container του docker daemon. Ο αριθμός των Pods που το Job θα δημιουργήσει και θα διαχειριστεί είναι ίσος με τον παράμετρο "parallelism" στο Job.



Διάγραμμα 2.5

## 2.3 Controller

Ο controller είναι ένα REST API που έχει γραφτεί χρησιμοποιώντας την γλώσσα Go. Η δουλειά του είναι να αποδέχεται αιτήματα από clients για τον προγραμματισμό εργασιών. Συνδέεται με τη βάση δεδομένων Redis για να δημιουργήσει την ουρά εργασιών και επικοινωνεί επίσης με το Kube API για να δημιουργήσει τους απαιτούμενους πόρους του Kubernetes (workers).

Πιο αναλυτικά, ο controller αποτελείται από έξι HTTP endpoints.

Το **POST /jobs** είναι υπεύθυνο για την φόρτωση της ουράς εργασιών και τη δημιουργία των πόρων του Kubernetes για την εκκίνηση των workers. Αποδέχεται JSON στην μορφή:

```
{
  "tasks":[
    "task1","task2","task3"
  ],
  "workers":1,
  "commitSha":"56e05fced"
}
```

Στον παραπάνω απόσπασμα κώδικα, παρουσιάζονται τρεις προδιαγραφές μεταβλητών. Η πρώτη είναι η “tasks”, η οποία αποτελεί έναν πίνακα που συγκεντρώνει όλες τις διεργασίες που πρόκειται να εκτελεστούν. Στην συνέχεια, έχουμε την “workers” που αναπαριστά τον αριθμό των workers που θα δημιουργηθούν μέσω του controller. Τέλος, το “commitSha” αναφέρεται στο συγκεκριμένο commit του αποθετηρίου, το οποίο αντιστοιχεί στον κώδικα που ο χρήστης επιθυμεί να επαληθεύσει.

Το **GET /jobs/{id}** όπου id είναι το id της εργασίας . Αυτό το endpoint επιστρέφει την κατάσταση του κάθε worker καθώς και την ώρα εκκίνησης και τερματισμού στην μορφή:

```
{
  "id": "1234",
  "tasks": [
    "task1",
    "task2",
    "task3",
    "task4"
  ],
  "workers": 2,
  "start": 1685537986702,
  "workersStatus": [
    {
      "id": "1a3309",
      "failed": false,
      "end": 1685538730178
    },
    {
      "id": "cdf921",
      "failed": false,
      "end": 1685538729883
    }
  ],
  "commitSha": "56e05fced"
```

Στο απόσπασμα κώδικα παρατηρούνται τόσο γνωστές μεταβλητές από το προηγούμενο απόσπασμα, όσο και νέες προσθήκες.

Εμφανίζεται πρώτα το "id", ένα μοναδικό αναγνωριστικό τύπου UUID, το οποίο χρησιμοποιείται για τη μοναδική ταυτοποίηση κάθε εργασίας.

Επακολουθεί το "start", ένα timestamp σε milliseconds, καταγράφοντας την ακριβή στιγμή έναρξης της εκτέλεσης της εργασίας.

Ακόμη, παρουσιάζεται το "workersStatus". Πρόκειται για μια λίστα που καταγράφει πληροφορίες σχετικά με την κατάσταση κάθε worker, συμπεριλαμβάνοντας το id του worker, τη μεταβλητή "failed", που γίνεται αληθής όταν τα tests παράγουν exit code διάφορο του μηδενός και το "end", ένα timestamp σε milliseconds, το οποίο αναπαριστά την χρονική στιγμή κατά την οποία ο worker σταμάτησε τη λειτουργία του, υποδεικνύοντας έτσι την τελική φάση της εργασίας του.

Το **DELETE /jobs/{id}** όπου id είναι το id της εργασίας. Αυτό το endpoint διαγράφει την ουρά εργασίας από την βάση δεδομένων καθώς και τους workers στο Kubernetes ακυρώνοντας έτσι την εργασία.

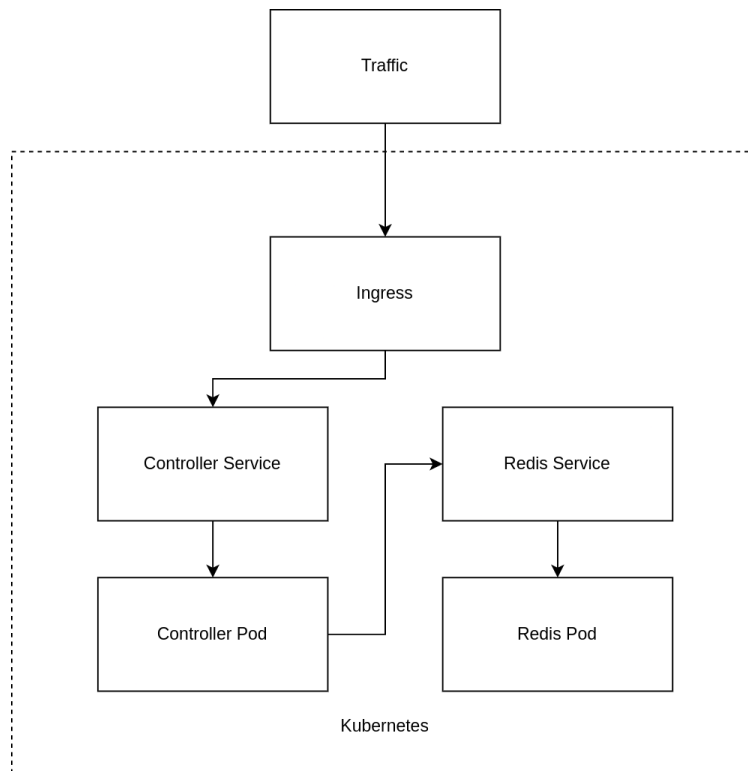
Το **GET /logs/{jobId}/{workerId}** όπου jobId είναι το id της εργασίας και workerId είναι το id ενός worker. Αυτό το endpoint επιστρέφει το αρχείο καταγραφής logs ενός συγκεκριμένου worker σε μορφή TXT. Παρακάτω φαίνεται ένα μικρό απόσπασμα των logs ενός worker.

```
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 51.542 s
- in org.keycloak.it.storage.database.dist.TransactionConfigurationDistTest
[INFO] Running org.keycloak.it.storage.map.HotRodStoreDistTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 25.647 s
- in org.keycloak.it.storage.map.HotRodStoreDistTest
[INFO] Running org.keycloak.it.storage.map.JPAStoreDistTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 23.064 s
- in org.keycloak.it.storage.map.JPAStoreDistTest
[INFO] Running org.keycloak.it.storage.map.MixedStoreDistTest
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 23.509 s
- in org.keycloak.it.storage.map.MixedStoreDistTest
[INFO]
[INFO] Results:
[INFO]
[INFO] Tests run: 4, Failures: 0, Errors: 0, Skipped: 0
[INFO]
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 02:13 min
[INFO] Finished at: 2023-06-11T21:39:19Z
```

Τέλος, το **GET /status**. Αυτό το endpoint επιστρέφει την κατάσταση του controller.

```
{"status": "ok"}
```

