ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Συγχρονισμός εγκατεστημένων εφαρμογών με αποθετήριο εικόνων container

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ορφέας Ιωάννης. Ζωγράφος

Αθήνα, Ιούνιος 2025

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

Συγχρονισμός εγκατεστημένων εφαρμογών με αποθετήριο εικόνων container

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Ορφέας Ιωάννης. Ζωγράφος

Επιβλέπων: Παναγιώτης Τσανάκας
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 27η Ιούνη 2025.

. . . . .	. . . . .	. . . . .
Παναγιώτης Τσανάκας	Δημήτριος Σούντρης	Σωτήριος Ξύδης
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ

Αθήνα, Ιούνιος 2025

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Application deployment synchronization with container image repository

DIPLOMA THESIS

Orfeas Ioannis. Zografos

Athens, June 2025

NATIONAL TECHINCAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF COMPUTER SCIENCE

Application deployment synchronization with container image repository

DIPLOMA THESIS

Orfeas Ioannis. Zografos

Supervisor: Panagiotis Tsanakas
Professor NTUA

Approved by the three-member examination committee on the 27th June 2025.

. . . . .	. . . . .	. . . . .
Panagiotis Tsanakas	Dimitrios Soundris	Sotirios Xydis
Professor NTUA	Professor NTUA	Professor NTUA

Athens, June 2025

. . . . . .

Ορφέας Ιωάννης. Ζωγράφος
Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών ΕΜΠ

στους γονείς μου

Simplicity is the ultimate sophistication.

— Leonardo da Vinci

Η στοιχειοθεσία του κειμένου έγινε με το XeTeX 0.999996.

Χρησιμοποιήθηκαν οι γραμματοσειρές Minion Pro, Myriad Pro και `Consolas`.

Περίληψη

Η εποχή του 'συννέφου' έφερε μαζί της νέες λύσεις σε προβλήματα που ταλαιπωρούσαν τις ομάδες λογισμικού από τότε που υπάρχει λογισμικό. Αυτά τα νέα εργαλεία έχουν ωριμάσει αρκετά ώστε ο κλάδος να έχει μια καλή κατανόηση των δυνατοτήτων τους. Μέχρι πρόσφατα, οι περισσότερες ομάδες έπρεπε να κατασκευάζουν τις δικές τους λύσεις για αυτά τα προβλήματα, συχνά από το μηδέν, πράγμα που οδήγησε σε σπατάλη λόγω επανάληψης και επένδυσης χρόνου σε προβλήματα που είχαν ήδη λυθεί, και τις περισσότερες φορές είναι άσχετα με το προϊόν που κάποια ομάδα αναπτύσσει.

Το CERN, όντας συχνά στην αιχμή της τεχνολογικής προόδου, υπήρξε ένας από τους πρώτους οργανισμούς που υιοθέτησαν αυτά τα εργαλεία, διαχειριζόμενο την υποδομή εσωτερικά. Έχει ήδη υιοθετήσει σε μεγάλο βαθμό τη χρήση του Kubernetes και της ανάπτυξης λογισμικού με βάση τα containers για το μεγαλύτερο μέρος του λογισμικού που στοχεύει στο ευρύ κοινό (βλ. web), και βρίσκεται αυτή τη στιγμή στη διαδικασία μετάβασης προς την ίδια κατεύθυνση και για κρίσιμης σημασίας εσωτερικό λογισμικό.

Αυτή η διπλωματική εργασία καταγράφει τη διαδικασία προσαρμογής των υπαρχόντων εργαλείων μιας συγκεκριμένης ομάδας του CERN, της SY-EPC-CCS, ώστε να αξιοποιήσει αυτό το νέο σύνολο εργαλείων. Περιγράφει τη μετάβαση από μια «παραδοσιακή», χειροκίνητα διαχειριζόμενη υποδομή σε μια πλήρως αυτοματοποιημένη, containerized υποδομή βασισμένη στο Kubernetes, συμβάλλοντας παράλληλα στην ανάπτυξη και την εξέλιξη του εσωτερικού cloud του CERN και την έξοδό του από το στάδιο του proof of concept.

Χρησιμοποιώντας εργαλεία που αποτελούν πρότυπα της βιομηχανίας, όπως — αλλά όχι μόνο — τα Kubernetes, Helm, ArgoCD και Docker Compose, καθώς και εσωτερικά σχεδιασμένα εργαλεία και συμβάσεις, καταφέραμε να μεταφέρουμε την ομάδα και τα εργαλεία της σε μια υποδομή που απαιτεί πολύ λιγότερη χειροκίνητη παρέμβαση για το μεγαλύτερο μέρος του κύκλου ανάπτυξης. Παράλληλα οι υποδομές ακολουθούν πλέον τη νοοτροπία του δηλωτικού προγραμματισμού, μειώνοντας το νοητικό φορτίο της χειροκίνητης παρέμβασης όπου αυτή παραμένει απαραίτητη.

Λέξεις-Κλειδιά

CERN, Kubernetes, DevOps, Cloud, Continuous Integration, Continuous Deployment, Infrastructure as Code, Gitlab, GitlabCI, Docker, Containers, ArgoCD

Abstract

The era of the cloud has brought with it new solutions to problems that have troubled software teams for as long as software itself has been around. These new tools have matured enough for the industry to have a good understanding of their strengths and weaknesses. Up until recently most teams would have to build their own solutions to these, often from scratch. This has led to a lot of duplication of effort, and a lot of time spent on problems that are not the core business of the team.

CERN, often being at the forefront of technological progress, has been one of the early adopters of these tools, handling the infrastructure side of it in-house. It has already strongly embraced the use of Kubernetes and container oriented development for most internet facing software it develops, and is currently in the process of moving towards the same direction for mission critical, internal software.

This thesis documents the process of adapting the existing tools of a specific team at CERN, SY-EPC-CCS to utilize this new paradigm. It describes the process of moving from ”traditional”, manually managed infrastructure to a fully automated, containerized, Kubernetes based one, and in the process, assisting in the development and iteration of the CERN internal cloud itself, and its exit from the proof of concept stage.

Using industry standard tools, including but not limited to Kuberntes, Helm, ArgoCD and Docker Compose, as well as custom made utilities and conventions we designed in-house, we managed to migrate the team and its tools to infrastructure that requires a lot less manual intervention for most of the development cycle, and which follows a declarative approach, reducing the mental overhead wherever manual intervention is still needed, while providing much needed Quality of Life improvements to the development team.

Keywords

CERN, Kubernetes, DevOps, Cloud, Continuous Integration, Continuous Deployment, Infrastructure as Code, Gitlab, GitlabCI, Docker, Containers, ArgoCD

Αντί Προλόγου

Ως είθισται, οφείλω εδώ να ευχαριστήσω όλα τα άτομα που άμεσα ή έμμεσα συνέβαλαν στην ολοκλήρωση αυτής της εργασίας, με βοήθησαν στην πορεία μου στη σχολή, στο CERN και γενικότερα στην προσωπική μου ζωή.

Στους καθηγητές μας, οι οποίοι παρά τις συχνά αντίξοες συνθήκες (π.χ. υποχρεωτικές εξ'αποστάσεως παράδόσεις μαθημάτων) μπόρεσαν να μας μεταδώσουν τη γνώση τους και σημαντικότερα το πάθος τους για το αντικείμενό τους. Συγκεκριμένα ονόματα που (ελπίζω) να θυμάμαι για πολλά χρόνια από τώρα είναι οι κ. Γκούμας, κ. Παπασπύρου, κ. Σούντρης, κ. Ράπτης, κ. Γλύτσης, και βεβαίως ο κ. Τσανάκας ο οποίος έδρασε κι ως επιβλέπων της παρούσας εργασίας. Μας δώσατε εργαλεία που έχουν ήδη αποδόσει καρπούς και είμαι σίγουρος ότι θα συνεχίσουν να το κάνουν και στο μέλλον.

Στους φίλους και συμφοιτητές μου, χωρίς τους οποίους η εμπειρία της σχολής και εκτός αυτής θα ήταν σίγουρα πολύ δυσκολότερη. Βασίλη, Κώστα, Γιώργο, Λυδία και Χριστίνα, σας ευχαριστώ που υποφέρατε τη γκρίνια μου και μου μοιραστήκατε τη δική σας, καθώς και στους παιδικούς μου φίλους Αντώνη, Βαγγέλη και Κυριάκο.

Στα μέλη της ομάδας SY-EPC-CCS του CERN, που μου έδειξαν πόσο γρήγορα μπορεί κανείς να νιώσει πραγματικά οικεία, παρά την απόσταση και τις γλώσσες που μας χωρίζουν. Με κάνατε να νιώσω χρήσιμος από την πρώτη μερα (ελπίζω πώς ήμουν πράγματι) και δε σταμάτησε η υποστήριξή σας μόνο στα σύνορα του CERN.

Τέλος, στην οικογένειά μου, στους γονείς μου Γιάννη και Ναταλία που με άντεξαν όλ'αυτά τα χρόνια, στην αδερφή μου Ιόλη που υπήρξε μια μόνιμη πηγή ανεμελιάς και

γέλιου, και στον αδερφό μου Άγγελο, που ήταν και είναι εκεί για μένα απ’την αρχή μέχρι σήμερα.

Ευχαριστώ

Ορφέας Ζωγράφος
Ιούνης 2025

Περιεχόμενα

1 Εισαγωγή

Στο πρώτο κεφάλαιο θα προσπαθήσουμε να περιγράψουμε το ιστορικό και τεχνολογικό υπόβαθρο της παρούσας εργασίας, να δώσουμε μια περιγραφή των ζητημάτων και των απαιτήσεων που καθοδήγησαν αυτή τη δουλειά, καθώς και τυχόν υπάρχουσες λύσεις όπου αυτές είναι εφαρμόσιμες. Θα τελειώσουμε με μια σύντομη περίληψη της προσπάθειας.

1.1 Κίνητρο

Παρόλο που το διαδίκτυο δε θεωρείται πλέον νέα τεχνολογία, ο τεχνολογικός κόσμος βρίσκεται ακόμα σε νηπιακή ηλικία όσον αφορά την ανακάλυψη τρόπων για να εξερευνήσει σωστά τις δυνατότητες που προσφέρει. Ένα καλό παράδειγμα είναι η έννοια του cloud computing, ένας σχετικά πρόσφατος όρος - μόνο από τη δεκαετία του 1990 - και με εξελίξεις όπως το Kubernetes να είναι ακόμα πιο πρόσφατες (2015). Το Kubernetes μαζί με το Docker, έχουν επαναστατήσει την έννοια του cloud, φέρνοντάς την στο προσκήνιο.

1.2 CERN

Το CERN έχει υπάρξει στο προσκήνιο τέτοιων εξελίξεων καθ' όλη τη διάρκεια της ύπαρξής του, με τη δημιουργία του World Wide Web να πιστώνεται σε ένα από τα μέλη του, αλλά και με λιγότερο γνωστά έργα όπως το LHC Computing Grid. Δεν είναι λοι-

πόν μεγάλη έκπληξη το γεγονός ότι το επόμενο βήμα στην υποδομή υπολογιστών του CERN διαμορφώνεται να είναι μια μετάβαση προς το Kubernetes. Στην πραγματικότητα, η τυπική ανάπτυξη για ένα μεγάλο μέρος των εφαρμογών του οργανισμού βασίζεται ήδη στο Kubernetes.

Αυτά τα εργαλεία, έχουν ήδη αποδείξει την ανεκτίμητη αξία τους στην απλοποίηση της διαδικασίας ανάπτυξης και διάθεσης εφαρμογών, γεγονός που τα καθιστά μια ευπρόσδεκτη προσθήκη στις ομάδες προγραμματιστών, παρόλο που υπάρχει ακόμη λίγο βάρος που σχετίζεται με τη χρήση τους.

Το τελευταίο υποσύνολο εσωτερικών εργαλείων που απομένει για να μεταβεί σε χρήση κοντέινερ και Kubernetes, είναι τα πιο αναγκαία, οι εφαρμογές του Τεχνικού Δικτύου (TN) του CERN, συνήθως κρίσιμες για την αποστολή του CERN εφαρμογές, που υπάρχουν και είναι προσβάσιμες μόνο εντός ενός απομονωμένου δικτύου κυρίως για λόγους ασφαλείας, και συνήθως απαιτούν ισχυρές εγγυήσεις υψηλής διαθεσιμότητας.

Αυτό είναι αυτό που προσπαθεί να λύσει ένα πρόσφατο κοινό έργο μεταξύ δύο τμημάτων του CERN (η πρωτοβουλία ATS-IT Kubernetes), παρέχοντας μια πλατφόρμα Kubernetes μέσα στο Τεχνικό Δίκτυο (TN).

1.3 SY-EPC-CCS

1.3.1 Συστήματα επιταχυντών (SY)

Το SY τμήμα είναι υπεύθυνο για την ανάπτυξη, συντήρηση και λειτουργία των συστημάτων επιταχυντών του CERN, συμπεριλαμβανομένων των συστημάτων που σχετίζονται με τη μεταφορά και την επεξεργασία της δέσμης, όπως τα συστήματα τροφοδοσίας ρεύματος, τα συστήματα ραδιοσυχνοτήτων και τα συστήματα απορρόφησης και αποκομιδής.

1.3.2 Μετατροπείς ηλεκτρικής ισχύος (EPC)

Το EPC τμήμα είναι υπεύθυνο για την ανάπτυξη, συντήρηση και λειτουργία των συστημάτων τροφοδοσίας ρεύματος του CERN, συμπεριλαμβανομένων των συστημάτων τροφοδοσίας ρεύματος για τους επιταχυντές, τις γραμμές μεταφοράς και τις περιοχές

πειραμάτων. Αυτό περιλαμβάνει τους μετατροπείς ισχύος που χρησιμοποιούνται για την τροφοδοσία των μαγνητών και άλλων συσκευών που αποτελούν τους επιταχυντές, καθώς και τις τροφοδοσίες ρεύματος για τους διάφορους ανιχνευτές και άλλες συσκευές που χρησιμοποιούνται στα πειράματα.

1.3.3 Λογισμικό ελέγχου μετατροπέων (CCS)

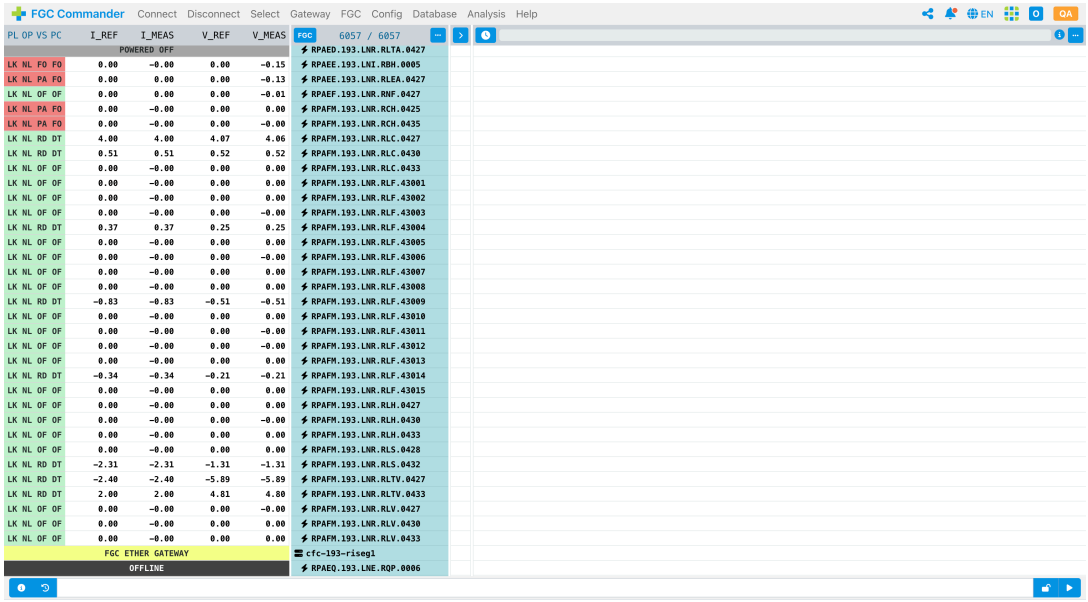
Το SY-EPC-CCS χειρίζεται το λογισμικό ελέγχου για τους μετατροπείς ισχύος που χρησιμοποιούνται στους επιταχυντές και άλλες υποδομές του CERN. Το τμήμα είναι χωρισμένο σε τρεις υποτμήματα: η ομάδα FGC, η οποία χειρίζεται το firmware και το λογισμικό χαμηλού επιπέδου που αλληλεπιδρά άμεσα με το υλικό ελέγχου (δημιουργημένο από το αδελφικό της τμήμα SY-EPC-CCE). Η ομάδα Tools, η οποία είναι υπεύθυνη για την ανάπτυξη των εφαρμογών που επιτρέπουν στους ειδικούς ελέγχου να διαμορφώνουν, να ρυθμίζουν και να επιλύουν προβλήματα διάφορων ρυθμίσεων παρέχοντας ένα φιλικό προς τον χρήστη περιβάλλον και ισχυρές δυνατότητες επεξεργασίας και οπτικοποίησης δεδομένων. Τέλος, η ομάδα Βιομηχανικών Ελέγχων, αναπτύσσει και κυρίως συντηρεί τη λειτουργικότητα όλων των πλατφορμών ελέγχου βασισμένων σε PLC.

Τα εργαλεία που παρέχονται από την ομάδα Tools είναι συχνά υπό συνεχή ανάπτυξη με νέες δυνατότητες να ζητούνται από τους χρήστες τους, και πρέπει να είναι υψηλής διαθεσιμότητας για να διευκολύνουν την έρευνα και την επίλυση προβλημάτων όσο το δυνατόν γρηγορότερα. Αυτές είναι μόνο δύο από τις αιτίες που η ομάδα αποφάσισε να ακολουθήσει το έργο ATS-IT Kubernetes ακόμα και πριν αυτό βγει από τη φάση Proof of Concept.

FGC Commander

Ο FGC Commander είναι το εργαλείο-στολίδι της ομάδας. Σκοπός του είναι να παρέχει έναν φιλικό προς τον χρήστη τρόπο για να επιθεωρεί και να ρυθμίζει όλους τους μετατροπείς ισχύος που βρίσκονται υπό την εποπτεία του EPC. Είναι μια εφαρμογή ιστού, που χρησιμοποιείται από ειδικούς από πολλές άλλες ομάδες, με διαφορετικά επίπεδα γνώσης υπολογιστών, και για αυτό το λόγο, παρέχει όσο το δυνατόν περισσότερες πληροφορίες μέσω οπτικών ενδείξεων, καθώς και όσο το δυνατόν περισσότερη βοήθεια για την εκτέλεση εντολών με σκοπό τη διάγνωση και την προσαρμογή της συμπεριφοράς

των μετατροπέων.



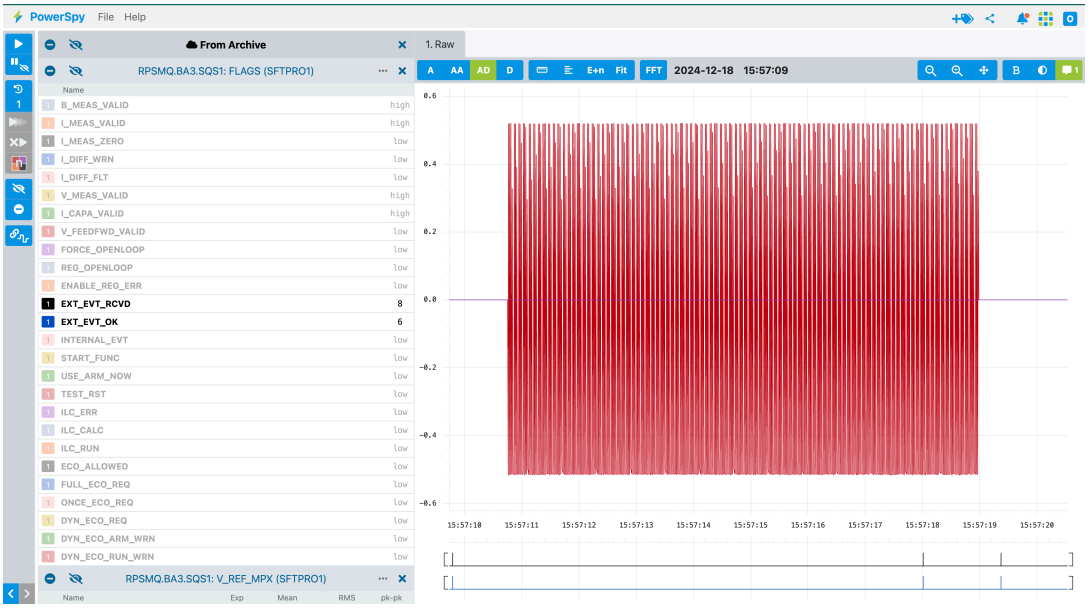
Σχήμα 1.1: *FGC Commander*

FGC API

Το FGC API είναι η ραχοκοκαλιά των περισσότερων εργαλείων της ομάδας. Είναι ένα κεντρικό RESTful API που παρέχει μια αφαίρεση πάνω από το firmware που τρέχει στους Frontend Υπολογιστές και τους μετατροπείς ισχύος. Χρησιμοποιείται από τις περισσότερες άλλες εφαρμογές ιστού για να αλληλεπιδράσουν με το υλικό και να στείλουν εντολές σε αυτό. Το γεγονός ότι είναι κεντρικό, σημαίνει ότι υπάρχει η μικρότερη δυνατή επανάληψη στον κώδικα, ειδικά για κρίσιμης σημασίας λογική, και επειδή είναι επίσης RESTful API, είναι εύκολο να κλιμακωθεί οριζόντια.

Powerspy

Το Powerspy είναι το κύριο εργαλείο διαγνωστικών και ανάλυσης. Χρησιμοποιεί μια προσαρμοσμένη βιβλιοθήκη οπτικοποίησης στο frontend, για να επιτρέπει πολύ υψηλής ανάλυσης γραφήματα, και στο backend συνδέεται με μια σειρά από διαφορετικές βάσεις δεδομένων, για να παρέχει πρόσβαση σε ιστορικά δεδομένα, που μπορούν να αποκτηθούν χειροκίνητα χρησιμοποιώντας την ίδια εφαρμογή, ή αυτόματα μετά από αποτυχίες ή άλλα γεγονότα.



Σχήμα 1.2: *Powerspy*

1.4 Διατύπωση προβλήματος

Το έργο του εκσυγχρονισμού της υποδομής των εργαλείων της ομάδας μπορεί να χωριστεί σε μερικά βήματα:

1. Προσαρμογή όλων των εφαρμογών για χρήση σε container
2. Προσθήκη υποστήριξης για τοπική ανάπτυξη εφαρμογών
3. Διάθεση των εφαρμογών μέσω υπολογιστικών συμπλεγμάτων Kubernetes
4. Αυτοματοποίηση ενημερώσεων και αναβαθμίσεων (CI/CD)

1.4.1 Προσαρμογή όλων των εφαρμογών για χρήση σε κοντέινερ

Οι βασικές δυσκολίες εδώ ήταν η αναγνώριση των εξαρτήσεων συστήματος κάθε εφαρμογής, οι οποίες τεκμηριώνονταν μόνο στις λίστες που χρησιμοποιήθηκαν για την αίτηση των συστημάτων απ'το IT department του CERN, και οι οποίες ήταν ήδη εγκατεστημένες στα μηχανήματα όπου οι εφαρμογές είχαν ήδη αναπτυχθεί. Επομένως, αυτή ήταν επίσης μια καλή ευκαιρία να τυποποιηθεί η διαχείριση των μεταβλητών περιβάλλοντος, και να αφαιρεθούν περιττές εξαρτήσεις συστήματος. Επιπλέον, έπρεπε να

αντιμετωπιστούν οι μεταβλητές περιβάλλοντος των εφαρμογών, ειδικά όταν πρόκειται για frontends εφαρμογών, καθώς αυτές αντικαθίστανται κατά τη διάρκεια της διαδικασίας κατασκευής της εφαρμογής, οδηγώντας σε ένα πρόβλημα $N \times M$ (N εφαρμογές-M στόχοι ανάπτυξης).

1.4.2 Προσθήκη υποστήριξης για τοπική ανάπτυξη εφαρμογών

Όπως αναφέρθηκε προηγουμένως, το CERN χρησιμοποιεί ένα Τεχνικό Δίκτυο (TN), το οποίο είναι παράλληλο με ένα Γενικό Δημόσιο Δίκτυο (GPN) που έχει πρόσβαση στο Διαδίκτυο, αλλά είναι προσβάσιμο μόνο μέσω συγκεκριμένων πυλών, κυρίως για λόγους ασφαλείας, αλλά και για νομικούς λόγους, καθώς το CERN έχει συμβάσεις που παρέχουν πρόσβαση σε λογισμικό μόνο εντός των δικτύων του.

Αυτό θέτει μερικές ενδιαφέρουσες προκλήσεις όταν προσπαθεί κανείς να υποστηρίξει την ανάπτυξη από τις τοπικές μηχανές των χρηστών, ειδικά σήμερα, όπου η τηλεργασία είναι μια πραγματικότητα πολλών χώρων εργασίας. Έπρεπε να βρεθούν λύσεις για την πρόσβαση στο εσωτερικό αποθετήριο πακέτων python του CERN, σε συγκεκριμένες εικόνες Docker από το αποθετήριο εικόνων κοντέινερ του CERN, και πιο σημαντικά, για τη δρομολόγηση όλης της δικτυακής κίνησης εφαρμογών μέσω του TN, για να προσομοιωθεί το πραγματικό περιβάλλον λειτουργίας και να υπάρχει πρόσβαση σε πραγματικές συσκευές κατά την ανάπτυξη εφαρμογών.

1.4.3 Διάθεση των εφαρμογών μέσω Kubernetes clusters

Η αντιστοίχηση ενός περιβάλλοντος Docker σε ένα Kubernetes, αν και πολύ πιο απλή από την αναπαραγωγή ενός bare-metal σε έναν νέο υπολογιστή, έχει ακόμα μερικές προκλήσεις. Στην περίπτωσή μας, υπήρχε η πρόσθετη ανησυχία να γίνει με τέτοιο τρόπο ώστε να υποστηρίζεται εύκολα η ανάπτυξη σε διαφορετικά clusters χρησιμοποιώντας την ίδια διαμόρφωση.

1.4.4 Αυτοματοποίηση ενημερώσεων και αναβαθμίσεων (CI/CD)

Τέλος, υπήρχε ανάγκη για αυτοματοποίηση των διαδικασιών ανάπτυξης και αναβάθμισης των εφαρμογών, ώστε να γίνεται όσο το δυνατόν πιο ανώδυνα. Εδώ ο ιδανικός

τελικός στόχος θα ήταν η αφαίρεση από τους προγραμματιστές όλων των λεπτομερειών που σχετίζονται με την κατασκευή και την διάθεση μιας νέας έκδοσης μιας εφαρμογής. Αν και είναι μια σχετικά κοινή απαίτηση στη βιομηχανία, τελικά απαιτήθηκε αρκετή έρευνα και μερικά προσαρμοσμένα εργαλεία και λύσεις.

1.5 Υπάρχουσες λύσεις

1.5.1 Προσθήκη υποστήριξης για τοπική ανάπτυξη εφαρμογών

Στο παρελθόν για την ανάπτυξη των εφαρμογών χρησιμοποιούνταν μια εικονική μηχανή που φιλοξενούνταν στο Τεχνικό Δίκτυο (ΤΝ), όπου οι προγραμματιστές πραγματοποιούσαν όλη την ανάπτυξη των εφαρμογών. Αυτή ήταν μια προβληματική λύση, καθώς η παροχή των υπολογιστών απ'το ΙΤ συχνά καθυστερούσε, προσέθετε πολύ φόρτο εργασίας διαχείρισης συστήματος όσον αφορά τις ενημερώσεις και τις αναβαθμίσεις του λειτουργικού συστήματος, καθώς και τη ρύθμιση και τη διαμόρφωση των μηχανών, και αύξησε την επιφάνεια επίθεσης του ίδιου του Τεχνικού Δικτύου.

