ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και Συστημάτων
Πληροφορικής

Σχεδιασμός και υλοποίηση μηχανισμών συνεργατικής διαχείρισης και ενορχήστρωσης εφαρμογών σε υποδομές edge-cloud εφαρμογών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Ταμπάκης

Επιβλέπων : Βαρβαρίγος Εμμανουήλ
Καθηγητής Ε.Μ.Π

Αθήνα, Σεπτέμβριος 2025

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Επικοινωνιών, Ηλεκτρονικής και Συστημάτων
Πληροφορικής

Σχεδιασμός και υλοποίηση μηχανισμών συνεργατικής διαχείρισης και ενορχήστρωσης εφαρμογών σε υποδομές edge-cloud εφαρμογών

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Παναγιώτης Ταμπάκης

Επιβλέπων : Βαρβαρίγος Εμμανουήλ
Καθηγητής Ε.Μ.Π

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22α Σεπτεμβρίου 2025

.....	.....	.....
Εμμανουήλ Βαρβαρίγος	Θεοδώρα Βαρβαρίγου	Ηρακλής Αβραμόπουλος
Καθηγητής, ΕΜΠ	Καθηγήτρια, ΕΜΠ	Καθηγητής, ΕΜΠ

Αθήνα, Σεπτέμβριος 2025

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

Περίληψη

Η ταχεία εξέλιξη των πληροφοριακών συστημάτων και η αυξανόμενη πολυπλοκότητα των σύγχρονων εφαρμογών έχουν δημιουργήσει νέες προκλήσεις στη διαχείριση και ενορχήστρωση κατανεμημένων υπολογιστικών περιβαλλόντων. Η παρούσα διπλωματική εργασία επικεντρώνεται στη μελέτη, σχεδίαση και υλοποίηση προηγμένων μηχανισμών συνεργατικής διαχείρισης και ενορχήστρωσης εφαρμογών σε κατανεμημένες υποδομές edge-cloud.

Η κύρια καινοτομία της εργασίας έγκειται στην ανάπτυξη μιας κατανεμημένης αρχιτεκτονικής ενορχήστρωσης που υποστηρίζει τη δημιουργία και διαχείριση δυναμικών συνεργατικών σχημάτων, γνωστών ως Associations, τα οποία μπορούν να εκτείνονται σε πολλαπλούς παρόχους υπηρεσιών και γεωγραφικές περιοχές.

Η εργασία παρουσιάζει την ανάπτυξη ενός ολοκληρωμένου πλαισίου διαχείρισης και ενορχήστρωσης containers, δημιουργώντας μια λύση που αντιμετωπίζει τις προκλήσεις των σύγχρονων κατανεμημένων εφαρμογών. Η αξιολόγηση του συστήματος πραγματοποιήθηκε μέσω πειραματικών σεναρίων που αποδεικνύουν την αποτελεσματικότητα της προσέγγισης.

Λέξεις Κλειδιά: Ενορχήστρωση εφαρμογών, Cloud-native, Edge computing, Κατανεμημένα συστήματα, Kubernetes, Microservices

Abstract

The rapid evolution of information systems and the increasing complexity of modern applications have created new challenges in the management and orchestration of distributed computing environments. This thesis focuses on the study, design, and implementation of advanced mechanisms for collaborative management and orchestration of applications in distributed edge-cloud infrastructures.

The main innovation of this work lies in developing a distributed orchestration architecture that supports the creation and management of dynamic collaborative schemes, known as Associations, which can span multiple service providers and geographical regions.

The work presents the development of a comprehensive container management and orchestration framework, creating a solution that addresses the challenges of modern distributed applications. The system evaluation was conducted through experimental scenarios that demonstrate the effectiveness of the approach.

Key Words: Application orchestration, Cloud-native, Edge computing, Distributed systems, Kubernetes, Microservices

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά τον καθηγητή του ΕΜΠ κ. Εμμανουήλ Βαρβαρίγο για την εμπιστοσύνη και την δυνατότητα που μου έδωσε να μελετήσω ένα θέμα με αυξανόμενο ερευνητικό ενδιαφέρον.

Ένα μεγάλο ευχαριστώ οφείλω στον μεταδιδακτορικό ερευνητή του εργαστηρίου κ. Αριστοτέλη Κρέτση για την κομβική βοήθεια και ουσιαστική καθοδήγηση του κατά την εκπόνηση της παρούσης διπλωματικής εργασίας

Τέλος, θα ήθελα να ευχαριστήσω ιδιαίτερα τους γονείς μου και την αδερφή μου για την στήριξη τους όλα αυτά τα χρόνια καθώς και τους φίλους και συμφοιτητές μου για τις στιγμές που περάσαμε και έχουνε χαραχθεί ανεξίτηλα στην μνήμη και την καρδιά μου.

Περιεχόμενα

Κατάλογος εικόνων

Κεφάλαιο 1: Εισαγωγή

Η ταχεία εξέλιξη των πληροφοριακών συστημάτων και η αυξανόμενη πολυπλοκότητα των σύγχρονων εφαρμογών έχουν δημιουργήσει νέες προκλήσεις στη διαχείριση και ενορχήστρωση κατανεμημένων υπολογιστικών περιβαλλόντων. Παραδοσιακά, οι εφαρμογές σχεδιάζονταν ως μονολιθικές δομές, όπου όλα τα συστατικά στοιχεία ήταν στενά συνδεδεμένα και λειτουργούσαν ως μια ενιαία μονάδα. Ωστόσο, αυτή η προσέγγιση παρουσιάζει σημαντικούς περιορισμούς όσον αφορά την επεκτασιμότητα, την ευελιξία και τη δυνατότητα συντήρησης, ιδιαίτερα σε περιβάλλοντα με υψηλές απαιτήσεις διαθεσιμότητας και απόδοσης.

Η μετάβαση προς την αρχιτεκτονική των μικροϋπηρεσιών (microservices) αντιπροσωπεύει μια θεμελιώδη αλλαγή στον τρόπο σχεδίασης και υλοποίησης των σύγχρονων εφαρμογών. Σε αυτό το μοντέλο, οι εφαρμογές διασπώνται σε αυτόνομες, χαλαρά συζευγμένες μονάδες, καθεμία εκ των οποίων εκτελεί μια συγκεκριμένη επιχειρησιακή λειτουργία και επικοινωνεί με τις υπόλοιπες μέσω καλά ορισμένων διεπαφών. Αυτή η προσέγγιση προσφέρει σημαντικά πλεονεκτήματα, συμπεριλαμβανομένης της δυνατότητας ανεξάρτητης ανάπτυξης, κλιμάκωσης και συντήρησης κάθε microservice, καθώς και της ευελιξίας στην επιλογή τεχνολογιών και πλατφορμών υλοποίησης.

Παράλληλα, η εμφάνιση και ευρεία υιοθέτηση των τεχνολογιών containerization έχει επιταχύνει την μετάβαση προς τις cloud-native εφαρμογές. Τα containers παρέχουν ένα ελαφρύ, φορητό και συνεπές περιβάλλον εκτέλεσης που ενσωματώνει την εφαρμογή και όλες τις εξαρτήσεις της, εξασφαλίζοντας προβλέψιμη συμπεριφορά ανεξαρτήτως του υποκειμένου περιβάλλοντος. Η συνδυαστική χρήση microservices και containers έχει καταστήσει δυνατή την ανάπτυξη εφαρμογών που μπορούν να εκμεταλλεύονται πλήρως τις δυνατότητες των σύγχρονων κατανεμημένων υποδομών.

1.1 Προκλήσεις στη Διαχείριση Κατανεμημένων Συστημάτων

Η μετάβαση προς τις αρχιτεκτονικές microservices, παρά τα αναμφισβήτητα πλεονεκτήματά της, εισάγει νέες και σύνθετες προκλήσεις στη διαχείριση και ενορχήστρωση των εφαρμογών. Η διαχείριση εκατοντάδων ή χιλιάδων containers που εκτελούν διαφορετικά microservices σε ένα κατανεμημένο περιβάλλον απαιτεί εξελιγμένους μηχανισμούς αυτοματοποίησης και συντονισμού. Οι παραδοσιακές μέθοδοι χειροκίνητης διαχείρισης καθίστανται ανεπαρκείς και επιρρεπείς σε σφάλματα όταν κλιμακώνονται σε τέτοια επίπεδα πολυπλοκότητας.

Η ανάγκη για δυναμική κατανομή των microservices βάσει κριτηρίων όπως η απόδοση, η διαθεσιμότητα των πόρων, οι πολιτικές ασφάλειας και οι γεωγραφικοί περιορισμοί, προσθέτει επιπλέον στρώματα πολυπλοκότητας. Επιπρόσθετα, η αυτόματη αντίδραση σε καταστάσεις αστοχίας, η κλιμάκωση κατ' απαίτηση και η εξασφάλιση της συνέπειας και ακεραιότητας των δεδομένων σε ένα κατανεμημένο περιβάλλον αποτελούν κρίσιμες απαιτήσεις που πρέπει να ικανοποιηθούν για την επίτευξη αξιόπιστων και επεκτάσιμων λύσεων.

Ιδιαίτερα σύνθετο καθίσταται το πρόβλημα όταν οι διαθέσιμοι υπολογιστικοί πόροι εκτείνονται σε πολλαπλές ετερογενείς υποδομές, που μπορεί να περιλαμβάνουν τοπικά κέντρα δεδομένων, περιφερειακούς κόμβους (edge nodes), δημόσια και ιδιωτικά clouds, καθώς και υβριδικές λύσεις. Η αποτελεσματική αξιοποίηση αυτού του κατανεμημένου φάσματος υπολογιστικών πόρων απαιτεί ευφυείς μηχανισμούς που μπορούν να λαμβάνουν αποφάσεις σε πραγματικό χρόνο, συνυπολογίζοντας παράγοντες όπως το κόστος λειτουργίας, τους χρόνους απόκρισης, τους κανονιστικούς περιορισμούς και τις απαιτήσεις συμμόρφωσης.

1.2 Η Ανάγκη για Ευφυή Ενορχήστρωση

Στο σύγχρονο τεχνολογικό τοπίο, η ενορχήστρωση εφαρμογών έχει εξελιχθεί από μια απλή διαδικασία ανάπτυξης και διαχείρισης σε μια πολύπλοκη, γνωσιακά καθοδηγούμενη διαδικασία που απαιτεί την ενσωμάτωση τεχνικών τεχνητής νοημοσύνης και μηχανικής μάθησης. Η ανάγκη για ευφυείς μηχανισμούς ενορχήστρωσης προκύπτει από την αδυναμία των παραδοσιακών, στατικών προσεγγίσεων να ανταποκριθούν στη δυναμικότητα και πολυπλοκότητα των σύγχρονων κατανεμημένων περιβαλλόντων.

Οι σύγχρονες εφαρμογές χαρακτηρίζονται από μεταβλητότητα στις απαιτήσεις φορτίου, ποικιλομορφία στις ανάγκες πόρων και αυστηρούς περιορισμούς στους χρόνους απόκρισης. Για την αντιμετώπιση αυτών των προκλήσεων, απαιτούνται συστήματα ενορχήστρωσης που μπορούν να προβλέπουν τις μελλοντικές ανάγκες, να προσαρμόζονται δυναμικά στις μεταβαλλόμενες συνθήκες και να βελτιστοποιούν συνεχώς τη διάθεση των πόρων. Αυτό περιλαμβάνει την ικανότητα αυτόματης κλιμάκωσης των υπηρεσιών βάσει προβλεπτικών μοντέλων, την έξυπνη τοποθέτηση εφαρμογών για μείωση της καθυστέρησης δικτύου και την προληπτική μετακίνηση φορτίων εργασίας για αποφυγή υπερφόρτωσης.

Επιπρόσθετα, η εμφάνιση νέων παραδειγμάτων υπολογισμού, όπως το edge computing και το fog computing, έχει δημιουργήσει ένα συνεχές φάσμα από τα IoT devices στο cloud, που απαιτεί ολιστική προσέγγιση στην ενορχήστρωση. Οι μηχανισμοί ενορχήστρωσης πρέπει να είναι ικανοί να διαχειρίζονται αυτό το ετερογενές περιβάλλον, λαμβάνοντας υπόψη τους περιορισμούς κάθε επιπέδου και εκμεταλλευόμενοι τα συγκριτικά πλεονεκτήματά τους.

1.3 Σκοπός και Συνεισφορά της Εργασίας

Η παρούσα διπλωματική εργασία επικεντρώνεται στη μελέτη, σχεδίαση και υλοποίηση προηγμένων μηχανισμών συνεργατικής διαχείρισης και ενορχήστρωσης εφαρμογών σε κατανεμημένες υποδομές edge-cloud. Ο κύριος στόχος είναι η ανάπτυξη ενός ευέλικτου και επεκτάσιμου συστήματος ενορχήστρωσης που μπορεί να λειτουργεί αποτελεσματικά σε ετερογενή περιβάλλοντα.

Το προτεινόμενο σύστημα βασίζεται στην επέκταση και εξέλιξη του πλαισίου ενορχήστρωσης που αναπτύχθηκε στα πλαίσια του ευρωπαϊκού έργου SERRANO [13]. Η κύρια καινοτομία της εργασίας έγκειται στην ανάπτυξη μιας κατανεμημένης αρχιτεκτονικής ενορχήστρωσης που υποστηρίζει τη δημιουργία και διαχείριση δυναμικών συνεργατικών σχημάτων, γνωστών ως

Associations (Ενώσεις), τα οποία μπορούν να εκτείνονται σε πολλαπλούς παρόχους υπηρεσιών και γεωγραφικές περιοχές.

Η προσέγγιση που υιοθετείται διαφοροποιείται από τις υπάρχουσες λύσεις μέσω της εισαγωγής μηχανισμών γνωσιακής ενορχήστρωσης που επιτρέπουν στους ενορχηστρωτές υπηρεσιών να λειτουργούν ως αυτόνομες οντότητες, ανταγωνιζόμενες για την αποδοτική και ταχεία αντιστοίχιση των εφαρμογών με τους διαθέσιμους πόρους. Αυτό το μοντέλο όχι μόνο βελτιώνει την συνολική απόδοση του συστήματος, αλλά και εισάγει στοιχεία ανθεκτικότητας και προσαρμοστικότητας που είναι κρίσιμα για τη λειτουργία σε δυναμικά περιβάλλοντα.

Η κύρια συνεισφορά της εργασίας περιλαμβάνει την ανάπτυξη ενός ολοκληρωμένου πλαισίου διαχείρισης και ενορχήστρωσης containers, δημιουργώντας μια λύση που μπορεί να αντιμετωπίσει τις προκλήσεις των σύγχρονων κατανεμημένων εφαρμογών. Το σύστημα υποστηρίζει τη διαφανή ενορχήστρωση εφαρμογών σε ετερογενείς υποδομές. Επιπλέον, η εργασία παρέχει μια πρακτική επίδειξη της αποτελεσματικότητας της προσέγγισης μέσω της υλοποίησης και αξιολόγησης σε πραγματικά σενάρια χρήσης.

1.4 Διάρθρωση της Εργασίας

Η παρούσα εργασία οργανώνεται σε επτά κεφάλαια που καλύπτουν διαδοχικά τόσο το θεωρητικό υπόβαθρο όσο και την πρακτική υλοποίηση της προτεινόμενης λύσης. Κάθε κεφάλαιο έχει σχεδιαστεί να παρέχει μια λεπτομερή και συνεκτική παρουσίαση των αντίστοιχων θεμάτων, διευκολύνοντας την κατανόηση της συνολικής προσέγγισης και των τεχνικών επιλογών.

Το δεύτερο κεφάλαιο παρουσιάζει το θεωρητικό υπόβαθρο που υποστηρίζει την εργασία, εστιάζοντας στις βασικές αρχές των εγγενών εφαρμογών υπολογιστικού νέφους (cloud-native applications), την αρχιτεκτονική microservices, τις τεχνολογίες containerization και τα σύγχρονα εργαλεία ενορχήστρωσης. Ιδιαίτερη έμφαση δίνεται στο Kubernetes ως την κυρίαρχη πλατφόρμα ενορχήστρωσης, αναλύοντας την αρχιτεκτονική του, τις δυνατότητές του και τα πλεονεκτήματά του έναντι εναλλακτικών λύσεων.

Το τρίτο κεφάλαιο εμβαθύνει στην ανάλυση του συστήματος και την παρουσίαση της προτεινόμενης σχεδίασης. Αρχικά, παρουσιάζεται το έργο SERRANO και ο ρόλος του ως θεμελιώδους πλαισίου για την ανάπτυξη της εργασίας. Στη συνέχεια, αναλύονται οι στόχοι της παρέμβασης και παρουσιάζεται λεπτομερώς η αρχιτεκτονική του Service Orchestrator, συμπεριλαμβανομένων των επιμέρους συστατικών του και των αλληλοεπιδράσεων τους.

Το τέταρτο κεφάλαιο αφιερώνεται στην υλοποίηση του συστήματος, παρέχοντας μια εκτενή περιγραφή των τεχνικών επιλογών, των APIs που αναπτύχθηκαν και των μηχανισμών αποθήκευσης και διαχείρισης δεδομένων. Ιδιαίτερη προσοχή δίνεται στην παρουσίαση της ροής επεξεργασίας των αιτημάτων για ενορχήστρωση και εκτέλεση (application deployment) και στον τρόπο με τον οποίο τα διάφορα συστατικά του συστήματος συνεργάζονται για την επίτευξη των επιθυμητών αποτελεσμάτων.

Τα κεφάλαια πέντε και έξι επικεντρώνονται στην αξιολόγηση της λύσης και την παρουσίαση των συμπερασμάτων αντίστοιχα. Το πέμπτο κεφάλαιο παρουσιάζει τα αποτελέσματα της πειραματικής αξιολόγησης του συστήματος μέσω συγκεκριμένων σεναρίων εφαρμογής, ενώ το έκτο κεφάλαιο συνοψίζει τα κύρια ευρήματα, αναδεικνύει τα δυνατά και αδύνατα σημεία της υλοποίησης και διατυπώνει προτάσεις για μελλοντικές βελτιώσεις και επεκτάσεις. Τέλος, το έβδομο κεφάλαιο παραθέτει την βιβλιογραφία που υποστήριξε την εκπόνηση της εργασίας.

Κεφάλαιο 2: Θεωρητικό Υπόβαθρο

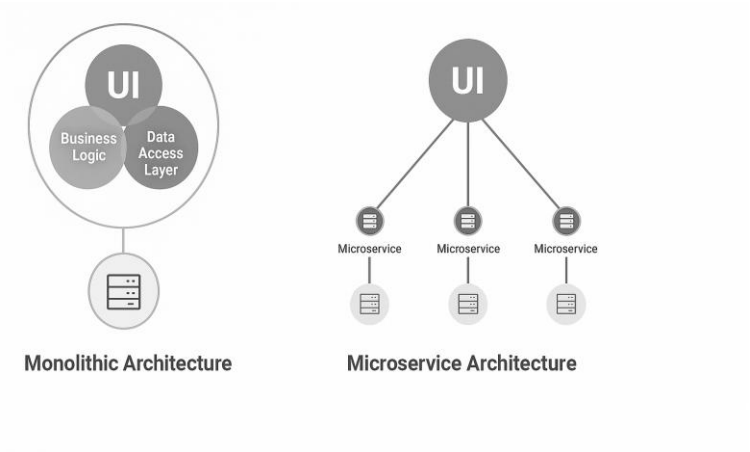
2.1 Cloud Native Εφαρμογές

2.1.1 Προσδιορισμός cloud-native εφαρμογών

Οι εγγενείς εφαρμογές υπολογιστικού νέφους (cloud-native applications) είναι προγράμματα λογισμικού που έχουν σχεδιαστεί ώστε να εκτελούνται σε δομές υπολογιστικού νέφους (cloud computing). Σκοπός αυτής της προσέγγισης είναι η αξιοποίηση των μοντέλων παροχής υπηρεσιών του cloud [1]. Η ελαστικότητα (elasticity) , η ανθεκτικότητα (resilience) και η ευελιξία (flexibility) που προσφέρονται επιτρέπουν στους οργανισμούς να δημιουργούν και να λειτουργούν επεκτάσιμες εφαρμογές σε σύγχρονα, δυναμικά περιβάλλοντα όπως δημόσια, ιδιωτικά και υβριδικά clouds [2].

2.1.2 Δομή cloud-native εφαρμογών

Οι cloud-native εφαρμογές διαφοροποιούνται από την παραδοσιακή μονολιθική αρχιτεκτονική σχεδίασης μιας εφαρμογής, όπου όλα τα στοιχεία (components) της εφαρμογής είναι στενά συνδεδεμένα μεταξύ τους και λειτουργούν ως μια υπηρεσία [3]. Αυτή η προσέγγιση έχει περιορισμούς όσον αφορά την επεκτασιμότητα, την ευελιξία και την δυνατότητα συντήρησης. Στην αρχιτεκτονική cloud-native οι εφαρμογές δημιουργούνται ως ένα σύνολο ανεξάρτητων, χαλαρά συζευγμένων microservices που μπορούν εύκολα να αναπτυχθούν και να κλιμακωθούν προς τα πάνω ή προς τα κάτω για να ανταποκριθούν στις μεταβαλλόμενες απαιτήσεις.



Εικόνα 1:Μονολιθική αρχιτεκτονική και αρχιτεκτονική βασισμένη σε μικροϋπηρεσίες.

Η αρχιτεκτονική των microservices περιλαμβάνει την κατανομή των εφαρμογών σε μικρά, ανεξάρτητα στοιχεία που να αναπτυχθούν και να διατηρηθούν ξεχωριστά. Κάθε microservice εστιάζει σε μια συγκεκριμένη λειτουργία και επικοινωνεί με τα άλλα microservices μέσω σαφώς καθορισμένων προγραμματιστικών διεπαφών (Application Programming Interface - API). Αυτή η προσέγγιση επιτρέπει στους προγραμματιστές να δημιουργούν εφαρμογές που

είναι πιο αρθρωτές, επεκτάσιμες και ανθεκτικές σε σφάλματα, καθώς οι αστοχίες σε μια μικροϋπηρεσία δεν επηρεάζουν ολόκληρη την εφαρμογή [4].

2.1.3 Τεχνολογίες υποδομής cloud-native εφαρμογών

Η ευρεία υιοθέτηση της αρχιτεκτονικής microservices έχει καταστήσει τα containers βασικό δομικό στοιχείο των σύγχρονων cloud-native εφαρμογών, καθώς παρέχουν έναν ευέλικτο και αποτελεσματικό τρόπο πακεταρίσματος, διανομής και εκτέλεσης μεμονωμένων υπηρεσιών. Τα containers αποτελούν ελαφριά, εκτελέσιμα πακέτα λογισμικού, τα οποία ενσωματώνουν όλα τα απαραίτητα συστατικά για την εκτέλεση μιας υπηρεσίας: τον πηγαίο κώδικα, το runtime, τις εξαρτώμενες βιβλιοθήκες, τις μεταβλητές περιβάλλοντος (environment variables), και τα εργαλεία του λειτουργικού συστήματος. Παρέχουν ένα απομονωμένο και συνεκτικό περιβάλλον εκτέλεσης, το οποίο διατηρείται αμετάβλητο καθ'όλη τη διάρκεια του κύκλου ζωής της εφαρμογής, εξασφαλίζοντας τη σταθερότητα και την προβλεψιμότητα της συμπεριφοράς της, ανεξαρτήτως διαφορών μεταξύ περιβαλλόντων (π.χ development ή staging). [5]

Ένα από τα βασικά πλεονεκτήματα των containers είναι η **φορητότητα (portability)** που προσφέρουν. Καθώς ενσωματώνουν ένα microservice και όλες τις εξαρτήσεις του σε μια αυτοτελή μονάδα, επιτρέπουν τη μεταφορά και την εκτέλεση της εφαρμογής με συνέπεια σε διαφορετικά περιβάλλοντα. Η δυνατότητα αυτή είναι κρίσιμη για microservice αρχιτεκτονικές, όπου κάθε υπηρεσία μπορεί να χρειαστεί να αναπτυχθεί σε διαφορετικές πλατφόρμες, από τοπικά συστήματα ανάπτυξης έως περιβάλλοντα παραγωγής στο cloud.

Παράλληλα, τα containers διευκολύνουν την **επεκτασιμότητα (scalability)** και την **ευελιξία (flexibility)** της εφαρμογής. Υποστηρίζουν την δυναμική κλιμάκωση, επιτρέποντας την εύκολη ανάπτυξη και διαχείριση των microservices βάσει των απαιτήσεων φορτίου. Αυτό δίνει την δυνατότητα στις εφαρμογές να προσαρμόζονται σε μεταβαλλόμενα επίπεδα ζήτησης με ελάχιστη προσπάθεια. Επιπλέον, η απομονωμένη φύση των containers επιτρέπει την αυτόνομη ενημέρωση ή επαναδιάθεση επιμέρους υπηρεσιών, χωρίς να επηρεάζεται το υπόλοιπο σύστημα. Αυτή η δυνατότητα είναι καθοριστική για την υποστήριξη σύγχρονων πρακτικών συνεχούς ενσωμάτωσης και συνεχούς παράδοσης (Continuous Integrations and Continuous Delivery (CI/CD)), καθώς επιτρέπει τη γρήγορη και ασφαλή διάθεση νέων χαρακτηριστικών, διορθώσεων και ενημερώσεων.

