Ελληνικό Μεσογειακό Πανεπιστήμιο

Πολυτεχνική Σχολή

Τμήμα Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών

Πρόγραμμα Σπουδών Μηχανικών Πληροφορικής Τ.Ε.

Πτυχιακή Εργασία

Ανάπτυξη και Διαχείριση Kubernetes Cluster για Κατανομή Φόρτου και Απομακρυσμένη Πρόσβαση σε Διαδικτυακές Υπηρεσίες

Φοιτητής: Βούλγαρης Αλέξανδρος

Αριθμός Μητρώου: ΤΠ3886

Επιβλέπων: Παναγιωτάκης Σπυρίδων

Αναπληρωτής Καθηγητής

ΗΡΑΚΛΕΙΟ 2025

Ευχαριστίες

Σε αυτό το σημείο θα ήθελα να εκφράσω τις ειλικρινείς μου ευχαριστίες σε όλους όσοι με στήριξαν καθ' όλη τη διάρκεια εκπόνησης της παρούσας πτυχιακής εργασίας. Η συμβολή τους, η καθοδήγηση και η ενθάρρυνσή τους υπήρξαν πραγματικά ανεκτίμητες.

Πρώτα απ' όλα, θα ήθελα να ευχαριστήσω θερμά τον καθηγητή μου, Δρ. Σπυρίδωνα Παναγιωτάκη, ο οποίος ανέλαβε και την επίβλεψη της εργασίας. Η συνεχής του υποστήριξη, οι εύστοχες παρατηρήσεις του και η προθυμία του να βοηθά σε κάθε στάδιο υπήρξαν καθοριστικοί παράγοντες για την ολοκλήρωση της προσπάθειας. Η καθοδήγησή του με βοήθησε να ξεπεράσω δυσκολίες και να παραμείνω προσηλωμένος στους στόχους μου.

Επίσης, θα ήθελα να ευχαριστήσω τους συμφοιτητές και φίλους μου, που στάθηκαν δίπλα μου, μοιράστηκαν τις γνώσεις τους και μου προσέφεραν στήριξη και κίνητρο σε απαιτητικές στιγμές. Η συνεργασία και η παρέα τους ήταν πολύτιμες σε όλη τη διάρκεια των σπουδών μου.

Τέλος, εκφράζω την εκτίμησή μου στο σύνολο του διδακτικού προσωπικού του Τμήματός μας για την ποιοτική εκπαίδευση και την έμπνευση που μας προσέφεραν. Η αφοσίωσή τους και η στήριξή τους συνέβαλαν ουσιαστικά στην επιστημονική και επαγγελματική μας εξέλιξη.

Περίληψη

Η παρούσα πτυχιακή εργασία παρουσιάζει την υλοποίηση μιας πλήρους τοπικής υποδομής Kubernetes με σκοπό την κατανομή φόρτου και την απομακρυσμένη πρόσβαση σε διαδικτυακές υπηρεσίες. Η υλοποίηση βασίστηκε σε έναν προσωπικό υπολογιστή με Windows 10 και αξιοποιεί εργαλεία όπως Docker Desktop, kubeadm, Prometheus, Grafana, Ingress Controller με HTTPS, MetalLB, Tailscale VPN και dnsmasq για τοπική επίλυση DNS. Το σύστημα περιλαμβάνει web server με 2 replicas, βάση δεδομένων MySQL σε διάταξη master-slave, και λειτουργίες autoscaling μέσω HPA. Η εργασία αποδεικνύει ότι ακόμη και σε οικιακό εξοπλισμό μπορούν να αναπτυχθούν υποδομές αντίστοιχες του cloud, διατηρώντας υψηλά επίπεδα ασφάλειας, ευελιξίας και αυτονομίας. Η υλοποίηση αξιολογείται μέσω stress tests και μετρικών απόδοσης, καταδεικνύοντας την καταλληλότητά της για περιβάλλοντα Industry 4.0 και εκπαιδευτική χρήση.

Abstract

This thesis presents the implementation of a complete on-premise Kubernetes infrastructure designed for load distribution and remote access to web services. The infrastructure was deployed on a personal Windows 10 computer using Docker Desktop and kubeadm, and integrates tools such as Prometheus, Grafana, HTTPS Ingress Controller, MetalLB, Tailscale VPN, and dnsmasq for local DNS resolution. The system includes a web server with two replicas, a master-slave MySQL database, and horizontal pod autoscaling (HPA). This work demonstrates that advanced cloud-like platforms can be successfully implemented on consumer-grade equipment, ensuring high levels of security, flexibility, and autonomy. The solution is evaluated through stress tests and performance metrics, showing its suitability for Industry 4.0 scenarios and educational environments.

Πίνακας περιεχομένων

Ευχαριστίες 2

Περίληψη 3

Abstract 3

Κεφάλαιο 1: Εισαγωγή 8

1.1 Κίνητρο 8

1.1.1 Πλεονεκτήματα των Τοπικών Υποδομών 8

1.1.2 Cloud-Native Τεχνολογίες σε Τοπικό Περιβάλλον 11

1.2 Αντικείμενο της Πτυχιακής Εργασίας και Σύγχρονη Σημασία του 14

1.3 Δομή της Πτυχιακής Εργασίας 16

Κεφάλαιο 2: Τεχνολογίες και Αρχιτεκτονικές της Τοπικής Υποδομής Kubernetes 18

2.1 Προοίμιο 18

2.2 Εικονικοποίηση vs. Containerization (Docker) 18

2.3 Ενορχήστρωση Containers με Kubernetes 20

2.3.1 Αρχιτεκτονική: Control Plane και Nodes 20

2.3.2 Υπηρεσίες, Load Balancing και Service Discovery στο Kubernetes 22

2.3.3 Ingress Controller και HTTPS με αυτοϋπογεγραμμένα πιστοποιητικά 25

2.4 Λογισμικο-κεντρικές Δικτυακές Τεχνολογίες: SDN, Network Slicing, Network Softwarization 26

2.5 Δικτύωση VPN με Tailscale 28

2.6 Παρατηρησιμότητα και Monitoring: Prometheus & Grafana 29

2.7 Αναπαραγωγή Δεδομένων στη MySQL (Master-Slave Replication) 31

2.8 Ενοποίηση της Τοπικής Υποδομής: Αυτοματοποίηση, Παρατηρησιμότητα, Ασφάλεια και Επεκτασιμότητα 33

Κεφάλαιο 3 – Σχεδιασμός Συστήματος 37

3.1 Ανάλυση Απαιτήσεων 37

3.2 Αρχιτεκτονική Συστήματος 39

3.2.1 Κόμβος Kubernetes και Υποδομή Cluster 40

3.2.2 Εφαρμογή Web & Αυτόματη Κλιμάκωση 41

3.2.3 Βάση Δεδομένων MySQL (Master/Replica) 42

3.2.4 Δρομολόγηση Εισόδου: Nginx Ingress Controller & MetalLB 44

3.2.5 Παρακολούθηση: Prometheus & Grafana 45

3.2.6 Δίκτυο VPN και Τοπικός DNS....47
3.3 Ροές Λειτουργίας του Συστήματος (Σενάρια)....49
3.3.1 Σενάριο Spike Φόρτου (Auto-Scaling Event)....49
3.3.2 Σενάριο Αστοχίας Pod (Self-Healing Event)....52
3.4 Εργαλεία και Τεχνολογίες που χρησιμοποιήθηκαν....56
Κεφάλαιο 4 – Υλοποίηση Συστήματος....61
4.1 Εγκατάσταση WSL και Docker Desktop....61
4.2 Εγκατάσταση και Ρύθμιση του MetalLB....61
4.3 Εγκατάσταση και Ρύθμιση του NGINX Ingress Controller....62
4.4 Ανάπτυξη Web Server με 2 Replicas....64
4.4.1 Web Deployment....64
Το αρχείο web-server-deployment.yaml (Εικόνα 4.4.1,4.4.2) περιλαμβάνει τη δήλωση του deployment που καθορίζει:....64
4.4.2 Web Service....65
4.5 Ανάπτυξη MySQL σε Master/Slave Αρχιτεκτονική με Helm....66
4.5.1 Προετοιμασία του Helm Chart....66
4.5.2 Εγκατάσταση του Chart με Ρυθμίσεις Replication....66
4.5.3 Επιβεβαίωση Λειτουργίας Replication....67
4.6 Εγκατάσταση Prometheus και Grafana με Helm....67
4.6.1 Προσθήκη των Helm Repositories....67
4.6.2 Εγκατάσταση του kube-prometheus-stack....68
4.6.3 Πρόσβαση στη Grafana....68
4.7 Υλοποίηση Τοπικού DNS με dnsproxy και CoreDNS (Εκτός Cluster)....70
4.7.1 dnsproxy: Lightweight DNS Forwarder....70
4.7.2 CoreDNS: Custom Resolver για .local....71
4.8 Ρύθμιση Tailscale VPN: Server στο Cluster – Exit Node στον Host....72
4.8.1 Tailscale Server εντός Cluster....72
4.8.2 Ρύθμιση Host ως Exit Node....74
4.8.3 Routes και DNS....75
4.9 Ορισμός Ingress Resources και Ρύθμιση Metrics για Monitoring....76
4.9.1 Δημιουργία Ingress για Web Application....76
4.9.2 Ingress για Grafana και Prometheus UI....77

4.9.3 Ενεργοποίηση Metrics για Nginx Ingress Controller 78
4.9.4 Ενεργοποίηση MySQL Exporter για Prometheus 79
4.10 Προσαρμογή Web Container και Metrics μέσω Nginx Stub Status 79
4.10.1 Δομή του Dockerfile 79
4.10.2 Entry Point Script 80
4.10.3 Nginx Stub Status 80
4.10.4 Περιεχόμενο Εφαρμογής 81
4.10.5 Δημιουργία του Docker Image και Αποθήκευση στο Τοπικό Docker Registry 82
4.10.6 Παρακολούθηση του Web Server μέσω Prometheus Exporter για Nginx 83
4.10.7 Ορισμός Πολιτικών Πόρων με LimitRange 87
4.11 Ορισμός ενός Horizontal Pod Autoscaler (HPA) 88
4.11.1 Horizontal Pod Autoscaler (HPA) 88
4.11.2 Stress Testing με Pods 89
4.11.3 Stress Testing Δικτύου με iperf3 89
4.11.4 Παρακολούθηση κατά τη διάρκεια Stress Test 90
4.11.5 Περιορισμοί Ρυθμού (Rate Limits) σε Ingress 91
Κεφάλαιο 5: Τρόπος Λειτουργίας και Αξιολόγηση Λειτουργίας Συστήματος 92
5.1 Διεπαφές Χρήστη και Παρεχόμενες Δυνατότητες 92
5.2 Αξιολόγηση Επιδόσεων και Πειράματα 92
5.2.1 Πείραμα: Κατανάλωση CPU & Scaling μέσω HPA 92
5.2.2 Πείραμα: Δικτυακή Απόδοση μέσω iperf3 94
5.2.3 Πείραμα: Rate Limiting & Απόρριψη Συνδέσεων 99
5.2.4 Παρακολούθηση Metrics και Visualization 100
5.3 Συμπεράσματα Επιδόσεων 101
Κεφάλαιο 6: Συμπεράσματα και Μελλοντικές Επεκτάσεις 102
6.1 Συμπεράσματα 102
6.2 Τι Δούλεψε Καλά 102
6.3 Τι Δεν Δούλεψε Ιδανικά / Δυσκολίες 102
6.4 Μελλοντικές Επεκτάσεις 103
Αναφορές 104
Γενική Θεωρία και Τεχνολογικό Υπόβαθρο 104

Τεχνολογίες Kubernetes και Υλοποίησης 104
Εργαλεία VPN, DNS και Monitoring 104
Προχωρημένες Τεχνικές και Μελλοντικές Επεκτάσεις 105
Παραρτήματα 106
Παράρτημα 5.1 106
Παράρτημα 5.2 107

Κεφάλαιο 1: Εισαγωγή

1.1 Κίνητρο

Η εκτεταμένη διάδοση του cloud computing την τελευταία δεκαετία έχει φέρει επανάσταση στον τρόπο με τον οποίο αναπτύσσονται και φιλοξενούνται οι εφαρμογές. Οι δημόσιες cloud πλατφόρμες προσφέρουν κλιμακούμενους πόρους και υπηρεσίες κατά παραγγελία, απαλλάσσοντας τους οργανισμούς από τη διαχείριση φυσικού εξοπλισμού. Ωστόσο, παρά τα οφέλη αυτά, εξακολουθεί να υπάρχει έντονο ενδιαφέρον για τοπικές (on-premises) υποδομές, όπου το σύνολο των υπηρεσιών και δεδομένων φιλοξενείται σε ιδιόκτητους servers εντός του οργανισμού. Οι λόγοι ποικίλλουν: από την ανάγκη πλήρους ελέγχου και προστασίας των δεδομένων, μέχρι την απαίτηση λειτουργίας ανεξάρτητα από το διαδίκτυο σε απομακρυσμένα ή ευαίσθητα περιβάλλοντα. Στο πλαίσιο αυτό, η παρούσα εργασία εστιάζει στην υλοποίηση μιας τέτοιας τοπικής υποδομής, αξιοποιώντας σύγχρονες τεχνολογίες που προέρχονται από τον κόσμο του cloud.

Με την ωρίμανση τεχνολογιών **containerization** και **orchestration**, έχει καταστεί εφικτή η δημιουργία ισχυρών on-premises συστημάτων με χαρακτηριστικά παρόμοια των cloud υποδομών. Ειδικότερα, η τεχνολογία **Kubernetes** – ένα ανοιχτού κώδικα σύστημα ορχήστρωσης κοντέινερ – επιτρέπει την αυτοματοποίηση της ανάπτυξης, κλιμάκωσης και διαχείρισης εφαρμογών σε κοντέινερ, ανεξαρτήτως υποκείμενης πλατφόρμας. Είναι αξιοσημείωτο ότι οι λεγόμενες *cloud-native* τεχνολογίες δεν περιορίζονται σε δημόσια data centers: σύμφωνα με το Cloud Native Computing Foundation (CNCF), οι τεχνικές αυτές στοχεύουν στην εκτέλεση κλιμακούμενων εφαρμογών σε σύγχρονα δυναμικά περιβάλλοντα όπως **δημόσια, ιδιωτικά ή υβριδικά cloud**. Με άλλα λόγια, τα ίδια εργαλεία που χρησιμοποιούνται σε έναν πάροχο cloud μπορούν να αξιοποιηθούν αυτούσια και σε μια τοπική εγκατάσταση ενός οργανισμού.

1.1.1 Πλεονεκτήματα των Τοπικών Υποδομών

Παρότι το δημόσιο cloud προσφέρει ευκολία στην κλιμάκωση και στη διαχείριση πόρων, τα τοπικά συστήματα εξακολουθούν να διαδραματίζουν κρίσιμο ρόλο σε πολλούς οργανισμούς λόγω των μοναδικών πλεονεκτημάτων που παρέχουν. Ακολουθούν συνοπτικά

μερικά από τα βασικά οφέλη μιας on-premises προσέγγισης, τα οποία αναδεικνύονται και από τη βιβλιογραφία:

- **Πλήρης Έλεγχος Δεδομένων και Υποδομής:** Σε μια τοπική εγκατάσταση, ο οργανισμός έχει αποκλειστική κυριότητα και έλεγχο τόσο του υλικού (servers, δικτύωση) όσο και του λογισμικού. Αυτό σημαίνει ότι μπορεί να διαχειρίζεται διαμορφώσεις, ενημερώσεις και αποθήκευση δεδομένων σύμφωνα με τις δικές του πολιτικές, εξασφαλίζοντας μεγαλύτερη ευελιξία και συμμόρφωση με εσωτερικούς κανόνες. Η δυνατότητα προσαρμογής του περιβάλλοντος στα μέτρα του οργανισμού είναι πολύ μεγαλύτερη σε σχέση με λύσεις σε public cloud, όπου συχνά υπάρχουν περιορισμοί από τον πάροχο.
- **Ενισχυμένη Ασφάλεια και Συμμόρφωση:** Τα δεδομένα παραμένουν **τοπικά αποθηκευμένα**, μέσα στα όρια του οργανισμού, γεγονός που μειώνει δραστικά τον κίνδυνο διαρροών ή μη εξουσιοδοτημένης πρόσβασης από τρίτους. Ο οργανισμός μπορεί να εφαρμόσει προσαρμοσμένα μέτρα ασφαλείας (firewalls, κρυπτογράφηση, πολιτικές πρόσβασης) κατάλληλα για τις δικές του ανάγκες, ελαχιστοποιώντας τις πιθανότητες παραβίασης. Επιπλέον, σε ισχυρά ρυθμιζόμενους τομείς (π.χ. υγεία, χρηματοοικονομικά) μια τοπική υποδομή διευκολύνει τη συμμόρφωση με κανονισμούς περί απορρήτου, καθώς όλα τα ευαίσθητα δεδομένα παραμένουν *in-house* και υπό τον άμεσο έλεγχο του οργανισμού.
- **Αυτονομία και Ανεξαρτησία από Παρόχους Cloud:** Ένα τοπικό σύστημα μειώνει στο ελάχιστο την εξάρτηση από τρίτους παρόχους. Η λειτουργία κρίσιμων εφαρμογών δεν επηρεάζεται από προβλήματα ή αλλαγές σε υπηρεσίες κάποιου cloud vendor, ούτε από πιθανές αυξήσεις κόστους χρήσης. Ο οργανισμός διατηρεί **πλήρη αυτονομία** ως προς τις αποφάσεις αναβάθμισης ή επέκτασης της υποδομής του, χωρίς τον κίνδυνο *vendor lock-in*. Επιπλέον, μια τέτοια υποδομή δεν βασίζεται καθόλου σε συνεχή σύνδεση στο διαδίκτυο για να λειτουργήσει: η απουσία ανάγκης για εξωτερικούς

servers ή online υπηρεσίες σημαίνει ότι το σύστημα μπορεί να συνεχίσει να λειτουργεί ακόμη και αν η σύνδεση Internet είναι ασταθής ή ανύπαρκτη. Αυτή η ανεξαρτησία εξασφαλίζει αδιάλειπτη πρόσβαση σε κρίσιμες εφαρμογές και δεδομένα, κάτι ζωτικής σημασίας σε απομακρυσμένες εγκαταστάσεις ή σε περιβάλλοντα όπου η αξιοπιστία του δικτύου δεν είναι εγγυημένη.

- **Λειτουργία Offline:** Συναφές με την αυτονομία είναι και το πλεονέκτημα της offline λειτουργίας. Σε αντίθεση με καθαρά cloud-based λύσεις που απαιτούν συνδεσιμότητα, ένα τοπικό σύστημα μπορεί να λειτουργεί σε πλήρη απομόνωση από το διαδίκτυο. Αυτό το χαρακτηριστικό το καθιστά ιδανικό για περιπτώσεις όπου απαιτείται *εγγυημένη διαθεσιμότητα* ανεξαρτήτως εξωτερικών παραγόντων – για παράδειγμα, σε βιομηχανικές εγκαταστάσεις παραγωγής όπου τυχόν διακοπή της επικοινωνίας δεν πρέπει να επηρεάζει τον έλεγχο των μηχανημάτων. Όπως αναφέρθηκε, η απουσία εξάρτησης από εξωτερικούς servers διασφαλίζει ότι οι τοπικές εφαρμογές παραμένουν διαθέσιμες ακόμη και χωρίς πρόσβαση.
- **Καταλληλότητα για Εκπαίδευση και Έρευνα:** Οι τοπικές υποδομές είναι ιδανικές για ακαδημαϊκά ή εκπαιδευτικά περιβάλλοντα, όπου μπορεί να μην είναι εφικτή ή επιθυμητή η χρήση εμπορικών cloud υπηρεσιών. Ένα on-premises cluster μπορεί να στηθεί σε εργαστηριακό χώρο ή ακόμα και σε προσωπικούς υπολογιστές, δίνοντας τη δυνατότητα σε φοιτητές και ερευνητές να αποκτήσουν πρακτική εμπειρία πάνω σε σύγχρονες τεχνολογίες χωρίς κόστος χρήσης cloud ή περιορισμούς πρόσβασης στο διαδίκτυο. Παράλληλα, όλα τα δεδομένα και τα αποτελέσματα των πειραμάτων παραμένουν εσωτερικά, κάτι που μπορεί να είναι σημαντικό για ερευνητικά έργα με ευαίσθητες πληροφορίες. Η εμπειρία hands-on σε ένα πραγματικό (έστω και μικρής κλίμακας) cluster βοηθά στην καλύτερη κατανόηση έννοιων όπως τα containers, τα pods, η ανάλυση μετρικών κ.ο.κ., σε ένα ασφαλές περιβάλλον όπου τυχόν λάθη δεν θα επηρεάσουν κάποιον παραγωγικό υπολογιστικό πόρο.

- **Ευελιξία για Πειραματισμό:** Ένα τοπικό σύστημα προσφέρει τον *χώρο* για δοκιμές και πειραματισμούς που δύσκολα θα επιχειρούνταν σε ένα παραγωγικό περιβάλλον. Οι διαχειριστές μπορούν να τροποποιήσουν ρυθμίσεις χαμηλού επιπέδου, να δοκιμάσουν νέα εργαλεία ή να προσομοιώσουν σενάρια βλάβης, χωρίς να ανησυχούν για διατάραξη κάποιας υπηρεσίας πελάτη. Στην κοινότητα των **homelabs** (προσωπικών εργαστηριακών υποδομών), αναφέρεται συχνά ότι ο βασικός σκοπός είναι το *"tinkering and learning"*, δηλαδή το να μπορεί κανείς να πειραματιστεί ελεύθερα και να μάθει από την εμπειρία. Αυτό ακριβώς το πλεονέκτημα εκμεταλλευόμαστε και στην παρούσα υλοποίηση: η δυνατότητα να ρυθμίσουμε εξ αρχής όλες τις λεπτομέρειες του cluster (π.χ. πιστοποιητικά TLS, κανόνες δικτύωσης, πολιτικές autoscaling) μας επιτρέπει να αναδείξουμε στην πράξη τη λειτουργία τους, αποκτώντας πολύτιμη γνώση. Επιπλέον, ο πειραματισμός σε τοπικό περιβάλλον έχει μηδενικό πρόσθετο κόστος χρήσης πόρων (πέραν του υπάρχοντος εξοπλισμού) και μηδενικό ρίσκο έκθεσης συστημάτων στο ευρύ Internet, γεγονός που διευκολύνει την καινοτομία.

Συνοψίζοντας, μια τοπική (on-premise) αρχιτεκτονική μπορεί να προσφέρει **έλεγχο**, **ασφάλεια**, **αυτονομία** και **ευελιξία** που σε πολλές περιπτώσεις υπερβαίνουν τα αντίστοιχα επίπεδα μιας δημόσιας cloud λύσης. Φυσικά, η κάθε προσέγγιση έχει και συμβιβασμούς – η τοπική υποδομή προϋποθέτει την ύπαρξη και διαχείριση του απαραίτητου υλικού, καθώς και την τεχνογνωσία για τη συντήρησή της. Ωστόσο, για οργανισμούς με τις προτεραιότητες που αναφέρθηκαν, τα οφέλη αυτά κάνουν την on-premises λύση εξαιρετικά ελκυστική.

1.1.2 Cloud-Native Τεχνολογίες σε Τοπικό Περιβάλλον

Οι τεχνολογίες **cloud-native** – όπως η αρχιτεκτονική microservices, οι κοντέινερ και η αυτοματοποίηση μέσω Kubernetes – συχνά συνδέονται στο μυαλό πολλών με μεγάλες δημόσιες υποδομές cloud. Στην πραγματικότητα, όμως, αυτές οι τεχνικές είναι *ουδέτερες ως προς το υπόβαθρο υλοποίησης* και μπορούν να εφαρμοστούν εξίσου σε ιδιόκτητες υποδομές. Όπως προαναφέρθηκε, οι cloud-native μέθοδοι έχουν σχεδιαστεί ώστε να λειτουργούν σε ποικιλία περιβαλλόντων (δημόσιων *και* ιδιωτικών). Έτσι, ένας οργανισμός μπορεί να χτίσει το δικό του “μικρό cloud” εσωτερικά, αξιοποιώντας ακριβώς τα ίδια λογισμικά που τρέχουν και σε κέντρα δεδομένων κολοσσών.

Το **Kubernetes** αποτελεί χαρακτηριστικό παράδειγμα: πρόκειται για την *de facto* πλατφόρμα ενορχήστρωσης κοντέινερ, η οποία υιοθετήθηκε ευρέως από πάροχους cloud (Google, Amazon, Microsoft κ.ά.), αλλά ταυτόχρονα διατίθεται ως ανοικτό λογισμικό που μπορεί να εγκατασταθεί σε οποιοδήποτε περιβάλλον (σε bare-metal servers, σε virtual machines, ακόμη και σε απλές συσκευές edge). Έχει αποδειχθεί στην πράξη ότι το Kubernetes είναι εξίσου αποτελεσματικό τόσο σε μεγάλα cloud data centers, όσο και σε **on-premises** εγκαταστάσεις μέσα σε οργανισμούς. Μολονότι συχνά αποκαλείται μεταφορικά «το λειτουργικό σύστημα του cloud», το Kubernetes προσαρμόζεται το ίδιο καλά σε σενάρια *edge computing* – δηλαδή σε κατανεμημένη υπολογιστική ισχύ πλησίον του σημείου όπου παράγονται τα δεδομένα.

Αυτή η δυνατότητα είναι ιδιαίτερα κρίσιμη για σύγχρονες εφαρμογές. Για παράδειγμα, στο πλαίσιο του **Industry 4.0** (Βιομηχανία 4.0), όπου εργοστασιακές μονάδες είναι γεμάτες από αισθητήρες IoT και μηχανές που παράγουν μεγάλο όγκο δεδομένων σε πραγματικό χρόνο, έχει σημασία η **τοπική επεξεργασία** αντί να σταλούν όλα τα δεδομένα σε ένα απομακρυσμένο cloud. Το λεγόμενο **edge computing** αναφέρεται ακριβώς σε αυτή την πρακτική: τη μεταφορά μέρους των υπολογισμών και της αποθήκευσης στην περιφέρεια του δικτύου, κοντά στις πηγές δεδομένων. Με αυτόν τον τρόπο μειώνεται δραστικά η καθυστέρηση (latency) και βελτιώνεται η ασφάλεια/ιδιωτικότητα, αφού τα δεδομένα δεν εγκαταλείπουν τον χώρο παραγωγής. Σύγχρονες τεχνολογίες λογισμικού είναι κεντρικές σε αυτή την εξέλιξη – χαρακτηριστικά, σε ειδική έκθεση για το Industrial Edge επισημαίνεται ότι η κοντεϊνοποίηση με Docker και η ορχήστρωση με Kubernetes αποτελούν θεμέλια λογισμικά που οδηγούν αυτή την τάση.

Παρόμοια εικόνα συναντάμε και στις **Έξυπνες Πόλεις (Smart Cities)**, όπου οι υποδομές της πόλης (φωτισμός, συγκοινωνίες, περιβαλλοντικοί αισθητήρες κ.ά.) παράγουν συνεχώς δεδομένα. Για να ληφθούν αποφάσεις σε πραγματικό χρόνο – για παράδειγμα, στη διαχείριση της κυκλοφορίας ή στην απόκριση σε έκτακτα περιστατικά – απαιτείται η επεξεργασία να γίνει όσο το δυνατόν πιο κοντά στο σημείο συλλογής των δεδομένων. Οι πόλεις μπορεί να διαθέτουν τοπικούς servers ή μικρά data centers σε καίρια σημεία, όπου αναπτύσσονται οι ίδιες cloud-native εφαρμογές που θα μπορούσαν να τρέχουν και στο

cloud, μόνο που εδώ λειτουργούν **αυτόνομα τοπικά** για λόγους αξιοπιστίας και ταχύτητας. Ακόμη και στα σύγχρονα δίκτυα **5G/6G**, η αρχιτεκτονική προβλέπει αποκεντρωμένους κόμβους υπολογισμού (π.χ. στους σταθμούς βάσης κινητής τηλεφωνίας) γνωστούς και ως MEC (Multi-access Edge Computing). Σε αυτούς τους κόμβους, κομμάτια του πυρήνα του δικτύου ή εφαρμογές υψηλού bandwidth τρέχουν σε κοντέινερ και orchestrators όπως το Kubernetes, επιτυγχάνοντας πολύ χαμηλή καθυστέρηση και μειώνοντας την κίνηση προς κεντρικούς servers. Όλες αυτές οι εξελίξεις τονίζουν ότι οι **cloud τεχνολογίες μπορούν να εφαρμοστούν και τοπικά** με μεγάλη επιτυχία – από το εργοστάσιο και την πανεπιστημιακή αίθουσα, μέχρι την έξυπνη πόλη και το δίκτυο επόμενης γενιάς.

Στο πλαίσιο αυτό, η παρούσα πτυχιακή εργασία περιγράφει την εγκατάσταση μιας **τοπικής υποδομής βασισμένης σε Kubernetes**. Η υλοποίηση πραγματοποιήθηκε σε έναν τυπικό προσωπικό υπολογιστή με Windows 10, χρησιμοποιώντας το Docker Desktop (με WSL2) και το εργαλείο *kubeadm* για την δημιουργία ενός Kubernetes cluster. Η επιλογή αυτού του περιβάλλοντος καταδεικνύει ότι ακόμη και σε ένα απλό PC μπορούν να εγκατασταθούν προηγμένες πλατφόρμες κοντέινερ. Πάνω σε αυτό το τοπικό cluster ενσωματώθηκαν πολλαπλά εργαλεία που συνήθως συναντώνται σε **cloud περιβάλλοντα**: η πλατφόρμα κοντέινερ **Docker** για την πακετοποίηση εφαρμογών, ένας σχεσιακός βάση δεδομένων **MySQL** με ρυθμίσεις *replication* (πρωτεύον/δευτερεύον) για εξασφάλιση ανοχής σε βλάβες και συνέχειας δεδομένων, ένας μηχανισμός παρακολούθησης **Prometheus** σε συνδυασμό με την πλατφόρμα οπτικοποίησης **Grafana** για τη συλλογή και παρουσίαση μετρικών του συστήματος, καθώς και ένας ***Ingress controller*** (π.χ. Nginx) με υποστήριξη **HTTPS** για την ασφαλή πρόσβαση των υπηρεσιών μέσω φιλικών URL. Επιπλέον, επειδή σε ένα τοπικό cluster δεν υπάρχει ενσωματωμένος load balancer από πάροχο, χρησιμοποιήθηκε το **MetalLB** ώστε να παρέχει λειτουργίες εξισορρόπησης φορτίου (παραχωρώντας εικονικές IP στο εσωτερικό δίκτυο για υπηρεσίες τύπου LoadBalancer). Για την ασφαλή απομακρυσμένη πρόσβαση στο cluster από το διαδίκτυο υιοθετήθηκε μια λύση **VPN** βασισμένη στο **Tailscale**, ενώ ρυθμίστηκε και τοπική επίλυση **DNS** ώστε οι υπηρεσίες να είναι προσβάσιμες με σταθερά ονόματα εντός του δικτύου. Συνολικά, η υποδομή αυτή συγκροτεί ένα πλήρες

οικοσύστημα μικροϋπηρεσιών και εργαλείων διαχείρισης, επιδεικνύοντας ότι **cloud-like λειτουργικότητα** μπορεί να επιτευχθεί πλήρως και εκτός cloud.

Η υλοποίηση που παρουσιάζεται σε αυτή την εργασία, παρότι σε μικρότερη κλίμακα, ευθυγραμμίζεται με τα παραπάνω: δείχνει ότι ένα **πλήρες cloud-native οικοσύστημα** μπορεί να στηθεί και να λειτουργήσει **εντός των ορίων ενός υπολογιστή ή ενός ιδιωτικού δικτύου**. Χρησιμοποιώντας ανοικτά εργαλεία όπως το Kubernetes, το Docker, το Prometheus, κ.ά., αποκτούμε μια πλατφόρμα στην οποία μπορούν να τρέξουν σύγχρονες εφαρμογές microservices χωρίς την ανάγκη κάποιου παρόχου cloud. Αυτό όχι μόνο προσφέρει πρακτικά οφέλη (έλεγχο, ασφάλεια, offline λειτουργία όπως συζητήθηκε), αλλά παράλληλα λειτουργεί και ως απόδειξη ότι η γνώση και οι πρακτικές του cloud μπορούν να αξιοποιηθούν οπουδήποτε. Με άλλα λόγια, ο διαχωρισμός “cloud έναντι on-premises” γίνεται όλο και πιο θολός: οι ίδιες αρχές και τεχνολογίες εφαρμόζονται και στις δύο περιπτώσεις, με τη διαφορά ότι στην τοπική υλοποίηση ο οργανισμός διαχειρίζεται τα πάντα μόνος του.

Εν κατακλείδι, η παρούσα πτυχιακή εργασία στοχεύει να αναδείξει τόσο την πρακτική πλευρά της υλοποίησης μιας τοπικής υποδομής Kubernetes με όλα τα απαραίτητα συμπληρωματικά εργαλεία, όσο και τη σημασία αυτής της προσέγγισης στο σημερινό τεχνολογικό τοπίο. Μέσω της υλοποίησης αυτής, γίνεται φανερό ότι αρχές όπως η αυτοματοποίηση, η παρακολούθηση συστημάτων, η ανθεκτικότητα υπηρεσιών (π.χ. μέσω replication) και η ασφαλής πρόσβαση, μπορούν να επιτευχθούν και εκτός των μεγάλων εμπορικών cloud, προσφέροντας στους οργανισμούς εναλλακτικές λύσεις που προάγουν τον έλεγχο και την καινοτομία.

1.2 Αντικείμενο της Πτυχιακής Εργασίας και Σύγχρονη Σημασία του

Στη σύγχρονη εποχή της Τέταρτης Βιομηχανικής Επανάστασης (Industry 4.0), των Έξυπνων Πόλεων (Smart Cities) και της εξάπλωσης των δικτύων πέμπτης και έκτης γενιάς (5G/6G), η ανάγκη για ευέλικτες, αυτοματοποιημένες και παρατηρήσιμες υποδομές πληροφορικής είναι εντονότερη από ποτέ. Οι τεχνολογίες **virtualization, containerization** και αυτοματοποιημένης διαχείρισης εφαρμογών έχουν μεταμορφώσει τον τρόπο με τον οποίο σχεδιάζονται και διαχειρίζονται σύγχρονα συστήματα. Παράλληλα, η ανάγκη για υψηλή

διαθεσιμότητα, επεκτασιμότητα, ασφάλεια και ταχύτητα ανάπτυξης υπηρεσιών έχει επιβάλει νέες προσεγγίσεις αρχιτεκτονικής και διαχείρισης υποδομών.