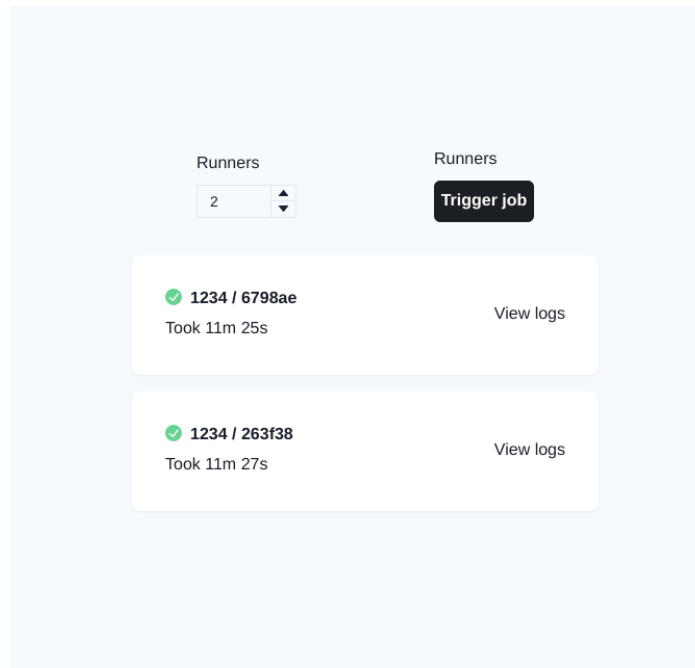
Στο πλαίσιο του Kubernetes, ο controller αποτελεί ένα Deployment. Το Deployment προκαλεί τη δημιουργία ενός Pod το οποίο συνδέεται με τη Redis μέσω ενός Service. Ο controller εκτίθεται στο δίκτυο του Kubernetes μέσω ενός άλλου Service, στο οποίο ένα Ingress προωθεί εξωτερικό traffic (κίνηση).



Διάγραμμα 2.6

## 2.4 Client

Ο client, που έχει δημιουργηθεί με το Next.js και έχει γραφεί σε TypeScript, επικοινωνεί με τον Controller για τον προγραμματισμό εργασιών και παρέχει ενημερώσεις πραγματικού χρόνου σχετικά με την πρόδοό τους. Επιπλέον, μέσω του client, η ανάκτηση και εμφάνιση των σχετικών αρχείων logs των workers γίνεται με ευκολία, προσφέροντας πλήρη εικόνα και κατανόηση της διαδικασίας συνολικά.



Διάγραμμα 2.7

```
5247 -- [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/junit/platform/junit-platform-commons/1.3.2/junit-platf
5248 -- [INFO] Downloading from central: https://repo.maven.apache.org/maven2/org/opentest4j/opentest4j/1.1.1/opentest4j-1.1.1.jar
5249 -- [INFO] Downloaded from central: https://repo.maven.apache.org/maven2/org/opentest4j/opentest4j/1.1.1/opentest4j-1.1.1.jar (7.1 k
5250 -- [INFO]
5251 -- [INFO] -----
5252 -- [INFO] T E S T S
5253 -- [INFO] -----
5254 [INFO] Running org.keycloak.it.storage.database.dist.DatabaseOptionsDistTest
5255 [INFO] Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 76.585 s - in org.keycloak.it.storage.database.dist.Database
5256 [INFO] Running org.keycloak.it.storage.database.dist.MariaDBDistTest
5257 [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 37.647 s - in org.keycloak.it.storage.database.dist.MariaDBD
5258 [INFO] Running org.keycloak.it.storage.database.dist.MySQLDistTest
5259 [INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 86.877 s - in org.keycloak.it.storage.database.dist.MySQLDis
5260 [INFO] Running org.keycloak.it.storage.database.dist.PostgreSQLDistTest
5261 [INFO] Tests run: 6, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 132.598 s - in org.keycloak.it.storage.database.dist.Postgre
5262 -- [INFO]
5263 -- [INFO] Results:
5264 -- [INFO]
5265 -- [INFO] Tests run: 10, Failures: 0, Errors: 0, Skipped: 0
5266 -- [INFO]
5267 -- [INFO] -----
5268 -- [INFO] BUILD SUCCESS
5269 -- [INFO] -----
5270 -- [INFO] Total time: 05:45 min
5271 -- [INFO] Finished at: 2023-06-11T21:43:04Z
5272 -- [INFO] -----
```

Διάγραμμα 2.8

## 2.5 Database

Η Βάση δεδομένων που επιλέχθηκε είναι η Redis. Επιλέχθηκε γιατί είναι μια in-memory βάση δεδομένων το οποίο σημαίνει πως οι εγγραφές και αναγνώσεις είναι πολύ αποδοτικές.[20] Επίσης υποστηρίζει transactions[21] για ατομικές διαδικασίες οι οποίες όπως αναφέρθηκε στο κεφάλαιο 2.2 είναι απαραίτητες για τους workers μαζί με την υψηλή απόδοση.

## 2.6 Storage

Το Google Cloud Storage χρησιμοποιήθηκε ως αποθηκευτικός χώρος για τα στατικά αρχεία. Ο κάθε worker κατεβάζει από αυτό το cache των προηγούμενων εκτελέσεων και ανεβάζει τα αρχεία καταγραφής (logs) του στο τέλος της κάθε εκτέλεσης.

## 2.7 Container registry

Οι συσκευασμένες εικόνες Docker για τον worker και τον controller έπρεπε να φιλοξενηθούν σε ένα "Container Registry". Με αυτόν τον τρόπο, οι κόμβοι του Kubernetes cluster θα μπορούσαν να κατεβάσουν το συσκευασμένο κώδικα και να τον εκτελέσουν. Το Container Registry που χρησιμοποιήθηκε ήταν το Google Container Registry.

## 2.8 Kubernetes cluster

### 2.8.1 Google Cloud

Το Google Cloud φιλοξένησε τον cluster ο οποίος εκτέλεσε τα tests. Χρησιμοποιήθηκαν 8 Virtual Machines με κωδική ονομασία n1-standard-4. Το καθένα παρείχε 4 vCPUs καθώς και 15GB RAM.<sup>[22]</sup> Χρησιμοποιήθηκε ακόμα μία στατική IP και ένας Load Balancer για να κάνει ανακατεύθυνση της εξωτερικής κίνησης του διαδικτύου προς τον Kubernetes cluster.

### 2.8.2 Terraform

Το Terraform χρησιμοποιήθηκε προκειμένου να βελτιώσει την ανάπτυξη της υποδομής του Google Cloud επιτρέποντας την περιγραφή της μέσω αρχείων με τη μορφή κώδικα.

### 2.8.3 Helm

Το Helm χρησιμοποιήθηκε προκειμένου να συσκευαστούν οι πόροι του Kubernetes cluster ως "Charts". Αυτό επέτρεψε την εύκολη ανάπτυξη και εγκατάσταση του συστήματος σε διάφορους clusters.

## 2.9 Πειραματικό περιβάλλον

Το παρόν σύστημα, το οποίο έχει υλοποιηθεί με έμφαση στην ευελιξία και την προσαρμοστικότητα, αποτελεί μια γενική πλατφόρμα που μπορεί να προσαρμοστεί και να τροποποιηθεί γρήγορα από τον χρήστη, ώστε να ανταποκρίνεται στις μοναδικές απαιτήσεις της εφαρμογής του. Η εξατομίκευση του συστήματος μπορεί να επιτευχθεί μέσω τριών βημάτων.