Τέλος, σε αντίθεση με τις παραδοσιακές εικονικές μηχανές (virtual machines), οι οποίες προσομοιώνουν ολόκληρη τη στοίβα υλικού, τα containers εικονοποιούν μόνο το επίπεδο του λειτουργικού συστήματος. Συγκεκριμένα, μοιράζονται τον ίδιο πυρήνα (kernel) του λειτουργικού, ενώ διατηρούν πλήρη απομόνωση της εφαρμογής και των εξαρτήσεων της. Το μοντέλο αυτό επιτρέπει την εκκίνηση containers σε ελάχιστο χρόνο με μειωμένες απαιτήσεις σε πόρους, προσφέροντας σημαντικά πλεονεκτήματα απόδοσης και αξιοποίησης της υποδομής σε σχέση με τις παραδοσιακές μεθόδους εικονοποίησης. Κατά συνέπεια, η συνδυαστική χρήση microservices και containers προσφέρει υψηλό βαθμό αποκέντρωσης, επαναληψιμότητας,

κλιμάκωσης και διαχείρισης, στοιχεία που αποτελούν τον ακρογωνιαίο λίθο του σύγχρονου cloud-native σχεδιασμού.

2.2 Ενορχήστρωση εφαρμογών

2.2.1 Δομή και λειτουργία

Η ραγδαία εξέλιξη του cloud computing και των microservices έχει καταστήσει τα containers βασικό συστατικό της σύγχρονης ανάπτυξης λογισμικού. Ωστόσο, καθώς τα πληροφοριακά συστήματα γίνονται ολοένα και πιο σύνθετα, η ανάγκη για αποτελεσματική διαχείριση μεγάλου αριθμού containers καθιστά την ενορχήστρωση απαραίτητη. Η ενορχήστρωση εφαρμογών με βάση τα containers αφορά τη διαδικασία αυτοματοποιημένης διαχείρισης του κύκλου ζωής τους σε κατανεμημένα περιβάλλοντα υποδομής, περιλαμβάνοντας την εγκατάσταση, παρακολούθηση, κλιμάκωση, συντονισμό και αποκατάσταση σφαλμάτων [6].

Η διαδικασία της ενορχήστρωσης περιλαμβάνει μια σειρά από διακριτά στάδια που συνεργάζονται ώστε να διασφαλίσουν τη σταθερή, αυτόματη και αποδοτική λειτουργία τέτοιου τύπου εφαρμογών σε υπολογιστικά περιβάλλοντα μεγάλης κλίμακας. Η πρώτη απαραίτητη προϋπόθεση είναι η **δημιουργία ενός υπολογιστικού συμπλέγματος (cluster)**, δηλαδή μιας δομής από συνδεδεμένα μηχανήματα ή κόμβους, οι οποίοι συνεργάζονται για να εκτελέσουν τις εφαρμογές. Το σύμπλεγμα αυτό περιλαμβάνει τόσο κόμβους ελέγχου (control plane), υπεύθυνους για τον συντονισμό και τη διαχείριση, όσο και κόμβους εκτέλεσης (worker nodes), οι οποίοι παρέχουν τους αναγκαίους υπολογιστικούς πόρους για την εκτέλεση των containers και των σχετικών υπηρεσιών. Η εγκατάσταση του cluster μπορεί να πραγματοποιηθεί είτε σε τοπική υποδομή είτε σε περιβάλλοντα cloud, με δυνατότητες υβριδικής διασύνδεσης.

Μετά τη συγκρότηση του cluster, ακολουθεί η παραμετροποίηση της επιθυμητής κατάστασης του συστήματος μέσω δηλωτικών αρχείων, τα οποία συντάσσονται ακολουθώντας πρότυπα περιγραφής και ανταλλαγής δεδομένων όπως YAML ή JSON. Σε αυτά τα αρχεία περιγράφεται η στοχευμένη αρχιτεκτονική των υπηρεσιών: ο αριθμός των containers που πρέπει να εκτελούνται ταυτόχρονα, οι σχέσεις εξάρτησης μεταξύ τους, οι απαιτήσεις σε υπολογιστικούς πόρους, καθώς και οι κανόνες διαθεσιμότητας και ασφάλειας. Η δηλωτική αυτή προσέγγιση ευθυγραμμίζεται με τις αρχές του "Infrastructure as Code", δίνοντας τη δυνατότητα αποθήκευσης και αναπαραγωγής της υποδομής μέσω εργαλείων ελέγχου έκδοσης.

Στη συνέχεια, ο ενορχηστρωτής καθορίζει την κατανομή των containers στους διαθέσιμους κόμβους. Η διαδικασία αυτή περιλαμβάνει την επιλογή των κατάλληλων κόμβων με βάση τις διαθέσιμες υπολογιστικές δυνατότητες, τις πολιτικές τοποθέτησης, αλλά και τους περιορισμούς που έχουν τεθεί στη δηλωτική παραμετροποίηση. Αμέσως μετά, τα containers ενεργοποιούνται στους αντίστοιχους κόμβους, πραγματοποιείται η λήψη των απαραίτητων εικόνων (images), η εκκίνηση των διεργασιών και η ρύθμιση των μεταβλητών περιβάλλοντος και του απαιτούμενου αποθηκευτικού χώρου (storage volumes).

Ένα κρίσιμο στάδιο στην συνολική διαδικασία ενορχήστρωσης και διαχείρισης σε τέτοια περιβάλλοντα υποδομών είναι αυτό της δικτύωσης και της εσωτερικής ανακάλυψης υπηρεσιών. Ο ενορχηστρωτής δημιουργεί ένα δυναμικό δικτυακό επίπεδο που επιτρέπει την απρόσκοπτη επικοινωνία μεταξύ των υπηρεσιών χωρίς τη χρήση στατικών διευθύνσεων. Οι υπηρεσίες αποκτούν ονόματα μέσω συστήματος DNS, ενώ διαχειρίζεται αυτόματα και τη κατανομή φορτίου (load balancing) μεταξύ τους.

Καθ' όλη τη διάρκεια λειτουργίας του συστήματος, ο ενορχηστρωτής εκτελεί συνεχώς έναν μηχανισμό παρακολούθησης και συμμόρφωσης, γνωστό και ως control loop. Μέσω αυτής της διαδικασίας, ελέγχεται διαρκώς αν η τρέχουσα κατάσταση του συστήματος συμφωνεί με την καθορισμένη επιθυμητή κατάσταση. Σε περιπτώσεις αστοχίας, όπως για παράδειγμα η αποτυχία ενός container ή η αποσύνδεση ενός κόμβου, ενεργοποιούνται αυτόματα ενέργειες αποκατάστασης, όπως η επανεκκίνηση ή η επανατοποθέτηση των υπηρεσιών σε άλλους κόμβους.

Η ενορχήστρωση περιλαμβάνει επίσης τη δυνατότητα οριζόντιας ή κάθετης κλιμάκωσης των υπηρεσιών, βάσει προγραμματισμένων κανόνων ή μετρικών σε πραγματικό χρόνο, όπως το ποσοστό χρήσης του επεξεργαστή ή της μνήμης. Ο ενορχηστρωτής μπορεί να προσθέσει ή να αφαιρέσει αντίγραφα υπηρεσιών ώστε να ανταποκριθεί στις εκάστοτε ανάγκες του φορτίου εργασίας. Τέλος, η διαδικασία ολοκληρώνεται με τη δυνατότητα αναβάθμισης των υπηρεσιών μέσω ελεγχόμενων μεθόδων όπως τα rolling updates ή τα blue/green deployments, καθώς και με την υποστήριξη rollback σε περίπτωση αποτυχίας, διασφαλίζοντας τη σταθερότητα και τη συνέχεια λειτουργίας του συστήματος.

2.2.2 Πλεονεκτήματα αυτοματοποιημένης ενορχήστρωσης σε edge-cloud υποδομές

Ένα από τα σημαντικότερα οφέλη της ενορχήστρωσης σε περιβάλλοντα edge-cloud υποδομών είναι η **αυξημένη διαθεσιμότητα** των εφαρμογών. Όταν προκύπτουν βλάβες σε κόμβους ή αποτυχίες εκτέλεσης containers, τα συγκεκριμένα συστήματα ενορχήστρωσης επεμβαίνουν αυτόματα μέσω μηχανισμών αυτοΐασης (self-healing). Επανατοποθετούν τα σχετικά containers σε λειτουργικά σημεία του συστήματος ή τα επανεκκινούν, εξασφαλίζοντας έτσι τη συνεχή διαθεσιμότητα των παρεχόμενων υπηρεσιών [7].

Ένα ακόμη βασικό πλεονέκτημα είναι η **δυναμική και αυτοματοποιημένη κλιμάκωση** των εφαρμογών. Τα εργαλεία ενορχήστρωσης μπορούν να δημιουργούν ή να τερματίζουν containers σε πραγματικό χρόνο, βάσει των συνθηκών φόρτου, της κατανάλωσης πόρων ή άλλων μετρικών απόδοσης. Αυτό επιτρέπει την ορθολογική χρήση των υποδομών, μειώνοντας τις καθυστερήσεις και διασφαλίζοντας βελτιωμένη εμπειρία χρήστη [7].

Η **αξιοποίηση των υπολογιστικών πόρων** γίνεται επίσης πιο αποτελεσματική μέσω της ενορχήστρωσης. Οι εφαρμογές κατανέμονται σε κοινόχρηστους κόμβους με βέλτιστο τρόπο, μειώνοντας την αδράνεια και αποφεύγοντας τη σπατάλη πόρων. Η δυνατότητα κατανομής φόρτου μεταξύ των pods ή containers σε επίπεδο συστήματος αποτελεί έναν κρίσιμο παράγοντα εξοικονόμησης κόστους [8].

Η **αυτοματοποιημένη διαχείριση** και παρακολούθηση είναι άλλο ένα σημαντικό όφελος. Μέσω δηλωτικής (declarative) παραμετροποίησης, οι διαχειριστές μπορούν να καθορίσουν την επιθυμητή κατάσταση του συστήματος, και τα εργαλεία ενορχήστρωσης αναλαμβάνουν να διατηρούν αυτόματα το περιβάλλον σε συμμόρφωση με αυτές τις παραμέτρους. Αυτό μειώνει τις ανάγκες για χειροκίνητες παρεμβάσεις και ελαχιστοποιεί τον κίνδυνο ανθρώπινου λάθους [6].

Τέλος, η **ευκολία ενσωμάτωσης εργαλείων ανάπτυξης (CI/CD)** και η υποστήριξη DevOps διαδικασιών καθιστούν την ενορχήστρωση απαραίτητο εργαλείο για σύγχρονα πληροφοριακά συστήματα. Η διαχείριση εκδόσεων, η σταδιακή αναβάθμιση εφαρμογών (rolling updates), και η δυνατότητα ταχείας ανάκτησης σε προηγούμενες εκδόσεις (rollbacks) υλοποιούνται με ασφάλεια και συνέπεια, ενισχύοντας την παραγωγικότητα και την ταχύτητα παράδοσης λογισμικού [8].

2.2.2 Σύγκριση εργαλείων ενορχήστρωσης

Η επιλογή του κατάλληλου εργαλείου ενορχήστρωσης container αποτελεί κρίσιμη απόφαση κατά τη σχεδίαση υποδομών cloud-native εφαρμογών. Η αξιολόγηση των κυριότερων εργαλείων, **Docker Swarm** [11], **Red Hat OpenShift** [10] και **Kubernetes** [12]**,** αναδεικνύει ουσιαστικές διαφορές ως προς την ευχρηστία, την επεκτασιμότητα, την ασφάλεια, την αυτοματοποίηση και την παραγωγική αξιοπιστία. [10], [11], [12]

Το Docker Swarm διακρίνεται για την απλότητά του, καθιστώντας το προσιτό σε όσους κάνουν τα πρώτα τους βήματα στον χώρο της ενορχήστρωσης containers. Η ενσωμάτωσή του στο περιβάλλον του Docker CLI επιτρέπει εύκολη εγκατάσταση και βασική λειτουργικότητα χωρίς περίπλοκες ρυθμίσεις. Αντίθετα, το Kubernetes διαθέτει πολυπλοκότερη αρχιτεκτονική και απαιτεί μεγαλύτερη καμπύλη εκμάθησης, όμως προσφέρει σημαντικά περισσότερες δυνατότητες παραμετροποίησης και ευελιξίας. Το OpenShift, ως εμπορική επέκταση του Kubernetes, προσπαθεί να ισορροπήσει μεταξύ ευχρηστίας και ισχύος, παρέχοντας φιλικό περιβάλλον χρήστη και αυτοματοποιημένες λειτουργίες, αλλά και αυξημένες εξαρτήσεις από το οικοσύστημα της Red Hat.

Ως προς την **κλιμάκωση** και την **επεκτασιμότητα**, το Kubernetes ξεχωρίζει λόγω της ικανότητάς του να διαχειρίζεται υποδομές μεγάλης κλίμακας με αποδοτική κατανομή φορτίου και έξυπνη αξιοποίηση πόρων. Το OpenShift ακολουθεί, επεκτείνοντας τις δυνατότητες του Kubernetes με έτοιμες ενσωματωμένες λειτουργίες. Το Docker Swarm υπολείπεται σε αυτόν τον τομέα, καθώς η κλιμάκωση που προσφέρει είναι περιορισμένη και λιγότερο ευέλικτη, καθιστώντας το καταλληλότερο για μικρές ή πειραματικές υλοποιήσεις.

Η **υποστήριξη κοινότητας** και το **ευρύτερο οικοσύστημα** είναι εμφανώς υπέρ του Kubernetes, το οποίο διαθέτει τη μεγαλύτερη και πιο ενεργή βάση χρηστών, προγραμματιστών και τεκμηρίωσης. Το OpenShift επωφελείται από αυτή την κοινότητα αλλά αναπτύσσεται με πιο ελεγχόμενο ρυθμό λόγω της εμπορικής του φύσης. Αντιθέτως, η υποστήριξη του Docker

Swarm έχει μειωθεί αισθητά τα τελευταία χρόνια, περιορίζοντας την ενεργή του συνεισφορά στον χώρο της ενορχήστρωσης.

Σε επίπεδο **ασφάλειας**, το OpenShift υπερέχει προσφέροντας ενσωματωμένους μηχανισμούς ελέγχου πρόσβασης, πολιτικές χρήστη, και προεγκατεστημένες πρακτικές ασφαλούς λειτουργίας. Το Kubernetes, αν και ιδιαίτερα ευέλικτο, απαιτεί πρόσθετη παραμετροποίηση ή ενσωμάτωση εργαλείων τρίτων για την επίτευξη αντίστοιχου επιπέδου ασφάλειας. Το Docker Swarm προσφέρει βασικές δυνατότητες ασφάλειας, οι οποίες όμως θεωρούνται ανεπαρκείς για σύνθετες ή παραγωγικές εγκαταστάσεις.

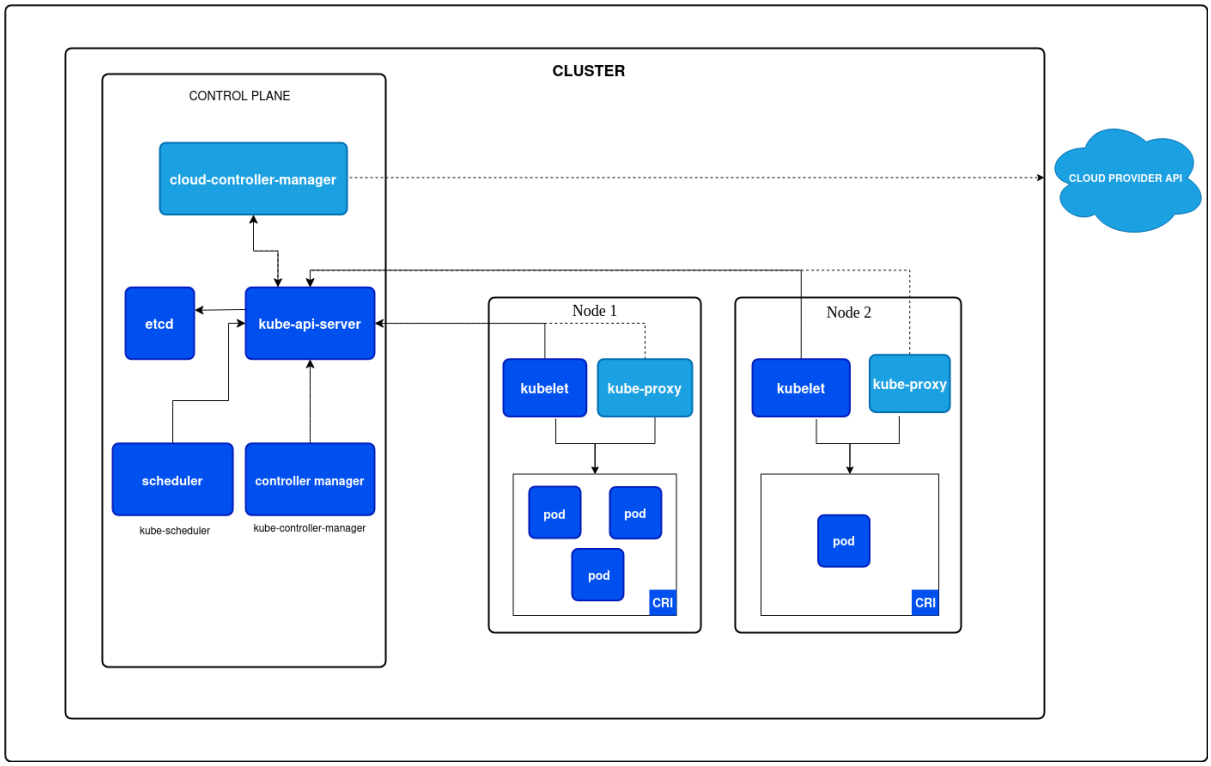
Σημαντικό πλεονέκτημα του OpenShift είναι η **εγγενής υποστήριξη για συστήματα CI/CD**, με ενσωματωμένα pipelines και έτοιμες ροές αυτοματοποίησης. Το Kubernetes προσφέρει εξαιρετικές δυνατότητες σύνδεσης με εξωτερικά εργαλεία όπως τα Jenkins [29], Tekton [30] και Argo CD [31], αλλά δεν περιλαμβάνει εργοστασιακά ενσωματωμένο μηχανισμό. Το Docker Swarm παρουσιάζει περιορισμένη ενσωμάτωση σε συστήματα αυτοματισμού, κάτι που αυξάνει το λειτουργικό βάρος στους διαχειριστές του.

Όσον αφορά την **καταλληλότητα για παραγωγικά περιβάλλοντα**, το Kubernetes έχει σχεδιαστεί εξαρχής για να υποστηρίζει κρίσιμης σημασίας υποδομές, με πλήρη υποστήριξη για υψηλή διαθεσιμότητα, αυτοΐαση και συνεχείς αναβαθμίσεις. Το OpenShift ενδείκνυται επίσης για περιβάλλοντα παραγωγής, προσφέροντας επιπλέον εργαλεία για διαχειριστικό έλεγχο. Το Docker Swarm, παρά τη λειτουργικότητά του, περιορίζεται κυρίως σε μικρής κλίμακας χρήσεις ή περιβάλλοντα ανάπτυξης. [9], [10], [11]

Η συγκριτική αυτή ανάλυση καταδεικνύει την υπεροχή του **Kubernetes ως το πιο πλήρες και ευέλικτο εργαλείο ενορχήστρωσης**. Παρά τις αρχικές του απαιτήσεις σε εκμάθηση και παραμετροποίηση, η επεκτασιμότητα, η ισχυρή κοινότητα υποστήριξης και η ευελιξία του το καθιστούν τη βέλτιστη επιλογή για σύγχρονες υποδομές cloud-native εφαρμογών.

2.2.3 Kubernetes

Το Kubernetes αποτελεί ένα εξελιγμένο σύστημα ανοικτού κώδικα για την ενορχήστρωση containers, το οποίο επιτρέπει την αυτόματη ανάπτυξη, κλιμάκωση και διαχείριση σύγχρονων, κατανεμημένων εφαρμογών. Σχεδιάστηκε με στόχο την υποστήριξη της αρχιτεκτονικής microservices και βασίζεται σε μια κατανεμημένη, αποκεντρωμένη αρχιτεκτονική που διαχωρίζεται λειτουργικά σε δύο βασικά επίπεδα: το **επίπεδο ελέγχου** (control plane), το οποίο συντονίζει τη συνολική λειτουργία του συστήματος, και τους **κόμβους εργασίας** (worker nodes), όπου λαμβάνει χώρα η πραγματική εκτέλεση των εφαρμογών.



Εικόνα 2: Διάγραμμα της αρχιτεκτονικής ενός kubernetes cluster

Θεμελιώδης μονάδα εκτέλεσης στο Kubernetes είναι το **Pod**, το οποίο αποτελεί την ελάχιστη λογική μονάδα ανάπτυξης και φιλοξενεί ένα ή περισσότερα containers που μοιράζονται το ίδιο δίκτυο και χώρο αποθήκευσης. Αντί να διαχειρίζεται μεμονωμένα containers, το Kubernetes προσεγγίζει την ενορχήστρωση σε επίπεδο Pod, επιτρέποντας έτσι συνεκτική διαχείριση containers που λειτουργούν από κοινού. Τα Pods είναι εφήμερα από τη φύση τους, δεν τροποποιούνται μετά τη δημιουργία τους και συνήθως αντικαθίστανται όταν απαιτείται ενημέρωση ή αποκατάσταση.

Το επίπεδο ελέγχου περιλαμβάνει τον **API Server**, ο οποίος λειτουργεί ως κύρια διεπαφή του συστήματος, προσφέροντας ένα RESTful API για την υποβολή εντολών και τη διαχείριση της κατάστασης. Ο **etcd** [15], ένα κατανεμημένο αποθετήριο δεδομένων τύπου key-value, χρησιμοποιείται για την αποθήκευση όλων των κρίσιμων δεδομένων κατάστασης και διαμόρφωσης του cluster. Ο **Scheduler** είναι υπεύθυνος για την τοποθέτηση των Pods σε κατάλληλους κόμβους, με βάση κριτήρια όπως η διαθεσιμότητα πόρων, οι περιορισμοί τοποθέτησης και οι απαιτήσεις ποιότητας υπηρεσίας. Ο **Controller Manager** συντονίζει την αυτόματη διαχείριση αντικειμένων του Kubernetes, παρακολουθώντας τη διαφορά μεταξύ της επιθυμητής και της πραγματικής κατάστασης, και ενεργοποιεί τις απαραίτητες ενέργειες για την επίτευξη συνέπειας. Τέλος, ο **Cloud Controller Manager** προσφέρει μηχανισμούς ενοποίησης με υποδομές υπολογιστικού νέφους, επιτρέποντας στο Kubernetes να αλληλεπιδρά με εξωτερικές υπηρεσίες όπως φόρτωση εξισορρόπησης, δυναμικό provisioning αποθηκευτικού χώρου κ.ά.0 [24]

Οι κόμβοι εργασίας είναι φυσικά ή εικονικά μηχανήματα που φιλοξενούν τα Pods και περιλαμβάνουν βασικά συστατικά όπως το **Kubelet** [22], έναν τοπικό πράκτορα που

επικοινωνεί με το επίπεδο ελέγχου και εξασφαλίζει ότι κάθε Pod εκτελείται όπως έχει καθοριστεί. Το **Kube-proxy** είναι υπεύθυνο για τη διαχείριση της κυκλοφορίας του δικτύου στον κόμβο, εφαρμόζοντας πολιτικές δρομολόγησης και εξισορρόπησης φορτίου [23]. Το **Container Runtime** αποτελεί το υπόστρωμα που αναλαμβάνει την εκτέλεση των containers μέσα σε κάθε Pod, με υποστήριξη για διάφορους μηχανισμούς όπως το Docker, το containerd ή το CRI-O. [19], [20], [21]

Η αρχιτεκτονική του Kubernetes έχει σχεδιαστεί με έμφαση στην αυτοθεραπεία, την κλιμάκωση κατ' απαίτηση, και την συνεχή ανάπτυξη χωρίς διακοπή υπηρεσίας (rolling updates και rollbacks). Ο μηχανισμός παρακολούθησης και επανεκκίνησης Pods που παρουσιάζουν αστοχία, σε συνδυασμό με τα ReplicaSets και Deployments, επιτρέπουν την κατασκευή συστημάτων υψηλής διαθεσιμότητας και ανθεκτικότητας. Παράλληλα, μέσω του αφαίρεσης πολύπλοκων υπηρεσιών και της ενοποίησης με μηχανισμούς παρακολούθησης (monitoring), καταγραφής (logging) και παρατήρησης (observability), το Kubernetes προσφέρει ένα πλήρες και επεκτάσιμο περιβάλλον για τη λειτουργία μοντέρνων, μικρο-εξυπηρετούμενων υποδομών.

Συμπερασματικά, το Kubernetes συνιστά μια αρχιτεκτονικά αρθρωτή και τεχνολογικά ώριμη πλατφόρμα, ικανή να καλύψει τις ανάγκες σύγχρονων υπολογιστικών περιβαλλόντων σε επίπεδο απόδοσης, ανθεκτικότητας και διαχειρισιμότητας, καθιστώντας το ουσιώδες εργαλείο για την υλοποίηση DevOps πρακτικών και cloud-native υποδομών μεγάλης κλίμακας.