Αν και οι περισσότερες συζητήσεις γύρω από αυτές τις τεχνολογίες αφορούν περιβάλλοντα **cloud**, είναι εξίσου σημαντικό να υπογραμμιστεί ότι οι ίδιες τεχνολογίες μπορούν να εφαρμοστούν αποτελεσματικά και σε τοπικές (on-premise) υποδομές. Η παρούσα πτυχιακή εργασία επικεντρώνεται στην ανάπτυξη μιας πλήρους και λειτουργικής τοπικής υποδομής, η οποία υλοποιήθηκε εξ ολοκλήρου σε περιβάλλον **Docker Desktop** με χρήση του **kubeadm** για τη δημιουργία ενός **Kubernetes cluster**. Η υποδομή φιλοξενείται σε έναν προσωπικό υπολογιστή με λειτουργικό σύστημα Windows 10, ενισχύοντας την ιδέα ότι σύγχρονες τεχνολογίες μπορούν να υλοποιηθούν και να αξιοποιηθούν σε μικρή κλίμακα, χωρίς εξάρτηση από υπηρεσίες cloud.

Η υποδομή που αναπτύχθηκε περιλαμβάνει:

- Έναν **web server** με 2 replicas, ο οποίος φιλοξενεί μια απλή εφαρμογή τύπου chat και αποθηκεύει τα μηνύματα σε βάση δεδομένων.
- Μια εγκατάσταση **MySQL με 2 replicas (master/slave)** για αποθήκευση των δεδομένων με υποστήριξη αναπαραγωγής.
- Παρακολούθηση της κατάστασης των pods και του συστήματος μέσω **Prometheus** και **Grafana**, με έτοιμα dashboards.
- Χρήση του **MetalLB** για την απόδοση στατικών IP διευθύνσεων εντός του τοπικού δικτύου, καθώς το περιβάλλον δεν υποστηρίζει ενσωματωμένο **load balancer**.
- Χρήση **Ingress Controller (Nginx)** με υποστήριξη **HTTPS** μέσω self-signed πιστοποιητικών.
- Διασύνδεση μέσω **Tailscale VPN** για απομακρυσμένη πρόσβαση στο τοπικό δίκτυο, παρακάμπτοντας περιορισμούς cgNAT.
- Τοπικό DNS resolving με dnsproxy και **coreDNS**, ώστε οι σελίδες να είναι προσβάσιμες από σταθερά domain names όπως **web.local, grafana.local κ.ά.**
- Stress tests της υποδομής, τόσο σε επίπεδο δικτύου (**με iperf3 και hey**), όσο και σε επίπεδο φόρτου pods με χρήση **Horizontal Pod Autoscaler** (HPA).

Η εργασία αυτή στοχεύει στο να καταδείξει ότι είναι εφικτό να αναπτυχθεί και να αξιολογηθεί μια ολοκληρωμένη υποδομή πληροφορικής, κατάλληλη για σενάρια Industry 4.0 ή εκπαιδευτικές/ερευνητικές εφαρμογές, χωρίς την ανάγκη χρήσης δημόσιων υπηρεσιών cloud. Παράλληλα, παρουσιάζεται η αξιοποίηση εργαλείων και τεχνολογιών που συναντάμε σε enterprise περιβάλλοντα, αλλά ενσωματωμένα και διαχειριζόμενα πλήρως τοπικά, προσφέροντας πλήρη έλεγχο, ασφάλεια, ευελιξία και ανεξαρτησία από τρίτους παρόχους.

1.3 Δομή της Πτυχιακής Εργασίας

Η πτυχιακή εργασία δομείται σε έξι κύρια κεφάλαια:

- **Κεφάλαιο 2**: Θεωρητικό Υπόβαθρο. Παρουσιάζονται οι βασικές τεχνολογίες και πρωτόκολλα που σχετίζονται με την εργασία. Γίνεται εισαγωγή στις έννοιες του **virtualization**, των **containers**, του **Kubernetes**, των αρχών δικτύωσης και ασφαλείας, και της παρακολούθησης συστημάτων.
- **Κεφάλαιο 3**: Σχεδιασμός Συστήματος. Αναλύονται οι απαιτήσεις του συστήματος, η αρχιτεκτονική του σχεδίαση, και τα επιμέρους υποσυστήματα. Περιλαμβάνονται διαγράμματα ροής και συνολική αποτύπωση του τεχνικού σχεδιασμού.
- **Κεφάλαιο 4**: Υλοποίηση Συστήματος. Παρουσιάζεται η διαδικασία εγκατάστασης και παραμετροποίησης της υποδομής, συμπεριλαμβανομένων των YAML αρχείων, των ρυθμίσεων του cluster και των εργαλείων που χρησιμοποιήθηκαν.
- **Κεφάλαιο 5**: Τρόπος Λειτουργίας. και Αξιολόγηση Παρουσιάζονται οι διεπαφές του συστήματος, στιγμιότυπα οθόνης (screenshots), και τα αποτελέσματα των stress tests και των μετρήσεων απόδοσης.
- **Κεφάλαιο 6**: Συμπεράσματα και Μελλοντικές Επεκτάσεις.

  Συζητούνται τα κύρια ευρήματα, οι δυσκολίες που αντιμετωπίστηκαν, και οι δυνατότητες επέκτασης ή βελτιστοποίησης της υποδομής.

- **Βιβλιογραφία**: Καταγράφονται όλες οι πηγές πληροφόρησης που χρησιμοποιήθηκαν.

Η εργασία φιλοδοξεί να προσφέρει ένα πλήρες παράδειγμα ανάπτυξης, διαχείρισης και παρακολούθησης τοπικής υποδομής πληροφορικής, αποδεικνύοντας ότι οι τεχνολογίες που

κυριαρχούν σε περιβάλλοντα **cloud** είναι εξίσου ισχυρές και αξιοποιήσιμες σε **on-premise** εγκαταστάσεις.

Κεφάλαιο 2: Τεχνολογίες και Αρχιτεκτονικές της Τοπικής Υποδομής Kubernetes

2.1 Προοίμιο

Σε αυτό το κεφάλαιο παρουσιάζονται οι κύριες τεχνολογίες και αρχιτεκτονικές που χρησιμοποιήθηκαν για την υλοποίηση μιας τοπικής υποδομής Kubernetes (on-premise). Εξηγούνται έννοιες όπως η **εικονικοποίηση** και η **containerization**, η ενορχήστρωση κοντέινερ με το Kubernetes (συμπεριλαμβανομένης της λειτουργίας του control plane και των worker nodes), τα συστήματα παρακολούθησης (monitoring) με Prometheus/Grafana, καθώς και σύγχρονες προσεγγίσεις δικτύωσης όπως τα δίκτυα οριζόμενα από λογισμικό (SDN), το network slicing και το network softwarization. Επιπλέον, αναλύονται τεχνολογίες δικτύου και υπηρεσιών όπως το Tailscale VPN, η εξισορρόπηση φορτίου και το service discovery στο Kubernetes (με εργαλεία όπως το MetalLB), η επίλυση DNS με χρήση dnsproxy/CoreDNS, η αναπαραγωγή βάσεων δεδομένων MySQL (master/slave replication) και η χρήση Ingress Controller με HTTPS (αξιοποιώντας self-signed certificates). Ιδιαίτερη έμφαση δίνεται στη λειτουργική περιγραφή και στην αρχιτεκτονική αυτών των τεχνολογιών, καθώς και στο πώς συμβάλλουν στην αυτοματοποίηση, την παρατηρησιμότητα, την ασφάλεια και την επεκτασιμότητα του συστήματος. Τέλος, αναδεικνύεται ο τρόπος με τον οποίο όλες αυτές οι τεχνολογίες συνδέονται και συνεργάζονται σε ένα ενιαίο, πλήρως τοπικό περιβάλλον (on-premise), σε αντίθεση με μια υλοποίηση στο cloud.

2.2 Εικονικοποίηση vs. Containerization (Docker)

Η **εικονικοποίηση (virtualization)** επιτρέπει τη δημιουργία πολλαπλών «εικονικών μηχανών» πάνω σε ένα φυσικό σύστημα, μέσω ενός επιπέδου αφαιρετικότητας πάνω από το υλικό. Ένας υπολογιστής μπορεί να διαιρεθεί σε πολλές ανεξάρτητες εικονικές μηχανές, καθεμία με το δικό της λειτουργικό σύστημα και πόρους, σαν να ήταν ξεχωριστός υπολογιστή. Το λογισμικό που το επιτυγχάνει ονομάζεται hypervisor, ο οποίος διαχειρίζεται τους πόρους υλικού και επιτρέπει σε πολλαπλά λειτουργικά συστήματα (τους “επισκέπτες”) να τρέχουν ταυτόχρονα πάνω στον ίδιο εξυπηρετητή (τον “host”). Αυτή η προσέγγιση

προσφέρει ισχυρή απομόνωση – μια αστοχία ή επίθεση σε ένα VM δεν επηρεάζει άμεσα άλλες – αλλά έχει και σημαντικό overhead, καθώς κάθε VM φέρει ολόκληρο λειτουργικό σύστημα, καταναλώνοντας περισσότερους πόρους σε CPU, μνήμη και χρόνο εκκίνησης.

Αντίθετα, το **containerization** (κοντεϊνοποίηση) είναι μια τεχνολογία ελαφρύτερης εικονικοποίησης σε επίπεδο λειτουργικού συστήματος. Αντί για πολλαπλά πλήρη λειτουργικά, τα **containers (κοντέινερ)** μοιράζονται τον ίδιο kernel του host λειτουργικού και απομονώνονται μεταξύ τους στο επίπεδο του χρήστη, των διεργασιών και του συστήματος αρχείων. Κάθε container περιέχει μόνο τη δική του εφαρμογή και τις βιβλιοθήκες/εξαρτήσεις που χρειάζεται, δημιουργώντας ένα πακέτο που μπορεί να τρέξει οπουδήποτε χωρίς αλλαγές. Τα containers είναι πολύ πιο ελαφριά από τις VMs και εκκινούν πολύ γρηγορότερα, επειδή δεν χρειάζεται να εκκινήσουν ένα πλήρες λειτουργικό σύστημα κάθε φορά. Αυτό σημαίνει μικρότερο αποτύπωμα σε πόρους και καθιστά τα containers ιδανικά για τη γρήγορη ανάπτυξη και κλιμάκωση σύγχρονων εφαρμογών (π.χ. μικροϋπηρεσιών). Από την άλλη πλευρά, σε θέματα απομόνωσης και ασφάλειας, οι εικονικές μηχανές υπερτερούν, καθώς κάθε VM είναι πλήρως απομονωμένη. Στα containers, μια ευπάθεια στον kernel του host θα μπορούσε θεωρητικά να επηρεάσει όλα τα containers, διότι μοιράζονται τον ίδιο kernel. Συνοψίζοντας, το containerization προσφέρει ταχεία ανάπτυξη, φορητότητα και επεκτασιμότητα, ενώ η παραδοσιακή εικονικοποίηση παρέχει πλήρη απομόνωση δημιουργώντας εικονικό υλικό (υλικό, δίκτυο, αποθήκευση) για κάθε instance.

Μια από τις πιο δημοφιλείς πλατφόρμες containerization είναι το **Docker**. Το Docker αποτελεί μια πλατφόρμα κοντέινερ ανοικτού κώδικα που παρέχει τη δυνατότητα στους developers να αναπτύσσουν, να εκτελούν και να διαχειρίζονται εφαρμογές σε διαφορετικά περιβάλλοντα με εύκολο και αποτελεσματικό τρόπο. Το Docker εισήγαγε έναν τυποποιημένο τρόπο «πακεταρίσματος» εφαρμογών και εξαρτήσεων σε εικόνες (images) και την εκτέλεσή τους ως απομονωμένα containers. Μια εικόνα Docker (Docker Image) είναι ένα αρχείο μόνο-για-ανάγνωση που περιλαμβάνει τις βασικές οδηγίες για τη δημιουργία ενός container (το λειτουργικό σύστημα, τις βιβλιοθήκες, τον κώδικα της εφαρμογής κ.ά.). Όταν εκτελείται μια εικόνα, δημιουργείται ένα container – μια ζωντανή εκτελούμενη

instance της εικόνας. Τα containers του Docker είναι φορητά και μπορούν να τρέξουν σε οποιοδήποτε σύστημα υποστηρίζει τον Docker, διευκολύνοντας έτσι τη μεταφορά εφαρμογών από το ένα περιβάλλον στο άλλο (ανάπτυξη, δοκιμή, παραγωγή) χωρίς δυσκολίες συμβατότητας. Επίσης, είναι πολύ ελαφριά και απομονωμένα μεταξύ τους, περιέχοντας μόνο το απαραίτητο λογισμικό και τις βιβλιοθήκες για την εφαρμογή. Αυτό επιτρέπει την καλύτερη αξιοποίηση πόρων και την ταχύτερη εκκίνηση/τερματισμό εφαρμογών σε σύγκριση με τις παραδοσιακές VMs.

2.3 Ενορχήστρωση Containers με Kubernetes

Η διαχείριση μεμονωμένων containers σε μεγάλη κλίμακα απαιτεί αυτοματισμό και συντονισμό – εδώ έρχεται η **ενορχήστρωση (orchestration)** των containers. Το **Kubernetes** (συντομογραφία: K8s) είναι το δημοφιλέστερο σύστημα ενορχήστρωσης containers, ανοικτού κώδικα, που χρησιμοποιείται για την αυτοματοποίηση της ανάπτυξης, της κλιμάκωσης και της διαχείρισης containerized εφαρμογών. Αρχικά αναπτύχθηκε από την Google, πλέον όμως υποστηρίζεται από το Cloud Native Computing Foundation (CNCF). Με το Kubernetes, οι διαχειριστές ορίζουν δηλωτικά την επιθυμητή κατάσταση του συστήματος (ποιες εφαρμογές/υπηρεσίες να τρέχουν, πόσα instances, ποιο network/tls config κ.λπ.) και το Kubernetes αναλαμβάνει να διατηρεί αυτό το επιθυμητό state αυτόματα. Βασικό χαρακτηριστικό του Kubernetes είναι ότι προσφέρει υψηλή διαθεσιμότητα για τις εφαρμογές, μέσω αυτόματης αντιγραφής (replication) των containers μιας εφαρμογής σε πολλούς κόμβους (nodes) του cluster. Έτσι, αν κάποιο container ή ακόμα και ολόκληρος κόμβος αποτύχει, οι υπόλοιποι θα συνεχίσουν να τρέχουν την εφαρμογή, διατηρώντας την προσβάσιμη.

2.3.1 Αρχιτεκτονική: Control Plane και Nodes

Ένα cluster Kubernetes αποτελείται από το **control plane** (επίπεδο ελέγχου) και ένα ή περισσότερους κόμβους εργασίας (**worker nodes**). Το control plane είναι υπεύθυνο για τη συνολική διαχείριση της κατάστασης του cluster, ενώ οι κόμβοι εκτελούν τα application workloads (containers).

Control Plane Components: Τα κύρια συστατικά του control plane περιλαμβάνουν μια σειρά από υπηρεσίες που συνεργάζονται στενά:

- **API Server (kube-apiserver)**: Ο κεντρικός διακομιστής που εκθέτει το Kubernetes API (REST API) προς το οποίο μιλούν όλα τα εργαλεία (kubectl, controllers, etc.). Όλες οι εντολές/αιτήσεις για δημιουργία, ενημέρωση ή διαγραφή αντικειμένων του Kubernetes (Pods, Services κ.λπ.) περνούν από εδώ.
- **Etcd**: Μια κατανεμημένη, αξιόπιστη βάση δεδομένων τύπου key-value, όπου το Kubernetes αποθηκεύει όλη τη κατάσταση και τις ρυθμίσεις του cluster (το desired state και το current state). Το etcd αποτελεί την "πηγή αλήθειας" για το cluster – ο API server το χρησιμοποιεί για να διαβάζει/γράφει δεδομένα.
- **Scheduler (kube-scheduler)**: Η λειτουργία που αναλαμβάνει να αναθέτει κάθε νέο Pod σε έναν κατάλληλο worker node. Ο scheduler εντοπίζει Pods που δεν έχουν ακόμα προγραμματιστεί (bound) σε κόμβο και, λαμβάνοντας υπόψη περιορισμούς πόρων και πολιτικές, αποφασίζει σε ποιον κόμβο θα το εκκινήσει.
- **Controller Manager (kube-controller-manager)**: Εκτελεί τους εσωτερικούς controllers του Kubernetes – δηλαδή διεργασίες που παρακολουθούν την κατάσταση του cluster και επεμβαίνουν για να διασφαλίσουν ότι πληρείται η επιθυμητή κατάσταση. Για παράδειγμα, ο ReplicaSet controller φροντίζει να τρέχουν πάντα N αντίγραφα ενός pod, δημιουργώντας ή καταστρέφοντας pods ανάλογα, ενώ άλλοι controllers συγχρονίζουν endpoints υπηρεσιών, διαχειρίζονται κόμβους που χάθηκαν κ.ο.κ.. (Σε περιβάλλον cloud, υπάρχει και ο cloud-controller-manager για ενοποίηση με το cloud provider, αλλά σε ένα αμιγώς τοπικό cluster δεν χρησιμοποιείται αυτή η συνιστώσα.)

Node Components: Κάθε worker node τρέχει τα δικά του components που εξασφαλίζουν την εκτέλεση των containers και τη δικτύωση:

- **Kubelet**: Ένας agent (daemon) που τρέχει σε κάθε node και λαμβάνει οδηγίες από το control plane. Ο kubelet αναλαμβάνει να διασφαλίζει ότι στον κόμβο τρέχουν όλα τα pods/containers που του έχουν ανατεθεί (επικοινωνώντας με το container runtime για τη δημιουργία και παρακολούθησή τους). Αν π.χ. ο scheduler προγραμματίσει ένα νέο

pod στον κόμβο, ο kubelet θα κατεβάσει την αντίστοιχη image και θα εκκινήσει το container.

- **Container Runtime**: Το λογισμικό στον κάθε node που εκτελεί πραγματικά τα containers. Μπορεί να είναι το Docker Engine ή άλλες συμβατές runtime μηχανές (π.χ. containerd, CRI-O). Ο kubelet διαμέσου του Container Runtime Interface (CRI) δίνει εντολές στο runtime για να δημιουργήσει/διαγράψει containers.
- **Kube-proxy**: Είναι μια υπηρεσία που τρέχει σε κάθε κόμβο και υλοποιεί το εικονικό δίκτυο υπηρεσιών του Kubernetes. Συγκεκριμένα, το kube-proxy δημιουργεί και διαχειρίζεται τους κανόνες δικτύωσης (iptables ή IPVS rules) στους κόμβους, ώστε να δρομολογούνται σωστά τα πακέτα προς τα pods που εξυπηρετούν μια υπηρεσία (Service). Με αυτόν τον τρόπο επιτυγχάνεται η εσωτερική εξισορρόπηση φορτίου (load balancing) μεταξύ πολλών pods που υλοποιούν την ίδια υπηρεσία.

Συνολικά, το Kubernetes έχει μια επεκτάσιμη και ευέλικτη αρχιτεκτονική. Υπάρχουν πρόσθετα components ή addons που μπορούν να τρέχουν στο cluster για επιπλέον λειτουργίες – για παράδειγμα, **DNS** (συνήθως με CoreDNS) για την επίλυση ονομάτων υπηρεσιών σε όλο το cluster, **Web UI (Dashboard)** για γραφική διαχείριση, **monitoring agents** κ.ά. Η αρχιτεκτονική μπορεί να προσαρμοστεί τόσο σε μικρά δοκιμαστικά περιβάλλοντα (π.χ. όλα τα components σε έναν μηχανισμό) όσο και σε μεγάλης κλίμακας παραγωγικές εγκαταστάσεις με πολλούς master και worker κόμβους για αυξημένη ανθεκτικότητα.

2.3.2 Υπηρεσίες, Load Balancing και Service Discovery στο Kubernetes

Μία από τις βασικές έννοιες του Kubernetes είναι η **Υπηρεσία (Service)** – ένας αφηρημένος πόρος που ορίζει έναν σταθερό τρόπο πρόσβασης σε ένα δυναμικό σύνολο pods. Καθώς τα pods είναι θνησιγενή και έχουν δυναμικές διευθύνσεις IP, μια Service παρέχει ένα σταθερό endpoint (όνομα DNS και εικονική IP) πίσω από το οποίο τα pods **εξισορροπούνται φορτιακά** (load-balanced). Το Kubernetes υποστηρίζει διάφορους τύπους Service για διαφορετικά σενάρια πρόσβασης:

- **ClusterIP**: Ο προεπιλεγμένος τύπος Service. Παρέχει μια εικονική IP (cluster-internal) μέσω της οποίας ένα σετ από pods (ομάδα με κοινό label selector) δέχονται traffic. Η IP

αυτή είναι προσβάσιμη μόνο εντός του cluster (από άλλα pods/nodes) και όχι από εξωτερικούς clients. Η εξισορρόπηση φορτίου γίνεται εσωτερικά από το kube-proxy, που προωθεί τα πακέτα σε ένα από τα pods του service.

- **NodePort**: Επεκτείνει το ClusterIP εκθέτοντας την υπηρεσία και εκτός cluster. Ορίζεται ένας συγκεκριμένος port (στην περιοχή 30000-32767) σε κάθε node, στον οποίο αν συνδεθεί κάποιος (στην IP του node), η κίνηση προωθείται στο ClusterIP service και από εκεί στα pods. Με αυτόν τον τρόπο, οι εξωτερικοί χρήστες μπορούν να προσπελάσουν την υπηρεσία χρησιμοποιώντας τη διεύθυνση IP οποιουδήποτε node και τον συγκεκριμένο NodePort.
- **LoadBalancer**: Αυτός ο τύπος Service χρησιμοποιείται κυρίως σε περιβάλλον cloud. Όταν δημιουργείται μια υπηρεσία τύπου LoadBalancer, το Kubernetes ζητά από τον υποκείμενο πάροχο (π.χ. AWS, GCP, Azure) να δημιουργήσει και να ρυθμίσει έναν εξωτερικό load balancer που δίνει πρόσβαση στην υπηρεσία. Ο cloud load balancer λαμβάνει μια δημόσια IP (ή DNS όνομα) και δρομολογεί την κίνηση προς τα NodePort endpoints του service, επιτρέποντας πρόσβαση από το διαδίκτυο. Σε ένα ιδιωτικό/on-premises cluster, όμως, δεν υπάρχει τέτοιος πάροχος – αν ζητηθεί μια υπηρεσία LoadBalancer, η External IP της θα μείνει σε **Pending** κατάσταση, διότι λείπει ο μηχανισμός δημιουργίας load balancer.
- **ExternalName**: Αυτός ο ειδικός τύπος Service δεν παρέχει IP/προώθηση, αλλά επιστρέφει έναν DNS alias (CNAME) προς κάποιο εξωτερικό όνομα. Δεν χρησιμοποιείται για τα containers απευθείας, αλλά για να γίνεται π.χ. alias μια εξωτερική υπηρεσία εντός του cluster (δεν θα επεκταθούμε περαιτέρω εδώ).

Στο πλαίσιο ενός πλήρως τοπικού cluster (χωρίς cloud provider), η απουσία ενσωματωμένου μηχανισμού εξωτερικού load balancing καλύπτεται από λογισμικό όπως το **MetalLB**. Το MetalLB είναι μια υλοποίηση load-balancer ειδικά για Kubernetes clusters σε **bare-metal** ή γενικά on-prem περιβάλλοντα, που χρησιμοποιεί τυπικά πρωτόκολλα δικτύωσης για να ανακοινώνει εικονικές IP στο τοπικό δίκτυο. Με το MetalLB εγκατεστημένο, κάθε φορά που δημιουργείται ένα Service τύπου LoadBalancer, το MetalLB αναλαμβάνει να εκχωρήσει σε αυτό μια διαθέσιμη **External IP** από ένα προκαθορισμένο εύρος (pool) διευθύνσεων του

τοπικού δικτύου. Ανάλογα με τη λειτουργία του, μπορεί είτε (α) να ανακοινώνει την IP μέσω **ARP/NDP** στους γειτονικούς δρομολογητές ώστε να δρομολογούν την κίνηση προς τους κόμβους που τρέχουν τα pods (Layer2 mode), είτε (β) να μιλάει μέσω **BGP** στο δίκτυο, διαφημίζοντας τις IP ώστε το δίκτυο να γνωρίζει ότι η συγκεκριμένη IP είναι προσβάσιμη μέσω του cluster (Layer3 mode). Έτσι, καλύπτεται το «κενό» που υπάρχει στα on-premise clusters: παρέχεται ένας πραγματικός εξωτερικός load balancer σε επίπεδο λογισμικού. Με το MetalLB, η External IP μιας υπηρεσίας παύει να είναι σε pending κατάσταση – διατίθεται κανονικά και είναι προσβάσιμη από clients του δικτύου.

Service Discovery (Εύρεση Υπηρεσιών): Για να μπορούν οι εφαρμογές και χρήστες να βρίσκουν τις υπηρεσίες, το Kubernetes ενσωματώνει ένα μηχανισμό DNS. Συγκεκριμένα, κάθε cluster εγκαθιστά ένα add-on DNS server (CoreDNS) που κρατά ενημερωμένες εγγραφές DNS για όλα τα Services και Pods. Έτσι, κάθε Service αποκτά ένα όνομα DNS της μορφής ονομα-υπηρεσίας.ονομα-namespace.svc.cluster.local το οποίο επιλύεται στην cluster IP της υπηρεσίας. Τα pods μπορούν απλά να κάνουν DNS query το όνομα της υπηρεσίας και να λάβουν την IP, μέσω της οποίας το kube-proxy θα τα προωθήσει σε ένα από τα διαθέσιμα pods. Αυτό αυτοματοποιεί πλήρως το service discovery εντός του cluster. Σε ένα τοπικό περιβάλλον, αν θέλουμε clients εκτός του cluster (π.χ. χρήστες στο LAN ή μέσω VPN) να επιλύουν ονόματα προς τις υπηρεσίες μας, μπορούμε να στήσουμε έναν τοπικό DNS resolver (π.χ. dnsproxy σε συνδυασμό με CoreDNS) που θα προωθεί ερωτήματα για συγκεκριμένα domain προς το DNS του Kubernetes. Για παράδειγμα, μπορεί να οριστεί ένας dnsproxy ώστε όλα τα ερωτήματα για έναν τοπικό τομέα π.χ. *.mylab.local να απαντώνται με τις IP που δίνει το MetalLB για το cluster. Το **dnsproxy** είναι ένα ελαφρύ DNS forwarder (by AdGuard) που μπορεί να εκτελεί τέτοιου είδους προώθηση αιτήσεων, ακόμη και υποστηρίζοντας κρυπτογράφηση DNS (DoH/DoT) αν χρειάζεται. Σε συνδυασμό με το CoreDNS (που μπορεί να παραμετροποιηθεί κατάλληλα), επιτρέπεται η ενιαία επίλυση ονομάτων τόσο εντός του cluster όσο και από συσκευές εκτός cluster, χωρίς να απαιτείται χειροκίνητη ρύθμιση αρχείων hosts.

2.3.3 Ingress Controller και HTTPS με αυτοϋπογεγραμμένα πιστοποιητικά

Εκτός από τις υπηρεσίες τύπου Service, το Kubernetes εισάγει τον πόρο **Ingress** για την έξυπνη δρομολόγηση HTTP/HTTPS αιτημάτων προς τις εφαρμογές. Ένας πόρος Ingress ορίζει κανόνες routing σε επίπεδο URL/hostname: για παράδειγμα, όλα τα αιτήματα προς https://app.example.local/login να προωθούνται στο αντίστοιχο Service ενός web server μέσα στο cluster. Ο ίδιος ο ορισμός Ingress είναι διακηρυκτικός – απαιτείται μια υλοποίηση (ο **Ingress Controller**) που θα διαβάζει αυτούς τους κανόνες και θα τους εφαρμόζει στην πράξη. Συνήθως ο Ingress Controller είναι ένας reverse proxy / load balancer HTTP που τρέχει ο ίδιος μέσα στο cluster (σε pods) και προσαρμόζεται δυναμικά. Δημοφιλείς Ingress controllers είναι το **NGINX Ingress**, το **HAProxy Ingress**, το **Traefik** κ.α. Στο τοπικό μας cluster, ο Ingress Controller παίζει τον ρόλο της πύλης εισόδου (gateway) για όλη την web κίνηση.

Με χρήση Ingress, μπορούμε να έχουμε ένα μόνο σημείο πρόσβασης (π.χ. μία διεύθυνση IP από το MetalLB, ή ένα DNS όνομα) και να δρομολογούμε πολλαπλές εφαρμογές πίσω από αυτό, ανάλογα με το URL ή το hostname. Αυτό μοιάζει με το γνωστό pattern του reverse proxy στα data centers. Επιπλέον, ο Ingress Controller μπορεί να τερματίζει συνδέσεις **HTTPS**. Δηλαδή, μπορεί να προσφέρει **TLS termination**: οι εξωτερικοί clients να κάνουν HTTPS σύνδεση ασφαλή προς τον Ingress (ο οποίος παρουσιάζει ένα πιστοποιητικό TLS), και έπειτα ο Ingress να προωθεί εσωτερικά την κίνηση στα pods (μπορεί είτε σε HTTP plain είτε επίσης κρυπτογραφημένα αν θέλουμε end-to-end TLS). Σε ένα τοπικό περιβάλλον χωρίς δημόσιο Certificate Authority, συχνά χρησιμοποιούμε **αυτοϋπογεγραμμένα πιστοποιητικά (self-signed certificates)** για το HTTPS. Δηλαδή, δημιουργούμε ένα ζεύγος κλειδιού/πιστοποιητικού και το υπογράφουμε μόνοι μας (ή από μια τοπική CA της υποδομής μας), το οποίο στη συνέχεια ρυθμίζουμε στον Ingress Controller. Με αυτό τον τρόπο, η επικοινωνία μεταξύ των χρηστών και του Ingress είναι κρυπτογραφημένη (εξασφαλίζοντας μυστικότητα των δεδομένων) ακόμα και σε τοπικό δίκτυο. Το μειονέκτημα είναι ότι τα self-signed certs δεν αναγνωρίζονται αυτομάτως ως έμπιστα από τα προγράμματα πλοήγησης, οπότε ο χρήστης θα δει προειδοποίηση ασφαλείας εκτός αν εγκαταστήσει το τοπικό CA στον υπολογιστή του. Παρ’ όλ’ αυτά, για εργαστηριακή χρήση ή περιορισμένο περιβάλλον αυτό είναι αποδεκτό. Εναλλακτικά, θα μπορούσε να

χρησιμοποιηθεί και μια υπηρεσία τύπου Let’s Encrypt με DNS-01 challenge σε ένα όνομα domain αν υπάρχει πρόσβαση, όμως σε πλήρως απομονωμένο περιβάλλον το self-signed είναι η πρακτική λύση.

2.4 Λογισμικο-κεντρικές Δικτυακές Τεχνολογίες: SDN, Network Slicing, Network Softwarization

Η παραδοσιακή δικτύωση βασιζόταν σε εξειδικευμένο hardware και στατικά διαμορφωμένο εξοπλισμό (switches, routers) για την προώθηση της κίνησης. Οι σύγχρονες απαιτήσεις για ευελιξία, απομόνωση υπηρεσιών και γρήγορη ανάπτυξη νέων λειτουργιών οδήγησαν σε μια επανάσταση στον χώρο των δικτύων: τα δίκτυα **οριζόμενα από λογισμικό (Software-Defined Networking - SDN)** και γενικότερα η **“επιλογή του δικτύου μέσω λογισμικού” (network softwarization)**.

Software-Defined Networking (SDN): Με απλά λόγια, SDN σημαίνει ότι ο **έλεγχος** ενός δικτύου απομακρύνεται από τον παραδοσιακό, αποκεντρωμένο τρόπο (όπου κάθε συσκευή έχει το δικό της κλειστό λογισμικό που λαμβάνει αποφάσεις δρομολόγησης), και αντί αυτού συγκεντρώνεται λογικά σε ένα κεντρικό σημείο ελέγχου (SDN controller). Ο πιο συνηθισμένος ορισμός του SDN μιλά για **συνολικό έλεγχο του δικτύου μέσω της λογικής κεντρικοποίησης της λειτουργίας του επιπέδου ελέγχου**, με τα διάφορα στοιχεία του δικτύου να αντιμετωπίζονται ως ενιαία οντότητα. Δηλαδή, στο SDN διαχωρίζεται σαφώς το **control plane** (που αποφασίζει πώς θα δρομολογηθούν οι ροές) από το **data plane** (που είναι οι συσκευές που προωθούν τα πακέτα). Ο controller – υλοποιημένος σε λογισμικό – επικοινωνεί με τους δρομολογητές/διακόπτες (data plane) μέσω ανοιχτών διεπαφών (όπως το πρωτόκολλο **OpenFlow** ή άλλες API) και εγκαθιστά κανόνες προώθησης σε αυτούς. Με αυτόν τον τρόπο, το δίκτυο γίνεται προγραμματιζόμενο: οι διαχειριστές ή ακόμα και ανώτερα επίπεδα λογισμικού μπορούν να καθορίσουν με κώδικα πώς θα συμπεριφέρεται το δίκτυο. Το SDN επιτρέπει, για παράδειγμα, τη δυναμική αλλαγή των διαδρομών της κίνησης, την εφαρμογή πολιτικών ασφάλειας κεντρικά για όλο το δίκτυο, καθώς και την εύκολη δημιουργία **εικονικών δικτύων** πάνω από τη φυσική υποδομή.

Network Slicing: Το network slicing είναι μια τεχνική (που απέκτησε ιδιαίτερη φήμη στο 5G δίκτυο) με την οποία δημιουργούνται πολλαπλά **εικονικά "φέτες" δικτύου** πάνω στην ίδια φυσική υποδομή. Κάθε slice είναι ένα λογικά ξεχωριστό δίκτυο, με τις δικές του παραμέτρους, τοπολογία, ακόμα και πολιτικές ασφάλειας ή ποιότητας υπηρεσίας, προσαρμοσμένες σε συγκεκριμένη χρήση. Η ιδέα είναι ότι ένας πάροχος (ή μια επιχείρηση) μπορεί να "κόψει" το δίκτυό του σε ξεχωριστά τμήματα: για παράδειγμα, ένα slice για υπηρεσίες χαμηλής καθυστέρησης και υψηλής αξιοπιστίας (π.χ. βιομηχανικός αυτοματισμός), ένα άλλο slice για ευρυζωνικό Internet υψηλού throughput για κοινόχρηστη χρήση, και ένα τρίτο slice για IoT συσκευές με μικρές απαιτήσεις. Όλα αυτά μοιράζονται το ίδιο φυσικό δίκτυο (κεραίες, καλώδια, δρομολογητές), αλλά επειδή είναι οριζόμενα από λογισμικό, κάθε slice αντιμετωπίζεται σαν ξεχωριστό δίκτυο με εγγυημένους πόρους. Στην πράξη, το SDN και οι εικονικές δικτυακές τεχνολογίες (όπως **VPNs, VLANs** και ειδικά στο 5G τα **Network Slice Subnets**) υλοποιούν το slicing. Αξίζει να σημειωθεί ότι αν και το slicing έγινε ευρέως γνωστό μέσω του 5G, η έννοια υπάρχει από παλιά: π.χ. ένα κλασικό VLAN σε Ethernet είναι μια μορφή «τεμαχισμού» ενός φυσικού LAN σε πολλαπλά λογικά LAN. Το SDN απλώς επιτρέπει να γίνεται αυτό σε μεγαλύτερη κλίμακα και με μεγαλύτερη ευελιξία, καθορίζοντας μέσω κώδικα διαφορετικούς κανόνες διαχείρισης πακέτων για διαφορετικές ροές.