1.5.2 Διάθεση των εφαρμογών μέσω Kubernetes clusters

Σχήμα 1.3: *Λογότυπο του Helm*

Μια υπάρχουσα λύση, αν μπορεί να χαρακτηριστεί έτσι, είναι η χρήση του Helm (βλ. [Hel]), που βοηθάει στο διαχωρισμό μεταξύ του στατικού και του μεταβλητού μέρους της διαμόρφωσης ενός υπολογιστικού συμπλέγματος. Αυτό αποδείχθηκε χρήσιμο για τη μείωση του φόρτου συντήρησης της διαμόρφωσης, ωστόσο, τα περισσότερα από τα προαπαιτούμενα και τα προβλήματα έπρεπε να αντιμετωπιστούν κατά περίπτωση.

1.5.3 Αυτοματοποίηση ενημερώσεων και αναβαθμίσεων (CI/CD)

Δεδομένου ότι αυτό είναι ένα σχετικά κοινό πρόβλημα, υπάρχουν μερικές λύσεις από την ανοιχτή κοινότητα αλλά και από τη βιομηχανία που ισχυρίζονται ότι χειρίζονται αυτό το μέρος του προβλήματος. Οι δύο κύριες είναι το ArgoCD και το Flux. Κάθε μία έχει τις δικές της σχεδιαστικές αποφάσεις και αδυναμίες, ωστόσο και οι δύο προσπαθούν να αντιμετωπίσουν το ζήτημα της Συνεχιζόμενης Παράδοσης (CD) σε έναν κόσμο όπου οι υπολογιστικές υποδομές απομακρύνονται απ'το διαδικαστικό, και κινούνται προς το δηλωτικό μοντέλο προγραμματισμού.

ArgoCD

Σχήμα 1.4: *Λογότυπο (και μασκότ) του ArgoCD*

Το ArgoCD έχει σχετικά αυστηρές σχεδιαστικές απόψεις, προσφέροντας μια πιο ευχάριστη εμπειρία κατά την εκκίνηση, και επικεντρώνεται περισσότερο στο να διευκολύνει τη συντήρηση και τη συγχρονισμό της διαμόρφωσης του cluster και των πόρων του Kubernetes.

Πίσω από το ArgoCD βρίσκεται κυρίως ο Ελεγκτής Εφαρμογών (Application Controller), ο οποίος χειρίζεται τους Προσαρμοσμένους Πόρους (Custom Resources) τύπου Application, οι οποίοι με τη σειρά τους χειρίζονται τα Helm charts που φιλοξενούνται σε ένα αποθετήριο git ή σε ένα αποθετήριο chart. Μετά την εγκατάσταση στο cluster, ο χρήστης ορίζει έναν πόρο Application για κάθε ένα από τα helm charts που θα αναπτυχθούν στο cluster, και ρυθμίζει το ArgoCD, μέσω της διεπαφής ιστοτόπου, ή απευθείας από μια γραμμή εντολών, να παρακολουθεί ένα συγκεκριμένο αποθετήριο, όπου παρακολουθούνται τα αρχεία ορισμού των πόρων Application. Στην πιο κοινή ρύθμιση, οι πόροι Application χρησιμοποιούνται για να καθορίσουν ένα αποθετήριο git όπου παρακολουθούνται τα helm charts, επιτρέποντας στο cluster να έχει πάντα την τελευταία έκδοση των charts.

Για τον συγχρονισμό των νέων εκδόσεων των κοντέινερ που χρησιμοποιούνται σε κάθε helm chart, η ομάδα του Argo αναπτύσσει ένα επιπλέον εργαλείο, το ArgoCD Image Updater, το οποίο ήδη υποστηρίζει τις περισσότερες κοινές περιπτώσεις χρήσης. Μπορεί να ελέγχει περιοδικά τα μητρώα εικόνων για νέες εικόνες των οποίων οι ετικέτες πληρούν κάποια κριτήρια, να εφαρμόζει ενημερώσεις και να δεσμεύει τις νέες ετικέτες που χρησιμοποιούνται πίσω σε ένα αποθετήριο git.

Flux

Σχήμα 1.5: *FluxCD logo*

Flux, από την άλλη πλευρά, είναι εξαιρετικά παραμετροποιήσιμο, και χωρίζεται όσο το δυνατόν περισσότερο σε πολλούς διαφορετικούς Προσαρμοσμένους Πόρους (CRDs) και ελεγκτές (Controllers), κάθε ένας από τους οποίους χειρίζεται ένα συγκεκριμένο μέρος των ευθυνών που χρειάζονται για να λειτουργήσει η Συνεχιζόμενη Παράδοση (CD) με το Kubernetes. Σχεδιαστικά, η ιδέα είναι η ίδια, αποτελούμενη από Ελεγκτές Kubernetes και Προσαρμοσμένους Πόρους. Το Flux 2, προήλθε από την αποτυχημένη προσπάθεια ενοποίησης του Argo και του Flux, και πρόσθεσε κάτι που ονομάζεται GitOps Toolkit, το οποίο επιτρέπει στο Flux να υποστηρίζει το σύνολο χαρακτηριστικών του ArgoCD image updater απευθείας.

1.6 Προτεινόμενη λύση

Κατά τη διάρκεια αυτής της προσπάθειας, λόγω των πόρων και της τεχνογνωσίας της ομάδας, ήταν ισχυρή απαίτηση να αποφευχθούν κατά το δυνατόν προσαρμοσμένες λύσεις, καθώς θα καθιστούσαν τη συντήρηση στο μέλλον πιο δύσκολη. Παρόλα αυτά, υπάρχουν μερικά μέρη όπου τελικά ήταν απαραίτητο να γίνει κάτι τέτοιο.

1.6.1 Προσθήκη υποστήριξης για τοπική ανάπτυξη εφαρμογών

Αυτή ήταν μια από τις περιπτώσεις όπου ήταν απαραίτητη μια προσαρμοσμένη λύση. Η εσωτερική τεκμηρίωση του CERN προτείνει τη χρήση διαφόρων τεχνικών προώθησης και αναφέρει ένα βοηθητικό πρόγραμμα που ονομάζεται sshuttle. Τελικά, το sshuttle χρησιμοποιήθηκε, αλλά με τέτοιο τρόπο ώστε να ξεκινά αυτόματα πριν από τις εφαρμογές, και να δρομολογεί όλη την δικτυακή κίνηση των εφαρμογών μέσω αυτού.

1.6.2 Διάθεση των εφαρμογών μέσω Kubernetes clusters

Για τη διάθεση σε clusters, δημιουργήσαμε ένα αρκετά προσαρμοσμένο σύνολο συμβάσεων στις οποίες σχεδόν όλες οι αναπτύξεις εφαρμογών συμμορφώθηκαν, και χρησιμοποίησαμε μια βιβλιοθήκη helm charts που δημιουργήθηκε σε συνεργασία με άλλο τμήμα του CERN. Αυτό διευκόλυνε πολύ την προσθήκη νέων εφαρμογών, και την ευκολία επαναδιαμόρφωσης, καθώς τώρα όλη η διαμόρφωση των εφαρμογών είναι δηλωτική, πράγμα που σημαίνει ότι δεν χρειάζεται να διαβάζουμε μέσα από σκριπτ για να διασφαλίσουμε ότι μια μικρή αλλαγή κάπου, δεν συγκρούεται ή δεν σπάει κάτι άλλο.

1.6.3 Αυτοματοποίηση ενημερώσεων και αναβαθμίσεων (CI/CD)

Ακολουθώντας το παράδειγμα άλλων ομάδων του CERN, και τις προτάσεις της ομάδας υποδομών, χρησιμοποιήσαμε το ArgoCD για την αυτοματοποίηση. Το Flux 2 ήταν ακόμα σχετικά νέο, και η προσέγγισή του σήμαινε ότι θα καταλήγαμε με περισσότερα Kubernetes Custom Resource που σχετίζονται με το Flux για συντήρηση, ενώ η προσέγγιση του ArgoCD είναι γενικά πιο φορητή σε περιβάλλοντα και διαμορφώσεις Kubernetes. Το αρχικό σχέδιο ήταν να χρησιμοποιήσουμε το ArgoCD Image Updater για την ενσωμάτωση git, αλλά ακόμα και μετά την προσπάθεια να ξεπεράσουμε τα προβλήματά του συμβάλλοντας στο έργο, οι συμβιβασμοί ήταν ακόμα πολλοί. Τελικά καταλήξαμε σε μια προσαρμοσμένη λύση που βασίζεται σε μεγάλο βαθμό στις δυνατότητες του ArgoCD, καθώς και στην πρότυπη γλώσσα helm, για να επιτύχουμε τα επιθυμητά αποτελέσματα.

1.7 Περίγραμμα

Τα επόμενα κεφάλαια θα είναι:

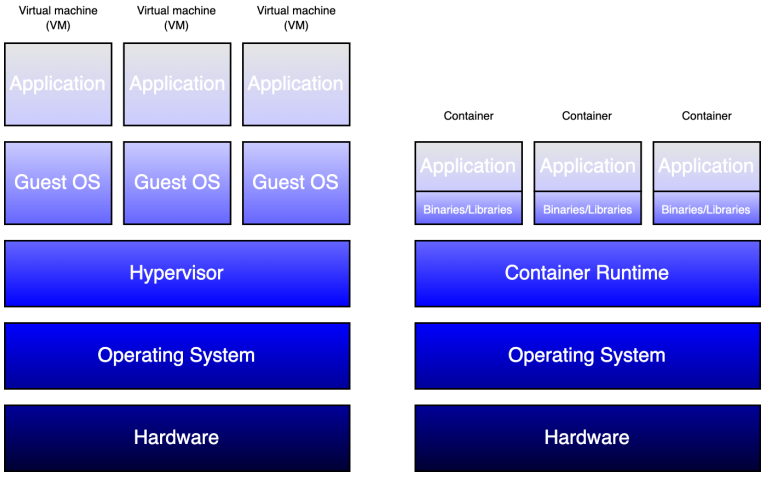
- **Κεφάλαιο 2:** Επισκόπηση της σχετικής ορολογίας και εννοιών
- **Κεφάλαιο 3:** Περιγραφή της υπάρχουσας τεχνολογίας και των προσθηκών μας
- **Κεφάλαιο 4:** Λεπτομέρειες υλοποίησης, δυσκολίες και λύσεις
- **Κεφάλαιο 5:** Απολογισμός, αποτελέσματα και μελλοντικό έργο

2 Υπόβαθρο

Το κεφάλαιο αυτό θα προσπαθήσει να δώσει μια γενική επισκόπηση της ορολογίας και των τεχνολογιών που απαιτούνται για τη κατανόηση της υπόλοιπης εργασίας. Αυτό συμπεριλαμβάνει την έννοια των container, του 'εγκιβωτισμού' (containerization), και κατ'επέκταση του Docker, καθώς επίσης και μια εισαγωγή στο Kubernetes. Τέλος μια περίληψη της έννοιας GitOps, και πιό συγκεκριμένα του GitlabCI.

2.1 Containers

Στη πορεία ανάπτυξης λογισμικού, τα προβλήματα διανομής και η φορητότητα είναι πάντα στο προσκήνειο, ειδικά όταν ένα λογισμικό βγαίνει απ'τα πρώιμα στάδια ανάπτυξης (αλλά κατά προτίμηση από νωρίτερα). Μοντέρνες γλώσσες και γλώσσες υψηλότερου επιπέδου αποφεύγουν κάποια απ'αυτά έχοντας runtime, χωρίς να χρειάζονται έτσι μεταγλώττιση για κάθε μηχάνημα-στόχο στο οποίο θα τρέξουν. Ωστότοσ, συνεχίζουν να υπάρχουν τα προβλήματα πακεταρίσματος και διαχείρισης βιβλιοθηκών. Ο εγκιβωτισμός (containerization) δοκιμάζει να μειώσει τέτοια προβλήματα, θέτοντας έναν προκαθορισμένο τρόπο να 'πακετάρεται' μια εφαρμογή μαζί με όλες τις εξαρτήσεις της, ενώ ταυτόχρονα βελτιώνει την απομόνωση μεταξύ των διεργασιών, και επομένως την ασφάλεια. Κατά κάποιο τρόπο, έχει ανταγωνιστική σχέση με τις παραδοσιακές Εικονικές Μηχανές για κάποιες απ'τις λειτουργίες τους.



Σχήμα 2.1: *Εικονικές μηχανές και Container*

Οι εικονικές μηχανές βασίζονται πάνω σ'ένα Hypervisor, ο οποίος πρακτικά προσομοιώνει σε επίπεδο λογισμικού, το υλικό ενός υπολογιστή, και αποτελούν ένα ολόκληρο νέο λειτουργικό σύστημα, πάνω απ'το προσομοιωμένο αυτό υλικό. Τα Containers, χρησιμοποιούν με έξυπνο τρόπο διάφορες δυνατότητες του Linux πυρήνα, όπως τα cgroups και τα kernel namespaces, ούτως ώστε να επιτρέψουν στις εφαρμογές εντός τους, να χρησιμοποιούν τον Linux πυρήνα του φιλοξενούντος (host) υπολογιστή για κλήσεις συστήματος και Είσοδο/Εξοδο, φέρνοντας έτσι την επίδοση σε σχεδόν ίδιο επίπεδο με τις διεργασίες που τρέχουν απευθείας στον φιλοξενούντα υπολογιστή. Γενικότερα μπορεί κανείς να σκέφτεται τα Container, ως μια προσομοίωση του Λειτουργικού συστήματος, πάνω στο λειτουργικό σύστημα του φιλοξενούντος υπολογιστή, σε αντίθεση με τις Εικονικές Μηχανές, που είναι προσομοίωση του υλικού, πάνω σ'έναν hypervisor.

2.1.1 Docker

Το Docker είναι η πιό ευρέως χρησιμοποιούμενη πλατφόρμα που υλοποιεί την ιδέα των Container, προσφέροντας εργαλεία για την κατασκευή, την διάθεση, και το τρέξιμο τους. Καθορίζει έναν στάνταρ τρόπο για τη δημιουργία και την διαχείριση Container. Η σουίτα Docker, συμπεριλαμβάνει εργαλεία όπως το το Docker Hub, ένα αποθετήριο εικονών για ευκολότερη διάθεση των εικονών Container, ένα κατασκευαστή αυτών τον εικονών (Builder), και βεβαίως το Docker Engine, για το τρέξιμο Container που υποστηρίζει επίσης ιδιαίτερες δυνατότητες Δικτύωσης Container και Volumes για την

διατήρηση αρχείων και δεδομένων μεταξύ διαφορετικών εκτελέσεων ενός Container.

Volumes

Τα Docker Volumes λοιπόν, είναι ένας μηχανισμός για τη διατήρηση δεδομένων μεταξύ εκτελέσεων ενός Container. Τα Container είναι από φύση τους εφήμερα, αλλά οι εφαρμογές που περιέχουν συχνά δεν είναι. Τα Volumes γίνονται mount σε κάποιο directory του Container, με παρόμοιο τρόπο που γίνεται mount ένα filesystem στο επίπεδο του Λειτουργικού Συστήματος. Δημιουργούνται είτε ξεχωριστά, είτε κατά την εκτέλεση ενός Container, και μπορούν να χρησιμοποιηθούν και για το διαμοιρασμό δεδομένων μεταξύ Container, εαν γίνει mount το ίδιο Volume σε διαφορετικά Container.

Δίκτυα

Οι λειτουργίες δικτύωσης του Docker, επιτρέπουν στα Container να επικοινωνούν μεταξύ τους, με το φιλοξενούν σύστημα ή με εξωτερικά δίκτυα. Παρέχει πολλή ευελιξία και έλεγχο ως προς τη συνδεσιμότητα, το Docker κάνει διαθέσιμους Drivers δικτύωσης με διαφορετικές δυνατότητες, για διαφορετικούς σκοπούς

- **Bridge:** Η προεπιλεγμένη μορφή δικτύωσης για αυτόνομα Container, απομονώνει το δίκτυο ενός Container απ'τον φιλοξενούντα υπολογιστή, αλλά επιτρέπει επικοινωνία μεταξύ διαφορετικών Container.
- **Host:** Αφαιρεί την απομόνωση, και επιτρέπει σ'ένα Container να χρησιμοποιεί το δίκτυακό stack του φιλοξενούντος υπολογιστή απευθείας.
- **None:** Απενεργοποιεί εντελώς υποστήριξη δικτυακών λειτουργιών.

Μερικά ακόμα, που δεν απαντώνται όμως στις συνήθεις χρήσεις:

- **Overlay:** Χρήσιμο σε περίπτωση που επιθυμείται σύνδεση διαφορετικών Docker διεργασιών, που ενδεχομένως τρέχουν και σε διαφορετικά μηχανήματα.
- **Ipvlan:** Επιτρέπει πλήρη έλεγχο στο IP επίπεδο, και μπορεί να χρησιμοποιηθεί για παράδειγμα για την σύνδεση ενός Container, με ένα υπάρχον, εξωτερικό IP δίκτυο.

- **Macvlan:** Μπορεί να χρησιμοποιηθεί για να εμφανίσει Containers ως φυσικές συσκευές σ'ένα δίκτυο.

2.1.2 Docker Compose

Το Docker Compose είναι ένα εργαλείο που παρέχεται απ'την ομάδα του Docker, που μπορεί να χρησιμοποιηθεί για να αποφευχθεί η συγγραφή πολύπλοκων script που κατασκευάζουν και τρέχουν Container ένα-ένα. Επιτρέπει στο χρήστη να καθορίσει όλα τα κομμάτια μιας εφαρμογής (που μπορεί να τρέχουν ως πολλά ξεχωριστά Container), καθώς και τις εξαρτήσεις μεταξύ τους, και μετά με μια απλή εντολή να τα τρέχει, χωρίς να πρέπει να ασχοληθεί ο ίδιος με το ποιά απ'όλα τα Container αυτά πρέπει να επανακατασκευαστούν, ή να ληφθούν απ'το αποθετήριό τους.

Σχήμα 2.2: *Λογότυπο και μασκότ του Docker Compose*

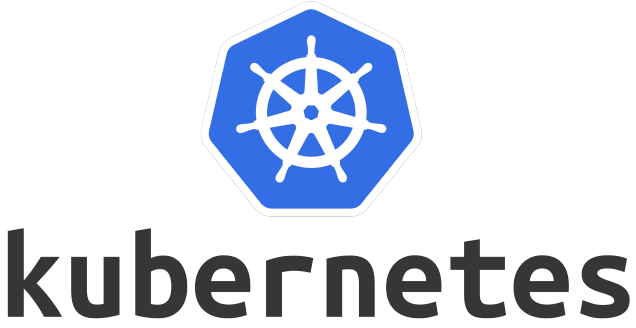
Η βασική μονάδα λειτουργίας του είναι το `docker-compose.yml`, ένα αρχείο που καθορίζει υπηρεσίες, δίκτυα και volumes που αποτελούν μια εφαρμογή. Μπορεί να επεκταθεί σε περισσότερα αρχεία, τα οποία μάλιστα μπορούν να χρησιμοποιηθούν για να αλλάζουν ή να προσθέτουν λειτουργία στη βασική μορφή της εφαρμογής, επανακαθορίζοντας κόμμάτια των πόρων.

```
version: '3.8'
services: # Application components or service dependencies
db:
# ...
network: stack-network
app-1-client:
# ...
network: stack-network
app-1-server:
# ...
network: stack-network
networks: # Docker network definitions and config
stack-network:
# ...
```

Listing 1: *Βασικές δυνατότητες του* `docker-compose.yml`

2.2 Kubernetes

Το ευκολότερο πακετάρισμα και κατά συνέπεια η διανομή λογισμικού, έχει διευκολύνει πολύ την κίνηση προς το σύννεφο, όμως το σύννεφο, είτε αυτοδιαχειριζόμενο είτε εξωτερικό, έχει δικά του προβλήματα. Εδώ λοιπόν έρχεται το Kubernetes. Το Kubernetes είναι ένα εργαλείο ενορχήστροσης, που αφαιρεί αρκετή απ'την πολυπλοκότητα της διάθεσης εφαρμογών σε συμπλέγματα υπολογιστών. Επιπλέον, ακολουθεί και αυτό δηλωτικής μορφής προγραμματισμό, που με λίγα λόγια σημαίνει ότι αφαιρεί αρκετά την ανάγκη για scripts. Καθώς το έργο αυτής της διπλωματικής βρίσκεται κατά κανόνα πάνω απ'το API του Kubernetes, θα περιγράψουμε κυρίως όρους από την οπτική ενός χρήστη του.



Σχήμα 2.3: *Λογότυπο Kubernetes*

Pods

Τα pods είναι η μικρότερη μονάδα που μπορεί να εκτελεστεί σ'έναν Kubernetes cluster, και αντιπροσωπεύει ένα ή περισσότερα στενά συνδεδεμένα μεταξύ τους Container, που μοιράζονται πόρους όπως χώρος αποθήκευσης και δίκτυο. Ο σκοπός τους είναι να φιλοξενούν μια εφαρμογή, ή ένα σύνολο από διεργασίες με στενή σχέση και έντονη απαίτηση για επικοινωνία μεταξύ τους, καθώς ένα pod δε μπορεί να καταμοιραστεί σε διαφορετικούς υπολογιστές του συμπλέγματος. Μπορούν να κλιμακωθούν οριζόντια, δημιουργώντας αντίγραφα (replicas) για την διαχείριση αυξημένης εισερχόμενης κίνησης.

Deployments

Τα Deployment χειρίζονται την επιθυμητή τελική κατάσταση μιας εφαρμογής, καθορίζοντας το πλήθος αντιγράφων, και την ομαλή μετάβαση σε επόμενες ή προηγούμενες εκδόσεις ενός λογισμικού. Αυτό επιτυγχάνεται με την σταδιακή αντικατάσταση υπαρχόντων pods με καινούρια σε περίπτωση απαίτησης αναβάθμισης, καθώς και με τη διατήρηση μιας N μήκους ιστορίας προηγούμενων καταστάσεων που μπορεί να χρησιμοποιηθεί για την επιστροφή σε προηγούμενη, λειτουργική κατάσταση σε περίπτωση προβλήματος.

Persistent Volumes

Τα Volumes, αντίστοιχα με τα Docker Volumes, αποτελούν τη λύση για μη εφήμερο αποθηκευτικό χώρο για τα Pods. Είναι μια αφαίρεση πάνω από διάφορα πραγματικά συστήματα αποθήκευσης, όπως τοπικούς δίσκους, ή Δικτυακά Συστήματα Αρχείων (NFS). Τα volume είναι απαραίτητα για πληθώρα εφαρμογών, όμως η δυσκολίες που έρχονται μ'αυτά, συχνά οδηγούν τους σχεδιαστές εφαρμογών να επιλέγουν να σχεδιάζουν τις εφαρμογές όσο το δυνατόν πιό εφήμερες (stateless).

Services

Services αποκαλούνται οι δομικές μονάδες υπεύθυνσες για την έκθεση των Pods, επιτρέποντας την επικοινωνία μεταξύ των, αλλά και εκτός του υπολογιστικού συμπλέγματος. Ένα service μπορεί να εκθέτει διαφορετικά Pods, για παράδειγμα αντίγραφα μια εφαρμογής, με αποτέλεσμα να εξασφαλίζεται η διαθεσιμότητα μιας εφαρμογής ακόμη κι αν υπάρχει πρόβλημα με ένα ή μερικά από αυτά. Τα Services έρχονται σε πολλές 'γεύσεις' όπως ClusterIP, NodePort, ή LoadBalancer, και σε συνδυασμό με τα Deployment, είναι αυτά που εξασφαλίζουν όσο το δυνατόν μεγαλύτερη διαθεσιμότητα των εφαρμογών.

Ingresses

Τέλος τα Ingresses είναι HTTP(S) load balancers, που κατευθύνουν κίνηση που έρχεται από εκτός του υπολογιστικού συμπλέγματος σε Services εντός αυτού, με βάση το path ή το host απ'όπου έρχεται η κίνηση. Συνήθως χειρίζονται την υλοποίηση του SSL, και είναι το τελευταίο κομμάτι του παζλ, που απαιτείται για την διάθεση μιας εφαρμογής από ένα υπολογιστικό σύμπλεγμα, στον έξω κόσμο.

2.2.1 Helm

Η βασική μονάδα λειτουργίας του Helm είναι το Helm chart, δηλαδή ένα σύνολο από πόρους Kubernetes (Pods, Ingresses, Services κλπ.) με προκαθορισμένη δομή και κυριότερα, προκαθορισμένο το ποιά κομμάτια τους διατίθενται προς παραμετροποίηση.

```
chart-name
├─ values.yaml
├─ templates
│  ├─ ...
│  ├─ template-a.yaml
│  └─ template-b.yaml
└─ Chart.yaml
```

Listing 2: *Τυπική δομή ενός helm chart*

Ένα αρχείο `values` χρησιμοποιείται για τον ορισμό των παραμέτρων που μπορούν να δεχτούν τα templates, που όπως είπαμε είναι απλοί πόροι Kubernetes. Το `Chart.yaml` αρχείο περιέχει μεταδεδομένα για το ίδιο το chart, όπως το όνομά του, η τρέχουσα έκδοση, και πιθανόν εξαρτήσεις σε άλλα chart. Το Helm και το cli του, μπορούν να χρησιμοποιηθούν για την εγκατάσταση, αναβάθμιση ή διαγραφή charts από ένα υπολογιστικό σύμπλεγμα, όπως και για τη δημιουργία νέων chart, ενώ είναι αυτό που χρησιμοποιεί και το ArgoCD για να 'φουσκώνει' (inflate) ένα chart πριν αυτό εφαρμοστεί στο υπολογιστικό σύμπλεγμα

2.3 GitOps

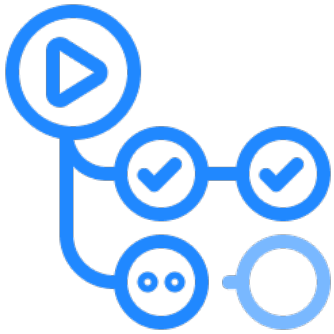
Έχοντας λοιπόν βρεθεί στον 'μαγικό' αυτό κόσμο, όπου η διάθεση λογισμικού είναι εύκολη, και η παραμετροποίησή του γίνεται με πλήρως δηλωτικό τρόπο, μπορούμε αρκετά ευκολότερα να αυτοματοποιήσουμε την διαδικασία ενημέρωση και αναβάθμισης εφαρμογών, δρέποντας ταυτόχρονα όλα τα οφέλη των εργαλείων version control. Ή τέλος πάντων αυτή είναι η ιδέα πίσω απ'τον όρο GitOps, ένα υποσύνολο του όρου DevOps.

Το GitOps βασίζεται πάνω σε εργαλεία όπως Kubernetes operators και συστήματα Continuous Delivery (CD), για να επιβλέπει αποθετήρια (repositories) κώδικα και να εφαρμόζει όποιες αλλαγές εντοπίζει σε πραγματικό χρόνο. Χρησιμοποιεί τα git commit ως ιστορικό αρχείο deployment, κάνοντας έτσι εύκολη τη διερεύνηση προβλημάτων, ειδικά σε μεγάλες ομάδες όπου τα μέλη αλλάζουν συχνά (όπως το CERN).