Κεφάλαιο 3: Ανάλυση Συστήματος & Σχεδίαση

3.1 Ενοποιημένη ενορχήστρωση πολλαπλών υποδομών

Για την μελέτη, τον σχεδιασμό, και την υλοποίηση των κατάλληλων μηχανισμών για την συνεργατική διαχείριση πολλαπλών επιμέρους edge-cloud υποδομών, βασιστήκαμε σε προηγούμενες εργασίες που σχετίζονταν με την ενοποιημένη διαχείριση τέτοιων συστημάτων. Ένα τέτοιο σύστημα αποτελεί το αντίστοιχο λογισμικό ανοιχτού κώδικα που αναπτύχθηκε από το ερευνητικό έργο **SERRANO** [13]. Πρόκειται για μια ερευνητική πρωτοβουλία που αποσκοπεί στην θεμελίωση ενός ολοκληρωμένου υπολογιστικού οικοσυστήματος που γεφυρώνει τον υπολογιστικό χώρο μεταξύ edge, cloud και υψηλών επιδόσεων υπολογιστικών συστημάτων (High Performance Computing). Στον πυρήνα του έργου βρίσκεται η ανάγκη για ένα ευέλικτο, ασφαλές και γνωσιακά ενισχυμένο περιβάλλον για την ανάπτυξη, ενορχήστρωση και εκτέλεση εφαρμογών σε ετερογενείς υποδομές, με απώτερο σκοπό τη μείωση της πολυπλοκότητας στην ανάπτυξη και την διαχείριση κατανεμημένων υπηρεσιών.

Το SERRANO υιοθέτησε καινοτόμες τεχνολογικές προσεγγίσεις, όπως η διαφανής και βελτιστοποιημένη διάθεση εφαρμογών με βάση τις απαιτήσεις τους (application-aware deployment), η ενσωμάτωση επιταχυντών υλικού (όπως GPU και FPGA) για την αύξηση των επιδόσεων, καθώς και η χρήση τεχνητής νοημοσύνης για την αυτοματοποιημένη και γνωσιακά καθοδηγούμενη διαχείριση των υπολογιστικών πόρων. Το SERRANO εισήγαγε επίσης σημαντικές καινοτομίες στην διαχείριση δεδομένων, υποστηρίζοντας τεχνικές ασφαλούς αποθήκευσης, επεξεργασίας και μετάδοσης, με έμφαση σε περιπτώσεις χρήσης που αφορούν την βιομηχανία, τις χρηματοοικονομικές υπηρεσίες και την ανάλυση αισθητήρων σε πραγματικό χρόνο. Συνολικά το έργο συνέβαλε ουσιαστικά στη διαμόρφωση του μοντέλου του cloud continuum, προσφέροντας μία ολιστική και προσαρμοσμένη προσέγγιση στην ανάπτυξη σύγχρονων, απαιτητικών εφαρμογών σε πολυδιάστατα υπολογιστικά περιβάλλοντα.

3.1.1 Σύστημα ενορχήστρωσης

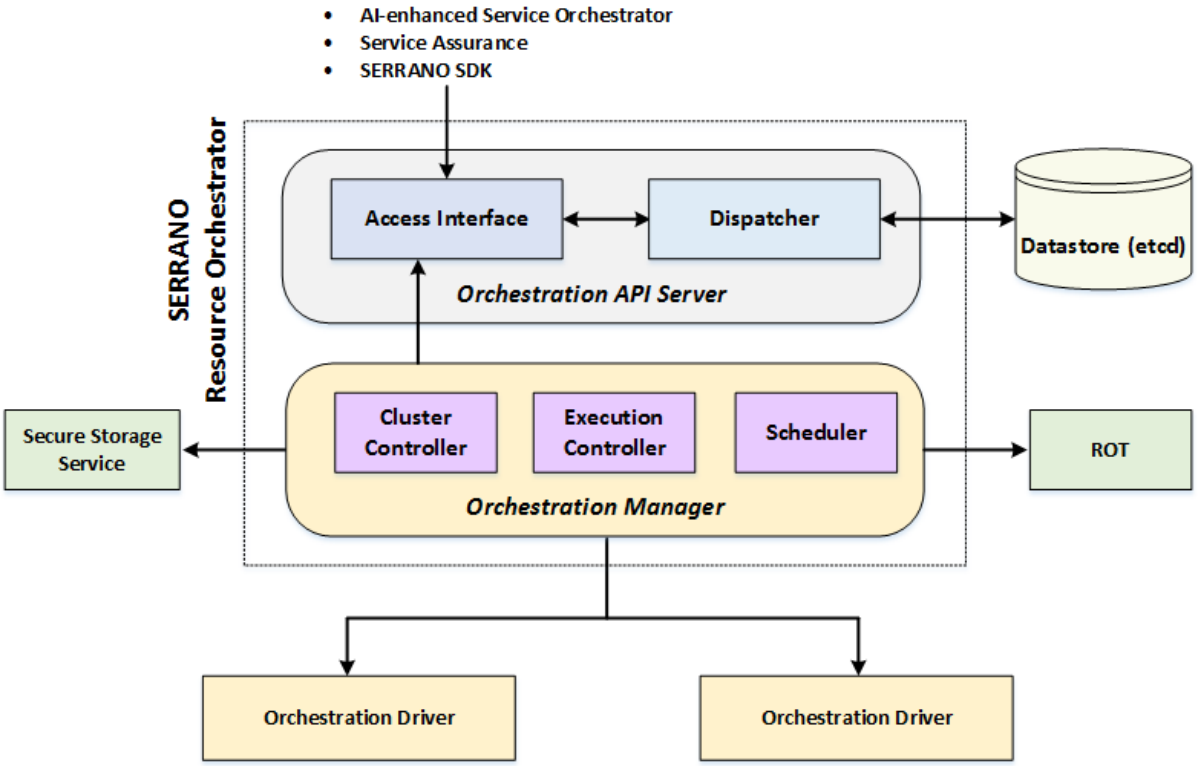
Ένα από τα πλέον κρίσιμα και καινοτόμα υποσυστήματα του έργου SERRANO είναι το σύστημα ενορχήστρωσης, το οποίο σχεδιάστηκε με σκοπό να επιτυγχάνει την ευφυή, δυναμική και αποδοτική ενορχήστρωση υπηρεσιών και πόρων στο πλαίσιο μιας ετερογενούς υπολογιστικής υποδομής που εκτείνεται από το cloud έως το edge.

Η αρχιτεκτονική του συγκεκριμένου συστήματος ενορχήστρωσης βασίζεται σε ένα **ιεραρχικό, μοντέλο ελέγχου δύο επιπέδων**, το οποίο αποσκοπεί στην επίτευξη ταυτόχρονης αποδοτικότητας, ευελιξίας, και επεκτασιμότητας κατά τη διαχείριση κατανεμημένων υπολογιστικών πόρων. Η στρατηγική αυτή ενσωματώνει δύο διακριτά αλλά αλληλοσυμπληρούμενα επίπεδα ενορχήστρωσης: (α) το **τοπικό επίπεδο** διαχείρισης πόρων και (β) το **κεντρικό επίπεδο** γνωσιακής ενορχήστρωσης.

Στο χαμηλότερο επίπεδο, κάθε επί μέρους υποδομή (όπως ένας edge κόμβος, ένα cloud cluster η ένας HPC πόρος) διαθέτει τον δικό της **τοπικό ενορχηστρωτή (Local Orchestrator),** ο οποίος είναι υπεύθυνος για την άμεση και λεπτομερή διαχείριση των διαθέσιμων φυσικών ή εικονικών πόρων. Οι τοπικοί ενορχηστρωτές αναλαμβάνουν την υλοποίηση χαμηλού επιπέδου πολιτικών, όπως η εκκίνηση και η διαχείριση container, η παρακολούθηση τοπικών μετρικών (π.χ χρήση CPU, μνήμης, επιτάχυνσης), και η ανταπόκριση σε βραχυπρόθεσμες μεταβολές του φορτίου. Η λειτουργία τους βασίζεται σε πλαίσια εξειδικευμένου λογισμικού όπως είναι το **Kubernetes** (προτιμητέα επιλογή για edge/cloud περιβάλλοντα) ή το **SLURM** [32] (σε HPC σενάρια), και χαρακτηρίζεται από αυτονομία και χαμηλό χρόνο απόκρισης.

Στο ανώτερο επίπεδο βρίσκεται ο **Resource Orchestrator**, ο οποίος επιτελεί τον ρόλο του γνωσιακά ενισχυμένου κεντρικού μηχανισμού ενορχήστρωσης. Ο Resource Orchestrator διαθέτει συνολική εποπτεία του υπολογιστικού συνεχούς και αξιοποιεί τεχνικές μηχανικής μάθησης, προβλεπτικά μοντέλα και πολιτικές πολλαπλών στόχων (multi-objective policies) για την εκτέλεση βελτιστοποιημένων αποφάσεων τοποθέτησης (placement), προγραμματισμού (scheduling) και μετακίνησης υπηρεσιών (load migration). Οι αποφάσεις βασίζονται σε μεταδεδομένα εφαρμογών, προτιμήσεις χρηστών και τη **τρέχουσα κατάσταση του συστήματος**, όπως αυτή καταγράφεται μέσω του *etcd* και των μηχανισμών παρακολούθησης.

Η παραπάνω **διεπίπεδη προσέγγιση** καθιστά τον μηχανισμό ενορχήστρωσης του SERRANO ικανό να λειτουργεί **με κατανεμημένο αλλά συγχρονισμένο τρόπο**, συνδυάζοντας την **τοπική ευελιξία** και **προσαρμοστικότητα** με την **ολιστική βελτιστοποίηση** σε επίπεδο συστήματος



Εικόνα 3: SERRANO Resource Orchestrator και υπηρεσίες (Πηγή: SERRANO Deliverable 5.4, Figure 76)

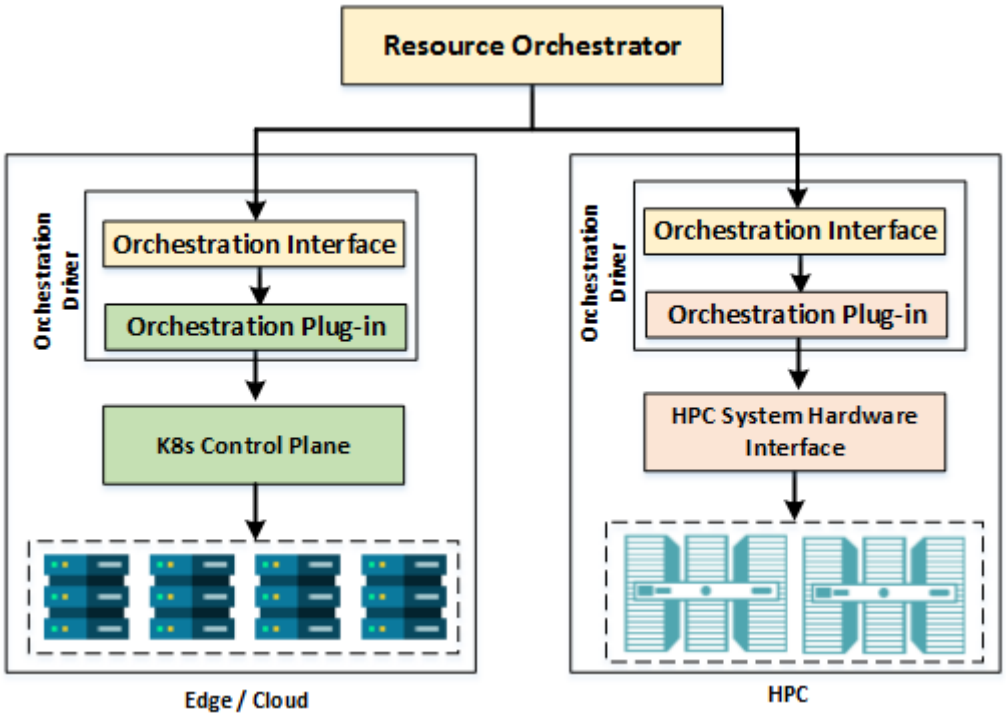
3.1.2 Διαδικασία ενορχήστρωσης

Η διαδικασία ενορχήστρωσης ενεργοποιείται όταν ο χρήστης υποβάλει, μέσω του RESTful API του SERRANO, ένα αίτημα για την εκτέλεση μιας εφαρμογής (deployment) στην ενοποιημένη υποδομή. Το αίτημα αυτό συνοδεύεται από μια περιγραφή της εφαρμογής (application descriptor), που αποτελεί έναν τυποποιημένο τρόπο περιγραφής των επιμέρους στοιχείων (microservices) της εφαρμογής. Η περιγραφή περιλαμβάνει αναλυτικά τις υπηρεσίες που απαρτίζουν την εφαρμογή (π.χ. μικροϋπηρεσίες ή υπολογιστικά workloads), τις μεταξύ τους εξαρτήσεις, τους περιορισμούς εκτέλεσης (όπως απαίτηση για GPU, εξάρτηση από άλλες υπηρεσίες ή δεδομένα), τις προτιμήσεις τοποθέτησης (placement affinities), καθώς και όρια χρόνου ή κόστους λειτουργίας.

Ο **Orchestration Manager**, μόλις εντοπίσει ένα νέο αντικείμενο τύπου Application στην Datastore, ενεργοποιεί μια εσωτερική ροή εργασιών που περιλαμβάνει διάφορους υπο-μηχανισμούς. Ο **Scheduler Controller** καλεί το **Resource Optimization Toolkit (ROT)** προκειμένου να επιλύσει το πρόβλημα της τοποθέτησης (placement) των υπηρεσιών στις διαθέσιμες υπολογιστικές υποδομές. Το ROT αποτελεί ένα **εξειδικευμένο υποσύστημα του SERRANO** υπεύθυνο για τη λήψη αποφάσεων βελτιστοποίησης, λαμβάνοντας υπόψη περιορισμούς εφαρμογών, προτιμήσεις χρηστών και δυναμικά χαρακτηριστικά του συστήματος. Ενσωματώνει ευρετικούς αλγόριθμους, τεχνικές μηχανικής μάθησης καθώς και μοντέλα πολλαπλών αντικρουόμενων στόχων (multi-objective optimization), ώστε να προσδιορίσει την καταλληλότερη κατανομή υπηρεσιών και πόρων με βάση κριτήρια όπως η

απόδοση, η ενεργειακή αποδοτικότητα ή η γεωγραφική εγγύτητα με τα δεδομένα. Ο σχεδιασμός του ROT επιτρέπει την επέκταση με νέους αλγορίθμους ανάθεσης, ανάλογα με τις ανάγκες των εφαρμογών ή τις πολιτικές της υποδομής.

Αφού ληφθούν οι αποφάσεις τοποθέτησης, ο Orchestrator δημιουργεί αντικείμενα τύπου Assignment, τα οποία καθορίζουν σε ποιον Local Orchestrator θα εκτελεστεί κάθε μέρος της εφαρμογής, και στη συνέχεια συντάσσει Bundle αντικείμενα που περιέχουν όλα τα απαιτούμενα **εκτελέσιμα μεταδεδομένα**, όπως αρχεία περιγραφής Kubernetes (YAML), τιμές παραμέτρων εκτέλεσης, πολιτικές κλιμάκωσης και στοιχεία ελέγχου πρόσβασης. Όλα τα παραπάνω αποθηκεύονται στο Datastore, το οποίο βασίζεται στην τεχνολογία etcd και επιτρέπει την **αυτοματοποιημένη παρακολούθηση** της κατάστασης μέσω του μηχανισμού *watch* , παρέχοντας ταυτόχρονα συνέπεια και διαθεσιμότητα.



Εικόνα 4: Orchestration Drivers(Πηγή: SERRANO Deliverable 5.4, Figure 82)

Η φάση υλοποίησης των αποφάσεων του Orchestration Manager πραγματοποιείται μέσω των **Orchestration Drivers**, οι οποίοι ακολουθούν ένα αρθρωτό σχεδιασμό και είναι υλοποιημένοι σε Python. Κάθε driver είναι υπεύθυνος για έναν συγκεκριμένο τύπο υποδομής και ενσωματώνει τον κατάλληλο μηχανισμό αλληλεπίδρασης με τον αντίστοιχο Local Orchestrator. Για παράδειγμα, ένας Kubernetes driver μετατρέπει το Bundle σε έγκυρα Kubernetes manifests και τα υποβάλλει στον API Server, ενώ ένας HPC driver χρησιμοποιεί τις κατάλληλες διεπαφές και scripts για να υποβάλει τις σχετικές εργασίες στο σύστημα SLURM. Οι drivers είναι επίσης υπεύθυνοι για τη συνεχή παρακολούθηση της προόδου εκτέλεσης, την αναφορά σφαλμάτων, την επικαιροποίηση του Datastore και την επιστροφή μετρικών εκτέλεσης για χρήση από τα ανώτερα επίπεδα.

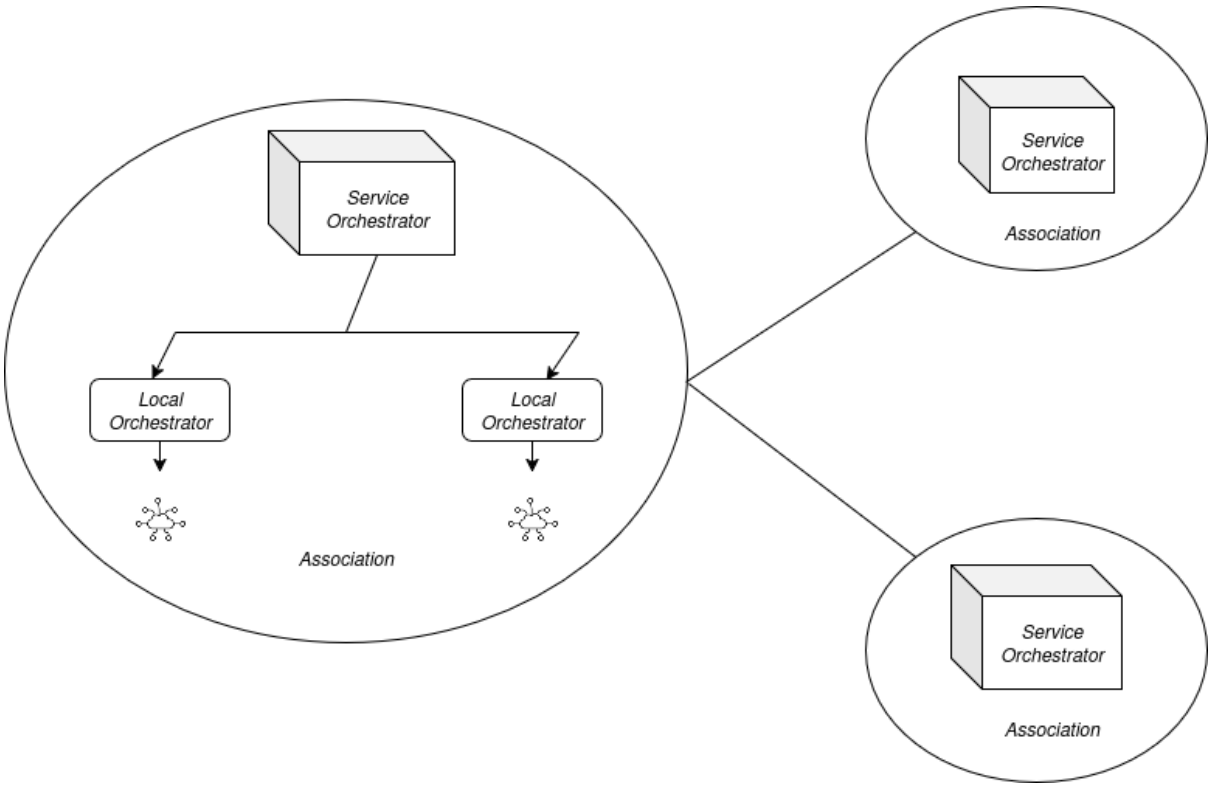
Σημαντικό είναι ότι η προσέγγιση του SERRANO ακολουθεί το δηλωτικό (declarative) πρότυπο ελέγχου, το οποίο σημαίνει ότι ο Resource Orchestrator δεν επιβάλλει ακριβή ακολουθία εντολών εκτέλεσης, αλλά ορίζει την επιθυμητή τελική κατάσταση του συστήματος (desired state). Οι Local Orchestrators, χρησιμοποιώντας τους εγγενείς τους μηχανισμούς, είναι υπεύθυνοι να επιτύχουν αυτή την κατάσταση, γεγονός που επιτρέπει μεγαλύτερη ευελιξία, προσαρμοστικότητα στις υποδομές και αποσύνδεση του ελέγχου από την εκτέλεση.

3.2 Στόχος παρέμβασης

Παραπάνω διαπιστώθηκε ότι μέσω του έργου SERRANO οι αυξημένες ανάγκες για αυτόματη και έξυπνη διαχείριση των διαθέσιμων υπολογιστικών πόρων μπορούν να καλυφθούν. Αυτό επιτυγχάνεται μέσω ενός προσαρμόσιμου ψηφιακού περιβάλλοντος, εμπλουτισμένου με μηχανισμούς μηχανικής γνώσης, διευκολύνοντας την σχεδίαση, τον συντονισμό και την λειτουργία λογισμικών σε ετερογενείς τεχνολογικές πλατφόρμες. Η ελαχιστοποίηση της τεχνικής πολυπλοκότητας που συνεπάγεται η υλοποίηση καθώς και η υποστήριξη κατανεμημένων συστημάτων αποτελούν τις βασικές αιτίες που το προαναφερθέν έργο αποτελεί θεμελιώδες κομμάτι της παρούσας διπλωματικής εργασίας. Σκοπός της οποίας είναι η σχεδίαση ενός ευέλικτου και πλήρους κατανεμημένου μηχανισμού για την ενορχήστρωση cloud native εφαρμογών στο πλαίσιο μιας κατανεμημένης ετερογενούς υποδομής.

Πιο συγκεκριμένα, θεωρούμε ένα κατανεμημένο περιβάλλον αποτελούμενο από πληθώρα ετερογενών κόμβων σε διαφορετικές γεωγραφικές τοποθεσίες, στο οποίο για την δημιουργία ενός ενιαίου συνεχούς υπολογιστικού συνεχούς (IoT-edge-cloud continuum) ακολουθείται ένα μοντέλο οργάνωσης που βασίζεται σε ομάδες συσχετιζόμενων πόρων (Associations). Τα Associations μπορούν να σχηματίζονται δυναμικά με βάση μια σειρά από κριτήρια (π.χ, γεωγραφική περιοχή, ειδών πόρων) και λειτουργούν αυτόνομα πάνω από ετερογενείς υποδομές καλύπτοντας πολλαπλούς παρόχους, διαμορφώνοντας ένα ενιαίο και αλληλοσυνδεμένο οικοσύστημα. Το υπολογιστικό συνεχές αυτό, βασισμένο στα Associations, επιτρέπει την αρμονική συνεργασία μεταξύ υποδομών edge και υπολογιστικού νέφους, επιτρέποντας την ανάπτυξη νέων υπερκατανεμημένων (hyper-distributed) και δυναμικών υπολογιστικά απαιτητικών εφαρμογών [16].

Στο πλαίσιο της παρούσας διπλωματικής εργασίας, θα επικεντρωθούμε στο κομμάτι της ενορχήστρωσης ενός κατανεμημένου συστήματος αποτελούμενο από πολλαπλά Associations. Σε επίπεδο Association, λειτουργεί ένας Ενορχηστρωτής Υπηρεσιών (Service Orchestrator) ως ενορχηστρωτής ανώτερου επιπέδου, ενώ διάφοροι Τοπικοί Ενορχηστρωτές (Local Orchestrators) διαχειρίζονται επιμέρους edge-cloud υποδομές που έχουν ενσωματωθεί στο οικοσύστημα. Επιπλέον, οι Service Orchestrators που είναι διαθέσιμοι στα επιμέρους Associations έχουν την δυνατότητα να συνεργάζονται μεταξύ τους, ώστε να μπορούν να εξυπηρετηθούν οι εφαρμογές κάνοντας ταυτόχρονα την καλύτερη δυνατή χρήση των διαθέσιμων υπολογιστικών πόρων. Το συγκεκριμένο κατανεμημένο μοντέλο ενορχήστρωσης επιτρέπει την αποδοτική και ευφυή διαχείριση πόρων και εργασιών σε ολόκληρο το συνεχές IoT–edge–cloud, εξασφαλίζοντας απρόσκοπτο συντονισμό και βελτιστοποίηση των φορτίων εργασίας εντός της πλατφόρμας.



Εικόνα 5: Σχηματική αναπαράσταση των Associations

Κατά το πρώτο στάδιο, οι Ενορχηστρωτές Υπηρεσιών λειτουργούν ως γνωσιακοί πράκτορες (cognitive agents), αξιοποιώντας την τοπική γνώση τους ώστε να είτε να συνεργαστούν είτε να ανταγωνιστούν μεταξύ τους για να επιτύχουν την αποδοτική και ταχεία αντιστοίχιση των εφαρμογών στα διαθέσιμα Associations. Στο δεύτερο στάδιο, κάθε Ενορχηστρωτής Υπηρεσιών εκχωρεί ευφυώς το αντίστοιχο τμήμα της συνολικής ροής εργασιών στις υποδομές που διαχειρίζεται. Για την υποστήριξη αυτής της διαδικασίας υιοθετείται μια ιεραρχική προσέγγιση ενορχήστρωσης. Οι αποφάσεις υψηλού επιπέδου λαμβάνονται από τον Ενορχηστρωτή Υπηρεσιών σε επίπεδο Associations και επιμέρους υποδομών, ενώ η λεπτομερής ανάθεση των εργασιών σε συγκεκριμένους πόρους υποδομής, πραγματοποιείται μέσω των εγγενών μηχανισμών ενορχήστρωσης (όπως K8s, K3s) κάθε πλατφόρμας.