Network Softwarization: Ο όρος αυτός εκφράζει συνολικά την τάση **«μετασχηματισμού του δικτύου μέσω λογισμικού»**, που περιλαμβάνει το SDN αλλά και το NFV (Network Function Virtualization) και γενικότερα την μετάβαση από κλειστά, εξειδικευμένα δίκτυα σε δίκτυα που υλοποιούνται πάνω σε κοινό hardware με ευέλικτο λογισμικό. Παλαιότερα, κάθε λειτουργία (router, firewall, load balancer) συχνά απαιτούσε ειδική συσκευή με ιδιοταγές λογισμικό. Με το softwarization, αυτές οι λειτουργίες μετατρέπονται σε **εφαρμογές λογισμικού** που μπορούν να τρέξουν σε commodity servers ή virtual machines, δίνοντας τη δυνατότητα οι πάροχοι να λειτουργούν τα δίκτυά τους σε κοινούς data centers, τρέχοντας διαφορετικά είδη εξοπλισμού ως virtualized network functions (VNFs). Αυτό **«αποσυνδέει» το δίκτυο** σε επίπεδα: το υποκείμενο hardware από το λογισμικό, και επιτρέπει μια πολύ πιο ανοικτή και ευέλικτη αρχιτεκτονική. Η δικτυακή υποδομή γίνεται πια παρόμοια με τον

κόσμο του cloud: μπορούν να προστεθούν ή να ενημερωθούν δυνατότητες δικτύου μέσω software updates, να **προγραμματιστούν** δίκτυα μέσω API, και να αυτοματοποιηθεί η διαχείρισή του όπως ακριβώς γίνεται το automation σε servers. Αυτή η επανάσταση βρίσκεται ακόμη σε εξέλιξη και μεταμορφώνει ολόκληρη τη βιομηχανία τηλεπικοινωνιών και δικτύων. Για τους διαχειριστές ενός on-premise cluster, οι αρχές του network softwarization αντικατοπτρίζονται σε λύσεις όπως: virtual switches/routers (π.χ. το Linux με iptables/ebtables λειτουργεί ως εικονικός δρομολογητής), software load balancers (όπως το MetalLB που αναφέραμε), VPN σε επίπεδο εφαρμογής (όπως το Tailscale παρακάτω), και γενικά η δυνατότητα να περιγραφεί το δίκτυο και τις πολιτικές του μέσω λογισμικού (Infrastructure as Code).

2.5 Δικτύωση VPN με Tailscale

Σε μια τοπική υποδομή που δεν βρίσκεται σε δημόσιο cloud, η ασφαλής απομακρυσμένη πρόσβαση και η διασύνδεση γεωγραφικά διασπασμένων κόμβων μπορεί να αποτελέσει πρόκληση. Εδώ έρχονται τα **Virtual Private Networks (VPNs)** και ειδικά οι σύγχρονες υλοποιήσεις τύπου mesh VPN όπως το **Tailscale**. Το Tailscale είναι μια υπηρεσία mesh VPN που απλουστεύει τη διαδικασία ασφαλούς διασύνδεσης συσκευών και υπηρεσιών μέσα από διαφορετικά δίκτυα. Βασίζεται στο ανοικτού κώδικα πρωτόκολλο **WireGuard** για την κρυπτογράφηση των επικοινωνιών, δημιουργώντας απευθείας κρυπτογραφημένες συνδέσεις μεταξύ των συσκευών (peer-to-peer) αντί να δρομολογεί όλη την κίνηση μέσω ενός κεντρικού κόμβου όπως οι παραδοσιακοί VPNs. Με άλλα λόγια, το Tailscale στήνει ένα ιδιωτικό overlay δίκτυο (το λεγόμενο *tailnet*) όπου κάθε συμμετέχουσα συσκευή λαμβάνει μια ιδιωτική IP και μπορεί να επικοινωνεί απευθείας με τις υπόλοιπες, σαν να βρίσκονταν στο ίδιο τοπικό δίκτυο.

Ένα από τα μεγάλα πλεονεκτήματα του Tailscale είναι η **ευχρηστία** του: η ανταλλαγή κλειδιών και η ρύθμιση των tunnels γίνεται αυτόματα μέσω ενός κεντρικού coordination server (control plane του Tailscale), ο οποίος όμως *δεν* βλέπει την ίδια την κίνηση (απλώς βοηθά στο να εδραιωθούν οι P2P συνδέσεις). Έτσι, μπορούμε εύκολα να συνδέσουμε πολλούς υπολογιστές, VM ή ακόμα και containers σε ένα κοινό ιδιωτικό δίκτυο χωρίς

περίπλοκες ρυθμίσεις firewall/NAT. Σε ένα σενάριο τοπικού cluster, μπορούμε να εγκαταστήσουμε το Tailscale σε κάθε node ή σε ένα gateway μηχάνημα, ώστε διαχειριστές/προγραμματιστές να συνδέονται μέσω Internet στο tailnet και να έχουν άμεση πρόσβαση στο cluster σαν να ήταν στο γραφείο. Αυτό παρέχει **ασφαλή απομακρυσμένη πρόσβαση** για διαχείριση (π.χ. kubectl προς το API server) ή για χρήση των υπηρεσιών που τρέχουν στο cluster, χωρίς να χρειάζεται να εκθέσουμε θύρες στο δημόσιο Internet. Επιπλέον, το Tailscale υποστηρίζει λειτουργίες όπως ACLs (Access Control Lists) για τον περιορισμό της επικοινωνίας μεταξύ κόμβων, integration με identity providers για αυθεντικοποίηση χρηστών, και ακόμα και **Magic DNS**, δηλ. δυνατότητα επίλυσης των ονομάτων των κόμβων μέσα στο tailnet χωρίς ξεχωριστή ρύθμιση DNS. Με αυτό τον τρόπο, το Tailscale ενισχύει την ασφάλεια του on-premise περιβάλλοντος (όλη η κίνηση είναι κρυπτογραφημένη) και την ευελιξία (οι developers μπορούν να εργάζονται απομακρυσμένα με ασφαλή τρόπο).

Συνολικά, το Tailscale και παρόμοια εργαλεία εντάσσονται στην φιλοσοφία του SDN/softwarization: αντί για παραδοσιακό VPN με στατικό server, έχουμε ένα **λογισμικό VPN mesh** που προσαρμόζεται δυναμικά, προσφέροντας υψηλότερο throughput και χαμηλότερη καθυστέρηση επειδή αποφεύγεται η συγκέντρωση όλης της κίνησης σε ένα κεντρικό σημείο. Η αρχιτεκτονική peer-to-peer βελτιώνει επίσης την αξιοπιστία – δεν υπάρχει ένα μοναδικό σημείο αποτυχίας που θα κατέρριπτε την συνδεσιμότητα όλων των κόμβων. Αυτό είναι ιδιαίτερα σημαντικό για τοπικές εγκαταστάσεις που απαιτούν συνεχή πρόσβαση από πολλαπλές τοποθεσίες.

2.6 Παρατηρησιμότητα και Monitoring: Prometheus & Grafana

Σε ένα κατανεμημένο σύστημα όπως ένα Kubernetes cluster, η **παρατηρησιμότητα (observability)** και η ενεργή **παρακολούθηση (monitoring)** είναι κρίσιμες για τη διασφάλιση της ομαλής λειτουργίας, της επίδοσης και της γρήγορης επίλυσης προβλημάτων. Δύο βασικά εργαλεία που χρησιμοποιούνται για αυτό το σκοπό είναι το **Prometheus** και το **Grafana**.

Prometheus: Είναι ένα ανοικτού κώδικα σύστημα monitoring και σύστημα καταγραφής χρονοσειρών (time-series database) ειδικά σχεδιασμένο για cloud-native περιβάλλοντα και κοντέινερ. Το Prometheus λειτουργεί με μοντέλο συλλογής δεδομένων τύπου *pull*: σε τακτά διαστήματα (π.χ. κάθε 15 δευτερόλεπτα) προβαίνει σε αιτήσεις (HTTP SCRAPE) προς προκαθορισμένα endpoints των εφαρμογών/συστημάτων για να συλλέξει μετρικές. Τα endpoints αυτά μπορεί να είναι οι ίδιοι οι εφαρμογικοί servers (αν είναι instrumented με Prometheus client libraries) ή ξεχωριστοί **exporters** που κάνουν expose μετρικές για συστήματα (π.χ. Node Exporter για metrics του λειτουργικού, MySQL exporter για DB metrics κ.λπ.). Όλες οι μετρικές αποθηκεύονται στο εσωτερικό χρόνο-σειριακό datastore του Prometheus, επιτρέποντας αποτελεσματική αποθήκευση και ανάκτηση για μεγάλους όγκους χρονικών δεδομένων. Το Prometheus προσφέρει την ισχυρή γλώσσα ερωτημάτων **PromQL**, με την οποία μπορούν να διατυπωθούν πολύπλοκα ερωτήματα πάνω στα δεδομένα (π.χ. ποσοστό σφαλμάτων HTTP 5xx την τελευταία 1 ώρα, μέσος χρόνος απόκρισης μιας υπηρεσίας κ.ά.) και να δημιουργηθούν **κανόνες alerting**. Μέσω alert rules, το Prometheus μπορεί να ενεργοποιήσει ειδοποιήσεις (συνεργαζόμενο με Alertmanager) όταν κάποια μέτρηση ξεπεράσει όριο (π.χ. υψηλή χρήση CPU, downtime υπηρεσίας κλπ.).

Grafana: Το Grafana είναι μια πολυπλατφορμική, ανοικτού κώδικα εφαρμογή οπτικοποίησης και analytics, που χρησιμοποιείται σε συνδυασμό με βάσεις metric όπως το Prometheus. Ο ρόλος του Grafana είναι να μετατρέπει τα δεδομένα που συλλέγει το Prometheus σε **διαδραστικούς πίνακες και γραφήματα** μέσω ενός web περιβάλλοντος. Συνδέεται ως **data source** στο Prometheus και επιτρέπει στον χρήστη να γράφει ερωτήματα (PromQL) και να τα παρουσιάζει σε γραφικές μορφές. Με το Grafana μπορούν εύκολα να δημιουργηθούν **dashboards** που συγκεντρώνουν πολλαπλά visualizations, δίνοντας μια ενοποιημένη εικόνα της κατάστασης του συστήματος σε πραγματικό χρόνο. Για παράδειγμα, ένα dashboard ενός web application μπορεί να περιέχει γραφήματα για τον ρυθμό αιτημάτων ανά δευτερόλεπτο, το latency, το ποσοστό σφαλμάτων, τη χρήση CPU/memory ανά υπηρεσία κ.ά., όλα ενημερωμένα ζωντανά. Επιπλέον, το Grafana υποστηρίζει τη σύνδεση πολλαπλών πηγών δεδομένων (όχι μόνο Prometheus – π.χ. Elasticsearch για logs, Loki, InfluxDB, Cloudwatch κ.ά.) και την **συσχέτιση** αυτών. Αυτό

σημαίνει ότι μπορούμε να έχουμε σε ένα dashboard τόσο metrics όσο και logs, για παράδειγμα, επιτρέποντας την ταχύτερη διάγνωση προβλημάτων (συσχέτιση γεγονότων). Το Grafana υποστηρίζει επίσης τη δημιουργία **alerts** (νέας γενιάς) όπου μπορεί να αξιοποιήσει τους ίδιους πίνακες για να παράγει ειδοποιήσεις, καθώς και διαμοιρασμό των dashboards με ομάδες, θέτοντας δικαιώματα πρόσβασης, κ.λπ.

Στο πλαίσιο της τοπικής μας υποδομής Kubernetes, το Prometheus εγκαθίσταται συνήθως μέσα στο cluster (π.χ. μέσω Helm chart ή operator) και ρυθμίζεται να ανακαλύπτει αυτόματα (Service Discovery) όλα τα pods/υπηρεσίες που εκθέτουν metrics (συχνά με annotations στα pods). Έτσι επιτυγχάνεται **αυτοματοποιημένο monitoring**: κάθε νέα υπηρεσία που γίνεται deploy με metrics, θα αρχίσει να συλλέγεται χωρίς πρόσθετη χειροκίνητη ρύθμιση. Το Grafana μπορεί επίσης να τρέχει εντός cluster ή σε ένα VM εκτός, και να συνδέεται στο Prometheus. Τα οφέλη που αποκομίζουμε είναι η **παρατηρησιμότητα** και η **έγκαιρη προειδοποίηση**: η ομάδα DevOps μπορεί να βλέπει τάσεις (π.χ. αύξηση φορτίου, αξιοποίηση πόρων) και να προλαμβάνει προβλήματα, ενώ αν κάτι πάει στραβά (π.χ. μια υπηρεσία πέσει ή επιβαρυνθεί), θα λάβει alerts για να επέμβει. Στην παραγωγή, τέτοια εργαλεία είναι ουσιαστικά για την αξιοπιστία (SRE πρακτικές), αλλά και σε μια τοπική εγκατάσταση είναι πολύ χρήσιμα για τη **βελτιστοποίηση** και την επίδειξη **observability** ως αρχής σχεδίασης.

2.7 Αναπαραγωγή Δεδομένων στη MySQL (Master-Slave Replication)

Η διαφύλαξη των δεδομένων και η διασφάλιση διαθεσιμότητάς τους είναι καίρια σε κάθε υποδομή. Στα συστήματα βάσεων δεδομένων, μια κοινή τεχνική για υψηλή διαθεσιμότητα και κατανομή φορτίου ανάγνωσης είναι η **αναπαραγωγή (replication)**. Στη MySQL – όπως και σε άλλες σχεσιακές βάσεις – υποστηρίζεται η αρχιτεκτονική **master-slave replication** (πρωτεύον-δευτερεύον ή source-replica). Σε αυτό το σχήμα, ένας κεντρικός server βάσης (ο **master**) δέχεται όλες τις εγγραφές (write operations) και είναι η αρχή της αλήθειας για τα δεδομένα. Ένας ή περισσότεροι **slave** servers συνδέονται στον master και λαμβάνουν μια συνεχή ροή ενημερώσεων, ώστε να διατηρούν ένα *συγχρονισμένο αντίγραφο* της βάσης δεδομένων.

Πιο συγκεκριμένα, ο master καταγράφει κάθε αλλαγή στα δεδομένα σε ένα ειδικό log (binary log). Οι slaves συνδέονται στον master και **ανακτούν αυτό το binary log** (ή τις νέες εγγραφές του) και εφαρμόζουν τις ίδιες αλλαγές στη δική τους βάση. Έτσι, οι slaves αντικατοπτρίζουν σε (σχεδόν) πραγματικό χρόνο τις εγγραφές του master. Η διαδικασία αυτή συνήθως είναι **ασύγχρονη**: ο master δεν περιμένει τους slaves να επιβεβαιώσουν ότι έλαβαν την αλλαγή πριν προχωρήσει. Αυτό σημαίνει ότι μπορεί να υπάρχει μια μικρή καθυστέρηση μεταξύ master και slave, αλλά και ότι ο master δεν επηρεάζεται επιδόσια από τους replicas. Υπάρχει και επιλογή **ημι-σύγχρονης** ή πλήρως σύγχρονης αναπαραγωγής (με acknowledgments), αλλά η default και πιο συνηθισμένη είναι η ασύγχρονη.

Τα οφέλη της master-slave αναπαραγωγής είναι πολλαπλά: (α) **Ανοχή σε βλάβες** – αν ο master αποτύχει, ένας από τους slaves μπορεί να προαχθεί σε νέο master (manual ή auto failover), μειώνοντας downtime. (β) **Κλίμακα στις αναγνώσεις** – οι εφαρμογές μπορούν να εκτελούν ερωτήματα μόνο-ανάγνωσης (SELECTs) στους slaves, αποφορτίζοντας τον master που χειρίζεται εγγραφές. Έτσι επιτυγχάνεται scaling σε read-heavy φόρτους εργασίας. (γ) **Backup** – ένας slave μπορεί να χρησιμοποιηθεί για backup ή αναλυτικά queries, ώστε να μην επηρεάζεται ο master.

Στην τοπική μας υλοποίηση, η εγκατάσταση MySQL replication σε containers (ή pods στο Kubernetes) σημαίνει ότι έχουμε π.χ. ένα Deployment για τον master και ένα για τον slave. Ο master εκκινεί με binary logging ενεργοποιημένο, και ο slave ρυθμίζεται με τα στοιχεία σύνδεσης στον master (host, log file & position ή GTID). Μόλις ξεκινήσει, ο slave θα αρχίσει να **“ακολουθεί”** τον master, αντιγράφοντας όλες τις νέες συναλλαγές. Πρέπει να διασφαλιστεί ότι ο αρχικός συγχρονισμός έγινε (π.χ. με ένα snapshot ή backup που φόρτωσε ο slave). Από εκεί και πέρα, η συντήρηση είναι σχετικά αυτόματη.

Είναι σημαντικό να παρακολουθούνται ενδείκτες όπως το **lag** (καθυστέρηση του slave πίσω από τον master), ώστε να γνωρίζουμε πόσο ενημερωμένο είναι το replica. Με το Prometheus/Grafana μπορούμε να συλλέξουμε μετρικές της MySQL (υπάρχουν exporters) για να παρακολουθούμε και αυτά τα στοιχεία. Εάν ο slave μείνει πίσω ή αποσυνδεθεί, αυτό πρέπει να ειδοποιείται για να αντιμετωπιστεί (π.χ. επανασυγχρονισμός).

Εναλλακτικές προσεγγίσεις replication στη MySQL περιλαμβάνουν το **MySQL Group Replication** ή το **Galera Cluster**, που επιτρέπουν multi-master σενάρια, αλλά για απλότητα και δεδομένου ότι θέλουμε ένα **master** και ένα **standby**, το κλασικό master-slave μοντέλο αρκεί και προσομοιώνει καταστάσεις πραγματικού κόσμου (π.χ. ένα production database server με ένα warm standby replica).

2.8 Ενοποίηση της Τοπικής Υποδομής: Αυτοματοποίηση, Παρατηρησιμότητα, Ασφάλεια και Επεκτασιμότητα

Όλες οι προαναφερθείσες τεχνολογίες και αρχιτεκτονικές συνδέονται μεταξύ τους για να υλοποιήσουν ένα πλήρες τοπικό περιβάλλον υποδομής Kubernetes που πληροί απαιτήσεις παρόμοιες με εκείνες ενός σύγχρονου cloud – αλλά με μέσα που ελέγχονται εξ ολοκλήρου on-premise. Βασικά ζητούμενα ήταν:

- **Αυτοματοποίηση**: Το Kubernetes και τα container technologies φέρνουν υψηλό βαθμό αυτοματοποίησης στην ανάπτυξη και διαχείριση εφαρμογών. Η δηλωτική φύση του Kubernetes (manifests, desired state) σε συνδυασμό με controllers που διατηρούν αυτό το state, αυτοματοποιεί διαδικασίες όπως η επανεκκίνηση crashed containers, ο εξισορροπημένος καταμερισμός pods σε νέους κόμβους, η αυτόματη κλιμάκωση pods ανάλογα με το φορτίο (Horizontal Pod Autoscaler), και η αυτόματη δέσμευση πόρων. Επιπλέον, εργαλεία όπως το Docker Compose (για απλούστερα setups) ή helm/Operators στο Kubernetes αυτοματοποιούν την εγκατάσταση σύνθετων υπηρεσιών (π.χ. MySQL replication operator). Στη δικτύωση, το SDN προσφέρει αυτοματοποίηση στη διαχείριση του traffic, ενώ το Tailscale αυτοματοποιεί τη σύσταση ασφαλών δικτυακών τοπολογιών χωρίς χειροκίνητες ρυθμίσεις σε routers. Έτσι, ο ανθρώπινος παράγοντας για επαναλαμβανόμενες εργασίες μειώνεται, αποτρέποντας λάθη και εξοικονομώντας χρόνο.
- **Παρατηρησιμότητα (Observability)**: Χωρίς τα κατάλληλα δεδομένα, ένα σύνθετο σύστημα θα ήταν «μαύρο κουτί». Με τον συνδυασμό Prometheus/Grafana, αποκτούμε πλήρη εικόνα του τι συμβαίνει μέσα στο cluster: από τη χαμηλού επιπέδου χρήση πόρων σε κάθε node (CPU, μνήμη, δίκτυο) μέχρι τους business δείκτες της κάθε

υπηρεσίας (αιτήματα/sec, error rates). Αυτό ενισχύει την **διαγνωστικότητα** – όταν κάτι πάει στραβά, οι μηχανικοί μπορούν να εντοπίσουν την αιτία (π.χ. ένα service που καθυστερεί λόγω υψηλού load ή ένα network πρόβλημα) μέσα από dashboards και alerts. Ειδικά σε ένα on-premise περιβάλλον, όπου δεν υπάρχει cloud provider να προσφέρει έτοιμες λύσεις monitoring, η εγκατάσταση αυτών των εργαλείων καλύπτει το κενό και εξασφαλίζει ότι η υποδομή είναι **διάφανη** στους διαχειριστές της. Η παρατηρησιμότητα ενισχύεται και από τον ίδιο τον σχεδιασμό της αρχιτεκτονικής: το Kubernetes βοηθά στην οργάνωση logs/metrics, ενώ το network softwarization (π.χ. SDN controllers) επιτρέπει και στο δίκτυο να παρέχει δεδομένα (flow logs, κ.λπ.). Όλα αυτά συνθέτουν ένα σύστημα όπου υπάρχει feedback loop – δεδομένα κατάστασης οδηγούν σε ενημερωμένες αποφάσεις για αλλαγές/βελτιστοποιήσεις.

- **Ασφάλεια**: Σε ένα on-premise cluster, η ασφάλεια πρέπει να σχεδιαστεί πολυεπίπεδα. Στο επίπεδο δικτύου, το Tailscale VPN εξασφαλίζει ότι η απομακρυσμένη πρόσβαση και η μεταξύ κόμβων επικοινωνία (αν γίνεται μέσω Internet) είναι κρυπτογραφημένη και περιορισμένη σε εξουσιοδοτημένους peers. Στο επίπεδο εφαρμογών, ο Ingress controller με HTTPS (έστω και self-signed) διασφαλίζει ότι και εντός του τοπικού LAN, οι ευαίσθητες ροές δεδομένων είναι κρυπτογραφημένες, μειώνοντας τον κίνδυνο υποκλοπών. Το Kubernetes από μόνο του παρέχει μηχανισμούς ασφάλειας: *namespaces* για πολυενοικίαση (multi-tenancy isolation), *Network Policies* (αν ενεργοποιηθούν) για έλεγχο της κυκλοφορίας μεταξύ pods, *Secrets* για διαχείριση credentials, κ.ά. Επιπλέον, μέσω του SDN/softwarization, η ασφάλεια γίνεται πιο ευέλικτη: κεντρικοποιημένος έλεγχος σημαίνει ότι μπορούμε να επιβάλλουμε ενημερώσεις πολιτικών σε όλο το δίκτυο με ένα κλικ (π.χ. blocking ενός IP ή throttling traffic). Η MySQL replication συμβάλλει έμμεσα στην ασφάλεια διαθεσιμότητας δεδομένων – σε περίπτωση καταστροφής του master, τα δεδομένα δεν χάνονται αλλά υπάρχουν στους replicas (πολλαπλά αντίγραφα). Τέλος, η φύση των containers βοηθά στην ασφάλεια μέσω απομόνωσης: κάθε υπηρεσία τρέχει σε δικό της container με περιορισμένους πόρους και δικαιώματα, μειώνοντας τον αντίκτυπο ενδεχόμενης παραβίασης (αν και πρέπει να λαμβάνονται μέτρα όπως image scanning, least privilege, κ.λπ.).

- **Επεκτασιμότητα**: Μία από τις θεμελιώδεις αρχές του cloud computing είναι η δυνατότητα εύκολης κλιμάκωσης (scaling) των υπηρεσιών, τόσο οριζόντια (νέοι κόμβοι/instances) όσο και κάθετα (περισσότεροι πόροι ανά instance). Στο on-premise περιβάλλον μας, πετυχαίνουμε επεκτασιμότητα αξιοποιώντας τις ίδιες τεχνικές: το Kubernetes επιτρέπει την οριζόντια αύξηση των pods για μια υπηρεσία (π.χ. από 2 replicas σε 4) μέσα σε δευτερόλεπτα, ενώ υποστηρίζει και τη σχετικά εύκολη προσθήκη νέων worker nodes στο cluster όταν χρειάζεται επιπλέον υπολογιστική ισχύς. Το MetalLB και ο Ingress εξασφαλίζουν ότι ακόμη και αν προσθέσουμε νέες υπηρεσίες ή replicas, η πρόσβαση σε αυτές θα συνεχίσει να λειτουργεί ομαλά (διανέμοντας το φορτίο). Η MySQL replication παρέχει επεκτασιμότητα στις αναγνώσεις – μπορούμε να προσθέσουμε και άλλους slave servers εάν οι ανάγκες ανάγνωσης αυξηθούν, χωρίς να επιβαρύνουμε τον master. Το network slicing/SDN μπορεί να συνεισφέρει σε επεκτασιμότητα δικτύου: π.χ. αν το cluster μεγαλώσει και χρειαστούν ξεχωριστά "κομμάτια" δικτύου για διαφορετικές ομάδες εφαρμογών ή περιβάλλοντα (dev/staging/prod) πάνω στην ίδια φυσική υποδομή, αυτό μπορεί να επιτευχθεί λογισμικά.

Ένα σημαντικό σημείο είναι ότι όλα τα παραπάνω πραγματοποιούνται **σε τοπική υποδομή**. Σε περιβάλλον cloud, ορισμένες από αυτές τις λειτουργίες παρέχονται αυτοματοποιημένα από τον πάροχο (π.χ. managed load balancers, cloud DNS, VPN gateways κλπ.). Στο on-premise cluster, χρειάστηκε να τις υλοποιήσουμε με open-source εργαλεία: το MetalLB αντί για έναν cloud LB, το dnsproxy/CoreDNS αντί για έναν managed DNS service, το Tailscale αντί για πιθανές cloud VPN λύσεις, δικούς μας Ingress controllers και πιστοποιητικά αντί για managed ingress (όπως AWS ELB/ALB με ACM certs), και φυσικά δικό μας monitoring αντί για υπηρεσίες τύπου CloudWatch. Αυτό προσθέτει στον αρχικό σχεδιασμό και προσπάθεια, αλλά δίνει και **πλήρη έλεγχο**: η υποδομή είναι απομονωμένη, τα δεδομένα δεν φεύγουν σε τρίτους, και η λύση μπορεί να προσαρμοστεί ακριβώς στις ανάγκες μας χωρίς εξάρτηση από internet connectivity ή συνδρομές.

Συμπερασματικά, η συνένωση όλων αυτών των τεχνολογιών δημιούργησε ένα **ενοποιημένο, ισχυρό on-premise περιβάλλον** που προσεγγίζει τις δυνατότητες του cloud

σε επίπεδο αυτοματοποίησης, παρακολούθησης, αξιοπιστίας και κλιμάκωσης, ενώ παράλληλα διατηρεί τα πλεονεκτήματα του τοπικού ελέγχου. Αυτή η αρχιτεκτονική αποδεικνύει πώς οι αρχές του cloud native μπορούν να εφαρμοστούν και εκτός cloud, αξιοποιώντας τα εργαλεία ανοικτού κώδικα και το software-defined παράδειγμα για κάθε πτυχή της υποδομής. Οι τεχνολογίες συνεργάζονται αρμονικά – π.χ. το Kubernetes παρέχει τον «σκελετό» για workloads, το Docker την συσκευασία, το SDN τη δικτυακή ευελιξία, το Prometheus/Grafana την ορατότητα, το Tailscale την ασφαλή συνδεσιμότητα, κ.ο.κ. – προσφέροντας τελικά ένα περιβάλλον που είναι **αυτόματο, ορατό, ασφαλές και επεκτάσιμο** σύμφωνα με τους στόχους της σύγχρονης διαχείρισης υποδομών.

Κεφάλαιο 3 – Σχεδιασμός Συστήματος

3.1 Ανάλυση Απαιτήσεων

Για την υλοποίηση ενός τοπικού cluster Kubernetes που να εξυπηρετεί μια web εφαρμογή συνομιλίας (chat app) με υψηλή διαθεσιμότητα και επεκτασιμότητα, προσδιορίστηκαν οι ακόλουθες απαιτήσεις:

- **Κλιμάκωση φορτίου** (Scalability): Το σύστημα πρέπει να προσαρμόζεται σε αυξομειώσεις του φόρτου εργασίας χωρίς ανθρώπινη επέμβαση. Συγκεκριμένα, όταν η χρήση CPU των pods της web εφαρμογής υπερβαίνει ένα προκαθορισμένο όριο (40%), θα πρέπει να προστίθενται αυτόματα επιπλέον pods ώστε να διαμοιράζεται το φορτίο. Η οριζόντια αυτόματη κλιμάκωση (Horizontal Pod Autoscaler – HPA) αναλαμβάνει να αυξάνει ή να μειώνει τον αριθμό των pods δυναμικά, μεταξύ ενός ελάχιστου (π.χ. 2 pods) και ενός μέγιστου (5 pods) ορίου, διατηρώντας την αξιοποίηση πόρων σε αποδεκτά επίπεδα. Έτσι, σε περιόδους αιχμής η εφαρμογή θα παρέχει απόκριση και σταθερότητα, ενώ σε χαμηλό φόρτο θα ελευθερώνονται πόροι.
- **Διαθεσιμότητα και Αντοχή σε Αστοχίες**: Η αρχιτεκτονική πρέπει να ανέχεται αποτυχίες μεμονωμένων containers ή pods χωρίς διακοπή της υπηρεσίας. Η web εφαρμογή θα τρέχει σε τουλάχιστον 2 αντίγραφα (replicas) ταυτόχρονα, ώστε αν κάποιο pod πέσει, τα υπόλοιπα να συνεχίσουν να εξυπηρετούν αιτήματα. Παρομοίως, η βάση δεδομένων θα υλοποιηθεί σε διάταξη master/slave (πρωτεύον–αντίγραφο) με δύο pods MySQL, έτσι ώστε τα δεδομένα να αντιγράφονται ασύγχρονα στο replica pod. Αυτό βελτιώνει την ανθεκτικότητα: σε περίπτωση πτώσης του πρωτεύοντος MySQL pod, το αντίγραφο διαθέτει πρόσφατα δεδομένα (έστω και με μικρή καθυστέρηση). Γενικά, το Kubernetes θα πρέπει να εξασφαλίζει αυτόματη επανόρθωση (self-healing) των pods, επανεκκινώντας ή αντικαθιστώντας τα όταν αυτά τερματίζονται απροσδόκητα.
- **Ασφάλεια & Πρόσβαση**: Όλη η εξωτερική πρόσβαση στις υπηρεσίες του cluster πρέπει να γίνεται με ασφαλή τρόπο. Για τον σκοπό αυτό, απαιτείται η χρήση HTTPS για το web περιβάλλον (π.χ. https://web.local) καθώς και για τα εργαλεία διαχείρισης/παρακολούθησης (Grafana, Prometheus) μέσω του Ingress Controller. Θα χρησιμοποιηθούν αυτο-υπογεγραμμένα πιστοποιητικά TLS για να κρυπτογραφείται η

κίνηση προς τα domain *.local. Παράλληλα, δεδομένου ότι το cluster θα τρέχει σε δίκτυο πίσω από Carrier-Grade NAT (CGNAT) χωρίς δημόσια IPv4, απαιτείται μια λύση απομακρυσμένης πρόσβασης. Το Tailscale VPN αποτελεί επιλογή που ικανοποιεί αυτή την ανάγκη, δημιουργώντας ένα ασφαλές ιδιωτικό δίκτυο (mesh VPN) μεταξύ του cluster και των πελατών/διαχειριστών. Ο Tailscale client είναι εγκατεστημένος στον host (Windows 10), ο οποίος έχει ρυθμιστεί ως exit node, επιτρέποντας στα υπόλοιπα pods και containers στο cluster να αποκτήσουν πρόσβαση προς τον έξω κόσμο, κάτι που είναι απαραίτητο για την ομαλή λειτουργία του CoreDNS και του dnsproxy που βρίσκονται εκτός του cluster. Μέσω του Tailscale, οι διαχειριστές μπορούν να συνδεθούν εξ αποστάσεως στο cluster (π.χ. να φορτώσουν τις σελίδες Grafana/Prometheus ή την web εφαρμογή) σαν να βρίσκονταν στο τοπικό δίκτυο, παρακάμπτοντας τους περιορισμούς του CGNAT.

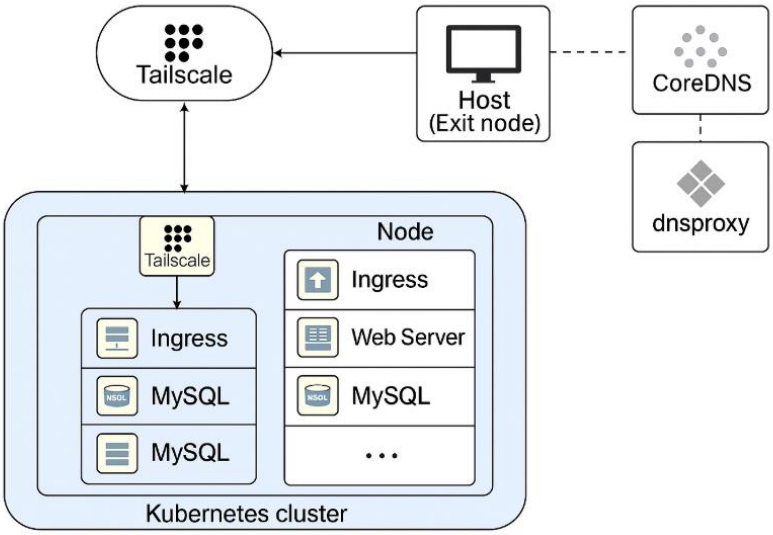
- **Ονοματολογία & DNS**: Για την εύκολη πρόσβαση των υπηρεσιών απαιτείται η χρήση φιλικών ονομάτων domain αντί απλών IP. Τα ονόματα web.local, grafana.local και prometheus.local θα χρησιμοποιηθούν για το web frontend, το περιβάλλον γραφημάτων Grafana και την κονσόλα παρακολούθησης Prometheus αντίστοιχα. Επειδή αυτά τα domains δεν είναι δημόσια, θα πρέπει να γίνεται τοπική επίλυσή τους σε διευθύνσεις IP του cluster. Το σύστημα DNS έχει ρυθμιστεί με dnsproxy και CoreDNS, οι οποίοι όμως τρέχουν εκτός του cluster, σε container στον host. Ο Tailscale έχει ρυθμιστεί να κατευθύνει τα αιτήματα DNS για την περιοχή .local προς αυτόν τον εξωτερικό resolver. Το CoreDNS είναι παραμετροποιημένο με στατικές εγγραφές για τα web.local, grafana.local και prometheus.local ώστε να επιστρέφει την IP του Ingress (192.168.1.240). Με αυτό τον τρόπο διασφαλίζεται ενιαία επίλυση DNS τόσο για τοπικούς όσο και για απομακρυσμένους πελάτες.
- **Παρακολούθηση & Ορατότητα (Observability):** Απαιτείται η συγκέντρωση μετρικών (CPU, μνήμη, δικτύου, αριθμού αιτήσεων, κτλ.) από όλα τα pods και στοιχεία του cluster, καθώς και η δυνατότητα οπτικοποίησης τους. Η στοίβα **Prometheus & Grafana** θα καλύψει αυτήν την ανάγκη. Ο Prometheus θα συλλέγει μετρήσεις τόσο από τα pods της εφαρμογής (μέσω ενσωματωμένων endpoints ή του Kubernetes metrics server) όσο και από την υποδομή (π.χ. usage κόμβου, κατάσταση pods), ενώ το Grafana θα παρέχει

ένα φιλικό περιβάλλον για διαγράμματα, πίνακες και συναγερμούς βασισμένους σε αυτά τα δεδομένα. Αυτή η δυνατότητα παρακολούθησης είναι κρίσιμη, ιδίως για να αξιολογηθεί η συμπεριφορά του HPA κατά το stress-testing (π.χ. να επιβεβαιωθεί ότι πράγματι αυξάνονται τα pods όταν η CPU > 40% και μειώνονται όταν ο φόρτος πέσει) και να εντοπιστούν πιθανά bottlenecks ή ανωμαλίες στο σύστημα. Επιπλέον, η παρακολούθηση επιτρέπει και στη **διαχείριση συμβάντων** – π.χ. θα διαπιστώνεται αμέσως αν κάποιο pod πέσει ή αν η βάση δεδομένων καθυστερεί να συγχρονιστεί.