Continuous Integration

Με τον όρο Continuous Integration (CI) εννοείται η τεχνική που προσπαθεί να διασφαλίσει ότι αλλαγές και προσθήκες σε υπάρχον λογισμικό, ενσωματώνονται ομαλά με τις εξαρτήσεις και τους εξαρτημένους αυτού. Συνήθως αυτό γίνεται με τη χρήση pipelines, που υποστηρίζονται απ'τους περισσότερους παρόχους εργαλείων code-versioning, με πιό γνωστές μορφές το Github Actions και το GitlabCI.

Σχήμα 2.4: *Λογότυπο GitlabCI*



Σχήμα 2.5: *Λογότυπο Github Actions*

Continuous Deployment

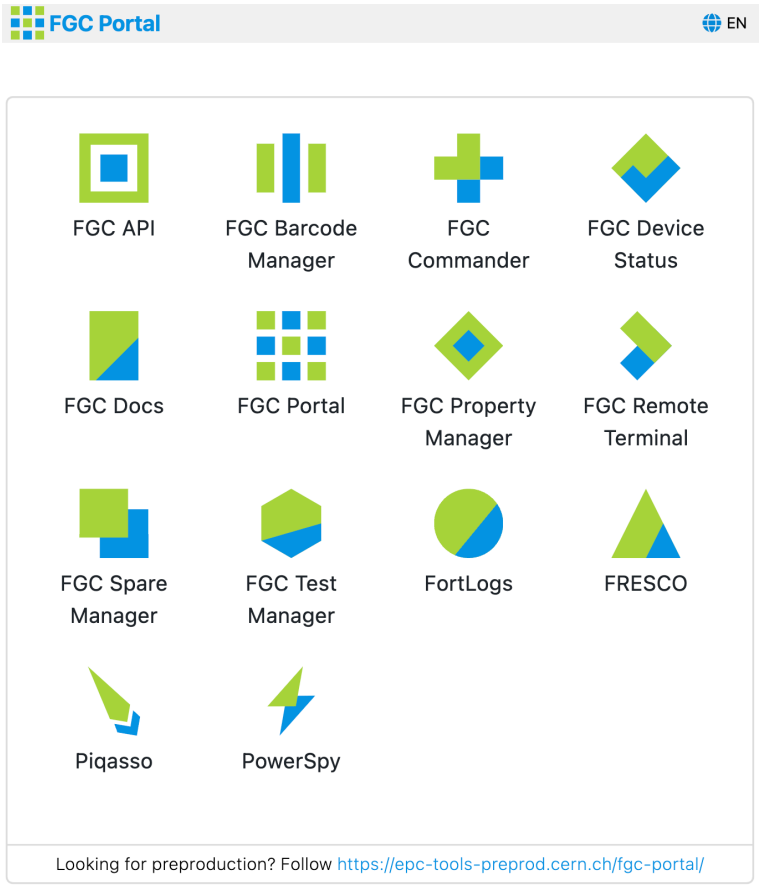
Το Continuous Delivery, βασίζεται πάνω στις διαβεβαιώσεις που παρέχει μια καλή υλοποίηση CI, και σκοπός της είναι η αυτοματοποίηση της διαδικασίας διάθεσης νέων εκδόσεων ενός λογισμικού. Προφανώς το να γίνεται εγκατάσταση κάθε αλλαγής ενός λογισμικού σε κάθε περιβάλλον που αυτό τρέχει δεν είναι πάντα επιθυμητό, οπότε χρησιμοποιούνται συμβάσεις και διαφορετικά περιβάλλοντα εγκατάστασης, ούτως ώστε να γίνεται μια σταδιακή ενσωμάτωση τον αλλαγών, με συχνά ονόματα για τα περιβάλλοντα αυτά να είναι QA, Dev, Preprod, Prod κλπ. περιβάλλοντα/στόχοι.

3 Σχεδίαση

Αυτό το κεφάλαιο δίνει τις λεπτομέρειες του σχεδιασμού των υπάρχοντων συστημάτων καθώς και των λύσεων που αναφέρθηκαν σε προηγούμενα κεφάλαια. Ξεκινάει με μια περιγραφή των υπάρχοντων συστημάτων, τονίζοντας τα σημεία που φαίνεται η ανάγκη για απομάκρυνση από αυτά, και κλείνει με μια ανάλυση της λειτουργίας των τμημάτων του παρόντος έργου

3.1 CCS Tools

Όπως αναφέραμε σε γενικές γραμμές και προηγουμένως, η Tools ομάδα του CCS τμήματος, ασχολείται με τη δημιουργία εύχρηστων εργαλείων χειρισμού των μετατροπέων ισχύος που παρέχει το EPC, είτε για την αρχική λειτουργία, την παραμετροποίηση, ή την διερεύνηση προβλημάτων τους. Αυτό το επιτυγχάνει αναπτύσσοντας ένα σύνολο εφαρμογών με τη βοήθεια ειδικών στους μετατροπείς ισχύος. Τα τελευταία χρόνια έχει συγκλίνει σ'ένα stack εφαρμογών που βασίζεται σε συνδυασμό FastAPI και VueJS, παρόλ'αυτά υπάρχουν μερικά παλαιότερα εργαλεία, που τρέχουν σε perl. Αυτός είναι άλλος ένας λόγος που η χρήση Container ήταν σχεδόν μονόδρομος, αφού αφαιρεί αρκετή απ'την πολυπλοκότητα που θα υπήρχε λόγω ανομοιογένειας των εφαρμογών.



Σχήμα 3.1: *Κατάλογος εφαρμογών που διατίθενται από την ομάδα*

3.2 Υπάρχον περιβάλλον διάθεσης

Το υπάρχον περιβάλλον είχε ήδη δει μερικές δεκαετίες χρήσης. Στη πράξη ήταν ένας απλός Apache server, και διεργασίες python-Systemd που έτρεχαν σε έναν Linux διακομιστή. Είχε φτιαχτεί απ'τους ίδιους τους ειδικούς που εργάζονταν πάνω στους μετατροπείς ισχύος, με σκοπό να αυτοματοποιήσουν μερικά κομμάτια της δουλειάς τους. Όσο οι ευθύνες της ομάδας επεκτείνονταν, παρά τις προσπάθειες που έγιναν για παραμονή στο ίδιο περιβάλλον, και τη βελτίωσή του, η πολυπλοκότητα του συστήματος αυξανόταν αρκετά, ειδικά όταν εμφανίστηκε ανάγκη για περισσότερα από ένα περιβάλλοντα διάθεσης εφαρμογών, και μάλιστα τη διάθεσή τους και εκτός CERN

Ειδικότερα, το σύστημα μπορούσε να χωριστεί σε μερικά μέρη:

- **Templates:** Κάθε εφαρμογή χρησιμοποιεί templates για να παραμετροποιήσει τους απαραίτητους πόρους της, συγκεκριμένα αρχεία διαμόρφωσης του Apache

vhost, αρχεία ορισμού υπηρεσιών systemd και απλά αρχεία `.env`, με placeholders για τις τιμές που πρέπει να προσαρμοστούν ανάλογα με τον στόχο διάθεσης.

```
    ...
# FGC API WebSockets
ProxyPass         "/fgc-api/api/devices/status-full"
↪  "ws://localhost:${FGC_API_PORT}/devices/status-full"
    ...
```

Listing 3: *αρχεία διαμόρφωσης Apache vhost*

```
    ...
[Service]
# Service definition
User=${REMOTE_DEPLOY_USER}
WorkingDirectory=${APP_DIR}/${FGC_API_APP_NAME}
    ...
```

Listing 4: *αρχεία ορισμού υπηρεσιών systemd*

```
    ...
STATUS_FILE=${STATUS_FILE}
FGC_DB_DRIVER=${FGC_DB_DRIVER}
FGC_DB_HOST=${FGC_DB_HOST}
FGC_DB_PORT=${FGC_DB_PORT}
    ...
```

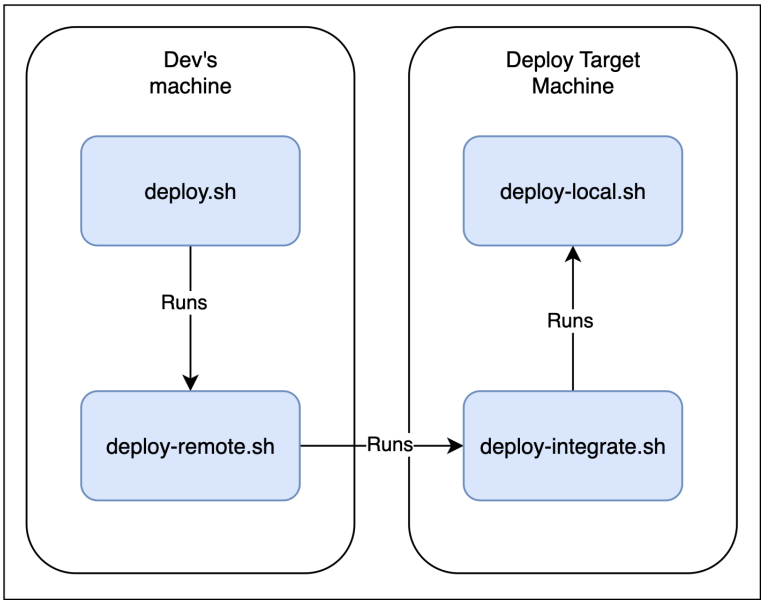
Listing 5: *Αρχείο μεταβλητών περιβάλλοντος*

- **Parameters:** Αυτά τα αρχεία περιέχουν όλες τις μεταβλητές περιβάλλοντος που χρειάζονται για την παραμετροποίηση της διάθεσης μιας εφαρμογής, καθώς και της ίδιας της εφαρμογής και των κομματιών απ'τα οποία αποτελείται.

```
        ...
DEV_MODE="true"
DEFAULT_APP="fgc-portal"
ENVIRONMENT_FULLNAME="development"
        ...
```

Listing 6: *Παράδειγμα αρχείου παραμέτρων*

- **Scripts:** Το τελευταίο κομμάτι είναι ένα σύνολο από script φλοιού που αυτοματοποιούν μερικά κομμάτια της διαδικασίας διάθεσης μιας εφαρμογής. Υποκαθιστούν τα templates, συμπιέζουν τους απαραίτητους πόρους, τους αντιγράφουν στον στόχο διάθεσης και επανεκκινούν τις απαραίτητες υπηρεσίες.



Σχήμα 3.2: *Προηγούμενη ροή διάθεσης εφαρμογών*

Γνωστά προβλήματα

Η διαδικασία προετοιμασίας ενός νέου στόχου διάθεσης εφαρμογών, που ήταν μια επαναλαμβανόμενη ανάγκη, καθώς το περιβάλλον ανάπτυξης κάθε νέου μέλους έπρεπε να είναι ένα αντίγραφο του κανονικού (production) περιβάλλοντος διάθεσης εφαρμογών, ήταν πολύ επιρρεπής σε σφάλματα. Νέες Εικονικές Μηχανές έπρεπε να παρέχονται μέσω αίτησης στο IT, πάνω στις οποίες έπρεπε να εγκατασταθούν οι απαραίτητες εξαρτήσεις συστήματος, κάτι που συχνά οδηγούσε σε προβλήματα λόγω αλλαγών στα αποθετήρια λογισμικού του CERN και στις εκδόσεις του λειτουργικού συστήματος που

παρέχονταν από το CERN. Συνολικά, αυτό καθυστερούσε την ένταξη νέων μελών στην ομάδα κατά περίπου μία έως δύο εβδομάδες, κάτι που είναι ένα μεγάλο πρόβλημα δεδομένης της υψηλής εναλλαγής προσωπικού στο CERN, και της ύπαρξης συμβολαίων που διαρκούν μόνο για λίγους μήνες.

Αλλαγές ή προσθήκες δυνατοτήτων στη διαδικασία διάθεσης ήταν πολύ χρονοβόρες και δύσκολες, λόγω της διαδικαστικής (procedural) φύσεως των script και της υποδομής. Ένα καλό παράδειγμα αυτού ήταν όταν η ομάδα χρειάστηκε να παρακάμψει συγκεκριμένες μεταβλητές περιβάλλοντος για συγκεκριμένες εφαρμογές όταν διατίθονταν σε κάποιον στόχο. Για να το κάνουμε αυτό με μη παρεμβατικό τρόπο, έπρεπε να παρακάμψουμε τη λογική υποκατάστασης μεταβλητών περιβάλλοντος του script, και να προσθέσουμε ένα άλλο επίπεδο γραμμένο σε python, καθώς το bash δεν μπορούσε να ρυθμιστεί εύκολα για το σκοπό αυτό. Ακόμα και μετά από προσεκτική επισκόπηση κάθε λειτουργικότητας, εξακολουθήσαμε να συναντάμε προβλήματα σε περιπτώσεις που δεν είχαμε προβλέψει, για τους επόμενους μήνες, κάτι που προκαλούσε καθυστερήσεις και πίεση στην ομάδα.

```
{
    "accwww.cern.ch": {
        "fgc-property-manager/server": {
            "FGC_DB_DRIVER": "oracle"
        }
    },
        // ...
```

Listing 7: *Παράδειγμα αρχείου παράκαμψης μεταβλητών περιβάλλοντος*

3.3 Προτεινόμενες λύσεις

Όπως περιγράφηκε συνοπτικά, οι προτεινόμενες λύσεις ήταν μια σύγχρονη προσέγγιση βασισμένη σε containers, παράλληλα με το kubernetes και τις μεθοδολογίες CI/CD. Μπορεί να χωριστεί σε μερικά μέρη που θα περιγραφούν στις επόμενες ενότητες.

3.3.1 Τοπική ανάπτυξη λογισμικού

Με την ομάδα να μετακινείται προς το Docker και τα containers, έγινε δυνατό να υποστηριχθεί η ανάπτυξη τοπικά στις μηχανές των προγραμματιστών, αποφεύγοντας την ανάγκη για μια νέα εικονική μηχανή για κάθε προγραμματιστή. Οι κύριες απαιτήσεις ήταν:

- **Απλότητα:** Η υποδομή πρέπει να'ναι όσο το δυνατόν απλούστερη και ευκολότερη στη χρήση, ειδικά όσον αφορά την εγκατάσταση και τη χρήση της.
- **Επεκτασιμότητα:** Πρέπει να είναι εύκολο να προστεθούν νέες εφαρμογές και να επεκταθεί το σύνολο δυνατοτήτων που παρέχεται σε αυτές τις εφαρμογές από την υποδομή.
- **Απόδοση:** Η υποδομή πρέπει να είναι γρήγορη. Γρήγορη στην εγκατάσταση, γρήγορη στην έναρξη της ανάπτυξης, και να έχει ελάχιστη επίπτωση στην απόδοση κατά την ανάπτυξη.

Και στην πλευρά των εφαρμογών

- **Πρόσβαση στο Τεχνικό Δίκτυο:** Οι εφαρμογές πρέπει να μπορούν να έχουν πρόσβαση σε άλλες υπηρεσίες και πόρους στο δίκτυο του CERN, όπως βάσεις δεδομένων, άλλα APIs, κ.λπ.
- **Πρόσβαση στα αποθετήρια λογισμικού του CERN:** Οι εφαρμογές πρέπει να μπορούν να έχουν πρόσβαση στα αποθετήρια λογισμικού του CERN, για να εγκαταστήσουν εξαρτήσεις και πακέτα, καθώς και να αποκτήσουν βασικές εικόνες docker που είναι ειδικές για το CERN.
- **Πρόσβαση στα λειτουργικά αρχεία:** Οι εφαρμογές πρέπει να μπορούν να έχουν πρόσβαση στα λειτουργικά αρχεία της ομάδας ccs, που παράγονται από άλλες υπηρεσίες ή προσαρμόζονται χειροκίνητα από ειδικούς, και φιλοξενούνται στο NFS του CERN.

Εδώ έρχεται το Docker Compose. Η ιδέα ήταν να χρησιμοποιηθεί το docker compose για να αφαιρεθεί όσο το δυνατόν περισσότερη υλοποίηση και συντήρηση από την ομάδα, ενώ ταυτόχρονα να πληρούνται οι απαιτήσεις.

Για να επιτευχούν οι απαιτήσεις από την πλευρά των εφαρμογών, θα χρειαζόταν δύο ξεχωριστοί μηχανισμοί proxy, καθώς η διαδικασία δημιουργίας εικόνων απαιτεί πρόσβαση στο Γενικό Δημόσιο Δίκτυο (GPN) του CERN και πιο συγκεκριμένα στο εσωτερικό αποθετήριο πακέτων Python του CERN που φιλοξενείται στο GPN, ενώ οι ίδιες οι εφαρμογές - όπως αναφέρθηκε - χρειάζονται πρόσβαση στο Τεχνικό Δίκτυο (TN) του CERN.

Δικτυακός διαμεσολαβητής για την κατασκευή των εικόνων

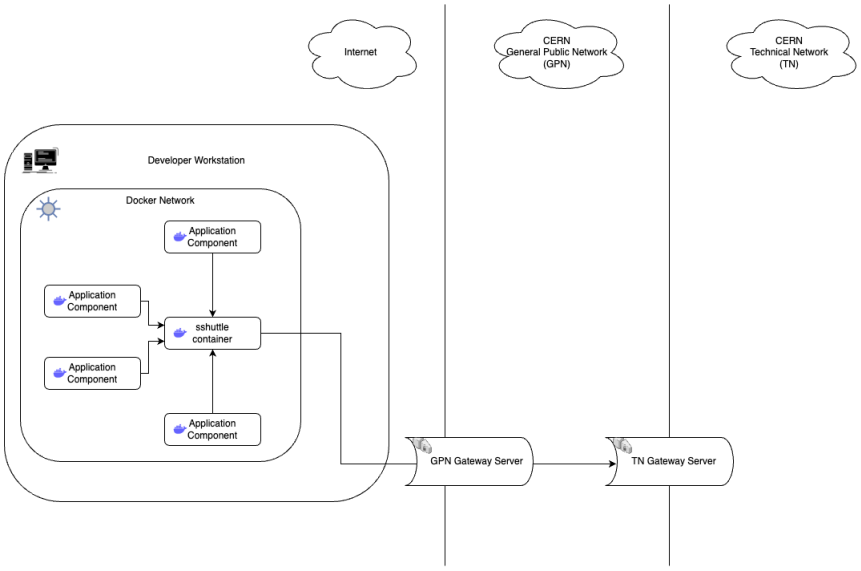
Για τη διαδικασία κατασκευής εικόνων, δεν μπορούσαμε (εύκολα) να προσαρμόσουμε τη διαμόρφωση δικτύωσης του κατασκευαστή docker, καθώς θα απαιτούσε τη χρήση ενός προσαρμοσμένου κατασκευαστή (docker builder), περιπλέκοντας τη λογική του setup και το κόστος συντήρησης. Υπήρχε επίσης η ιδέα της διαμόρφωσης της προώθησης μέσω διαμεσολαβητή (proxy) σε επίπεδο daemon του docker, αλλά αυτό σήμαινε ότι τα βοηθητικά προγράμματα του setup θα απαιτούσαν δικαιώματα root, και θα προσέθετε την πολυπλοκότητα της ενεργής παρακολούθησης για τυχόν εικόνες που πρέπει να κατασκευαστούν, ώστε να επανεκκινείται ο daemon με τις σωστές ρυθμίσεις διαμεσολαβητή, να τις κατασκευάζει, και στη συνέχεια να επανακκινείται με τις αρχικές ρυθμίσεις, έτσι ώστε η λειτουργικότητα του daemon να μην εξαρτάται από την ύπαρξη της σύνδεσης διαμεσολαβητή.

Ευτυχώς μπορούσαμε να εκμεταλλευτούμε το γεγονός ότι η `python` και κατά επέκταση το `pip` μπορούν να ρυθμιστούν να χρησιμοποιούν έναν διακομιστή μεσολάβησης, απλά ορίζοντας τις μεταβλητές περιβάλλοντος `HTTP_PROXY`, `HTTPS_PROXY` και `ALL_PROXY`, και το σημαντικό είναι ότι αυτές μπορούν να λειτουργήσουν με διακομιστές μεσολάβησης `socks5`, οι οποίοι είναι εύκολο να ρυθμιστούν αν έχει κανείς πρόσβαση ssh σε μια μηχανή στο δίκτυο στόχο. Αυτό σήμαινε ότι μπορούσαμε να ρυθμίσουμε έναν προσωρινό διακομιστή μεσολάβησης socks5 σε μια μηχανή στο δίκτυο GPN, να ορίσουμε αυτή τη μεταβλητή και να την εξάγουμε σε όλο το σύστημα, έτσι ώστε να είναι προσβάσιμη από τη διεργασία `pip` μέσα στον κατασκευαστή docker, και στη συνέχεια να την καταργήσουμε μετά την ολοκλήρωση της διαδικασίας κατασκευής.

Δικτυακός διαμεσολαβητής για τις εφαρμογές

Ο διαμεσολαβητής για τις εφαρμογές δεν μπορούσε να ακολουθήσει τον ίδιο σχεδιασμό, καθώς οι διεργασίες που κάνουν δικτυακή κίνηση χρησιμοποιούν ποικιλία γλωσσώ και συστημάτων, και δεν προσφέρουν όλες την ίδια υποστήριξη για διαμεσολάβηση όπως το `pip`, ενώ η υποστήριξη διαμεσολάβησης του openssh μπορεί να είναι αργή. Χρειαζόταν μια πιο συστηματική και ανθεκτική προσέγγιση, για να προσομοιωθεί το περιβάλλον δικτύωσης της εφαρμογής με έναν αδιαφανή τρόπο. Η εσωτερική τεκμηρίωση του CERN αναφέρει το `sshuttle` ως ένα εργαλείο που μπορεί να είναι χρήσιμο για την υποστήριξη της τηλεργασίας, αλλά δεν παρέχει πολλές πληροφορίες.

Το `sshuttle` παρέχει τις δυνατότητες ενός τυπικού VPN, απαιτώντας μόνο πρόσβαση ssh σε μια μηχανή στο δίκτυο στόχο. Συγκεντρώνει την κίνηση TCP και στις δύο πλευρές, διακομιστή και πελάτη, συναρμολογώντας και αποσυναρμολογώντας τα πακέτα, αποφεύγοντας το πρόβλημα του TCP-over-TCP, και ενεργώντας μόνο στο επίπεδο της εφαρμογής. Απαιτεί δικαιώματα root, αλλά αυτό είναι ένα πρόβλημα που μπορεί να ξεπεραστεί τρέχοντας το μέσα σε ένα κοντέινερ docker. Το τελικό αποτέλεσμα είναι ένα κοντέινερ που λειτουργεί ως δρομολογητής για όλα τα άλλα κοντέινερ, που ξεκινά και σταματά εύκολα παράλληλα με το υπόλοιπο setup, και δεν απαιτεί ανυψωμένη πρόσβαση στη μηχανή ανάπτυξης.



Σχήμα 3.3: *Ανάπτυξη εφαρμογών από το εξωτερικό του δικτύου του CERN*

3.3.2 Kubernetes

Στη διαδικασία σχεδιασμού εδώ, είχαμε την τύχη να έχουν ληφθεί ήδη μερικές απ'τις σχεδιαστικές επιλογές σχετικά με το CI/CD, συγκεκριμένα τη χρήση του ArgoCD. Αυτό περιόρισε τις διαθέσιμες επιλογές, καθώς το ArgoCD μας καθοδηγεί σε κάποιες κατευθύνσεις, συμπεριλαμβανομένης της χρήσης του Helm. Η κύρια μονάδα ανάπτυξης του ArgoCD είναι το Application, ένας προσαρμοσμένος πόρος Kubernetes που χρησιμοποιείται για να δηλώσει λεπτομέρειες σχετικά με την επιθυμητή κατάσταση μιας εφαρμογής, όπως η πολιτική συγχρονισμού ή η τοποθεσία και ο τύπος των πόρων kubernetes της εφαρμογής.

Η κοινότητα του ArgoCD έχει καταλήξει σε δύο βασικούς τρόπους για την ανάπτυξη εφαρμογών, το app-of-apps pattern και το ApplicationSet pattern.

ApplicationSet

Τα ApplicationSets είναι ένας όχι έντονα παραμετροποιήσιμος τρόπος διάθεσης πολλαπλών εφαρμογών, οι οποίες μοιράζονται μια παρόμοια διαμόρφωση, αλλά έχουν διαφορετικές τιμές για μερικές από τις παραμέτρους τους. Χρησιμοποιεί διάφορους τύπους "γεννητριών" για να κάνει το templating, ένα παράδειγμα των οποίων είναι ο γεννήτορας `git`, και οι υποτύποι του οι γεννήτριες `git`-dir και `git`-file, οι οποίες χρησιμοποιούν τη δομή καταλόγου και τα ονόματα αρχείων αντίστοιχα ως μεταβλητές για το πρότυπο. Είναι ένα ισχυρό εργαλείο, ειδικά για περιπτώσεις χρήσης όπου μια ενιαία ArgoCD instance διαχειρίζεται διαφορετικά clusters, καθώς μπορεί εύκολα να διευκολύνει τη διάθεση της ίδιας εφαρμογής σε πολλαπλά clusters, με διαφορετικές διαμορφώσεις. Ωστόσο στην περίπτωση μας κάθε cluster έπρεπε να έχει το δικό του ArgoCD instance, για να εξασφαλιστεί ότι τα clusters είναι εντελώς απομονωμένα μεταξύ τους, κάτι που ήταν απαραίτητο λόγω του στόχου της ομάδας να παρέχει τις εφαρμογές και σε εξωτερικά εργαστήρια και συνεργάτες. Ένα άλλο πρόβλημα είναι ότι με αυτό το μοτίβο, οι εφαρμογές πρέπει να δηλώνονται κάτω από το ίδιο chart, το οποίο στη συνέχεια ανάλογα με τις παραμέτρους του, ενεργοποιεί ή απενεργοποιεί κάθε εφαρμογή, αυξάνοντας έτσι την εξάρτηση μεταξύ των διαφόρων εφαρμογών.