3.3 Σχεδίαση

Το περιβάλλον που αποτελεί το αντικείμενο μελέτης της παρούσας διπλωματικής εργασίας συνιστά ένα εξελιγμένο κατανεμημένο σύστημα πολλαπλών παρόχων (multi-provider), το οποίο έχει σχεδιαστεί με στόχο την απρόσκοπτη και πλήρως αυτοματοποιημένη εκτέλεση cloud-native εφαρμογών. Η αρχιτεκτονική αυτή αντιμετωπίζει τις σύνθετες προκλήσεις που προκύπτουν από την ανάγκη συντονισμού και ενορχήστρωσης πόρων σε ετερογενή περιβάλλοντα, όπου πολλαπλοί οργανισμοί συνεργάζονται για την υλοποίηση κοινών στόχων.

Για την επίτευξη αυτού του φιλόδοξου στόχου, υιοθετήθηκε μια στρατηγική προσέγγιση που βασίζεται στην επέκταση και ενίσχυση του υπάρχοντος Resource Orchestrator απο το έργο SERRANO, ώστε να δημιουργηθεί ένα αρχικό πρωτότυπο για τον Service Orchestrator. Αυτή η απόφαση λήφθηκε μετά από εκτενή ανάλυση των υφιστάμενων λύσεων και την αναγνώριση των περιορισμών τους στο πλαίσιο των σύγχρονων απαιτήσεων για διαλειτουργικότητα και κλιμάκωση. Επιπλέον, η συγκεκριμένη προσέγγιση μας επιτρέπει να αξιοποιήσουμε την ήδη υπάρχουσα λειτουργία από την πλευρά του Resource Orchestrator και την ενορχήστρωση cloud-native υποδομών σε πολλαπλές υποδομές edge-cloud.

3.3.1 Επέκταση Service Orchestrator με objects

Η επέκταση στην οποία κατέληξε η παρούσα μελέτη χαρακτηρίζεται από την προσθήκη κρίσιμων αντικειμένων (Orchestration API Objects) που ενισχύουν σημαντικά τις δυνατότητες του αρχικού συστήματος. Αυτά τα νέα στοιχεία έχουν σχεδιαστεί με γνώμονα την παροχή υψηλού επιπέδου αφαίρεσης (high-level abstraction) και την απλοποίηση των διαδικασιών ανάπτυξης, ενώ ταυτόχρονα διατηρούν την ευελιξία και την προσαρμοστικότητα που απαιτούνται σε σύνθετα κατανεμημένα περιβάλλοντα.

Το **Associations Deployment** αποτελεί το βασικότερο και πιο σημαντικό στοιχείο της προτεινόμενης λύσης, συνιστώντας μια περιγραφή υψηλού επιπέδου που ενσωματώνει όλες τις απαραίτητες πληροφορίες για την επιτυχή εκτέλεση μιας cloud-native εφαρμογής σε πολλαπλά Associations. Αυτό το αντικείμενο λειτουργεί ως ένα κεντρικό μοντέλο δεδομένων που επιτρέπει τον συντονισμό και την ενορχήστρωση όλων των επιμέρους διαδικασιών ανάθεσης και εκτέλεσης, παρέχοντας μια ενιαία προσέγγιση στη διαχείριση σύνθετων cloud-native εφαρμογών σε ένα περιβάλλον συνεργατικών υποδομών.

Η σημασία του Associations Deployment εκδηλώνεται μέσα από τη δυνατότητά του να αφαιρεί την πολυπλοκότητα που σχετίζεται με την κατανεμημένη φύση του συστήματος, παρέχοντας στους χρήστες μια απλοποιημένη διεπαφή για την περιγραφή των απαιτήσεών τους. Ταυτόχρονα, το αντικείμενο αυτό διατηρεί την απαραίτητη λεπτομέρεια και ευελιξία που επιτρέπει την προσαρμογή σε διαφορετικά σενάρια χρήσης και περιβάλλοντα ανάπτυξης.

Το Associations Deployment ενσωματώνει δύο άξονες πληροφοριών: αφενός την περιγραφή της εφαρμογής (application specification) και αφετέρου τους στόχους ανάπτυξης που ορίζονται από τον χρήστη (user-defined deployment objectives). Αυτή η διττή φύση του αντικειμένου επιτρέπει την ολιστική προσέγγιση στη διαχείριση των cloud-native εφαρμογών, συνδυάζοντας τεχνικές προδιαγραφές με επιχειρησιακές απαιτήσεις και στρατηγικούς στόχους.

Η περιγραφή της εφαρμογής περιλαμβάνει λεπτομερείς πληροφορίες σχετικά με την αρχιτεκτονική, τις υπηρεσίες, τις διεπαφές και τις τεχνικές προδιαγραφές που απαιτούνται για την επιτυχή λειτουργία της εφαρμογής στο cloud-native περιβάλλον. Από την άλλη πλευρά, οι

στόχοι ανάπτυξης αντικατοπτρίζουν τις προτιμήσεις και τους περιορισμούς που θέτει ο χρήστης, όπως απαιτήσεις απόδοσης, ασφάλειας, κόστους και διαθεσιμότητας.

Ο κύκλος ζωής του Associations Deployment χαρακτηρίζεται από σαφώς καθορισμένες φάσεις και υπευθυνότητες. Η δημιουργία και διαγραφή του αντικειμένου τελεί υπό την αποκλειστική ευθύνη του εξυπηρετητή του Orchestration API, ο οποίος λειτουργεί ως κεντρικό σημείο ελέγχου για όλες τις λειτουργίες που σχετίζονται με τη διαχείριση των Associations Deployments. Αυτή η συγκεντρωτική προσέγγιση εξασφαλίζει τη συνοχή και την ακεραιότητα των δεδομένων, ενώ ταυτόχρονα παρέχει έναν ενιαίο σημείο ελέγχου για την εποπτεία των λειτουργιών.

Παράλληλα, η παρακολούθηση και αξιοποίηση του Associations Deployment ανατίθεται στον Orchestration Manager, ο οποίος αναλαμβάνει την ερμηνεία των προδιαγραφών και την υλοποίηση των απαραίτητων ενεργειών για την επίτευξη των επιθυμητών αποτελεσμάτων. Ο διαχωρισμός αυτός των ευθυνών συμβάλλει στη δημιουργία μιας ευέλικτης και κλιμακούμενης αρχιτεκτονικής που μπορεί να προσαρμοστεί σε διαφορετικές απαιτήσεις και σενάρια χρήσης.

Παρακάτω παρουσιάζεται η δομή του αντικειμένου Associations Deployment

Πεδίο	**Τύπος**
uuid	string
name	string
kind	string
user_uuid	string
application_uuid	string
associations	list
deployments	list
deployments_status	list
logs	list
status	integer
updated_by	string
created_at	integer
updated_at	integer

Βάσει της παρεχόμενης δομής δεδομένων, το Associations Deployment χαρακτηρίζεται από ένα εκτενές σύνολο ιδιοτήτων για την αποτελεσματική διαχείριση cloud-native εφαρμογών:

uuid (String): Αποτελεί το μοναδικό αναγνωριστικό του Associations Deployment, εξασφαλίζοντας την απόλυτη μοναδικότητα του αντικειμένου εντός του κατανεμημένου συστήματος. Αυτό το πεδίο είναι ιδιαίτερα κρίσιμο σε περιβάλλοντα όπου πολλαπλά στιγμιότυπα (instances) μπορεί να συνυπάρχουν και να αλληλεπιδρούν.

name (String): Παρέχει μια φιλική προς τον χρήστη ονομασία για το Associations Deployment, διευκολύνοντας την αναγνώριση και διαχείριση από τους διαχειριστές και προγραμματιστές. Αυτό το πεδίο συχνά αντικατοπτρίζει την επιχειρησιακή λογική ή τον σκοπό της ανάπτυξης.

kind (String): Καθορίζει τον τύπο ή την κατηγορία της περιγραφής, επιτρέποντας στο σύστημα να εφαρμόσει τις κατάλληλες πολιτικές και διαδικασίες ανάλογα με τη φύση της εφαρμογής. Αυτή η ιδιότητα συμβάλλει στην ταξινόμηση και οργάνωση των διαφορετικών τύπων περιγραφής εργασιών..

user_uuid (String): Συνδέει την περιγραφή με τον συγκεκριμένο χρήστη που την δημιούργησε, εξασφαλίζοντας την ιχνηλασιμότητα και την εφαρμογή των κατάλληλων πολιτικών ασφάλειας και εξουσιοδότησης.

application_uuid (String): Αναφέρεται στο μοναδικό αναγνωριστικό της εφαρμογής που πρόκειται να αναπτυχθεί, δημιουργώντας μια σαφή σύνδεση μεταξύ του Deployment και της υποκείμενης εφαρμογής. Αυτή η αναφορά επιτρέπει στο σύστημα να ανακτήσει όλες τις απαραίτητες πληροφορίες σχετικά με την εφαρμογή.

associations (List): Περιέχει τη λίστα των Associations στις οποίες θα πραγματοποιηθεί η ανάθεση της εφαρμογής για εκτέλεση. Αυτή η ιδιότητα είναι κεντρικής σημασίας για τον καθορισμό του εύρους και της κατανομής της ανάθεσης στο κατανεμημένο σύστημα.

deployments (List): Αποθηκεύει λεπτομερείς πληροφορίες για τα επιμέρους Deployments που συνθέτουν το συνολικό Associations Deployment. Κάθε στοιχείο αυτής της λίστας αντιπροσωπεύει ένα συγκεκριμένο στιγμιότυπο της εφαρμογής σε ένα συγκεκριμένο Association.

deployments_status (List): Παρακολουθεί την κατάσταση κάθε επιμέρους Deployment, παρέχοντας πληροφορίες σχετικά με την πρόοδο και την επιτυχία των λειτουργιών ανάθεσης και εκτέλεσης. Αυτό το πεδίο είναι κρίσιμο για την παρακολούθηση και διαχείριση της συνολικής διαδικασίας εξυπηρέτησης της εφαρμογής.

logs (List): Συλλέγει και διατηρεί logs από όλες τις λειτουργίες που σχετίζονται με το Associations Deployment, παρέχοντας ένα ολοκληρωμένο ιστορικό για ελέγχους αποσφαλμάτωσης (debugging), διαφάνειας (auditing) και συμμόρφωσης (compliance).

status (Integer): Αντιπροσωπεύει την τρέχουσα κατάσταση του Associations Deployment χρησιμοποιώντας έναν κωδικοποιημένο ακέραιο αριθμό. Αυτή η προσέγγιση επιτρέπει την αποδοτική αποθήκευση και σύγκριση καταστάσεων, ενώ παράλληλα διευκολύνει την υλοποίηση state machine λογικής.

updated_by (String): Καταγράφει την ταυτότητα του χρήστη ή του συστήματος που πραγματοποίησε την τελευταία ενημέρωση του αντικειμένου, συμβάλλοντας στην ιχνηλασιμότητα των αλλαγών και την εφαρμογή πολιτικών ελέγχου προσβάσεων.

created_at (Integer): Αποθηκεύει την χρονική στιγμή (timestamp) της δημιουργίας του Associations Deployment, παρέχοντας μια σαφή χρονική αναφορά για την έναρξη του κύκλου ζωής του αντικειμένου. Αυτή η πληροφορία είναι χρήσιμη για αναφορά στοιχείων (reporting), ανάλυση δεδομένων (analytics) και διαχείριση του κύκλου ζωής (lifecycle management).

updated_at (Integer): Καταγράφει το timestamp της τελευταίας τροποποίησης, επιτρέποντας στο σύστημα και στους χρήστες να παρακολουθούν την εξέλιξη και τις αλλαγές του Deployment. Αυτό το πεδίο είναι κρίσιμο για την εφαρμογή μηχανισμών κλειδώματος (optimistic locking) και μηχανισμών επίλυσης συγκρούσεων (conflict resolution).

Το Associations Deployment αποτελεί μια ολοκληρωμένη τεχνική λύση που επιτρέπει την διαχείριση και ανάθεση cloud-native εφαρμογών σε συνεργατικά και κατανεμημένα περιβάλλοντα πολλαπλών φορέων που οργανώνονται με βάση τα Associations. Η σχεδίασή του ενσωματώνει σύγχρονες αρχές τεχνολογίας λογισμικού (software engineering) και υπολογιστικού νέφους (cloud computing), παρέχοντας μια ισχυρή βάση για την υλοποίηση εξελιγμένων λύσεων ενορχήστρωσης (orchestration). Η πλούσια δομή των ιδιοτήτων του και ο καλά καθορισμένος κύκλος ζωής του το καθιστούν ιδανικό για την αντιμετώπιση των σύνθετων προκλήσεων που χαρακτηρίζουν τα σύγχρονα κατανεμημένα συστήματα.

Συμπληρωματικά Αντικείμενα της Αρχιτεκτονικής

Πέραν του κεντρικού Associations Deployment αντικειμένου, η προτεινόμενη επέκταση του Service Orchestrator ενσωματώνει δύο επιπλέον κρίσιμα αντικείμενα που συμπληρώνουν και ενισχύουν τη συνολική λειτουργικότητα του συστήματος. Αυτά τα αντικείμενα, το Application και το Association, αποτελούν τα θεμελιώδη δομικά στοιχεία που επιτρέπουν την διαχείριση cloud-native εφαρμογών σε κατανεμημένα περιβάλλοντα.

Το **Application** αποτελεί την αφηρημένη αναπαράσταση μιας cloud-native εφαρμογής εντός του κατανεμημένου συστήματος ενορχήστρωσης. Αυτό το αντικείμενο ενθυλακώνει όλες τις απαραίτητες πληροφορίες που χαρακτηρίζουν μια εφαρμογή ανεξάρτητα από το περιβάλλον ανάθεσης και εκτέλεσης της, παρέχοντας μια ενιαία και συνεκτική περιγραφή που μπορεί να αξιοποιηθεί σε διαφορετικά σενάρια εκτέλεσης.

Η σημασία του Application εκδηλώνεται μέσα από τη δυνατότητά του να λειτουργεί ως πρότυπο (template) για τη δημιουργία πολλαπλών Deployments σε διαφορετικές Associations. Αυτή η προσέγγιση εξασφαλίζει συνέπεια (consistency) και δυνατότητα αναπαραγωγής (reproducibility) στις διαδικασίες ανάπτυξης, ενώ ταυτόχρονα επιτρέπει την προσαρμογή σε τοπικές απαιτήσεις και περιορισμούς.

Παρακάτω παρουσιάζεται η δομή του αντικειμένου Application

Πεδίο	**Τύπος**
uuid	string
name	string
user_uuid	string
deployment_objectives	string
logs	list
status	integer
updated_by	string
created_at	integer
updated_at	integer

Το συγκεκριμένο αντικείμενο έχει τις παρακάτω παραμέτρους:

uuid (String): Το μοναδικό αναγνωριστικό που διασφαλίζει την απόλυτη μοναδικότητα της εφαρμογής εντός του συστήματος. Το αναγνωριστικό χρησιμοποιείται ως πρωτεύον κλειδί (primary key) για όλες τις αναφορές και συσχετίσεις με άλλα αντικείμενα του συστήματος.

name (String): Παρέχει μια περιγραφική ονομασία της εφαρμογής που διευκολύνει την αναγνώριση και διαχείριση από τους χρήστες. Αυτό το πεδίο συχνά αντικατοπτρίζει την επιχειρησιακή λογική ή τη λειτουργική περιγραφή της εφαρμογής.

user_uuid (String): Συνδέει την εφαρμογή με τον κάτοχό της, διασφαλίζοντας σαφή ιχνηλασιμότητα ιδιοκτησίας (proper ownership tracking) και την εφαρμογή των κατάλληλων

πολιτικών ελέγχου πρόσβασης (access control policies). Αυτή η σύνδεση είναι κρίσιμη για περιβάλλοντα πολλαπλών χρηστών και διαφορετικών κατηγοριών εφαρμογών.

description (String): Προσφέρει λεπτομερή περιγραφή της εφαρμογής, συμπεριλαμβανομένου του σκοπού, των λειτουργιών και των χαρακτηριστικών της.

deployment_objectives (String): Καθορίζει τις απαιτήσεις και τις προτεραιότητες του χρήστη σχετικά με την ανάθεση και εκτέλεση της εφαρμογής. Περιλαμβάνει πληροφορίες σχετικά με στόχους απόδοσης (performance targets), απαιτήσεις διαθεσιμότητας (availability requirements), περιορισμούς ασφαλείας (security constraints) και στόχους βελτιστοποίησης πόρων (resource optimization goals).

logs (List): Διατηρεί ένα πλήρες ιστορικό (comprehensive log history) που καταγράφει όλες τις σημαντικές λειτουργίες και συμβάντα (events) που σχετίζονται με την εφαρμογή. Αυτή η πληροφορία είναι κρίσιμη για εντοπισμό και διόρθωση σφαλμάτων (debugging), λογιστικό έλεγχο (auditing) και σκοπούς συμμόρφωσης (compliance purposes).

status (Integer): Αντιπροσωπεύει την τρέχουσα κατάσταση της εφαρμογής χρησιμοποιώντας έναν κωδικοποιημένο ακέραιο αριθμό που επιτρέπει αποδοτική κατηγοριοποίηση και φιλτράρισμα (filtering).

updated_by, created_at, updated_at: Παρέχουν βασικά μεταδεδομένα (essential metadata) για την παρακολούθηση των αλλαγών (tracking), τη διατήρηση ιστορικού ελέγχου (audit trails) και την υλοποίηση κατάλληλων μηχανισμών ελέγχου εκδόσεων (version control mechanisms).

Το Association αναπαριστά τα διαφορετικά Associations που συμμετέχουν στο κατανεμημένο σύστημα. Κάθε Association αποτελεί μια ανεξάρτητη οντότητα που διατηρεί τη δική της υποδομή, πολιτικές και πόρους, ενώ ταυτόχρονα συμμετέχει στο ευρύτερο οικοσύστημα ενός περιβάλλοντος πολλαπλών εμπλεκόμενων φορέων.

Η έννοια του Association είναι κεντρικής σημασίας για την υλοποίηση της αρχιτεκτονικής όπου διαφορετικοί οργανισμοί συνεργάζονται διατηρώντας την αυτονομία τους. Το αντικείμενο Association ενθυλακώνει όλες τις απαραίτητες πληροφορίες που χαρακτηρίζουν έναν οργανισμό ως στόχο ανάπτυξης (deployment target) και πάροχο πόρων (resource provider).

Παρακάτω παρουσιάζεται η δομή του αντικειμένου Association

Πεδίο	**Τύπος**
uuid	string
name	string
labels	list

configuration	string
clusters	list
logs	list
status	integer
created_at	integer
updated_at	integer

uuid (String): Το μοναδικό αναγνωριστικό που εξασφαλίζει την απόλυτη διάκριση κάθε Association εντός του συστήματος. Αυτό το αναγνωριστικό είναι κρίσιμο για τη δρομολόγηση (routing), την κατανομή πόρων (resource allocation) και την επιβολή πολιτικών (policy enforcement) σε ολόκληρο το κατανεμημένο σύστημα.

name (String): Προσφέρει μια ξεκάθαρη και αναγνωρίσιμη ονομασία για κάθε Association, διευκολύνοντας την επικοινωνία και την αναφορά μεταξύ των εμπλεκόμενων φορέων του συστήματος.

labels (List): Παρέχει έναν ευέλικτο μηχανισμό ταξινόμησης και κατηγοριοποίησης των Associations χρησιμοποιώντας ζεύγη κλειδιού-τιμής (key-value pairs) ή ετικέτες (tags). Αυτή η λειτουργικότητα επιτρέπει σύνθετη αναζήτηση (sophisticated querying), φιλτράρισμα (filtering) και εφαρμογή πολιτικών (policy application) βάσει των χαρακτηριστικών των Associations.

clusters (List): Καταγράφει τα διαθέσιμα clusters που ανήκουν στην Association. Αυτή η πληροφορία είναι κρίσιμη για τον προγραμματισμό πόρων (resource planning), τη διαχείριση χωρητικότητας (capacity management) και τις αποφάσεις χρονοπρογραμματισμού των deployments .

configuration (String): Ενσωματώνει τις τεχνικές παραμέτρους και πολιτικές που χαρακτηρίζουν την Association. Αυτό περιλαμβάνει ρυθμίσεις δικτύου (network configurations), πολιτικές ασφαλείας (security policies), ποσοστώσεις πόρων (resource quotas) και άλλες παραμέτρους λειτουργίας που επηρεάζουν τον τρόπο ανάπτυξης εφαρμογών.

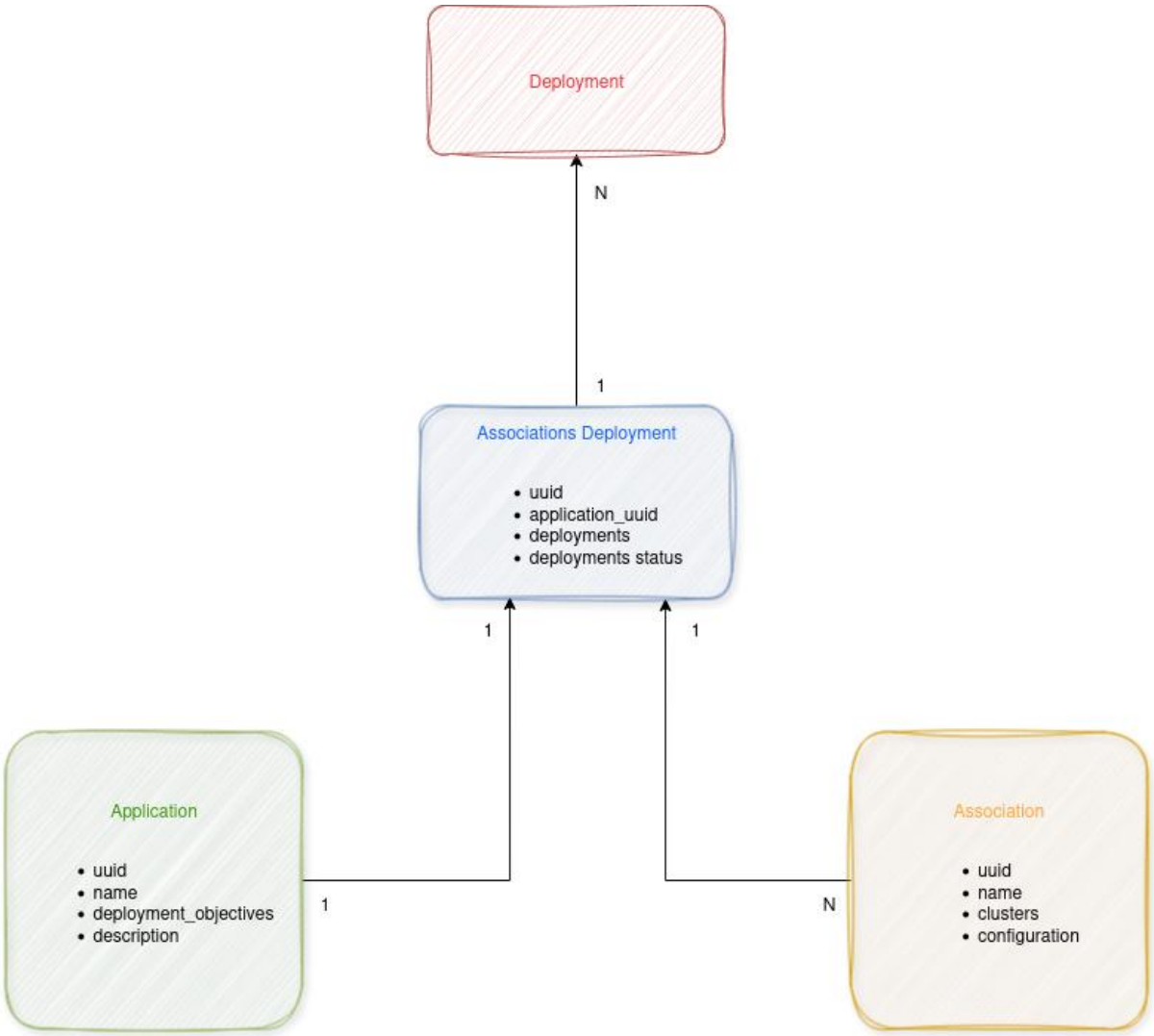
logs (List): Διατηρεί αναλυτικά αρχεία καταγραφής όλων των λειτουργιών και συμβάντων που σχετίζονται με την Association, παρέχοντας ορατότητα στις δραστηριότητες και την απόδοση του οργανισμού εντός του συστήματος.

status (Integer): Αναπαριστά την κατάσταση του Association, επιτρέποντας στο σύστημα να λαμβάνει τεκμηριωμένες αποφάσεις σχετικά με την καταλληλότητά της για νέα deployments.

created_at, updated_at: Παρέχουν χρονικά μεταδεδομένα που υποστηρίζουν τη διαχείριση κύκλου ζωής και την ανάλυση ιστορικών δεδομένων των Associations.