- **Υποδομή & Πόροι:** Το cluster θα υλοποιηθεί σε έναν μόνο φυσικό μηχανήματα/κόμβο, χρησιμοποιώντας ένα Windows 10 host με Docker Desktop. Ο διαθέσιμος υλικός εξοπλισμός περιλαμβάνει **επεξεργαστή AMD Ryzen 5 3600XT** (6 πυρήνες / 12 νήματα) και **μνήμη 16GB RAM**. Αυτό το επίπεδο πόρων θεωρείται επαρκές για έναν single-node cluster ανάπτυξης με πολλαπλά pods και υπηρεσίες. Ο δίσκος 1TB παρέχει άφθονο χώρο για εικόνες container και persistent data (π.χ. volumes MySQL, δεδομένα Grafana, κτλ.). Ένα μέρος των απαιτήσεων είναι η βέλτιστη αξιοποίηση αυτών των πόρων: το cluster πρέπει να λειτουργεί ομαλά εντός των ορίων του μηχανήματος, χωρίς να προκαλεί υπερφόρτωση του host. Για παράδειγμα, μέσω του HPA θα αποφεύγεται η άσκοπη κατανάλωση CPU/RAM από υπεράριθμα pods σε χαμηλό φόρτο, ενώ η χρήση ελαφρών λύσεων όπως το MetalLB (αντί βαρύτερων load balancers) και το CoreDNS εξασφαλίζει μικρό αποτύπωμα.

3.2 Αρχιτεκτονική Συστήματος

Η συνολική αρχιτεκτονική του συστήματος απεικονίζεται στο Σχήμα 3.1. Πρόκειται για ένα ενιαίο cluster Kubernetes με έναν κόμβο (single-node) που τρέχει επάνω σε Windows 10 μέσω Docker Desktop. Όλα τα υποσυστήματα – web server, βάση δεδομένων, ingress controller, load balancer, monitoring, DNS – φιλοξενούνται ως containers/pods μέσα σε αυτό το cluster. Παρακάτω περιγράφονται αναλυτικά τα κύρια δομικά στοιχεία της αρχιτεκτονικής:



Σχήμα 3.1: Αρχιτεκτονική του τοπικού Kubernetes cluster και των υποσυστημάτων του.

Ο μοναδικός κόμβος (VM Docker Desktop) φιλοξενεί όλα τα pods: δύο pods του web server πίσω από Service, δύο pods MySQL (master και replica), Ingress Controller (τύπου LoadBalancer service με IP από το MetalLB), σύστημα παρακολούθησης (Prometheus, Grafana) και Tailscale VPN server. Εκτός του cluster, ένας coreDNS και ένας DNS proxy αναλαμβάνει την επίλυση των τοπικών ονομάτων σε συνεργασία με το CoreDNS, ενώ το Tailscale VPN επιτρέπει σε απομακρυσμένους χρήστες να συνδεθούν στο περιβάλλον χρησιμοποιώντας τα ίδια ονόματα (με ρυθμίσεις Split DNS).

3.2.1 Κόμβος Kubernetes και Υποδομή Cluster

Ο Kubernetes κόμβος υλοποιείται στον host Windows 10 μέσω Docker Desktop. Ο Docker Desktop παρέχει έναν ενσωματωμένο single-node Kubernetes cluster αλλά εναλλακτικά χρησιμοποιήθηκε το kubeadm σε συνδυασμό με WSL2. Στην πράξη, πρόκειται για ένα εικονικοποιημένο περιβάλλον (VM) Linux μέσα από το οποίο τρέχουν τα pods και τα απαραίτητα Kubernetes components (API server, scheduler, controller manager, etcd) όλα στον ίδιο κόμβο. Έχει γίνει άρση του taint του master ώστε ο κόμβος (που είναι ταυτόχρονα control-plane) να δέχεται και pods εφαρμογών, δεδομένου ότι δεν υπάρχουν worker nodes.

Υλικό (Host): Ο φυσικός υπολογιστής λειτουργεί ως βάση για το cluster. Ο εξαπύρηνος επεξεργαστής και η μνήμη 16GB εξασφαλίζουν ότι μπορεί να τρέξει το overhead του Kubernetes μαζί με πολλαπλά containers. Για παράδειγμα, κάθε pod MySQL θα δεσμεύει

μερικές εκατοντάδες MB μνήμης, ο Prometheus αρκετά για τη βάση δεδομένων μετρικών, ενώ το Grafana λιγότερα. Ο web server (chat app) pods πιθανώς θα είναι ελαφρύς (π.χ. με nginx + PHP) αλλά σε peak load μπορεί να αυξήσει την κατανάλωση CPU, κάτι που δικαιολογεί και το σχετικά χαμηλό threshold 40% για το scaling. Ο δίσκος 1TB χρησιμοποιείται τόσο για την αποθήκευση των docker images όσο και για volumes των stateful pods: η MySQL θα χρησιμοποιήσει PersistentVolume (π.χ. μέσω hostPath ή CSI driver του Docker Desktop) ώστε τα δεδομένα της να διατηρούνται αν το pod επανεκκινηθεί, ενώ και ο Prometheus θα γράφει τα δεδομένα του σε όγκο για persistence.

Δικτύωση Cluster: Το δίκτυο του cluster είναι το εσωτερικό δίκτυο του Docker Desktop VM. Κάθε Service τύπου ClusterIP παίρνει μια εικονική IP (Service Cluster IP) προσβάσιμη εντός του cluster, ενώ τα pods επικοινωνούν μέσω των δικών τους IP (pod CIDR) και της υπηρεσίας DNS (CoreDNS) για name resolution μεταξύ τους. Για να εκθέσουμε υπηρεσίες προς τα έξω (εκτός cluster), χρησιμοποιείται είτε NodePort/LoadBalancer service είτε Ingress. Εδώ υιοθετήθηκε η λύση Ingress, που είναι πιο ευέλικτη για πολλαπλά services πίσω από έναν ενιαίο frontend, σε συνδυασμό με το MetalLB ώστε το Ingress να αποκτήσει πραγματική IP.

3.2.2 Εφαρμογή Web & Αυτόματη Κλιμάκωση

Η web εφαρμογή συνομιλίας αναπτύσσεται ως **Deployment** στο Kubernetes, με αρχικό μέγεθος 2 pods (replicas). Τα pods αυτά είναι ομοιόμορφα instances της ίδιας εφαρμογής (stateless microservice), οπότε μπορούν να δεχτούν ανεξάρτητα εισερχόμενα αιτήματα. Μπροστά από τα pods υπάρχει ένα **Service** (τύπου ClusterIP) που τα ομαδοποιεί και παρέχει ένα σταθερό σημείο πρόσβασης (DNS όνομα και virtual IP) εντός του cluster. Το Service αυτό χρησιμοποιείται τόσο από το Ingress controller για την προώθηση εξωτερικών αιτημάτων, όσο και από άλλα pods αν επικοινωνούν εσωτερικά με το web service.

Για να επιτευχθεί αυτόματη κλιμάκωση, έχει οριστεί ένας **Horizontal Pod Autoscaler (HPA)** στόχος για το Deployment της εφαρμογής. Οι παράμετροι του HPA είναι: ελάχιστα pods = 2, μέγιστα pods = 5, target CPU utilization = 30%. Αυτό σημαίνει ότι ο HPA controller (που τρέχει ως μέρος του control plane) θα ελέγχει περιοδικά (ανά ~15 δευτερόλεπτα) τους μέσους δείκτες CPU των pods και, εάν ο μέσος όρος ξεπερνά το 30% της αιτούμενης CPU, θα αυξάνει τον αριθμό των pods. Αντίστροφα, αν η ζήτηση πέσει και τα pods έχουν χαμηλή

χρήση CPU, ο HPA θα μειώσει σταδιακά τα replicas προς το ελάχιστο. Αυτή η ικανότητα **αυτόματης οριζόντιας κλιμάκωσης** είναι ουσιώδης για να ανταποκρίνεται η εφαρμογή σε spikes φορτίου χωρίς μη αυτόματη παρέμβαση, και αποτελεί βασική απαίτηση που τέθηκε. Όπως αναφέρει η βιβλιογραφία, το HPA διασφαλίζει ότι "αν η χρήση CPU ξεπεράσει ένα όριο (π.χ. 40%), προστίθενται περισσότερα pods για να κατανεμηθεί το φορτίο, ενώ όταν η ζήτηση πέσει στο φυσιολογικό, ο αριθμός των pods μειώνεται και πάλι).

Για την υλοποίηση του HPA χρειάζεται ο Kubernetes Metrics Server, ο οποίος είναι εγκατεστημένος στο cluster (ο Docker Desktop Kubernetes συνήθως τον περιλαμβάνει). Ο Metrics Server συλλέγει μετρικές χρήσης πόρων από τα kubelets και τις καθιστά διαθέσιμες μέσω του Metrics API. Ο HPA controller χρησιμοποιεί αυτές τις μετρικές (π.χ. cpu_utilization των pods του deployment) για να αποφασίσει πώς να προσαρμόσει τα replicas. Σημειώνεται ότι το όριο 40% αφορά το ποσοστό της αιτούμενης CPU (requests) που χρησιμοποιείται – στη συγκεκριμένη περίπτωση, τα pods πιθανώς να έχουν οριστεί με ένα CPU request (ή κάποιο μερίδιο), οπότε 40% σημαίνει ότι κατά μέσο όρο χρησιμοποιούν 0.4 CPU. Το σχετικά χαμηλό αυτό threshold επελέγη ώστε να δοκιμαστεί το scaling ακόμη και σε μέτριο φορτίο (καθώς μια web εφαρμογή συνομιλίας μπορεί να μην είναι πολύ βαριά σε CPU, αλλά θέλουμε να επιδείξουμε την αυτόματη κλιμάκωση ακόμη και με μικρό πολλαπλασιασμό χρηστών).

3.2.3 Βάση Δεδομένων MySQL (Master/Replica)

Το σύστημα αποθήκευσης δεδομένων υλοποιείται με **MySQL** σε διάταξη **master-slave** (ή πρωτεύον-αντίγραφο). Αυτό πραγματοποιείται στο Kubernetes μέσω ενός **StatefulSet** που δημιουργεί δύο pods MySQL: το ένα ενεργεί ως *master* (πρωτεύουσα βάση) και το δεύτερο ως *slave* (αντίγραφο) που συγχρονίζεται ασύγχρονα. Η χρήση StatefulSet (αντί απλού Deployment) επιτρέπει τον σταθερό προσδιορισμό των pods (π.χ. mysql-primary-0 ως master, mysql-secondary-0 ως replica) και τη διατήρηση persistent volume δεδομένων ανά pod. Σύμφωνα με τις οδηγίες της Kubernetes, μια τέτοια τοπολογία περιλαμβάνει "έναν μόνο primary server και ένα ή περισσότερα replicas, με ασύγχρονη αναπαραγωγή σειριακών log". Στην περίπτωσή μας χρησιμοποιούνται δύο pods, οπότε έχουμε έναν primary και ένα replica.

Η επικοινωνία master-slave γίνεται μέσω του μηχανισμού **binlog replication** της MySQL. Το master pod έχει ενεργοποιημένο το binary logging και δικαίωμα να δέχεται συνδέσεις replica, ενώ το replica pod έχει ρυθμιστεί σε read-only λειτουργία (super-read-only) ώστε να μην δέχεται τοπικές εγγραφές παρά μόνο μέσω του replication stream από το master. Η αρχική ρύθμιση (π.χ. ορισμός του master host, credentials για replication user, κ.λπ.) μπορεί να γίνεται μέσω init containers ή scripts κατά την δημιουργία των pods. Επίσης, δύο Services συνήθως ορίζονται: ένα ClusterIP service για το master (ώστε οι εφαρμογές να συνδέονται στη MySQL primary μέσω σταθερού hostname, π.χ. mysql-primary.default.svc.cluster.local), και ένα ξεχωριστό Service για το replica (αν χρειάζεται πρόσβαση π.χ. για read-only queries ή monitoring).

Πλεονεκτήματα διάταξης master/replica: Παρότι σε ένα single-node cluster δεν μπορούμε να επιτύχουμε υψηλή διαθεσιμότητα απέναντι σε αστοχία του ίδιου του μηχανήματος, η ύπαρξη replica προσφέρει οφέλη. Πρώτον, λειτουργεί ως **μηχανισμός backup** – σε περίπτωση λογικού σφάλματος στο master (π.χ. διαγραφή δεδομένων κατά λάθος), το replica θα μπορούσε (θεωρητικά) να διακοπεί πριν εφαρμόσει τις αλλαγές. Δεύτερον, δίνει τη δυνατότητα **κατανομής φορτίου ανάγνωσης**: αν υπήρχαν πολλές αναγνωστικές αιτήσεις, θα μπορούσαν να εξυπηρετηθούν και από το replica (αν και στην εφαρμογή μας πιθανότατα όλες οι λειτουργίες πάνε στο master). Τρίτον, μας επιτρέπει να παρατηρήσουμε τη συμπεριφορά ενός stateful συστήματος στο Kubernetes – π.χ. πώς γίνονται restart τα pods με διατήρηση της ταυτότητάς τους, πώς επανέρχεται ο συγχρονισμός μετά από διακοπή, κ.λπ.

Περιορισμοί: Σημειώνεται ότι σε αυτή τη βασική υλοποίηση δεν υπάρχει αυτοματοποιημένο failover. Εάν το master pod αποτύχει, το cluster θα προσπαθήσει να το επανεκκινήσει (διατηρώντας τον ίδιο persistent volume). Δεν γίνεται αυτόματη προαγωγή του replica σε νέο master – κάτι τέτοιο θα απαιτούσε πρόσθετη λογική ή operator. Συνεπώς, το σενάριο απώλειας του master αντιμετωπίζεται με downtime ίσο με τον χρόνο επανεκκίνησης του pod. Το replica συνεχίζει να λειτουργεί αλλά σε read-only κατάσταση (θα μπορούσε προσωρινά να χρησιμοποιηθεί για ανάγνωση, αλλά η εφαρμογή μας λογικά δεν το υποστηρίζει). Παρ' όλα αυτά, σε περίπτωση που το master επανέλθει, ο replica

μηχανισμός θα **επανασυγχρονίσει** τυχόν χαμένες εγγραφές (αν για παράδειγμα έμεινε λίγο πίσω πριν το crash, θα εφαρμόσει τα υπόλοιπα binlog entries). Το γεγονός ότι πρόκειται για τοπικό μη-critical περιβάλλον επιτρέπει αυτή την απλούστευση.

3.2.4 Δρομολόγηση Εισόδου: Nginx Ingress Controller & MetalLB

Για την έκθεση των υπηρεσιών (web εφαρμογή, Grafana, Prometheus) προς τον έξω κόσμο, χρησιμοποιείται ένα **Ingress**. Συγκεκριμένα, έχει αναπτυχθεί στο cluster ο **NGINX Ingress Controller**, που είναι μια υλοποίηση Ingress controller βασισμένη στο δημοφιλές web server/reverse proxy Nginx. Ο Ingress controller τρέχει ως Deployment (συνήθως με 1 replica) σε ένα δικό του namespace (π.χ. ingress-nginx). Περιλαμβάνει έναν Nginx server που παραμετροποιείται δυναμικά μέσω των Ingress resources του Kubernetes. Σε κάθε Ingress resource μπορούν να οριστούν κανόνες δρομολόγησης: για παράδειγμα, αιτήματα προς host web.local να προωθούνται στο Service της web εφαρμογής, αιτήματα προς grafana.local να προωθούνται στο Service του Grafana, κ.ο.κ. Επιπλέον, υποστηρίζει **τερματισμό SSL**: φορτώνουμε ένα TLS πιστοποιητικό (εν προκειμένω self-signed) και ο Nginx θα αποκρυπτογραφεί την HTTPS κίνηση, εξυπηρετώντας τα περιεχόμενα μέσω HTTP προς τα εσωτερικά services. Ο Ingress έτσι λειτουργεί ως **αντιστροφος διαμεσολαβητής (reverse proxy)** με δυνατότητες load balancing και SSL termination, εκθέτοντας τις εσωτερικές υπηρεσίες του cluster προς τα έξω.

Σε περιβάλλον cloud, μια Ingress συνήθως συνοδεύεται από έναν cloud load balancer που δίνει δημόσια IP. Σε on-premise περιβάλλοντα (όπως το δικό μας τοπικό cluster), δεν υπάρχει αυτή η πολυτέλεια. Εδώ αξιοποιούμε το **MetalLB**, ένα λογισμικό load-balancer που λειτουργεί εντός του cluster. Το MetalLB μπορεί να εκχωρήσει **στατικές IP** από ένα προκαθορισμένο εύρος (pool) σε οποιαδήποτε Kubernetes Service τύπου LoadBalancer. Στην περίπτωση μας, ο Nginx Ingress Controller δημιουργεί ένα Service LoadBalancer (π.χ. ingress-nginx-controller) και, χάρη στο MetalLB, αντί να μείνει χωρίς External IP, του αποδίδεται μια IP από το εύρος που ορίσαμε (π.χ. 192.168.1.240). Το MetalLB λειτουργεί σε λειτουργία layer2: ουσιαστικά **αναγγέλλει** (με ARP) στο τοπικό δίκτυο ότι η IP αυτή αντιστοιχίζεται στον κόμβο του cluster. Εφόσον το cluster είναι μοναδικός κόμβος, το

MetalLB Speaker (daemon) τρέχει εκεί και απαντά στα ARP requests για την συγκεκριμένη IP, δρομολογώντας την κίνηση στον ingress controller.

Με αυτόν τον τρόπο, οποιοσδήποτε πελάτης εντός του ίδιου δικτύου (ή συνδεδεμένος μέσω VPN) μπορεί να προσπελάσει τις υπηρεσίες χρησιμοποιώντας την IP του ingress. Στη δική μας υλοποίηση, όμως, οι πελάτες δεν θα χρειαστεί να θυμούνται την IP – καθώς ρυθμίσαμε DNS ονόματα web.local, grafana.local, κ.λπ. που θα επιλύονται σε αυτή την IP. Ο Ingress controller, ανάλογα με το Host header του αιτήματος HTTPS, κατευθύνει το traffic: π.χ. **https://web.local → Service web-app**, **https://grafana.local → Service grafana**, **https://prometheus.local → Service prometheus**. Έτσι επιτυγχάνεται ένα **ενιαίο σημείο εισόδου** (single frontend IP/port) για πολλαπλές υπηρεσίες.

Ασφαλές πιστοποιητικό: Έχει δημιουργηθεί ένα self-signed πιστοποιητικό wildcard για *.local ή ξεχωριστά πιστοποιητικά για κάθε domain, τα οποία αποθηκεύτηκαν ως Kubernetes Secret. Το Ingress resource αναφέρεται σε αυτό το TLS Secret, ώστε ο Nginx να το φορτώσει. Καθώς είναι αυτο-υπογεγραμμένα, οι browsers θα εμφανίζουν προειδοποίηση μη έμπιστου πιστοποιητικού, όμως για λόγους δοκιμών αυτό είναι αποδεκτό. Θα μπορούσε εναλλακτικά να δημιουργηθεί ένα τοπικό Certificate Authority και οι clients να εμπιστεύονται το root CA (έτσι ώστε να μην εμφανίζεται προειδοποίηση), αλλά αυτό ξεφεύγει από το scope της εργασίας. Σημαντικό είναι ότι η κίνηση είναι **κρυπτογραφημένη** και εντός του τοπικού δικτύου/VPN, προσφέροντας εμπιστευτικότητα.

3.2.5 Παρακολούθηση: Prometheus & Grafana

Για την επίτευξη ορατότητας στο σύστημα, ενσωματώθηκε η δημοφιλής λύση **Prometheus** για συλλογή μετρικών και η **Grafana** για παρουσίασή τους. Ο Prometheus τρέχει ως Deployment (ένα pod) στο cluster, εκτελώντας τον Prometheus server ο οποίος περιοδικά πραγματοποιεί **scraping** (HTTP GET σε endpoints /metrics) από τους στόχους που του καθορίζονται. Στο cluster μας, οι κύριοι στόχοι είναι: οι κόμβοι/containers της εφαρμογής (web pods), η MySQL (αν τρέχει exporter), καθώς και τα ίδια τα Kubernetes components. Συγκεκριμένα, με την ανάπτυξη του Prometheus αξιοποιήθηκε το **Prometheus** με Helm για να προστίθενται αυτόματα targets μέσω service discovery στο cluster (π.χ. όλα τα pods με

συγκεκριμένα annotations κ.λπ.). Για παράδειγμα, ο kubelet ή ο Metrics Server μπορεί να εκθέτουν metrics για CPU/Mem pods, οπότε ο Prometheus τα συλλέγει.

Τα δεδομένα αυτά αποθηκεύονται στο **χρονικά σειριακό αποθετήριο** (time-series database) που ενσωματώνει ο Prometheus. Από πλευράς πόρων, του έχει δοθεί persistent volume για να διατηρεί τα δεδομένα μετρικών (έστω κι αν σε αυτή την εργασία δεν είναι κρίσιμη η μακροχρόνια διατήρηση, είναι καλό να μην χάνονται σε κάθε restart).

Το Grafana, από την άλλη, είναι εγκατεστημένο ως ξεχωριστό Deployment (ένα pod με έναν container που τρέχει τον Grafana server). Αφού εκκινήσει, έχει διαμορφωθεί μια **Data Source** που δείχνει προς τον Prometheus (συνήθως μέσω του ClusterIP Service του Prometheus). Έτσι, το Grafana μπορεί να κάνει queries (με γλώσσα PromQL) στα δεδομένα του Prometheus και να απεικονίζει τα αποτελέσματα σε διαγράμματα. Έχουν δημιουργηθεί πίνακες ελέγχου (dashboards) για κρίσιμες πληροφορίες: χρήση CPU/μνήμης ανά pod, αριθμός ενεργών pods της εφαρμογής (για να παρατηρείται το scaling), latency αιτήσεων αν μετράται, ρυθμός εισερχόμενων αιτήσεων HTTP, κατάσταση βάσης δεδομένων, κ.λπ. Αυτές οι πληροφορίες **βοηθούν στην αξιολόγηση του συστήματος** – για παράδειγμα, μπορούμε να δούμε σε πραγματικό χρόνο στο Grafana την απόκριση του HPA όταν τρέχει το stress test, ή να εντοπίσουμε αν η MySQL replica καθυστερεί να συγχρονίσει (π.χ. metric replication lag).

Επιπλέον, ο Prometheus μπορεί να λειτουργήσει και ως **πηγή autoscaling** (custom metrics) αλλά στη δική μας περίπτωση αυτό δεν χρησιμοποιήθηκε – βασιστήκαμε στον Metrics Server για το HPA. Ωστόσο, ο Prometheus θα μπορούσε να αξιοποιηθεί για πιο προηγμένα σενάρια autoscaling (με Kubernetes Adapter για custom metrics) ή για alerts (Alertmanager). Σημειωτέον, η Grafana και ο Prometheus εκτίθενται και αυτοί μέσω του Ingress (στα domains grafana.local και prometheus.local αντίστοιχα), οπότε είναι προσβάσιμοι μέσω HTTPS για ευκολία και ασφάλεια.

Οι πλατφόρμες παρακολούθησης αυτές θεωρούνται βέλτιστες πρακτικές σε Kubernetes περιβάλλον, καθώς “εργαλεία όπως ο Prometheus (που συγκεντρώνει μετρικές containers) και το Grafana (που παρέχει πίνακες ελέγχου) βοηθούν στη συγκέντρωση και προβολή σημαντικών πληροφοριών”. Στο πλαίσιο της παρούσας εργασίας, επιβεβαιώνουν ότι το

σύστημα ικανοποιεί τις απαιτήσεις (π.χ. πράγματι γίνεται scale-out στους 5 pods υπό φορτίο, οι πόροι του κόμβου επαρκούν, κ.ο.κ.) αλλά και προσφέρουν εκπαιδευτική αξία εξοικείωσης με τα εργαλεία διαχείρισης.

3.2.6 Δίκτυο VPN και Τοπικός DNS

Tailscale VPN: Λόγω της δικτυακής απομόνωσης (CGNAT) του host, δεν υπάρχει η δυνατότητα απευθείας πρόσβασης από το Internet στις υπηρεσίες του cluster. Για να επιτραπεί σε απομακρυσμένους χρήστες (π.χ. τον επιβλέποντα ή για δοκιμές επίδειξης) να συνδεθούν στο περιβάλλον, χρησιμοποιείται το Tailscale VPN. Το Tailscale δημιουργεί ένα ιδιωτικό overlay δίκτυο πάνω από το δημόσιο Internet χρησιμοποιώντας το πρωτόκολλο WireGuard. Κάθε συμμετέχων στο tailnet αποκτά μία ιδιωτική IP διεύθυνση εντός του εύρους 100.x.y.z/8, και όλοι οι κόμβοι μπορούν να επικοινωνούν μεταξύ τους με ασφάλεια.

Στην παρούσα υλοποίηση, ο Tailscale server έχει εγκατασταθεί ως container pod εντός του Kubernetes cluster. Ο host (Windows 10) λειτουργεί ως Tailscale client, και έχει ρυθμιστεί ως exit node, ώστε να παρέχει έξοδο προς το διαδίκτυο στα υπόλοιπα pods του cluster μέσω του Tailscale tunnel. Αυτό είναι κρίσιμο ώστε το CoreDNS και το dnsproxy (που τρέχουν εκτός του cluster) να μπορούν να επιλύουν εξωτερικά domains ή να απαντούν σε αιτήματα από το VPN.

Οι απομακρυσμένοι χρήστες εγκαθιστούν τον Tailscale client και συνδέονται στο ίδιο tailnet. Η πρόσβαση σε διευθύνσεις όπως https://web.local γίνεται με διαφάνεια: τα DNS αιτήματα δρομολογούνται μέσω του Tailscale VPN προς τον host (που λειτουργεί ως DNS forwarder) και η κίνηση μεταφέρεται μέσω του exit node στο cluster. Χρησιμοποιείται το Split DNS του Tailscale, ώστε μόνο αιτήματα για domains .local να κατευθύνονται προς τον τοπικό resolver μας, ενώ τα υπόλοιπα domains συνεχίζουν να επιλύονται από τον προεπιλεγμένο DNS provider του client.

Τοπικός DNS (dnsproxy & CoreDNS): Για να επιτευχθεί σωστή επίλυση των τοπικών domains, έχει στηθεί ένας DNS resolver εκτός του cluster, ως container στον ίδιο τον host. Ο resolver αυτός βασίζεται στο εργαλείο dnsproxy, το οποίο δέχεται αιτήματα DNS για domains .local και τα προωθεί στον CoreDNS (ο οποίος επίσης τρέχει εκτός cluster, ως ανεξάρτητο container). Ο CoreDNS έχει παραμετροποιηθεί ώστε να επιστρέφει τις στατικές IP διευθύνσεις του Ingress Controller για τα web.local, grafana.local και prometheus.local.

Ο dnsproxy λειτουργεί ως gateway: λαμβάνει τα DNS queries από τους clients μέσω Tailscale, αναγνωρίζει τα .local suffixes και τα προωθεί στο CoreDNS. Ο CoreDNS απαντά με τις σωστές IP (π.χ. 192.168.1.240 για το Ingress). Ο dnsproxy στέλνει την απάντηση πίσω μέσω του VPN στον client. Για μη τοπικά domains, είτε απορρίπτει είτε τα προωθεί στον κανονικό DNS (π.χ. Cloudflare ή Tailscale DNS).

Με απλά λόγια:

- **Εντός cluster:** Τα pods δεν χρησιμοποιούν τον CoreDNS για .local domains, καθώς ο resolver βρίσκεται εκτός cluster. Ωστόσο, μέσω κατάλληλων ρυθμίσεων (π.χ. routes μέσω Tailscale exit node), μπορούν να στείλουν queries προς τον εξωτερικό CoreDNS resolver, αν χρειαστεί.
- **Εκτός cluster (τοπικά ή μέσω VPN):** Ο host, που τρέχει το dnsproxy, απαντά στα DNS queries των clients. Το Tailscale κατευθύνει τα αιτήματα .local στον dnsproxy. Αυτός προωθεί στο CoreDNS και επιστρέφει τις σωστές IPs. Έτσι, το https://web.local οδηγεί σωστά στον Ingress του cluster.

Αυτός ο διαχωρισμός resolver εντός και εκτός cluster εξυπηρετεί το σενάριο τοπικής (on-premise) υποδομής, όπου ο host έχει κεντρικό ρόλο στην διασύνδεση υπηρεσιών. Αντί να διαχειριστούν χειροκίνητα το /etc/hosts σε κάθε client, χρησιμοποιούμε ένα δυναμικό DNS setup με split DNS και ονοματολογία .local.

Συνοψίζοντας:

- **Ο Tailscale server** είναι pod εντός cluster.
- **Ο host είναι Tailscale client** και έχει οριστεί ως exit node.
- **Ο CoreDNS και dnsproxy** τρέχουν εκτός του cluster, ως containers στον host.
- **Το DNS resolving** των .local domains γίνεται από τους απομακρυσμένους χρήστες μέσω Tailscale DNS (split DNS), οδηγώντας τελικά στον dnsproxy και από εκεί στον CoreDNS.
- **Όλες οι υπηρεσίες** (web.local, grafana.local, prometheus.local) γίνονται πλήρως προσβάσιμες από απομακρυσμένους χρήστες, μέσω του VPN, χωρίς καμία αλλαγή στον τρόπο χρήσης (URLs, HTTPS, κλπ.).

3.3 Ροές Λειτουργίας του Συστήματος (Σενάρια)

Σε αυτή την ενότητα παρουσιάζονται διαγράμματα ροής και περιγραφές για δύο βασικά σενάρια λειτουργίας του συστήματος: (α) **Απότομη αύξηση φόρτου (spike)** και πώς αντιδρά το σύστημα μέσω του autoscaling, και (β) **Αστοχία pods** και πώς το cluster αυτοθεραπεύεται. Αυτά τα σενάρια αποτελούν ουσιαστικές δοκιμές για να επαληθευτεί ο σχεδιασμός.

3.3.1 Σενάριο Spike Φόρτου (Auto-Scaling Event)

Περιγραφή σεναρίου: Ένας μεγάλος αριθμός χρηστών εισέρχεται ταυτόχρονα στην εφαρμογή chat, στέλνοντας μηνύματα και αιτήματα HTTP. Αυτό προκαλεί απότομη αύξηση στον φόρτο εργασίας των pods της web εφαρμογής. Θα εξετάσουμε πώς το Kubernetes ανταποκρίνεται μέσω του HPA.

Ροή γεγονότων:

- ***Αρχική κατάσταση****:* Το σύστημα βρίσκεται σε ομαλή λειτουργία με 2 pods web εφαρμογής. Κάθε pod χρησιμοποιεί χαμηλό % της CPU (π.χ. 10-20%) και ο HPA διατηρείται στο ελάχιστο replica count (2 pods). Ο ingress δρομολογεί τα εισερχόμενα αιτήματα κυκλικά στα 2 pods (load balancing).
- ***Έναρξη φόρτου:*** Ξαφνικά, νέοι χρήστες αρχίζουν να στέλνουν μεγάλο αριθμό αιτημάτων (π.χ. μέσω ενός stress test εργαλείου όπως το **hey** με πολλαπλά concurrency). Κάθε pod web αρχίζει να επεξεργάζεται περισσότερα αιτήματα ανά δευτερόλεπτο. Η χρήση CPU στα pods ανεβαίνει γρήγορα πάνω από το 30%. Ας υποθέσουμε ότι φτάνει το 70% στον μέσο όρο των pods.
- ***Συλλογή μετρικών:*** Ο Kubernetes Metrics Server ή/και οι Prometheus exporters συλλέγουν τις μετρήσεις CPU. Ο Metrics Server ιδιαίτερα ενημερώνει το API Server με τις τρέχουσες τιμές utilization.
- ***Απόφαση HPA:*** Ο Horizontal Pod Autoscaler controller εκτελεί τον βρόχο ελέγχου του. Διαβάζει από το API τις μετρικές CPU για το deployment της web εφαρμογής. Διαπιστώνει ότι το **μέσο ποσοστό CPU** των pods είναι π.χ. 70%, πολύ πάνω από τον στόχο 30%. Σύμφωνα με τον αλγόριθμο του HPA, θα **υπολογίσει τον νέο επιθυμητό αριθμό pods**. Συνήθως ο τύπος είναι:

$$\text{νέα replicas} = \left\lceil \frac{\text{τρέχουσα χρήση}}{\text{στόχος χρήση}} \times \text{τρέχον replicas} \right\rceil$$

Σε αριθμούς: (70%/30%) * 2 = 5 ceiling → **5 pods**. Ο HPA αποφασίζει να αυξήσει τα pods σε 5.