App-of-apps

Το App-of-Apps μοτίβο, χρησιμοποιεί ένα Application resource - σημείο εισόδου - για να κάνει τα υπόλοιπα διαθέσιμα. Είναι αρκετά περισσότοερ παραμετροποιήσιμο, καθώς τα υπόλοιπα Application μπορούν να χειρίζονται ένα κανονικό Helm chart, το οποίο μπορεί εύκολα να προσαρμοστεί για να αναπτυχθεί σε περιβάλλοντα Kubernetes.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
    ...
spec:
    source:
        repoURL: <git-repo-url>
        path: <chart-containing-all-apps-as-resources>
    ...
    helm:
    ...
```

Listing 8: *Παράδειγμα πόρου τύπου Application του ArgoCD*

Παράλληλα με μια σωστά διαμορφωμένη ArgoCD instance, μπορεί να φέρει τις εφαρμογές που διατίθενται σ'ένα υπολογιστικό σύμπλεγμα Kubernetes, από την παρούσα στην επιθυμητή κατάσταση, διατηρώντας μια μοναδική 'πηγή αλήθειας' για την κατάσταση αυτή. Είναι επίσης δυνατό να χρησιμοποιηθεί το ίδιο αποθετήριο για τον κώδικα και τη διαμόρφωση του υπολογιστικού συμπλέγματος, κάτι που είναι ένα μεγάλο πλεονέκτημα για ομάδες που χρησιμοποιούν ένα monorepo. Αυτοί είναι οι κύριοι λόγοι για τους οποίους επιλέξαμε αυτό τον τρόπο σχεδιασμού στο τέλος.

Διαχείριση μυστικών

Μια σημαντική πλευρά κάθε διάθεσης είναι η διαχείριση των μυστικών. Το Kubernetes έχει ένα ενσωματωμένο σύστημα διαχείρισης μυστικών, που μπορεί να χρησιμοποιηθεί για την αποθήκευση και διαχείριση ευαίσθητων πληροφοριών, όπως κωδικούς πρόσβασης, OAuth tokens και ssh κλειδιά, αλλά η πολυπλοκότητα έγκειται στο να φτάσουν τα μυστικά αυτά μέχρι εκεί. Ένα χρήσιμο εργαλείο ήταν το `sops`, το οποίο μπορεί να χρησιμοποιηθεί για την κρυπτογράφηση και αποκρυπτογράφηση αρχείων, και συγκε-

κριμένα για την κρυπτογράφηση τιμών μυστικών, πριν την αποθήκευσή τους σε ένα git αποθετήριο. Έτσι, μπορεί να διατηρείται ένα ιστορικό για τις τιμές τους (κρυπτογραφημένες) και τις αλλαγές τους, χωρίς να εκτίθεται η ευαίσθητη πληροφορία, εφόσον γίνεται σωστή διαχείριση του κλειδιού. Υπάρχει ένα χρήσιμο εργαλείο για την εργασία με τα clusters και το `sops`, το `helm-secrets`, το οποίο μπορεί να ενσωματωθεί με το ArgoCD μαζί με το `helm`, για τη διαχείριση των μυστικών με πιο φιλικό προς τον χρήστη τρόπο [jkr].

3.3.3 CI/CD & Αυτοματοποίηση

Σε αυτό το σημείο ήμασταν κλειδωμένοι για τις περισσότερες από τις σχεδιαστικές αποφάσεις που σχετίζονται με το CI/CD και την αυτοματοποίηση. Σε γενικές γραμμές, αυτό που χρειάζεται είναι:

- **Αυτόματη δοκιμή του λογισμικού**
- **Αυτόματη κατασκευή των εικόνων**
- **Αυτόματη διάθεση στα περιβάλλοντα διάθεσης**

Τα πρώτα δύο είναι ευθύνη των αυτοματοποιημένων pipelines, και είναι αυτοτελή στην πλευρά του αποθετηρίου gitlab. Το τελευταίο είναι λίγο πιο περίπλοκο, καθώς απαιτεί καταρχήν κάποια μορφή επικοινωνίας (συγχρονισμένη ή μη) μεταξύ των pipelines και του υπολογιστικού συμπλέγματος.

Αυτόματη δοκιμή & κατασκευή

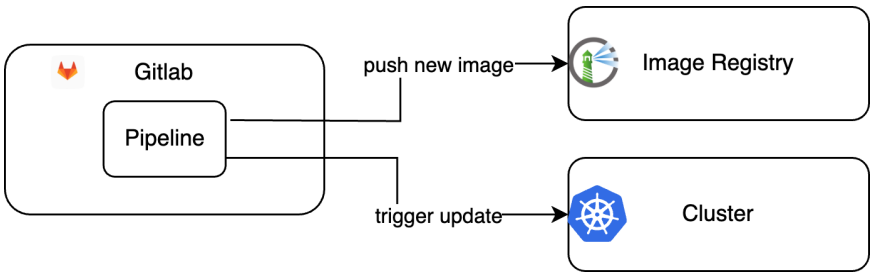
Το CERN επιβάλει τη χρήση του εσωτερικού του Gitlab ως σύστημα ελέγχου εκδόσεων για όλα τα έργα. Επομένως, ήταν μια φυσική επιλογή να χρησιμοποιηθεί το GitlabCI για το CI/CD pipeline. Το GitlabCI είναι ένα πολύ ισχυρό εργαλείο, που μπορεί να χρησιμοποιηθεί για την αυτοματοποίηση των δοκιμών και της κατασκευής εφαρμογών, καθώς και της διάθεσής τους με λίγο περισσότερη προσπάθεια. Τα pipelines του GitlabCI ορίζονται χρησιμοποιώντας ένα αρχείο `.gitlab-ci.yml`, το οποίο ερμηνεύεται στον διακομιστή gitlab, ο οποίος με τη σειρά του ορίζει και εκτελεί εργασίες σε gitlab runners. Οι gitlab runners είναι μηχανές που έχουν ρυθμιστεί κατάλληλα για να

εκτελούν τις εργασίες, και μπορούν να είναι είτε κοινόχρηστοι runners, που παρέχονται από το CERN, είτε ιδιωτικοί runners, που μπορούν να ρυθμιστούν από την ομάδα. Στην περίπτωσή μας, το CERN παρέχει δύο σχετικούς runners, έναν προρυθμισμένο με Docker και έναν με Kaniko, μια εναλλακτική του Docker που δεν απαιτεί δικαιώματα root. Αρχικά ο σχεδιασμός ήταν να χρησιμοποιηθεί ο runner Docker, αλλά για λόγους ασφαλείας, η υλοποίησή απαιτούσε τη δημιουργία ενός νέου VM για κάθε εκτέλεση, κάτι που ήταν προβληματικά αργό, και η ομάδα άλλαξε σε Kaniko runner, και οι δύο runners φαίνονται παρόμοιοι στον τελικό χρήστη.

Μετά από αυτό ήταν αρκετά απλό. Ένα σύνολο εργασιών (jobs) θα δημιουργούνταν ανά εφαρμογή, που θα ενεργοποιούνταν όποτε γίνονταν αλλαγές στην εφαρμογή, και θα εκτελούσαν δοκιμές, κατασκευή και (προαιρετικά) διάθεση.

Αυτόματη διάθεση

Μια πρώτη ιδέα ήταν να γίνεται απευθείας ενημέρωση του υπολογιστικού συμπλέγματος από τα pipelines. Αυτό θα μπορούσε να γίνει δημιουργώντας ένα service account με τα απαραίτητα δικαιώματα στο cluster, και χρησιμοποιώντας αυτό από τα pipelines για να ενημερώνει συγκεκριμένους πόρους Kubernetes. Ωστόσο, αυτό παραβιάζει τη δηλωτική φύση αυτού που επιδιώκουμε, καθώς δεν θα υπήρχε καμία μοναδική πηγή αλήθειας εκτός από το ίδιο το cluster, και στην περίπτωση του ArgoCD, το argo απλά επανασυγχρονίζει οποιεσδήποτε χειροκίνητες αλλαγές πίσω στην κατάσταση που καθορίζεται στο αποθετήριο.

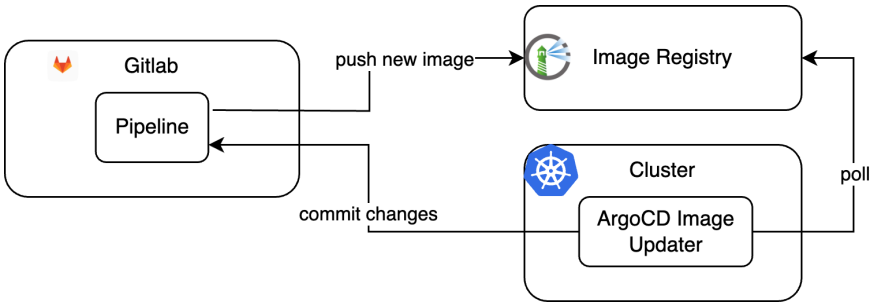


Σχήμα 3.4: *Από Pipeline στο υπολογιστικό σύμπλεγμα*

Η δεύτερη μας ιδέα, την οποία εξετάσαμε διεξοδικά, ήταν η χρήση του εργαλείου του Argo που έχει δημιουργηθεί ειδικά γι'αυτό, το `ArgoCD Image Updater`. Η ομάδα αρχικά εξέφρασε αμφιβολίες λόγω του εργαλείου να βρίσκεται υπό ενεργή ανάπτυξη, αλλά

παρόλα αυτά το βάρος συντήρησης μιας προσαρμοσμένης λύσης θεωρήθηκε μεγαλύτερο κόστος. Αυτό τελικά το πληρώσαμε στο τέλος, καθώς υπήρχαν μερικά προβλήματα που ήταν ανυπέρβλητα, και καταλήξαμε να πάμε με μια προσαρμοσμένη λύση ούτως ή άλλως.

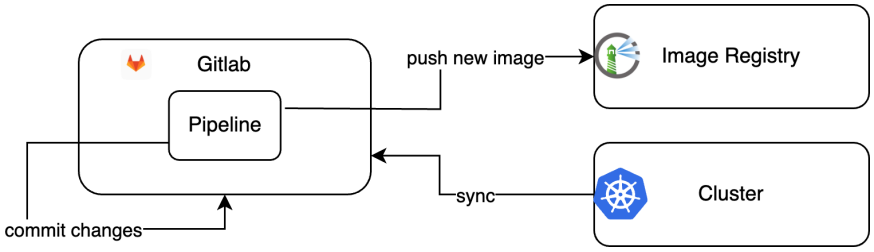
Το ArgoCD Image Updater είναι ένα επιπλέον φορτίο εργασίας που εκτελείται στο υπολογιστικό σύμπλεγμα, ελέγχοντας περιοδικά ένα αποθετήριο εικόνων για νέες εκδόσεις εικόνων που ακολουθούν προαιρετικά ένα μοτίβο που έχει καθορίσει ο χρήστης. Όταν βρεθεί μια νέα έκδοση, αντιγράφει κάποιο προκαθορισμένο git αποθετήριο, ενημερώνει ένα αρχείο με παρακάμψεις για τις εκδόσεις των εικόνων, κάνει commit και σπρώχνει τις αλλαγές, διασφαλίζοντας έτσι ότι η μοναδική πηγή αλήθειας είναι σε ένα αποθετήριο git. Αποφεύγει την ανάγκη για νέα διαπιστευτήρια ή δικαιώματα, επαναχρησιμοποιώντας τα διαπιστευτήρια που έχει ήδη το ArgoCD για να έχει πρόσβαση στο αποθετήριο git. Αλλά όπως ήδη αναφέρθηκε, λόγω των περιορισμών του, οι οποίοι θα συζητηθούν λεπτομερώς στο επόμενο κεφάλαιο, επιλέξαμε να προχωρήσουμε με μια προσαρμοσμένη λύση.



Σχήμα 3.5: *ArgoCD Image Updater*

Η 'χειροποίητη' λύση λειτουργούσε με παρόμοιο τρόπο, αλλά είχε το πρόσθετο πλεονέκτημα ότι δεν χρειάζεται να γίνεται polling, καθώς μεταφέρει την ενεργοποίηση της ενημέρωσης στην πλευρά του pipeline, επιτρέποντας έτσι συγχρονισμένες ενημερώσεις. Συνειδητοποιήσαμε ότι θα μπορούσαμε να χρησιμοποιήσουμε τη βασική λειτουργία του ArgoCD, δηλαδή την αυτόματη επαναφορά της τρέχουσας στην επιθυμητή κατάσταση για να επιτύχουμε το ίδιο αποτέλεσμα.

Τα απαιτούμενα από τον τελικό χρήστη ήταν να υποστηρίζονται χειροκίνητες ενημερώσεις, καθώς και αυτόματη παρακολούθηση της τελευταίας έκδοσης μιας εφαρμογής, με φιλικό προς τον χρήστη τρόπο. Για να το πετύχουμε αυτό, θα χρειαζόμασταν κάτι σαν αυτό:



Σχήμα 3.6: *Προσαρμοσμένη ροή CI/CD*

όπου ο συγχρονισμός γίνεται από το ArgoCD. Θα πρέπει να είναι δυνατή η ενεργοποίηση του pipeline αυτόματα σε αλλαγές κώδικα ή χειροκίνητα με την αποστολή μιας συγκεκριμένης ετικέτας git, ή τη δημιουργία ενός Merge Request. Το pipeline θα κατασκευάζει την εικόνα, θα την σπρώχνει στο registry, και στη συνέχεια θα ενημερώνει τον πόρο Application του ArgoCD, ο οποίος θα ενεργοποιήσει τον ελεγκτή του ArgoCD να συγχρονίσει την κατάσταση του cluster με την επιθυμητή κατάσταση, ενημερώνοντας έτσι την εφαρμογή.

4 Υλοποίηση

Σ'αυτό το κεφάλαιο θα εμβαθύνουμε στην πιο τεχνική πλευρά του σχεδιασμού που περιγράφηκε προηγουμένως. Διαχωρίζει την υλοποίηση σε όσο το δυνατόν πιο διακριτά συστατικά και αναλύει ποιες δυσκολίες έπρεπε να ξεπεραστούν. Τελειώνει δίνοντας μια επισκόπηση ολόκληρης της εργασίας σε δράση. Όπως και πριν, τα κύρια συστατικά είναι:

- The **Περιβάλλον τοπικής ανάπτυξης**.
- The **Διάθεση μέσω υπολογιστικού συμπλέγματος**.
- The **Αυτοματοποιήσεις στην ανάπτυξη και διάθεση των εφαρμογών**.

4.1 Τοπική ανάπτυξη

Μετά την δημιουργία των κοντέινερ για όλες τις εφαρμογές και υπηρεσίες, δημιουργήθηκε ένα αρχείο `docker-compose.yml`, το οποίο αργότερα επεκτάθηκε με ένα `docker-compose.telework` αρχείο που μπορούσε να χρησιμοποιηθεί προαιρετικά για να προσθέσει δυνατότητες τηλεργασίας στην εγκατάσταση. Λόγω περιορισμών που σχετίζονται με τη σειρά εκκίνησης του docker compose και το proxy που είναι απαραίτητο για τη διαδικασία κατασκευής που αναφέρθηκε προηγουμένως, δημιουργήθηκε ένα περιτύλιγμα script για να αφαιρέσει μερικά βήματα και τις εντολές `docker compose`.

Η διαδικασία είναι σχετικά απλή. Ο χρήστης καλεί το script με τα ονόματα των εφαρμογών που θέλει να ξεκινήσει, και το script αναλαμβάνει τα υπόλοιπα.

```
./script/start-local-development.sh fgc-commander-client fgc-portal
```

Listing 9: *Παράδειγμα κλήσης του περιτυλίγματος script για την τοπική ανάπτυξη*

Σε επισκόπηση - περισσότερες λεπτομέρειες θα δοθούν όπου είναι απαραίτητο - το script:

1. εξασφαλίζει ότι υπάρχει ένα external docker network, ούτως ώστε διαδοχικές επικλήσεις της εντολής `docker compose` να μπορούν να δημιουργήσουν κοντέινερ που υπάρχουν στο ίδιο δίκτυο εύκολα, και να αποφευχθούν συγκρούσεις με τα δίκτυα του CERN. Επίσης εξασφαλίζει την ύπαρξη ενός docker volume για τη βάση δεδομένων CCS, ώστε να διατηρούνται οι αλλαγές των προγραμματιστών στα δεδομένα δοκιμών.

```
# Create a docker network, if it does not exist already,
# with an explicit subnet to prevent conflicts with CERN addresses.
if [[ -z $(docker network ls -f name=${DOCKER_NETWORK} --format '{{
↪  .Name }}') ]]; then
echo '==> Creating a docker network...'
docker network create \
    --subnet '192.168.50.0/24' \
    $DOCKER_NETWORK
fi

# Create external volume for the CCS DB if does not exist.
CCS_DB_VOLUME='ccs-db-data'
if [[ -z $(docker volume ls --filter "name=${CCS_DB_VOLUME}" --format
↪  "{{.Name}}") ]]; then
echo '==> Creating a docker volume for the local CCS DB instance...'
docker volume create ${CCS_DB_VOLUME}
fi
```

Listing 10: *Εξασφάλιση ύπαρξης docker δικτύου και volume*

2. κατεβάζει εκ των προτέρων εικόνες Container απ'το αποθετήριο εικόνων του CERN.

```
# Prepull images proxying from inside the CERN network
source $LOCAL_DEVELOPMENT_DIR/docker-in-docker/utils.sh
# Finds all 'FROM ${PREPULL_PROJECT}' lines in all Dockerfiles in
↪  the ./app directory, skipping the node_modules subdirectories,
# strips them down to only the image name, and prepulls them.
PREPULL_IMAGES=$(find ./app -type d -name node_modules -prune -o
↪  -type f -name '*Dockerfile*' -print0 | \
    xargs -0 grep --no-filename "FROM ${PREPULL_PROJECT}" | \
    cut -d' ' -f2 | sort -u | sed -E "s/^${PREPULL_REGISTRY}\///" \
)
prepull_if_not_exist $PREPULL_REGISTRY "${PREPULL_IMAGES[@]}"
```

Listing 11

3. ξεκινά το κοντέινερ `socks5` proxy και `sshuttle` proxy, και κάνει διαθέσιμη τη διεύθυνση IP του κοντέινερ στο εσωτερικό δίκτυο docker μέσω μεταβλητών περιβάλλοντος.

```
echo '==> Starting the proxy container...' >&2
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml up -d sshuttle
# Get the IP address of the `ccs-sshuttle` container in the
↪  `ccstools` network.
export SSHUTTLE_IP=$(docker inspect -f '{{
↪  .NetworkSettings.Networks.ccstools.IPAddress }}' ccs-sshuttle)

# Start the socks-proxy, used during the build time.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml --profile setup up -d socks-proxy
# Set the ALL_PROXY environment variable to use the socks-proxy
↪  container.
export ALL_PROXY='socks5h://socks-proxy:1080'
```

Listing 12: *Εκκίνηση του κοντέινερ `sshuttle` και `socks5` proxy*

4. εκτελεί την κατάλληλη εντολή `docker compose` για να ξεκινήσει τις ζητούμενες εφαρμογές από το stack της ομάδας. Η εντολή αυτή είναι η εξής:

```
# Start the applications.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml $DEBUGGER_CONFIG --profile app up
↪  -d $COMPOSE_OPTIONS $@
```

Listing 13: *Εκκίνηση των ζητούμενων εφαρμογών*

5. κλείνει το κοντέινερ `sshuttle` proxy, καθώς είναι απαραίτητο μόνο κατά τη διάρκεια πιθανής κατασκευής των εικόνων

```
# Stop the socks-proxy container after container builds have
↪  completed.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml --profile setup rm --force --stop
↪  socks-proxy >&2
```

Listing 14: *Τερματισμός του κοντέινερ `sshuttle` proxy*

4.1.1 Δημιουργία του Docker δικτύου και Volume

Χρησιμοποιήθηκε επίσημη εικόνα του `postgresql`, τροποποιημένη ελαχίστως για να δημιουργεί ονοματισμένες βάσεις κατά την εκκίνηση σε περίπτωση που δεν υπάρχουν, όπως δίνονται χωρισμένες με κόμμα σε μορφή μεταβλητής περιβάλλοντος.

```
environment:
    POSTGRES_MULTIPLE_DATABASES:
    ↪  ${FGC_DB_DATABASE},${FORTLOGS_DB_DATABASE} #,${FGC_TEST...
```

Listing 15: *Μορφή μεταβλητής περιβάλλοντος για την δημιουργία των βάσεων δεδομένων*

```
#!/bin/bash
set -e
set -u

function create_user_and_database() {
    local database=$1
    echo "  Creating user and database '$database'"
    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
        CREATE USER $database;
        CREATE DATABASE $database;
        GRANT ALL PRIVILEGES ON DATABASE $database TO $database;
EOSQL
}
if [ -n "$POSTGRES_MULTIPLE_DATABASES" ]; then
    echo "Multiple database creation requested:
    ↪  $POSTGRES_MULTIPLE_DATABASES"
    for db in $(echo $POSTGRES_MULTIPLE_DATABASES | tr ',' ' '); do
        create_user_and_database $db
    done
    echo "Multiple databases created"
fi
```

Listing 16: *Σκρίπτ έναρξης του container της Postgres*

Είναι επίσης δυνατόν να γεμίσουν οι δημιουργημένες βάσεις δεδομένων, για παράδειγμα με ένα στιγμιότυπο δεδομένων που έχει ληφθεί από εκδοχές των βασεών όπως χρησιμοποιούνται από την διατεθημένη μορφή των εφαρμογών, αλλά χρειάζεται λίγη χειροκίνητη παρέμβαση, ειδικά όταν πρόκειται για την δημιουργία αυτών των στιγμιοτύπων. Μετά την δημιουργία ενός στιγμιοτύπου, το παρακάτω script εκτελείται κατά την εκκίνηση του κοντέινερ της βάσης δεδομένων, αμέσως μετά το προηγούμενο που εμφανίζεται παραπάνω.

```
psql \
    --dbname="$FGC_DB_DATABASE" \
    --username="$POSTGRES_USER" \
    --no-password \
    </sql-dumps/dbod-fgc-dev-dump.sql
```

Listing 17: *'Τέμισμα' των δημιουργημένων βάσεων δεδομένων*

Και τα δύο scripts ρυθμίζονται να εκτελούνται κατά την εκκίνηση του κοντέινερ της βάσης δεδομένων, εκμεταλλευόμενα τον μηχανισμό της εικόνας `postgres` για τη ρύθμιση της εκκίνησης της βάσης δεδομένων, ο οποίος είναι ορατός στο 18, όπου τα scripts αντιγράφονται στο κοντέινερ και μετονομάζονται για να επιβάλουν τη σειρά εκτέλεσής τους, με βάση τον προηγουμένως αναφερθέν μηχανισμό της εικόνας `postgres`, τεκμηριωμένο με περισσότερες λεπτομέρειες στην επίσημη τεκμηρίωση [Pos].

```
FROM postgres:15.1
COPY create-multiple-postgresql-databases.sh
↪  /docker-entrypoint-initdb.d/
↪  1_create-multiple-postgresql-databases.sh
COPY populate-databases.sh
↪  /docker-entrypoint-initdb.d/2_populate-databases.sh
COPY --chown=postgres dbod-fgc-dev-dump.sql
↪  /sql-dumps/dbod-fgc-dev-dump.sql
COPY --chown=postgres dbod-fgc-test-manager-dump.sql
↪  /sql-dumps/dbod-fgc-test-manager-dump.sql
```

Listing 18: *Ρύθμιση για το τρέξιμο των σκρίπτ κατά την εκκίνηση του container βάσης δεδομένων*

4.1.2 Λήψη εικόνων εκ των προτέρων

Για τις περισσότερες από τις εικόνες που χρειάζεται η ομάδα, το CERN καθιστά το αποθετήριο εικόνων GPN `https://registry.cern.ch` προσβάσιμο ακόμη και από το διαδίκτυο. Ωστόσο, λόγω ζητημάτων αδειοδότησης, η βασική εικόνα python που χρησιμοποιήσαμε, είναι περιορισμένη, και μπορεί να μεταφορτωθεί μόνο ενώ βρίσκεστε στο δίκτυο του CERN, προκαλώντας την αποτυχία της εντολής `docker compose`. Για αυτόν

τον λόγο, χρησιμοποιήθηκε μια λύση με χρήση του `sshuttle` και του `docker-in-docker`(DinD), για να μεταφορτωθούν οι εικόνες πριν την κλήση της εντολής `docker compose`, αν χρειαστεί. Το Docker In Docker είναι - όπως υποδηλώνει το όνομα - ένα κοντέινερ docker προρυθμισμένο με έναν πελάτη docker. Εκτελώντας docker μέσα σε ένα κοντέινερ, μπορούμε να έχουμε πλήρη έλεγχο του περιβάλλοντος εκτέλεσής του και των συνθηκών εκκίνησης, χωρίς να χρειάζεται δικαιώματα root στον μηχανισμό φιλοξενίας ή να τροποποιούμε οποιαδήποτε συμπεριφορά του εγγενούς χρόνου εκτέλεσης και πελάτη docker του φιλοξενούντος μηχανήματος. Αυτό μας επέτρεψε να χρησιμοποιήσουμε το `sshuttle` - με τον ίδιο τρόπο που χρησιμοποιείται για τα runtimes των εφαρμογών - ως δρομολογητή, για να δρομολογήσουμε όλη την κίνηση DinD μέσω του GPN του CERN, και έτσι να μπορέσουμε να τραβήξουμε την εικόνα και να την εξαγάγουμε στον εγγενή πελάτη docker του φιλοξενούντος μηχανήματος. Η υλοποίηση της κύριας λογικής δίνεται παρακάτω, με ένα παράδειγμα χρήσης στο 48, παραπάνω.

```
# Starts a dind image proxied through GPN and copies auth
↪  configuration to it.
# Optional first argument is the proxy host
setup_proxied_dind () {
    export PROXY_HOST="${1:-$DEFAULT_PROXY_HOST}"
    docker compose -f docker-compose.yml -f
    ↪  docker-compose.teleworking.yml up -d sshuttle
    export SSHUTTLE_IP=$(docker inspect -f '{{
    ↪  .NetworkSettings.Networks.ccstools.IPAddress }}'
    ↪  ccs-sshuttle)
    docker compose -f docker-compose.yml -f
    ↪  docker-compose.teleworking.yml up -d dind
    ESCAPED_DOCKER_CONFIG=$(sed 's/"/\\"/g' <<<
    ↪  $(generate_docker_config))
    docker exec ccs-dind bash -c "mkdir -p /root/.docker && echo
    ↪  \"$ESCAPED_DOCKER_CONFIG\" > /root/.docker/config.json"
}

# Stops the dind image and the sshuttle container.
tear_down_proxied_dind () {
    # ...
}

# First arg is the registry name, rest are images to pull.
prepull_images () {
    REGISTRY=$1
    # ...
    for image in "${@:2}"; do
    image=$REGISTRY/$image
    echo "==> Pre-Pulling $image into DinD container's docker
    ↪  daemon"
    docker exec -t ccs-dind bash -c "docker pull $image"
    echo "==> Loading $image to local docker daemon..."
    docker exec ccs-dind bash -c "docker save $image" | docker load
    done
}
# First arg is the registry name, rest are images to pull.
prepull_if_not_exist () {
    # ...
    REGISTRY=$1
    if ! images_exist_locally $@; then # Do nothing if all images
    ↪  requested exist locally
        echo "==> Prepulling images from $REGISTRY through proxy..."
        setup_proxied_dind
        prepull_images $@
        tear_down_proxied_dind
    fi
}
```

Listing 19: *Υλοποίηση λήψης εικόνων με χρήση DinD*

4.1.3 Δρομολόγηση

Και τα δύο συστήματα δρομολόγησης υλοποιούνται ως επιπλέον υπηρεσίες docker compose, στο προηγουμένως αναφερθέν `docker-compose.teleworking.yml`.

sshuttle

Το service `sshuttle` εξασφαλίζει ότι ο docker daemon δίνει στο κοντέινερ τις απαραίτητες δυνατότητες:

```
sshuttle:
build:
    context: ./deployment/local/sshuttle
    dockerfile: Dockerfile
container_name: ccs-sshuttle
profiles:
    - setup
environment:
    USERNAME: ${USERNAME} # ssh user to log in as
    PROXY_HOST: ${PROXY_HOST} # ssh host to log in to
cap_add:
    - NET_ADMIN
sysctls:
    - net.ipv4.ip_forward=1
volumes:
    # host's ssh key is mounted into the container to use for
    ↪  connecting to the proxy target.
    - "./deployment/local/id_ed25519:/root/.ssh/id_ed25519:ro"
command: ["sh", "start.sh"]
```

Listing 20: *Ορισμός compose service του δρομολογητή sshuttle*

ενώ το ίδιο το container είναι ένα απλό κοντέινερ με δυνατότητες python, που έχει εγκατεστημένο το `openssh` και το `sshuttle`, και εκτελεί τα παρακάτω κατά την εκκίνηση, για να ρυθμιστεί ως δρομολογητής:

```
# Allow IP masquerading (i.e., NAT).
iptables -t nat -A POSTROUTING -s 192.168.50.0/24 -o eth0 -j
↪  MASQUERADE
# Accept paceket forwarding (i.e., routing).
iptables -A FORWARD -o eth0 -j ACCEPT

ssh-add /root/.ssh/id_ed25519

sshuttle --dns --listen 0.0.0.0 -x sshuttle:22 --ssh-cmd "ssh -v -i
↪  /root/.ssh/id_ed25519 -o StrictHostKeyChecking=no -o
↪  ForwardAgent=yes" -r ${USERNAME}@${PROXY_HOST} 0/0
```

Listing 21: *Σημείο εισόδου του container sshuttle*

Socks

The `socks` proxy service simply starts an `openssh` capable container, and invokes the necessary `ssh` command.

Το service του `socks` δρομολογητή απλώς εκκινεί ένα container με δυνατότητες `openssh`, και εκτελεί την απαραίτητη εντολή `ssh`.