Σχέσεις και Αλληλεπιδράσεις

Ο ρόλος και η χρησιμότητα των νέων αντικειμένων (Application, Association, Associations Deployment) που προστέθηκαν στα πλαίσια της συγκεκριμένης εργασίας γίνεται πιο κατανοητός μέσω των σχέσεων και αλληλεπιδράσεων που υπάρχουν μεταξύ τους. Το Application παρέχει το "τι" της ανάθεσης και εκτέλεσης, το Association καθορίζει το "πού" σε επίπεδο συνεργατικών πόρων, και το Associations Deployment εκφράζει το "πώς" και το "πότε".



Εικόνα 6: Σχέσεις ανάμεσα στα αντικείμενα του Orchestration API

Αυτή η τριαδική σχέση επιτρέπει την υλοποίηση εξελιγμένων στρατηγικών ανάθεσης και εκτέλεσης για συνεργατικές και υπερ-κατανεμημένες υποδομές edge-cloud που μπορούν να προσαρμοστούν σε διαφορετικά πλαίσια οργάνωσης των υποδομών, τεχνικές απαιτήσεις και επιχειρηματικούς στόχους. Η αρθρωτή οργάνωση που προσφέρει το προτεινόμενο σχήμα

οργάνωσης επιτρέπει επίσης την ανεξάρτητη εξέλιξη και διαχείριση κάθε επιμέρους στοιχείου της εφαρμογής, ενισχύοντας την διατήρηση και επέκταση του συστήματος.

Η διαχείριση των Associations Deployments υλοποιείται μέσω ενός ολοκληρωμένου RESTful API που ακολουθεί τις βέλτιστες πρακτικές του web service design και παρέχει μια διεπαφή για την αλληλεπίδραση με το σύστημα ενορχήστρωσης. Το API έχει σχεδιαστεί με γνώμονα τις αρχές της REST αρχιτεκτονικής, εξασφαλίζοντας επεκτασιμότητα (scalability), ευκολία συντήρησης (maintainability) και διαλειτουργικότητα (interoperability).

3.3.2 Rest Endpoints

Το κεντρικό endpoint του API ακολουθεί τη δομή:

/api/v1/service_orchestrator/associations/deployments

Αυτή η ιεραρχική δομή αντικατοπτρίζει τη λογική οργάνωση του συστήματος, όπου το **service_orchestrator** αποτελεί το κύριο service, τα **associations** αντιπροσωπεύουν τις οντότητες πολλαπλών φορέων, και τα **deployments** τα αντικείμενα διαχείρισης ανάπτυξης εφαρμογών.

HTTP Verb	Endpoint	Περιγραφή
POST	"/api/v1/orchestrator/associations/deployments"	Δημιουργία νέου Associations Deployment.
GET	"/api/v1/orchestrator/associations/deployments"	Ανάκτηση όλων των υπαρχόντων Associations Deployments.
GET	"/api/v1/orchestrator/associations/deployments/{uuid}"	Ανάκτηση συγκεκριμένου Associations Deployment μέσω του μοναδικού αναγνωριστικού του.
GET	"/api/v1/orchestrator/associations/deployments/logs/{uuid}"	Ανάκτηση logging πληροφοριών που σχετίζονται με συγκεκριμένο Associations Deployment μέσω του μοναδικού αναγνωριστικού του.
PUT	"/api/v1/orchestrator/associations/deployments"	Ενημέρωση υπάρχοντος Associations Deployment
DELETE	"/api/v1/orchestrator/associations/deployments/{uuid}"	Διαγραφή συγκεκριμένου Associations Deployment

3.3.2.1 Λεπτομερής Ανάλυση των HTTP Methods και Endpoints

GET /api/v1/service_orchestrator/associations/deployments Αυτό το endpoint υλοποιεί τη λειτουργία ανάκτησης όλων των υπαρχόντων Associations Deployments στο σύστημα. Η επιστρεφόμενη απάντηση περιλαμβάνει μια ολοκληρωμένη λίστα με όλα τα εγγενώς σχεδιασμένα για το υπολογιστικό νέφος (cloud-native) deployments εφαρμογών που είναι ενεργά σε όλα τα Associations. Αυτή η λειτουργία είναι κρίσιμη για εφαρμογές πίνακα ελέγχου (dashboard applications), εργαλεία παρακολούθησης (monitoring tools) και διαχειριστικές διεπαφές (administrative interfaces) που χρειάζονται μια συνολική εικόνα του συστήματος.

GET /api/v1/service_orchestrator/associations/deployments/{uuid} Παρέχει λεπτομερείς πληροφορίες για ένα συγκεκριμένο Associations Deployment μέσω του μοναδικού αναγνωριστικού του. Αυτό το endpoint επιτρέπει την ανάκτηση αναλυτικών μεταδεδομένων, συμπεριλαμβανομένων των αρχείων καταγραφής (logs) των deployments, πληροφοριών κατάστασης (status information) και λεπτομερειών παραμετροποίησης (configuration details). Είναι ιδιαίτερα χρήσιμο για εντοπισμό και διόρθωση σφαλμάτων (debugging), παρακολούθηση (monitoring) και λεπτομερή επιθεώρηση των συγκεκριμένων deployments.

GET /api/v1/service_orchestrator/associations/deployments/logs/{uuid} Ειδικευμένο endpoint για την ανάκτηση των πληροφοριών καταγραφής που σχετίζονται με ένα συγκεκριμένο deployment. Αυτή η λειτουργία παρέχει πρόσβαση σε καταγραφές σε πραγματικό χρόνο και ιστορικά αρχεία καταγραφής, επιτρέποντας στους προγραμματιστές και τους διαχειριστές συστημάτων να παρακολουθούν την εκτέλεση, να εντοπίζουν προβλήματα και να αναλύουν την απόδοση των deployments σε λεπτομερές επίπεδο.

POST /api/v1/service_orchestrator/associations/deployments Αυτό το κρίσιμο endpoint χειρίζεται τη δημιουργία νέων εφαρμογών σχεδιασμένων για το υπολογιστικό νέφος (cloud-native application deployments) σε πολλαπλά Associations. Η αίτηση περιλαμβάνει όλες τις απαραίτητες προδιαγραφές, συμπεριλαμβανομένων των ρυθμίσεων της εφαρμογής, των πολιτικών ανάπτυξης (deployment policies) και των στόχων που ορίζονται από τον χρήστη (user-defined objectives). Το σύστημα επεξεργάζεται αυτές τις πληροφορίες και ξεκινάει την αυτοματοποιημένη διαδικασία ανάπτυξης σε όλα τα καθορισμένα Associations.

PUT /api/v1/service_orchestrator/associations/deployments Επιτρέπει την ενημέρωση των ήδη υπαρχόντων ρυθμίσεων των deployments. Αυτή η λειτουργία είναι ουσιώδης για σενάρια όπου οι απαιτήσεις αλλάζουν, οι ρυθμίσεις χρειάζονται λεπτομερή προσαρμογή, ή όταν πρέπει να γίνουν σταδιακές ενημερώσεις στις εφαρμογές. Το endpoint υποστηρίζει μερικές ενημερώσεις, επιτρέποντας την τροποποίηση συγκεκριμένων πεδίων χωρίς να επηρεάζεται η υπόλοιπη παραμετροποίηση.

DELETE /api/v1/service_orchestrator/associations/deployments/{uuid} Υλοποιεί τη λειτουργία τερματισμού και καθαρισμού των deployments. Αυτό το endpoint ξεκινάει μια ελεγχόμενη διαδικασία απόσυρσης που περιλαμβάνει την απομάκρυνση των εκτελούμενων εφαρμογών από τα καθορισμένα Associations, τον καθαρισμό των πόρων και την ενημέρωση

των σχετικών μεταδεδομένων. Η διαδικασία σχεδιάστηκε να είναι ομαλή, εξασφαλίζοντας την ελάχιστη διατάραξη στις υπηρεσίες που εκτελούνται.

Το API ενσωματώνει ισχυρούς μηχανισμούς ασφάλειας που περιλαμβάνουν έλεγχο ταυτότητας (authentication), έλεγχο εξουσιοδότησης (authorization) και καταγραφή ελέγχου (audit logging). Κάθε αίτημα υπόκειται σε επικύρωση και ελέγχους πρόσβασης που εξασφαλίζουν ότι μόνο εξουσιοδοτημένοι χρήστες μπορούν να εκτελέσουν τις σχετικές λειτουργίες.. Το σύστημα υποστηρίζει έλεγχο πρόσβασης βάσει ρόλων (RBAC) και λεπτομερή δικαιώματα που επιτρέπουν στους διαχειριστές να ορίσουν λεπτομερείς πολιτικές προσβάσεων.

Όλα τα API endpoints επιστρέφουν δομημένες αποκρίσεις σε μορφή JSON που ακολουθούν συνεπείς κανόνες μορφοποίησης. Το σύστημα υλοποιεί ολοκληρωμένο χειρισμό σφαλμάτων με περιγραφικά μηνύματα σφάλματος, κατάλληλους κώδικες κατάστασης HTTP (HTTP status codes) και λεπτομερή συμφραζόμενα σφάλματος που διευκολύνουν τον εντοπισμό και την διόρθωση προβλημάτων. Οι αποκρίσεις περιλαμβάνουν μεταδεδομένα όπως χρονικές σημάνσεις (timestamps), αναγνωριστικά αιτημάτων και πληροφορίες σελιδοποίησης όπου είναι εφαρμόσιμο.

Το Associations Deployment αποτελεί μια ολοκληρωμένη προσέγγιση στη διαχείριση cloud-native εφαρμογών σε κατανεμημένα, συνεργατικά περιβάλλοντα πολλαπλών φορέων. Η σχεδίαση του ενσωματώνει σύγχρονες αρχές τεχνολογίας λογισμικού και υπολογιστικού νέφους, παρέχοντας μια ισχυρή βάση για την υλοποίηση εξελιγμένων λύσεων ενορχήστρωσης. Η πλούσια δομή των ιδιοτήτων του, ο καλά καθορισμένος κύκλος ζωής του και το ολοκληρωμένο RESTful API το καθιστούν ιδανικό για την αντιμετώπιση των σύνθετων προκλήσεων που χαρακτηρίζουν τα σύγχρονα κατανεμημένα συστήματα. Η συνδυασμένη προσέγγιση που περιλαμβάνει τόσο τη δομή δεδομένων όσο και τη διεπαφή του API δημιουργεί ένα ισχυρό και ευέλικτο πλαίσιο για την αυτοματοποιημένη ενορχήστρωση εφαρμογών σχεδιασμένων για το υπολογιστικό νέφος.

Σημαντικό κομμάτι της υλοποίησης του Service Orchestrator είναι η παρακολούθηση συγκεκριμένων Datastore Topics (keys) από τον Orchestration Manager για τυχόν ενημερώσεις στα αντικείμενα του API με σκοπό τον κατάλληλο συντονισμό στα πλαίσια του συστήματος.

Τα υπάρχοντα topics αναφέρονται στον ακόλουθο πίνακα:

API Object	**Topic**
Deployment	/service/orchestrator/deployments/deployment/uuid
Storage Policy	/service/orchestrator/storage_policies/policy/uuid

Cluster	/service/orchestrator/clusters/cluster/uuid
Assignment	/service/orchestrator/assignments/cluster_uuid/assignment/uuid
Bundle	/service/orchestrator/bundles/bundle/uuid

Κατά τον ίδιο τρόπο για την παρακολούθηση των Associations Deployment το Datastore Key που εξυπηρετεί τους σκοπούς της υπάρχουσας εργασίας είναι το /service/orchestrator/associations_deployment/uuid

Κεφάλαιο 4: Υλοποίηση

Όπως έχει αναφερθεί η σχεδίαση ενός κατανεμημένου συστήματος αποτελούμενο από Associations για την αυτοματοποιημένη ενορχήστρωση και εκτέλεση (deployment) cloud-native εφαρμογών είναι ο κύριος άξονας της διπλωματικής εργασίας. Για τον σκοπό αυτό πραγματοποιήθηκαν αλλαγές στον κώδικα των σημαντικότερων στοιχείων της αρχιτεκτονικής του συστήματος.
Το Orchestrator API, αποτελεί το σημείο εισόδου για όλες τις αλληλεπιδράσεις με το σύστημα. Είναι υλοποιημένο με την χρήση του FastAPI [17] framework της Python και παρέχει RESTful μεθόδους που επιτρέπουν την υποβολή αιτημάτων για εκτέλεση εφαρμογών και γενικότερα για παρακολούθηση και διαχείριση του κύκλου ζωής των εφαρμογών. Στον κώδικα προστέθηκαν μέθοδοι που διαχειρίζονται κλήσεις που πραγματοποιούνται στα endpoints που αφορούν τα Associations Deployments. Για την διαχείριση των HTTP POST αιτημάτων που αφορούν την δημιουργία νέων Associations Deployments, δημιουργήθηκε η μέθοδος ***post_associations_deployment***.

```python
@app.post("/api/v1/orchestrator/associations/deployments", status_code=201)
async def post_associations_deployment(associationsDeployment: AssociationsDeployment):
    logger.info("Incoming request for new application associations deployment ...")
    associationsDeployment_uuid = str(uuid.uuid4())
    logger.debug("Assigned associations deployment uuid '%s'" % associationsDeployment_uuid)
    data = associationsDeployment.dict()
    if not associationsDeployment.name:
        data["name"] = associationsDeployment_uuid
    data["associationsDeployment_uuid"] = associationsDeployment_uuid
    data["kind"] = "AssociationsDeployment"

    # Validate deployments inside AssociationsDeployment
    for idx, deployment in enumerate(data.get("deployments", [])):
        if "name" not in deployment:
            deployment["name"] = f"deployment-{idx}"
        if "deployment_description" not in deployment:
            return {"error": f"Deployment {idx} missing deployment_description"}, 400

    self.__dispatcher.create_associations_deployment(data)
    return {"associations_deployment_uuid": associationsDeployment_uuid}
```

Εικόνα 7: Μέθοδος post_associations_deployment

Η μέθοδος αυτή λαμβάνει ως παράμετρο ένα αντικείμενο τύπου Associations Deployment και σε περίπτωση επιτυχίας επιστρέφει μια απάντηση με κωδικό κατάστασης (Status Code) 201.

Στην αρχή, φροντίζουμε να εξάγουμε όλες τις απαραίτητες πληροφορίες από το εισερχόμενο αίτημα. Δημιουργούμε ένα μοναδικό αναγνωριστικό (UUID) για το νέο AssociationsDeployment και συλλέγουμε τα δεδομένα περιγραφής των microservices της εφαρμογής από το αντίστοιχο πεδίο του αιτήματος (deployment_description).

Κατόπιν, εφαρμόζεται ένας έλεγχος επαλήθευσης των δεδομένων, ώστε να διασφαλιστεί η ποιότητα των πληροφοριών πριν την περαιτέρω επεξεργασία τους. Συγκεκριμένα, ελέγχουμε εάν το αίτημα εξυπηρέτησης περιέχει τα υποχρεωτικά πεδία *"name"* και *"deployment_description"*. Σε περίπτωση που λείπει το *"deployment_description"* διακόπτεται άμεσα η διαδικασία και επιστρέφεται ένα σφάλμα με κωδικό 400, συνοδευόμενο από ένα σαφές και κατανοητό μήνυμα που εξηγεί στον χρήστη το ακριβές πρόβλημα.

Αφού επιβεβαιώσουμε ότι όλα τα δεδομένα είναι έγκυρα και πλήρη, προχωράμε στο τελικό στάδιο της διαδικασίας. Καλούμε τον *Dispatcher* για να πραγματοποιήσει τη δημιουργία του αντικειμένου Associations Deployment στην Datastore και επιστρέφουμε το μοναδικό αναγνωριστικό του δημιουργημένου αντικειμένου.

Με αυτόν τον τρόπο, διασφαλίζουμε ότι η μέθοδός ακολουθεί μια οργανωμένη και αξιόπιστη δομή που εγγυάται την ακεραιότητα των δεδομένων και παρέχει ξεκάθαρη ανατροφοδότηση σε περίπτωση σφάλματος.

Αμέσως μετά το στρώμα API βρίσκεται ο Dispatcher, ο οποίος λειτουργεί ως ο κεντρικός συντονιστής της αρχικής φάσης επεξεργασίας των αιτημάτων. Ο ρόλος του Dispatcher είναι πολυδιάστατος, καθώς αναλαμβάνει την προετοιμασία των αντικειμένων για την ενορχήστρωση και εκτέλεση (deployment), τον εμπλουτισμό τους με απαραίτητα μεταδεδομένα, την αρχικοποίηση των δομών παρακολούθησης και την αποθήκευση τους στην Datastore που βασίζεται στο κατανεμημένο σύστημα αποθήκευσης etcd. Η χρήση του etcd ως κεντρικού αποθετηρίου δεδομένων εξασφαλίζει την υψηλή διαθεσιμότητα και τη συνέπεια των δεδομένων σε όλο το κατανεμημένο σύστημα.

Στον κώδικα του Dispatcher υλοποιήσαμε την μέθοδο ***create_associations_deployment***. Η μέθοδος αυτή αποτελεί το κεντρικό σημείο όπου συντονίζονται όλες οι απαραίτητες ενέργειες για την αρχικοποίηση και αποθήκευση ενός νέου Associations Deployment.

```python
def create_associations_deployment(self, params):

    #create application
    application = self.create_application(params)
    self.__etcdClient.put("/service/orchestrator/applications/application/%s" % application["application_uuid"], json.dumps(application))

    params["application_uuid"] = application["application_uuid"]
    params["associations"] = []
    params["deployments_status"] = []
    params["logs"] = [{"timestamp": int(time.time()), "event": "Associations Deployment description received."}]
    params["status"] = status.AssociationsDeployment.SUBMITTED
    params["updated_by"] = "Orchestration.API"
    params["created_at"] = int(time.time())
    params["updated_at"] = int(time.time())

    # Clean deployment descriptions
    for deployment in params.get("deployments", []):
        if "deployment_description" in deployment:
            deployment["deployment_description"] = deployment["deployment_description"].replace("\\r", "")

    logger.info("Cleaned deployment descriptions")

    logger.debug("Create Associations Deployment API object for deployment '%s'" % params["associationsDeployment_uuid"])

    self.__etcdClient.put("/service/orchestrator/associations/deployments/deployment/%s" % params["associationsDeployment_uuid"], json.dumps(params))
```

Εικόνα 8: Μέθοδος create_associations_deployment

Η διαδικασία ξεκινάει με την δημιουργία ενός νέου Application με την χρήση της βοηθητικής μεθόδου ***create_application***.

```
def create_application(self, associationsDeploymentParams):
    try:
        application = Application()
        print(f"Created application: {application}")
        print(f"Application dict: {application.dict()}")
        application_uuid = str(uuid.uuid4())
        application_data = application.dict()
        if not application.name:
            application_data["name"] = application_uuid
        application_data["application_uuid"] = application_uuid
        application_data["kind"] = "Application"
        application_data["description"] = application_data["description"].replace("\\r", "")
        application_data["deployment_objectives"] = associationsDeploymentParams["deployments"]
        application_data["logs"] = [{"timestamp": int(time.time()), "event": "Application created."}]
        application_data["status"] = status.Application.SUBMITTED
        application_data["updated_by"] = "Orchestration.API"
        application_data["created_at"] = int(time.time())
        application_data["updated_at"] = int(time.time())
    except Exception as e:
        print(f"Error creating application: {e}")
        raise

    return application_data
```

Εικόνα 9: Μέθοδος create_application

Στη συνέχεια, εμπλουτίζουμε τις παραμέτρους που λάβαμε με επιπλέον μεταδεδομένα που είναι απαραίτητα για τη σωστή λειτουργία του συστήματος. Συγκεκριμένα, προσθέτουμε το μοναδικό αναγνωριστικό του αντικειμένου Application που μόλις δημιουργήθηκε, αρχικοποιούμε κενές λίστες για τα Associations και τα deployment statuses, και δημιουργούμε τη πρώτη εγγραφή καταγραφής (logs) που καταγράφει την παραλαβή της περιγραφής του Association Deployment.

Επίσης, ορίζουμε τον ορισμό της αρχικής κατάστασης του αιτήματος ενορχήστρωσης και εκτέλεσης (deployment) ως "SUBMITTED" και την καταγραφή των χρονικών σημάνσεων δημιουργίας και τελευταίας ενημέρωσης. Για λόγους παρακολούθησης, σημειώνουμε ότι η ενημέρωση έγινε από το "Orchestration.API".

Ένα σημαντικό βήμα είναι ο καθαρισμός των Deployment Descriptions, αφαιρώντας τυχόν ανεπιθύμητους χαρακτήρες όπως τα "\r" που μπορεί να προκαλέσουν προβλήματα κατά την επεξεργασία.

Τέλος, αποθηκεύουμε όλες τις πληροφορίες του Association Deployment στην Datastore, χρησιμοποιώντας το μοναδικό του αναγνωριστικό ως κλειδί.

Το λειτουργικό τμήμα Orchestration API Interface αποτελεί ένα κρίσιμο στοιχείο της συνολικής αρχιτεκτονικής του Service Orchestrator, και αποτελεί κομμάτι της υπηρεσίας Orchestration Manager. Η λειτουργία του βασίζεται στο PyQt framework [33] και παρέχει έναν αποδοτικό μηχανισμό επικοινωνίας βασισμένης σε συμβάντα μεταξύ των διαφόρων στοιχείων του συστήματος. Μέσω της χρήσης του μηχανισμού παρακολούθησης που προσφέρει η Datastore, το συγκεκριμένο λειτουργικό τμήμα παρακολουθεί συνεχώς για νέα αιτήματα

εξυπηρέτησης (deployments) και παράγει ειδοποιήσεις όταν εντοπίζει αλλαγές, επιτρέποντας έτσι την ασύγχρονη και αποσυζευγμένη επικοινωνία μεταξύ των συστατικών μονάδων.

Στον κώδικα του αρχείου ***orchestrationAPIInterface.py*** κάθε αντικείμενο έχει τα δικά του κλειδιά και τους αντίστοιχους παρατηρητές.

```
self.__etcdClient.add_watch_prefix_callback("/service/orchestrator/deployments/deployment/",
                                            self.__etcd_watch_callback)
self.__etcdClient.add_watch_prefix_callback("/service/orchestrator/kernels/kernel/",
                                            self.__etcd_watch_kernels_callback)
self.__etcdClient.add_watch_prefix_callback("/service/orchestrator/storage_policies/policy/",
                                            self.__etcd_watch_storage_policies_callback)
```

Εικόνα 10: Watcher για κλειδιά με prefix των υφιστάμενων αντικειμένων

Κατά τον ίδιο τρόπο προσθέσαμε έναν παρατηρητή για την Datastore ώστε να παρακολουθούμε τις αλλαγές σε όλα τα κλειδιά που ξεκινούν με το πρόθεμα **"/service/orchestrator/associations/deployments/deployment/"**.

```
self.__etcdClient.add_watch_prefix_callback("/service/orchestrator/associations/deployments/deployment/",
                                            self.__etcd_watch_associations_deployment_callback)
```

Εικόνα 11: Watcher για κλειδιά με prefix σχετικό με Associations Deployment

Ο παρατηρητής καλεί την μέθοδο ***__etcd_watch_associations_deployment_callback*** κάθε φορά που γίνεται κάποια αλλαγή σε αυτά τα κλειδιά.

```
def __etcd_watch_associations_deployment_callback(self, etcd_event):
    logger.info("Associations Deployment event(s)...")
    for event in etcd_event.events:
        value = event.value.decode("utf-8")
        if len(value) == 0: # Delete event
            logger.info("Termination event for key '%s'" % event.value.decode("utf-8"))
            return
        else:
            event_data = json.loads(value)
            if event_data["updated_by"] == "Orchestration.API":
                logger.info("Associations Deployment event for key '%s'" % event.key.decode("utf-8"))
                print("Associations Deployment event for key '%s'" % event.key.decode("utf-8"))
                self.associationsDeploymentRequest.emit(event_data)
```

Εικόνα 12: Μέθοδος etcd_watch_associations_deployment_callback

Αρχικά, ελέγχουμε το είδος της αλλαγής που έχει γίνει. Αν διαπιστώσουμε ότι κάποιο deployment έχει διαγραφεί (το οποίο γίνεται αντιληπτό από το γεγονός ότι το περιεχόμενο είναι κενό), καταγράφουμε αυτή την πληροφορία μέσω logging και τερματίζουμε την επεξεργασία.

Όταν πρόκειται για νέο αίτημα εξυπηρέτησης, μετατρέπουμε τα δεδομένα JSON σε μορφή που μπορεί να υποβληθεί προς επεξεργασία. Στην συνέχεια, εφαρμόζεται ένας έλεγχος για να διασφαλιστεί ότι η αλλαγή προέρχεται από το σωστό σύστημα και πιο συγκεκριμένα από το “Orchestration.API”.

Αυτός ο έλεγχος είναι σημαντικός γιατί στο κατανεμημένο σύστημα που αναπτύξαμε, διάφορα υποσυστήματα μπορεί να κάνουν αλλαγές στα ίδια δεδομένα. Εμείς όμως θέλουμε να πυροδοτούνται αντιδράσεις μόνο στις αλλαγές που κάνει το κεντρικό API του orchestrator.

