NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF INDUSTRIAL ELECTRIC DEVICES AND DECISION SYSTEMS

Model-driven adaptation of Function-as-a-Service applications

PhD THESIS

Andreas Ant. Tsagkaropoulos

Athens, April 2022

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DIVISION OF INDUSTRIAL ELECTRIC DEVICES AND DECISION SYSTEMS

Model-driven adaptation of Function-as-a-Service applications (Αναπροσαρμογή Function-as-a-Service εφαρμογών βασισμένη σε μοντέλα)

PhD THESIS

Andreas Ant. Tsagkaropoulos

Advisory Committee: *Gregoris Mentzas,* Professor NTUA (supervisor)
Ioannis Psarras, Professor NTUA
Dimitrios Askounis, Professor NTUA

Approved by the seven-member examination committee on the th of April 2022.

Gregoris Mentzas Professor NTUA	*Ioannis Psarras* Professor NTUA	*Dimitris Askounis* Professor NTUA
Dimitris Apostolou Professor University of Piraeus	*Ioannis Verginadis* Assistant Professor AUEB	*Christos Papatheodorou* Professor NKUA
	Dimitrios Tsoumakos Associate Professor NTUA	

Athens, April 2022

…….
Andreas Ant. Tsagkaropoulos / Ανδρέας Αντ. Τσαγκαρόπουλος
Doctor of Engineering N.T.U.A. / Διδάκτωρ Μηχανικός Ε.Μ.Π.

Abstract

This doctoral dissertation is situated in the research field of FaaS processing, cloud topology adaptation and administration. It consists of an approach suggesting the definition of a cloud and edge processing topology using the TOSCA standard. Also, the presented approach includes a novel way of updating the processing topology based on the criteria set by the DevOps.
Concerning the description of the applications, research on the most important works from the state of the art was carried out, focusing on contemporary generic application description languages. Also, the expressivity of these application description languages is briefly presented, and a comparison of the modelling extensions and approach suggested in this work was made against one of the most important and ubiquitous description languages (Terraform).
Related to the adaptation of applications, research on the available adaptation methods was performed, and a new rule-based methodology suggesting the use of 'Severity' of the situation of a topology is described. Severity values are obtained by factoring in all of the dynamic metric values involved in a violated rule, and based on these a relevant adaptation action is suggested. The exploitation of Severity values using different algorithms (techniques) allows different ways of countering workloads.
In the context of the aforementioned research four software systems were created, of which two are open-sourced. The first is related to the creation of application topology descriptions, leveraging the new TOSCA extensions. The remaining are related to analysis of Severity techniques and the creation of adaptation actions based on Severity techniques, although in two of them additional techniques which are widely used in the industry were also implemented. One of these systems is a simulator.
Severity was proven to allow the definition of techniques leading to improved results, as these were determined by a chosen utility function. The experiments which were carried both in the level of simulations but also in a realistic testbed, indicate the need to make appropriate decisions on the technique which should be used based on the workload which is served. The successful use of Severity in the context of OpenFaas, a well-known and realistic FaaS platform, to perform the adaptation indicates the feasibility of the approach.

Keywords: FaaS applications, Cloud computing, Edge computing, application models, application adaptation, TOSCA, Severity, OpenFaaS