```
socks-proxy:
build:
    context: ./deployment/local/socks-proxy
    dockerfile: Dockerfile
container_name: ccs-socks-proxy
profiles:
    - setup
restart: unless-stopped
expose:
    - 1080
volumes:
    - "./deployment/local/id_ed25519:/root/.ssh/id_ed25519:ro"
command: ["ssh", "-NT", "-D", "socks-proxy:1080", "-o",
↪  "StrictHostKeyChecking=no", "-o", "ExitOnForwardFailure=no",
↪  "${USERNAME}@${SOCKS_PROXY_HOST}"]
```

Listing 23: *Ορισμός του service του δρομολογητή socks*

Ρύθμιση πελάτη του δρομολογητή

Τέλος χρησιμοποιώντας yaml anchors, όλες οι υπηρεσίες compose των εφαρμογών επεκτείνονται με την απαραίτητη ρύθμιση. Για τον δρομολογητή socks, όπως αναφέρθηκε στο προηγούμενο κεφάλαιο, μια απλή μεταβλητή περιβάλλοντος είναι αρκετή για να ανακαλύψουν και να χρησιμοποιήσουν οι διαδικασίες pip των κοντέινερ, ενώ για τον δρομολογητή sshuttle, προστίθεται ένα script entrypoint στα κοντέινερ των εφαρμογών, που ορίζει το κοντέινερ sshuttle ως την προεπιλεγμένη διαδρομή για την κυκλοφορία στο διαδίκτυο.

```yaml
x-teleworking-base-service: &teleworking-config
    build:
        args:
        # socks proxy used only during build time.
        ALL_PROXY: ${ALL_PROXY-}
    environment:
        # sshuttle proxy used during runtime.
        SSHUTTLE_IP: ${SSHUTTLE_IP:-}
    entrypoint: ["/app/container-entrypoint.teleworking.sh"]
    volumes:
        - ./script/container-entrypoint.teleworking.sh:/app/
        ↪  container-entrypoint.teleworking.sh
# ...
# Extend services
fgc-api: *teleworking-config
fgc-commander-server: *teleworking-python-config
# ...
```

Listing 24: *Επέκταση του service για υποστήριξη δρομολογητή*

```
# Make the route through the 'sshuttle' container the default route.
ip route replace default via $SSHUTTLE_IP

exec "${@}" # Execute the command passed to the entrypoint, as
↪  normal.
```

Listing 25: *Σημείο εισόδου του πελάτη του δρομολογητή*

4.1.4 Έναρξη εφαρμογών

Στην κύρια `docker compose` εντολή που εκκινεί τις εφαρμογές (Listing 13), φαίνονται δύο επιπλέον σημεία παραμετροποίησης `$DEBUGGER_CONFIG` και `$COMPOSE_OPTIONS`. Το δεύτερο χρησιμοποιείται απλώς για να περάσουν έξτρα παράμετροι όπως `--build` και `--force-recreate`, που μπορούν να χρησιμοποιηθούν κατά τη διάρκεια της τοπικής ανάπτυξης για να επιβληθεί ανακατασκευή των εικονών.Το πρώτο προσθέτει υποστήριξη για λειτουργία debugger, προεραιτικά. Η μεταβλητή περιέχει το μονοπάτι ενός ακόμη `docker-compose.yml` αρχείου το οποιό παρακάμπτει μερικές ρυθμίσεις των compose service.

```
# ...
fgc-api:
    ports:
        - ”127.0.0.1:5679:5678”
    command: [”sh”, ”-c”, ”python -m debugpy --log-to-stderr --listen
    ↪  0.0.0.0:5678 -m uvicorn --host 0.0.0.0 --port 8080 --reload
    ↪  server.main:app”]
fgc-commander-server:
    ports:
        - ”127.0.0.1:5681:5678”
    command: [”sh”, ”-c”, ”python -m debugpy --log-to-stderr --listen
    ↪  0.0.0.0:5678 -m uvicorn --host 0.0.0.0 --port 8080 --reload
    ↪  server.main:app”]
# ...
```

Listing 26: *Υποστήριξη για debugging διεργασιών python σε docker compose*

Στη πράξη, αυτό που παρακάμπτει είναι η εντολή που εκκινεί τις εφαρμογές, τυλίγοντάς την με μια έξτρα διεργασία `debugpy`, και κάνοντας κάποια port του container προσβάσιμα απ’το δίκτυο του φιλοξενούντος υπολογιστή, για σύνδεση μέσω ενός debugpy client, όπως για παράδειγμα έχει το VSCode.

4.1.5 Τερματισμός

Τέλος, ο δρομολογητής `socks` τερματίζεται, ούτως ώστε να αποφευχθούν συγκρούσεις κατά τη διάρκεια της λειτουργίας των εφαρμογών, και να βελτιωθεί η επίδοση των

συστημάτων όπου γίνεται η ανάπτυξη των εφαρμογών.

4.1.6 Επισκόπηση του σκριπτ υποστήριξης τοπικής ανάπτυξης

```
./script/start-local-development.sh -h
Usage: start-local-development.sh [OPTIONS] [SERVICE...]

Start the local containerised development

Options:
    -a                          Run all apps.
    -b                          Build services. (--build option
    ↪  for docker-compose)
    -d                          Disable VS Code debugger.
    -h                          Print help.
    -n                          No proxy.
    -o                          Use production operational files
    ↪  instead of development operational files.
    -r                          Recreate services. Useful when you
    ↪  have proxy connection issues or want to clear ad hoc changes
    ↪  to the containers. (--force-recreate option for
    ↪  docker-compose)
```

Listing 27: *Μύνημα βοήθεια του σκρίπτ εκκίνησης τοπικής ανάπτυξης*

4.2 Διάθεση σε υπολογιστικά συμπλέγματα

Έχοντας πρόσβαση στο συνεργατικό δοκιμαστικό πρόγραμμα των τμημάτων ATS-IT για Kubernetes στο Τεχνικό Δίκτυο σε πρώιμο στάδιό του, είχαμε τη δυνατότητα να δημιουργούμε υπολογιστικά συμπλέγματα κατά βούληση, με επιπλέον υποστήριξη για τις εξής βασικές λειτουργίες:

- **LetsEncrypt:** Για αυτόματη δημιουργία και ανανέωση πιστοποιητικών SSL.
- **CSI-NFS-Driver:** Που χρησιμοποιήθηκε για την υποστήριξη της λειτουργίας των Persistent Volumes.

4.2.1 ArgoCD

Το ArgoCD χρειάστηκε να το εγκαταστήσουμε εμείς. Η διαδικασία είναι γενικότερα απλή, πλην δύο προσθηκών που ήταν απαραίτητες για την υποστήριξη της ταυτοποίησης oidc του CERN και το προαναφερθέν `helm-secrets` για την αποκρυπτογράφηση μυστικών που έχουν κρυπτογραφηθεί με τη χρήση `sops`.

```
# Create the argocd namespace.
kubectl create ns argocd
# Create the secret containing private key for decrypting secrets.
kubectl apply -n argocd -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
    name: helm-secrets-private-keys
type: Opaque
data:
    key.asc: $(base64 -i <path-to-file-with-private-key>)
EOF
# Install ArgoCD.
helm install argocd ./argo-cd  -n argocd
```

Listing 28: *Εγκατάσταση ArgoCD*

Στη συνέχεια, χρησιμοποιώντας τη μέθοδο app-of-apps, μπορούμε να κάνουμε διαθέσιμες τις εφαρμογές μας, παραμετροποιώντας τες με ένα `values` αρχείο yaml με όποιες ρυθμίσεις χρειάζονται για τον συγκεκριμένο κάθε φορά στόχο, με μόνη μας παρέμβαση την εγκατάσταση ενός κύριου Application, το οποίο εμπεριέχει τις εφαρμογές. Σημειώνεται ότι δίνεται και η δυνατότητα να παρακολουθούνται κλαδιά git εκτός του κυρίως, χρήσιμη λειτουργία και την ανάπτυξη και την επιδιόρθωση σφαλμάτων της ίδιας της μεθόδου διάθεσης εφαρμογών

```
export CCS_REVISION=<revision> export CCS_ENV=<environment> &&
↪  envsubst < entrypoint.yaml | kubectl apply -f -
```

Listing 29: *Εντολή εγκατάστασης του κύριου Application*

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
    # ...
    source:
        repoURL: "https://gitlab.cern.ch/ccs/fgc_web.git"
        targetRevision: "${CCS_REVISION}"
        path: deployment/kubernetes/entrypoint
        helm:
        valueFiles:
            - /deployment/kubernetes/.envs/${CCS_ENV}.yaml
            - secrets+gpg-import-kubernetes://argocd/
              ↪ helm-secrets-private-keys#key.asc?../.envs/
              ↪ ${CCS_ENV}-secrets.yaml
        # ...
    destination:
        server: "https://kubernetes.default.svc"
        namespace: argocd
```

Listing 30: *App of Apps* `entrypoint.yaml`

```
entrypoint
├─ values.yaml
├─ templates
│  ├─ ...
│  ├─ ...
│  ├─ fgc-commander.yaml
│  └─ fgc-api.yaml
└─ Chart.yaml
```

Listing 31: *Δομή αρχείων του κυρίως chart*

Με αυτό τον τρόπο η διάθεση νέων εφαρμογών είναι απλή, καθώς αρκεί να κάνουμε commit ένα νέο Application στο φάκελο `templates` παραπάνω, που να δείχνει σε ένα νέο chart, επίσης αποθηκευμένο στο αποθετήριο, και οι αλλαγές εφαρμόζονται αυτόματα στο σύμπλεγμα. Υποστηρίζεται επίσης η απενεργοποίηση εφαρμογών σε συγκεκριμένα περιβάλλοντα στόχους μέσω του αρχείου `<env>.yaml` για το συγκεκριμένο περιβάλλον στόχου -χωρίς να χρειάζεται να αλληλεπιδράσουμε άμεσα με το σύμπλεγμα για να εφαρμόσουμε τις αλλαγές- προσθέτοντας μια συνθήκη στο σχετικό αρχείο

`<app-name>.yaml` στα `entrypoint/templates`.

```
{{ if (dig ”fgc-api” ”server” ”tag” false .Values.apps) -}}
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
    name: fgc-api
    # ...
{{- end -}}
```

Listing 32: *Υποστήριξη απενεργοποίησης εφαρμογών στο `entrypoint/template/fgc-api.yaml`*

4.2.2 Applications

Για τα ίδια τα Applicatiion, χρησιμοποιήσαμε ένα chart helm για το καθένα. Οι περισσότεροι απ'τους απαραίτητους πόρους ήταν οι ίδιοι για όλες τις εφαρμογές, οπότε δημιουργήσαμε και συντηρούσαμε παράλληλα με μια άλλη ομάδα του CERN, ένα κοινό chart βιβλιοθήκη. Το chart χειρίζεται όλους τους κοινούς πόρους και δημιουργήθηκε με τέτοιο τρόπο ώστε να ελαχιστοποιεί τις γνώσεις που απαιτούνται για τη χρήση του. Για κάθε εφαρμογή, χρησιμοποιήθηκαν τρία αρχεία `values`:

- **`values.yaml`**: Οι στατικές γενικές τιμές για όλες τις εφαρμογές σε όλους τους στόχους διάθεσης.
- **`.envs/<env>.yaml`**: Οι ανά στόχο διάθεσης τιμές.
- **`.envs/<env>-secrets.yaml`**: Τα μυστικά που αφορούν τον τρέχοντα στόχο διάθεσης, κρυπτογραφημένα με `sops`.

Η εντολή που αναφέρθηκε στο Listing 29, και συγκεκριμένα η μεταβλητή `CCS_ENV` χρησιμοποιείται για να επιλέξει το τμήμα `<env>` των παραπάνω αρχείων `values`.

Πέρα από αυτό, η διάθεση των περισσότερων εφαρμογών είναι αρκετά τυποποιημένη, με τους εξής πόρους ανά εφαρμογή:

- **`Deployment`**: Χειρίζεται το βασικό container της εφαρμογής.

- **`Service`**: Εκθέτει την εφαρμογή στο σύμπλεγμα.
- **`Ingress`**: Εκθέτει την εφαρμογή στο διαδίκτυο.
- **`ConfigMap & Secret`**: Παρέχει στην εφαρμογή την απαραίτητη παραμετροποίηση, που προέρχεται απ'τα αρχεία `values`.

Τα frontend των εφαρμογών χρειάζονταν όμως λίγο παραπάνω σκέψη, καθώς τα `Vue.js` frontends σε παραγωγή είναι απλώς στατικά αρχεία και πρέπει να διακομίζονται από έναν web server. Ταυτόχρονα, οι μεταβλητές περιβάλλοντος ενσωματώνονται στις εφαρμογές κατά τη διάρκεια της διαδικασίας κατασκευής, κάτι που θα δημιουργούσε ένα πρόβλημα $N \times M$, καθώς θα έπρεπε να κατασκευάσουμε N frontends για M στόχους. Η λύση ήταν να χρησιμοποιήσουμε ένα μόνο κοντέινερ `nginx`, να κατασκευάσουμε τα frontends στο σύμπλεγμα και να τοποθετήσουμε τα παραγόμενα αρχεία κατασκευής σε αυτό κατά τη διάρκεια της εκτέλεσης, χρησιμοποιώντας ένα `ConfigMap` για να παραμετροποιήσουμε τις διαδικασίες κατασκευής, και έναν κοινόχρηστο volume για να μοιραστούμε τα αρχεία κατασκευής μεταξύ των builders του frontend και του κοντέινερ `nginx`. Αυτή η προσέγγιση είχε το πρόσθετο πλεονέκτημα ότι δεν χρειαζόταν ένα επιπλέον Argo Application resource ανά frontend, καθώς όλα διαχειρίζονταν από το ίδιο Application resource, που με λίγο helm templating, δημιουργούσε ένα νέο Deployment για κάθε builder frontend.

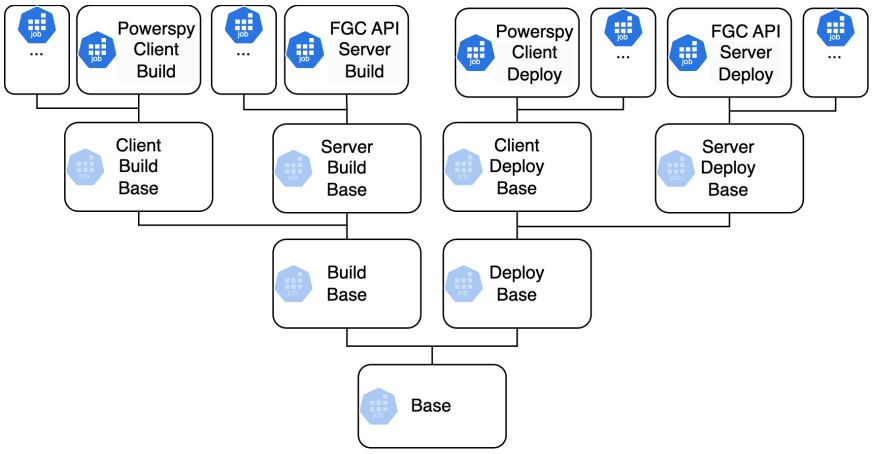
```
{{- range $name, $details := .Values.apps }} # loop over app keys in
↪  <env>.yaml
{{- if hasKey $details ”client” }} # if app key has a ’client’ key
{{- $args := dict ”Args” (dict ”name” $name ”details” $details) }}
{{- $context := merge $args $ }}
# Create a deployment for the frontend builder by merging the custom
↪  deployment with the library one.
{{ include ”library.util.merge” (list $context
↪  ”web-server.client.deployment” ”library.deployment”) }}
---
{{- end }}
{{- end }}
```

Listing 33: *Δημιουργία των deployments για τα frontends*

4.3 Αυτοματοποιήσεις στην ανάπτυξη και διάθεση των εφαρμογών

4.3.1 Συνεχής ενσωμάτωση (CI)

Τα Gitlab jobs ήταν δομημένα με τέτοιο τρόπο ώστε να αφαιρούν όσο το δυνατόν περισσότερη επανάληψη, κάτι που δυστυχώς δεν ήταν εφικτό σε όλα τα σημεία, καθώς η σύνταξη του Gitlab CI είναι σε κάποια σημεία κακώς τεκμηριωμένη και σε άλλα δεν επιτρέπει χρήσιμη συμπεριφορά για τεχνικούς λόγους. Τα jobs περιορίζονταν να τρέχουν μόνο στο κύριο branch, και τόσο τα build όσο και τα deploy jobs ενεργοποιούνταν αυτόματα όταν γίνονταν αλλαγές συγκεκριμένες για μια εφαρμογή στο κύριο branch, αλλά μπορούσαν επίσης να τρέξουν κατόπιν αιτήματος, κάνοντας push ένα git tag στο αποθετήριο που ακολουθεί μια συγκεκριμένη σύνταξη. Υπήρξε επίσης σκέψη για προσθήκη lint και test jobs για κάθε εφαρμογή, ωστόσο οι περισσότερες δεν είχαν εκτενή -ή καθόλου- test suites, οπότε απενεργοποιήθηκε.



Σχήμα 4.1: *Δομή των gitlab jobs*

Τα build jobs, χρησιμοποιούσαν το `Dockerfile` κάθε εφαρμογής και χειρίζονταν την κατασκευή και την αποστολή των εικόνων στο αποθετήριο εικόνων του CERN. Κάθε εικόνα αποστέλλεται χρησιμοποιώντας μερικές διαφορετικές ετικέτες για να διευκολύνει τη χρήση και την παρακολούθηση του από ποιο git commit κατασκευάστηκε. Η κύρια προσπάθεια ήταν να ρυθμιστεί σωστά και να επεκταθεί η βασική εικόνα `kaniko` που παρέχεται από το CERN, και λίγο trial & error για να ανακαλύψουμε πόση από

τη λογική αυτόματης ενεργοποίησης των jobs του gitlab μπορούσαμε να μεταφέρουμε στα βασικά jobs και να αποφύγουμε την επανάληψη.

4.3.2 Συνεχής διάθεση (CD)

Θα ξεκινήσουμε με μια περιγραφή της πρώτης προσπάθειας, χρησιμοποιώντας το ArgoCD-Image-Updater, το οποίο παρόλο που δεν ήταν αυτό που τελικά επιλέξαμε, επηρέασε την τελική προσέγγιση και βοήθησε στην πιο ακριβή ανακάλυψη των απαιτήσεων, οδηγώντας σε μια πολύ απλούστερη -αν και χειροποίητη- λύση.

Απόπειρα με ArgoCD Image Updater

Όπως αναφέρθηκε παραπάνω, η ομάδα είχε περιορισμένο προϋπολογισμό και μέγεθος, οπότε ένας από τους κύριους παράγοντες σχεδίασης ήταν να αποφευχθούν οι χειροποίητες/προσαρμοσμένες λύσεις όσο το δυνατόν περισσότερο. Με αυτό κατά νου, αποφασίσαμε να χρησιμοποιήσουμε το δικό του plugin του ArgoCD για CD, παρόλο που ήταν -και εξακολουθεί να είναι- σε κατάσταση προ-κυκλοφορίας.

Η χρήση του ArgoCD-Image-Updater όταν χρησιμοποιείται ήδη το ArgoCD είναι μια απλή διαδικασία δύο βημάτων, ειδικά όταν χρησιμοποιούνται helm charts. Πρώτα έπρεπε να το αναπτύξουμε στο σύμπλεγμα, κάτι που στην περίπτωσή μας μπορούσε να γίνει συμπεριλαμβάνοντας το επίσημο chart του ως εξάρτηση στο Deployment του ArgoCD και ρυθμίζοντάς το λίγο μέσω του αρχείου `values`. Στη συνέχεια, έπρεπε να επισημάνουμε τους πόρους ArgoCD Application με τα κατάλληλα annotation, μέσω των οποίων μπορεί να καθοριστεί η πολιτική ενημέρωσης κάθε εικόνας.

```
dependencies:
# ...
- name: argo-cd
    version: 5.36.11
    repository:
    ↪ "oci://registry-tn.⌟
    ↪ cern.ch/common"
# ...
```

Listing 34: *Εγκατάσταση ArgoCD Image Updater*

```
kind: Application
metadata:
  annotations:
    argocd-image-updater.⌟
    ↪ argoproj.io/⌟
    ↪ image-list:
    ↪ web-server=<repo>/⌟
    ↪ <image>
```

Listing 35: *Ρύθμιση ενός ArgoCD Application για χρήση του Image Updater*

Όμως, λόγω της χρήσης ενός git αποθετηρίου για την αποθήκευση των ενημερώσεων ετικετών εικόνας, και της ομάδας να έχει ισχυρές απαιτήσεις να παραμείνει σε ένα μόνο monorepo που περιλαμβάνει όλο τον κώδικα εφαρμογής και τη διαμόρφωση του συμπλέγματος, θα έπρεπε να επιβαρυνθούμε με το κόστος της κλωνοποίησης του αποθετηρίου στο σύμπλεγμα, για να ενημερώσει το ArgoCD-Image-Updater. Αυτό δεν θα ήταν απαραίτητα εμπόδιο, αν το Image-Updater είχε υποστήριξη για ρηχές κλωνοποιήσεις, ή αν μπορούσε να διατηρήσει το git clone μεταξύ των ενημερώσεων. Περάσαμε λίγο χρόνο υλοποιώντας το τελευταίο και το δοκιμάσαμε με επιτυχία σε ένα σύμπλεγμα, αλλά συνειδητοποιήσαμε μετά ότι η μέθοδος διατήρησης δεν επέτρεπε την παρακολούθηση πολλαπλών διαφορετικών συμπλεγμάτων χρησιμοποιώντας το ίδιο branch στο ίδιο repo. Επιπλέον, η ομάδα ανάπτυξης του plugin δεν ήταν ιδιαίτερα ενεργή και η προσαρμογή μας, καθώς και μια αναφορά σφάλματος έμειναν αναπάντητες για πολύ καιρό, πολύ μετά την υλοποίηση της εναλλακτικής μας λύσης.

Προσαρμοσμένη λύση

Τα deploy jobs που χρησιμοποιήσαμε στην προσαρμοσμένη λύση είναι λίγο πιο ενδιαφέροντα, καθώς δεν προκαλούν άμεσα τις αναπτύξεις. Όπως αναφέρθηκε προηγουμένως, η ιδέα ήταν να εξαρτηθούμε από το σύνολο λειτουργιών του Argo για τον συγχρονισμό μεταξύ του αποθετηρίου και των συμπλεγμάτων. Αυτό απλοποίησε πολύ την υλοποίηση, αφαιρώντας κάθε πολυπλοκότητα που προκύπτει από τη ρύθμιση της επικοινωνίας μεταξύ διαφορετικών συστημάτων -σε αυτή την περίπτωση, αποθετηρίου

και συμπλεγμάτων- χειροκίνητα. Ως αποτέλεσμα, τα deploy jobs κατέληξαν να χρειάζεται απλώς να ενημερώνουν ένα μόνο αρχείο στο αποθετήριο.

Το αρχείο αυτό είναι ένα επιπλέον helm `values` αρχείο που προστέθηκε στο 4.2.2 παραπάνω, ονομαζόμενο `latest.yaml`. Περιέχει την τελευταία ετικέτα εικόνας για κάθε κοντέινερ που κατασκευάζεται από τα gitlab pipeline. Σε συνδυασμό με λίγο helm templating logic στις αναπτύξεις, μπορούμε προαιρετικά να κάνουμε τις αναπτύξεις να παρακολουθούν την τελευταία κατασκευή, για οποιοδήποτε κοντέινερ σε οποιονδήποτε στόχο, ορίζοντας την ετικέτα συγκεκριμένου στόχου σε `'latest'`.

```
# Gets .Vales.apps.fgc-api.server.tag (deployment target tag)
{{- $envTag := (index .Values.apps ”fgc-api” ”server” ”tag”) }}
# Gets .Values.latest.fgc-api.server.tag (latest build tag)
{{- $latestTag := (index .Values.latest ”fgc-api” ”server” ”tag”) }}
# If the target specific tag is latest, use the latest build tag.
{{- $overrideTag := (eq ”latest” $envTag) | ternary $latestTag
↪  $envTag }}
{{- $overrideRepository := (index .Values.apps ”fgc-api” ”server”
↪  ”repository”) }}
{{- $overrideDict := dict ”Values” (dict ”image” (dict ”repository”
↪  $overrideRepository ”tag” $overrideTag) ) }}
{{- $context := merge $overrideDict . -}}
# Instantiate a deployment with the proper tag.
{{- include ”library.deployment” $context -}}
```

Listing 36: *Λογική Helm templating για την παρακολούθηση της τελευταίας κατασκευής*

```
.deploy_base:
  stage: deploy
  # ...
  script:
    - git config user.email "deploy-bot@runner.com"
    - git config user.name "GitLab Deploy Bot"
    # Setup remote using access token from Gitlab CI Variables
    - git remote add gitlab_origin https://
    ↪  oauth2:${CI_GITLAB_ACCESS_TOKEN}@gitlab.cern.ch/ccs/fgc_web
    # Checkout master branch (single source of truth for all
    ↪  targets)
    - git checkout master
    - git pull
    # Update latest tag of the app
    - yq --inplace ".latest.${image_name}.tag =
    ↪  \"${CI_COMMIT_SHORT_SHA}\" | .latest.${image_name}.tag
    ↪  style=\"double\"" deployment/kubernetes/.envs/latest.yaml
    - git add deployment/kubernetes/.envs/latest.yaml
    - COMMIT_MESSAGE="Update ${image_name} latest to
    ↪  ${CI_COMMIT_SHORT_SHA}"
    - git commit -m "${COMMIT_MESSAGE}"
    - git push gitlab_origin master -o ci.skip && break
```

Listing 37: *Deploy job του Gitlab CI*

5 Επίλογος

Το τελευταίο κεφάλαιο συνοψίζει τα κύρια μέρη της προσπάθειας και των συνεισφορών. Παρέχει μια εκτίμηση του σχεδιασμού και της υλοποίησης, καθώς και προτείνει πιθανές επεκτάσεις και βελτιώσεις που θα μπορούσαν να προστεθούν στο μέλλον, κάποιες από τις οποίες ήδη βρίσκονται σε εξέλιξη.

5.1 Συμπερασματικά Σχόλια

Το αρχικό ζητούμενο της θέσης ήταν η βελτίωση της υποδομής ανάπτυξης και διάθεσης. Για να το επιτύχουμε αυτό, χωρίσαμε το έργο και αντιμετωπίσαμε κάθε μέρος ξεχωριστά, διασφαλίζοντας τη συμβατότητα μεταξύ τους όπου ήταν απαραίτητο:

- Μεταφέραμε την υποδομή ανάπτυξης σε τοπικό μοντέλο, διευκολύνοντας το στήσιμο και τη χρήση του, και διασφαλίζοντας ότι είναι συνεπές μεταξύ διαφορετικών προγραμματιστών και συστημάτων.
- Βελτιώσαμε την υποδομή διάθεσης εφαρμογών, κάνοντάς την πιο ανθεκτική, κλιμακούμενη και εύκολη στην αναπαραγωγή.
- Βελτιώσαμε την εμπειρία ανάπτυξης και διάθεσης με CI/CD, αφαιρώντας εμπόδια και πολυπλοκότητα μεταξύ των προγραμματιστών και των περιβαλλόντων διάθεσης.