Όταν επιβεβαιώσουμε ότι η αλλαγή είναι έγκυρη, στέλνουμε ένα σήμα στα άλλα μέρη του συστήματος μέσω της μεθόδου ***emit()***.

Συγκεκριμένα, εκπέμπεται το σήμα ***associationsDeploymentRequest*** το οποίο μέσω της σύνδεσης που έχουμε ορίσει με την μέθοδο **connect()** στο ***orchestrationManagerInstance.py***, προωθείται αυτόματα στην μέθοδο ***handle_associations_deployment_request()*** του Orchestration Manager που θα παρουσιαστεί στην συνέχεια.

```
self.orchestratorAPIInterface.associationsDeploymentRequest.connect(self.orchestrationManager.handle_associations_deployment_request)
```

Εικόνα 13: Σύνδεση σήματος associationsDeploymentRequest με Orchestration Manager

Ο Orchestration Manager είναι υπεύθυνος για την επεξεργασία των αιτημάτων ανάπτυξης, τη λήψη αποφάσεων σχετικά με την τοποθέτηση των εφαρμογών, την επικοινωνία με εξωτερικά συστήματα όπως το Decision Engine και τη διαχείριση του κύκλου ζωής των αιτημάτων ενορχήστρωσης και εκτέλεσης (deployments). Η σχεδίαση του επιτρέπει την εύκολη επέκταση με νέες λειτουργίες και την προσαρμογή σε διαφορετικές απαιτήσεις χωρίς να επηρεάζεται η βασική λειτουργικότητα.

Στον κώδικα του αρχείου ***orchestrationManager.py*** έγινε προσθήκη της μεθόδου ***handle_associations_deployment_request()***. Η μέθοδος αυτή καλείται κάθε φορά που εκπέμπεται το σήμα ***associationsDeploymentRequest*** όπως είδαμε προηγουμένως.

```
def handle_associations_deployment_request(self, request):
    logger.info("Handle associations deployment request...")
    logger.debug(json.dumps(request))

    # Update status to PENDING
    self.orchestrationManagerLogInfo.emit({
        "uuid": request["associationsDeployment_uuid"],
        "kind": requestType.ASSOCIATIONS_DEPLOYMENT,
        "status": status.AssociationsDeployment.PENDING,
        "logs": [{"timestamp": int(time.time()),
                  "event": "Set Associations Deployment status to Pending."}]
    })

    if not self.__rotInterface.schedule_associations_deployment(request):
                self.orchestrationManagerLogInfo.emit({"uuid": request["deployment_uuid"],
                                                       "kind": requestType.ASSOCIATIONS_DEPLOYMENT,
                                                       "status": status.AssociationsDeployment.FAILED,
                                                       "logs": [{"timestamp": int(time.time()),
                                                                 "event": "Submission to Decision Engine failed"}]})
```

Εικόνα 14: Μέθοδος handle_associations_deployment_request

Όπως φαίνεται και από την εικόνα 14, η μέθοδος αυτή χειρίζεται το αίτημα του Associations Deployment στην πλευρά του Orchestration Manager. Αρχικά καταγράφουμε τις πληροφορίες του αιτήματος.

Στην συνέχεια, με την χρήση του σήματος ***orchestrationManagerLogInfo*** ενημερώνουμε την κατάσταση του Associations Deployment ως “PENDING” καταγράφοντας την χρονική στιγμή και μια σχετική περιγραφή για το συμβάν.

Κατόπιν, καλείται η μέθοδος ***schedule_associations_deployment()*** που δημιουργήσαμε στο αρχείο ***rotInterface.py***. Μέσω της μεθόδου αυτής ο Service Orchestrator επικοινωνεί με την εξωτερική υπηρεσία Decision Engine με σκοπό να λάβουμε τις αποφάσεις ενορχήστρωσης μέσω εξειδικευμένων αλγορίθμων. Στην συνέχεια ο Service Orchestrator αντιστοιχίζει τις αποφάσεις αυτές σε συγκεκριμένες ενέργειες που εξασφαλίζουν την κατάλληλη κατανομή των microservices του Associations Deployment στα επιμέρους Deployments του υφιστάμενου συστήματος.

Για την υλοποίηση των επεκτάσεων της διπλωματικής δεν χρειάστηκε από την πλευρά μας οποιαδήποτε επέκταση ή αλλαγή της λειτουργίας της Decision Engine καθώς μια έκδοση που υποστήριζει σχετικές αποφάσεις παραχωρήθηκε από το εργαστήριο HSCNL με σκοπό να χρησιμοποιηθεί για την ολοκλήρωση της διπλωματικής εργασίας. Έτσι στα πλαίσια της συγκεκριμένης εργασίας, η Decision Engine λειτουργεί σαν ένα εξωτερικό σύστημα και επικεντρωθήκαμε στην υλοποίηση των κατάλληλων ενεργειών για την αξιοποίηση αυτών των αποφάσεων στα πλαίσια της επέκτασης της λειτουργία του Service Orchestrator.

Για την διαχείριση των απαντήσεων που λαμβάνουμε από την Decision Engine, χρησιμοποιούμε την μέθοδο ***__handle_rot_response*** που βρίσκεται στο αρχείο ***orchestrationManager.py***.

```
def __handle_rot_response(self, response):

    if not response:
        logger.error("Invalid Decision Engine response (None), unable to serve the deployment request.")
        return None

    if "kind" not in response:
        logger.error("Invalid Decision Engine response (not include kind parameter), unable to server the deployment

    if response["kind"] == requestType.SERRANO_KERNEL:
        self.__kernel_request_response(response)
    elif response["kind"] == requestType.SERRANO_FaaS:
        self.__faas_request_response(response)
    elif response["kind"] == requestType.SERRANO_STORAGE_POLICY:
        self.__storage_policy_request_response(response)
    elif response["kind"] == requestType.SERRANO_DEPLOYMENT:
        self.__deployment_request_response(response)
    elif response["kind"] == requestType.ASSOCIATIONS_DEPLOYMENT:
        self.__associations_deployment_request_response(response)
```

Εικόνα 15: Μέθοδος __handle_rot_response

Η μέθοδος αυτή λειτουργεί ως κεντρικός διαχειριστής που αναλύει την απάντηση και την κατευθύνει στην κατάλληλη μέθοδο ανάλογα με τον τύπο του αιτήματος. Αρχικά εφαρμόζουμε βασικούς ελέγχους εγκυρότητας για να διασφαλιστεί η χρησιμότητα της απάντησης που λάβαμε. Ελέγχουμε εάν η απάντηση είναι κενή (None) και εάν περιέχει το απαραίτητο πεδίο *"kind"* που μας επιτρέπει να προσδιορίσουμε τον τύπο του αιτήματος. Σε περίπτωση που κάποιος από αυτούς τους ελέγχους αποτύχει, καταγράφουμε το αντίστοιχο σφάλμα και διακόπτουμε την επεξεργασία.

Όταν η Decision Engine μας επιστρέφει μια απάντηση όπου το πεδίο *"kind"* έχει τιμή "*requestType.ASSOCIATIONS_DEPLOYMENT*" κατευθύνουμε την ροή εκτέλεσης στην μέθοδο ***__associations_deployment_request_response***. Αυτή η μέθοδος αποτελεί τον κεντρικό μηχανισμό μέσω του οποίου διαχειριζόμαστε και κατανέμουμε τις επιμέρους περιγραφές εκτέλεσης (deployments) στους κατάλληλους ενορχηστρωτές.

```
def __associations_deployment_request_response(self, response):
    associationsDeployment = response["associationsDeployment"]
    logger.debug("About to fetch associations")

    associations = self.get_all_associations()
```

Εικόνα 16: Αρχικό σημείο της μεθόδου __associations_deployment_request_response

Αρχικά εξάγουμε όλα τα δεδομένα του Associations Deployment από την απάντηση και ανακτούμε όλα τα διαθέσιμα Associations από την Datastore μέσω της μεθόδου ***get_all_associations()***.

```
def get_all_associations(self):
    associations = {}

    # Get all key-value pairs with the prefix
    for value, metadata in etcdClient.get_prefix("/service/orchestrator/associations/association/"):
        # Extract the UUID from the key
        association_id = metadata.key.decode('utf-8').split('/')[-1]

        # Parse the JSON data
        association_data = json.loads(value.decode('utf-8'))

        # Store in dictionary with UUID as key
        associations[association_id] = association_data

    return associations
```

Εικόνα 17: Μέθοδος get_all_associations

Στην συνέχεια, οργανώνουμε τα deployments σε ομάδες βάσει του group_id τους χρησιμοποιώντας τη μέθοδο ***group_objects_by_group_id***.

```
grouped_deployments = self.group_objects_by_group_id(associationsDeployment)
```

Εικόνα 18: Σημείο κλήσης μεθόδου group_objects_by_group_id

Η μέθοδος αυτή αναλύει τις YAML περιγραφές για κάθε deployment και τα κατηγοριοποιεί βάσει του πεδίου *group_id* που βρίσκεται στο πεδίο *metadata.labels*.

```
def group_objects_by_group_id(self, associationsDeployment):
    grouped_objects = {}

    if 'deployments' not in associationsDeployment:
        return grouped_objects

    for deployment in associationsDeployment['deployments']:
        try:
            yaml_content = yaml.safe_load(deployment['deployment_description'])

            if (yaml_content and
                'metadata' in yaml_content and
                'labels' in yaml_content['metadata'] and
                'group_id' in yaml_content['metadata']['labels']):

                group_id = yaml_content['metadata']['labels']['group_id']

                # Initialize the group if it doesn't exist
                if group_id not in grouped_objects:
                    grouped_objects[group_id] = []

                # Add the deployment description to the group
                grouped_objects[group_id].append(deployment)

        except yaml.YAMLError as e:
            print(f"Error parsing YAML for deployment {deployment.get('name', 'unknown')}: {e}")
            continue

    return grouped_objects
```

Εικόνα 19: Μέθοδος group_objects_by_group_id

Το επόμενο στάδιο περιλαμβάνει την επεξεργασία κάθε Assignment που περιέχεται στην απάντηση. Για κάθε Assignment, συλλέγουμε όλες τις περιγραφές των deployments που ανήκουν στα group_ids που καθορίζονται στο πεδίο “deployments” των Assignments.

```
for assignment in response["assignments"]:
    association_uuid = assignment["association_uuid"]
    deployments = assignment["deployments"]

    # Collect all deployment descriptions for this assignment
    all_deployment_descriptions = []

    for deployment in deployments:
        group_id = deployment["group_id"]

        # Get all deployment descriptions for this group_id
        if group_id in grouped_deployments:
            for deployment_obj in grouped_deployments[group_id]:
                all_deployment_descriptions.append(deployment_obj['deployment_description'])
```

Εικόνα 20: Συλλογή περιγραφών των deployments

Κατόπιν δημιουργούμε μια ανανεωμένη και ενοποιημένη YAML περιγραφή που περιέχει όλες τις περιγραφές των microservices που θα έχουν ανατεθεί για εκτέλεση στο εκάστοτε Association. Οι συγκεκριμένες πληροφορίες περιέχονται στην απάντηση που λαμβάνει ο Service Orchestrator από την Decision Engine, και για την υλοποίηση της παραπάνω διαδικασίας έχει δημιουργηθεί η μέθοδος ***create_combined_yaml()***.

```
# Find the corresponding association
if association_uuid in associations:
    association_data = associations[association_uuid]
    print(f"Found association: {association_data['name']}")

    # Create combined YAML from all deployment descriptions
    combined_yaml = self.create_combined_yaml(all_deployment_descriptions)
```

Εικόνα 21: Σημείο κλήσης μεθόδου create_combined_yaml

Η μέθοδος αυτή επί της ουσίας λαμβάνει μια λίστα με περιγραφές σε YAML μορφή και τις συνδυάζει σε μια ενιαία περιγραφή που μπορεί να αξιοποιηθεί στο επόμενο στάδιο για την εκτέλεση των εφαρμογών σε κάθε Association..

```
def create_combined_yaml(self, deployment_descriptions):
    """Combine multiple deployment descriptions into a single YAML"""
    combined_docs = []

    for description in deployment_descriptions:
        try:
            # Parse each YAML description
            yaml_doc = yaml.safe_load(description)
            combined_docs.append(yaml_doc)
        except yaml.YAMLError as e:
            print(f"Error parsing YAML: {e}")
            continue

    # Create a single YAML with multiple documents
    combined_yaml = yaml.dump_all(combined_docs, default_flow_style=False)
    return combined_yaml
```

Εικόνα 22: Μέθοδος create_combined_yaml

Ένα κρίσιμο σημείο είναι η λήψη απόφασης σχετικά με τον τρόπο εκτέλεσης των deployments. Αφού συλλέξουμε όλες τις περιγραφές των deployments και δημιουργήσουμε την ενοποιημένη YAML περιγραφή, πρέπει να καθορίσουμε που θα εκτελεστούν αυτά τα deployments. Αυτή η απόφαση βασίζεται στην απάντηση που έχουμε λάβει από την Decision Engine για κάθε Association.

Ελέγχουμε την παρουσία του πεδίου *"configuration"* στα δεδομένα του Association. Αυτό το πεδίο λειτουργεί ως δείκτης που καθορίζει την πορεία του deployment. Όταν το πεδίο

configuration υπάρχει και περιέχει μια τιμή, αυτό σημαίνει ότι έχει οριστεί ένας εξωτερικός ενορχηστρωτής. Η τιμή του πεδίου configuration περιέχει το URL του εξωτερικού ενορχηστρωτή όπου θα πρέπει να σταλούν τα deployments.

```
# Process based on association configuration
if association_data.get('configuration'):
    # This is for external service orchestrator
    print(f"External orchestrator: {association_data['configuration']}")
    print("-" * 80)  # separator line
    # Make request to external orchestrator
    result = self.send_to_external_orchestrator(
        association_data['configuration'],
        combined_yaml
    )
    if result:
        print("External deployment request successful")
    else:
        print("External deployment request failed")
```

Εικόνα 23: Σημείο κλήσης μεθόδου send_to_external_orchestrator

Στην περίπτωση που εντοπίσουμε πεδίο configuration, κατευθύνουμε το deployment προς τον εξωτερικό ενορχηστρωτή, έναν διαφορετικό Service Orchestrator. Καλούμε τη μέθοδο ***send_to_external_orchestrator()*** περνώντας το URL από το πεδίο configuration μαζί με την κατάλληλη YAML περιγραφή που περιλαμβάνει το σύνολο των περιγραφών που σχετίζονται με τα microservices που έχουν ανατεθεί στο Association που διαχειρίζεται ο εξωτερικός ενορχηστρωτής.

```python
def send_to_external_orchestrator(self, configuration_url, combined_yaml):
    """Send deployment to external orchestrator"""
    try:
        # Construct the API endpoint
        api_endpoint = f"{configuration_url}/api/v1/orchestrator/deployments"

        # Prepare the request body
        payload = {
            "deployment_description": combined_yaml
        }

        # Make the POST request
        response = requests.post(
            api_endpoint,
            json=payload,
            headers={'Content-Type': 'application/json'}
        )

        # Check if the request was successful
        if response.status_code == 200 or response.status_code == 201:
            print(f"Successfully sent deployment to external orchestrator: {api_endpoint}")
            print(f"Response: {response.json()}")
            return response.json()
        else:
            print(f"Failed to send deployment to external orchestrator: {response.status_code}")
            print(f"Error: {response.text}")
            return None

    except requests.exceptions.RequestException as e:
        print(f"Error making request to external orchestrator: {e}")
        return None
```

Εικόνα 24: Μέθοδος send_to_external_orchestrator

Αυτή η μέθοδος αναλαμβάνει να μετατρέψει το αρχείο YAML σε κατάλληλη JSON περιγραφή και το αποστείλει μέσω HTTP POST αιτήματος στον καθορισμένο εξωτερικό ενορχηστρωτή. Ο εξωτερικός ενορχηστρωτής θα αναλάβει στη συνέχεια την εκτέλεση και διαχείριση των αναπτύξεων στο δικό του περιβάλλον.

Στην αντίθετη περίπτωση, όταν το πεδίο *configuration* δεν υπάρχει ή είναι κενό, αυτό υποδηλώνει ότι τα επόμενα βήματα πρέπει να εκτελεστούν τοπικά από τον τρέχοντα ενορχηστρωτή.

```python
else:
    # This is for current service orchestrator
    print("Current service orchestrator - deploy locally")
    print("-" * 80)  # separator line
    # Deploy using serrano deployment
    deployment_uuid = self.create_local_deployment(combined_yaml, association_uuid)
    print(f"Created local deployment with UUID: {deployment_uuid}")
```

Εικόνα 25: Σημείο κλήσης μεθόδου create_local_deployment

Σε αυτή την περίπτωση, χρησιμοποιούμε τη μέθοδο ***create_local_deployment()*** για να δημιουργήσουμε ένα νέο αντικείμενο Deployment στον ίδιο τον Service Orchestrator που διαχειρίζεται και το αρχικό αίτημα ενορχήστρωσης και εκτέλεσης (Association Deployment object).

```python
def create_local_deployment(self, deployment_description, association_uuid):
    deployment_uuid = str(uuid.uuid4())

    # Create the deployment parameters similar to create_deployment
    deployment_params = {
        "deployment_uuid": deployment_uuid,
        "deployment_description": deployment_description.replace("\\r", ""),
        "assignments": [],
        "assignments_status": [],
        "logs": [{"timestamp": int(time.time()), "event": "Deployment description received from association deployment."}],
        "status": status.Deployment.SUBMITTED,
        "updated_by": "Orchestration.API",
        "created_at": int(time.time()),
        "updated_at": int(time.time()),
        "kind": "Deployment"
    }

    logger.debug("Create Deployment API object for deployment '%s' from association '%s'" % (deployment_uuid, association_uuid))

    # Put it in etcd
    etcdClient.put("/service/orchestrator/deployments/deployment/%s" % deployment_uuid,
                   json.dumps(deployment_params))

    return deployment_uuid
```

Εικόνα 26: Μέθοδος create_local_deployment

Η μέθοδος αυτή παίρνει την YAML περιγραφή από το προηγούμενο βήμα και την μετατρέπει σε ένα Deployment αντικείμενο για τον Orchestrator API μαζί με όλα τα απαραίτητα μεταδεδομένα, συμπεριλαμβανομένου ενός μοναδικού αναγνωριστικού, των καταγραφικών εγγραφών, και των χρονικών σημάνσεων. Το νέο αντικείμενο Deployment αποθηκεύεται στην Datastore και θα εξυπηρετηθεί από τα λειτουργικά συστήματα του ενορχηστρωτή με βάση τις λειτουργίες που ήδη υποστηρίζει..

Κεφάλαιο 5: Επίδειξη Λειτουργίας

Προκειμένου να δημιουργήσουμε ένα Proof of Concept για την επίδειξη της λειτουργικότητας του προτεινόμενου συστήματος, θα υλοποιήσουμε ένα πειραματικό περιβάλλον που περιλαμβάνει δύο Service Orchestrator σε μια κατανεμημένη διαμόρφωση. Κάθε στιγμιότυπο του Service Orchestrator θα διαχειρίζεται δύο ή περισσότερα ανεξάρτητα συστήματα Kubernetes, δημιουργώντας έτσι ένα σενάριο που προσομοιώνει πραγματικές συνθήκες ανάπτυξης σε πολλαπλά συστήματα με υψηλή πολυπλοκότητα. Αυτή η αρχιτεκτονική επιλογή στοχεύει στην επίδειξη των βασικών δυνατοτήτων του συστήματος σε περιβάλλον που προσεγγίζει τις απαιτήσεις παραγωγής.

5.1 Υποδομή επίδειξης λειτουργίας

Για την υλοποίηση της απαραίτητης υποδομής για την επίδειξη του επιλεγμένου σεναρίου λειτουργίας χρησιμοποιήθηκε το λογισμικό **kind (Kubernetes in Docker)** [25]. Το kind αποτελεί ένα σημαντικό εργαλείο ανάπτυξης που επιτρέπει τη δημιουργία και διαχείριση πλήρως λειτουργικών υποδομών Kubernetes σε τοπικό περιβάλλον μέσω της αξιοποίησης των Docker containers ως εικονικών κόμβων (virtual worker nodes). Η προσέγγιση αυτή παρέχει μια πρακτική εναλλακτική λύση στην παραδοσιακή μεθοδολογία που απαιτεί τη χρήση πολλαπλών φυσικών ή εικονικών μηχανημάτων για την υλοποίηση ενός cluster. Παρότι το kind αναπτύχθηκε αρχικά ως εργαλείο δοκιμών για το ίδιο το Kubernetes, έχει εξελιχθεί σε ένα ευρέως αποδεκτό εργαλείο για ανάπτυξη σε τοπικό επίπεδο και συνεχής ενοποίηση (continuous integration) [26]. Η καινοτομία του kind έγκειται στην ικανότητά του να προσφέρει μια πλήρη εμπειρία με σημαντικά μειωμένες απαιτήσεις σε υπολογιστικούς πόρους.

Η αρχιτεκτονική του kind βασίζεται σε μια προηγμένη υλοποίηση containerization που ενσωματώνει όλα τα κρίσιμα στοιχεία ενός Kubernetes κόμβου εντός ειδικά διαμορφωμένων Docker images. Κάθε container περιλαμβάνει τα απαραίτητα συστατικά όπως το kubelet, το container runtime και άλλα βασικά λειτουργικά τμήματα, ενώ χρησιμοποιεί το kubeadm για την αυτοματοποιημένη εκκίνηση και διαμόρφωση κάθε κόμβου [25]. Η μεθοδολογία αυτή επιτρέπει τη δημιουργία πολύπλοκων σεναρίων πολλαπλών κόμβων που προσομοιώνουν με ακρίβεια τη συμπεριφορά πραγματικών παραγωγικών περιβαλλόντων (production environments). Το εργαλείο υποστηρίζει προηγμένες διαμορφώσεις για δημιουργία υποδομών με πολλαπλούς κόμβους και διαθέτει ενσωματωμένες δυνατότητες διαχείρισης δικτύου μεταξύ των κόμβων [27], προσφέροντας έτσι ένα ολοκληρωμένο περιβάλλον δοκιμών και ανάπτυξης.

Στο πλαίσιο της παρούσας διπλωματικής εργασίας, η επιλογή του kind ως βασική πλατφόρμα υλοποίησης της απαραίτητης υποδομής για την επίδειξη του προτεινόμενου συστήματος στηρίζεται σε ισχυρά τεχνικά επιχειρήματα. Το kind παρέχει ένα ιδανικό περιβάλλον για την ανάπτυξη και δοκιμή Kubernetes-native εφαρμογών, επιτρέποντας την εκτέλεση

Deployments, τον εντοπισμό και διόρθωση σφαλμάτων, καθώς και την επικύρωση αλλαγών πριν την ενσωμάτωσή τους σε παραγωγικά περιβάλλοντα [28]. Η δυνατότητα ταχείας δημιουργίας και κατάργησης επιμέρους εικονικών υποδομών (clusters) διευκολύνει την εκτέλεση επαναλαμβανόμενων πειραμάτων και τη δοκιμή εναλλακτικών διαμορφώσεων με ελάχιστο λειτουργικό κόστος. Ιδιαίτερα σημαντικό είναι το γεγονός ότι οι υποδομές που δημιουργούνται μέσω του kind επιτυγχάνουν πλήρη συμμόρφωση με τους επίσημους Kubernetes ελέγχους, διασφαλίζοντας την αξιοπιστία και εγκυρότητα των αποτελεσμάτων [27]. Επιπλέον, η απλότητα εγκατάστασης και λειτουργίας του εργαλείου εξασφαλίζει υψηλό βαθμό αναπαραγωγής των πειραματικών αποτελεσμάτων, επιτρέποντας σε άλλους ερευνητές να επαληθεύσουν και να επεκτείνουν τα ευρήματα χωρίς σημαντικές απαιτήσεις σε υποδομές.

Για την υλοποίηση της πειραματικής διαμόρφωσης, έγινε δημιουργία τριών Kubernetes υποδομών (clusters) μέσω του kind, τα οποία θα διαχειρίζονται δύο διαφορετικοί Service Orchestrators ακολουθώντας μια συνεργατική και κατανεμημένη διαμόρφωση. Η συγκεκριμένη επιλογή αποσκοπεί στην αποτύπωση ρεαλιστικών σεναρίων ανάθεσης και εκτέλεσης εφαρμογών σε πολλαπλές ενοποιημένες υποδομές (federated multi-cluster deployment) που αντικατοπτρίζουν την πολυπλοκότητα σύγχρονων εταιρικών περιβαλλόντων (enterprise environments) όπου πολλαπλά στιγμιότυπα (instances) του συστήματος διαχείρισης χρειάζεται να συνεργάζονται για την ενορχήστρωση υπηρεσιών με βάση μια κατανεμημένη αρχιτεκτονική.

Τα τρία clusters (cluster1, cluster2, cluster3) δημιουργήθηκαν με *single-node* διαμόρφωση, όπου κάθε cluster περιλαμβάνει έναν control-plane κόμβο που διαχειρίζεται όλες τις κρίσιμες λειτουργίες του αντίστοιχου cluster. Η διαμόρφωση αυτή, παρότι ελαχιστοποιημένη σε πόρους, διατηρεί πλήρη συμβατότητα με τις προδιαγραφές Kubernetes και επιτρέπει την εκτέλεση των απαραίτητων ενεργειών για την επίδειξη της συνεργατικής λειτουργικότητας των επιμέρους Service Orchestrators.