Το πρώτο βήμα περιλαμβάνει τη σύνταξη του `job.sh`, ενός script που καθορίζει τον τρόπο εκτέλεσης των διάφορων tasks, προσφέροντας έτσι ακόμη μεγαλύτερο έλεγχο στον χρήστη.

Το δεύτερο βήμα είναι η σύνταξη του shell script `setup.sh`, ενός κρίσιμου εργαλείου που επιτρέπει την προσαρμογή της εγκατάστασης και της διαμόρφωσης του συστήματος.

Τέλος, το τρίτο βήμα αφορά τη δημιουργία ενός εξατομικευμένου Docker image, που θα είναι ειδικά σχεδιασμένο για την εφαρμογή, προσφέροντας ένα περιβάλλον εκτέλεσης που είναι πλήρως συμβατό με τις ανάγκες της. Το image αυτό περιλαμβάνει τα `script setup.sh` και `job.sh`.

Για τις ανάγκες των πειραμάτων, επιλέχθηκε το αποθετήριο Keycloak του οργανισμού Redhat. Πρόκειται για μια ευρέως αναγνωρισμένη, open-source πλατφόρμα διαχείρισης ταυτοποίησης και εξουσιοδότησης (Identity and Access Management - IAM), που εξασφαλίζει και διαχειρίζεται την πρόσβαση σε εφαρμογές και υπηρεσίες.<sup>[23]</sup>

Το ανοικτό αποθετήριο του Keycloak επιτρέπει σε κάθε ενδιαφερόμενο να παρακολουθεί την πορεία της ανάπτυξής του, καθώς και τις διάφορες εκτελέσεις των tests που διεξάγονται, μέσα από το αποθετήριό του στο Github.

Τα συγκεκριμένα tests, τα οποία αξιολογήθηκαν μέσα από την οπτική του κατανεμημένου συστήματος εκτέλεσης, ονομάζονται "Storage tests" και βρίσκονται εντός του αποθετηρίου. Μία εκτέλεση αυτής της συλλογής test αποτελείται από οκτώ διαφορετικά test suites όπου το καθένα εξετάζει την ορθή ενσωμάτωση του Keycloak με τις διάφορες βάσεις δεδομένων που υποστηρίζει. Έτσι, κάθε test επιβεβαιώνει την αξιοπιστία και την αποδοτικότητα του Keycloak σε διάφορα περιβάλλοντα βάσεων δεδομένων, ενισχύοντας την πρακτική του χρήση.

### 2.9.1 Σύνταξη `setup.sh`

Στο παρακάτω απόσπασμα κώδικα, αντικρίζουμε την υλοποίηση του `setup.sh` script, μιας κρίσιμης ενότητας της εφαρμογής.

```
#!/bin/bash
set -eu

docker pull postgres:alpine
docker pull mariadb:10.5.9
docker pull mysql:latest
docker pull quay.io/infinispan/server:14.0.9.Final
docker pull mcr.microsoft.com/mssql/server:2019-latest

git clone --branch main --depth 1
https://github.com/keycloak/keycloak.git && cd keycloak
git fetch --depth 1 origin $COMMIT_SHA
git checkout $COMMIT_SHA

./mvnw install -nsu -B -pl quarkus/tests/integration -am
-DskipTests
```

Η διαδικασία που περιγράφεται εδώ ξεκινάει κατεβάζοντας τα Docker images τα οποία περιλαμβάνουν τις αναγκαίες βάσεις δεδομένων για την εκτέλεση των integration tests. Αυτές οι βάσεις δεδομένων είναι οι κεντρικοί παράγοντες που χρησιμοποιούνται για τη δοκιμή και την επικύρωση της εφαρμογής.

Ακολουθεί η διαδικασία του git clone, μια κρίσιμη ενέργεια που επιτρέπει την αντιγραφή του αποθετηρίου στον worker. Αμέσως μετά πραγματοποιείται η χρήση της εντολής git checkout, η οποία επιτρέπει την μετάβαση στο συγκεκριμένο commit που έχει επιλεγεί από τον χρήστη μέσω της παραμέτρου commitSha.

Σε αυτό το σημείο, η εγκατάσταση των dependencies ξεκινάει με την χρήση του Maven, ενός ισχυρού εργαλείου που διευκολύνει τη διαχείριση των dependencies στην Java.<sup>[24]</sup> Αυτά τα dependencies είναι ζωτικής σημασίας για την εκτέλεση του προγράμματος, καθώς περιλαμβάνουν βιβλιοθήκες και πακέτα που η εφαρμογή αναμένει να είναι παρόντα.

Τέλος, σημειώνεται ότι οι εντολές για την εγκατάσταση αυτών των dependencies, καθώς και για την εκτέλεση των tests, μπορούν να βρεθούν στο αποθετήριο του Keycloak. Αυτό δείχνει την ενσωμάτωση και την συμβατότητα του setup.sh script με τον ευρύτερο υπόστρωμα του Keycloak, καθώς οι χρήστες μπορούν να ανατρέξουν στο αποθετήριο για επιπλέον πληροφορίες και κατευθυντήριες γραμμές.

### 2.9.2 Σύνταξη job.sh

Το παρακάτω απόσπασμα κώδικα παρουσιάζει την λεπτομερή υλοποίηση του script job.sh. Αυτό το script, που είναι ένας κεντρικός πυλώνας του συστήματος, εκτελεί ζωτικής σημασίας λειτουργίες, παρόμοια με το setup.sh, αλλά με ορισμένες σημαντικές διαφορές.

```
#!/bin/bash
./mvnw test -nsu -B -pl quarkus/tests/integration -Ptest-database
-Dtest=${TASK}
```

Όπως συνέβη κατά τη σύνταξη του `setup.sh`, το εργαλείο Maven χρησιμοποιείται για την εκτέλεση των `tests`, καθιστώντας τη διαδικασία πιο αποδοτική και αυτοματοποιημένη. Ωστόσο, το πραγματικά ξεχωριστό στοιχείο αυτού του script είναι η χρήση της μεταβλητής περιβάλλοντος `"TASK"`.

Η μεταβλητή `"TASK"` αντιπροσωπεύει ένα ή περισσότερα `tests`, ανάλογα με την διαίρεση που ορίστηκε από τον χρήστη. Αν για παράδειγμα το `granularity` ήταν `Coarse-Grained` τότε η κάθε μεταβλητή `"TASK"` μπορεί να αντιπροσωπεύει πολλαπλά `tests`. Η χρήση της μεταβλητής `"TASK"` είναι σημαντική, καθώς επιτρέπει στο σύστημα να προσαρμοστεί στις μοναδικές ανάγκες της εφαρμογής, εκτελώντας συγκεκριμένα `tests` σε συγκεκριμένους `workers`.

Όπως αναφέρθηκε προηγουμένως στην ενότητα 2.2, το script `job.sh` εκτελείται πολλές φορές, κάθε φορά με διαφορετικές τιμές για τη μεταβλητή `"TASK"`. Αυτή η δυνατότητα προσαρμογής και επανάληψης καθιστά το `job.sh` ιδιαίτερα προσαρμοστικό και ευέλικτο στον χειρισμό διαφορετικών προκλήσεων και διαμορφώσεων `testing`.