Περίληψη

Η διδακτορική διατριβή τοποθετείται στην περιοχή της προσαρμογής και διαχείρισης υπολογιστικών τοπολογιών νέφους, της ειδικής κατηγορίας εφαρμογών FaaS. Προτείνει μια μεθοδολογία με την οποία μπορεί να γίνει ο ορισμός της τοπολογίας με βάση το TOSCA standard συμπεριλαμβάνοντας και πόρους στο άκρο του δικτύου. Επίσης, η μεθοδολογία συμπεριλαμβάνει μια καινοτόμα προσέγγιση για τη δυναμική ανανέωση της επεξεργαστικής τοπολογίας βάσει κριτηρίων που θέτει ο διαχειριστής της εφαρμογής.
Σχετικά με το καθορισμό της περιγραφής των εφαρμογών, έγινε βιβλιογραφική επισκόπηση των σημαντικότερων προτάσεων που έχουν προταθεί, και βασίζονται σε σύγχρονες γενικές γλώσσες περιγραφής εφαρμογών. Επίσης συνοψίστηκε η εκφραστικότητα της κάθε γλώσσας περιγραφής σε σχέση με την αναλυόμενη περιοχή και πραγματοποιήθηκε σύγκριση με μία από τις σημαντικότερες και πιο ευρέως χρησιμοποιούμενες γλώσσες περιγραφής (Terraform).
Ως προς το ζήτημα της αναπροσαρμογής των εφαρμογών, πραγματοποιήθηκε έρευνα των υπαρχόντων τρόπων αναπροσαρμογής, και προτάθηκε μία νέα μεθοδολογία που βασίζεται σε κανόνες και αξιολογεί τη ‘Σοβαρότητα’ μιας κατάστασης (Severity) προκειμένου να προτείνει αντίστοιχες ενέργειες αναπροσαρμογής. Οι τιμές της ‘Σοβαρότητας’ προκυπτουν λαμβάνοντας υπόψιν όλες τις δυναμικές τιμές των μετρικών που σχετίζονται με τον εκάστοτε κανόνα. Η αξιοποίηση των τιμών της σοβαρότητας με διαφορετικές τεχνικές επιτρέπει διαφορετικούς τρόπους αντιμετώπισης των υπολογιστικών φορτίων.
Πλαισιώνοντας τη παραπάνω έρευνα, δημιουργήθηκαν τέσσερα αυτοτελή υπολογιστικά συστήματα εκ των οποίων τα δύο αποτελούν λογισμικό ανοικτού κώδικα. Το πρώτο από αυτά σχετίζεται με την δημιουργία περιγραφών τοπολογίας εφαρμογών, ενσωματώνοντας τις νέες επεκτάσεις για τη γλώσσα TOSCA. Τα υπόλοιπα τρία αφορούν την πραγματοποίηση αναπροσαρμογής τοπολογιών αξιοποιώντας τεχνικές που βασίζονται πρώτιστα στην έννοια της Σοβαρότητας, ενώ σε δύο από τα συστήματα υποστηρίχθηκαν και άλλοι αλγόριθμοι οι οποίοι χρησιμοποιούνται ευρέως στη παραγωγή σήμερα. Ένα από τα παραπάνω συστήματα επιτρέπει τη διεξαγωγή προσομοιώσεων.
Η νέα μεθοδολογία αναπροσαρμογής δείχτηκε ότι επιτρέπει τον ορισμό τεχνικών που οδηγούν σε καλύτερα αποτελέσματα σε αρκετές περιπτώσεις φορτίων μέσα από αντικειμενική συνάρτηση. Τα πειράματα που πραγματοποιήθηκαν σε επίπεδο προσομοιώσεων αλλά και πραγματικής τοπολογίας αποκαλύπτουν την ανάγκη επιλογής διαφορετικών τεχνικών ανάλογα με το υπολογιστικό φορτίο. Η πειραματική εφαρμογή της μεθοδολογίας αναπροσαρμογής που έγινε αξιοποιώντας μια δημοφιλή, πραγματική πλατφόρμα (OpenFaas) καταδεικνύει το εφικτό της προσέγγισης.

Λέξεις κλειδιά: Εφαρμογές FaaS, Υπολογιστικό νέφος, Υπολογιστική άκρου, μοντέλα εφαρμογών, αναπροσαρμογή τοπολογίας, TOSCA, Σοβαρότητα, OpenFaaS

Acknowledgements

This doctoral dissertation marks the completion of an exciting and thoroughly illuminating period, during which I greatly profited from the postgraduate program of the School of Electrical and Computer Engineering of the National Technical University of Athens.
Had it not been for the interest, guidance and encouragement of Professor Gregoris Mentzas, this work would have not been completed. The opportunities he presented to me, his trust and his willingness to help in the completion of this work were extraordinary, and I am thankful for his encouragement. Moreover, I am thankful to Assistant Professor Ioannis Verginadis, for his continuous help, deep understanding, and encouragement to advance scientifically. I should also not omit to mention Professor Dimitris Apostolou for his valuable comments, his encouragement and his constructive feedback in order to advance and improve this work.
I would also like to thank the other two members of the three-member advisory committee, Professor Ioannis Psarras and Professor Dimitris Askounis, as well as Professor Christos Papatheodorou and Professor Dimitrios Tsoumakos for the honor they have done to me to participate in the seven-member committee for the dissertation.
I would like also to thank all the members of IMU and especially Fotis Paraskevopoulos and Nikos Papageorgiou with whom I have collaborated in research projects and publications. Our cooperation was very important to define and refine aspects of this work.
However, if it had not been for the constant support and guidance of my parents Antonios Tsagkaropoulos and Panagiota Chatzigiannaki, as well as my brothers Efstratios-Evangelos Tsagkaropoulos and Spyridon Tsagkaropoulos, this dissertation would not have been completed. To my family therefore goes a whole-hearted thank you, and a deep wish that I can stand up to their sacrifices.
Finally, and most importantly I would like to thank God for the extremely favorable circumstances in which my work was carried out, and His – visible to the writer – help in carrying out this work.