Η επιλογή single-node διαμόρφωσης για όλα τα clusters δεν υποβαθμίζει την εγκυρότητα του σεναρίου επίδειξης των μηχανισμών που αναπτύχθηκαν, καθώς το kind διασφαλίζει πλήρη συμβατότητα με το Kubernetes API και επιτρέπει την εκτέλεση όλων των βασικών λειτουργιών του συστήματος. Αυτή η προσέγγιση εξασφαλίζει επίσης βέλτιστη αξιοποίηση των περιορισμένων διαθέσιμων υπολογιστικών πόρων σε τοπικό επίπεδο, επιτρέποντας την παράλληλη λειτουργία πολλαπλών clusters στο ίδιο φυσικό μηχάνημα χωρίς υποβάθμιση της απόδοσης.

Η επιλογή για τη χρήση δύο διαφορετικών Service Orchestrators υπογραμμίζει την κατανεμημένη φύση του υλοποιημένου συστήματος καθώς και την επίδειξη των δυνατοτήτων συνεργασίας μεταξύ πολλαπλών Service Orchestrators για την υποστήριξη συνεργατικής ενορχήστρωσης cloud-native εφαρμογών σε κατανεμημένες υποδομές. Κάθε Service Orchestrator αναλαμβάνει τη διαχείριση συγκεκριμένων clusters, ενώ η συνολική ενορχήστρωση στην ενοποιημένη υποδομή επιτυγχάνεται μέσω των επεκτάσεων που σχεδιάστηκαν και υλοποιήθηκαν στα πλαίσια της παρούσας διπλωματικής εργασίας.

Κάθε cluster δημιουργήθηκε με την εντολή ***kind create cluster --name [cluster-name]***, χρησιμοποιώντας τις προεπιλεγμένες διαμορφώσεις του kind που εγγυώνται συμβατότητα με το Kubernetes.. Η απλότητα αυτής της μεθοδολογίας επιτρέπει την ταχεία αναπαραγωγή της πειραματικής διαμόρφωσης και διευκολύνει την επικύρωση των αποτελεσμάτων από τρίτους ερευνητές.

```
panos@ubuntu-usb:~/Thesis/kind-poc$ kind create cluster --name cluster1 --wait 300s
Creating cluster "cluster1" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Waiting ≤ 5m0s for control-plane = Ready ⏳
 • Ready after 13s 💚
Set kubectl context to "kind-cluster1"
You can now use your cluster with:

kubectl cluster-info --context kind-cluster1

Thanks for using kind! 😊
panos@ubuntu-usb:~/Thesis/kind-poc$ kind create cluster --name cluster2 --wait 300s
Creating cluster "cluster2" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Waiting ≤ 5m0s for control-plane = Ready ⏳
 • Ready after 17s 💚
Set kubectl context to "kind-cluster2"
You can now use your cluster with:

kubectl cluster-info --context kind-cluster2

Have a nice day! 👋
panos@ubuntu-usb:~/Thesis/kind-poc$ kind create cluster --name cluster3 --wait 300s
Creating cluster "cluster3" ...
 ✓ Ensuring node image (kindest/node:v1.27.3) 🖼
 ✓ Preparing nodes 📦
 ✓ Writing configuration 📜
 ✓ Starting control-plane 🕹️
 ✓ Installing CNI 🔌
 ✓ Installing StorageClass 💾
 ✓ Waiting ≤ 5m0s for control-plane = Ready ⏳
 • Ready after 16s 💚
Set kubectl context to "kind-cluster3"
You can now use your cluster with:

kubectl cluster-info --context kind-cluster3
```

Εικόνα 27: Δημιουργία των clusters

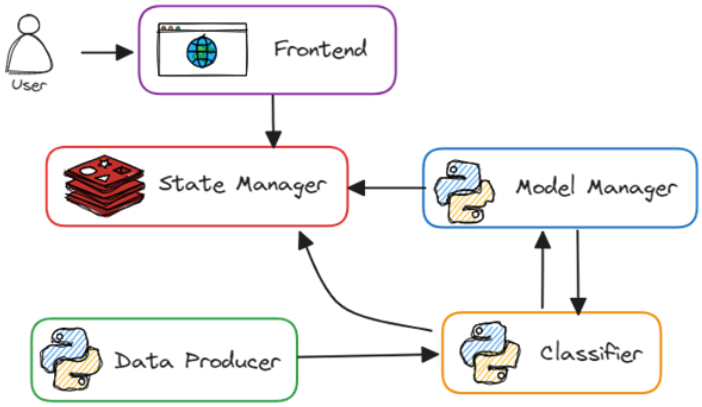
Η επιτυχής δημιουργία και λειτουργία των τριών clusters επιβεβαιώθηκε μέσω των κατάλληλων ***kubectl*** εντολών, διασφαλίζοντας ότι κάθε cluster βρίσκεται σε κατάσταση *Ready* και είναι έτοιμο για την φιλοξενία εφαρμογών. Αυτή η διαμόρφωση παρέχει ένα αξιόπιστο και λειτουργικό περιβάλλον για την επίδειξη των δυνατοτήτων των προτεινόμενων επεκτάσεων του Service Orchestrator για την συνεργατική ενορχήστρωση cloud-native εφαρμογών σε κατανεμημένα περιβάλλοντα υποδομών με πολλαπλά clusters..

```
panos@ubuntu-usb:~/Thesis/kind-poc$ kind get clusters
cluster1
cluster2
cluster3
panos@ubuntu-usb:~/Thesis/kind-poc$ kubectl get nodes
NAME                     STATUS   ROLES           AGE   VERSION
cluster3-control-plane   Ready    control-plane   94s   v1.27.3
panos@ubuntu-usb:~/Thesis/kind-poc$ kubectl get nodes --context kind-cluster1
NAME                     STATUS   ROLES           AGE    VERSION
cluster1-control-plane   Ready    control-plane   5m2s   v1.27.3
panos@ubuntu-usb:~/Thesis/kind-poc$ kubectl get nodes --context kind-cluster2
NAME                     STATUS   ROLES           AGE     VERSION
cluster2-control-plane   Ready    control-plane   3m28s   v1.27.3
panos@ubuntu-usb:~/Thesis/kind-poc$ kubectl get nodes --context kind-cluster3
NAME                     STATUS   ROLES           AGE     VERSION
cluster3-control-plane   Ready    control-plane   2m34s   v1.27.3
panos@ubuntu-usb:~/Thesis/kind-poc$ kubectl config get-contexts | grep kind
          kind-cluster1   kind-cluster1   kind-cluster1
          kind-cluster2   kind-cluster2   kind-cluster2
*         kind-cluster3   kind-cluster3   kind-cluster3
panos@ubuntu-usb:~/Thesis/kind-poc$
```

Εικόνα 28: Χρήση kubectl για την επιβεβαίωση της δημιουργίας των clusters

5.2 Πιλοτική εφαρμογή

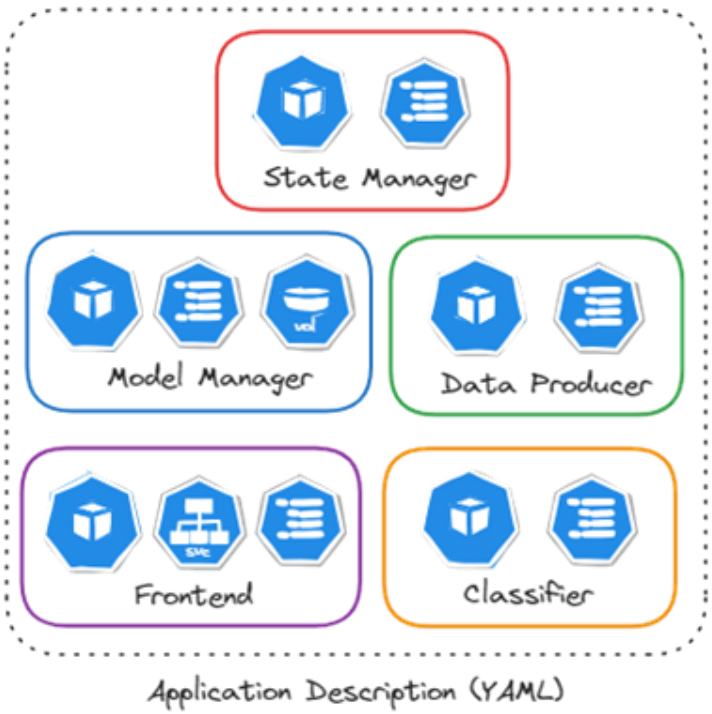
Προκειμένου να αναδείξουμε την λειτουργικότητα του συστήματος θα βασιστούμε σε μια εγγενή εφαρμογή υπολογιστικού νέφους που μας παρείχε το εργαστήριο HSCNL [35]. Η εφαρμογή αυτή αποτελείται από πέντε microservices όπως φαίνεται και στην εικόνα 29. Πρόκειται για μια εφαρμογή που συλλέγει δεδομένα από IoT κόμβους, εκτελεί μοντέλα μηχανικής μάθησης (machine learning), και αποθηκεύει τα αποτελέσματα. Επιπρόσθετα, περιλαμβάνει μια υπηρεσία που ενεργοποιεί την επανεκπαίδευση του μοντέλου με βάση τα αποτελέσματα των προβλέψεων. Ένα απλό, διαδικτυακό περιβάλλον χρήστη (UI) προσφέρει στους χρήστες εύκολη πρόσβαση στα αποτελέσματα της εφαρμογής. Κάθε microservice επιτελεί συγκεκριμένο ρόλο στην ροή εργασίας. Αυτά τα microservices αλληλεπιδρούν απρόσκοπτα για να διασφαλίσουν την αποδοτική συλλογή δεδομένων, την επεξεργασία και την αποθήκευση τους καθώς και τη διαχείριση του μοντέλου.



Εικόνα 29: Εφαρμογή επίδειξης

Το σύνολο δεδομένων της εφαρμογής περιλαμβάνει τιμές που σχετίζονται με την ποιότητα των μήλων, με το μοντέλο να πραγματοποιεί την ταξινόμησή τους. Το σύνολο δεδομένων και το μοντέλο προέρχονται από το "Kaggle's Apple Quality Dataset" [34]. Ο σχεδιασμός αυτός προσφέρει ένα απλό αλλά λειτουργικό παράδειγμα, αναδεικνύοντας βασικές δυνατότητες του μηχανισμού ενορχήστρωσης που αναπτύχθηκε στα πλαίσια της παρούσας εργασίας.

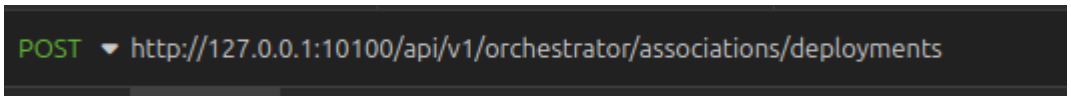
Για τη διαχείριση της ανάπτυξης και λειτουργίας κάθε microservice, χρησιμοποιούνται κατάλληλα K8s/K3s YAML αρχεία. Αυτά τα αρχεία ορίζουν κρίσιμα αντικείμενα (Kubernetes objects), όπως τα Deployment, ConfigMap, Service και PersistentVolume. Η εικόνα 30 παρέχει αναλυτική απεικόνιση των συγκεκριμένων αντικειμένων K8s/K3s που σχετίζονται με κάθε microservice.



Εικόνα 30: Απεικόνιση αντικειμένων του κάθε microservice

5.3 Αποτελέσματα

Ξεκινάμε την διαδικασία εκτελώντας ένα POST αίτημα στο URL http://127.0.0.1:10100/api/v1/orchestrator/associations/deployments με την χρήση του εργαλείου Insomnia [36].



Το περιεχόμενο του αιτήματος είναι σε JSON μορφή και παρουσιάζεται στις ακόλουθες εικόνες:


```
{
  "name": "toy-application",
  "deployments": [
    {
      "name": "frontend-deployment",
      "user_token": "",
      "deployment_description": "kind: Deployment\napiVersion: apps/v1\nmetadata:\n  name: frontend\n  namespace: default\n  labels:\n
group_id: s1\n    app: frontend\nspec:\n  selector:\n    matchLabels:\n      app: frontend\n  replicas: 1\n  template:\n    metadata:\n
    labels:\n        app: frontend\n    spec:\n      imagePullSecrets:\n      - name: empyrean-registry-key\n      hostNetwork: true\n
    containers:\n        - name: frontend\n          image: ictserrano/serrano:frontend_v2\n          imagePullPolicy: Always\n
command: [\"/home/hscnl/myenv/bin/python3\", \"/home/hscnl/app.py\"]\n          ports:\n            - containerPort: 8088\n
protocol: TCP\n          volumeMounts:\n            - name: frontend-config\n              mountPath: /etc/app\n      volumes:\n
- name: frontend-config\n          configMap:\n            name: frontend-config"
    },
    {
      "name": "frontend-configmap",
      "deployment_description": "apiVersion: v1\nkind: ConfigMap\nmetadata:\n  name: frontend-config\n  namespace: default\n  labels:\n
  group_id: s1\ndata:\n conf.json: |\n   {\n     \"app_port\": 8088,\n     \"redis_ip\": \"147.102.16.114\",\n     \"redis_port\":
30080,\n     \"websocket_service\": \"ws://147.102.16.114:30032\"\n   }"
    },
    {
      "name": "frontend-service",
      "deployment_description": "kind: Service\napiVersion: v1\nmetadata:\n  name: frontend-svc\n  namespace: default\n  labels:\n
app: frontend\n    group_id: s1\nspec:\n  type: NodePort\n  ports:\n    - port: 8088\n      targetPort: 8088\n      protocol: TCP\n
nodePort: 30088\n  selector:\n    app: frontend"
    },
    {
      "name": "data-producer-deployment",
      "deployment_description": "kind: Deployment\napiVersion: apps/v1\nmetadata:\n  name: data-producer\n  namespace: default\n
labels:\n    group_id: dpr\nspec:\n  selector:\n    matchLabels:\n      app: data-producer\n  replicas: 1\n  template:\n    metadata:\n
    labels:\n        app: data-producer\n    spec:\n      imagePullSecrets:\n      - name: empyrean-registry-key\n      containers:\n
  - name: data-producer\n          image: ictserrano/serrano:data-producer_v2\n          command: [\"python3\",
\"/home/hscnl/data_producer.py\"]\n          imagePullPolicy: Always\n          ports:\n          - containerPort: 6380\n            protocol:
TCP\n          volumeMounts:\n          - name: data-producer-config\n            mountPath: /etc/app\n      volumes:\n      - name: data-
producer-config\n          configMap:\n            name: data-producer-config"
    },
    {
      "name": "data-producer-configmap",
      "deployment_description": "apiVersion: v1\nkind: ConfigMap\nmetadata:\n  name: data-producer-config\n  namespace: default\n
labels:\n    group_id: dpr\ndata:\n  conf.json: |\n    {\n      \"total_data_size\": 38,\n      \"data_batch_size\": 10,\n
\"batch_interval\":[15, 25],\n      \"IP\": \"147.102.16.114\",\n      \"PORT\": 30080\n    }"
    },
```


Εικόνα 31: Περιεχόμενο αιτήματος (α)

Εικόνα 32: Περιεχόμενο αιτήματος (β)

Εικόνα 33: Περιεχόμενο αιτήματος (γ)

Το πρώτο σημείο αλληλεπίδρασης με το σύστημα είναι το Orchestrator API με την μέθοδο ***post_associations_deployment*** όπως είδαμε στο κομμάτι της υλοποίησης.

```
INFO:Orchestrator.OrchestratorAPI:Incoming request for new application associations deployment ...
```

Κοιτώντας το αρχεία καταγραφών (log file) βλέπουμε ότι έχει ληφθεί επιτυχώς το αίτημα για το Associations Deployment.
Στην συνέχεια το σύστημα αναθέτει στο Associations Deployment το μοναδικό αναγνωριστικό του, το καταχωρεί στο αρχείο καταγραφών και επαληθεύει τα δεδομένα και την ποιότητα των πληροφοριών.

```
DEBUG:Orchestrator.OrchestratorAPI:Assigned associations deployment uuid '03ca0fb8-f7b8-41ea-8b0d-fbe9b8831d55'
INFO:Orchestrator.OrchestratorAPI:Validated deployments inside AssociationsDeployment
```

```
INFO:     127.0.0.1:36188 - "POST /api/v1/orchestrator/associations/deployments HTTP/1.1" 201 Created
```

Κατόπιν καλείται ο Dispatcher που δημιουργεί το Associations Deployment και το καταχωρεί στην Datastore.

```
INFO:Service.Orchestrator.Dispatcher:Cleaned deployment descriptions
DEBUG:Service.Orchestrator.Dispatcher:Create Associations Deployment API object for deployment '03ca0fb8-f7b8-41ea-8b0d-fbe9b8831d55'
```

Ο Orchestration Manager που παρακολουθεί για αλλαγές στην Datastore κάνοντας εγγραφή με βάση το συγκεκριμένο prefix που αφορά τα Associations Deployments "ξυπνάει" αυτόματα από το σύστημα και εκτελεί την μέθοδο ***__etcd_watch_associations_deployment_callback***

```
INFO:Orchestrator.OrchestrationManager:Orchestration Manager is ready ...
INFO:Orchestrator.Manager.OrchestrationAPIInterface:Associations Deployment event(s)...
INFO:Orchestrator.Manager.OrchestrationAPIInterface:Associations Deployment event for key
'/service/orchestrator/associations/deployments/deployment/03ca0fb8-f7b8-41ea-8b0d-fbe9b8831d55'
```

Το επόμενο βήμα περιλαμβάνει την επικοινωνία του Orchestration Manager με την εξειδικευμένη εξωτερική υπηρεσία του συστήματος, την Decision Engine. Μέσω αυτής επιτυγχάνουμε την λήψη των απαιτούμενων πληροφοριών για την κατανομή του Associations Deployment στα επιμέρους Deployments του υφιστάμενου συστήματος. Στην παρακάτω εικόνα παρουσιάζουμε την απάντηση της Decision Engine σχετικά με την ανάθεση των microservices της πιλοτικής εφαρμογής.

```
Decision Engine Response:
{
  "kind": "AssociationsDeployment",
  "assignments": [
    {
      "association_uuid": "f7144a4a-2b64-47a0-93f1-3b91e7c5f138",
      "deployments": [
        {
          "name": "state-manager-deployment",
          "score": 3.344,
          "group_id": "s3"
        },
        {
          "name": "model-manager-deployment",
          "score": 3.569,
          "group_id": "s2"
        },
        {
          "name": "frontend-deployment",
          "score": 2.933,
          "group_id": "s1"
        }
      ]
    },
    {
      "association_uuid": "c593b878-19ac-4690-b40e-0e812eecbf98",
      "deployments": [
        {
          "name": "classifier-deployment",
          "score": 3.38,
          "group_id": "s4"
        },
        {
          "name": "data-producer-deployment",
          "score": 1.23,
          "group_id": "dpr"
        }
      ]
    }
  ],
  "instructions": {},
```


Εικόνα 34: Η απόφαση της decision engine για ανάθεση microservices σε Associations

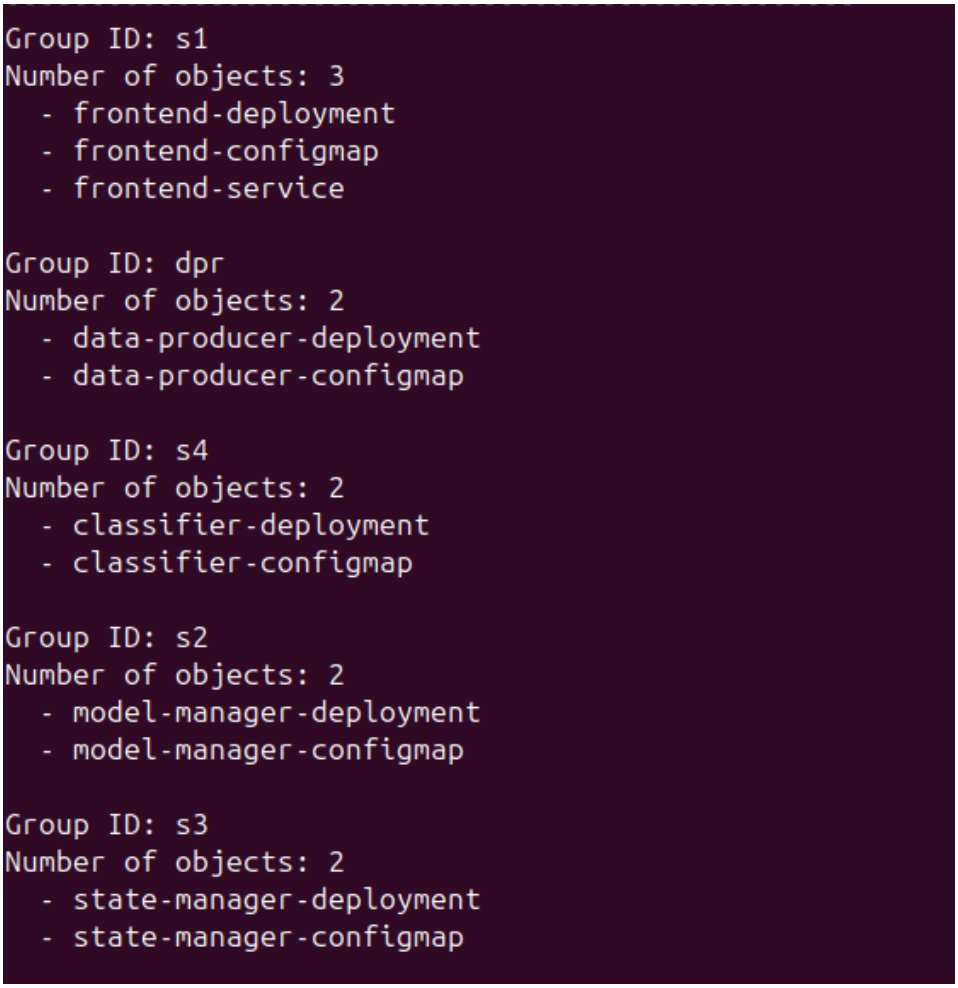
Μπορούμε να διακρίνουμε από την απάντηση της Decision Engine, ότι τα microservices της εφαρμογής έχουν κατανεμηθεί σε δύο Associations, των οποίων τις αναλυτικές πληροφορίες εξάγουμε στην συνέχεια από την Datastore.

```
DEBUG:Orchestrator.OrchestrationManager:About to fetch associations
```

Εικόνα 35: Αποθηκευμένες Associations στην etcd

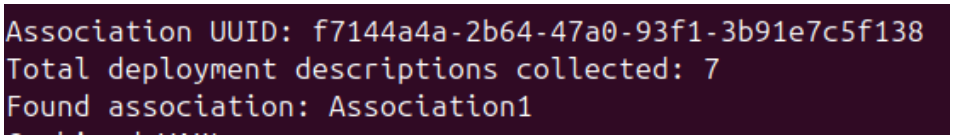
Κάνουμε την παραδοχή ότι το Association που δεν έχει τιμή στο πεδίο “configuration” είναι αυτό που περιέχει τον τοπικό ενορχηστρωτή.

Το επόμενο βήμα περιλαμβάνει την οργάνωση όλων των σχετικών περιγραφών για την ανάθεση και εκτέλεση των microservices στα επιμέρους Associations με κριτήριο το *group_id* τους.



Εικόνα 36: Ομαδοποίηση deployments με κριτήριο το group_id

Στη συνέχεια, πραγματοποιείται η επεξεργασία κάθε Assignment που περιλαμβάνεται στην απάντηση της Decision Engine. Για κάθε Assignment, συλλέγονται οι περιγραφές των deployments που αντιστοιχούν στα group_ids, όπως αυτά ορίζονται στο πεδίο “deployments” του εκάστοτε αντικειμένου. Μετά τη συλλογή των απαραίτητων δεδομένων, δημιουργείται ένα ενιαίο αρχείο σε μορφή YAML, το οποίο περιλαμβάνει το σύνολο των σχετικών deployment περιγραφών. Τέλος, απαιτείται η επιλογή του κατάλληλου σημείου εκτέλεσης για τα deployments, απόφαση η οποία καθορίζεται με βάση τις πληροφορίες που προκύπτουν από την απάντηση της Decision Engine για κάθε αντίστοιχο Association.