### 2.9.3 Σύνταξη Dockerfile

Για να συντάξει κανείς το εξατομικευμένο `Dockerfile` πρέπει να επεκτείνει το βασικό `docker image` του `worker` το οποίο αναφέρθηκε στην ενότητα 2.2.4. Τα scripts `setup.sh` και `job.sh` πρέπει να ενσωματωθούν στο κατασκευασμένο `image`. Στο παρακάτω απόσπασμα κώδικα φαίνεται η διαδικασία με την οποία έγινε αυτό για το `keycloak`. Το `Dockerfile` ξεκινάει κατεβάζοντας το `image` του `worker` για να το επεκτείνει και συνεχίζει εγκαθιστώντας το `OpenJDK 17` και το `Maven` τα οποία χρησιμοποιούνται από το `Keycloak`. Στη συνέχεια αντιγράφει το `setup.sh` και το `job.sh` τα οποία συντάχτηκαν. Επιπροσθέτως ορίζει τον χρήστη καθώς και την εντολή που θα εκτελέσει όταν το `Kubernetes` δημιουργήσει το `container` το οποίο χρησιμοποιεί αυτό το `image`.

Η διαδικασία σύνταξης ενός εξατομικευμένου `Dockerfile` καταλαμβάνει πρωταγωνιστική θέση στην υλοποίηση του συστήματος. Αυτό απαιτεί την επέκταση της βασικής εικόνας `Docker` του `worker`, όπως αναφέρεται στην ενότητα 2.2.4, με προσθήκη κατάλληλων στοιχείων για τη συγκεκριμένη εφαρμογή. Ανάμεσα στα στοιχεία αυτά, περιλαμβάνεται η ενσωμάτωση των scripts `setup.sh` και `job.sh`, τα οποία αποτελούν απαραίτητο μέρος της τελικής εικόνας `Docker`.

Στο ακόλουθο απόσπασμα κώδικα, παρουσιάζεται η ακριβής διαδικασία.

```
FROM gcr.io/distributed-tests-370616/worker
```

```

RUN apt-get update -y \
    && apt-get install -y --no-install-recommends \
    openjdk-17-jdk \
    maven \
    && rm -rf /var/lib/apt/lists/*

COPY worker/setup.sh worker/job.sh /usr/bin/
USER worker

ENTRYPOINT ["/bin/bash", "-c"]
CMD ["entrypoint-dind.sh"]

```

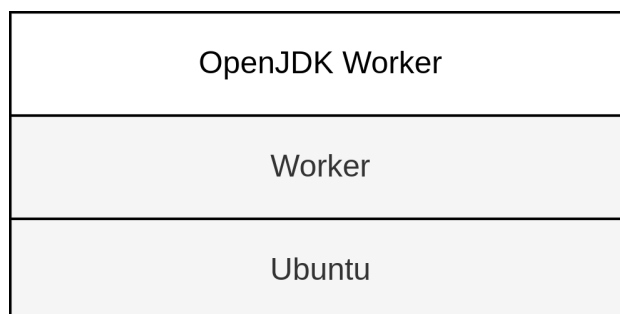
Αρχικά το Dockerfile επεκτείνει τη βασική εικόνα του worker. Συγκεκριμένα, πραγματοποιείται λήψη της αρχικής εικόνας του worker από το Google Artifact Registry. Η λήψη αυτή αποτελεί τον πυρήνα του Dockerfile, πάνω στον οποίο θα προστεθούν οι επιπλέον επιθυμητές λειτουργίες.

Έπειτα, γίνεται εγκατάσταση του OpenJDK 17 και του Maven, καθώς αυτά αποτελούν απαραίτητα εργαλεία για τη λειτουργία του Keycloak. Το OpenJDK 17 είναι ένας open-source εκτελεστής Java, ο οποίος χρησιμοποιείται για την εκτέλεση των εφαρμογών Java, ενώ το Maven αποτελεί ένα εργαλείο για τη διαχείριση και την αυτοματοποίηση του software project.

Ακολουθεί η αντιγραφή των scripts setup.sh και job.sh. Αυτά τα scripts, τα οποία έχουν προηγουμένως συνταχθεί, ενσωματώνονται στην εικόνα Docker, καθορίζοντας έτσι τη συμπεριφορά του worker.

Τέλος, το Dockerfile καθορίζει τον χρήστη που θα εκτελεί τις εντολές, καθώς και την εντολή που θα εκτελείται κατά την δημιουργία του container από το Kubernetes. Αυτή η εντολή, που είναι ουσιαστικά η entry point της εικόνας Docker, ορίζει την πρωταρχική συμπεριφορά του container μόλις ξεκινήσει.

Στο παρακάτω διάγραμμα έχουμε μια εικονική αναπαράσταση των κομματιών από τα οποία αποτελείται ο τελικός worker.



Διάγραμμα 2.9

## Κεφάλαιο 3

### Αξιολόγηση των αποτελεσμάτων

Πραγματοποιήθηκαν τρία ξεχωριστά πειράματα, στο καθένα από τα οποία εφαρμόστηκε μια διαφορετική μεθοδολογία, για να παρατηρηθεί πώς αυτές οι μεταβλητές επηρεάζουν την απόδοση των δοκιμών.

#### 3.1 Σειριακή εκτέλεση (Coarse-Grained)

Στο πείραμα αυτό, όλες οι δοκιμές εκτελέστηκαν σε έναν μόνο worker με Coarse-Grained διαχείριση διεργασιών. Αυτό σημαίνει ότι η συλλογή των tests χωρίστηκε σε ένα μόνο ένα μεγάλο, αυτόνομο κομμάτι. Το σενάριο αυτό επιλέχθηκε προκειμένου να αποφευχθεί επιπρόσθετη κίνηση στο δίκτυο για τη διαχείριση διεργασιών.

Τα αποτελέσματα ήταν τα εξής.

```
{
  "id": "1234",
  "tasks": [
    "PostgreSQLDistTest, MariaDBDistTest#testSuccessful, MySQLDistTest#testSuccessful, DatabaseOptionsDistTest, JPAStoreDistTest, HotRodStoreDistTest, MixedStoreDistTest, TransactionConfigurationDistTest"
  ],
  "workers": 1,
  "start": 1687176923150,
  "workersStatus": [
    {
      "id": "42c6f4",
      "failed": false,
      "end": 1687178043400
    }
  ],
  "commitSha": "38994de056bdc38749a2c264f49fffc3dfcbc1e5"
}
```

Ο χρόνος εκτέλεσης  $T_s$  υπολογίζεται μέσω της πράξης  $(end - start) / 1000$ . Οι μεταβλητές  $end$  και  $start$  βρίσκονται στο παραπάνω απόσπασμα κώδικα και αποτελούν timestamps σε milliseconds.

$$T_s = 1120.25s$$

## 3.2 Σειριακή εκτέλεση (Fine-Grained)

Στο δεύτερο πείραμα, ακολουθήθηκε η ίδια σειριακή μεθοδολογία, με τη διαφορά ότι οι διεργασίες διαμερίστηκαν σε "Fine-Grained" κομμάτια. Αυτό σημαίνει ότι το test suite χωρίστηκε σε ένα κομμάτι για κάθε test. Αυτή η προσέγγιση επιτρέπει την αξιολόγηση του πρόσθετου φορτίου στο δίκτυο σε σύγκριση με το πρώτο πείραμα.