Αυτά ήδη έχουν διευκολύνει την ένταξη νέων μελών στην ομάδα και την έναρξη συνεισφοράς στις προσπάθειες της ομάδας, καθώς και την εστίαση των υπαρχόντων μελών

στην ανάπτυξη αντί για την αντιμετώπιση προβλημάτων ή τη διαχείριση συστημάτων.

Κατά τη διάρκεια αυτού του έτους+ (14 μήνες) δουλειάς, συχνά συναντήσαμε εμπόδια και προκλήσεις. Μερικές φορές βρίσκαμε σφάλματα στην υποδομή που παρέχεται από το εσωτερικό τμήμα IT του CERN, όπως η ανακάλυψη ότι ο χρόνος σε μερικούς κόμβους του cluster αποκτούσε σταθερή απόκλιση, προκαλώντας απρόβλεπτες συμπεριφορές στις εφαρμογές. Άλλες φορές συναντούσαμε περιορισμούς στα εργαλεία που είχαμε επιλέξει, όπως τα προβλήματα του ArgoCD-Image-Updater όταν χειριζόταν αποθετήρια που ήταν πολύ μεγάλα. Και διαρκώς αντιμετωπίζαμε ζητήματα που εισάγονταν εξωτερικά ή είχαν παραληφθεί αρχικά, για να διασφαλίσουμε ότι η ομάδα μπορούσε να συνεχίσει να εργάζεται χωρίς διακοπές.

5.2 Μελλοντικό Έργο

Αυτό που εν τέλη δημιουργήσαμε είναι μια σταθερή βάση για την ομάδα να χτίσει πάνω της, και ένας σχετικά ανθεκτικός τρόπος για να το κάνει αυτό, ωστόσο εξακολουθεί να είναι ένα έργο σε εξέλιξη, υποστηρίζοντας ένα σχετικά μικρό υποσύνολο από όλα τα οφέλη που μπορούν να προκύψουν από τα εργαλεία που χρησιμοποιούμε τώρα.

- **Γεω-Εφεδρεία:** Το εσωτερικό TN cloud του CERN επεκτείνεται αυτή τη στιγμή σε ένα δεύτερο κέντρο δεδομένων σε διαφορετική περιοχή του CERN, όπου μπορούμε να διεκδικήσουμε μερικούς ακόμη κόμβους για να βελτιώσουμε περαιτέρω την αξιοπιστία των υπηρεσιών μας. Αυτό έχει ήδη δοκιμαστεί, και η προσθήκη νέων κόμβων σε ένα υπάρχον cluster είναι τόσο απλή όσο η εκτέλεση μιας εντολής.

- **Καταγραφή και Παρακολούθηση:** Αυτή τη στιγμή χρησιμοποιείται ο πίνακας ελέγχου του ArgoCD για την παρακολούθηση της κατάστασης των εφαρμογών. Αυτό είναι σχετικά βασικό και εφήμερο, οπότε εξετάζονται άλλα εργαλεία, όπως το Prometheus και το Grafana, για να παρέχουν μια πιο ανθεκτική και μακροχρόνια λύση.

- **Επιβολή SemVer μέσω CI/CD:** Η Semantic Versioning ήδη εφαρμόζεται προεραιτικά μέσω των εργαλείων που έχουν δημιουργηθεί για να βοηθήσουν στην

αλληλεπίδραση με τις ροές εργασίας CI/CD. Ωστόσο, αυτό θα μπορούσε να επιβληθεί περαιτέρω, προσθέτοντας λογική στις ίδιες τις ροές εργασίας για να αποκλείονται αιτήματα merge που δεν συμμορφώνονται με την έκδοση.

- **Αυτοματοποίηση της δημιουργίας νέων εφαρμογών και υπηρεσιών:** Ένα από τα κύρια οφέλη της μετάβασης σε μια δηλωτική προσέγγιση υποδομής βασισμένη σε yaml, είναι ότι είναι πλέον εύκολη η διαχείρισή τους προγραμματιστικά. Αυτό σημαίνει ότι είναι πλέον δυνατό να αυτοματοποιηθούν πολλές διαδικασίες που προηγουμένως ήταν μόνο τεκμηριωμένες σε αρχεία README και έγγραφα, και έπρεπε να ακολουθούνται χειροκίνητα. Αυτό περιλαμβάνει τη δημιουργία νέων εφαρμογών και υπηρεσιών, καθώς και την ανάπτυξη διάθεση εφαρμογών σε νέα περιβάλλοντα.

- **Ενσωμάτωση του Cilium Gateway API:** Οι κλάστερ που παρέχονται από το τμήμα υποδομής του CERN χρησιμοποιούν το Cilium ως CNI. Αυτό σημαίνει ότι μπορούμε να εκμεταλλευτούμε την υποστήριξη του Cilium για το Kubernetes Gateway API, για να παρέχουμε έναν πιο ανθεκτικό και ασφαλή τρόπο έκθεσης υπηρεσιών στον έξω κόσμο και μεταξύ τους.

Εν τέλη, το σημαντικό είναι ότι η ομάδα ήδη χρησιμοποιεί ενεργά τα εργαλεία που έχουμε αναπτύξει, και αυτά έχουν οδηγήσει σε άμεσα ορατές βελτιώσεις στην εμπειρία τους και την ικανότητά τους να παράγουν αποτελέσματα. Η ελπίδα είναι ότι θα φτάσει σύντομα σε ένα σημείο όπου θα μπορούν να θεωρηθούν πλήρη σε χαρακτηριστικά, αφού η ομάδα αποκτήσει περισσότερη αυτοπεποίθηση σε αυτά, και οποιαδήποτε ανάγκη για χειροκίνητη παρέμβαση ελαχιστοποιηθεί ή μεταφερθεί στην ευθύνη της ομάδας υποδομής. Αυτό θα αφαιρέσει οποιαδήποτε ανάγκη για έναν ρόλο μηχανικού υποδομής στην ομάδα, εκπληρώνοντας έτσι τον τελικό μας στόχο... να γίνουμε περιττοί.

1 Introduction

The first chapter will attempt to summarize the historical and technological backdrop of the current endeavor, give a description of the issues and the requirements that guided this work, as well as any existing solutions wherever applicable. It will end with a brief summary of the endeavor itself.

1.1 Motivation

Even though the internet would no longer be considered a new technology, the technological world is still in its infancy when it comes to finding ways to properly explore the possibilities it provides. One good such example is the idea of cloud computing, a relatively recent term - only since the 1990s - and with advancements like Kubernetes being even more recent (2015). Kubernetes along with Docker, have revolutionized the idea of the cloud, bringing it into the mainstream.

1.2 CERN

CERN has been at the forefront of such developments throughout its existence, with the creation of the World Wide Web credited to one of its members, but also with lesser known projects such as the LHC Computing Grid. It is therefore no big surprise that the next step in CERN's computing infrastructure is shaping up to be a move towards Kubernetes. In fact the standard deployment for a big part of the organization's applications is already based on Kubernetes.

These tools however, have already proven to be invaluable in simplifying development and deployment concerns, which makes them a welcome addition to developer teams, even if there still is overhead to using them.

The last remaining subset of internal tools that remains to be migrated to using containers and Kubernetes, is the most integral, CERN's Technical Network (TN) applications, usually mission critical applications, that exist and are accessible only within a (fire)walled off Computer Network mainly for security purposes, and usually require strong high-availability guarantees.

This is what a recent joint project between two CERN departments (the ATS-IT Kubernetes initiative) is attempting to solve, by providing a Kubernetes platform inside the Technical Network (TN).

1.3 SY-EPC-CCS

1.3.1 Accelerator Systems (SY)

The SY department stands for Accelerator Systems department, and is responsible for the accelerator beam-related technical systems of Beam Instrumentation(BI), Beam Transfer(ABT), Electrical Power Converters(EPC), Radio Frequency(RF), Targets, Collimators and Absorbers(STI). It is responsible for maintaining, operating and upgrading the accelerators themselves.

1.3.2 Electrical Power Converters (EPC)

The EPC group is in charge of the design, development, procurement, construction, installation, operation and maintenance of electrical power systems for all accelerators, transfer lines, experimental areas and tests facilities at CERN. This includes the power converters used to power the magnets and other devices that make up the accelerators, as well as the power supplies for the various detectors and other equipment used in the experiments.

1.3.3 Converter Controls Software (CCS)

The SY-EPC-CCS section at CERN (Accelerator Systems - Electrical Power Converters - Converter Controls Software) handles as the name implies the Controls Software for Electrical Power Converters used at most accelerators and other infrastructure in CERN. The section is split into three subsections the FGC team, which handles the firmware and low-level software that directly interfaces with the Controls hardware (created by its sister section SY-EPC-CCE). The Tools team, which is responsible for developing the applications that allow controls experts to configure, adjust and troubleshoot various setups by providing a user friendly interface and powerful data processing and visualization capabilities. Finally the Industrial Controls team, develops and mainly maintains the functionality of all PLC based controls platforms.

The tools provided by the Tools team are often mostly under active development with new features being requested by their users, and need to be highly available to facilitate investigating and fixing problems as fast as possible. These are just two of the reasons the team opted to onboard the ATS-IT Kubernetes project even before it exited its Proof of Concept stage.

FGC Commander

FGC Commander is the crown jewel of the team. It's goal is to provide a user-friendly way to inspect and adjust all the power converters that are under the section's overview. It is a web application, used by experts from many other teams, with varying degrees of computer literacy, and for this reason, provides as much information as possible through visual queues, and as much assistance as possible for running commands to diagnose and adjust the behaviour of the converters.

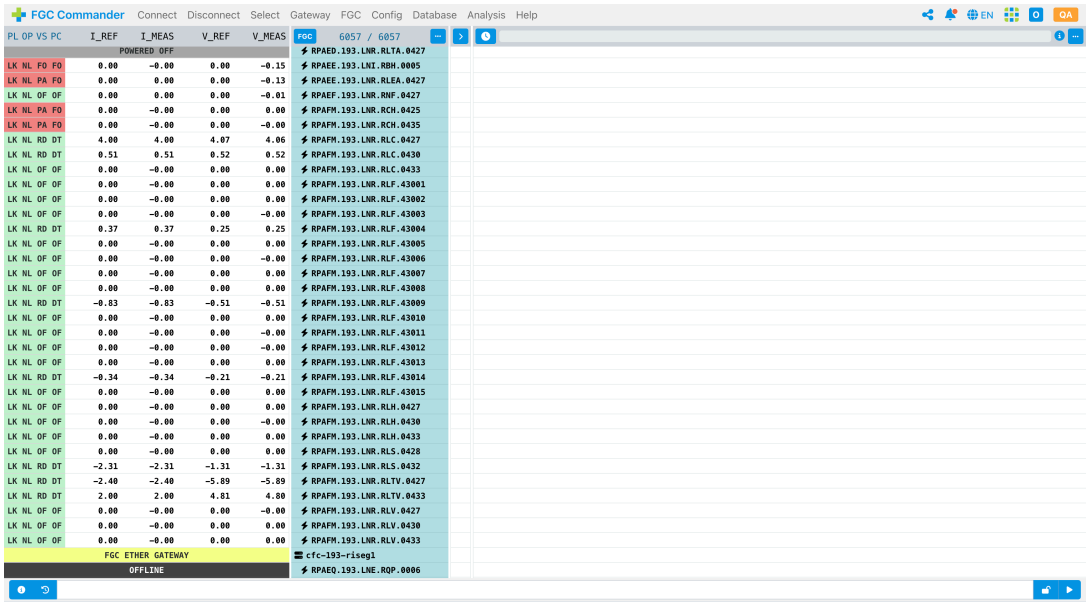


Figure 1.1: *FGC Commander*

FGC API

FGC API is the backbone of most of the team's tools. It's a central RESTful API that provides an abstraction over the firmware that runs on the Front-End-Computers and the power converters themselves. This is what most other web apps use to query the hardware and send commands to it. It being centralized, means there is the least amount of repetition in the codebase, especially for mission critical logic, and since it also is a RESTful API, it's easy to horizontally scale.

Powerspy

Powerspy is the main diagnostics and analysis tool. It uses a custom made visualization library on the frontend, to allow for very high definition graphs, and on the backend it connects with a couple different databases, to provide access to historical data, that can be acquired manually using the same app, or automatically after failures or other events.

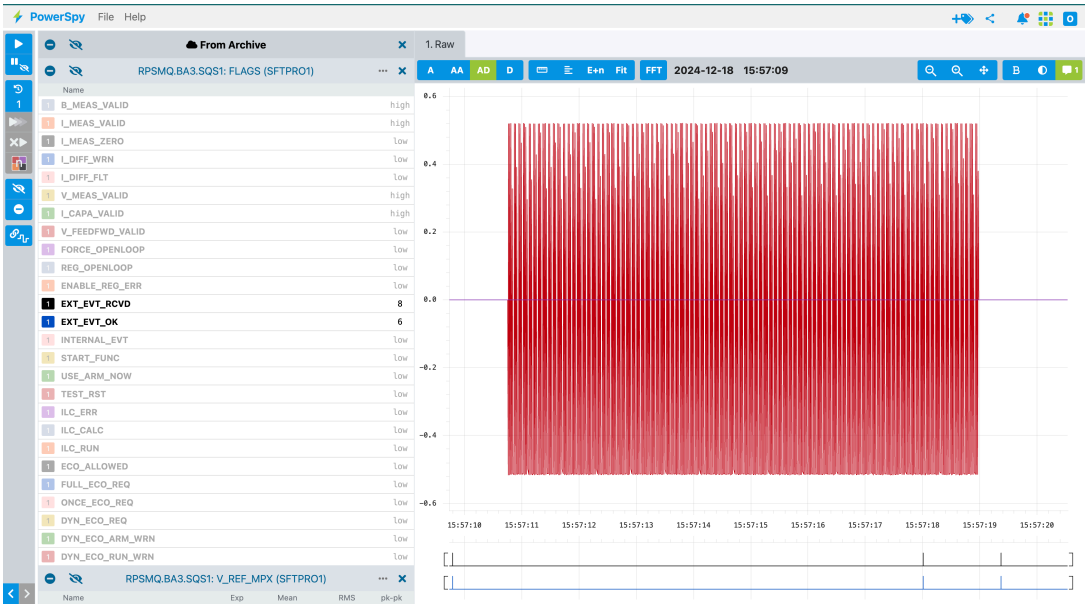


Figure 1.2: *Powerspy*

1.4 Problem Statement

The task of modernizing the infrastructure of the team's tools can be split into a few steps:

1. Containerizing all application components
2. Adding support for local application development
3. Configuring deployments to Kubernetes clusters
4. Automating updates & upgrades (CI/CD)

1.4.1 Containerizing all application components

The main difficulties here lay in identifying each application's system dependencies, which where only documented on the provisioning manifests of the machines where the applications where already deployed. Therefore this was also a good opportunity to standardize environment variable handling, and remove unnecessary system and application dependencies. Furthermore, we had to tackle application environment variables, especially when it came to application clients (frontends), since those are ”baked” into

the application build at build-time, leading an $N \times M$ problem (N apps-M deployment targets)

1.4.2 Adding support for local application development

As previously mentioned, CERN utilizes a Technical Network (TN), this is alongside a General Public Network (GPN) which has Internet Access, but is only accessible through specific gateways, mainly for security, but also for contracting reasons, since CERN sometimes has contracts that provide access to software only inside its networks.

This poses some interesting challenges when trying to support development from user's local machines, especially nowadays, when teleworking has become standard in workplaces. Solutions needed to be found to allow for accessing the CERN internal python package registry, specific Docker images from the CERN container image registry, and more importantly, routing all application network traffic through the TN, to simulate the actual running environment and have access to real devices during application development.

1.4.3 Configuring deployments to Kubernetes clusters

Mapping a docker environment to a kubernetes one, even though a lot more straightforward than replicating a bare-metal one to a new host, can still be challenging, in our case there was the added concern of doing so in a way that easily supports deploying to different clusters using the same configuration.

1.4.4 Automating updates & upgrades (CI/CD)

Finally, a need for automation to assist with updating and upgrading an application as painlessly as possible. Here the ideal end-goal would be abstracting away from the developers all of the details related to building and deploying a new version of an application. Although a relatively common requirement in the industry, it still ended up requiring quite a bit of investigation and a good amount of custom tools and solutions.

1.5 Existing Solutions

1.5.1 Adding support for local application development

The previously used technique to achieve similar results was by providing each team member with a Virtual Machine that would reside within the Technical Network (TN), where they would then perform all development. This was a sub-par solution, since it led to delays for provisioning the machines, added a lot of sysadmin overhead in regards to OS updates and upgrades, as well as setting up and configuring the machines, and increased the attack surface of the Technical Network itself.

1.5.2 Configuring deployments to Kubernetes clusters

Figure 1.3: *Helm logo*

An existing solution, if it could be characterized as such, is the use of Helm (see [Hel]), to assist with separating the configurable from the static parts of the configuration. That proved useful for reducing the amount of maintenance overhead of the configuration however, most of the requirements and issues had to be addressed on a case by case basis

1.5.3 Automating updates & upgrades (CI/CD)

Given the fact that this is a relatively common issue, there are a few community and enterprise driven solutions that claim to handle this part of the problem. The two main ones however are ArgoCD and Flux. Each one comes with its own design decisions and shortcomings, both however attempt to tackle the issue of Continuous Delivery

in a world that is moving away from imperative deployments, and towards declarative infrastructure.

ArgoCD

Figure 1.4: *ArgoCD logo (and mascot)*

ArgoCD is overall more opinionated, offering a nicer experience when starting out, and focuses more on making maintainance and synchronization of cluster configuration and Kuberentes resources easier.

Under the hood it is implemented using mainly the Application Controller which handles Kubernetes Custom Resources of type Application, which in turn, handle Helm charts hosted in a git repo, or a chart repository. After installation on a cluster, the user defines an Application resource for each of the helm charts that are to be deployed on the cluster, and configures ArgoCD, via the webUI or the CLI client to monitor a specific repository, where the Application resource definition files are tracked. In the most common setup, the Application resources are used to specify a git repository where the helm charts are tracked, allowing the cluster to always have the latest version of the charts deployed.

For keeping up-to-date with new versions of the containers used in each helm chart, the Argo team, is currently developing an extra tool, ArgoCD Image Updater, which already supports most of the common use cases. It can periodically check image registries for new images whose tags fulfil some criteria, rollout updates and commit the new tags used back to a git repository.

Flux

Figure 1.5: *FluxCD logo*

Flux in turn, is highly configurable, and split as much as possible into a lot of different Custom Resource Definitions (CRDs) and controllers, each of which handles a specific part of what is needed to make CD work neatly with Kubernetes, implementation-wise however, the idea is the same, comprising of Kubernetes Controllers and Custom Resources. Flux 2, emerged out of the failed effort to consolidate Argo and Flux, and added something called the GitOps Toolkit, which allows Flux to support the feature-set of ArgoCD image updater out-of-the-box.

1.6 Proposed Solution

Throughout this endeavour, due to the team's resources and expertise, it was a strong requirement, to avoid bespoke and custom solutions as much as possible, since they would make maintenance down the line harder. Even so, there are a few parts, where it ended up begin necessary.

1.6.1 Adding support for local application development

This was one of the cases where a bespoke solution was necessary. CERN internal documentation does suggest using various proxying techniques and mentions a utility called sshuttle. In the end, sshuttle was indeed used, but in a way where it was automatically started before applications, and used to route all application network traffic through it.

1.6.2 Configuring deployments to Kubernetes clusters

For the deployments to the clusters, we created a fairly customized set of conventions that almost all application deployments adhered to, and utilized a helm chart library that was created in collaboration with another CERN section. This made it a lot easier to add new applications, and the ease of reconfigurations, since now all application configuration is declarative, which means no need to read through scripts upon scripts to ensure a minor change somewhere, doesn't somehow conflict with or break anything.

1.6.3 Automating updates & upgrades (CI/CD)

Following suit with other CERN teams, and the suggestions of the infrastructure team, we used ArgoCD for automation. Flux 2 was still relatively new, and its approach meant we'd end up with more Flux specific Kubernetes resources to maintain, whereas ArgoCD's approach is in general more portable across Kubernetes environments and configurations. The initial plan was to use ArgoCD Image Updater for the git integration, but even after trying to overcome its shortcomings by contributing to the project itself, the tradeoffs where still to many. In the end we went with a custom solution that relies heavily on the features of ArgoCD itself, as well as helm templating, to achieve the desired results.

1.7 Outline

The following chapters will be:

- **Chapter 2:** Overview of the relevant terminology and concepts
- **Chapter 3:** Description of the existing tech as well as our additions
- **Chapter 4:** Implementation details hurdles and solutions
- **Chapter 5:** Evaluation of the results and future work

2 Background

This chapter attempts to give an overview of the terminology and technologies needed to follow the rest of the work. That includes the concept of containers, containerization and by extension Docker, as well as an introduction into Kubernetes. Finally, a summary of GitOps and in particular GitlabCI basics.

2.1 Containers

In the process of creating software the problems of distribution and portability are always at the forefront, especially when exiting the phases of early development (but hopefully earlier). Modern and higher level languages avoid some of these pains by having a runtime and therefore not requiring target specific recompilations, however they need the runtime to be distributed alongside the software, and they still suffer from the issues of package and library management. Containerization attempts to mitigate these problems by defining a standard way to package any application alongside its dependencies. At the same time it attempts to face the issues of runtime isolation and security. In that sense it ”competes” with traditional Virtual Machines for its use-cases.

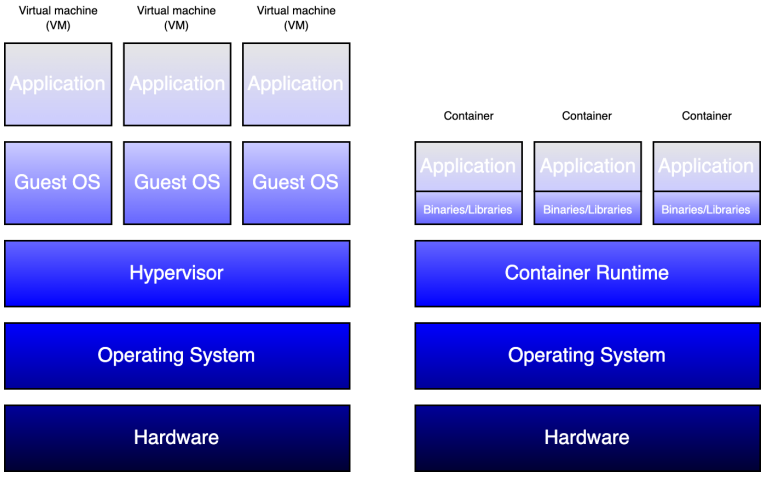


Figure 2.1: *Virtual Machines and containers*

Virtual Machines rely on a Hypervisor that in effect virtualizes the computer hardware, and then run a full OS on top of that simulated hardware. Containers cleverly utilize Linux kernel features, like cgroups and kernel namespaces, allowing applications to use the host's kernel for system calls and IO, improving performance to a near native level. In general containers can be described as a virtualization of the OS, based on the host's OS, in contrast to Virtual Machines, that are a virtualization of the hardware, based on a Hypervisor.

2.1.1 Docker

Docker is a widely used platform that implements containerization, offering tools to build, ship, and run containers. It standardizes how containers are created and managed, providing a consistent way to package applications and their dependencies. Docker's ecosystem includes tools like Docker Engine, for running containers, Docker Hub, a registry for sharing container images, a builder for building images, networking capabilities that allow the running containers to communicate in customizable ways, as well as volumes, for persisting data across container runs.

Volumes

Docker volumes are a mechanism to persist data across container runs. Containers are by nature ephemeral, but applications are often not. Volumes are mounted onto

directories in containers similarly to mounting filesystems on an OS. They can be created standalone, or during a container's creation, and can also be used to share data across containers, by mounting the same volume on two or more.

Networks

Docker networking enables containers to communicate with each other, with the host system, or with external networks. It abstracts and manages the networking aspect of containers, providing flexibility and control over connectivity. Docker offers several network drivers, each serving different purposes:

- **Bridge:** The default network for standalone containers, providing isolated environments with inter-container communication.
- **Host:** Removes network isolation, allowing containers to use the host's network stack directly.
- **None:** Completely disables networking, isolating the container from any network.

There are a few more which are not relevant for common usecases:

- **Overlay:** Useful to connect docker daemons, and therefore connect containers on different machines
- **Ipvlan:** Allows full control over the IP layer, and can therefore be used to make containers integrate directly to an existing network.
- **Macvlan:** Can mask containers to appear as physical devices on a network.

2.1.2 Docker Compose

Docker compose is an official utility provided by the Docker team, that can be used to avoid the need for complex scripts that build, create and run containers one by one. It allows for defining all application components and their dependencies in a single file, and then running them with a single command, without needing to worry for example about which images need to be built or pulled.

Figure 2.2: *Docker Compose Mascot and Logo*

Its main unit of operation is the `docker-compose.yml`, a file that defines the services, networks, and volumes that make up an application. It can be extended by additional files, that can be used to override or extend the base configuration, by redefining parts of the resources.

```
version: '3.8'
services: # Application components or service dependencies
db:
# ...
network: stack-network
app-1-client:
# ...
network: stack-network
app-1-server:
# ...
network: stack-network
networks: # Docker network definitions and config
stack-network:
# ...
```

Listing 38: *Main `docker-compose.yml` features*

2.2 Kubernetes

Easier packaging and distribution of software, has made it easier for developers to move to the cloud, but the cloud, be it in-house or external, comes with its own set of issues.

That is where Kubernetes comes into play. Kubernetes is an orchestration tool, that can abstract away much of the complexity of deploying applications on clusters, a cluster being just a set of (not necessarily uniform) servers/computers. On top of that, it allows deployment to be declarative, which is a fancy way to say, there is much less need for scripts. Since the work of this paper takes place above the Kubernetes API, we will summarize the main Kubernetes user-side terms and objects.

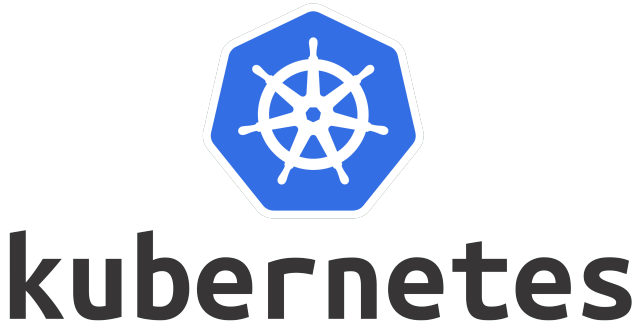


Figure 2.3: *Kubernetes Logo*

Pods

Pods are the smallest deployable units in Kubernetes, representing one or more tightly coupled containers that share resources like storage, network, and namespaces. They are designed to host a single application or a set of closely related processes, ensuring efficient communication. Pods can scale horizontally by creating replicas to handle increased load, and all the containers within a pod are deployed on the same machine to increase communication performance.

Deployments

Deployments manage the desired state of an application by defining the number of replicas and ensuring updates or rollbacks occur smoothly. They provide declarative updates, enabling controlled changes to Pods while maintaining availability. This is accomplished by gradually swapping older versions of an application with newer ones, as well as maintaining an N long history of previous configurations, that can be used to rollback smoothly.

Persistent Volumes

Volumes in Kubernetes provide persistent storage for Pods, ensuring data durability beyond a container's lifecycle. They can be backed by various storage systems, such as local disks, cloud storage, or networked file systems. Volumes are essential for many kinds of applications, although due to the prevalence of the cloud, there's a trend among developers to make applications as stateless as possible.