Table of Contents

Glossary

Term	Explanation
Amazon CDK	Amazon Cloud Development Kit
AMI	Amazon Machine Image
API	Application Programming Interface
AWS	Amazon Web Services
BYON	Bring Your Own Node
CDN	Content Delivery Network
CPU	Central Processing Unit
DevOps	Development and Operations
DNS	Domain Name Service
DSL	Domain-Specific Language
FaaS	Function-as-a-Service
I/O	Input/Output
IDE	Integrated Development Environment
IP	Internet Protocol
OASIS	Organization for the Advancement of Structured Information Standards
OCCI	Open Cloud Computing Interface
PM	Physical Machine
REST	Representational State Transfer
SME	Small to Mid-sized Enterprise
TOSCA	Topology and Orchestration Specification for Cloud Applications
UI	User Interface
UML	Unified Modelling Language
USD	United States Dollars
VM	Virtual Machine
YAML	YAML Ain't Markup Language

1. Εκτεταμένη Ελληνική περίληψη – Extended Greek Abstract

Η υπολογιστική νέφους αποτελεί πλέον ένα αναπόσπαστο τμήμα της επιστήμης των υπολογιστών, αλλά και της καθημερινότητας, όχι μόνο των μεγαλύτερων επιχειρήσεων τεχνολογίας, αλλά και των μικρότερων. Ωστόσο, η χρήση των πόρων που παρέχονται από τα διάφορα υπολογιστικά νέφη εκ μέρους μιας εταιρείας για την εγκατάσταση μιας εφαρμογής νέφους εγκυμονεί τον κίνδυνο του κλειδώματος πελάτη. Το γεγονός αυτό οφείλεται στην πολυπλοκότητα των τεχνολογιών, και στην έλλειψη κοινά αποδεκτών και ευρέως υιοθετημένων προτύπων. Η εισαγωγή νέων υποδειγμάτων εφαρμογών όπως το Function-as-a-Service που προσφέρει ακόμη πιο λεπτομερείς δυνατότητες ανταπόκρισης στο υπολογιστικό φορτίο μιας εφαρμογής επιτείνει το υπάρχον πρόβλημα. Επιπλέον, η χρήση απομακρυσμένων συσκευών στο άκρο του δικτύου και πόρων από άλλους προμηθευτές υπηρεσιών, οδηγεί στην ανάγκη της υποστήριξης δυνατοτήτων βελτιστοποίησης. Προκειμένου να διευκολύνουμε τη δημιουργία εφαρμογών Function-as-a-Service οι οποίες θα μπορούν να διαχειριστούν τα παραπάνω ζητήματα, προτείνουμε την επέκταση της TOSCA. Σύμφωνα με την εργασία των Bergmayr et al. [1], η TOSCA αποτελεί μία από τις δεσπόζουσες γλώσσες μοντελοποίησης εφαρμογών, κυρίως χρησιμοποιώντας το υπολογιστικό νέφος. Υπάρχουν αρκετές [2–4] υλοποιήσεις που βασίζονται στη χρήση της γλώσσας, οι οποίες υποστηρίζονται με τακτικές αναβαθμίσεις.