Τα αποτελέσματα ήταν τα εξής.

```
{
  "id": "1234",
  "tasks": [
    "PostgreSQLDistTest",
    "MariaDBDistTest#testSuccessful",
    "MySQLDistTest#testSuccessful",
    "DatabaseOptionsDistTest",
    "JPASoreDistTest",
    "HotRodStoreDistTest",
    "MixedStoreDistTest",
    "TransactionConfigurationDistTest"
  ],
  "workers": 1,
  "start": 1687175558487,
  "workersStatus": [
    {
      "id": "169550",
      "failed": false,
      "end": 1687176791194
    }
  ],
  "commitSha": "38994de056bdc38749a2c264f49fffc3dfcbc1e5"
}
```

$$T_p = 1232.707s$$

Εκτελέστηκαν δηλαδή σε περισσότερο χρόνο από ότι χρειάστηκε για να εκτελεστούν χωρίς την χρήση της ουράς διεργασιών. Το αποτέλεσμα αυτό είναι αναμενόμενο καθώς η κάθε ανάγνωση/εγγραφή από/προς την βάση δεδομένων είναι ένα επιπρόσθετο φορτίο  $p$  για το σύστημα. Επιπλέον, πρέπει να λάβουμε υπόψιν ότι το Maven (το εργαλείο κατασκευής κώδικα που χρησιμοποιείται από το Keycloak) χρειάζεται κάποιον απροσδιόριστο χρόνο  $c$  για την εκκίνηση ενός test περιβάλλοντος ο οποίος επαναλαμβάνεται κάθε φορά που καλείται από το shell.

### 3.3 Παράλληλη εκτέλεση (Fine-Grained)

Στο τελευταίο πείραμα, τα δοκιμαστικά tests εκτελέστηκαν τρεις φορές παράλληλα με δύο, τέσσερεις και οκτώ workers αντίστοιχα, χρησιμοποιώντας επίσης "Fine-Grained" διαχείριση διεργασιών. Τα αποτελέσματα ήταν τα εξής.

Με δύο workers:

$$T_p = 1058.475s$$

```
{
  "id": "1234",
  "tasks": [
    "PostgreSQLDistTest",
    "MariaDBDistTest#testSuccessful",
    "MySQLDistTest#testSuccessful",
    "DatabaseOptionsDistTest",
    "JPASoreDistTest",
    "HotRodStoreDistTest",
    "MixedStoreDistTest",
    "TransactionConfigurationDistTest"
  ],
  "workers": 2,
  "start": 1687174062459,
  "workersStatus": [
    {
      "id": "a8907f",
      "failed": false,
      "end": 1687175120934
    },
    {
      "id": "dcf883",
      "failed": false,
      "end": 1687175122617
    }
  ],
  "commitSha": "38994de056bdc38749a2c264f49fffc3dfcbc1e5"
}
```

Με τέσσερις workers:

$$T_p = 840.928s$$

```
{
  "id": "1234",
  "tasks": [
    "PostgreSQLDistTest",
    "MariaDBDistTest#testSuccessful",
    "MySQLDistTest#testSuccessful",
    "DatabaseOptionsDistTest",
    "JPASStoreDistTest",
    "HotRodStoreDistTest",
    "MixedStoreDistTest",
    "TransactionConfigurationDistTest"
  ],
  "workers": 4,
  "start": 1687171347317,
  "workersStatus": [
    {
      "id": "b9ce83",
      "failed": false,
      "end": 1687172188245
    },
    {
      "id": "2c9c12",
      "failed": false,
      "end": 1687172187542
    },
    {
      "id": "e1fec2",
      "failed": false,
      "end": 1687172188748
    },
    {
      "id": "93d9e1",
      "failed": false,
      "end": 1687172189351
    }
  ],
  "commitSha": "38994de056bdc38749a2c264f49fffc3dfcbc1e5"
}
```

Με οκτώ workers:

$$T_p = 751.756s$$

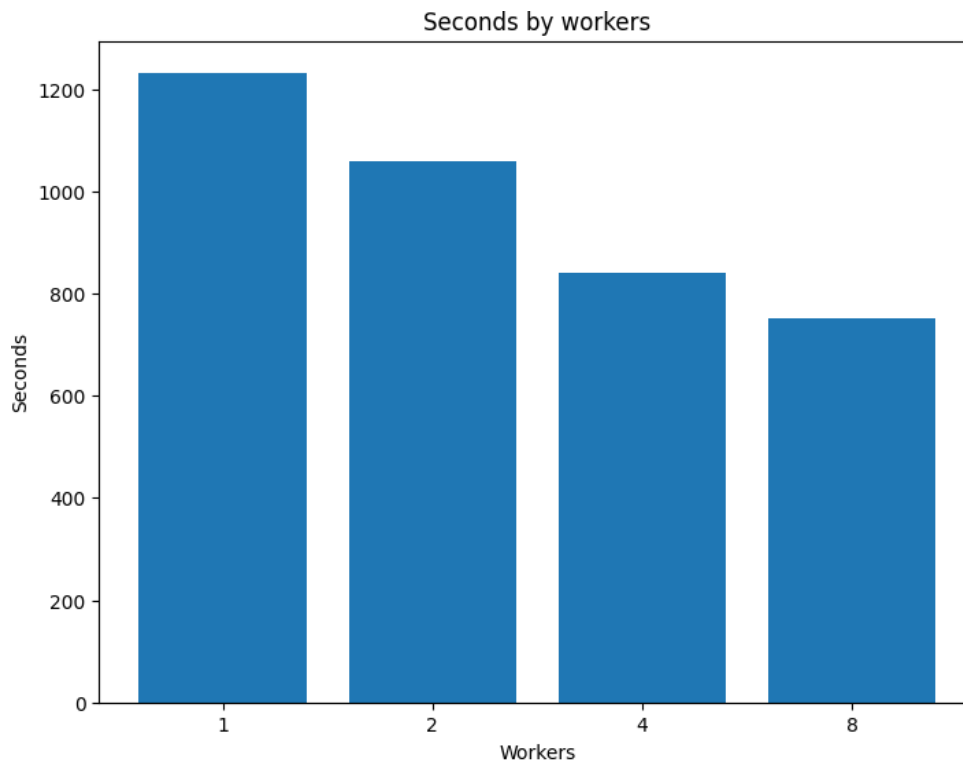
```
{
  "id": "1234",
  "tasks": [
    "PostgreSQLDistTest",
    "MariaDBDistTest#testSuccessful",
    "MySQLDistTest#testSuccessful",
    "DatabaseOptionsDistTest",
    "JPASoreDistTest",
    "HotRodStoreDistTest",
    "MixedStoreDistTest",
    "TransactionConfigurationDistTest"
  ],
  "workers": 8,
  "start": 1687173128719,
  "workersStatus": [
    {
      "id": "2517f9",
      "failed": false,
      "end": 1687173880475
    },
    {
      "id": "489105",
      "failed": false,
      "end": 1687173880879
    },
    {
      "id": "398c0a",
      "failed": false,
      "end": 1687173879169
    },
    {
      "id": "7aa250",
      "failed": false,
      "end": 1687173878914
    },
    {
      "id": "62007d",
```

```

    "failed":false,
    "end":1687173880576
  },
  {
    "id":"d84003",
    "failed":false,
    "end":1687173879270
  },
  {
    "id":"5d22bd",
    "failed":false,
    "end":1687173880677
  },
  {
    "id":"393ae7",
    "failed":false,
    "end":1687173879671
  }
],
"commitSha":"38994de056bdc38749a2c264f49fffc3dfcbc1e5"
}