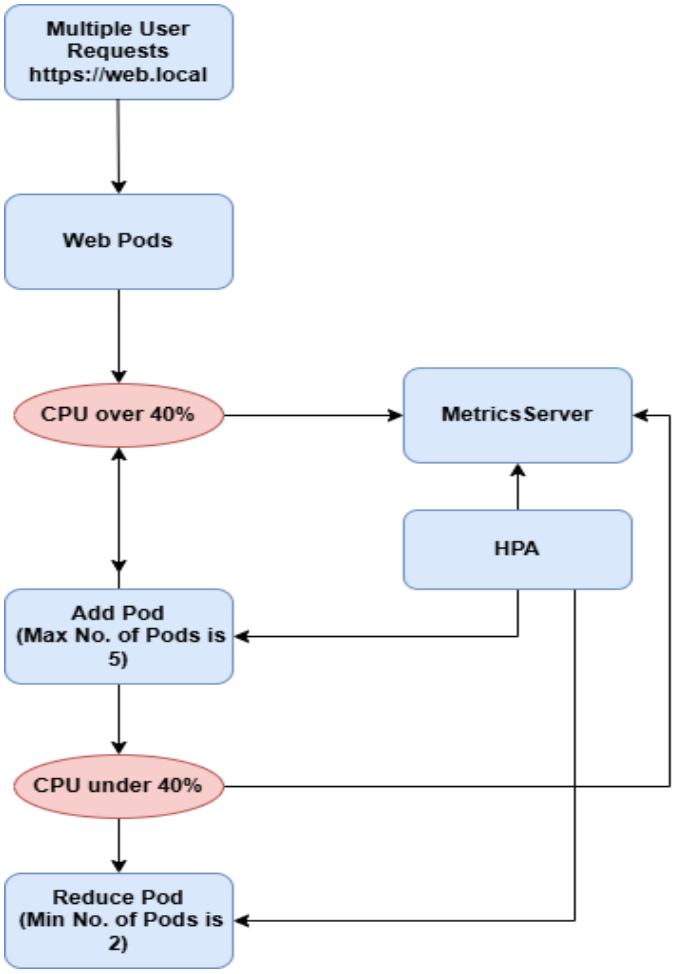
- ***Κλιμάκωση (scale-out):*** Ο Kubernetes API server ενημερώνει το Deployment ότι το .spec.replicas πρέπει να γίνει 5. Το Deployment controller αντιδρά δημιουργώντας 2 νέα ReplicaSet pods. Ο scheduler του Kubernetes προγραμματίζει και τα δύο νέα pods στον διαθέσιμο κόμβο (μοναδικός κόμβος). Τα images της εφαρμογής είναι ήδη στην cache, οπότε τα pods ξεκινούν σχετικά γρήγορα. Μέσα σε λίγα δευτερόλεπτα, τα νέα pods είναι σε κατάσταση Running και έτοιμα (Ready).
- ***Ενημέρωση υπηρεσίας:*** Το Service της web εφαρμογής ανιχνεύει αυτόματα τα νέα pods (μέσω labels selector) και τα προσθέτει στα endpoints του. Ο Nginx Ingress Controller, ο οποίος παρακολουθεί τα endpoints του Service (μέσω των Kubernetes APIs), ενημερώνει και αυτός το configuration του: πλέον έχει 5 upstream servers αντί για 2. Δεν απαιτείται επανεκκίνηση – η προσθαφαίρεση pods γίνεται δυναμικά.
- ***Διαχείριση αυξημένου φορτίου:*** Τώρα το εισερχόμενο φορτίο κατανέμεται σε 5 pods αντί για 2. Αυτό μειώνει το μέσο usage CPU ανά pod. Τα νέα pods αρχίζουν να εξυπηρετούν μέρος των αιτημάτων (ίσως μετά από μερικά readiness checks). Οι χρήστες πιθανώς δεν αντιλαμβάνονται κάποια σημαντική καθυστέρηση – μπορεί την πρώτη στιγμή η απόκριση να είχε αυξημένο latency λόγω υψηλής χρήσης, αλλά μετά το scale-out η καθυστέρηση μειώνεται καθώς κάθε pod χειρίζεται λιγότερες παράλληλες αιτήσεις.
- ***Σταθεροποίηση:*** Ο HPA βλέπει τώρα ότι η χρήση CPU πέφτει ξανά προς το 30-40% χάρη στα περισσότερα pods. Εφόσον η χρήση είναι πλέον εντός ορίου, δεν γίνονται περαιτέρω αλλαγές. Ο φόρτος μπορεί να παραμείνει υψηλός, αλλά το σύστημα ανταποκρίνεται ομαλά.
- ***Ορατότητα:*** Στο μεταξύ, ο Prometheus καταγράφει τον αριθμό των replicas (metric από το Kubernetes API) καθώς και τη χρήση CPU κάθε pod. Το Grafana μπορεί να δείχνει σε dashboard την **απότομη αύξηση pods** χρονικά συσχετισμένη με την αύξηση CPU. Αυτό

επιβεβαιώνει ότι «αν η χρήση CPU υπερβεί το όριο (40%), ο HPA προσθέτει pods για να διαμοιράσει το φορτίο».

- ***Λήξη spike:*** Υποθέτουμε ότι μετά από κάποιο διάστημα το traffic υποχωρεί (π.χ. οι χρήστες αποχωρούν ή σταματά το τεστ). Τότε, τα pods μπορεί να υποχρησιμοποιούνται (π.χ. < 10% CPU). Μετά από το σταθερό διάστημα χάριτος (stabilization window), ο HPA θα αποφασίσει **scale-down**. Ίσως μειώσει τα pods από 5 σε 2 ξανά, προκειμένου να μη σπαταλώνται πόροι. Το Deployment θα τερματίσει τα έξτρα pods (πιθανόν ένα-ένα ή δύο μαζί). Ο Ingress και το Service αφαιρούν τα τερματισμένα pods από τα endpoints τους, σταματώντας να τους στέλνουν traffic. Η κατάσταση επανέρχεται στην αρχική ηρεμία.

Στο παρακάτω σχήμα (Σχήμα 3.2) μας δείχνει το διάγραμμα ροής των γεγονότων και τη λειτουργία του HPA.

Σημείωση: Στο συγκεκριμένο σενάριο, καθ' όλη τη διάρκεια, το cluster είναι μονόκομβο. Αυτό σημαίνει ότι η προσθήκη pods αυξάνει τον φόρτο στον ίδιο φυσικό μηχανήμα. Υπήρχε επαρκής CPU/RAM στον host (12 λογικοί πυρήνες) ώστε να σηκώσει ταυτόχρονα 5 pods web, 2 pods MySQL, 1 Prometheus, 1 Grafana κλπ. Το monitoring θα έδειχνε επίσης την συνολική χρήση πόρων του node (οποία αν πλησίαζε επικίνδυνα το 100% θα ήταν προβληματικό). Στην περίπτωσή μας, η προσαρμογή των pods σταμάτησε στο maxReplicas=5, το οποίο το θέσαμε λαμβάνοντας υπόψη το όριο των πόρων.



Σχήμα 3.2: Αυξομείωση web pod με χρήση HPA (Horizontal Pod Autoscaler) και Kubernetes Metrics Server.

3.3.2 Σενάριο Αστοχίας Pod (Self-Healing Event)

Περιγραφή σεναρίου: Ένα από τα pods του συστήματος παρουσιάζει σφάλμα και τερματίζεται απροσδόκητα. Θα εξετάσουμε πώς το Kubernetes ανιχνεύει και αντιμετωπίζει αυτή την αστοχία, είτε πρόκειται για pod της web εφαρμογής είτε της βάσης δεδομένων. Στο παρακάτω σχήμα (Σχήμα 3.3) μας δείχνει το διάγραμμα ροής των γεγονότων ενός Self-Healing Event.

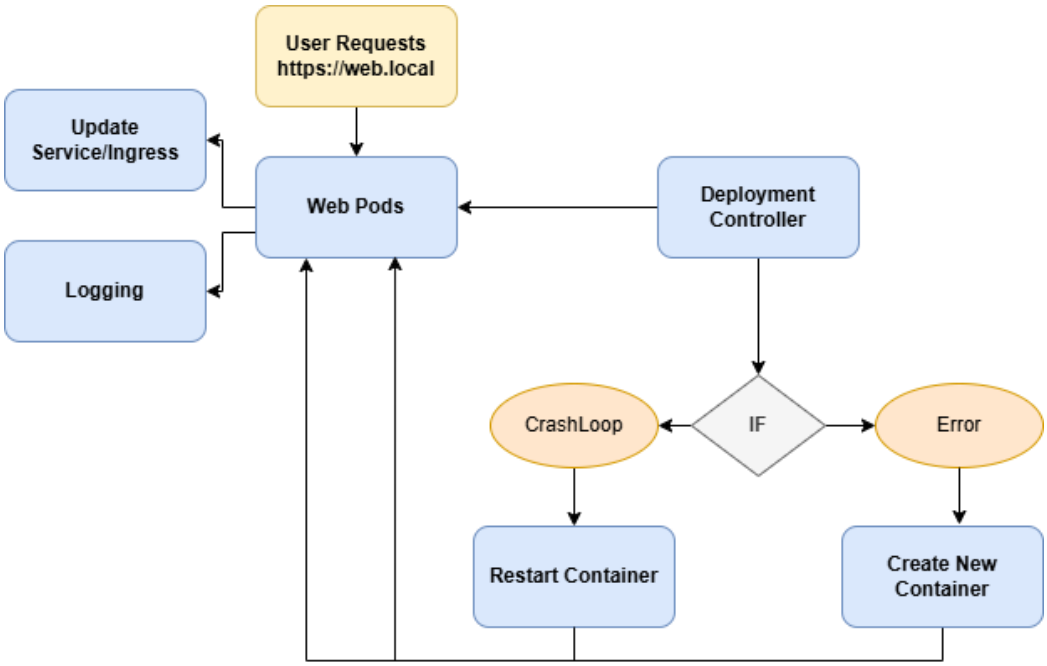
Ροή γεγονότων (για web pod):

1. ***Κανονική λειτουργία****:* Έστω ότι υπάρχουν 2 pods της εφαρμογής σε λειτουργία. Το Ingress στέλνει αιτήματα και στα 2. Οι liveness probes είναι περασμένες, το HPA είναι σταθερό.

2. ***Αστοχία**:* Ξαφνικά, το ένα pod (π.χ. web-app-xyz) κρασάρει – είτε λόγω ενός ανεπεξέργαστου exception στην εφαρμογή, είτε επειδή ο node σκοτώνει τη διαδικασία (π.χ. out-of-memory killer) ή για οποιονδήποτε λόγο. Το container τερματίζεται και το Kubernetes σημειώνει το pod σε κατάσταση CrashLoopBackOff ή Error.
3. ***Ανίχνευση**:* Το Deployment controller παρατηρεί ότι τώρα υπάρχουν **Actual pods = 1** ενώ **Desired pods = 2** (όπως ορίζει το deployment spec). Αυτή η ασυμφωνία οδηγεί σε δράση.
4. ***Απομάκρυνση προβληματικού pod**:* Το Kubernetes θα συλλέξει logs ή events για το crash (τα οποία θα δούμε στον Prometheus ίσως ως συμβάν ή απλώς στα logs). Το pod web-app-xyz θεωρείται πλέον *failed*. Αν ο λόγος τερματισμού ήταν μόνιμος (error code όχι restartable), πιθανόν να μην επιχειρηθεί επανεκκίνηση του ίδιου container. Συνήθως, τα Deployments αντί για restart in-place δημιουργούν νέο pod αντικατάστασης.
5. ***Δημιουργία νέου pod**:* Ο ReplicaSet του Deployment ξεκινά ένα νέο pod (π.χ. web-app-abc) για να επανέλθει ο αριθμός στο 2. Ο scheduler το τοποθετεί στον κόμβο (μοναδική επιλογή). Κατεβάζει το image αν χρειάζεται (πιθανώς ήδη διαθέσιμο) και το εκκινεί. Μέσα σε σύντομο χρόνο, το νέο web pod είναι up και περνά τα readiness checks.
6. ***Ενημέρωση Service/Ingress**:* Το Service της εφαρμογής αφαιρεί αυτόματα το παλιό pod από τα endpoints (μόλις ανιχνεύσει ότι πέθανε, ή δεν αποκρίνεται στον health check). Προσθέτει το νέο pod web-app-abc μόλις αυτό γίνει Ready. Ο Ingress controller ενημερώνεται ανάλογα – αφαιρεί το crashαρισμένο backend και αρχίζει να στέλνει traffic στο νέο pod. Από τη σκοπιά του τελικού χρήστη, μπορεί να παρατηρήθηκε ένα μικρό glitch (ίσως κάποια αιτήματα που πήγαν στο pod τη στιγμή που κράσαρε απέτυχαν), αλλά γενικά η υπηρεσία συνεχίζει να λειτουργεί καθώς το δεύτερο pod εξυπηρετούσε ενδιάμεσα και σύντομα επανήλθε το δεύτερο.
7. ***Αναφορά/Logging**:* Το συμβάν καταγράφεται: ο Prometheus μπορεί να έχει κανόνα alert που να ενεργοποιείται όταν πέσει ένα pod (π.χ. αν το up metric του πάει στο 0). Η Grafana μπορεί να εμφανίσει ένα alert στον πίνακα. Σε παραγωγικό περιβάλλον, θα

ειδοποιούσε DevOps. Εδώ απλά το παρατηρούμε στα dashboards. Το Kubernetes events θα δείχνει ότι το pod web-app-xyz πέθανε και δημιουργήθηκε το web-app-abc.

8. ***Σταθεροποίηση:*** Το σύστημα επανέρχεται στην κανονική του κατάσταση με 2 υγιή pods. Ο HPA μπορεί να μην επηρεάστηκε καθόλου από το σύντομο αυτό συμβάν (εκτός ίσως αν μετρήθηκε χαμηλότερο μέσο CPU και προσπάθησε να κλιμακώσει, αλλά δεδομένου του μικρού χρόνου είναι απίθανο να αντιδράσει έτσι).



Σχήμα 3.3: Διάγραμμα ροής σε περίπτωση σεναρίου αστοχίας ενός web pod.

Ροή γεγονότων (για MySQL pod):

Αντίστοιχα, ας δούμε εν συντομία το τι γίνεται αν κρασάρει η MySQL:

- Αν το **MySQL master pod** αποτύχει: Το StatefulSet παρακολουθεί τα pods. Βλέπει ότι λείπει το mysql-0 και θα επιχειρήσει να το **επανεκκινήσει** αυτόματα (StatefulSet always maintains each replica). Θα γίνει restart container εντός του ίδιου pod (αν ο κόμβος είναι up). Αν ολόκληρο το pod χαθεί, θα δημιουργηθεί εκ νέου με το ίδιο όνομα και θα επαναπροσαρτήσει τον persistent volume με τα δεδομένα. Κατά την εκκίνηση, η MySQL μπορεί να ανιχνεύσει ότι δεν έκλεισε σωστά και να κάνει recovery (αναπαράγοντας τα binlogs). Το replica **mysql-1** στο διάστημα αυτό πιθανώς θα σταματήσει να λαμβάνει νέες εγγραφές (αφού δεν υπάρχει master). Μόλις το master επανέλθει, ο replica

connector θα ξανασυνδεθεί και θα συγχρονίσει ό,τι χάθηκε. Η εφαρμογή web ίσως αντιμετώπισε αποτυχία σε μερικα requests που επιχειρούσαν εγγραφή κατά το downtime (θα λάμβανε σφάλμα σύνδεσης DB), αλλά αυτά είναι παροδικαά. Δεν υπάρχει αυτόματη προαγωγή του replica, οπότε μέχρι να επιστρέψει το master, οι εγγραφές στη βάση δεν είναι δυνατές (το σύστημα σε αυτό το διάστημα ουσιαστικά λειτουργεί σε degraded mode – μόνο αναγνώσεις από το replica θα μπορούσαν θεωρητικά να γίνουν). Ο διαχειριστής θα ενημερωθεί για τη βλάβη (π.χ. μέσω alert) και θα εξετάσει τα logs της MySQL.

- Αν το **MySQL replica pod** αποτύχει: Το StatefulSet θα αναδημιουργήσει το mysql-1. Σε αυτήν την περίπτωση ο αντίκτυπος είναι μικρότερος. Η κύρια βάση συνεχίζει να εξυπηρετεί κανονικά αιτήσεις (εγγραφές/αναγνώσεις). Το replica μένει πίσω μέχρι να επιστρέψει. Μόλις επανεκκινηθεί το pod, θα συνδεθεί στο master και θα ζητήσει τα binlog entries από το τελευταίο σημείο που είχε λάβει. Αυτό θα συμβεί αυτόματα καθώς η MySQL έχει ρυθμιστεί (CHANGE MASTER TO ...) να γνωρίζει τον master host και log position. Οπότε σύντομα ο replica θα καλύψει τη διαφορά και θα συγχρονίσει πλήρως. Η εφαρμογή και οι χρήστες πιθανότατα δεν αντιλαμβάνονται τίποτα, εκτός ίσως από ελαφρώς αυξημένο φορτίο στο master όσο έλειπε ο replica (αν υποθέταμε ότι εξυπηρετούσε και αναγνώσεις πριν, τώρα όλες οι αναγνώσεις πήγαν στο master προσωρινά).

Monitoring & Alerts: Και στα δύο pods MySQL, ο Prometheus/Grafana μπορεί να παρακολουθεί την κατάσταση replication (lag, IO thread status κ.λπ.). Σε περίπτωση πτώσης του master, το lag θα μεγαλώσει (καθώς ο replica δεν λαμβάνει τίποτα), και μπορεί να υπάρχει alert “replication halted”. Αυτές είναι λεπτομέρειες πέραν του βασικού αντικειμένου, αλλά είναι σημαντικό να αναφερθεί ότι το σύστημα προσφέρει τις πληροφορίες ώστε ο διαχειριστής να καταλάβει τί συμβαίνει και να επέμβει αν χρειάζεται (π.χ. σε παρατεταμένη αποτυχία του master, ίσως να κάνει χειροκίνητα promote το replica).

Γενικά, το Kubernetes επαληθεύει την αρχή του **self-healing**: *"Διατηρεί συνεχώς την επιθυμητή κατάσταση των εφαρμογών – αν κάτι διαφέρει, ενεργεί για να το διορθώσει"*. Σε συνδυασμό με replication μηχανισμούς (για βάση) και redundancy (πολλαπλά pods για

stateless), αυτό προσφέρει υψηλή διαθεσιμότητα στο βαθμό που το επιτρέπει η φυσική μας υποδομή (ένας κόμβος).

3.4 Εργαλεία και Τεχνολογίες που χρησιμοποιήθηκαν

Σε αυτήν την ενότητα συνοψίζονται όλα τα εργαλεία λογισμικού (και ορισμένα στοιχεία υλικού) που αξιοποιήθηκαν για την υλοποίηση, μαζί με τον ρόλο τους στο σύστημα:

- **Docker Desktop (Windows 10):** Παρέχει το υπόβαθρο για την εκτέλεση containers στο λειτουργικό Windows, μέσω ενός ελαφρού Linux VM χρησιμοποιώντας το WSL2. Η έκδοση Docker Desktop που χρησιμοποιήθηκε περιλάμβανε ενσωματωμένη υποστήριξη Kubernetes (single-node cluster). Αυτό απλοποίησε το στήσιμο του cluster χωρίς χειροκίνητη εγκατάσταση kubeadm. Ο Docker Engine χρησιμοποιείται ως container runtime για όλα τα pods. Μέσω του Docker Desktop UI/CLI μπορούμε να παρακολουθούμε εν μέρει τους πόρους, αλλά βασικά το cluster διαχειρίζεται μέσω kubectl.
- **Kubernetes (v1.26+):** Το σύστημα ορχήστρωσης κοντέινερ που αποτελεί τον πυρήνα της αρχιτεκτονικής. Παρέχει τους controllers (Deployment, StatefulSet, HPA, Service, Ingress) και τα APIs για αυτοματοποίηση. Η εκδοχή είναι αρκετά σύγχρονη ώστε να υποστηρίζει όλα τα χρησιμοποιούμενα features (HPA v2beta2, Ingress API networking.k8s.io, etc.). Το cluster ρυθμίστηκε να τρέχει με εξουσιοδότηση δικτύου Calico (προεπιλογή Docker Desktop) και coreDNS για name resolution.
- **Chat Web Application:** Πρόκειται για την εφαρμογή που αναπτύχθηκε ως μέρος της πτυχιακής (ή αξιοποιήθηκε έτοιμη). Σημαντικό είναι ότι πακεταρίστηκε σε μορφή Docker image και δηλώθηκε στο Deployment. Η εφαρμογή σχεδιάστηκε να είναι **stateless**: δεν αποθηκεύει συνεδρίες ή δεδομένα τοπικά στο container, παρά μόνο στη MySQL. Έτσι, μπορεί να κλιμακώνεται οριζόντια εύκολα. Επίσης, εκτίθεται μέσω HTTP (πιθανώς REST API + WebSockets). Χάρη στο Ingress, οι χρήστες την προσπελάζουν στο https://web.local. Το autoscaling δοκιμάστηκε με αυτή την εφαρμογή για να διασφαλιστεί ότι λειτουργεί (π.χ. με αυξανόμενα threads του εργαλείου **hey** που

κάνουν requests ταυτόχρονα, μπορούμε να προκαλέσουμε CPU spike και να πυροδοτήσουμε τον HPA).

- **MySQL Server 8.x:** Το δημοφιλές σύστημα διαχείρισης σχεσιακών βάσεων δεδομένων. Χρησιμοποιήθηκε η επίσημη εικόνα Docker MySQL (π.χ. mysql:8.0) με κατάλληλες ρυθμίσεις για replication (–server-id, –log-bin για master, --replicate-do-db, etc). Ένα αρχείο init SQL ρυθμίζει τον χρήστη replica και τα δικαιώματα. Το StatefulSet εξασφάλισε ότι τα pods έχουν σταθερά hostnames (mysql-0.mysql για master, mysql-1.mysql για replica) ώστε οι ρυθμίσεις replication να είναι αξιόπιστες. Η MySQL χρησιμοποιεί όγκο (PersistentVolumeClaim) έτσι ώστε τα δεδομένα (database files) να μην χάνονται σε restart. Η επιλογή MySQL έγινε επειδή είναι ένα πραγματικό DBMS που προσθέτει ρεαλισμό (αντί για μια in-memory ή απλή storage). Δοκιμάστηκε ότι η εφαρμογή μπορεί να συνδεθεί στο master και να εκτελεί CRUD πράξεις. Στο Grafana, μέσω ενός plugin ή μέσω exporters (π.χ. **prom/mysqld-exporter** container), παρακολουθήθηκε η κατάσταση της βάσης.
- **NGINX Ingress Controller:** Επιλογή ingress υλοποίησης λόγω της ευρείας υποστήριξης και τεκμηρίωσης. Εγκαταστάθηκε ακολουθώντας τις οδηγίες της κοινότητας (π.χ. kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/... για να δημιουργηθούν deployment, service, configmaps). Έτρεξε στο namespace ingress-nginx. Ένας Service τύπου LoadBalancer (ingress-nginx-controller) δημιουργήθηκε αυτόματα. Το ingress config δημιουργήθηκε ως εξής:
  - Host web.local -> Service/web-app port 80
  - Host grafana.local -> Service/grafana port 3000
  - Host prometheus.local -> Service/prometheus port 9090

    Επιπλέον, μια ανακατεύθυνση HTTP->HTTPS έχει ρυθμιστεί (με annotation ή config) ώστε αν κάποιος καλέσει http://web.local να γίνεται redirect σε https. Ο Ingress Controller απλοποίησε πολύ την πρόσβαση, αλλιώς θα έπρεπε να ανοίξουμε ξεχωριστά NodePorts για καθεμιά υπηρεσία και να θυμόμαστε custom ports.

- **MetalLB:** Η λύση software load balancer, έκδοση 0.13+. Εγκαταστάθηκε στο namespace metallb-system (deployment controller + daemonset speaker). Ρυθμίστηκε με ConfigMap που ορίζει ένα pool π.χ. 192.168.1.240-192.168.1.250 (μοναδική IP). Αυτή η IP ανατέθηκε στον ingress service. Η επιλογή layer2 mode έκανε broadcast ARP: έτσι ο router/συσκευές βλέπουν τη MAC του host να απαντά για την .200 IP. Εναλλακτικά, αν το δίκτυο δεν επέτρεπε ARP, θα μπορούσε να χρησιμοποιηθεί BGP mode (αν είχαμε router που μιλά BGP), αλλά σε οικιακό δίκτυο το ARP είναι εντάξει. Μετά την ρύθμιση, το kubectl get svc ingress-nginx-controller -o wide εμφάνιζε External-IP = 192.168.1.240, ένδειξη ότι δούλεψε. Χωρίς το MetalLB, το External-IP θα έμενε <pending> και δεν θα μπορούσαμε να προσπελάσουμε το ingress εξωτερικά.
- **Prometheus:** Όπως περιγράφηκε, εργαλείο συλλογής μετρικών. Πιθανόν αναπτύχθηκε μέσω του **kube-prometheus-stack** (Helm chart) που περιέχει Prometheus, Grafana, Alertmanager προρυθμισμένα για Kubernetes. Αυτό παρέχει πολλά πλεονεκτήματα: π.χ. έτοιμα Grafana dashboards για nodes, pods, deployments, HPA. Ο Prometheus στο cluster ρύθμισε targets αυτόματα (service discovery). Περιλήφθηκε η συλλογή metrics από Metrics Server, από cAdvisor (pods resource usage) κ.λπ. Το retention ίσως τέθηκε σε μερικές ημέρες για να μην αυξηθεί πολύ ο όγκος δεδομένων. Επίσης, δημιουργήθηκαν alerts test (π.χ. CPU usage > 80% για 5 λεπτά) απλώς για δοκιμή.
- **Grafana:** Εφαρμογή web για διαγράμματα. Αφού τέθηκε το data source Prometheus, εισήχθησαν κάποια δημοφιλή dashboards από το Grafana community (p.χ. Kubernetes cluster overview, Deployment HPA status κ.α.). Επίσης, δημιουργήθηκε ένα custom dashboard ειδικά για την εφαρμογή μας, που δείχνει το **Web App Autoscaling**: γράφημα με τον αριθμό pods στο χρόνο, με overlay τη CPU utilization. Αυτό βοήθησε στην οπτική επαλήθευση του HPA threshold. Το Grafana προστατεύτηκε με απλό auth (user/password) ώστε αν και είναι σε εσωτερικό δίκτυο, να μην είναι ανοιχτό χωρίς έλεγχο – ιδίως επειδή μέσω Tailscale είναι προσβάσιμο εκτός.
- **Tailscale:** Υπηρεσία VPN mesh. Η ρύθμιση έγινε δημιουργώντας έναν λογαριασμό (μέσω OAuth GitHub) και προσθήκη των συσκευών. Ο host γράφτηκε με tailscale up --accept-routes --accept-dns, και οι clients (λ.χ. laptop) με το app. Στον admin panel,

ορίστηκε under DNS settings: DNS server = 100.100.100.100 (Tailscale MagicDNS) και search domain = arbiter.local (τυχόν domain tailnet) – αυτά για default. Για το split-DNS, προστέθηκε Nameserver = [IP host] για domain suffix .local. Αυτό κατηύθυνε όλα τα .local queries στο dnsproxy. Το Tailscale εκτέλεσε σωστά το ρόλο του: μόλις συνδέεται ο client, λαμβάνει IP π.χ. 100.XX και προσθέτει route για τα 192.168.50.0/24 μέσω του host (εφόσον accept-routes). Έτσι μπορούσε να φτάσει την IP του ingress. Όλη η επικοινωνία κρυπτογραφείται (WireGuard, ChaCha20) και επιπλέον απαιτεί authentication, οπότε θεωρείται ασφαλής.

- **dnsproxy (AdGuard DNS Proxy):** Ένα ελαφρύ εργαλείο (πηγή στο GitHub adguardteam) που ακούει σε συγκεκριμένη πόρτα UDP/TCP 53 στο host. Ρυθμίστηκε να κάνει forward τα ερωτήματα στο CoreDNS του cluster (ο οποίος άκουγε μόνο εντός cluster IP, οπότε χρησιμοποιήθηκε η IP του docker VM ή έγινε mapping port). Εναλλακτικά, θα μπορούσε να χρησιμοποιηθεί κατευθείαν ο CoreDNS αν ακούει στην tailnet IP. Το dnsproxy έχει το πλεονέκτημα ότι μπορεί να συνδυάζει upstreams: π.χ. “για .local πήγαινε σε 10.0.0.2 (CoreDNS), για όλα τα άλλα πήγαινε σε 1.1.1.1 (Cloudflare)” αλλά στη δική μας χρήση το 2ο δεν χρειαζόταν, απλώς το tailscale μπορούσε να συνεχίσει με το public DNS του.
- **Εργαλεία φόρτου (hey, stress pods):** Για τις δοκιμές δημιουργήθηκαν είτε εξωτερικά είτε εντός cluster pods που παράγουν τεχνητό φόρτο. Το εργαλείο **hey** (https://github.com/rakyll/hey) είναι ένα CLI γραμμένο σε Go που επιτρέπει να σταλεί μεγάλο πλήθος HTTP αιτημάτων ως benchmark. Χρησιμοποιήθηκε π.χ. με παραμέτρους -c 100 -q 50 -z 2m https://web.local/chat για να προσομοιώσει 100 ταυτόχρονους χρήστες που στέλνουν 50 αιτήματα το δευτερόλεπτο για 2 λεπτά. Αυτό αρκούσε να αυξήσει αισθητά τη CPU των pods. Επίσης, δημιουργήθηκε ένα απλό **stress pod** (alpine + yes > /dev/null) για να δοκιμαστεί η αντίδραση του cluster σε node stress (αν και δεν είχαμε cluster autoscaler για node scaling, μόνο για παρατήρηση). Τα αποτελέσματα αυτών των tests καταγράφηκαν και παρουσιάστηκαν (βλ. Κεφάλαιο Αποτελεσμάτων) ώστε να αξιολογηθούν τα thresholds και η συμπεριφορά του συστήματος.

Συνολικά, ο σχεδιασμός του συστήματος συνδύασε πολλές τεχνολογίες ανοιχτού κώδικα για να πετύχει τους στόχους. Κάθε εργαλείο επιλέχθηκε με γνώμονα την καταλληλότητά του σε

τοπικό περιβάλλον: π.χ. το MetalLB ειδικά σχεδιάστηκε για on-prem clusters, το Tailscale για εύκολη P2P VPN χωρίς περίπλοκες ρυθμίσεις, το Prometheus/Grafana ως στάνταρ για cloud native monitoring, κ.ο.κ. Το αποτέλεσμα είναι ένα ολοκληρωμένο οικοσύστημα, μικρής κλίμακας αλλά αντιπροσωπευτικό πραγματικών deployment, που καλύπτει απαιτήσεις από επίπεδο δικτύου μέχρι εφαρμογής και επιδεικνύει σημαντικές ιδιότητες του Kubernetes (αυτοματοποίηση κλιμάκωσης, αυτοΐαση, φορητότητα υπηρεσιών). Οι επόμενες φάσεις της εργασίας περιλαμβάνουν την αξιολόγηση της απόδοσης αυτού του συστήματος υπό διάφορες συνθήκες και τη συζήτηση πιθανών βελτιώσεων ή επεκτάσεων του σχεδιασμού.

Κεφάλαιο 4 – Υλοποίηση Συστήματος

4.1 Εγκατάσταση WSL και Docker Desktop

Σε αυτό το στάδιο, περιγράφονται αναλυτικά τα βήματα που απαιτούνται για να δημιουργηθεί το υπόβαθρο πάνω στο οποίο θα χτιστεί το Kubernetes cluster. Πιο συγκεκριμένα, παρουσιάζεται η εγκατάσταση του **WSL 2** και του **Docker Desktop**, τα οποία χρησιμοποιούνται για την εκτέλεση containers και τη δημιουργία του ενός και μοναδικού κόμβου (node) του τοπικού cluster.

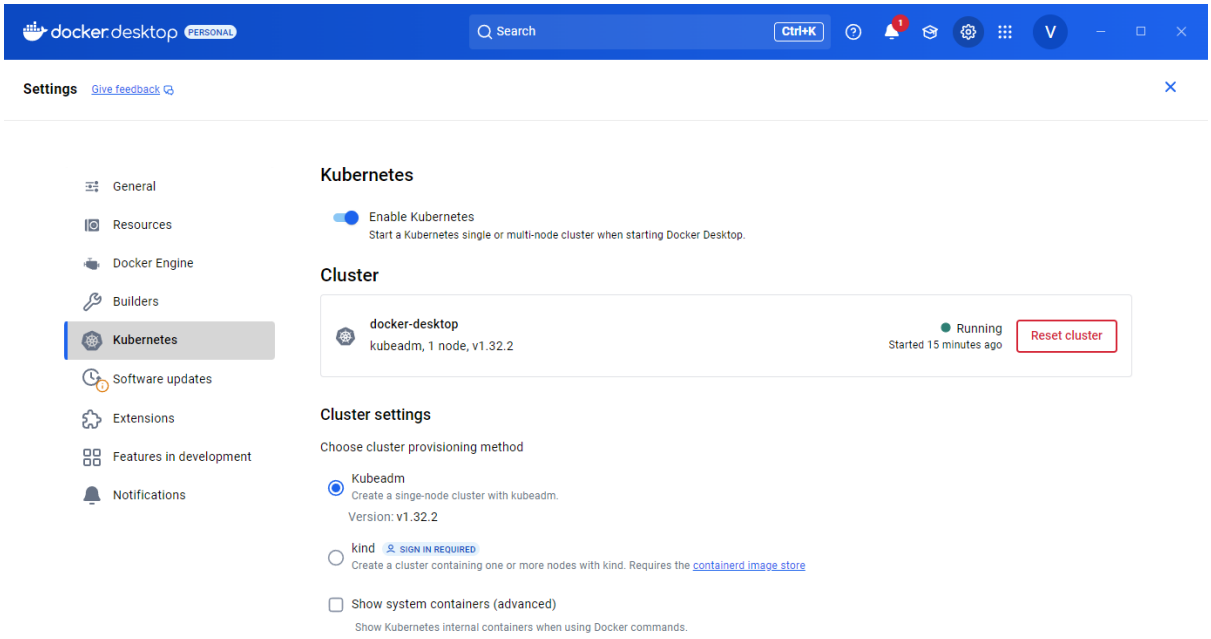
Βήμα 1: Ενεργοποίηση του WSL στο Windows 10

Για την εκτέλεση του Docker Desktop σε Windows 10, είναι απαραίτητο να ενεργοποιηθεί το **Windows Subsystem for Linux (WSL)** και να εγκατασταθεί μια κατάλληλη Linux διανομή (όπως Ubuntu). Αυτό επιτρέπει στο Docker να χρησιμοποιεί ένα lightweight VM βασισμένο σε Linux.

wsl --install

Αφού έχει ενεργοποιηθεί το WSL, γίνεται εγκατάσταση του Docker Desktop για Windows, ο οποίος προσφέρει ένα πλήρες περιβάλλον containerization και υποστηρίζει Kubernetes cluster ενός κόμβου.

Παρακάτω ενεργοποιούμε το **kubernetes** απο τα settings με **kubeadm** (Εικόνα 4.1.1)**.**



Εικόνα 4.1.1 Ενεργοποίηση Kubernetes στο Docker Desktop με kubeadm

4.2 Εγκατάσταση και Ρύθμιση του MetalLB

Το **MetalLB** είναι ένα load-balancer για Kubernetes clusters που δεν τρέχουν σε περιβάλλον cloud και δεν έχουν native υποστήριξη από παρόχους, όπως είναι η περίπτωση του παρόντος τοπικού (on-premise) συστήματος. Το MetalLB επιτρέπει στο cluster να εκχωρεί στατικές IP διευθύνσεις σε υπηρεσίες τύπου LoadBalancer, κάνοντάς τες προσβάσιμες στο τοπικό δίκτυο.