Αν και δημοφιλής, ωστόσο η ενορχήστρωση (orchestration) εφαρμογών με τη χρήση της TOSCA υπόκειται σε αρκετούς περιορισμούς που αποτρέπουν την απευθείας ενσωμάτωσή της σε ένα υβριδικό περιβάλλον αποτελούμενο από συσκευές στο άκρο του δικτύου και πόρους στο υπολογιστικό νέφος. Πρώτον, η TOSCA δεν έχει κάποιον εγγενή μηχανισμό ο οποίος να επιτρέπει την ιχνηλάτηση των εγκαταστάσεων της εφαρμογής κατά το χρόνο εκτέλεσης λαμβάνοντας υπόψιν τις όποιες αποφάσεις αναπροσαρμογής [5]. Αυτή η σημαντική παράμετρος έχει επιτρέψει σε λύσεις όπως την Terraform (https://www.terraform.io) και το Ansible (https://www.ansible.com) να αναδυθούν και να αποσπάσουν ικανό ποσοστό της αγοράς. Δεύτερον, χρησιμοποιώντας κάποια από τις υλοποιήσεις [2–4] είναι αναγκαίο να οριστούν εκ των προτέρων οι παράμετροι της εφαρμογής, οι πάροχοι νέφους που πρόκειται να

χρησιμοποιηθούν όπως και τα χαρακτηριστικά των εικονικών μηχανών που απαιτούνται. Η ανάγκη να προσδιοριστεί αυτή η πληροφορία περιορίζει τη δυναμικότητα σε εγκαταστάσεις (εφαρμογών) που χρησιμοποιούν αυτές τις πλατφόρμες, η οποία δυναμικότητα είναι ωστόσο ένα αναπόσπαστο στοιχείο των περιβαλλόντων που χρησιμοποιούν συσκευές στο άκρο του δικτύου (edge) ή μικτούς πόρους (fog – πόρους τόσο στο άκρο του δικτύου όσο και στο υπολογιστικό νέφος). Εν τέλει, ακόμη και όταν γίνει με μη αυτόματο τρόπο αναπροσαρμογή της εφαρμογής, η δεν μπορούν να υποστηριχθούν δυνατότητες βελτιστοποίησης. Αποδεικνύεται λοιπόν δυσκολία της συντήρησης ενός μοντέλου μιας ρεαλιστικής εφαρμογής, με δυναμική επεξεργαστική τοπολογία, η οποία πιθανόν να κλιμακώνεται με μη αναμενόμενο τρόπο. Τα δύο προβλήματα που προααναφέρθηκαν είναι ορατά και στη Terraform και στο Ansible, και μετατοπίζουν τις ευθύνες των διαχειριστών των εφαρμογών (DevOps) σε μια προσπάθεια να διατηρήσουν την εύρυθμη λειτουργία μιας τοπολογίας νέφους η οποία εντούτοις πρέπει να επιλύσει αλληλοσυγκρουόμενες απαιτήσεις όπως το χαμηλό κόστος με καλή ποιότητα υπηρεσίας. Περαιτέρω, ένα ακόμη πρόβλημα που επηρεάζει και την TOSCA αλλά και τις τεχνολογίες Terraform, Ansible είναι η αδυναμία μοντελοποίησης μικτών (fog) εφαρμογών ή εφαρμογών ‘άνευ εξυπηρετητή’ (serverless).

Από την εμπειρία που συλλέχθηκε κατά την εργασία για τη πλατφόρμα PrEstoCloud [6], εξάγεται η θέση ότι σε δυναμικά περιβάλλοντα όπως αυτά που προαναφέρθηκαν, είναι άμεση η ανάγκη της ελάφρυνσης των βασικών καθηκόντων που επωμίζεται ο διαχειριστής των εφαρμογών, μεταφέροντάς τα σε κατάλληλο υποστηρικτικό λογισμικό. Το λογισμικό αυτό χρειάζεται να υποστηρίξει το διαχωρισμό των πόρων μεταξύ του χρόνου σχεδίασης και του χρόνου εκτέλεσης. Με το τρόπο αυτό ο διαχειριστής της εφαρμογής θα μπορεί να μοντελοποιήσει τις απαιτήσεις εγκατάστασης της εφαρμογής σε όσο υψηλότερο επίπεδο αφαίρεσης είναι εφικτό, χωρίς να είναι αναγκαίο να ληφθούν συγκεκριμένες αποφάσεις σχετικά με την αρχική εγκατάσταση αλλά και τη μετέπειτα βελτιστοποίηση ή αναπροσαρμογή των εφαρμογών. Για την επίτευξη της ανάπτυξης αυτού του λογισμικού προτείνονται αφενός επεκτάσεις στη γλώσσα TOSCA – που αποτελεί τη κανονική τρέχουσα επιλογή για τη μοντελοποίηση εφαρμογών νέφους – και δεύτερον προτείνεται ένα πειραματικό λογισμικό που