```

Παρατηρώντας τα αποτελέσματα, διαπιστώνεται ότι τα πειράματα που εκτελέστηκαν παράλληλα με Fine-Grained διαχείριση των tests ολοκληρώθηκαν πιο γρήγορα σε σύγκριση με την σειριακή εκτέλεση με Coarse-Grained διαχείριση των tests. Αυτό είναι σημαντικό, επειδή όπως αποδείχθηκε στο προηγούμενο πείραμα, η Fine-Grained διαχείριση είναι συνδεδεμένη με το επιπλέον φορτίο επικοινωνίας και εκκίνησης. Ωστόσο, αν και υπάρχουν αυτές οι πρόσθετες απαιτήσεις, η βελτιστοποίηση που προσφέρει η Fine-Grained διαχείριση των tests αποδεικνύει την αξία της, αφού επιτυγχάνει καλύτερη απόδοση ακόμη και με αυτό το επιπλέον κόστος.



Διάγραμμα 3.1  
(Χρόνος εκτέλεσης των tests, ανάλογα με τον αριθμό των workers)

Όπως παρουσιάζεται στο διάγραμμα 3.1, μπορεί κανείς να παρατηρήσει μια σχεδόν γραμμική μείωση του χρόνου εκτέλεσης των tests, η οποία συμβαδίζει με την αύξηση του αριθμού των workers. Αυτό υποδεικνύει ότι όσο περισσότεροι εργάτες συμμετέχουν στην εκτέλεση των tests, τόσο πιο γρήγορα ολοκληρώνεται η διαδικασία. Ωστόσο, είναι σημαντικό να αναφερθεί ότι αυτή η γραμμική μείωση δεν είναι απόλυτη και μπορεί να υπάρξουν κάποιες μικρές αποκλίσεις, λόγω διαφόρων παραγόντων όπως η επικοινωνία μεταξύ των workers, η διαχείριση των πόρων και άλλων συνθηκών του συστήματος.

### 3.4 Συμπέρασμα

Αν λάβουμε υπόψιν ότι υπάρχει κάποιος χρόνος εκκίνησης  $c$  για κάθε εκτέλεση test suite, τότε μπορούμε να εκφράσουμε την σειριακή εκτέλεση χωρίς ουρά με την παρακάτω συνάρτηση.

$$Ts = s + a \cdot n + c$$

Όπου  $s$  είναι ο χρόνος αρχικοποίησης του περιβάλλοντος (εγκατάσταση dependencies και compile του κώδικα),  $a$  είναι ο μέσος όρος εκτέλεσης ενός μεμονωμένου test και  $n$  είναι ο αριθμός όλων των tests στη συγκεκριμένη συλλογή από test suites.

Αν επίσης λάβουμε υπόψιν ότι υπάρχει κάποιος επιπρόσθετος χρόνος  $p$  που αντιστοιχεί με τον συνολικό χρόνο επικοινωνίας με την βάση δεδομένων για κάθε εκτέλεση ενός test, τότε μπορούμε να εκφράσουμε μια εκτέλεση με ουρά με την παρακάτω συνάρτηση.

$$Tp = s + \lceil \frac{n}{w} \rceil \cdot (a + c + p)$$

Τότε ξέρουμε πως η χρήση του κατανεμημένου παράλληλου συστήματος θα μειώσει την ταχύτητα εκτέλεσης αν η παρακάτω συνθήκη ισχύει για  $a, c, p, n, w > 0$

$$Tp < Ts$$

$$(a + c + p) \cdot \lceil \frac{n}{w} \rceil < a \cdot n + c$$

Γνωρίζοντας λοιπόν τις τιμές των  $s$  και  $c$ , το κρίσιμο στοιχείο που πρέπει να εξεταστεί είναι το  $p$ , που αναφέρεται στο parallization overhead. Το parallization overhead, δηλαδή το επιπλέον κόστος που προκύπτει από την παραλληλοποίηση του προβλήματος, μπορεί να περιλαμβάνει διάφορους παράγοντες όπως η επικοινωνία μεταξύ των διαφόρων επεξεργαστών, τον συγχρονισμό των διαφορετικών εργασιών και άλλα σχετικά θέματα. Επομένως, για μια πιο ακριβή και αποτελεσματική αξιολόγηση της συνολικής απόδοσης, είναι σημαντικό να ληφθεί υπόψη και αυτό το στοιχείο.

## Κεφάλαιο 4

### Μελλοντικές επεκτάσεις

#### 4.1 Διανομή κοινών εργασιών

Αν και η ταχύτητα εκτέλεσης κάποιων εργασιών ενδέχεται να βελτιωθεί χρησιμοποιώντας αυτή την προσέγγιση, είναι σημαντικό να αναγνωρίσουμε ότι οι απαιτούμενοι υπολογιστικοί πόροι αυξάνονται παράλληλα. Όπως αναφέραμε στο κεφάλαιο 3, η εκτέλεση των tests με δύο workers διπλασίασε την ανάγκη για υπολογιστικούς πόρους. Παρότι αυτό μπορεί να μην φαίνεται σημαντικό στο πλαίσιο της παρούσας απόδειξης εργασιακής ικανότητας (proof of concept), σε ένα ευρύτερο σύστημα που διανέμει εργασίες συνεχώς, θα επιθυμούσαμε να διατηρούμε μέρος των υπολογιστικών πόρων που απασχολούνται από τους workers, διαθέσιμους για μεγαλύτερα χρονικά διαστήματα. Επομένως, θα μπορούσαμε να διαχειριστούμε πιο οικονομικά τον χρόνο χρήσης των πόρων από κάθε worker, διανέμοντας τις κοινές διεργασίες από όλους τους workers σε έναν μόνο.

Για παράδειγμα, στο επιτελούμενο πείραμα, η κοινή διεργασία μεταξύ όλων των workers ήταν η εκτέλεση του script setup.sh. Σε ένα πιο αποδοτικό οικονομικά σύστημα, αυτό το script θα εκτελείτο από έναν μόνο worker, ο οποίος θα αποθήκευε τα αποτελέσματα σε ένα Storage Bucket. Κατόπιν, οι υπόλοιποι workers θα κατέβαζαν και θα χρησιμοποιούσαν τα αποτελέσματα προτού αρχίσουν να προσλαμβάνουν διεργασίες από την ουρά διεργασιών. Με αυτόν τον τρόπο, θα είχαμε  $(w-1)$  workers διαθέσιμους για μια χρονική περίοδο  $s$ , όπου  $w$  είναι ο αριθμός των workers και  $s$  ο χρόνος εκτέλεσης του script setup.sh. Αυτή η στρατηγική αποτελεί μια συνετή προσέγγιση για την αποδοτική χρήση των υπολογιστικών πόρων, αποφεύγοντας περιττές επαναλήψεις και επιτυγχάνοντας ταυτόχρονα οικονομίες κλίμακας.