Η εγκατάσταση και ρύθμιση του MetalLB περιλαμβάνει δύο βασικά στάδια:

Βήμα 1: Εγκατάσταση MetalLB στο cluster

Η εγκατάσταση γίνεται μέσω YAML manifests, με χρήση του kubectl apply.

```
kubectl apply -f
https://raw.githubusercontent.com/metallb/metallb/v0.13.11/config/manifests/metallb-native.yaml
```

Βήμα 2: Ρύθμιση address pool (ConfigMap)

Το **MetalLB** απαιτεί τη δήλωση ενός range από IP διευθύνσεις που μπορεί να εκχωρεί στις υπηρεσίες. Το range αυτό πρέπει να είναι μέρος του τοπικού σου LAN (π.χ. 192.168.1.240-192.168.1.250), χωρίς να προκαλεί συγκρούσεις με άλλες συσκευές (Εικόνα 4.2.1).

```
kubectl apply -f metallb-config.yaml
```

metallb-config.yaml

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  namespace: metallb-system
  name: config
data:
  config: |
    address-pools:
    - name: default
      protocol: layer2
      addresses:
      - 192.168.1.240-192.168.1.250
```

Εικόνα 4.2.1 ConfigMap για να οριστεί το address pool.

4.3 Εγκατάσταση και Ρύθμιση του NGINX Ingress Controller

Ο **NGINX Ingress Controller** αποτελεί βασικό στοιχείο στην αρχιτεκτονική του cluster, λειτουργώντας ως ενιαίο σημείο εισόδου (Single Point of Entry) για όλα τα εξωτερικά αιτήματα HTTP/HTTPS προς τις εφαρμογές που τρέχουν στο εσωτερικό του Kubernetes. Στην παρούσα υλοποίηση, χρησιμοποιήθηκε η εκδοχή του NGINX Ingress Controller που διατίθεται επίσημα από την κοινότητα Kubernetes, και η εγκατάσταση έγινε μέσω του **Helm**, του πιο διαδεδομένου package manager για Kubernetes.

Βήμα 1: Προσθήκη Helm repository και ενημέρωση

```
helm repo add ingress-nginx https://kubernetes.github.io/ingress-nginx

helm repo update
```

Βήμα 2: Εγκατάσταση Ingress Controller με Helm

Η εγκατάσταση περιλαμβάνει την παράμετρο --set controller.service.type=LoadBalancer ώστε να ανατεθεί στο Ingress μία IP μέσω MetalLB. Επίσης, μπορεί να έχουν ρυθμιστεί options για HTTPS, nodeSelector, ή άλλες επιλογές.

```
helm install ingress-nginx ingress-nginx/ingress-nginx `
 --namespace ingress-nginx `
 --create-namespace `
 -f C:\k8s\ingress-nginx-values.yaml `
 --set controller.service.type=LoadBalancer `
 --set controller.service.loadBalancerIP=192.168.1.240
```

Αυτό που αλλάχτηκε στο **ingress-nginx-values.yaml** είναι το παρακάτω

```
controller:
 deployment:
  minReadySeconds: 20
  progressDeadlineSeconds: 60
```

Βήμα 3: Δημιουργία tls secret για HTTPS

```
openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
 -keyout tls.key -out tls.crt -subj "/CN=*.local"
```

```
kubectl create secret tls ingress-nginx-tls `
 --cert=C:\k8s\tls.crt `
 --key=C:\k8s\tls.key `
 --namespace ingress-nginx
```

Βήμα 4: Επιβεβαίωση εγκατάστασης και απόδοση IP

Μετά την εγκατάσταση, το service ingress-nginx-controller θα πρέπει να λάβει IP από το MetalLB. Αυτό μπορεί να επαληθευτεί με kubectl get svc (Εικόνα 4.3.1).

```
PS C:\Windows\system32> kubectl get svc -n ingress-nginx
NAME                                 TYPE           CLUSTER-IP       EXTERNAL-IP                 PORT(S)                      AGE
ingress-nginx-controller             LoadBalancer   10.108.251.32    192.168.1.240,100.74.6.5    80:31251/TCP,443:31845/TCP   33d
ingress-nginx-controller-admission   ClusterIP      10.104.237.254   <none>                      443/TCP                      29d
ingress-nginx-metrics                ClusterIP      10.99.65.15      <none>                      10254/TCP                    21d
```

Εικόνα 4.3.1 Έλεγχος ορισμού external IP στον Nginx Ingress Controller

4.4 Ανάπτυξη Web Server με 2 Replicas

Η βασική εφαρμογή της πτυχιακής εργασίας είναι ένας απλός **web server τύπου chat**, ο οποίος δέχεται και αποθηκεύει μηνύματα σε μια βάση δεδομένων MySQL. Για να εξασφαλιστεί ανθεκτικότητα και ισορροπία φορτίου, η εφαρμογή τρέχει σε **δύο αντίγραφα (replicas)** υπό τη μορφή Kubernetes pods. Η ανάπτυξη έγινε με χρήση δύο YAML αρχείων: ενός Deployment και ενός Service.

4.4.1 Web Deployment

Το αρχείο web-server-deployment.yaml (Εικόνα 4.4.1,4.4.2) περιλαμβάνει τη δήλωση του deployment που καθορίζει:

- Το container image του web server
- Τον αριθμό των replicas (2)
- Τα ports που εκτίθενται (συνήθως 80)
- Την πολιτική επανεκκίνησης
- Όνομα και labels για τη συσχέτιση με το service

web-server-deployment.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web-server
  namespace: default
spec:
  replicas: 2
  selector:
    matchLabels:
      app: web-server
  template:
    metadata:
      labels:
        app: web-server
    spec:
      containers:
      - name: web-server
        image: voulgaris/web-php-nginx.v2:latest
        imagePullPolicy: Always
        ports:
        - containerPort: 80
        - containerPort: 443
        env:
        - name: MYSQL_HOST
          value: "my-mysql-primary"
        - name: MYSQL_USER
          value: "root"
        - name: MYSQL_PASSWORD
          value: "yourRootPassword"
        - name: MYSQL_DATABASE
          value: "mydatabase"
        resources:
          requests:
            cpu: 100m
          limits:
            cpu: 500m
```

Εικόνα 4.4.1 web-server-deployment.yaml το deployment του web server με 2 replicas.

```yaml
        volumeMounts:
        # Πρώτα το override του default site
        - name: empty-default
          mountPath: /etc/nginx/sites-enabled/default
          subPath: default
        # Μετά το custom config
        - name: nginx-config
          mountPath: /etc/nginx/conf.d/default.conf
          subPath: default.conf

      - name: nginx-prometheus-exporter
        image: nginx/nginx-prometheus-exporter:latest
        args:
          - -nginx.scrape-uri=http://localhost/nginx_status
        ports:
        - containerPort: 9113

      volumes:
      - name: empty-default
        configMap:
          name: empty-default
      - name: nginx-config
        configMap:
          name: nginx-config
          items:
          - key: default.conf
            path: default.conf
```

Εικόνα 4.4.2 (Συνέχεια) web-server-deployment.yaml το deployment του web server με 2 replicas.

4.4.2 Web Service

Το αρχείο web-server-service.yaml (Εικόνα 4.4.3) καθορίζει ένα Kubernetes service τύπου ClusterIP (ή NodePort αν επιλέχθηκε έτσι), που κάνει expose τα pods με βάση τα labels τους. Αυτό επιτρέπει την πρόσβαση στον web server από άλλα pods ή από τον Ingress Controller.

web-server-service.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  name: web-server-service
  namespace: default
spec:
  selector:
    app: web-server
  ports:
  - protocol: TCP
    port: 80
    targetPort: 80
  type: ClusterIP
```

Εικόνα 4.4.3 Service για τον web-server για να γίνουν expose τα pods με βάση τα labels τους.

4.5 Ανάπτυξη MySQL σε Master/Slave Αρχιτεκτονική με Helm

Για την αποθήκευση των μηνυμάτων του web chat, χρησιμοποιήθηκε ένα σύστημα βάσης δεδομένων **MySQL** σε **αρχιτεκτονική αναπαραγωγής τύπου Master/Slave (Primary/Replica)**. Η χρήση replication εξασφαλίζει αυξημένη διαθεσιμότητα και προστασία από απώλεια δεδομένων, ενώ ανοίγει δυνατότητες για future read-scaling και disaster recovery.

Η ανάπτυξη έγινε με χρήση του **Helm**, αξιοποιώντας το επίσημο chart της Bitnami για το MySQL, το οποίο υποστηρίζει native replication μεταξύ του primary και των replicas.

4.5.1 Προετοιμασία του Helm Chart

Αρχικά, προστέθηκε το Bitnami repository:

```
helm repo add bitnami https://charts.bitnami.com/bitnami

helm repo update
```

4.5.2 Εγκατάσταση του Chart με Ρυθμίσεις Replication

Η εγκατάσταση έγινε με ορισμένες σημαντικές παραμέτρους (Εικόνα 4.5.1), όπως:

- architecture=replication για ενεργοποίηση Master/Slave
- auth.rootPassword για τον κωδικό διαχειριστή
- replica.replicaCount=1 για 1 replica (σύνολο 2 pods: 1 master + 1 slave)

mysql-values.yaml

```yaml
architecture: replication

replication:
  enabled: true
  slaveReplicas: 1

auth:
  rootPassword: "yourRootPassword"
  replicationPassword: "yourReplPassword"
  database: "mydatabase"
  username: "myuser"
  password: "mypassword"

primary:
  service:
    port: 3306

secondary:
  service:
    port: 3306

metrics:
  enabled: true
  service:
    port: 9104
```

Εικόνα 4.5.1 Αρχείο values για εγκατάσταση μαζί με το helm chart για ορισμό των παραμέτρων που χρειάζονται.

4.5.3 Επιβεβαίωση Λειτουργίας Replication

Μετά την εγκατάσταση, το Helm chart δημιουργεί δύο pods: mysql-primary και mysql-secondary-0 (ή με ανάλογα ονόματα) και τη δημιουργία των persistent volumes (Εικόνα 4.5.1).

```
PS C:\Windows\system32> kubectl get pods -A |findstr mysql
default          my-mysql-primary-0                                2/2     Running   10 (157m ago)   21d
default          my-mysql-secondary-0                              2/2     Running   40 (157m ago)   33d
PS C:\Windows\system32> kubectl get pvc
NAME                      STATUS   VOLUME                                     CAPACITY   ACCESS MODES   STORAGECLASS   VOLUMEATTRIBUTESCLASS   AGE
data-my-mysql-primary-0   Bound    pvc-27b25949-f3ab-4956-9151-8e1bfaefb563   8Gi        RWO            hostpath       <unset>                 33d
data-my-mysql-secondary-0 Bound    pvc-5569fb67-e10a-44f5-a05a-920eeaf212a7   8Gi        RWO            hostpath       <unset>                 33d
```

Εικόνα 4.5.1 Έλεγχος δημιουργίας των pods της mysql.

4.6 Εγκατάσταση Prometheus και Grafana με Helm

Για την παρακολούθηση (monitoring) των pods και υπηρεσιών εντός του Kubernetes cluster, αξιοποιήθηκαν τα εργαλεία **Prometheus** και **Grafana**, τα οποία είναι industry-standard λύσεις για τη συλλογή και οπτικοποίηση μετρικών. Η εγκατάστασή τους έγινε μέσω Helm charts, τα οποία παρέχουν ολοκληρωμένες υλοποιήσεις με τις απαραίτητες προεπιλεγμένες ρυθμίσεις.

4.6.1 Προσθήκη των Helm Repositories

```
helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo add grafana https://grafana.github.io/helm-charts

helm repo update
```

4.6.2 Εγκατάσταση του kube-prometheus-stack

Το chart kube-prometheus-stack περιλαμβάνει:

- Prometheus Operator
- Prometheus Server
- Grafana
- Node Exporter
- Kube-state-metrics

#Δημιουργία του namespace

```
kubectl create namespace monitoring
```

#Πρέπει να εγκαταστήσουμε πρώτα τα CRDs του Prometheus

```
helm install prometheus-operator-crds prometheus-community/prometheus-operator-crds --namespace monitoring
```

#Εγκατάσταση Prometheus

```
helm install prometheus bitnami/kube-prometheus `
 --namespace monitoring `
 --set nodeExporter.enabled=false
```

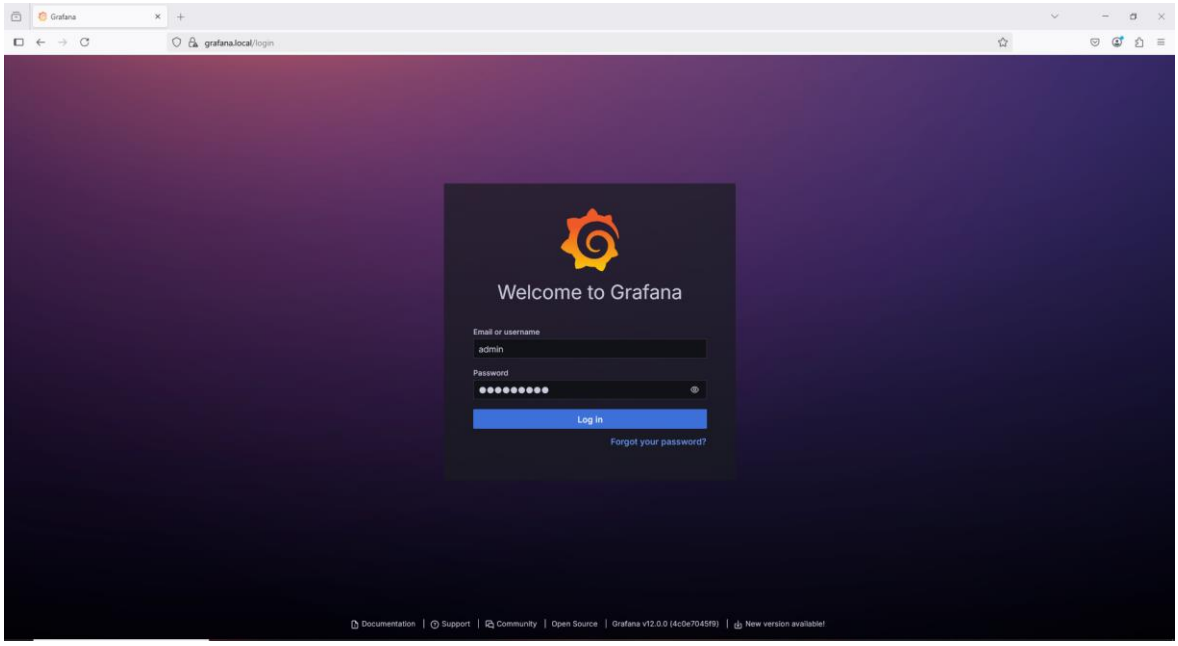
4.6.3 Πρόσβαση στη Grafana

Μετά την εγκατάσταση, η Grafana έγινε προσβάσιμη μέσω του Ingress Controller στην τοπική διεύθυνση https://grafana.local.(Θα λειτουργεί αφού εγκατασταθεί ο DNS) (Εικόνα 4.6.1,4.6.2,4.6.3)

Για να συνδεθεί ο χρήστης:

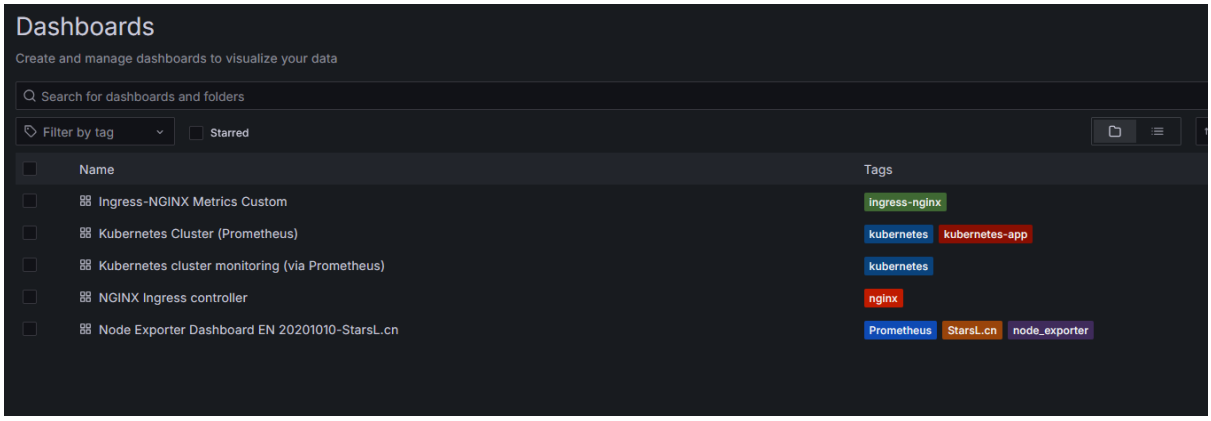
- Username: admin
- Password: admin1234

Login Page



Εικόνα 4.6.1 Login Page του Grafana

Dashboards



Εικόνα 4.6.2 Dashboards για έλεγχο του cluster.

Cluster Monitoring Dashboard



Εικόνα 4.6.3 Dashboard με δεδομένα χρήσης του cluster (CPU,Memory, κτλπ).

Σημείωση: Χρησιμοποιήθηκαν έτοιμα dashboards που ήταν σχετικά με τις ανάγκες μας.

4.7 Υλοποίηση Τοπικού DNS με dnsproxy και CoreDNS (Εκτός Cluster)

Για να επιτευχθεί η επίλυση φιλικών domain names όπως web.local, grafana.local και prometheus.local τόσο τοπικά όσο και απομακρυσμένα μέσω VPN, υλοποιήθηκε ένας **τοπικός μηχανισμός DNS resolving** που βασίζεται στα εργαλεία **dnsproxy** και **CoreDNS**, τα οποία **τρέχουν εκτός του Kubernetes cluster**.

Αυτό το setup εξυπηρετεί τον στόχο της διαφάνειας και της αυτοματοποιημένης επίλυσης DNS χωρίς την ανάγκη για τροποποίηση αρχείων hosts σε κάθε client.

4.7.1 dnsproxy: Lightweight DNS Forwarder

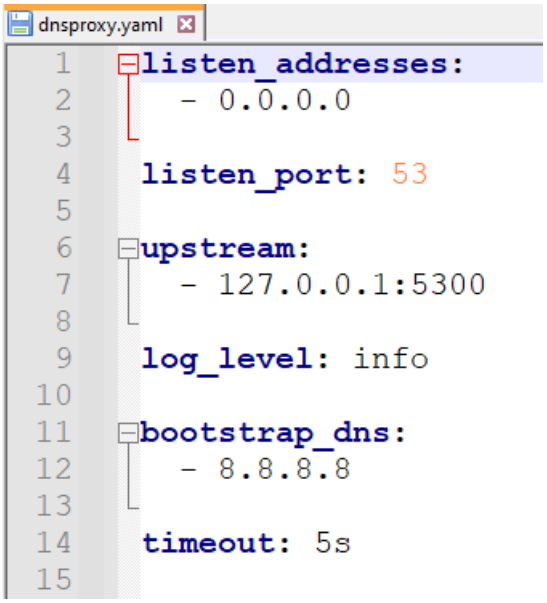
Το εργαλείο **dnsproxy** (της AdGuard) εγκαταστάθηκε ως container στον host. Λειτουργεί ως ενδιάμεσος resolver ο οποίος:

- Ακούει αιτήματα DNS (π.χ. από Tailscale clients)
- Αναγνωρίζει domain names που λήγουν σε .local
- Προωθεί αυτά τα αιτήματα στον **CoreDNS**
- Προωθεί τα υπόλοιπα αιτήματα στο default DNS (π.χ. Tailscale DNS ή Cloudflare)

Πρώτα πρέπει να κατεβάσουμε το αρχείο **dnsproxy-windows-amd64-v0.56.1.zip** απο το github repo της AdGuard Releases · AdguardTeam/dnsproxy

Έπειτα το κάνουμε extract στον φάκελο C:/dnsproxy

Εκεί δημιουργούμε ένα αρχείο **dnsproxy.yaml** (Εικόνα 4.7.1)



Εικόνα 4.7.1 Ρυθμίσεις dnsproxy.

Μετά, τρέχουμε το παρακάτω (Εικόνα 4.7.2):

```
PS C:\> cd .\dnsproxy\
PS C:\dnsproxy> .\dnsproxy.exe --config-path=dnsproxy.yaml
dnsproxy config path: dnsproxy.yaml
2025/06/21 13:51:00.301623 INFO dnsproxy starting version=v0.75.5 revision=f00be4d branch=HEAD commit_time=1746559772
2025/06/21 13:51:00.313131 INFO upstream mode is set prefix=dnsproxy mode=load_balance
2025/06/21 13:51:00.313131 INFO cache disabled prefix=dnsproxy
2025/06/21 13:51:00.313131 INFO starting dns proxy server prefix=dnsproxy
2025/06/21 13:51:00.313131 INFO creating udp server socket prefix=dnsproxy addr=0.0.0.0:53
2025/06/21 13:51:00.314195 INFO listening to udp prefix=dnsproxy addr=[::]:53
2025/06/21 13:51:00.314733 INFO creating tcp server socket prefix=dnsproxy addr=0.0.0.0:53
2025/06/21 13:51:00.315264 INFO listening to tcp prefix=dnsproxy addr=[::]:53
2025/06/21 13:51:00.315264 INFO entering udp listener loop prefix=dnsproxy addr=[::]:53
2025/06/21 13:51:00.315264 INFO entering listener loop prefix=dnsproxy proto=tcp addr=[::]:53
```

Εικόνα 4.7.2 Εκτέλεση dnsproxy.

4.7.2 CoreDNS: Custom Resolver για .local

Ο **CoreDNS** εγκαταστάθηκε επίσης ως container εκτός cluster και ρυθμίστηκε με custom **Corefile**, ώστε:

- Να απαντά μόνο για τις ζώνες .local (Εικόνα 4.7.3)
- Να επιστρέφει στατικές IP (π.χ. του Ingress Controller) για web.local, grafana.local, prometheus.local (Εικόνα 4.7.4)
- Να ακούει μόνο αιτήματα που έρχονται από το dnsproxy

```
docker-compose up -d
```

docker-compose.yaml

Corefile docker-compose.yaml

```yaml
services:
  coredns:
    image: coredns/coredns:latest
    container_name: coredns
    command: -conf /etc/coredns/Corefile
    volumes:
      - ./Corefile:/etc/coredns/Corefile
    ports:
      - "5300:53/udp"
      - "5300:53/tcp"
    restart: unless-stopped


```

Εικόνα 4.7.3 Docker-compose αρχείο για την εγκατάσταση του Coredns

Corefile

Corefile docker-compose.yaml

```
. {
    hosts {
        100.74.6.5 grafana.local web.local prometheus.local
        fallthrough
    }
    forward . 8.8.8.8
    log
    errors
}

```

Εικόνα 4.7.4 Corefile για τον Coredns

Σημείωση: Η IP 100.74.6.5 είναι η IP του tailscale VPN που θα φτιαχτεί αμέσως μετά.

4.8 Ρύθμιση Tailscale VPN: Server στο Cluster – Exit Node στον Host

Για την απομακρυσμένη πρόσβαση στο τοπικό Kubernetes cluster, η εργασία αξιοποίησε το Tailscale, μια zero-config λύση VPN βασισμένη στο WireGuard, κατάλληλη για περιβάλλοντα με περιορισμούς NAT (όπως CGNAT). Η υλοποίηση περιλαμβάνει:

- Tailscale server ως pod εντός του cluster, ώστε όλα τα pods να είναι μέλη του tailnet.
- Tailscale client στον host (Windows), ρυθμισμένο ως exit node, ώστε το cluster να αποκτά πρόσβαση στο ευρύτερο δίκτυο και να μπορεί να επικοινωνεί με τον εξωτερικό DNS resolver (dnsproxy/CoreDNS).

4.8.1 Tailscale Server εντός Cluster

Ο server εγκαταστάθηκε ως Deployment, με τα παρακάτω χαρακτηριστικά:

- *Image: tailscale/tailscale*
- *Όρισμα init: tailscaled και tailscale up*
- *Ενεργοποιημένο --accept-routes για να δέχεται routes από το exit node*

Πρώτα χρειάζεται να στηθούν τα RBAC rules (Εικόνα 4.8.1).Το RBAC, ή Role-Based Access Control, είναι ένα σύστημα διαχείρισης πρόσβασης που επιτρέπει την οργάνωση των χρηστών σε ρόλους και την εκχώρηση δικαιωμάτων με βάση αυτούς τους ρόλους. Στο πλαίσιο του Tailscale (Εικόνα 4.8.2), το RBAC βοηθά στον καθορισμό των επιπέδων πρόσβασης των χρηστών στο δίκτυο, διασφαλίζοντας ότι οι χρήστες έχουν μόνο τα δικαιώματα που χρειάζονται για την εκτέλεση των καθηκόντων τους, βελτιώνοντας έτσι την ασφάλεια.

tailscale-rbac.yaml

tailscale-rbac.yaml | tailscale.yaml

```yaml
apiVersion: v1
kind: ServiceAccount
metadata:
  name: tailscale
  namespace: tailscale
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: tailscale
  namespace: tailscale
rules:
- apiGroups: [""]
  resources: ["secrets"]
  resourceNames: ["tailscale-state"]
  verbs: ["get", "update", "create"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: tailscale
  namespace: tailscale
subjects:
- kind: ServiceAccount
  name: tailscale
  namespace: tailscale
roleRef:
  kind: Role
  name: tailscale
  apiGroup: rbac.authorization.k8s.io

```

Εικόνα 4.8.1 RBACK Rules για το Tailscale

tailscale.yaml

tailscale-rbac.yaml | tailscale.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: tailscale
  namespace: tailscale
spec:
  serviceAccountName: tailscale
  hostNetwork: true
  dnsPolicy: ClusterFirstWithHostNet
  containers:
    - name: tailscale
      image: tailscale/tailscale:latest
      securityContext:
        capabilities:
          add: ["NET_ADMIN", "SYS_MODULE"]
        privileged: true
      env:
        - name: TS_AUTHKEY
          valueFrom:
            secretKeyRef:
              name: tailscale-auth
              key: TS_AUTHKEY
        - name: TS_ROUTES
          value: "192.168.1.0/24"
        - name: TS_ADVERTISE_TAGS
          value: "tag:k8s"
        - name: TS_KUBE_SECRET
          value: "tailscale-state"
        - name: TAILSCALE_SOCKET
          value: "/tmp/tailscaled.sock"
      command: ["/bin/sh", "-c"]
      args:
        - |
          tailscaled --state=kube:tailscale-state --socket=/tmp/tailscaled.sock --tun=userspace-networking &
          sleep 5
          tailscale --socket=/tmp/tailscaled.sock up --auth-key=$TS_AUTHKEY \
            --accept-routes \
            --exit-node=100.74.6.5 \
            --exit-node-allow-lan-access=true \
            --accept-dns=false \
            --advertise-tags=$TS_ADVERTISE_TAGS
          wait
  restartPolicy: Always
```

Εικόνα 4.8.2 Δημιουργία tailscale pod εντός του cluster και ρυθμίσεις.

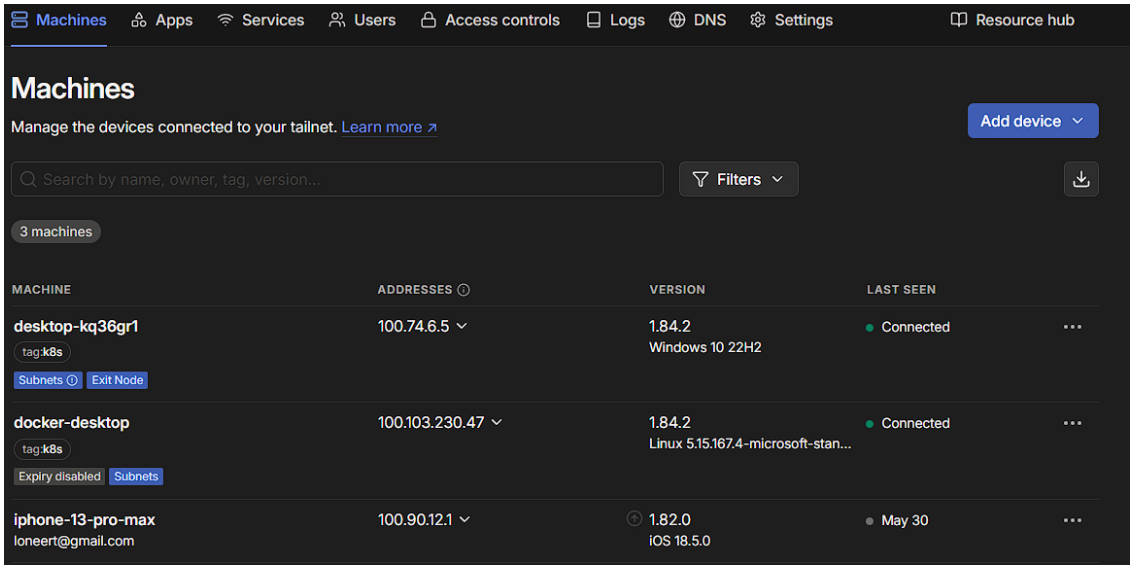
4.8.2 Ρύθμιση Host ως Exit Node

Tailscale admin console

Εδώ φαίνονται όλοι οι χρήστες που έχουν συνδεθεί στο Tailscale VPN (Εικόνα 4.8.3) και φαίνεται οτι και το host μηχάνημα (desktop-kq36gr1) έχει ρυθμιστεί και σαν exit node. Από το admin panel του Tailscale, εγκρίθηκε ο host ως exit node και ο server pod στο cluster επιλέχθηκε να χρησιμοποιεί το συγκεκριμένο exit node.

Με αυτόν τον τρόπο:

- Το pod αποκτά πρόσβαση στο Internet μέσω του host
- Μπορεί να φτάσει τον dnsproxy/CoreDNS (που είναι στον host)
- Εξασφαλίζεται λειτουργία DNS resolving και προσβασιμότητα των υπηρεσιών



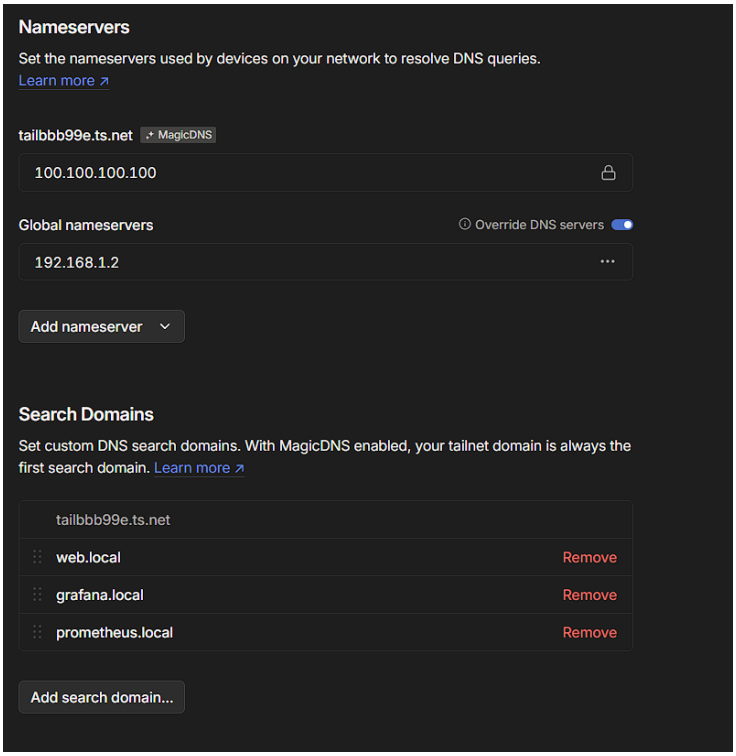
Εικόνα 4.8.3 Tailscale Dashboard με τις μηχανές συνδεδεμένες στο VPN

Tailscale DNS Settings

4.8.3 Routes και DNS

Στο Tailscale panel ορίστηκε Split DNS (Εικόνα 4.8.4):

- Domain: local
- Nameserver: IP του host (Tailscale IP)
- Ορίστηκε επίσης route προς 192.168.1.0/24 για πρόσβαση στα τοπικά IPs



Εικόνα 4.8.4 Tailscale DNS settings

4.9 Ορισμός Ingress Resources και Ρύθμιση Metrics για Monitoring

Για την πρόσβαση στις επιμέρους υπηρεσίες του cluster μέσω HTTPS, δημιουργήθηκαν ξεχωριστά **Ingress resources**, τα οποία λειτουργούν κάτω από τον κοινό **Nginx Ingress Controller**. Παράλληλα, παραμετροποιήθηκαν exporters και service monitors ώστε το Prometheus να συλλέγει metrics από τον Ingress Controller και τη βάση MySQL, τα οποία στη συνέχεια αποτυπώνονται σε dashboards του Grafana.

4.9.1 Δημιουργία Ingress για Web Application

Το αρχείο web-ingress.yaml (Εικόνα 4.9.1) ορίζει την πρόσβαση στην κύρια web εφαρμογή:

- Domain: web.local
- Path: /
- TLS ενεργοποιημένο με self-signed certificate
- Ανακατεύθυνση στον αντίστοιχο Kubernetes service του web server

web-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: web-ingress
  namespace: default
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/limit-connections: "10"
    nginx.ingress.kubernetes.io/limit-rpm: "60"
    nginx.ingress.kubernetes.io/limit-rps: "5"
    nginx.ingress.kubernetes.io/limit-burst-multiplier: "3"
    nginx.ingress.kubernetes.io/limit-rate-after: "5"
    nginx.ingress.kubernetes.io/limit-rate: "500"
spec:
  tls:
  - hosts:
    - web.local
    secretName: ingress-nginx-tls
  rules:
  - host: web.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: web-server-service
            port:
              number: 80
```

Εικόνα 4.9.1 Δημιουργία ingress για την web εφαρμογή

4.9.2 Ingress για Grafana και Prometheus UI

Στο αρχείο monitor-ingress.yaml (Εικόνα 4.9.2) δηλώθηκαν τα Ingress rules για:

- grafana.local → UI του Grafana
- prometheus.local → UI του Prometheus

Και τα δύο domains προστατεύονται με HTTPS και δρομολογούνται στον αντίστοιχο service port.

monitor-ingress.yaml

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: monitoring-ingress
  namespace: monitoring
  annotations:
    kubernetes.io/ingress.class: "nginx"
    nginx.ingress.kubernetes.io/ssl-redirect: "true"
    nginx.ingress.kubernetes.io/limit-connections: "10"
    nginx.ingress.kubernetes.io/limit-rpm: "60"
    nginx.ingress.kubernetes.io/limit-rps: "5"
    nginx.ingress.kubernetes.io/limit-burst-multiplier: "3"
    nginx.ingress.kubernetes.io/limit-rate-after: "5"
    nginx.ingress.kubernetes.io/limit-rate: "500"
spec:
  tls:
  - hosts:
    - grafana.local
    - prometheus.local
    secretName: ingress-tls
  rules:
  - host: grafana.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: grafana
            port:
              number: 80
  - host: prometheus.local
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: prometheus-kube-prometheus-prometheus
            port:
              number: 9090

```

Εικόνα 4.9.1 Δημιουργία ingress για Grafana και Prometheus

4.9.3 Ενεργοποίηση Metrics για Nginx Ingress Controller

Για να συλλέγονται μετρικές από τον Ingress Controller:

- Δημιουργήθηκε το ingress-nginx-metrics-svc.yaml (Εικόνα 4.9.3) για να εκθέσει /metrics endpoint
- Ορίστηκε ServiceMonitor μέσω ingress-nginx-servicemonitor.yaml (Εικόνα 4.9.4), που επιτρέπει στον Prometheus να «scrape» τα metrics
- Η Grafana μπορεί να απεικονίζει στατιστικά όπως requests/second, latency, HTTP status codes, κ.ά.

my-mysql-metrics.yaml | ingress-nginx-metrics-svc.yaml | ingress-nginx-servicemonitor

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx-metrics
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/component: controller
spec:
  type: ClusterIP
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/component: controller
  ports:
    - name: metrics
      port: 10254
      targetPort: 10254
      protocol: TCP

```

Εικόνα 4.9.3 Δημιουργία metrics endpoint για ingress controller

my-mysql-metrics.yaml | ingress-nginx-metrics-svc.yaml | ingress-nginx-servicemonitor.yaml

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: ingress-nginx
  namespace: monitoring
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: ingress-nginx
  namespaceSelector:
    matchNames:
      - ingress-nginx
  endpoints:
    - port: metrics
      interval: 15s

```

Εικόνα 4.9.4 Δημιουργία servicemonitor για να επιτραπεί το scraping από το Prometheus.