μπορεί να χρησιμοποιήσει τις επεκτάσεις της TOSCA. Η εργασία αυτή εστιάζει σε εφαρμογές FaaS – εφαρμογές νέφους που δεν χρησιμοποιούν εμπορικές πλατφόρμες FaaS αλλά, απεναντίας, υλοποιούν όλη την απαραίτητη λειτουργικότητα που τους χρειάζεται.

Συμπληρώνοντας τη παραπάνω προσπάθεια, στην διατριβή αυτή παρουσιάζεται η έννοια της 'Σοβαρότητας', η οποία οδηγεί σε καλύτερες επιδόσεις για αρκετές περιπτώσεις υπολογιστικών φορτίων. Ο θεωρητικός ορισμός της 'Σοβαρότητας' συνοδεύεται από πρωτότυπο λογισμικό που τον χρησιμοποιεί. Η χρήση αλγορίθμων αναπροσαρμογής με βάση τη Σοβαρότητα επιτρέπει την εξακρίβωση της αιτίας και του αποτελέσματος κάθε πρότασης αναπροσαρμογής της εφαρμογής.

Συνολικά, η εργασία επιζητεί να απαντήσει στα εξής ερευνητικά ερωτήματα:

- Ποιες σημασιολογικές βελτιώσεις πρέπει να γίνουν στην TOSCA προκειμένου να υποστηρίζει μικτές εγκαταστάσεις εφαρμογών;
- Ποια μεθοδολογία πρέπει να ακολουθηθεί προκειμένου να μοντελοποιηθούν εγκαταστάσεις εφαρμογών που υλοποιούν κάποιο υπόδειγμα αποκεντρωμένης εκτέλεσης;
- Πώς μπορούν να ενσωματωθούν στη μοντελοποίηση πλευρές σχετικές με τη βελτιστοποίηση;
- Πώς συγκρίνεται η μεθοδολογία μας για την ελαστικότητα με άλλες μεθοδολογίες που παρέχουν δυνατότητες ελαστικότητας στις εφαρμογές;
- Πώς συγκρίνεται η επίδοση τεχνικών που βασίζονται στη 'Σοβαρότητα' με άλλες προσεγγίσεις που χρησιμοποιούνται από εμπορικά συστήματα, στην ικανοποίηση των μετρικών ποιότητας υπηρεσίας;

Προκειμένου να απαντηθουν τα παραπάνω ερωτήματα, παρουσιάζεται αφενός η επέκταση της YAML TOSCA [7], αφετέρου δε αναλύεται η αναπτυχθείσα προσέγγιση για την αναπροσαρμογή της εφαρμογής.

Στην ενότητα 1.1 παρουσιάζονται περιληπτικά ερευνητικές εργασίες σχετικές με το αντικείμενο της διατριβής. Στις ενότητες 1.2 - 1.4 περιγράφονται περιληπτικά η προσέγγιση που αναπτύχθηκε και οι επεκτάσεις της TOSCA. Στην ενότητα 1.5 περιγράφεται η έννοια της

Σοβαρότητας και στην ενότητα 1.6 τα βασικά πορίσματα των πειραμάτων που έγιναν σε πραγματική επεξεργαστική τοπολογία προκειμένου να αντιμετωπιστούν μεταβαλλόμενα φορτία με την αναπροσαρμογή της τοπολογίας. Στην ενότητα 1.7 περιγράφονται τα κύρια συμπεράσματα και κατακλείεται η ελληνική περίληψη της εργασίας.