#### 4.2 Load Balancing Optimization

Βάσει των αποτελεσμάτων που παρουσιάζονται από τα πειραματικά δεδομένα, διαπιστώνουμε ότι ο χρόνος που απαιτείται για την επικοινωνία και εκκίνηση κάθε test δεν μπορεί να θεωρηθεί ως αμελητέος. Υπάρχει μια αξιοσημείωτη διαφορά τριών και μισού λεπτών μεταξύ της σειριακής εκτέλεσης και της σειριακής εκτέλεσης με Fine-Grained διαμέριση εργασίας. Όπως προηγουμένως αναφέρθηκε στην ενότητα 1.6.1, αυτό το φορτίο εκκίνησης και επικοινωνίας αποτελεί το αποτέλεσμα της Fine-Grained διαμέρισης εργασίας.

Παρ' όλα αυτά, αν είχαμε επιλέξει τον Coarse-Grained parallelism, θα αντιμετωπίζαμε πιθανώς το πρόβλημα της υψηλής πιθανότητας για χειρότερο Load Balancing.

Αυτό θα σήμαινε ότι ορισμένοι workers θα καταλάμβαναν περισσότερο χρόνο για την εκτέλεση tests συγκριτικά με άλλους, που πιθανώς θα μπορούσαν να παραμένουν αδρανείς. Αυτό το πρόβλημα προκύπτει από την αδυναμία μας να προβλέψουμε τον χρόνο που θα χρειαστεί για την εκτέλεση κάθε διεργασίας. Ωστόσο, αν είχαμε τη δυνατότητα να προβλέψουμε αυτόν τον χρόνο εκτέλεσης, θα μπορούσαμε να υλοποιήσουμε έναν συνδυασμό στατικών και δυναμικών αλγορίθμων Load Balancing. Αυτό θα μας επέτρεπε να εφαρμόσουμε τον Coarse-Grained parallelism

κατά την εκκίνηση της εργασίας, επιτυγχάνοντας με αυτόν τον τρόπο βελτιστοποίηση του συστήματος.

Για παράδειγμα, αν υποθέσουμε ότι έχουμε  $w = 2$ ,  $n = 4$  και  $T = [2, 2, 3, 3]$  όπου  $w$  αντιπροσωπεύει τον αριθμό των workers,  $n$  τον αριθμό των διεργασιών και  $T$  τους χρόνους εκτέλεσης κάθε διεργασίας, τότε ο στόχος μας είναι να επιτύχουμε ισορροπία στην εκτέλεση του συστήματος, αναθέτοντας τις διεργασίες έτσι ώστε το άθροισμα του χρόνου εκτέλεσης του κάθε worker να έχει όσο το δυνατόν λιγότερη απόκλιση από τον άλλο.

Αυτή η βελτιστοποίηση θα μπορούσε πιθανώς να επιτευχθεί μέσω της εκμετάλλευσης ιστορικών δεδομένων χρόνων εκτέλεσης από προηγούμενα logs καθώς και τον συνδυασμό τους με εφαρμογές αλγορίθμων Machine Learning. Μάλιστα, μια έρευνα που διεξήχθη το 2022 έδωσε ενθαρρυντικά αποτελέσματα σχετικά με τη βελτιστοποίηση του Load Balancing μέσω της χρήσης Machine Learning σε περιβάλλοντα cloud computing.<sup>[25]</sup>

### 4.3 Caching Optimization

Παρά την αναμφίβολη βελτίωση στην ταχύτητα που παρατηρήθηκε στο σύστημα συγκριτικά με το παρελθόν, ένα σημαντικό ποσοστό χρόνου εξακολουθεί να δαπανάται για την εγκατάσταση των dependencies από το δίκτυο. Οι εφαρμογές που λαμβάνονται διαμέσου αυτής της διαδικασίας είναι κυρίως Docker images ή Maven artifacts.

Για αυτόν τον λόγο, προτείνεται η αποθήκευση αυτών των πακέτων σε έναν εξωτερικό δίσκο, ο οποίος θα είναι προσβάσιμος από τους workers για mount. Αυτό θα επέτρεπε τη διατήρηση των πακέτων μας σε μόνιμη βάση, ακόμα και μετά την διαγραφή των workers.

Πέραν αυτού, υπάρχει η δυνατότητα χρήσης ενός τοπικού Artifact Registry, που θα λειτουργούσε ως κεντρικό σημείο διανομής των πακέτων. Ενδεικτικά, το Sonatype Nexus είναι ένα γνωστό λογισμικό που παρέχει αυτήν την λειτουργικότητα, επιτρέποντας την αποτελεσματική διαχείριση και διανομή των πακέτων.

### 4.4 Stream των logs κατά την εκτέλεση

Η τρέχουσα υλοποίηση του worker ανεβάζει τα αρχεία καταγραφής μόνο μετά την ολοκλήρωση όλων των διεργασιών που έχει εκτελέσει. Αυτό ενδεχομένως μπορεί να καθιστά το σύστημα λιγότερο φιλικό προς τον χρήστη, καθώς δεν επιτρέπει την εμφάνιση των αποτελεσμάτων των tests σε πραγματικό χρόνο, το οποίο μπορεί να επιβραδύνει τη διαδικασία επίλυσης ενός πιθανού προβλήματος σε περίπτωση αποτυχίας των tests. Για τους ανωτέρω λόγους, θα ήταν επιθυμητό οι workers να αποστέλλουν τα logs κατά τη διάρκεια της εκτέλεσης των διεργασιών. Είναι κρίσιμο αυτή η αποστολή να προσθέτει όσο το δυνατόν μικρότερη επιβάρυνση, ώστε να μην επιφέρει αρνητικές επιπτώσεις στην εκτέλεση των tests. Υπάρχουν τεχνολογίες, όπως το Fluentd<sup>[26]</sup>, που ειδικεύονται στην συλλογή των αρχείων καταγραφής και θα μπορούσαν ενδεχομένως να ενσωματωθούν στους workers.

# Γλωσσάρι

|    |                                            |                                      |
|----|--------------------------------------------|--------------------------------------|
| 1  | API<br>(Application Programming Interface) | Διασύνδεση Προγραμματισμού Εφαρμογών |
| 2  | Artifact Registry                          | Μητρώο Πακέτων                       |
| 3  | Cache                                      | Κρυφή μνήμη                          |
| 4  | Client                                     | Πρόγραμμα πελάτης                    |
| 5  | Cluster                                    | Συστάδα υπολογιστών                  |
| 6  | Coarse-Grained Parallelism                 | Παραλληλισμός με χοντρή επικοινωνία  |
| 7  | Commit                                     | Επιβεβαίωση                          |
| 8  | Container                                  | Κιβώτιο                              |
| 9  | Controller                                 | Ελεγκτής                             |
| 10 | Database                                   | Βάση δεδομένων                       |
| 11 | Deadlock                                   | Αδιέξοδος                            |
| 12 | Dependencies                               | Εξαρτήσεις                           |
| 13 | Distributed Computing                      | Κατανεμημένος υπολογισμός            |
| 14 | End-to-end tests                           | Αλληπαλά τεστ                        |
| 15 | Endpoint                                   | Ακραίο σημείο                        |
| 16 | Fine-Grained Parallelism                   | Παραλληλισμός με λεπτή επικοινωνία   |
| 17 | Flaky tests                                | Αναξιόπιστα τεστ                     |
| 18 | Granularity (επικοινωνία)                  | Επικοινωνία                          |
| 19 | Guest                                      | Επισκέπτης                           |
| 20 | Hardware                                   | Υλικό σύστημα                        |
| 21 | Host                                       | Εξυπηρετητής                         |
| 22 | Image                                      | Εικόνα                               |