Βλέπουμε ότι το μοναδικό αναγνωριστικό για το Association που περιλαμβάνεται στο πρώτο Assignment της απάντησης αντιστοιχεί στο Association1. Για αυτό το Association θα δημιουργηθεί ένα ενοποιημένο YAML αρχείο που θα περιλαμβάνει επτά περιγραφές deployments.

```
Combined YAML:
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    group_id: s3
  name: state-manager
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: state-manager
  template:
    metadata:
      labels:
        app: state-manager
    spec:
      containers:
      - command:
        - python3
        - /home/hscnl/state_manager.py
        image: ictserrano/serrano:state-manager_v2
        imagePullPolicy: Always
        name: state-manager
        ports:
        - containerPort: 6380
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/app
          name: state-manager-config
      imagePullSecrets:
      - name: empyrean-registry-key
      volumes:
      - configMap:
          name: state-manager-config
        name: state-manager-config
---
```

Εικόνα 37: Ενοποιημένο YAML με τις περιγραφές των deployments (α)

```
apiVersion: v1
data:
  conf.json: "{\n  \"IP\": \"147.102.16.114\",\n  \"PORT\": 30080\n}"
kind: ConfigMap
metadata:
  labels:
    group_id: s3
  name: state-manager-config
  namespace: default
```

Εικόνα 38: Ενοποιημένο YAML με τις περιγραφές των deployments (β)

```yaml
---
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    group_id: s2
  name: model-manager
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: model-manager
  template:
    metadata:
      labels:
        app: model-manager
    spec:
      containers:
      - command:
        - python3
        - /home/hscnl/model_manager.py
        image: ictserrano/serrano:model-manager_v2
        imagePullPolicy: Always
        name: model-manager
        ports:
        - containerPort: 6380
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/app
          name: model-manager-config
      imagePullSecrets:
      - name: empyrean-registry-key
      volumes:
      - configMap:
          name: model-manager-config
        name: model-manager-config
---
```

Εικόνα 39: Ενοποιημένο YAML με τις περιγραφές των deployments (γ)

```yaml
apiVersion: v1
data:
  conf.json: "{\n  \"IP\": \"147.102.16.114\",\n  \"PORT\": 30080\n}"
kind: ConfigMap
metadata:
  labels:
    group_id: s2
  name: model-manager-config
  namespace: default
```

Εικόνα 40: Ενοποιημένο YAML με τις περιγραφές των deployments (δ)

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: frontend
    group_id: s1
  name: frontend
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: frontend
  template:
    metadata:
      labels:
        app: frontend
    spec:
      containers:
      - command:
        - /home/hscnl/myenv/bin/python3
        - /home/hscnl/app.py
        image: ictserrano/serrano:frontend_v2
        imagePullPolicy: Always
        name: frontend
        ports:
        - containerPort: 8088
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/app
          name: frontend-config
      hostNetwork: true
      imagePullSecrets:
      - name: empyrean-registry-key
      volumes:
      - configMap:
          name: frontend-config
        name: frontend-config
```

Εικόνα 41: Ενοποιημένο YAML με τις περιγραφές των deployments (ε)

```yaml
apiVersion: v1
data:
  conf.json: "{\n  \"app_port\": 8088,\n  \"redis_ip\": \"147.102.16.114\",\n  \"\
    redis_port\": 30080,\n  \"websocket_service\": \"ws://147.102.16.114:30032\"\n\
    }"
kind: ConfigMap
metadata:
  labels:
    group_id: s1
  name: frontend-config
  namespace: default
```

Εικόνα 42: Ενοποιημένο YAML με τις περιγραφές των deployments (στ)

```
apiVersion: v1
kind: Service
metadata:
  labels:
    app: frontend
    group_id: s1
  name: frontend-svc
  namespace: default
spec:
  ports:
  - nodePort: 30088
    port: 8088
    protocol: TCP
    targetPort: 8088
  selector:
    app: frontend
  type: NodePort
```

Εικόνα 43: Ενοποιημένο YAML με τις περιγραφές των deployments (ζ)

```
Current service orchestrator - deploy locally
```

Βλέπουμε ότι το σύστημα αναγνώρισε ότι το Deployment θα εκτελεστεί στον τοπικό ενορχηστρωτή καθώς το Association1 δεν έχει τιμή στο πεδίο "configuration". Οπότε προχωράει την δημιουργία του Deployment και την αποθήκευση του στην Datastore.

```
DEBUG:Orchestrator.OrchestrationManager:Create Deployment API object
    for deployment 'c1cae341-47f4-4133-ad19-c9a11bbe7384' from association 'f7144a4a-2b64-47a0-93f1-3b91e7c5f138'
--------------------------------------------------------------------------------
Created local deployment with UUID: c1cae341-47f4-4133-ad19-c9a11bbe7384
```

Οπότε αφού το Deployment καταχωρηθεί στην Datastore, αναμένουμε να ειδοποιηθεί ο αντίστοιχος Orchestration Manager (καθώς θα παρατηρηθεί αλλαγή στο prefix που αφορά τα Deployment) και να εκτελεστεί η τυπική διαδικασία.

```
INFO:Orchestrator.Manager.OrchestrationAPIInterface:Deployment event(s) ...
INFO:Orchestrator.Manager.OrchestrationAPIInterface:Deployment event for key
'/service/orchestrator/deployments/deployment/c1cae341-47f4-4133-ad19-c9a11bbe7384'
INFO:Orchestrator.OrchestrationManager:Handle deployment request ...
DEBUG:Orchestrator.OrchestrationManager:{"deployment_uuid": "c1cae341-47f4-4133-ad19-c9a11bbe7384", "deployment_description": "apiVe
```

Πράγματι, παρατηρώντας το αρχείο καταγραφών βλέπουμε ότι έχει ακολουθηθεί η διαδικασία για ένα τυπικό Deployment.

Για το επόμενο Assignment που περιλαμβάνεται στην απάντηση της Decision Engine ακολουθείται η ίδια διαδικασία.

```
Association UUID: c593b878-19ac-4690-b40e-0e812eecbf98
Total deployment descriptions collected: 4
Found association: Association2
```

Το YAML που θα δημιουργηθεί θα περιέχει 4 περιγραφές για deployments.

```
Combined YAML:
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    group_id: s4
  name: classifier
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: classifier
  template:
    metadata:
      labels:
        app: classifier
    spec:
      containers:
      - command:
        - /home/hscnl/myenv/bin/python
        - /home/hscnl/classifier_sklearn.py
        image: ictserrano/serrano:classifier_v2
        imagePullPolicy: Always
        name: classifier
        ports:
        - containerPort: 6380
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/app
          name: classifier-config
      imagePullSecrets:
      - name: empyrean-registry-key
      volumes:
      - configMap:
          name: classifier-config
        name: classifier-config
---
apiVersion: v1
data:
  conf.json: "{\n  \"IP\": \"147.102.16.114\",\n  \"PORT\": 30080\n}"
kind: ConfigMap
metadata:
  labels:
    group_id: s4
  name: classifier-config
  namespace: default
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    group_id: dpr
  name: data-producer
  namespace: default
spec:
  replicas: 1
  selector:
    matchLabels:
      app: data-producer
  template:
    metadata:
      labels:
        app: data-producer
    spec:
      containers:
      - command:
        - python3
        - /home/hscnl/data_producer.py
        image: ictserrano/serrano:data-producer_v2
        imagePullPolicy: Always
        name: data-producer
        ports:
        - containerPort: 6380
          protocol: TCP
        volumeMounts:
        - mountPath: /etc/app
          name: data-producer-config
      imagePullSecrets:
      - name: empyrean-registry-key
      volumes:
      - configMap:
          name: data-producer-config
        name: data-producer-config
```

```yaml
apiVersion: v1
data:
  conf.json: "{\n  \"total_data_size\": 38,\n  \"data_batch_size\": 10,\n  \"batch_interval\"\
    :[15, 25],\n  \"IP\": \"147.102.16.114\",\n  \"PORT\": 30080\n}"
kind: ConfigMap
metadata:
  labels:
    group_id: dpr
  name: data-producer-config
  namespace: default
```

Για αυτό το Association το σύστημα θα αναγνωρίσει ότι θα εκτελεστεί σε εξωτερικό ενορχηστρωτή καθώς το πεδίο “configuration” του έχει τιμή.

```
External orchestrator: http://127.0.0.1:10200
```

Όντως παρατηρούμε ότι το Deployment για το δεύτερο Assignment θα εκτελεστεί σε ενορχηστρωτή που ακούει σε διαφορετική πόρτα από αυτή που ακούει ο τοπικός ενορχηστρωτής.

Κεφάλαιο 6: Συμπέρασμα & Μελλοντική Εργασία

6.1 Συμπεράσματα

Η παρούσα διπλωματική εργασία επικεντρώθηκε στη μελέτη, σχεδίαση και υλοποίηση προηγμένων μηχανισμών συνεργατικής διαχείρισης και ενορχήστρωσης cloud-native εφαρμογών σε κατανεμημένες υποδομές edge-cloud. Μέσω της επέκτασης του πλαισίου ενορχήστρωσης που αναπτύχθηκε στα πλαίσια του ευρωπαϊκού έργου SERRANO, επιτεύχθηκε η δημιουργία ενός ευέλικτου και επεκτάσιμου συστήματος που αξιοποιεί γνωσιακές τεχνικές για τη λήψη βέλτιστων αποφάσεων κατανομής πόρων.

Η κύρια συνεισφορά της εργασίας συνίσταται στην ανάπτυξη μιας κατανεμημένης αρχιτεκτονικής ενορχήστρωσης που υποστηρίζει τη δημιουργία και διαχείριση δυναμικών συνεργατικών σχημάτων, των Associations. Τα σχήματα αυτά μπορούν να εκτείνονται σε πολλαπλούς παρόχους υπηρεσιών και γεωγραφικές περιοχές, διαμορφώνοντας ένα ενιαίο οικοσύστημα που υλοποιεί το μοντέλο του συνεχούς φάσματος IoT-edge-cloud. Η προσέγγιση που υιοθετήθηκε διαφοροποιείται από τις υπάρχουσες λύσεις μέσω της κατανεμημένης οργάνωσης που επιτρέπει στους επιμέρους ενορχηστρωτές υπηρεσιών να λειτουργούν ως αυτόνομες οντότητες ενώ ταυτόχρονα μπορούν να συνεργάζονται για την αποδοτική αντιστοίχιση των εφαρμογών στους διαθέσιμους πόρους σε όλο το εύρος των IoT-edge-cloud υποδομών.

Η επέκταση του Service Orchestrator με νέα εσωτερικά αντικείμενα API (API objects) αποτελεί σημαντική επέκταση που ενισχύει τις δυνατότητες του αρχικού συστήματος. Τα νέα αντικείμενα παρέχουν υψηλού επιπέδου αφαίρεση και απλοποιούν τις διαδικασίες ενορχήστρωσης και εκτέλεσης εφαρμογών σε ετερογενής υποδομές που ενοποιούνται μέσω της υιοθέτησης του μοντέλου οργάνωσης των Associations, διατηρώντας την απαραίτητη ευελιξία που χαρακτηρίζει τα σύνθετα κατανεμημένα περιβάλλοντα. Η δημιουργία μιας ολοκληρωμένης διεπαφής προγραμματισμού εφαρμογών REST που συμμορφώνεται με τις βέλτιστες πρακτικές παρέχει τυποποιημένη διεπαφή για την αλληλεπίδραση με το σύστημα, εξασφαλίζοντας επεκτασιμότητα και διαλειτουργικότητα.

Η πειραματική επίδειξη της λειτουργικότητας μέσω της υλοποίησης ενός παραδείγματος επιβεβαίωσε στην πράξη την βιωσιμότητα της προτεινόμενης λύσης. Το σύστημα επέδειξε ικανότητα διαχείρισης μιας εγγενούς εφαρμογής υπολογιστικού νέφους σε κατανεμημένο περιβάλλον. Η αρθρωτή σχεδίαση και η υιοθέτηση δηλωτικού προτύπου ελέγχου επιτρέπουν την εύκολη επέκταση και προσαρμογή σε διαφορετικές τεχνικές απαιτήσεις.

Η συνεισφορά της εργασίας στην επιστημονική κοινότητα εκδηλώνεται μέσω της προτυποποίησης νέων μοντέλων δεδομένων για συνεργατική ενορχήστρωση, της επέκτασης υπαρχόντων πλαισίων με δυνατότητες διαχείρισης υπερ-κατανεμημένων υποδομών, και της ανάπτυξης μεθοδολογιών για την αυτόματη κατανομή εφαρμογών σε ετερογενείς υποδομές. Η

εμπειρική επικύρωση της προσέγγισης μέσω πρακτικής υλοποίησης παρέχει σημαντικές πληροφορίες για τη βιωσιμότητα τέτοιων συστημάτων.

6.2 Περιορισμοί και προκλήσεις

Παρά τα σημαντικά αποτελέσματα που επιτεύχθηκαν, η εργασία παρουσιάζει ορισμένους περιορισμούς που χρήζουν αναγνώρισης. Η πειραματική αξιολόγηση πραγματοποιήθηκε σε περιορισμένο περιβάλλον με σχετικά απλή εφαρμογή, γεγονός που καθιστά απαραίτητη την εκτενέστερη αξιολόγηση σε μεγαλύτερη κλίμακα για την πλήρη επικύρωση της αποδοτικότητας και κλιμακωσιμότητας του συστήματος. Επιπλέον, η εξάρτηση από την υπάρχουσα Decision Engine του πλαισίου SERRANO υποδηλώνει την ανάγκη για βελτιστοποίηση των αλγορίθμων λήψης αποφάσεων ειδικά για το περιβάλλον των Associations.

Η διαχείριση ασφάλειας και ταυτότητας σε κατανεμημένα περιβάλλοντα πολλαπλών παρόχων αποτελεί σημαντική πρόκληση που απαιτεί την ανάπτυξη εξελιγμένων λύσεων. Παράλληλα, η απουσία ολοκληρωμένων μηχανισμών παρακολούθησης σε επίπεδο Associations περιορίζει την αποτελεσματική διαχείριση σε παραγωγικά περιβάλλοντα. Η διατήρηση συνέπειας κατάστασης μεταξύ κατανεμημένων ενορχηστρωτών και η διαχείριση αποτυχιών παρουσιάζουν επίσης τεχνικές προκλήσεις που χρήζουν περαιτέρω διερεύνησης.

6.3 Κατευθύνσεις μελλοντικής εργασίας

Η συνέχιση της ερευνητικής εργασίας μπορεί να επικεντρωθεί σε διάφορες κατευθύνσεις που θα ενισχύσουν τις δυνατότητες του συστήματος και θα επεκτείνουν τις εφαρμογές του. Η ανάπτυξη προηγμένων μηχανισμών ασφάλειας αποτελεί άμεση προτεραιότητα, περιλαμβάνοντας κρυπτογράφηση επικοινωνίας, ισχυρότερους μηχανισμούς πιστοποίησης και ολοκληρωμένους μηχανισμούς ελέγχου. Παράλληλα, η βελτιστοποίηση των αλγορίθμων κατανομής με την ενσωμάτωση παραγόντων όπως γεωγραφικοί περιορισμοί, κόστος λειτουργίας και δυναμικές συνθήκες φόρτου θα βελτιώσει σημαντικά την αποδοτικότητα του συστήματος.

Η ενσωμάτωση ολοκληρωμένων μηχανισμών παρατηρησιμότητας και επιτήρησης αποτελεί κρίσιμη κατεύθυνση που θα επιτρέψει την αποτελεσματική παρακολούθηση και διαχείριση σε πραγματικό χρόνο. Η υποστήριξη ετερογενών τεχνολογιών ενορχήστρωσης θα επεκτείνει τις δυνατότητες του συστήματος, ενώ η ανάπτυξη προηγμένων δυνατοτήτων διαχείρισης κύκλου ζωής εφαρμογών θα ενισχύσει την πρακτική χρησιμότητά του.

Σε μακροπρόθεσμο ορίζοντα, η ενσωμάτωση τεχνικών μηχανικής μάθησης για προβλεπτική ανάλυση και αυτόματη βελτιστοποίηση παρουσιάζει σημαντικές δυνατότητες. Η διερεύνηση τεχνικών ομοσπονδιακής μάθησης (federated learning) για κατανεμημένη εκπαίδευση μοντέλων μεταξύ Associations, διατηρώντας την ιδιωτικότητα δεδομένων, αποτελεί προηγμένη ερευνητική κατεύθυνση. Επιπλέον, η αξιοποίηση τεχνολογιών blockchain για την

εισαγωγή αποκεντρωμένων μοντέλων εμπιστοσύνης συνιστά καινοτόμο κατεύθυνση διεύρυνσης της υπάρχουσας λειτουργικότητας

6.4 Συνολική αξιολόγηση

Η παρούσα διπλωματική εργασία αποτελεί σημαντική συνεισφορά στην επιστημονική περιοχή της κατανεμημένης ενορχήστρωσης εφαρμογών και προς την κατεύθυνση της πραγματοποίησης του οράματος του IoT-edge-cloud continuum. Η ανάπτυξη καινοτόμων μηχανισμών συνεργατικής ενορχήστρωσης συνεισφέρει στην επίλυση κρίσιμων προκλήσεων που αντιμετωπίζουν οι σύγχρονες κατανεμημένες εφαρμογές σε ετερογενή περιβάλλοντα.

Η επιτυχής υλοποίηση και πειραματική επίδειξη του συστήματος επιβεβαιώνει την πρακτική βιωσιμότητα της προτεινόμενης προσέγγισης και δημιουργεί νέες ευκαιρίες για περαιτέρω έρευνα. Οι προτεινόμενες κατευθύνσεις μελλοντικής εργασίας παρέχουν ορισμένα από τα αναγκαία βήματα για την εξέλιξη του συστήματος προς μια πιο ώριμη και παραγωγικά έτοιμη (production ready) λύση.

Η συνεχής εξέλιξη των τεχνολογιών cloud computing, edge computing και IoT καθιστά επιτακτική την ανάγκη για προσαρμοστικά και ευέλικτα συστήματα ενορχήστρωσης. Η παρούσα εργασία αποτελεί σημαντικό βήμα προς αυτή την κατεύθυνση, παρέχοντας θεωρητικές και πρακτικές βάσεις για μελλοντικές εξελίξεις στον τομέα της κατανεμημένης ενορχήστρωσης εγγενών εφαρμογών υπολογιστικού νέφους σε υπερ-κατανεμημένα περιβάλλοντα.

Βιβλιογραφία

[1] Akamai, "What are cloud-native applications?", *Akamai*, [Online]. Available: https://www.akamai.com/glossary/what-are-cloud-native-applications. [Accessed: 1-May-2025].

[2] Cloud Native Computing Foundation, Who We Are, [Online]. Available: https://www.cncf.io/about/who-we-are/. [Accessed: 1-May-2025].

[3] M. Meenu, *Cloud-Native vs Traditional Architecture: A Comparative Deep Dive*, Medium, Feb. 6, 2024. [Online]. Available: https://medium.com/@hellomeenu1/cloud-native-vs-traditional-architecture-a-comparative-deep-dive-ff65d8929f73. [Accessed: 1-May-2025].

[4] S. Roy, *Microservices in the Cloud-Native Architecture*, Medium, Dec. 18, 2023. [Online]. Available: https://medium.com/@iamsoumyadip/microservices-in-the-cloud-native-architecture-723eb65f2f33. [Accessed: 1-May-2025].

[5] Aqua Security, *Microservices and Containerization*, Aqua Cloud Native Academy. [Online]. Available: https://www.aquasec.com/cloud-native-academy/docker-container/microservices-and-containerization/. [Accessed: 1-May-2025].

[6] S. Susnjara and I. Smalley, "What Is Container Orchestration?", *IBM*, Oct. 22, 2024. [Online]. Available: https://www.ibm.com/think/topics/container-orchestration. [Accessed: 7-May-2025].

[7] Google Cloud, *What Is Container Orchestration?*, [Online]. Available: https://cloud.google.com/discover/what-is-container-orchestration. [Accessed: 7-May-2025].

[8] Red Hat, *What is container orchestration?*, [Online]. Available: https://www.redhat.com/en/topics/containers/what-is-container-orchestration#what-is-container-orchestration-used-for. [Accessed: 7-May-2025].

[9] Kubernetes Authors, *Best Practices*, Kubernetes Documentation. [Online]. Available: https://kubernetes.io/docs/setup/best-practices/. [Accessed: 7-May-2025].

[10] Red Hat, *OpenShift Container Platform 4.18 Documentation*, [Online]. Available: https://docs.redhat.com/en/documentation/openshift_container_platform/4.18/. [Accessed: 7-May-2025].

[11] Docker Inc., *Swarm mode*, Docker Documentation. [Online]. Available: https://docs.docker.com/engine/swarm/. [Accessed: 7-May-2025].

[12] S. Susnjara and I. Smalley, "What Is Kubernetes?", *IBM*, Mar. 11, 2024. [Online]. Available: https://www.ibm.com/think/topics/kubernetes. [Accessed: 17-May-2025].

[13] SERRANO Project, *Transparent Application Deployment in a Secure, Accelerated and Cognitive Cloud Continuum*, [Online]. Available: https://ict-serrano.eu/. [Accessed:18-May-2025].

[14] T. Panagiotou et al., "SERRANO: A Cognitive Cloud Continuum Framework for Transparent and Secure Application Deployment," in *Proc. of DATE Conf.*, 2023. [Online]. Available: https://past.date-conference.com/proceedings-archive/2023/DATA/4016.pdf [Accessed: 18-May-2025].

[15] etcd Authors, *etcd – A distributed, reliable key-value store*, [Online]. Available: https://etcd.io [Accessed: 18-May-2025]

[16] EMPYREAN Project, *Trustworthy, Cognitive and AI-Driven Collaborative Associations of IoT Devices and Edge Resources for Data Processing*. [Online]. Available: https://empyrean-horizon.eu/. [Accessed: 19-May-2025].

[17] S. Ramírez, *FastAPI*, [Online]. Available: https://fastapi.tiangolo.com/. [Accessed: 2-June-2025].

[18] Pydantic, *BaseModel*, [Online]. Available: https://docs.pydantic.dev/latest/api/base_model/#pydantic.BaseModel. [Accessed: 2-June-2025].

[19] containerd Authors, *containerd – An industry-standard container runtime with an emphasis on simplicity, robustness and portability*, [Online]. Available: https://containerd.io/. [Accessed: 18-June-2025].

[20] CRI-O Authors, *CRI-O – an OCI-based Kubernetes Container Runtime Interface implementation*, [Online]. Available: https://cri-o.io/. [Accessed: 18-Jun-2025].

[21] Kubernetes Authors, *Container Runtimes*, Kubernetes Documentation, last modified Jun 11, 2025. [Online]. Available: https://kubernetes.io/docs/setup/production-environment/container-runtimes/. [Accessed: 18-Jun-2025].

[22] Kubernetes Authors, *kubelet*, Kubernetes Documentation, last modified Feb. 25, 2025. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/. [Accessed: 18-Jun-2025].

[23] Kubernetes Authors, *kube-proxy*, Kubernetes Documentation, last modified Apr. 24, 2025. [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/. [Accessed: 18-Jun-2025].

[24] Kubernetes Authors, *Cluster Architecture*, Kubernetes Documentation, last modified Oct. 20, 2024. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/. [Accessed: 18-Jun-2025].

[25] kind Authors, kind – a tool for running local Kubernetes clusters using Docker container "nodes", [Online]. Available: https://kind.sigs.k8s.io/. [Accessed: 18-Jun-2025].

[26] Carrión, C. Kubernetes as a Standard Container Orchestrator - A Bibliometric Analysis. J Grid Computing 20, 42 (2022). https://doi.org/10.1007/s10723-022-09629-8. [Accessed: 18-Jun-2025].

[27] C. Adamson, "Running Local Kubernetes Clusters with Kind: Using Docker Containers as Nodes," LinkedIn, Jan. 12, 2024. [Online]. Available: https://www.linkedin.com/pulse/running-local-kubernetes-clusters-kind-using-docker-nodes-ada. [Accessed: 18-Jun-2025].

[28] Better Stack Community, Getting Started with Kind for Local Kubernetes Development, Better Stack Community, Mar. 31, 2025. [Online]. Available: https://betterstack.com/community/guides/scaling-docker/kind/. [Accessed: 18-Jun-2025]

[29] Kohsuke Kawaguchi, *Jenkins – an open-source automation server*, Jenkins. [Online]. Available: https://www.jenkins.io/. [Accessed: 18-Jun-2025].

[30] Tekton Authors, *Tekton – a cloud-native framework for CI/CD*, [Online]. Available: https://tekton.dev/. [Accessed: 18-Jun-2025].

[31] Argo Project Authors, *Argo CD – Declarative continuous delivery tool for Kubernetes*, [Online]. Available: https://argoproj.github.io/cd/. [Accessed: 18-Jun-2025].

[32] SchedMD, *Slurm Workload Manager Documentation*, last modified May 23, 2025. [Online]. Available: https://slurm.schedmd.com/documentation.html. [Accessed: 18-Jun-2025].

[33] M. Fitzpatrick, *PyQt5 Tutorial – Create GUI Applications with Python (Qt 5)*, Python GUIs, last updated May 19, 2025. [Online]. Available: https://www.pythonguis.com/pyqt5-tutorial/. [Accessed: 18-Jun-2025].

[34] N. Elgiriyewithana, *Apple Quality*, Kaggle Dataset, 2023. [Online]. Available: https://www.kaggle.com/datasets/nelgiriyewithana/apple-quality. [Accessed: 18-Jun-2025].

[35] E. M. Varvarigos (Director), *High-Speed Communication Networks Laboratory*, School of Electrical and Computer Engineering, National Technical University of Athens. [Online]. Available: http://hscnl.ece.ntua.gr/. [Accessed: 18-Jun-2025].

[36] Kong Inc., *Insomnia – The collaborative API development platform for REST, GraphQL, gRPC, SOAP & WebSockets*, [Online]. Available: https://insomnia.rest/. [Accessed: 18-Jun-2025].