Services

Services expose Pods to enable stable communication between components or with external systems. They provide abstraction by distributing traffic across a set of Pods using labels, ensuring reliability even if individual Pods fail. Services can operate at various scopes, like ClusterIP, NodePort, or LoadBalancer. A Service alongside a Deployment can increase the uptime of an application by ensuring that incoming traffic is always routed to a replica of the app that is currently running.

Ingresses

Ingresses manage external access to Services, offering fine-grained routing, such as host or path-based rules. They act as an HTTP(S) load balancer, directing traffic to Services within the cluster. Ingresses often include SSL termination, enhancing application security and scalability. They are the final piece of the puzzle needed to make an app deployed on a cluster, accessible from the outside world.

2.2.1 Helm

Helm's main unit of operation is the helm chart, a fancy name for a collection of Kubernetes resources, with a predefined structure and interface to parametrize.

```
chart-name
├─ values.yaml
├─ templates
│  ├─ ...
│  ├─ template-a.yaml
│  └─ template-b.yaml
└─ Chart.yaml
```

Listing 39: *Typical Helm Chart structure*

A `values` file is used to define the parameters that can be passed to the templates, and the templates themselves are the Kubernetes resources that will be deployed. The `Chart.yaml` file contains metadata about the chart, like its name, version, and dependencies. Helm and its cli can be used to install, upgrade, or delete charts, and it can also be used to create new charts from scratch, and is used under the hood by ArgoCD for example, to inflate the charts before applying them to the cluster.

2.3 GitOps

Having now moved into this ”magic” world were software distribution is easy, and configuration is all declarative, it is possible to leverage it to automate the process of updating and upgrading apps, while at the same time benefiting from all the tools that exist for code version control. All of the above is captured in the terms GitOps, and by extension DevOps in the industry.

GitOps leverages tools like Kubernetes operators or continuous delivery systems to monitor repositories and apply changes in real-time. It ensures versioning, and easy rollback by treating Git commits as deployment records, making it easy to keep track of what changed when, which comes especially useful in larger teams with high turnover (such as CERN's)

Continuous Integration

Continuous Integration (CI) is the practice of using automation tools to ensure that additions and modifications to existing software, integrates well with dependents and

dependencies. The category of tools used for this are called pipelines, and are offered by most code-versioning tool providers, Github Actions and GitlabCI being two of the most common ones.

Figure 2.4: *GitlabCI Logo*

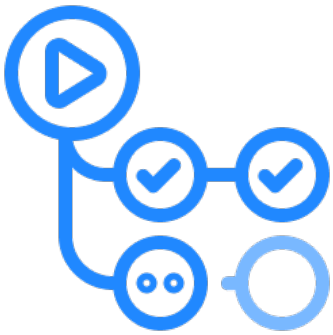


Figure 2.5: *Github Actions Logo*

Continuous Deployment

Continuous Delivery, depends upon the guarantees given by proper CI implementations, and automates the deployment of new application versions. Of course deploying every change everywhere is not something one usually wants, so conventions and different deployment environments can be used, to gradually progress the changes through, terms like QA, Dev, Preprod, Prod are common names for targets. There are various techniques to achieve this, be it with automatic repository reconciliation, image registry monitoring and even webhooks or direct ssh access from pipelines (though usually not recommended).

3 Design

This chapter goes into detail over the design of the existing systems and solutions suggested in previous chapters. It begins with describing the existing systems and highlighting the need to migrate away from them, and ends with outlining the design of the various parts and components of this work.

3.1 CCS Tools

As mentioned briefly before, the Tools team of CCS, handles providing nicer user experience in handling anything related to EPC groups power converters, be it setup, configuration or troubleshooting. It does this by developing and developing and mainting a set of applications, alongside converter experts. The team has slowly been settling on a FastAPI-VueJS stack, however, there are still some older apps and tools, that are perl-based. This is another reason why containerization is a very appealing direction for the team, since it can abstract away most differences related to development stacks.

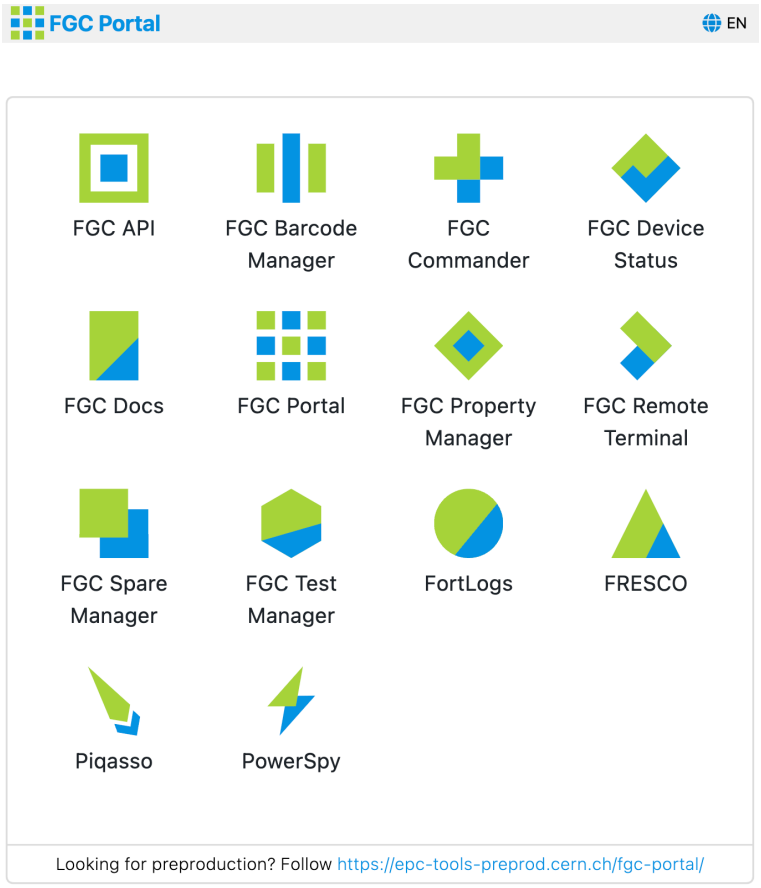


Figure 3.1: *Application catalog on production*

3.2 Existing deployment environment

The previous setup had been in use for a few decades. It was a simple Apache Server and a few Python Systemd services running on a bare metal Linux server, initially created for one or two apps, by the experts themselves, who wanted tools to automate some parts of their work. Even though efforts made to standardize and simplify parts of it, as the scope of the team's responsibilities grew, so did the complexity of the setup, especially when the need arose for more deployment targets, such as development, testing, and production environments, as well as targets outside of the CERN network.

In more detail, the setup could be separated into a few parts:

- **Templates:** Every application has a few resource templates, which are apache vhost configuration files, systemd service definition files and plain `.env` files, with placeholders for values that need to be adjusted per deployment target.

```
    ...
# FGC API WebSockets
ProxyPass         "/fgc-api/api/devices/status-full"
↪  "ws://localhost:${FGC_API_PORT}/devices/status-full"
    ...
```

Listing 40: *Apache vhost configuration template*

```
    ...
[Service]
# Service definition
User=${REMOTE_DEPLOY_USER}
WorkingDirectory=${APP_DIR}/${FGC_API_APP_NAME}
    ...
```

Listing 41: *Systemd service definition template*

```
    ...
STATUS_FILE=${STATUS_FILE}
FGC_DB_DRIVER=${FGC_DB_DRIVER}
FGC_DB_HOST=${FGC_DB_HOST}
FGC_DB_PORT=${FGC_DB_PORT}
    ...
```

Listing 42: *Environment variables template*

- **Parameters:** These are files containing all the environment variables that are needed to parametrize the application's deployment, as well as the application components themselves

```
        ...
DEV_MODE="true"
DEFAULT_APP="fgc-portal"
ENVIRONMENT_FULLNAME="development"
        ...
```

Listing 43: *Parameters file example*

- **Scripts:** The final piece, is a set of shell scripts that automate parts of the process of deploying an application. They substitute the templates, zip the necessary resources, copy them to the target server, and restart any services or components necessary.

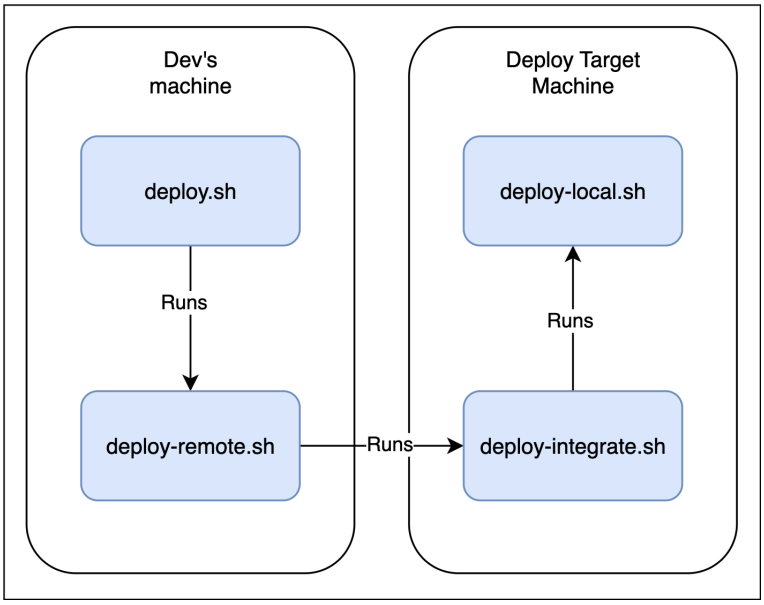


Figure 3.2: *Old deployment flow*

Known Issues

Setting up a new deployment target, which was a recurring need since every new member's development environment had to be a one-to-one copy of the production environment, was a very error-prone process. New machines had to be requested, on which the necessary system dependencies had to be installed, which often led due to changes in CERN's software repositories and the OS versions that CERN provided. In total, it ended up delaying new member's onboarding by about one to two weeks, which is a big issue given CERN's high turnover, and the existence of contracts that last only for a few months.

Making adjustments or adding features to the deployment process proved to be very time-consuming and difficult, because of the imperative nature of the scripts and the setup. A good example of this was when the team needed to override specific environment variables for specific applications when deployed on some target. To do this in a non intrusive way, we needed to highjack the script's env var substitution logic, and add

another layer written in python, since bash could not be easily set up to the task. Even after meticulous overview of every functionality, we still ran into edge case for the next couple months that caused delays and stress to the team.

```
{
    "accwww.cern.ch": {
        "fgc-property-manager/server": {
            "FGC_DB_DRIVER": "oracle"
        }
    },
        // ...
```

Listing 44: *Overrides file example*

3.3 Proposed designs

As briefly described, the suggested solutions were a modern container-based approach, alongside kubernetes and CI/CD methodologies. It can be broken down into a few parts that will be described in the following sections.

3.3.1 Local Development

With the team moving towards Docker and containers, it became possible to support development locally on the dev's machines themselves, avoiding the need for a new VM for each developer. The main requirements where:

- **Simplicity:** The setup should be as simple as possible, especially when it comes to setting up and using it.
- **Extensibility:** It should be easy to add new applications and extend the featureset provided to these applications by the setup.
- **Speed:** The setup should be fast. Fast to set up, fast to start development, and incur minimal performance overhead on development.

And on the application side:

- **Technical Network Access:** The applications need to be able to access other services and resources on the CERN network, such as databases, other APIs, etc.
- **CERN Repositories Access:** The applications need to be able to access CERN's software repositories, to install dependencies and packages, as well as get CERN specific docker base images.
- **Operational File Access:** The applications need to be able to access ccs team's operational files, that are produced by other services or manually adjusted by experts, and are housed on CERN NFS.

Enter Docker Compose. The idea was to use docker compose to remove as much implementation and maintainance overhead from the team as possible, while fulfilling the requirements.

To achieve the application requirements, there would need to be two separate proxying mechanisms, since the image building process requires access to CERN's General Public Network (GPN) and more specifically to CERN's internal Python package repository which is hosted in the GPN, while the applications themselves - as mentioned - need access to CERN's Technical Network (TN).

Build Proxy

For the build process, we could not (easily) adjust the networking configuration of the docker builder, since it would entail the use of a custom builder, complicating the setup's logic and maintainance overhead. There was also the idea of configuring proxying on the docker daemon level, but this meanth that the setup's utilities would require root access, and it would add the complexity of needing to actively check if any images need to be built, in order to restart the daemon with the correct proxy settings, build them, and then restart it again with the original settings, so that the daemon's functionality is not dependent upon the existence of the proxy connection.

Luckily, we could leverage the fact that `python` and by extension `pip` can be configured to use a proxy server, just by setting the `HTTP_PROXY`, `HTTPS_PROXY` and `ALL_PROXY` environment variables, and importantly, these can work with `socks5` proxies, which are easy to setup if one has ssh access to a machine in the target network. This meant that

we could setup a temporary socks5 proxy on a machine in the GPN network, set this variable and export it system-wide so that it is accessible by the `pip` process inside the docker builder, and then unset it after the build process is done.

Application Proxy

The application proxy could not follow the same design, since the processes that do networking traffic are varied under the hood, and not all of them offer the same support for proxying as `pip` does, plus openssh's proxy support can be slow. A more systemic and robust approach was needed, to effectively emulate the application's networking environment in an opaque way. The internal CERN documenation, mentions `sshuttle` as a tool that can be useful to support teleworking, but doesn't give too much info.

Sshuttle gives the features of a normal VPN, requiring only ssh access to a machine in the target network. It grabs TCP traffic on both ends, server and client side, assembling and disassembling the packets, avoiding the issue of doing TCP-over-TCP, and just acting on the application layer. It requires root access, but this is a problem that can be overcome by running it inside a docker container. The end result is a container that acts as a router for all the other containers, that starts and stops easily alongside the rest of the setup, and requires no elevated access on the dev machine.

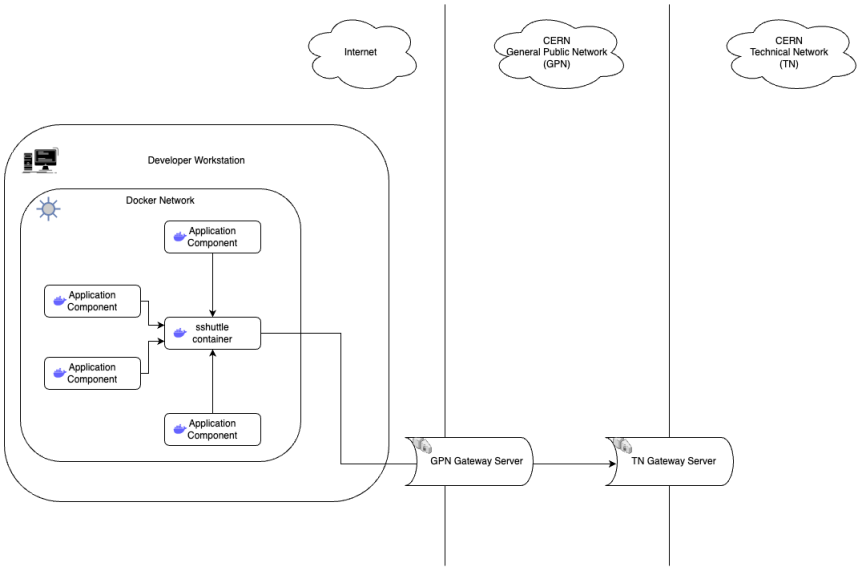


Figure 3.3: *Application development from outside CERN's Network*

3.3.2 Kubernetes

While designing this component, we had the luck of already having settled upon some of the design decisions related to CI/CD, more specifically the use of ArgoCD. This narrowed down the available options, since ArgoCD already points us in some directions, including the usage of Helm. ArgoCD's main unit of deployment is the Application, a custom Kubernetes resource used to declare details about the desired state of an application, such as sync policy or the location and type of applications kubernetes resources.

The ArgoCD community has settled on two main ways of deploying applications, the app-of-apps pattern and the ApplicationSet pattern.

ApplicationSet

ApplicationSets are an opinionated way of deploying multiple applications, that share a similar configuration, but have different values for some of their parameters. It uses various types of "generators" to do the templating, one example of which being the `git` generator, and its sub-types the `git`-dir and `git`-file generators, which use the directory structure and file names accordingly variables for the template. It is a powerful tool, especially for usecases where a single ArgoCD instance manages different clusters, since it can easily facilitate the deployment of the same application on multiple clusters, with different configurations. However in our case each cluster needed to have its own ArgoCD instance, to ensure that the clusters are completely isolated from each other, which was necessary due to the team's aim of providing the applications to external labs and partners. Another issue is that with this pattern, the applications need to be declared under the same chart, that then depending on its parameters, enables or disables each application, thus incresing the coupling between the apps.

App-of-apps

App-of-Apps, is a pattern that uses a single - entrypoint - Application resource to deploy the rest. It is a lot less opinionated, since the rest of the Applications can each be handling a normal Helm chart, that can easily be adjusted to be deployed to most Kubernetes environments.

```
apiVersion: argoproj.io/v1alpha1
kind: Application
    ...
spec:
    source:
        repoURL: <git-repo-url>
        path: <chart-containing-all-apps-as-resources>
    ...
    helm:
    ...
```

Listing 45: *ArgoCD Application resource*

Alongside a properly configured ArgoCD instance, it can handle the reconciliation of the cluster's desired state, and the active state of the applications, which can assist the team a lot in maintaining the applications, since there is a single source of truth for the cluster configuration. It's also possible to use the same repository for the code and the cluster config, which is a big plus in teams that are using a monorepo. These are the main reasons why we chose this pattern in the end.

Secrets Management

One important aspect of any deployment is the handling of secrets. Kubernetes has a built-in secrets management system, that can be used to store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys, but populating and storing the values of the secrets is where the complexity lies, especially when working in a team and with multiple clusters. A useful tool is `sops`, which can be used to encrypt and decrypt files, and can be used to encrypt the values of the secrets, and then store them in the repository. This way, the secrets can be versioned, and the team can have a clear view of what secrets are being used where, and by whom, without exposing them, assuming that the key is properly handled. There is a useful tool to help when working with clusters and `sops`, called `helm-secrets`, which can be integrated with ArgoCD alongside `helm`, to manage the secrets in a more user-friendly way [jkr].

3.3.3 CI/CD & Automation

By this point, we were locked in for most of the design decisions related to CI/CD & Automation. In general what is needed is:

- **Automated Tests**
- **Automated Builds**
- **Automated Deployments**

The first two, are the responsibility of automated pipelines, and are self contained on the gitlab repository's side. The last one is a bit more complicated since it requires in principle some kind of communication (synchronous or not) between the pipelines and the cluster.

Automated Builds & Tests

CERN enforces use of its internal enterprise Gitlab instance as version control for all projects. Therefore it was a natural choice to use GitlabCI for the CI/CD pipeline. GitlabCI is a very powerful tool, that can be used to automate the testing and building of applications, as well as the deployment of them with a little more effort. GitlabCI pipelines are defined using a `.gitlab-ci.yml` file, that is interpreted on the gitlab server, which in turn defines and runs jobs on gitlab runners. Gitlab runners are machines that have been properly configured to run the jobs, and can be either shared runners, provided by CERN, or private runners, that can be configured by the team itself. In our case CERN provides two relevant runners, one preconfigured with Docker, and one with Kaniko, a Docker alternative that doesn't require root access. Initially the design was to use the Docker runner, but for security reasons, its implementation under the hood required the generation of a new VM for every run, which was problematically slow, and the team switched to the Kaniko runner, both of which look similar to the end user.

After that it was pretty straightforward. A set of jobs would be created per application, that would be triggered whenever changes to the application where pushed, and would run tests, build and(optionally) deployment.

Automated Deployments

One first idea of how to do this is directly trigger an update on the cluster from the pipelines. This could be done by creating a service account with the necessary permissions cluster-side, and using that from the pipelines to patch specific Kubernetes resources. However, this breaks the declarative nature of what we're aiming for, since there wouldn't be any source of truth other than the cluster itself, plus, in the case of ArgoCD, argo simply reconciles any manual changes back to the state specified on the repository.

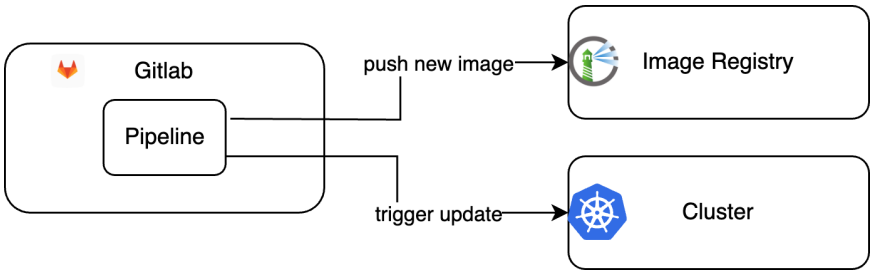


Figure 3.4: *Pipeline to Cluster*

A second idea that we explored thoroughly, was the use of Argo's tool made specifically for this, `ArgoCD Image Updater`. The team originally expressed doubts due to the tool being in under active development, but nonetheless the maintainance burden of a custom solution was deemed a bigger cost. This ended up biting us in the end, since there where a couple of issues that where insurmountable, and we ended up going with a custom solution anyway.

ArgoCD Image updater is an extra workload running on the cluster, periodically checking an image registry for new versions of images that optionally follow a user specified pattern. When found, it pulls a git repo, updates an override file, commits and pushes the changes, thus ensuring the single source of truth requirement is met. It avoids needing any new credentials or permissions by reusing the credentials ArgoCD already has to access the git repository. But as already mentioned, due to the limitations, which will be discussed in detail in the next chapter, we chose to go with a custom solution.

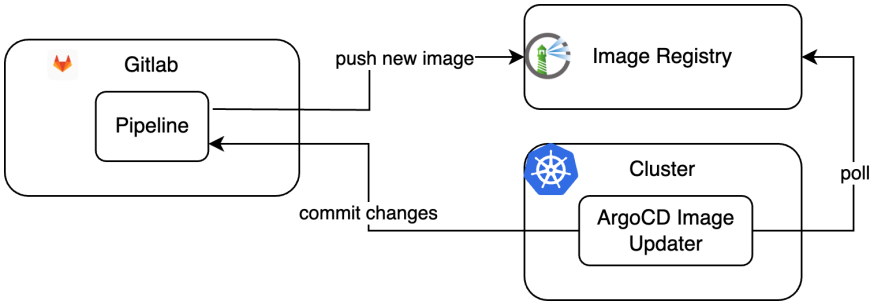


Figure 3.5: *ArgoCD Image Updater*

The custom solution operated in a similar manner, but had the added benefit of not needing to poll, because it moved the update trigger to the pipeline side, thus allowing for synchronous updates. We realized that we could use ArgoCD's main functionality, that is, the automatic reconciliation of the cluster's current state desired state, to achieve the same result.

The end user requirements where to have support for manually triggered updates, as well as automatically tracking the lastest version of an application, in a user friendly way. To get this, we would need something like this:

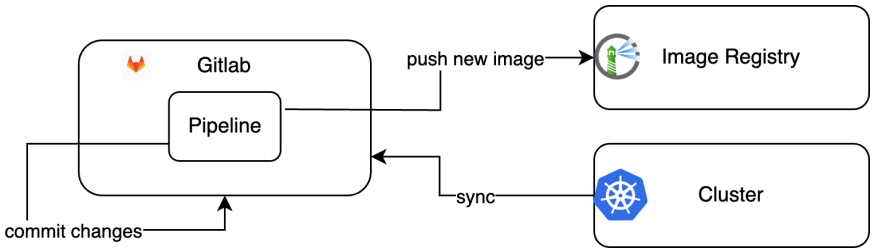


Figure 3.6: *Custom CI/CD Flow*

where the `syncing` is handled by ArgoCD itself. It should be possible to trigger the pipeline automatically on code changes or manually by pushing a specific git tag, or creating a Merge Request. The pipeline would then build the image, push it to the registry, and then update the ArgoCD Application resource, which would trigger the ArgoCD controller to reconcile the cluster's state with the desired state, thus updating the application.

4 Implementation

This chapter will delve into the more technical side of the previously described design. It separates the implemnetation into as distinct as possible components, and analyzes what difficulties needed to be overcome. It concludes by giving an overview of the entire work in action. As before the main components are

- The **Local Development** environment.
- The **Cluster Deployment**.
- The **Development & Deployment Automation**.

4.1 Local Development

After the containerization of all applications and services, a `docker-compose.yml` file was created, and later extended via a `docker-compose.teleworking.yml` file, that could be optionally used to add teleworking capabilities to the setup. Due to limitations related to docker compose's startup order and the proxy necessary for the build process mentioned previously, a wrapper script was created to abstract around a few steps and `docker compose` command invocations.