|    |                                       |                             |
|----|---------------------------------------|-----------------------------|
| 23 | Integration tests                     | Τεστ ενσωμάτωσης            |
| 24 | Least Connections                     | Ελάχιστες Συνδέσεις         |
| 25 | Load Balancing (ισορροπία του φόρτου) | Ισορροπία του φόρτου        |
| 26 | Logs                                  | Αρχεία καταγραφής           |
| 27 | Machine Learning                      | Μηχανική Μάθηση             |
| 28 | Main                                  | Κύρια                       |
| 29 | Microservices                         | Μικροϋπηρεσίες              |
| 30 | Multi-stage build                     | Κατασκευή πολλαπλών σταδίων |
| 31 | Noisy Neighbor                        | Θορυβώδης γείτονας          |
| 32 | Operating System                      | Λειτουργικό σύστημα         |
| 33 | Parallel Computing                    | Παράλληλος υπολογισμός      |
| 34 | Pipeline                              | Διοχέτευση                  |
| 35 | Process                               | Διεργασία                   |
| 36 | Race condition                        | Συνθήκη Αγώνα               |
| 37 | Scaling                               | Κλιμάκωση                   |
| 38 | Server                                | Διακομιστής                 |
| 39 | Storage Bucket                        | Κάδος αποθήκευσης           |
| 40 | Stream                                | Ροή δεδομένων               |
| 41 | Task                                  | Έργο                        |
| 42 | Test suites                           | Ακολουθία από τεστς         |
| 43 | Timestamp                             | Χρονοσφραγίδα               |
| 44 | Traffic                               | Κίνηση                      |
| 45 | Transaction                           | Συναλλαγή                   |

|    |                                      |                                  |
|----|--------------------------------------|----------------------------------|
| 46 | Unit tests                           | Τεστ μονάδων                     |
| 47 | UUID (Universally Unique Identifier) | Παγκοσμίως Μοναδική ταυτότητα    |
| 48 | Virtualization                       | Εικονικοποίηση                   |
| 49 | Weighted Least Connections           | Σταθμισμένες Ελάχιστες Συνδέσεις |
| 50 | Weighted Response Time               | Σταθμισμένος Χρόνος Απόκρισης    |
| 51 | Worker                               | Εργάτης                          |

# Βιβλιογραφία

- [1] C. Ziftci and D. Cavalcanti. *De-Flake Your Tests : Automatically Locating Root Causes of Flaky Tests in Code At Google*. 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), Adelaide, SA, Australia, 2020, pp. 736-745, doi: 10.1109/ICSME46990.2020.00083.
- [2] O'Connor, Rory V.; Akkaya, Mariye Umay; Kemaneci, Kerem; Yilmaz, Murat; Poth, Alexander; Messnarz, Richard (2015-10-15). *Systems, Software and Services Process Improvement: 22nd European Conference*, EuroSPI 2015, Ankara, Turkey, September 30 -- October 2, 2015. Proceedings. Springer. ISBN 978-3-319-24647-5.
- [3] Redhat (2022). *What is CI/CD*, <https://www.redhat.com/en/topics/devops/what-is-ci-cd>
- [4] John Down, Chad Kittel, Paolo Slavatori, Daniel Scott-Raynsford, Arsen Vladimirovskiy. *Noisy Neighbor antipattern*, <https://learn.microsoft.com/en-us/azure/architecture/antipatterns/noisy-neighbor/noisy-neighbor>
- [5] *Virtualization in education* (PDF). IBM. October 2007.
- [6] Wikipedia. *Virtualization*, <https://en.wikipedia.org/wiki/Virtualization>
- [7] O'Gara, Maureen. *Ben Golub, Who Sold Gluster to Red Hat, Now Running dotCloud*. SYS-CON Media
- [8] Docker. *What is a container ?*, <https://www.docker.com/resources/what-container/>
- [9] Kubernetes (2023). *Why you need Kubernetes and what can it do*, <https://kubernetes.io/docs/concepts/overview/#why-you-need-kubernetes-and-what-can-it-do>
- [10] Kubernetes (2023). *Kubernetes API*, <https://kubernetes.io/docs/concepts/overview/kubernetes-api/>
- [11] Wikipedia. *Load Balancing (computing)*, [https://en.wikipedia.org/wiki/Load\\_balancing\\_\(computing\)](https://en.wikipedia.org/wiki/Load_balancing_(computing))
- [12] Stallings, William (2015). *Operating Systems: Internals and Design Principles*. Pearson. p. 409. ISBN 978-0-13-380591-8.
- [13] Kwiatkowski, Jan (9 September 2001). *Evaluation of Parallel Programs by Measurement of Its Granularity. Parallel Processing and Applied Mathematics*. Lecture Notes in Computer Science.

[14] Barney, Blaise. *Introduction to Parallel Computing*. Lawrence Livermore National Laboratory  
<https://hpc.llnl.gov/documentation/tutorials/introduction-parallel-computing-tutorial##DesignGranularity>

[15] Tanenbaum, Andrew S.; Steen, Maarten van (2002). *Distributed systems: principles and paradigms*. Upper Saddle River, NJ: Pearson Prentice Hall.

[16] Wikipedia. *Distributed computing*.  
[https://en.wikipedia.org/wiki/Distributed\\_computing](https://en.wikipedia.org/wiki/Distributed_computing)

[17] Padua, David (2011). *Encyclopedia of Parallel Computing*. Springer. p. 524. ISBN 9780387097657

[18] Docker. *Multi-stage builds*. <https://docs.docker.com/build/building/multi-stage/>

[19] Jérôme Petazzoni (2020). *Do not use Docker in Docker for CI*.  
<https://jpetazzo.github.io/2015/09/03/do-not-use-docker-in-docker-for-ci/>

[20] Michael Vizard. *The Rise of In-Memory Databases*. Slashdot.  
<https://web.archive.org/web/20121009191558/http://slashdot.org/topic/datacenter/the-rise-of-in-memory-databases/>

[21] Redis. *Transactions*.  
<https://redis.io/docs/manual/transactions/#:~:text=Redis%20Transactions%20allow%20the%20execution,are%20serialized%20and%20executed%20sequentially>

[22] Google Cloud. *General Purpose Machines*.  
[https://cloud.google.com/compute/docs/general-purpose-machines#n1\\_machines](https://cloud.google.com/compute/docs/general-purpose-machines#n1_machines)

[23] Wikipedia. *Keycloak*. <https://en.wikipedia.org/wiki/Keycloak>

[24] Apache. *What is Maven ?* <https://maven.apache.org/what-is-maven.html>

[25] Adil, M., Nabi, S., Aleem, M., Diaz, V. G., & Lin, J. C.-W. (2023). *CA-MLBS: content-aware machine learning based load balancing scheduler in the cloud environment*. Expert Systems, 40( 4), e13150. <https://doi.org/10.1111/exsy.13150>

[26] Wikipedia. *Fluentd*. <https://en.wikipedia.org/wiki/Fluentd>