4.9.4 Ενεργοποίηση MySQL Exporter για Prometheus

Με το my-mysql-metrics.yaml (Εικόνα 4.9.5) έγινε deploy ένας **Prometheus MySQL Exporter**, ο οποίος:

- Συνδέεται στο master της βάσης
- Συλλέγει πληροφορίες όπως queries per second, active connections, innodb usage
- Στέλνει δεδομένα στον Prometheus server

my-mysql-metrics.yaml | ingress-nginx-metrics-svc.yaml | ingress-nginx-servicemonitc

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: mysql-servicemonitor
  namespace: default
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: mysql
  namespaceSelector:
    matchNames:
    - default
  endpoints:
  - port: metrics
    interval: 15s
```

Εικόνα 4.9.5 Δημιουργία metrics endpoint για την mysql

4.10 Προσαρμογή Web Container και Metrics μέσω Nginx Stub Status

Ο web server που φιλοξενεί την κύρια εφαρμογή (chat) έχει χτιστεί ως custom Docker image, το οποίο βασίζεται σε PHP και Nginx. Εκτός από τη βασική λειτουργικότητα, έγινε και παραμετροποίηση ώστε να προσφέρει **ενδείξεις λειτουργίας (metrics)** μέσω του endpoint /nginx_status, αξιοποιήσιμες από τον Prometheus.

4.10.1 Δομή του Dockerfile

Ο Dockerfile που χρησιμοποιήθηκε (Εικόνα 4.10.1):

- Εγκαθιστά PHP και Nginx
- Αντιγράφει την εφαρμογή (index.php)
- Ρυθμίζει custom default.conf για το virtual host
- Εκκινεί τον php-fpm και nginx με custom entrypoint script

Dockerfile

```
FROM php:8.2-fpm

RUN apt-get update && apt-get install -y nginx \
    && docker-php-ext-install mysqli pdo pdo_mysql

COPY default.conf /etc/nginx/conf.d/default.conf
COPY index.php /var/www/html/index.php
COPY start2.sh /start.sh

RUN chmod +x /start.sh
RUN ls -l /start.sh && file /start.sh

EXPOSE 80

CMD ["/start.sh"]

```

Εικόνα 4.10.1 Dockerfile δημιουργία image για το indext.php

4.10.2 Entry Point Script

Το αρχείο start2.sh (Εικόνα 4.10.2) εκτελεί:

1. Το php-fpm στο background
2. Το nginx σε foreground mode (ώστε να διατηρηθεί το container ενεργό)

start2.sh

```
#!/bin/sh

# Ξεκίνα το php-fpm στο background
php-fpm &

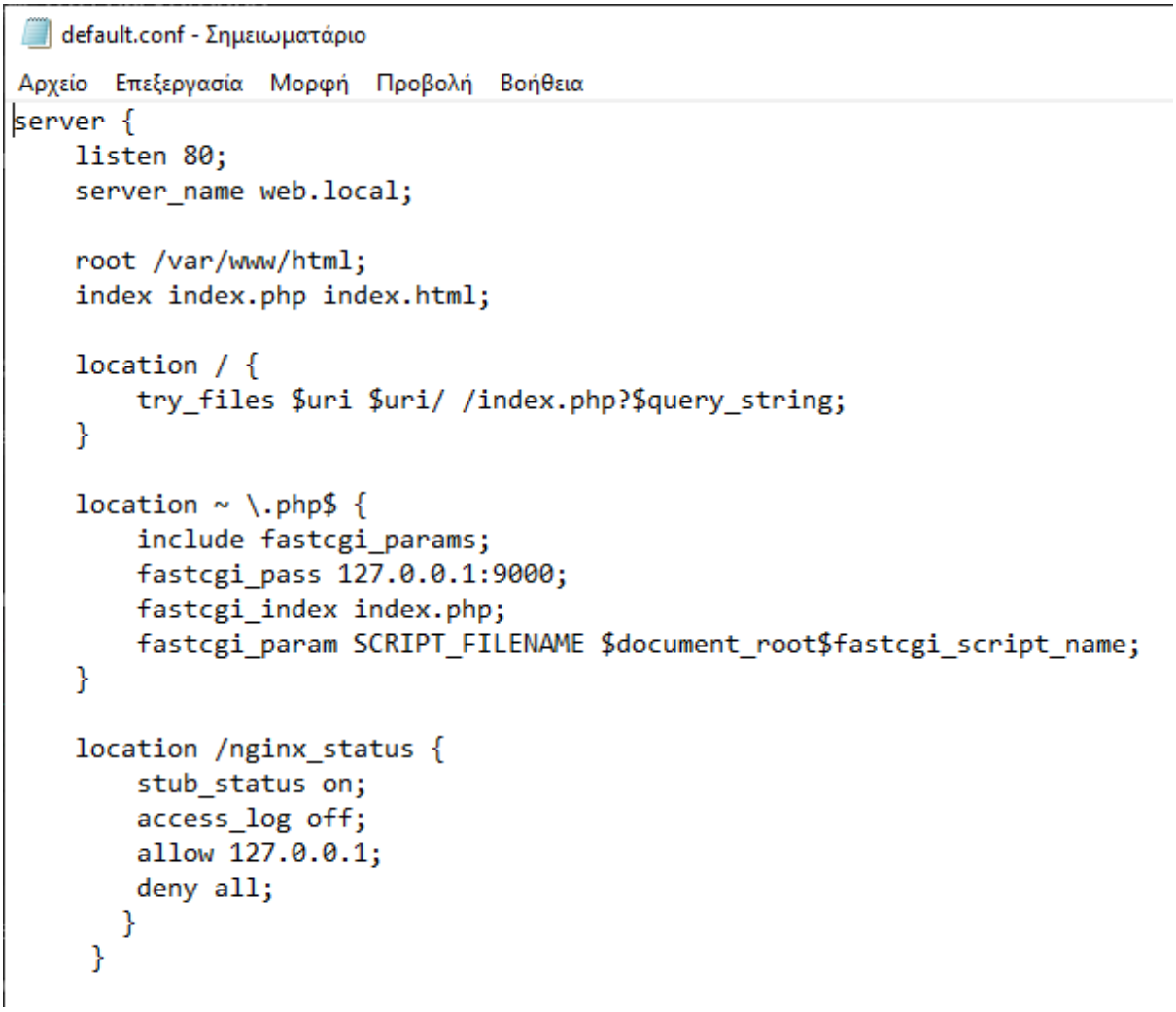
# Ξεκίνα το nginx στο foreground για να μην τερματίσει το container
nginx -g "daemon off;"

```

Εικόνα 4.10.2 Αρχικοποίηση php-fpm

4.10.3 Nginx Stub Status

Στο αρχείο default.conf (Εικόνα 4.10.3), υπάρχει η παρακάτω ενότητα:



Εικόνα 4.10.3 Default.conf επιτρέπει την έκθεση βασικών ενδείξεων του nginx.

Αυτό επιτρέπει την έκθεση των βασικών ενδείξεων του Nginx (π.χ. active connections, handled requests), τοπικά στο container. Το Prometheus μπορεί να συλλέξει αυτές τις τιμές είτε με sidecar exporter είτε με nginx-prometheus-exporter.

4.10.4 Περιεχόμενο Εφαρμογής

Το βασικό αρχείο index.php (Εικόνα 4.10.4) υλοποιεί το interface ενός απλού chat system:

- Δέχεται μηνύματα μέσω HTML μορφής
- Τα αποθηκεύει σε MySQL μέσω σύνδεσης με PDO
- Εμφανίζει τα τελευταία καταχωρημένα μηνύματα με timestamp

index.php

```php
<?php
echo "Served by pod: " . gethostname();
$servername = getenv('MYSQL_HOST') ?: 'my-mysql-primary';
$username = getenv('MYSQL_USER') ?: 'root';
$password = getenv('MYSQL_PASSWORD') ?: 'yourRootPassword';
$dbname = getenv('MYSQL_DATABASE') ?: 'mydatabase';

$conn = new mysqli($servername, $username, $password, $dbname);
if ($conn->connect_error) {
    die("Connection failed: " . $conn->connect_error);
}

if ($_SERVER['REQUEST_METHOD'] == 'POST' && !empty($_POST['message'])) {
    $msg = $conn->real_escape_string($_POST['message']);
    $sql = "INSERT INTO messages (content) VALUES ('$msg')";
    if ($conn->query($sql) === TRUE) {
        echo "New record created successfully.<br>";
    } else {
        echo "Error: " . $conn->error . "<br>";
    }
}
?>

<form method="POST" action="">
    <label for="message">Message:</label>
    <input type="text" id="message" name="message" required>
    <input type="submit" value="Submit">
</form>

<?php
$result = $conn->query("SELECT id, content FROM messages ORDER BY id DESC");
if ($result->num_rows > 0) {
    echo "<ul>";
    while($row = $result->fetch_assoc()) {
        echo "<li>" . htmlspecialchars($row['content']) . "</li>";
    }
    echo "</ul>";
} else {
    echo "No messages found.";
}
$conn->close();
?>
```

Εικόνα 4.10.5 index.php είναι το landing page για το chat.

4.10.5 Δημιουργία του Docker Image και Αποθήκευση στο Τοπικό Docker Registry

Μετά την προετοιμασία όλων των απαραίτητων αρχείων (Dockerfile, index.php, default.conf, start2.sh), έγινε build του custom Docker image με την εξής εντολή:

`docker build -t voulgaris/web-php-nginx.v2:latest .`

Το image κατασκευάζεται τοπικά και στη συνέχεια χρησιμοποιείται ως βάση για το Kubernetes Deployment που δημιουργεί τα web server pods. Καθώς η υλοποίηση είναι εξ ολοκλήρου τοπική (on-premise), το image αποθηκεύεται στο **Docker Engine** του host που διαχειρίζεται το cluster μέσω Docker Desktop και είναι άμεσα προσβάσιμο για χρήση στο kubelet.

4.10.6 Παρακολούθηση του Web Server μέσω Prometheus Exporter για Nginx

Για να εξασφαλιστεί η **πλήρης παρακολούθηση** της απόδοσης του web server, υλοποιήθηκε ένας **Nginx Prometheus Exporter**, ο οποίος συλλέγει και εκθέτει στατιστικά από το endpoint stub_status του Nginx σε μορφή συμβατή με Prometheus.

Αυτό επιτρέπει την **ανεξάρτητη παρακολούθηση του web server**, πέρα από το monitoring που γίνεται μέσω του Ingress Controller.

Nginx Configuration

Δύο αρχεία ConfigMap (ή inline mounted configs) ρυθμίζουν το περιβάλλον του Nginx:

- nginx-config.yaml για HTTP-only (Εικόνα 4.10.6)
- nginx-config-https.yaml για HTTPS με self-signed certificate (Εικόνα 4.10.7)

Και τα δύο περιέχουν οδηγίες όπως:

```
location /nginx_status {
  stub_status on;
  access_log off;
  allow 127.0.0.1;
  deny all;
}
```

Αυτό επιτρέπει την πρόσβαση στο /nginx_status endpoint μόνο τοπικά (όπως απαιτείται από τον exporter).

nginx-config.yaml | nginx-config-https.yaml | nginx-exporter-deployment.yaml | nginx-exporter-service.yaml | nginx-service

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config
  namespace: default
data:
  default.conf: |
    server {
      listen 80 default_server;
      server_name web.local;

      root /var/www/html;
      index index.php index.html;

      location / {
        try_files $uri $uri/ /index.php?$query_string;
      }

      location ~ \.php$ {
        include fastcgi_params;
        fastcgi_pass 127.0.0.1:9000;
        fastcgi_index index.php;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
      }

      location /nginx_status {
        stub_status on;
        access_log off;
        allow 127.0.0.1;
        deny all;
      }
    }
```

Εικόνα 4.10.6 Configmap για ρύθμιση περιβάλλοντος Nginx (HTTP)

nginx-config.yaml | nginx-config-https.yaml | nginx-exporter-deployment.yaml | nginx-exporter-service.yaml | nginx-servicemon

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nginx-config-https
  namespace: default
data:
  default.conf: |
        server {
            listen 443 ssl;
            server_name localhost;

            ssl_certificate /path/to/cert.pem;
            ssl_certificate_key /path/to/key.pem;

            root /var/www/html;
            index index.php index.html;

            location / {
            try_files $uri $uri/ /index.php$is_args$args;
            }

            location ~ \.php$ {
            include fastcgi_params;
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
            fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
            }

            location /nginx_status {
            stub_status on;
            access_log off;
            allow 127.0.0.1;
            deny all;
            }
        }
```

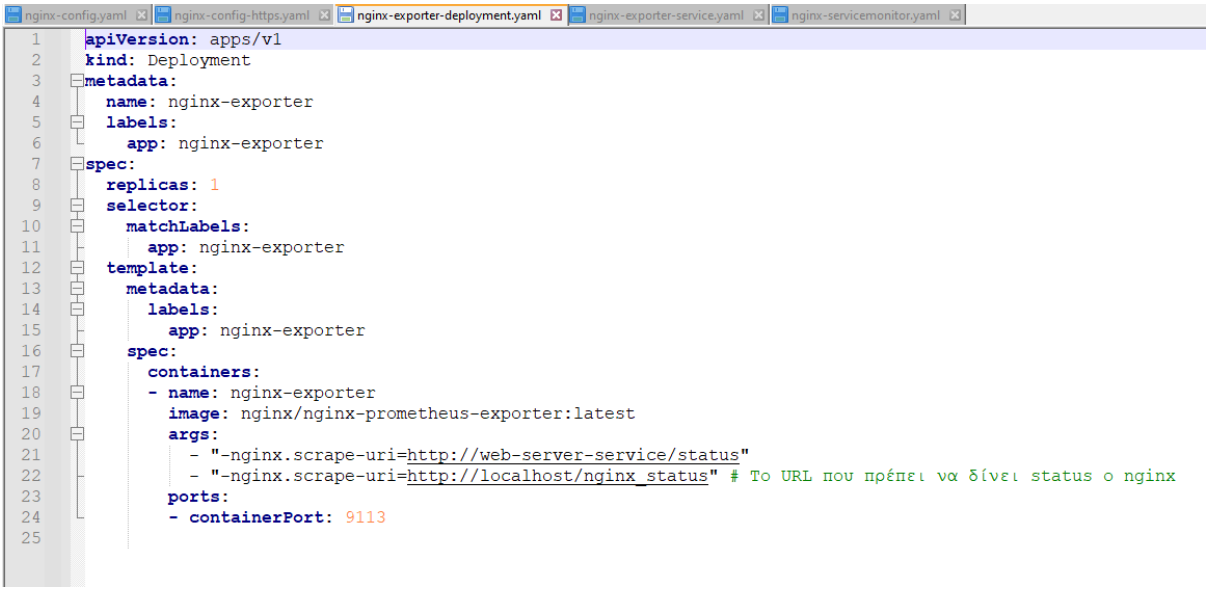
Εικόνα 4.10.7 Configmap για ρύθμιση περιβάλλοντος Nginx (HTTPS)

Exporter Deployment

Ο Prometheus Exporter για Nginx αναπτύχθηκε μέσω του αρχείου nginx-exporter-deployment.yaml (Εικόνα 4.10.8). Ο exporter:

- Τρέχει σε ξεχωριστό pod
- Συνδέεται στο web server (ή Nginx Ingress, ανάλογα τη ρύθμιση)
- Μεταφράζει τα δεδομένα από το /nginx_status σε Prometheus metrics

Χρησιμοποιεί το image nginx/nginx-prometheus-exporter.



Εικόνα 4.10.8 Nginx exporter για Prometheus

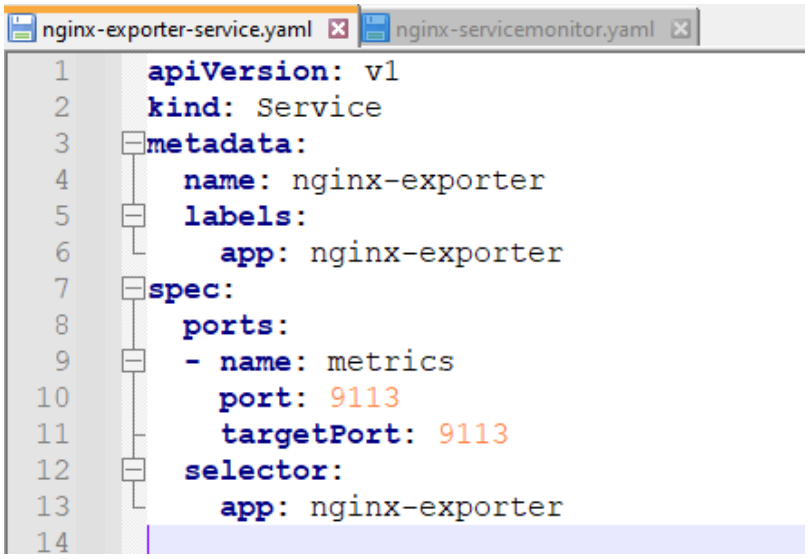
Exporter Service & Monitoring

Δημιουργήθηκε service nginx-exporter-service.yaml (Εικόνα 4.10.9) τύπου ClusterIP ώστε να εκτίθεται το /metrics endpoint.

Στη συνέχεια, με τον nginx-servicemonitor.yaml (Εικόνα 4.10.10) δηλώθηκε το service στον Prometheus Operator για να αρχίσει το scraping.

Με αυτό τον τρόπο, τα metrics εμφανίζονται στα dashboards του Grafana, και καλύπτουν ενδείξεις όπως:

- Active connections
- Accepted/handled requests
- Current reading/writing/waiting states
- Requests per second



Εικόνα 4.10.9 Απόδοση cluster IP στον nginx exporter.

nginx-exporter-service.yaml | nginx-servicemonitor.yaml

```yaml
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: nginx-exporter
  labels:
    release: prometheus
spec:
  selector:
    matchLabels:
      app: nginx-exporter
  endpoints:
  - port: metrics
    interval: 15s
```

Εικόνα 4.10.10 Servicemonitor για να γίνει expose για scraping προς το Prometheus.

4.10.7 Ορισμός Πολιτικών Πόρων με LimitRange

Για την αποτροπή υπερκατανάλωσης πόρων από pods και containers στο cluster, ορίστηκαν αντικείμενα τύπου LimitRange. Αυτά καθορίζουν προκαθορισμένα **όρια χρήσης CPU και RAM** για pods και containers που δεν καθορίζουν δικά τους resources.

Έγινε χρήση δύο LimitRange objects:

limitrange-default.yaml (Εικόνα 4.10.11)

Εφαρμόζεται στο **προεπιλεγμένο namespace**, και περιλαμβάνει:

- Default requests και limits για CPU και RAM
- Καλύπτει όλα τα pods που δημιουργούνται χωρίς explicit δηλώσεις πόρων

Έτσι διασφαλίζεται ότι ακόμη και pods χωρίς resources εντολές δεν θα μπορούν να υπερφορτώσουν το node.

limitrange-default.yaml | limitrange-monitoring.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: default-resource-limits
  namespace: default
spec:
  limits:
    - default:
        cpu: 300m
        memory: 750Mi
      defaultRequest:
        cpu: 300m
        memory: 500Mi
      type: Container
```

Εικόνα 4.10.11 Δημιουργία limitrange (ορίου κατανάλωσης πόρων) για το default namespace.

limitrange-monitoring.yaml (Εικόνα 4.10.12)

Αφορά το namespace monitoring (ή παρόμοιο) που περιέχει τα pods του Prometheus, Grafana και exporters.

Διασφαλίζει ότι:

- Τα monitoring εργαλεία δεν θα καταναλώσουν υπερβολικούς πόρους
- Υπάρχει ελεγχόμενο όριο για τις υπηρεσίες παρακολούθησης, ώστε να διασφαλιστεί η σταθερότητα του συστήματος

limitrange-default.yaml | limitrange-monitoring.yaml

```yaml
apiVersion: v1
kind: LimitRange
metadata:
  name: monitoring-resource-limits
  namespace: monitoring
spec:
  limits:
    - default:
        cpu: 300m
        memory: 750Mi
      defaultRequest:
        cpu: 100m
        memory: 500Mi
      type: Container
```

Εικόνα 4.10.12 Δημιουργία limitrange (ορίου κατανάλωσης πόρων) για το monitoring namespace.

Η χρήση LimitRange σε συνδυασμό με HPA (Horizontal Pod Autoscaler) επιτρέπει τον **έλεγχο και τη δυναμική κλιμάκωση των pods με ασφαλή τρόπο**.

4.11 Ορισμός ενός Horizontal Pod Autoscaler (HPA)

4.11.1 Horizontal Pod Autoscaler (HPA)

Ορίστηκε HPA για το deployment του web server (Εικόνα 4.11.1), με τις εξής ρυθμίσεις:

- **Στόχος χρήσης CPU:** 30%
- **Ελάχιστος αριθμός pods:** 2
- **Μέγιστος αριθμός pods:** 5

Αυτό σημαίνει ότι όταν η μέση χρήση CPU στα υπάρχοντα pods υπερβεί το 30%, θα προστεθεί νέο pod, έως τον μέγιστο αριθμό 5. Αντίστοιχα, όταν το φορτίο πέσει, τα pods μειώνονται.

web-server-hpa.yaml

```yaml
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: web-server-hpa
  namespace: default
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: web-server
  minReplicas: 2
  maxReplicas: 5
  metrics:
  - type: Resource
    resource:
      name: cpu
      target:
        type: Utilization
        averageUtilization: 40
```

Εικόνα 4.11.1 Δημιουργία HPA για τον web-server (έπειτα αλλάχτηκε σε 30% για λόγους testing)

4.11.2 Stress Testing με Pods

Για τη δοκιμή της αντίδρασης του HPA, χρησιμοποιήθηκαν pods τύπου stress, βασισμένα στο εργαλείο stress-ng (Εικόνα 4.11.2). Τα pods εκτελούσαν συνεχείς υπολογισμούς ώστε να αυξήσουν την κατανάλωση CPU στον web server.

stress-test.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: stress-test
spec:
  containers:
    - name: stress
      image: polinux/stress
      command: ["stress"]
      args: ["--cpu", "2", "--vm", "1", "--vm-bytes", "600M", "--timeout", "60s"]
      resources:
        requests:
          memory: "500Mi"
          cpu: "100m"
        limits:
          memory: "750Mi"
          cpu: "500m"
```

Εικόνα 4.11.2 Stress test με το εργαλείο stress-ng.

4.11.3 Stress Testing Δικτύου με iperf3

Για τη δοκιμή του **δικτυακού throughput και latency**, εγκαταστάθηκε το εργαλείο iperf3 σε μορφή:

- Ένα pod ως server και το service του (iperf3 -s) (Εικόνα 4.11.3 , 4.11.4)
- Ένα pod ως client (iperf3 -c <ip του server>)

Αυτό επέτρεψε τη μέτρηση της δικτυακής επίδοσης **εντός του cluster**, και αποκάλυψε την ικανότητα **load balancing του Nginx Ingress Controller και του MetalLB** σε επίπεδο L4.

iperf3-server.yaml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: iperf3-server
  labels:
    app: iperf3
spec:
  replicas: 2
  selector:
    matchLabels:
      app: iperf3
  template:
    metadata:
      labels:
        app: iperf3
    spec:
      containers:
      - name: iperf3
        image: networkstatic/iperf3
        command: [ "sh", "-c", "echo I am $(hostname); iperf3 -s -V" ]
        ports:
        - containerPort: 5201
```

Εικόνα 4.11.3 Δημιουργία iperf3 server pod εντός του cluster

iperf3-service.yaml

```yaml
# iperf3-service.yaml
apiVersion: v1
kind: Service
metadata:
  name: iperf3-service
spec:
  selector:
    app: iperf3
  ports:
  - protocol: TCP
    port: 5201
    targetPort: 5201
```

Εικόνα 4.11.4 Δημιουργία iperf3 server service εντός του cluster

4.11.4 Παρακολούθηση κατά τη διάρκεια Stress Test

Κατά τη διάρκεια των δοκιμών:

- **Το Grafana** εμφάνιζε real-time γραφήματα CPU χρήσης, pods count, και latency
- **Το Prometheus** κατέγραφε τα metrics από όλα τα layers (pods, ingress, node)

Μέσω αυτών:

- Παρατηρήθηκε η **αυτόματη δημιουργία νέων pods** από το HPA όταν αυξανόταν το φορτίο
- Καταγράφηκαν **αιχμές δικτύου** και επεξεργαστικής χρήσης
- Διασφαλίστηκε ότι η εφαρμογή παρέμενε **λειτουργική και σταθερή** υπό πίεση

Σημείωση: Για την παρακολούθηση των pods από τον HPA, έχει εγκατασταθεί και ρυθμιστεί **Kubernetes Metrics Server**, ο οποίος συλλέγει δεδομένα όπως CPU και memory utilization σε πραγματικό χρόνο από το kubelet κάθε node. Η παρουσία του είναι απαραίτητη για τη λειτουργία του HPA καθώς αποτελεί την πηγή μετρικών (metrics API) πάνω στις οποίες βασίζεται η απόφαση για scaling.

4.11.5 Περιορισμοί Ρυθμού (Rate Limits) σε Ingress

Για την αποτροπή υπερβολικών αιτημάτων σε κρίσιμες υπηρεσίες (όπως το web interface ή τα monitoring dashboards), εφαρμόστηκαν **rate limiting κανόνες** στο επίπεδο του Ingress Controller. Οι περιορισμοί αυτοί αποτρέπουν την **κατάρρευση των pods από κακόβουλη ή υπερβολική χρήση** και αποτελούν βασική πρακτική hardening ενός cluster.

Rate Limits στο web-ingress.yaml

Στο Ingress της κύριας εφαρμογής (web.local), εφαρμόστηκαν annotations όπως:

```
nginx.ingress.kubernetes.io/limit-connections: "20"
nginx.ingress.kubernetes.io/limit-rpm: "60"
nginx.ingress.kubernetes.io/limit-rps: "10"
```

Αυτά σημαίνουν:

- **Μέγιστες ταυτόχρονες συνδέσεις ανά IP:** 20
- **Μέγιστος αριθμός requests ανά λεπτό:** 60
- **Μέγιστος αριθμός requests ανά δευτερόλεπτο:** 10

Οι τιμές αυτές επιτρέπουν μια φυσιολογική χρήση, αλλά μπλοκάρουν κακόβουλα scripts, DoS ή bots.

Rate Limits στο monitor-ingress.yaml

Παρόμοιοι περιορισμοί τέθηκαν και στο Ingress που εκθέτει το Grafana και το Prometheus (π.χ. grafana.local, prometheus.local):

```
nginx.ingress.kubernetes.io/limit-rps: "5"
nginx.ingress.kubernetes.io/limit-burst-multiplier: "3"
```

Αυτό περιορίζει την υπερβολική χρήση των monitoring dashboards, κάτι χρήσιμο ειδικά σε περιβάλλοντα με περιορισμένους πόρους.

Οι περιορισμοί αυτοί εφαρμόστηκαν μέσω annotations του Nginx Ingress Controller και ισχύουν **σε επίπεδο εισόδου (Layer 7)**.

Με αυτόν τον τρόπο ενισχύεται η **σταθερότητα και ασφάλεια** της υποδομής, ειδικά υπό συνθήκες stress testing.

Κεφάλαιο 5: Τρόπος Λειτουργίας και Αξιολόγηση Λειτουργίας Συστήματος

5.1 Διεπαφές Χρήστη και Παρεχόμενες Δυνατότητες

Η υλοποίηση παρέχει πρόσβαση σε τρία βασικά περιβάλλοντα χρήστη μέσω HTTPS:

- **Web Application** στο https://web.local: Παρέχει ένα απλό σύστημα αποστολής και εμφάνισης μηνυμάτων (τύπου chat), το οποίο αποθηκεύει τα δεδομένα σε MySQL database. Ο χρήστης μπορεί να:
  - Πληκτρολογήσει και στείλει μήνυμα
  - Δει όλα τα μηνύματα σε πίνακα με χρονοσφραγίδα
  - Αλληλεπιδράσει με ένα responsive web interface βασισμένο σε PHP & Nginx
- **Grafana Dashboard** στο https://grafana.local:
  - Δυνατότητα παρακολούθησης real-time μετρικών από Ingress, pods, node και services
  - Εμφάνιση ειδικών dashboards για web server & Nginx exporter
  - Εμφάνιση HPA scaling (pods count ανά στιγμή)
- **Prometheus UI** στο https://prometheus.local:
  - Επιτρέπει αναζητήσεις σε raw metrics
  - Παροχή γραφημάτων με queries (π.χ. rate(http_requests_total))
  - Δοκιμή targets και alert rules

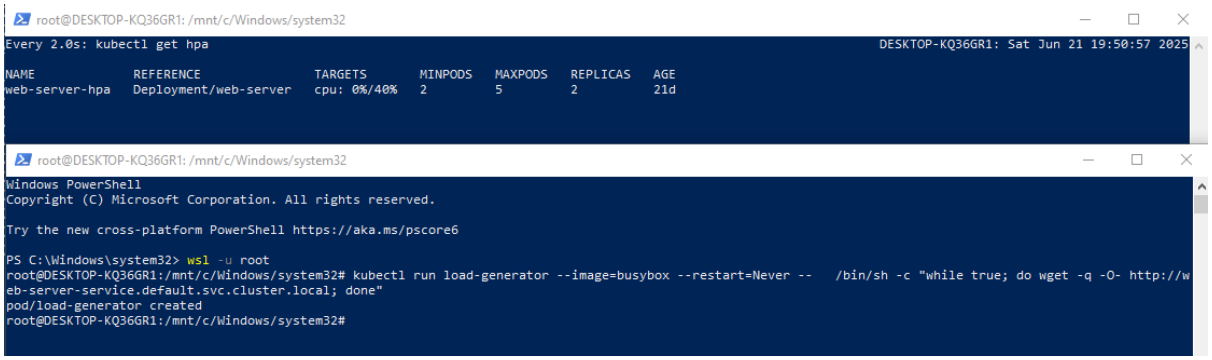
Η πρόσβαση σε όλα τα παραπάνω γίνεται μέσω Tailscale VPN και DNS resolving από CoreDNS μέσω dnsproxy, όπως περιγράφηκε στο Κεφάλαιο 3.

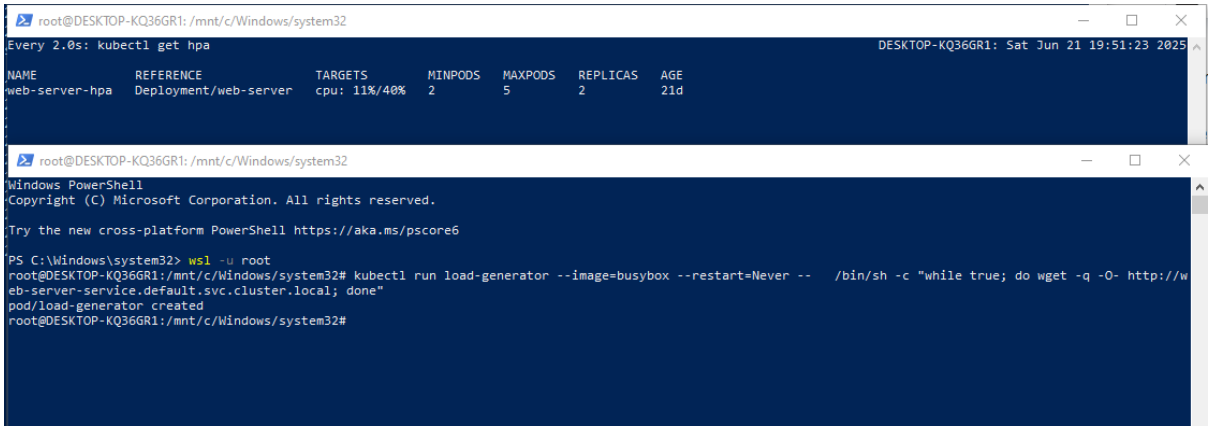
5.2 Αξιολόγηση Επιδόσεων και Πειράματα

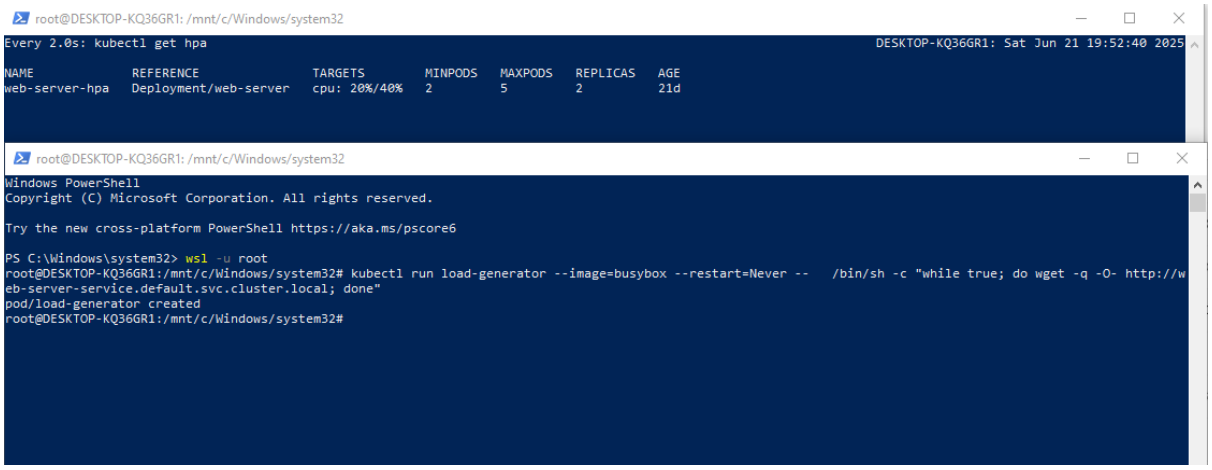
Για την ποσοτική αξιολόγηση της αξιοπιστίας και της προσαρμοστικότητας του συστήματος, πραγματοποιήθηκαν στοχευμένα πειράματα.