The process is relatively straightforward. The user invokes the script with the names of the applications they want to start, and the script takes care of the rest.

```
./script/start-local-development.sh fgc-commander-client fgc-portal
```

Listing 46: *Local development wrapper script invocation example*

As an overview - more details will be given where necessary - it:

1. ensures existence of an external docker network, so that different invocations of the `docker compose` command can create containers that exist in the same network easily, and avoid conflicts with CERN's networks. Also ensures existence of a docker volume for the CCS DB, to persist developer changes to the test data.

```
# Create a docker network, if it does not exist already,
# with an explicit subnet to prevent conflicts with CERN addresses.
if [[ -z $(docker network ls -f name=${DOCKER_NETWORK} --format '{{
↪  .Name }}') ]]; then
echo '==> Creating a docker network...'
docker network create \
    --subnet '192.168.50.0/24' \
    $DOCKER_NETWORK
fi

# Create external volume for the CCS DB if does not exist.
CCS_DB_VOLUME='ccs-db-data'
if [[ -z $(docker volume ls --filter "name=${CCS_DB_VOLUME}" --format
↪  "{{.Name}}") ]]; then
echo '==> Creating a docker volume for the local CCS DB instance...'
docker volume create ${CCS_DB_VOLUME}
fi
```

Listing 47: *Ensure existence of docker network and volume*

2. prepulls CERN exclusive images from the CERN image registry.

```
# Prepull images proxying from inside the CERN network
source $LOCAL_DEVELOPMENT_DIR/docker-in-docker/utils.sh
# Finds all 'FROM ${PREPULL_PROJECT}' lines in all Dockerfiles in
↪  the ./app directory, skipping the node_modules subdirectories,
# strips them down to only the image name, and prepulls them.
PREPULL_IMAGES=$(find ./app -type d -name node_modules -prune -o
↪  -type f -name '*Dockerfile*' -print0 | \
    xargs -0 grep --no-filename "FROM ${PREPULL_PROJECT}" | \
    cut -d' ' -f2 | sort -u | sed -E "s/^${PREPULL_REGISTRY}\///" \
)
prepull_if_not_exist $PREPULL_REGISTRY "${PREPULL_IMAGES[@]}"
```

Listing 48: *Prepull CERN exclusive images*

3. starts `socks5` and `sshuttle` proxies and makes their internal docker network IP available through environment variables.

```
echo '==> Starting the proxy container...' >&2
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml up -d sshuttle
# Get the IP address of the `ccs-sshuttle` container in the
↪  `ccstools` network.
export SSHUTTLE_IP=$(docker inspect -f '{{
↪  .NetworkSettings.Networks.ccstools.IPAddress }}' ccs-sshuttle)

# Start the socks-proxy, used during the build time.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml --profile setup up -d socks-proxy
# Set the ALL_PROXY environment variable to use the socks-proxy
↪  container.
export ALL_PROXY='socks5h://socks-proxy:1080'
```

Listing 49: *Start `socks5` and `sshuttle` proxies*

4. Invokes the proper `docker compose` command to start the requested applications from the team's stack.

```
# Start the applications.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml $DEBUGGER_CONFIG --profile app up
↪  -d $COMPOSE_OPTIONS $@
```

Listing 50: *Start requested applications*

5. tears down the `socks5` proxy, since it's required only during build time.

```
# Stop the socks-proxy container after container builds have
↪  completed.
docker compose -f docker-compose.yml -f
↪  docker-compose.teleworking.yml --profile setup rm --force --stop
↪  socks-proxy >&2
```

Listing 51: *Tear down the `socks5` proxy*

4.1.1 Network and Volume Creation

A postgresql official image is used, adjusted to create named databases given in a comma separated list as an environment variable, when it detects that the mounted volume is empty.

```
environment:
    POSTGRES_MULTIPLE_DATABASES:
    ↪  ${FGC_DB_DATABASE},${FORTLOGS_DB_DATABASE} #,${FGC_TEST...
```

Listing 52: *DB compose service environemnt variable*

```
#!/bin/bash
set -e
set -u

function create_user_and_database() {
    local database=$1
    echo "  Creating user and database '$database'"
    psql -v ON_ERROR_STOP=1 --username "$POSTGRES_USER" <<-EOSQL
        CREATE USER $database;
        CREATE DATABASE $database;
        GRANT ALL PRIVILEGES ON DATABASE $database TO $database;
EOSQL
}
if [ -n "$POSTGRES_MULTIPLE_DATABASES" ]; then
    echo "Multiple database creation requested:
    ↪  $POSTGRES_MULTIPLE_DATABASES"
    for db in $(echo $POSTGRES_MULTIPLE_DATABASES | tr ',' ' '); do
        create_user_and_database $db
    done
    echo "Multiple databases created"
fi
```

Listing 53: *Postgres container entrypoint script*

It is also possible to populate the generated databases, for example with a snapshot of data taken from production or preproduction instances, although it requires a bit of manual intervention, especially when it comes to generating the `dumps`. After a dump has been generated, the following script is ran during the startup of the database container, immediately after the previous one shown.

```
psql \
    --dbname="$FGC_DB_DATABASE" \
    --username="$POSTGRES_USER" \
    --no-password \
    </sql-dumps/dbod-fgc-dev-dump.sql
```

Listing 54: *Populate databases script*

Both scripts are configured to run at container startup by piggybacking on `postgres`

images mechanism for configuring database startup, which is visible on listing 55, where the scripts are copied into the container, and are renamed to enforce their run order, based on the previously mentioned mechanism of the `postgres` image, documented in more detail in the official documentation [Pos].

```
FROM postgres:15.1
COPY create-multiple-postgresql-databases.sh
 ↪  /docker-entrypoint-initdb.d/
 ↪  1_create-multiple-postgresql-databases.sh
COPY populate-databases.sh
 ↪  /docker-entrypoint-initdb.d/2_populate-databases.sh
COPY --chown=postgres dbod-fgc-dev-dump.sql
 ↪  /sql-dumps/dbod-fgc-dev-dump.sql
COPY --chown=postgres dbod-fgc-test-manager-dump.sql
 ↪  /sql-dumps/dbod-fgc-test-manager-dump.sql
```

Listing 55: *Setting scripts to be run at database container startup*

4.1.2 Image prepull

For most of the images needed by the team, CERN makes its GPN image registry `https://registry.cern.ch` accessible even from the internet. However, due to licensing issues, the base python image we used, is restricted, and can be only pulled while inside the CERN network causing the `docker compose` command to fail. For that reason, a workaround using `sshuttle` and `docker-in-docker`(DinD) was used, to pull the images before the `docker compose` command invocation if necessary. Docker In Docker is - as the name suggests - a docker container preconfigured with a docker client. By running docker inside a container, we can have full control over its runtime environment and startup conditions, without needing root permissions on the host machine, or altering any behaviour of the host's native docker runtime & client. This allowed us to use `sshuttle` - in the same way its used for the applications' runtimes - as a router, to route all DinD traffic through CERN's GPN, thus being able to pull the image and extract it into the host's native docker client. The implementation of the main logic is given below, with an example usage in listing 48, above.

```
# Starts a dind image proxied through GPN and copies auth
↪  configuration to it.
# Optional first argument is the proxy host
setup_proxied_dind () {
    export PROXY_HOST="${1:-$DEFAULT_PROXY_HOST}"
    docker compose -f docker-compose.yml -f
    ↪  docker-compose.teleworking.yml up -d sshuttle
    export SSHUTTLE_IP=$(docker inspect -f '{{
    ↪  .NetworkSettings.Networks.ccstools.IPAddress }}'
    ↪  ccs-sshuttle)
    docker compose -f docker-compose.yml -f
    ↪  docker-compose.teleworking.yml up -d dind
    ESCAPED_DOCKER_CONFIG=$(sed 's/"/\\"/g' <<<
    ↪  $(generate_docker_config))
    docker exec ccs-dind bash -c "mkdir -p /root/.docker && echo
    ↪  \"$ESCAPED_DOCKER_CONFIG\" > /root/.docker/config.json"
}

# Stops the dind image and the sshuttle container.
tear_down_proxied_dind () {
    # ...
}

# First arg is the registry name, rest are images to pull.
prepull_images () {
    REGISTRY=$1
    # ...
    for image in "${@:2}"; do
    image=$REGISTRY/$image
    echo "==> Pre-Pulling $image into DinD container's docker
    ↪  daemon"
    docker exec -t ccs-dind bash -c "docker pull $image"
    echo "==> Loading $image to local docker daemon..."
    docker exec ccs-dind bash -c "docker save $image" | docker load
    done
}
# First arg is the registry name, rest are images to pull.
prepull_if_not_exist () {
    # ...
    REGISTRY=$1
    if ! images_exist_locally $@; then # Do nothing if all images
    ↪  requested exist locally
        echo "==> Prepulling images from $REGISTRY through proxy..."
        setup_proxied_dind
        prepull_images $@
        tear_down_proxied_dind
    fi
}
```

Listing 56: *DinD image prepull implementation*

4.1.3 Proxying

Both proxy systems, are themselves implemented as extra docker compose services, in the previously mentioned `docker-compose.teleworking.yml`.

sshuttle

The `sshuttle` service ensures that the docker daemon gives the container the necessary capabilities:

```
sshuttle:
build:
    context: ./deployment/local/sshuttle
    dockerfile: Dockerfile
container_name: ccs-sshuttle
profiles:
    - setup
environment:
    USERNAME: ${USERNAME} # ssh user to log in as
    PROXY_HOST: ${PROXY_HOST} # ssh host to log in to
cap_add:
    - NET_ADMIN
sysctls:
    - net.ipv4.ip_forward=1
volumes:
    # host's ssh key is mounted into the container to use for
    ↪  connecting to the proxy target.
    - "./deployment/local/id_ed25519:/root/.ssh/id_ed25519:ro"
command: ["sh", "start.sh"]
```

Listing 57: *Sshuttle proxy service definition*

while the container is a simple python capable container that has `openssh` and `sshuttle` installed, and runs the following on startup, to configure itself as a router:

```
# Allow IP masquerading (i.e., NAT).
iptables -t nat -A POSTROUTING -s 192.168.50.0/24 -o eth0 -j
↪  MASQUERADE
# Accept paceket forwarding (i.e., routing).
iptables -A FORWARD -o eth0 -j ACCEPT

ssh-add /root/.ssh/id_ed25519

sshuttle --dns --listen 0.0.0.0 -x sshuttle:22 --ssh-cmd "ssh -v -i
↪  /root/.ssh/id_ed25519 -o StrictHostKeyChecking=no -o
↪  ForwardAgent=yes" -r ${USERNAME}@${PROXY_HOST} 0/0
```

Listing 58: *Sshuttle container entrypoint*

Socks

The `socks` proxy service simply starts an `openssh` capable container, and invokes the necessary `ssh` command.

```
socks-proxy:
build:
    context: ./deployment/local/socks-proxy
    dockerfile: Dockerfile
container_name: ccs-socks-proxy
profiles:
    - setup
restart: unless-stopped
expose:
    - 1080
volumes:
    - "./deployment/local/id_ed25519:/root/.ssh/id_ed25519:ro"
command: ["ssh", "-NT", "-D", "socks-proxy:1080", "-o",
↪  "StrictHostKeyChecking=no", "-o", "ExitOnForwardFailure=no",
↪  "${USERNAME}@${SOCKS_PROXY_HOST}"]
```

Listing 59: *Socks proxy service definition*

Proxy Client Configuration

Finally using yaml anchors, all of the applications' compose services are extended with the necessary configuration. For the socks proxy, as mentioned in the previous chapter, a simple environment variable is enough for the containers' pip processes to discover and use, whereas for the sshuttle proxy, an entrypoint script is added to the application containers, that sets the sshuttle container as the default route for internet traffic.

```
x-teleworking-base-service: &teleworking-config
    build:
        args:
        # socks proxy used only during build time.
        ALL_PROXY: ${ALL_PROXY-}
    environment:
        # sshuttle proxy used during runtime.
        SSHUTTLE_IP: ${SSHUTTLE_IP:-}
    entrypoint: ["/app/container-entrypoint.teleworking.sh"]
    volumes:
        - ./script/container-entrypoint.teleworking.sh:/app/
        ↪  container-entrypoint.teleworking.sh
# ...
# Extend services
fgc-api: *teleworking-config
fgc-commander-server: *teleworking-python-config
# ...
```

Listing 60: *Service extension for proxy support*

```
# Make the route through the 'sshuttle' container the default route.
ip route replace default via $SSHUTTLE_IP

exec "${@}" # Execute the command passed to the entrypoint, as
↪  normal.
```

Listing 61: *Proxy client container entrypoint*

4.1.4 Application start

In the main `docker compose` command that starts the apps (Listing 50), one can see two extra configuration points used `$DEBUGGER_CONFIG` and `$COMPOSE_OPTIONS`. The latter, is simply used to support passing extra options to the `docker compose` command, such as `--build` and `--force-recreate`, which are often required to force container or service rebuilds during development. The former is to make support for python debugger integration, optional. The `./script/start-local-development.sh` script by default sets the value of `DEBUGGER_CONFIG` to `-f docker-compose.debug.yml`, which overrides some configuration on the services of the applications to add support for python debugging.

```
# ...
fgc-api:
    ports:
        - "127.0.0.1:5679:5678"
    command: ["sh", "-c", "python -m debugpy --log-to-stderr --listen
    ↪  0.0.0.0:5678 -m uvicorn --host 0.0.0.0 --port 8080 --reload
    ↪  server.main:app"]
fgc-commander-server:
    ports:
        - "127.0.0.1:5681:5678"
    command: ["sh", "-c", "python -m debugpy --log-to-stderr --listen
    ↪  0.0.0.0:5678 -m uvicorn --host 0.0.0.0 --port 8080 --reload
    ↪  server.main:app"]
# ...
```

Listing 62: *Docker Compose python debugging support*

In practice, it overrides the command run at application start, to run the application under a debugpy process, which can export a debugging interface through a port to the host machine. This allows for the use of a python debugger, such as `vscode`'s debugger, to attach to the running application, and set breakpoints, inspect variables, and step through the code.

4.1.5 Teardown

Finally the `socks` proxy is torn down, to avoid conflicts with the applications' runtimes and improve performance, as the proxy is only needed during the build time.

4.1.6 Local development script overview

```
./script/start-local-development.sh -h
Usage: start-local-development.sh [OPTIONS] [SERVICE...]

Start the local containerised development

Options:
    -a                          Run all apps.
    -b                          Build services. (--build option
    ↪  for docker-compose)
    -d                          Disable VS Code debugger.
    -h                          Print help.
    -n                          No proxy.
    -o                          Use production operational files
    ↪  instead of development operational files.
    -r                          Recreate services. Useful when you
    ↪  have proxy connection issues or want to clear ad hoc changes
    ↪  to the containers. (--force-recreate option for
    ↪  docker-compose)
```

Listing 63: *Start script help message*

4.2 Cluster Deployment

Having access to the ATS-IT Kubernetes PoC collaboration, we were able to create clusters on command, that came preconfigured with some of the necessary tools:

- **LetsEncrypt:** For automatic SSL certificate generation and renewal.
- **CSI-NFS-Driver:** To be used as backend storage for Persistent Volumes.

4.2.1 ArgoCD

However ArgoCD itself, needed to be bootstraped by us. The process was relatively straightforward, as the cluster was already preconfigured with the necessary tools, and the only thing that needed to be done was to create a namespace for ArgoCD, and install our configured version of ArgoCD, that had added support for CERN's oidc authentication and the previously mentioned `helm-secrets` plugin for decrypting `sops` secrets.

```
# Create the argocd namespace.
kubectl create ns argocd
# Create the secret containing private key for decrypting secrets.
kubectl apply -n argocd -f - <<EOF
apiVersion: v1
kind: Secret
metadata:
    name: helm-secrets-private-keys
type: Opaque
data:
    key.asc: $(base64 -i <path-to-file-with-private-key>)
EOF
# Install ArgoCD.
helm install argocd ./argo-cd  -n argocd
```

Listing 64: *ArgoCD bootstraping*

After that, using the app-of-apps pattern, we can deploy our applications on the cluster, customizing by providing a `values` yaml file, with the target specific options, by deploying an entrypoint Application, which in turn points to an entrypoint chart on the gitlab repo that includes the rest of the Applications we want, as templates. We can also parametrize the deployment to follow non-master branches, by setting the `CCS_REVISION` environment variable.

```
export CCS_REVISION=<revision> export CCS_ENV=<environment> &&
↪  envsubst < entrypoint.yaml | kubectl apply -f -
```

Listing 65: *Entrypoint application command*

```yaml
apiVersion: argoproj.io/v1alpha1
kind: Application
    # ...
    source:
        repoURL: "https://gitlab.cern.ch/ccs/fgc_web.git"
        targetRevision: "${CCS_REVISION}"
        path: deployment/kubernetes/entrypoint
        helm:
        valueFiles:
            - /deployment/kubernetes/.envs/${CCS_ENV}.yaml
            - secrets+gpg-import-kubernetes://argocd/
              ↪ helm-secrets-private-keys#key.asc?../.envs/
              ↪ ${CCS_ENV}-secrets.yaml
        # ...
    destination:
        server: "https://kubernetes.default.svc"
        namespace: argocd
```

Listing 66: *App of Apps* `entrypoint.yaml`

```
entrypoint
├─ values.yaml
├─ templates
│  ├─ ...
│  ├─ ...
│  ├─ fgc-commander.yaml
│  └─ fgc-api.yaml
└─ Chart.yaml
```

Listing 67: *Directory structure of entrypoint chart*

This way, deploying new applications is as simple as commiting a new Application in the `templates` directory above, that points to a new chart, also housed in the repo, and the changes are automatically mapped to the cluster. We can also support disabling apps in specific target environments trough the `<env>.yaml` file for the target environment -again without needing to directly interact with the cluster to apply the changes- by adding a conditional to the relevant `<app-name>.yaml` file in `entrypoint`'s templates.

```
{{ if (dig "fgc-api" "server" "tag" false .Values.apps) -}}
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
    name: fgc-api
    # ...
{{- end -}}
```

Listing 68: *Support for disabling apps in `entrypoint/template/fgc-api.yaml`*

4.2.2 Applications

For the applications themselves, we used one helm chart for each. Most of the resources needed for them where the same across all applications, so we also created and maintained alongside a different CERN team, a common library chart. The chart handled all common resources and was created in a way as to minimize the knowledge of its inner workings required to use it. For every app, three `values` files were used

- **`values.yaml`**: The static global values for all applications across all deployment targets.
- **`.envs/<env>.yaml`**: The values specific to the current deployment target.
- **`.envs/<env>-secrets.yaml`**: The secrets specific to the current deployment target, encrypted with `sops`.

The command mentioned in Listing 65, and specifically the `CCS_ENV` variable is used to choose the `<env>` part of the above `values` files.

Beyond that, most of the deployments are pretty standard, per application backend:

- **`Deployment`**: Handling the main app container.
- **`Service`**: Exposing the app to the cluster.
- **`Ingress`**: Exposing the app to the internet.
- **`ConfigMap & Secret`**: For providing the app with necessary configuration taken from the `values` files.

The web frontends of the applications however required a bit more thought, considering that `Vue.js` frontends in production are simply static files and need to be served by a web server. At the same time, the environment variables are baked into the applications on build time, which would create an $N \times M$ problem, as we would need to build N frontends for M targets. The solution was to use a single `nginx` container, build the frontends on the cluster and place the resulting build artifacts in it on the fly, using a `ConfigMap` to parametrize the build processes, and a shared volume to share the build artifacts between the frontend builders and the `nginx` container. This approach had the added benefit of not needing an extra Argo Application resource per frontend, since they were all handled by the same Application resource, that with a bit of helm templating, generated a new Deployment for every frontend builder.

```
{{- range $name, $details := .Values.apps }} # loop over app keys in
↪  <env>.yaml
{{- if hasKey $details "client" }} # if app key has a 'client' key
{{- $args := dict "Args" (dict "name" $name "details" $details) }}
{{- $context := merge $args $ }}
# Create a deployment for the frontend builder by merging the custom
↪  deployment with the library one.
{{ include "library.util.merge" (list $context
↪  "web-server.client.deployment" "library.deployment") }}
---
{{- end }}
{{- end }}
```

Listing 69: *Web-Server deployment helm templating logic*

4.3 Development and Deployment Automation

4.3.1 Continuous Integration

The Gitlab jobs where structured in a way to remove as much duplication as possible, which was dissapointingly not all, since Gitlab CI syntax is in places poorly documented and in others disallows useful behaviour for technical reasons. The jobs where restricted to only run on the master branch, and both build & deploy jobs where automatically triggered when changes specific to an application where pushed to master, but could also

be run on demand, by pushing a git tag to the repo that follows a specific syntax. There was also consideration for adding lint and test jobs for each application component, however most didn't have extensive -or any at all- test suites, so it was disabled.

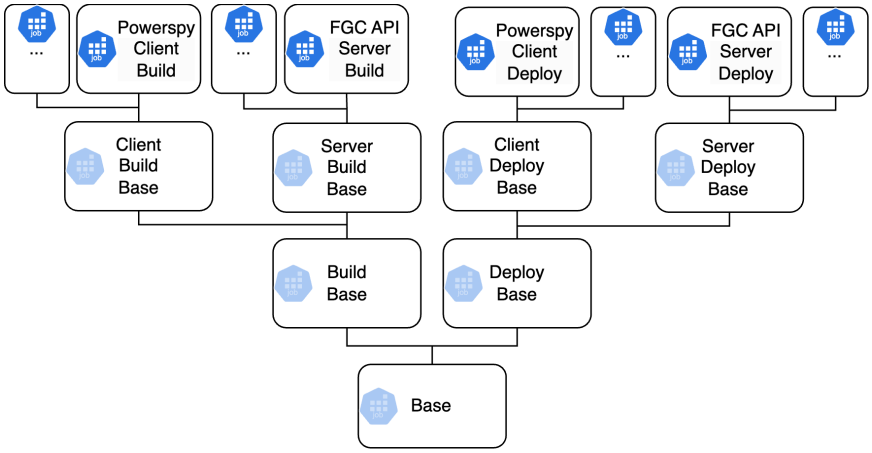


Figure 4.1: *Structure of gitlab jobs*

The build jobs, used each application's `Dockerfile` and handled building and pushing the images to CERN's image registry. Each image is pushed using a couple different tags to make it easier to use and trace from which git commit it was built. Main effort was correctly configuring and extending the base `kaniko` image provided by CERN, and a bit of trial & error to discover how much of the trigger logic gitlab allowed us to abstract away into the base jobs.

4.3.2 Continuous Deployment

We'll start with a description of the first attempt, using ArgoCD-Image-Updater, which even though did not end up being the one we went for, it influenced the final approach and helped more accurately discover the requirements, leading to a far simpler -albeit custom- solution.

ArgoCD Image Updater Attempt

As previously mentioned, due to the team's budget and size, one of the main considerations for most of the design decisions was to avoid custom solutions as much as possible. With that in mind, we -at first- decided to go with ArgoCD's own plugin as a solution for CD, even though it was -and still is- in a pre-release state.

Using ArgoCD-Image-Updater when on is already using ArgoCD is a simple two step process, especially when using helm charts. First we needed to deploy it on the cluster, which in our case could be done by including its official chart in our ArgoCD deployment as a dependency, and configuring it a bit through our `values` file. Then we needed to mark our ArgoCD Application resources with proper annotations through which the update policy of each image can be specified.

```yaml
dependencies:
# ...
- name: argo-cd
    version: 5.36.11
    repository:
    ↪ "oci://registry-tn.
    ↪ cern.ch/common"
# ...
```

Listing 70: *ArgoCD Image Updater installation*

```yaml
kind: Application
metadata:
  annotations:
    argocd-image-updater.
    ↪ argoproj.io/
    ↪ image-list:
    ↪ web-server=<repo>/
    ↪ <image>
```

Listing 71: *ArgoCD Application configuration for Image Updater*

However, due to using a git repo for persisting the image tag updates, and the team having a strong requirements to stay in one single monorepo that includes all application code and cluster configuration, we would have to incur the overhead of cloning the repository on the cluster, for ArgoCD-Image-Updater to update. This would not necessarily be a deal breaker, if Image-Updater had support for shallow clones, or if it could persist the git clone between updates. We spent a little bit of time implementing the latter, and tested it successfully on a cluster, but realized after that the persistent method didn't allow for tracking multiple different clusters using the same branch on the same repo. In addition, the plugin development team was not particularly active and our adjustment, as well as a bug report went unanswered for a long time, well after we had already implemented our alternative solution.

Custom Solution

The deploy jobs used in the custom solution, are a bit more interesting however, since they don't directly cause the deployments themsleves. As previously mentioned the idea

was to depend on Argo's featurset for the synchronization between the repository and the clusters. This simplified the implementation a lot, removing any complexity that comes with configuring communication between different systems -in this case, repo and clusters- manually. As a result the deploy jobs only ended up needing to update a single file in the repo.

That single file is an extra helm `values` file added to the list in 4.2.2 above, named `latest.yaml`. It contains the latest image tag for every container built by the pipelines. In combination with a bit of helm templating logic in the deployments, we can optionally make the deployments track the latest build, for any container on any target, by setting its target specific tag to `'latest'`.

```
# Gets .Vales.apps.fgc-api.server.tag (deployment target tag)
{{- $envTag := (index .Values.apps ”fgc-api” ”server” ”tag”) }}
# Gets .Values.latest.fgc-api.server.tag (latest build tag)
{{- $latestTag := (index .Values.latest ”fgc-api” ”server” ”tag”) }}
# If the target specific tag is latest, use the latest build tag.
{{- $overrideTag := (eq ”latest” $envTag) | ternary $latestTag
↪  $envTag }}
{{- $overrideRepository := (index .Values.apps ”fgc-api” ”server”
↪  ”repository”) }}
{{- $overrideDict := dict ”Values” (dict ”image” (dict ”repository”
↪  $overrideRepository ”tag” $overrideTag) ) }}
{{- $context := merge $overrideDict . -}}
# Instantiate a deployment with the proper tag.
{{- include ”library.deployment” $context -}}
```

Listing 72: *Helm templating logic for optionally tracking latest build*

```
.deploy_base:
  stage: deploy
  # ...
  script:
    - git config user.email "deploy-bot@runner.com"
    - git config user.name "GitLab Deploy Bot"
    # Setup remote using access token from Gitlab CI Variables
    - git remote add gitlab_origin https://
    ↪  oauth2:${CI_GITLAB_ACCESS_TOKEN}@gitlab.cern.ch/ccs/fgc_web
    # Checkout master branch (single source of truth for all
    ↪  targets)
    - git checkout master
    - git pull
    # Update latest tag of the app
    - yq --inplace ".latest.${image_name}.tag =
    ↪  \"${CI_COMMIT_SHORT_SHA}\" | .latest.${image_name}.tag
    ↪  style=\"double\"" deployment/kubernetes/.envs/latest.yaml
    - git add deployment/kubernetes/.envs/latest.yaml
    - COMMIT_MESSAGE="Update ${image_name} latest to
    ↪  ${CI_COMMIT_SHORT_SHA}"
    - git commit -m "${COMMIT_MESSAGE}"
    - git push gitlab_origin master -o ci.skip && break
```

Listing 73: *GitlabCI deploy job*

5 Conclusion

The final chapter summarizes the main parts of the effort and contributions. It provides an assessment of the design and implementation as well as suggests possible extensions and improvements that could be added in the future, some of which are already being worked on.

5.1 Concluding Remarks

The starting mandate of the position was to improve the development and deployment infrastructure, a work that was long overdue. To achieve this, we split the task and faced each part separately, ensuring compatibility between them wherever needed:

- Localized the development environment, making it easier to set up and use, and ensuring that it is consistent across all developers.
- Improved the deployment infrastructure, making it more robust scalable and easy to replicate.
- Improved the development and deployment experience with CI/CD, removing hurdles and complexity between the developers and the deployment enviroments.

This has already made it easier for new team members to join and start contributing to the team's efforts, and for existing members to focus on development rather than troubleshooting or system administration.

During this year+ (14 months) of work, we were often met with obstacles and challenges. Sometimes we would find bugs in infrastructure provided by CERN's internal IT department, such as discovering that some of the cluster's nodes' time was steadily drifting out of sync, causing very confusing behaviours on the application side. Other times we would hit limitations in the tools we had previously chosen, such as ArgoCD-Image-Updater's issues when handling repositories that where too large. And all the while we needed to face issues that where externally introduced or missed in the first place, to ensure the team could continue working without interruptions.

5.2 Future Work

What we ended up creating is a solid foundation for the team to build upon, and a relatively robust way to do so, however it still is a work in progress, supporting a relatively small subset of all the benefits that can be gained from the tools we are now using.

- **Georedundancy:** CERN's internal TN cloud is currently being extending to a second data center in a different area of CERN, where we can claim a few more nodes to further improve the reliability of our services. This has already been tested, and the onboarding of new nodes to an existing cluster is as straightforward as running a command.

- **Logging & Monitoring:** Currently ArgoCD's dashboard is used to monitor the deployments' states. This is relatively basic and ephemeral, so other tools are being looked into, such as Prometheus and Grafana, to provide a more robust and long-lasting solution.

- **Enforce SemVer through CI/CD:** Semantic Versioning is already softly enforced through the utilities that have been created to assist in interacting with the CI/CD pipelines. However, this could be further enforced, by adding logic in the pipelines themselves to block merge requests that have non compliant versioning.

- **Automate creation of new apps and services:** One of the main benefits of moving to a yaml-based, declarative approach to infrastructure, is that they are now easy to handle programmatically. This means it's now possible to automate a lot of

procedures that previously where only documented in README files and docs, and had to be manually followed. This includes the creation of new apps and services, as well as the deployment of existing apps to new environments.

- **Integrate Cilium's gateway API:** The clusters provided by CERN's infrastructure department, are using Cilium as a CNI. This means that we can leverage Cilium's support for the Kubernetes Gateway API, to provide a more robust and secure way to expose services to the outside world and each other.

In the end, what is important is that the team is already actively using the tools we've developed, and they have resulted in immediately visible imrpovement in their experience and ability to produce results. The hope is that it'll soon reach a point where it can be considered feature complete, after the team gains some more confidence in it, and any need for manual intervention is minimized or moved to be the responsibility of the infrastructure team. This would remove any need for an infrastructure engineering role in the team, thus fulfilling the our ultimate goal... making ourselves obsolete.

Bibliography

[Hel] Helm, *Helm Docs*, `https://helm.sh/docs/`, [Online; accessed 8-July-2025].

[jkr] jkroepke, *Helm Secrets ArgoCD Integration Docs*, `https://github.com/jkroepke/helm-secrets/wiki/ArgoCD-Integration`, [Online; accessed 8-July-2025].

[Pos] PostgreSQL, *PostgreSQL Docker Image Docs*, `https://hub.docker.com/_/postgres`, [Online; accessed 8-July-2025].