5.2.1 Πείραμα: Κατανάλωση CPU & Scaling μέσω HPA

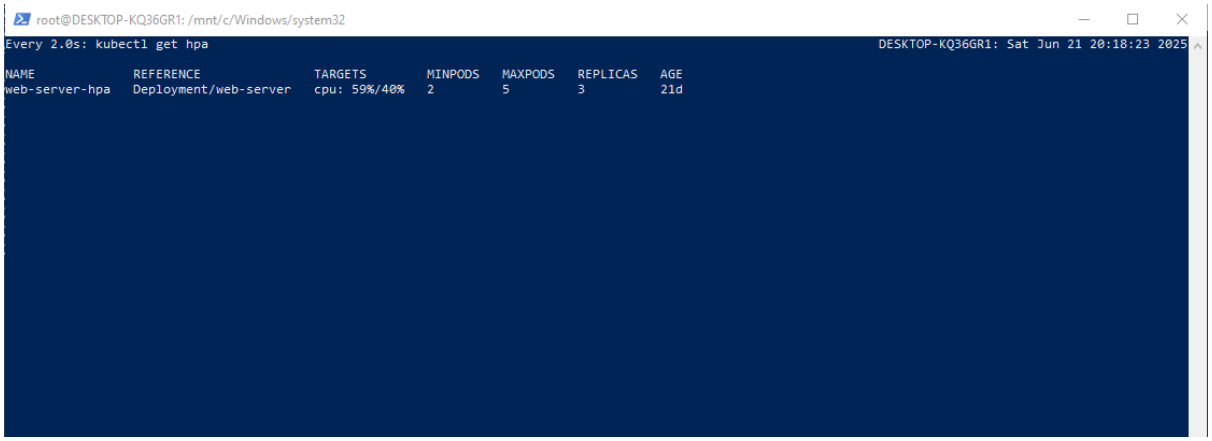
- **Σενάριο**: Ένα pod τύπου stress χρησιμοποιήθηκε για να προκαλέσει spike στη CPU του web server.
- **Παρατήρηση**: Ο Horizontal Pod Autoscaler ενεργοποιήθηκε και κλιμάκωσε τα pods από 2 σε 3 μέσα σε 5 λεπτά. Στις παρακάτω εικόνες φαίνεται η αλλαγή στον αριθμό των replicas σύμφωνα με το χρόνο πάνω δεξιά.
- **Δεδομένα**:
  - Χρήση CPU στο pod: 55-60% avg
  - Μέση απόκριση HPA: ~40 sec

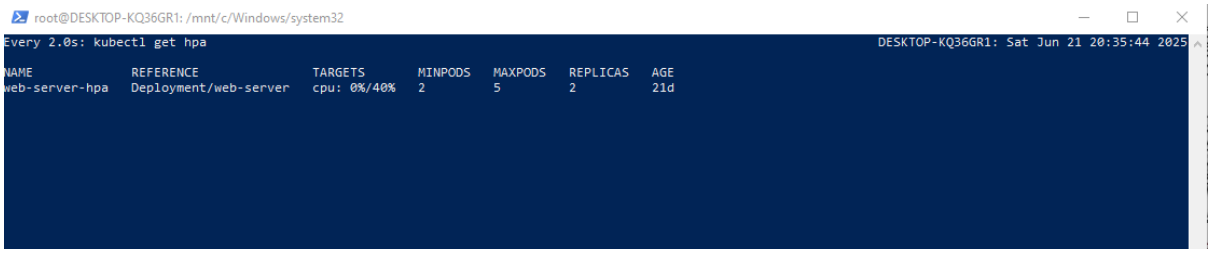




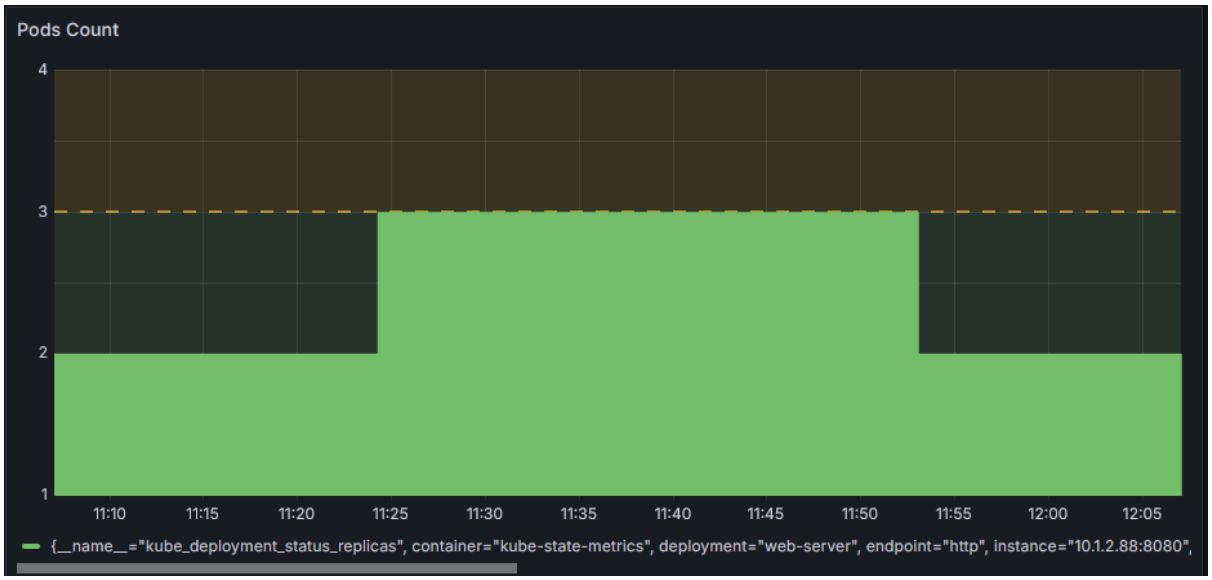


Εδώ χρειαστήκαμε παραπάνω απο ένα load generators (3 στο σύνολο) διότι είναι αρκετά ελαφρύς ο web-server μας και δεν ανέβαζε το επιθυμητό CPU percentage έτσι ώστε να φανεί το stress test που προσπαθούσαμε να κάνουμε.





Παρακάτω βλέπουμε και το γράφημα του στο Grafana που δημιουργήθηκε γι' αυτο το σκοπό. (Εικόνα 5.2.1)



Εικόνα 5.2.1

Γράφημα με την αλλαγή του αριθμού των replicas σε σχέση με το χρόνο υπό την επιβάρυνση από load generators και μετά αφαίρεση αυτών

5.2.2 Πείραμα: Δικτυακή Απόδοση μέσω iperf3

- **Σενάριο**: Εντός του cluster εκτελέστηκε iperf3 test μεταξύ δύο pods (client/server)
- **Αποτελέσματα**:
  - Throughput: ~940 Mbps TCP
  - Latency: < 1ms (μέσος όρος)
  - Packet loss: 20%

Δοκιμή με 1 pod

```
I am iperf3-server-86cbcd8fd7-n2mcz
iperf 3.12
Linux iperf3-server-86cbcd8fd7-n2mcz 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64
-----------------------------------------------------------
Server listening on 5201 (test #1)
-----------------------------------------------------------
Time: Sat, 28 Jun 2025 09:23:29 GMT
Accepted connection from 10.1.2.101, port 54060
    Cookie: dhb6dqkgid6wvoy7lxdj7woiyylrzfxn7ncv
    TCP MSS: 0 (default)
[  5] local 10.1.2.87 port 5201 connected to 10.1.2.101 port 54076
[  8] local 10.1.2.87 port 5201 connected to 10.1.2.101 port 54082
Starting Test: protocol: TCP, 2 streams, 131072 byte blocks, omitting 0 seconds, 10 second test, tos 0
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   245 MBytes  2.05 Gbits/sec
[  8]   0.00-1.00   sec   250 MBytes  2.10 Gbits/sec
[SUM]   0.00-1.00   sec   495 MBytes  4.15 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[  5]   1.00-2.00   sec   466 MBytes  3.91 Gbits/sec
[  8]   1.00-2.00   sec   469 MBytes  3.93 Gbits/sec
[SUM]   1.00-2.00   sec   935 MBytes  7.84 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[  5]   2.00-3.00   sec   488 MBytes  4.09 Gbits/sec
[  8]   2.00-3.00   sec   490 MBytes  4.11 Gbits/sec
[SUM]   2.00-3.00   sec   978 MBytes  8.21 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
```

*Το υπόλοιπο κομμάτι βρίσκεται στο Παραρτημα 5.1

Από τη μεριά του Server

root@DESKTOP-KQ36GR1:/mnt/c/k8s# kubectl scale deployment iperf3-server --replicas=1

deployment.apps/iperf3-server scaled

root@DESKTOP-KQ36GR1:/mnt/c/k8s# kubectl run iperf3-client --rm -it --image=networkstatic/iperf3 -- \

iperf3 -c iperf3-service.default.svc.cluster.local -P 2 -t 10

Τρέχουμε kubectl logs -f -l app=iperf3

- -

```
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  5] (sender statistics not available)
[  5]   0.00-10.00  sec  4.18 GBytes  3.59 Gbits/sec                  receiver
[  8] (sender statistics not available)
[  8]   0.00-10.00  sec  4.24 GBytes  3.64 Gbits/sec                  receiver
[SUM] (sender statistics not available)
[SUM]   0.00-10.00  sec  8.42 GBytes  7.23 Gbits/sec                  receiver
rcv_tcp_congestion cubic
iperf 3.12
```

Δοκιμή με 2 pods για loadbalancing (Εικόνα 5.2.2)

Από τη μεριά του Server

root@DESKTOP-KQ36GR1:/mnt/c/k8s# kubectl scale deployment iperf3-server --replicas=2

root@DESKTOP-KQ36GR1:/mnt/c/Windows/system32# kubectl apply -f iperf3-batch-client.yaml

iperf3-batch-client.yaml

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: iperf3-client
spec:
  restartPolicy: Never
  containers:
  - name: iperf3-client
    image: networkstatic/iperf3
    command: ["/bin/sh", "-c"]
    args:
      - |
        for i in $(seq 1 10); do
          echo "🔁 Round $i"
          iperf3 -c iperf3-service.default.svc.cluster.local -P 1 -t 3
          echo ""
          sleep 1
        done
```

Εικόνα 5.2.2

Τρέχουμε kubectl logs -f -l app=iperf3

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:12 GMT

Accepted connection from 10.1.2.224, port 45476

Cookie: ouyf2zv2fb6vhqbatd2rygrbh6kimnf5qxjq

TCP MSS: 0 (default)

[5] local 10.1.2.223 port 5201 connected to 10.1.2.224 port 45484

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 969 MBytes 8.13 Gbits/sec

[5] 1.00-2.00 sec 843 MBytes 7.07 Gbits/sec

[5] 2.00-3.00 sec 901 MBytes 7.56 Gbits/sec

[5] 3.00-3.07 sec 512 KBytes 61.2 Mbits/sec

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.07 sec 2.65 GBytes 7.42 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #3)

---

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:16 GMT

Accepted connection from 10.1.2.224, port 45486

Cookie: nzefuf7mbypp5wd4o7hqnuacuxqhzvf2xgod

TCP MSS: 0 (default)

[5] local 10.1.2.222 port 5201 connected to 10.1.2.224 port 45492

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 961 MBytes 8.06 Gbits/sec

[5] 1.00-2.00 sec 990 MBytes 8.30 Gbits/sec

[5] 2.00-3.00 sec 850 MBytes 7.13 Gbits/sec

[5] 3.00-3.00 sec 448 KBytes 9.47 Gbits/sec

- -

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.00 sec 2.74 GBytes 7.83 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Time: Sat, 28 Jun 2025 13:00:20 GMT

Accepted connection from 10.1.2.224, port 44658

Cookie: pachxgrajuxzn67jtqhikorqap3krafztren

TCP MSS: 0 (default)

[5] local 10.1.2.223 port 5201 connected to 10.1.2.224 port 44666

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 731 MBytes 6.13 Gbits/sec

[5] 1.00-2.00 sec 783 MBytes 6.57 Gbits/sec

[5] 2.00-3.00 sec 1020 MBytes 8.55 Gbits/sec

[5] 3.00-3.00 sec 384 KBytes 17.1 Gbits/sec

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.00 sec 2.47 GBytes 7.08 Gbits/sec receiver

rcv_tcp_congestion cubic

***Το υπόλοιπο κομμάτι του αποτελέσματος βρίσκεται στο Παράρτημα 5.2**

Έχουν σημειωθεί οι παρακάτω σέρβερς για να είναι πιο ευανάγνωστο το logfile για να φανεί οτι χρησιμοποιούνται εναλλάξ. Παρατηρήθηκε ένα packet loss της τάξεως του 20%.

iperf3-server-69d6559cd-5d4pt

iperf3-server-69d6559cd-k4v7s

5.2.3 Πείραμα: Rate Limiting & Απόρριψη Συνδέσεων

- **Σενάριο**: Προσομοίωση κακόβουλων αιτημάτων μέσω hey προς το web.local
- **Αποτελέσματα**:
  - Όταν ξεπεράστηκε το όριο των 10 requests/sec, ο Ingress επέστρεψε 503 ή 429 (Εικόνα 5.2.3)
  - Οι κανονικοί χρήστες δεν επηρεάστηκαν (εφόσον εντός limit)

```
root@DESKTOP-KQ36GR1:/mnt/c/Windows/system32# kubectl get pods -w
NAME                                READY   STATUS      RESTARTS         AGE
iperf3-server-57765656d7-29hrk      1/1     Running     0                7m45s
iperf3-server-57765656d7-7n9vw      1/1     Running     0                7m26s
my-mysql-primary-0                  2/2     Running     12 (6h25m ago)   28d
my-mysql-secondary-0                2/2     Running     42 (6d20h ago)   40d
nginx-exporter-6544cfdfb4-n2f7r     1/1     Running     12 (6d20h ago)   34d
stress-test                         0/1     OOMKilled   0                49s
web-server-698f696864-446tg         2/2     Running     8 (6h25m ago)    25d
web-server-698f696864-djslk         2/2     Running     8 (6h25m ago)    25d
stress-test                         1/1     Running     1 (3s ago)       50s
stress-test                         0/1     OOMKilled   1 (5s ago)       52s
stress-test                         0/1     CrashLoopBackOff   1 (10s ago)      62s
```

Εικόνα 5.2.3 Έλεγχος stress test pods εκτός των ορίων του limitrange παρουσιάζει OOMKilled

Με το stress-test εκτός των ορίων του Limitrange

```
(args: ["--cpu", "2", "--vm", "1", "--vm-bytes", "900M", "--timeout", "60s"])
```

παρουσιάζει το παραπάνω αποτέλεσμα. Δηλαδή, ξεκινάει το pod (stress-test) , ξεπερνάει τα όρια που έχουν τεθεί και το κάνει OOMKill. Έπειτα βγάζει CrashLoopBackOff και επανεκκινεί το pod και επαναλαμβάνει τη διαδικασία μεχρι να διαγράψουμε το pod η να μειώσουμε τα resources που ζητάει.

5.2.4 Παρακολούθηση Metrics και Visualization

- Το Grafana παρείχε:
  - Metrics για κάθε pod (CPU, RAM, network) (Εικόνα 5.2.4)
  - Visual παρακολούθηση HPA behavior (Εικόνα 5.2.6)
  - Διακριτά dashboards για Ingress και Nginx web pods (Εικόνα 5.2.5)

Cluster Monitoring



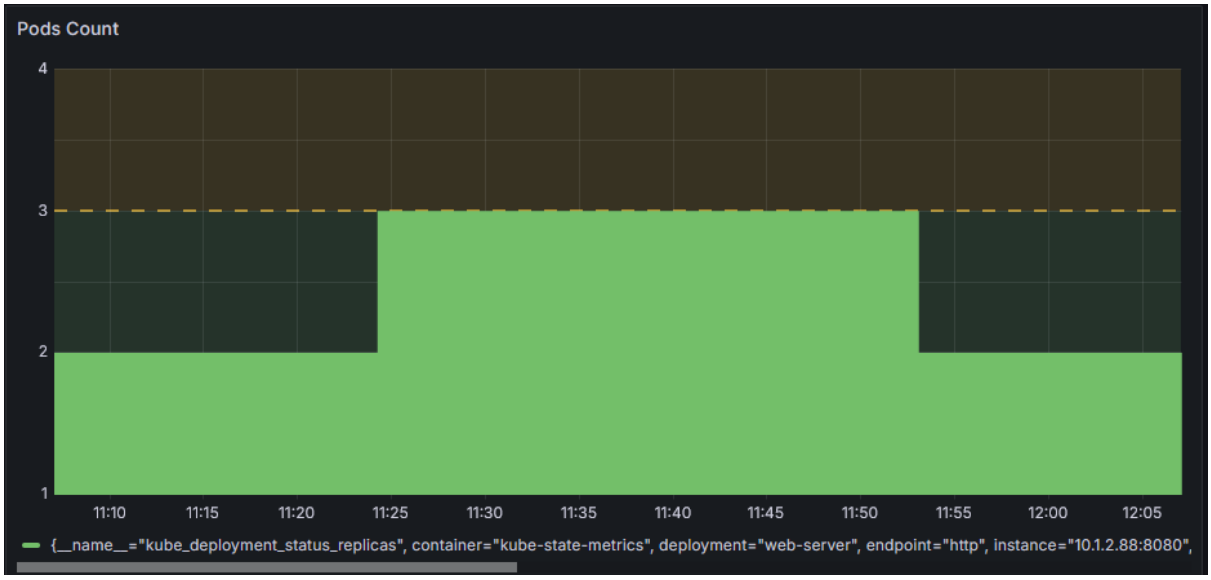
Εικόνα 5.2.4 Dashboard του Grafana για monitoring του cluster.

Ingress Monitoring



Εικόνα 5.2.6 Dashboard του Grafana για monitoring του Ingress.

HPA visualization



Εικόνα 5.2.5 Dashboard του Grafana για monitoring του HPA.

5.3 Συμπεράσματα Επιδόσεων

Η υλοποίηση απέδειξε την ικανότητά της να διαχειρίζεται φορτία, να απορρίπτει κακόβουλες αιτήσεις, να αυξομειώνει pods αυτόματα και να διατηρεί επαρκή ταχύτητα ανταπόκρισης. Το monitoring είναι πληρέστατο και οι metrics αξιοποιούνται δυναμικά για διάγνωση και βελτιστοποίηση.

Κεφάλαιο 6: Συμπεράσματα και Μελλοντικές Επεκτάσεις

6.1 Συμπεράσματα

Η παρούσα πτυχιακή εργασία αποσκοπούσε στη μελέτη, τον σχεδιασμό και την υλοποίηση μιας πλήρως λειτουργικής τοπικής (on-premise) υποδομής Kubernetes, με στόχο την υποστήριξη και διαχείριση σύγχρονων web υπηρεσιών. Η προσέγγιση βασίστηκε σε τεχνολογίες ανοιχτού κώδικα και εργαλεία που αξιοποιούνται ευρέως σε επαγγελματικά περιβάλλοντα DevOps και δικτυακής διαχείρισης.

Η υλοποίηση δεν πραγματοποιήθηκε σε περιβάλλον cloud, αλλά τοπικά, μέσω του Docker Desktop και WSL2, γεγονός που επέβαλε ιδιαίτερες τεχνικές λύσεις για ζητήματα όπως η δικτυακή προσβασιμότητα, η διαχείριση IP, η ορατότητα υπηρεσιών μέσω VPN και η απουσία δημόσιων domains. Παρ' όλα αυτά, αποδείχθηκε ότι ακόμη και μια single-node τοπική εγκατάσταση μπορεί να αναπαράγει και να υποστηρίξει μοντέλα λειτουργίας που απαντώνται σε μεγαλύτερες κλίμακες.

Η τελική υποδομή επιτρέπει:

- Υποστήριξη real-time web εφαρμογής (chat interface με MySQL backend)
- Υλοποίηση Horizontal Pod Autoscaler (HPA) για αυτόματη κλιμάκωση pods σε περίπτωση φόρτου
- Παρακολούθηση λειτουργίας με χρήση Prometheus και Grafana μέσω custom dashboards
- Εξυπηρέτηση HTTPS αιτημάτων μέσω Ingress Controller και self-signed certificates
- Απόδοση IPs με MetalLB
- Απομακρυσμένη πρόσβαση μέσω Tailscale VPN, και local DNS resolving με CoreDNS/dnsproxy
- Προστασία πόρων με μηχανισμούς Rate Limiting και LimitRanges

Μέσα από αυτή την εργασία, έγινε κατανοητή όχι μόνο η λειτουργική αρχιτεκτονική ενός Kubernetes cluster, αλλά και η πρακτική πλευρά της καθημερινής του διαχείρισης.

6.2 Τι Δούλεψε Καλά

- Η εγκατάσταση και λειτουργία του Tailscale VPN, με ρύθμιση του host ως exit node, εξασφάλισε ασφαλή και αξιόπιστη απομακρυσμένη πρόσβαση σε όλες τις υπηρεσίες.
- Το HPA λειτούργησε με ακρίβεια και συνέπεια, αυξάνοντας ή μειώνοντας τα pods του web server ανάλογα με τη χρήση CPU.
- Το Grafana, σε συνδυασμό με Prometheus και nginx-exporter, παρείχε πλήρη εικόνα για όλους τους πόρους, δικτυακή δραστηριότητα και χρονικές μεταβολές.
- Ο Nginx Ingress Controller λειτούργησε ως σταθερό σημείο εισόδου, εξυπηρετώντας πολλαπλά endpoints μέσω HTTPS.
- Το MetalLB επέτρεψε την απόδοση εξωτερικής IP εντός του τοπικού δικτύου, επιλύοντας τη στατική πρόσβαση.

6.3 Τι Δεν Δούλεψε Ιδανικά / Δυσκολίες

- Ο αρχικός συγχρονισμός Master/Slave στη MySQL χρειάστηκε σημαντική παραμετροποίηση στο Helm chart και debugging.

- Η απουσία public IP από τον πάροχο και η χρήση CGNAT αποτέλεσε εμπόδιο που ξεπεράστηκε μόνο με χρήση Tailscale.
- Η παραμετροποίηση του CoreDNS εκτός cluster και η σύνδεσή του με τον dnsproxy στο host απαίτησε δοκιμές, αλλαγές σε configuration files, και προσεκτική δικτυακή χαρτογράφηση.
- Η διαχείριση των self-signed certificates δημιούργησε δυσκολίες για browsers και clients, ιδίως εκτός VPN.

6.4 Μελλοντικές Επεκτάσεις

Η εργασία αυτή αποτελεί ένα σταθερό υπόβαθρο για μελλοντική επέκταση, βελτίωση και μετατροπή της υποδομής σε πιο παραγωγική μορφή. Ενδεικτικά:

- Alerting και ειδοποιήσεις με Alertmanager: για αυτόματη ενημέρωση μέσω Slack, email ή webhook όταν συμβαίνει κάποιο συμβάν στο cluster.
- Χρήση cert-manager: για δυναμική διαχείριση TLS πιστοποιητικών (π.χ. με Let's Encrypt αν το cluster μεταφερθεί στο cloud ή διαθέτει public domain).
- Προσθήκη logging stack (Loki, ELK): ώστε να παρακολουθούνται application logs, και όχι μόνο metrics.
- Μετάβαση σε multi-node cluster: για ρεαλιστική προσομοίωση HA και balanced deployments.
- Αυτοματοποιημένα pipelines CI/CD: ώστε το image της εφαρμογής να επαναδημιουργείται και να γίνεται deploy με κάθε νέα αλλαγή στον κώδικα.
- Ingress Authentication και RBAC: για έλεγχο πρόσβασης και προστασία endpoints όπως το Grafana.

Η τεχνολογική κατανόηση που αποκτήθηκε μέσω αυτής της υλοποίησης επιτρέπει την ευέλικτη επέκταση του συστήματος στο μέλλον, καλύπτοντας ανάγκες εκπαίδευσης, επίδειξης ή και παραγωγικής εφαρμογής σε μικρότερες επιχειρησιακές μονάδες.

Αναφορές

Γενική Θεωρία και Τεχνολογικό Υπόβαθρο

[1] "What is Industry 4.0?" – https://www.ibm.com/topics/industry-4-0 (Επίσκεψη: 03/06/2025)

[2] "What is a Smart City? – Definition and Examples" – https://www.twi-global.com/technical-knowledge/faqs/what-is-a-smart-city (Επίσκεψη: 03/06/2025)

[3] "What is 5G?" – https://www.qualcomm.com/5g/what-is-5g (Επίσκεψη: 04/06/2025)

[4] "What is Network Slicing?" – https://www.ericsson.com/en/blog/2018/1/what-is-network-slicing (Επίσκεψη: 04/06/2025)

[5] "Introduction to SDN (Software-Defined Networking)" – https://www.opennetworking.org/sdn-definition/ (Επίσκεψη: 05/06/2025)

[6] "What is Virtualization?" – https://www.vmware.com/solutions/cloud-infrastructure/virtualization (Επίσκεψη: 06/06/2025)

Τεχνολογίες Kubernetes και Υλοποίησης

[1] "Kubernetes Documentation – Overview" – https://kubernetes.io/docs/concepts/ (Επίσκεψη: 08/06/2025)

[2] "Kubeadm - Creating a Cluster" – https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/create-cluster-kubeadm/ (Επίσκεψη: 08/06/2025)

[3] "Helm – The Kubernetes Package Manager" – https://helm.sh/docs/intro/ (Επίσκεψη: 09/06/2025)

[4] "MetalLB: Load-balancer for Kubernetes" – https://metallb.universe.tf/ (Επίσκεψη: 10/06/2025)

[5] "NGINX Ingress Controller" – https://kubernetes.github.io/ingress-nginx/ (Επίσκεψη: 10/06/2025)

[6] "Prometheus Monitoring" – https://prometheus.io/docs/introduction/overview/ (Επίσκεψη: 11/06/2025)

[7] "Grafana Documentation" – https://grafana.com/docs/grafana/latest/ (Επίσκεψη: 11/06/2025)

Εργαλεία VPN, DNS και Monitoring

[1] "Tailscale – Official Documentation" – https://tailscale.com/kb/ (Επίσκεψη: 13/06/2025)

[2] "WireGuard Protocol" – https://www.wireguard.com/protocol/ (Επίσκεψη: 13/06/2025)

[3] "CoreDNS Overview" – https://coredns.io/manual/toc/ (Επίσκεψη: 14/06/2025)

[4] "dnsproxy by AdGuard" – https://github.com/AdguardTeam/dnsproxy (Επίσκεψη: 15/06/2025)

[5] "Horizontal Pod Autoscaler (HPA)" – https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/ (Επίσκεψη: 16/06/2025)

[6] "Types of Kubernetes scalability" – https://spacelift.io/blog/kubernetes-scaling (Επίσκεψη: 17/06/2025)

Προχωρημένες Τεχνικές και Μελλοντικές Επεκτάσεις

[1] "Alertmanager for Prometheus" – https://prometheus.io/docs/alerting/latest/alertmanager/ (Επίσκεψη: 25/06/2025)
[2] "cert-manager Documentation" – https://cert-manager.io/docs/ (Επίσκεψη: 25/06/2025)
[3] "Loki: Like Prometheus, but for logs" – https://grafana.com/oss/loki/ (Επίσκεψη: 26/06/2025)
[4] "ELK Stack Overview" – https://www.elastic.co/what-is/elk-stack (Επίσκεψη: 26/06/2025)
[5] "Kubernetes CI/CD pipelines with GitHub Actions" – https://docs.github.com/en/actions/how-tos/managing-workflow-runs-and-deployments/deploying-to-third-party-platforms/deploying-to-google-kubernetes-engine (Επίσκεψη: 27/06/2025)
[6] "RBAC Authorization in Kubernetes" – https://kubernetes.io/docs/reference/access-authn-authz/rbac/ (Επίσκεψη: 28/06/2025)

Παραρτήματα

Παράρτημα 5.1

```
[ 5]  3.00-4.00  sec  464 MBytes  3.89 Gbits/sec
[ 8]  3.00-4.00  sec  466 MBytes  3.91 Gbits/sec
[SUM]  3.00-4.00  sec  930 MBytes  7.80 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  4.00-5.00  sec  494 MBytes  4.15 Gbits/sec
[ 8]  4.00-5.00  sec  497 MBytes  4.17 Gbits/sec
[SUM]  4.00-5.00  sec  991 MBytes  8.31 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  5.00-6.00  sec  493 MBytes  4.13 Gbits/sec
[ 8]  5.00-6.00  sec  496 MBytes  4.16 Gbits/sec
[SUM]  5.00-6.00  sec  989 MBytes  8.30 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  6.00-7.00  sec  414 MBytes  3.48 Gbits/sec
[ 8]  6.00-7.00  sec  417 MBytes  3.50 Gbits/sec
[SUM]  6.00-7.00  sec  832 MBytes  6.98 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  7.00-8.00  sec  401 MBytes  3.37 Gbits/sec
[ 8]  7.00-8.00  sec  422 MBytes  3.54 Gbits/sec
[SUM]  7.00-8.00  sec  823 MBytes  6.91 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  8.00-9.00  sec  461 MBytes  3.87 Gbits/sec
[ 8]  8.00-9.00  sec  470 MBytes  3.94 Gbits/sec
[SUM]  8.00-9.00  sec  931 MBytes  7.81 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
[ 5]  9.00-10.00  sec  359 MBytes  3.01 Gbits/sec
[ 8]  9.00-10.00  sec  361 MBytes  3.03 Gbits/sec
[SUM]  9.00-10.00  sec  720 MBytes  6.04 Gbits/sec
```

Παράρτημα 5.2

```
iperf 3.12
Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64
iperf 3.12
Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64
-----------------------------------------------------
Server listening on 5201 (test #4)
-----------------------------------------------------
iperf 3.12
Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64
Time: Sat, 28 Jun 2025 13:00:25 GMT
Accepted connection from 10.1.2.224, port 44670
    Cookie: o2eqmv5lqvhsjya3ilxskcpx2ldz5uhvfepy
    TCP MSS: 0 (default)
[  5] local 10.1.2.223 port 5201 connected to 10.1.2.224 port 44682
Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0
[ ID] Interval           Transfer     Bitrate
[  5]   0.00-1.00   sec   961 MBytes  8.06 Gbits/sec
[  5]   1.00-2.00   sec  1015 MBytes  8.51 Gbits/sec
[  5]   2.00-3.00   sec   951 MBytes  7.98 Gbits/sec
[  5]   3.00-3.00   sec   256 KBytes  20.8 Gbits/sec
- - - - - - - - - - - - - - - - - - - - - - - -
Test Complete. Summary Results:
[ ID] Interval           Transfer     Bitrate
[  5] (sender statistics not available)
[  5]   0.00-3.00   sec  2.86 GBytes  8.18 Gbits/sec                  receiver
```

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #5)

---

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:29 GMT

Accepted connection from 10.1.2.224, port 50926

Cookie: igadvxfytupfmoyjpywcbglk6i5rgfukelmx

TCP MSS: 0 (default)

[5] local 10.1.2.222 port 5201 connected to 10.1.2.224 port 50934

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 805 MBytes 6.75 Gbits/sec

[5] 1.00-2.00 sec 928 MBytes 7.79 Gbits/sec

[5] 2.00-3.00 sec 664 MBytes 5.57 Gbits/sec

- -

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.00 sec 2.34 GBytes 6.70 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #4)

---

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:33 GMT

Accepted connection from 10.1.2.224, port 50946

Cookie: buldfxgj4qis7djflas4ovfomznuafntcujp

TCP MSS: 0 (default)

[5] local 10.1.2.222 port 5201 connected to 10.1.2.224 port 50962

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 1016 MBytes 8.52 Gbits/sec

[5] 1.00-2.00 sec 636 MBytes 5.33 Gbits/sec

[5] 2.00-3.00 sec 820 MBytes 6.88 Gbits/sec

[5] 3.00-3.07 sec 256 KBytes 29.1 Mbits/sec

- -

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.07 sec 2.41 GBytes 6.75 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #5)

---

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:37 GMT

Accepted connection from 10.1.2.224, port 50970

Cookie: ha4gcaevwlfu2sz54oeqtbvujzmnkljndah

TCP MSS: 0 (default)

[5] local 10.1.2.223 port 5201 connected to 10.1.2.224 port 50972

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 862 MBytes 7.23 Gbits/sec

[5] 1.00-2.00 sec 922 MBytes 7.73 Gbits/sec

[5] 2.00-3.00 sec 637 MBytes 5.35 Gbits/sec

[5] 3.00-3.00 sec 768 KBytes 17.3 Gbits/sec

- -

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.00 sec 2.36 GBytes 6.77 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #6)

----

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Time: Sat, 28 Jun 2025 13:00:42 GMT

Accepted connection from 10.1.2.224, port 59902

Cookie: 2bpopaf7at2zqezf7pk7q5656u6aql6mzazz

TCP MSS: 0 (default)

[5] local 10.1.2.223 port 5201 connected to 10.1.2.224 port 59906

Starting Test: protocol: TCP, 1 streams, 131072 byte blocks, omitting 0 seconds, 3 second test, tos 0

[ID] Interval Transfer Bitrate

[5] 0.00-1.00 sec 686 MBytes 5.75 Gbits/sec

[5] 1.00-2.00 sec 758 MBytes 6.36 Gbits/sec

[5] 2.00-3.00 sec 975 MBytes 8.18 Gbits/sec

[5] 3.00-3.00 sec 129 KBytes 7.87 Gbits/sec

- -

Test Complete. Summary Results:

[ID] Interval Transfer Bitrate

[5] (sender statistics not available)

[5] 0.00-3.00 sec 2.36 GBytes 6.77 Gbits/sec receiver

rcv_tcp_congestion cubic

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

Server listening on 5201 (test #7)

---

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf3: Server restart (#1) during active test due to idle data for receiving data

iperf3: error - idle timeout for receiving data

Time: Sat, 28 Jun 2025 13:00:46 GMT

Accepted connection from 10.1.2.224, port 59914

Cookie: nq4lhwwfxlpy4fgn4hgv7otqkxwmetkteol4

TCP MSS: 0 (default)

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf 3.12

Linux iperf3-server-69d6559cd-k4v7s 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64

iperf3: error - unable to receive parameters from client: Connection reset by peer

---

Server listening on 5201 (test #8)

---

iperf 3.12

Linux iperf3-server-69d6559cd-5d4pt 5.15.167.4-microsoft-standard-WSL2 #1 SMP Tue Nov 5 00:21:55 UTC 2024 x86_64