1.1.Σχετική εργασία με τη διατριβή

Η γλώσσα TOSCA [7] είναι ένα πρότυπο της OASIS και βασίζεται στον ορισμό εφαρμογών νέφους δια μέσου προτύπων ή σχεδίων. Υπάρχουν αρκετές υλοποιήσεις (ενδεικτικά οι [2–4]), αν και καμία από αυτές δεν υποστηρίζεται επίσημα. Οι λεπτομέρειες που παρέχονται στα σχέδια που δημιουργούνται μέσα από κάποια από τις παραπάνω υλοποιήσεις δίνουν την πλήρη εικόνα μιας εφαρμογής, ωστόσο δυσχεραίνουν τη κατανόηση της δομής της. Για το λόγο αυτό προτείνεται η δημιουργία δύο μοντέλων TOSCA, ενός περισσότερο αφηρημένου που επιτρέπει παρεμβάσεις βελτιστοποίησης, και ενός περισσότερο συγκεκριμένου – που προέρχεται από το πρώτο και είναι σε θέση να κωδικοποιεί τεχνικές λεπτομέρειες για την εφαρμογή.

Και άλλες προσεγγίσεις έχουν αναπτυχθεί για την υποστήριξη της χρήσης FaaS στη TOSCA, όπως οι [8] και [9]. Ωστόσο η διατριβή αυτή εστιάζει σε επεκτάσεις της TOSCA που απευθύνονται σε όσες εφαρμογές ορίζονται με απευθείας χρήση των υπολογιστικών υποδομών και όχι σε εκείνες που χρησιμοποιούν προϋπάρχουσες πλατφόρμες Function-as-a-Service. Επίσης, από τη διατριβή προβλέπεται η χρήση μικτών πόρων - για το συνδυασμό αυτών των παραγόντων δεν έχει ληφθεί πρόνοια. Η CAMEL [10,11] είναι επίσης μια εξελιγμένη γλώσσα περιγραφής εφαρμογών στο cloud, ωστόσο περιγράφει τις εφαρμογές FaaS από τη πλευρά του καταναλωτή (όχι του σχεδιαστή όπως υπονοείται παραπάνω) και δεν έχει ουσιαστική υποστήριξη για υπολογιστική άκρου (edge computing). Εμπορικές προσεγγίσεις όπως της Terraform, του Pulumi ή και τα εργαλεία που οι ίδιοι οι πάροχοι προσφέρουν (Amazon CDK, Azure Resource Manager templates, Cloudformation AWS templates, Google Cloud Deployment Manager templates) καθιστούν πολύ συγκεκριμένες τις λεπτομέρειες της τοπολογίας της εφαρμογής, δυσκολεύοντας i) την γενίκευσή της ii) τη βελτιστοποίησή της.

Επιπλέον, τα εργαλεία των παρόχων χρησιμοποιούνται για τον ορισμό υπηρεσιών μόνο στο πάροχο που τα προσφέρει.

Συμπληρώνοντας την παραπάνω εικόνα, αξίζει να επισημανθεί ότι τα τελευταία χρόνια έχουν ανακοινωθεί διάφορα συστήματα FaaS τα οποία υποστηρίζουν υπολογιστική άκρου. Τα συστήματα αυτά παρέχουν τη δυνατότητα επίτευξης καλύτερων επιδόσεων για τις εφαρμογές που επιζητούν χαμηλούς χρόνους απόκρισης, ή μεγαλύτερη ιδιωτικότητα ή χαμηλότερη ενεργειακή κατανάλωση σε εφαρμογές που το απαιτούν. Αντιπροσωπευτικά συστήματα του είδους είναι το Amazon GreenGrass [12] και το Azure IoT Edge [13].

Από τις ερευνητικές προσπάθειες που έχουν γίνει στο χώρο, αξίζει να αναφερθεί η προσπάθεια των van Lingen et al. [14] οι οποίοι επέκτειναν τη γλώσσα YANG [15] προκειμένου να υποστηρίζει μικτούς κόμβους (και να τους χρησιμοποιεί τόσο στο άκρο του δικτύου όσο και στο νέφος). Η εργασία αυτή επικεντρώνεται στις επεκτασεις της TOSCA η οποία αποτελεί ένα δημοφιλές πρότυπο που υποστηρίζεται από αρκετά εργαλεία αυτοματοποίησης, επομένως η επαναχρησιμοποίησή τους (ή έστω, παραλλαγών τους) θα είναι δυνατή.

Στο πλαίσιο αυτής της εργασίας παρέχεται ένα σύστημα που είναι σε θέση να δημιουργήσει αφηρημένα μοντέλα TOSCA με βάση τις προδιαγραφές που θέτει ο διαχειριστής της εφαρμογής νέφους.

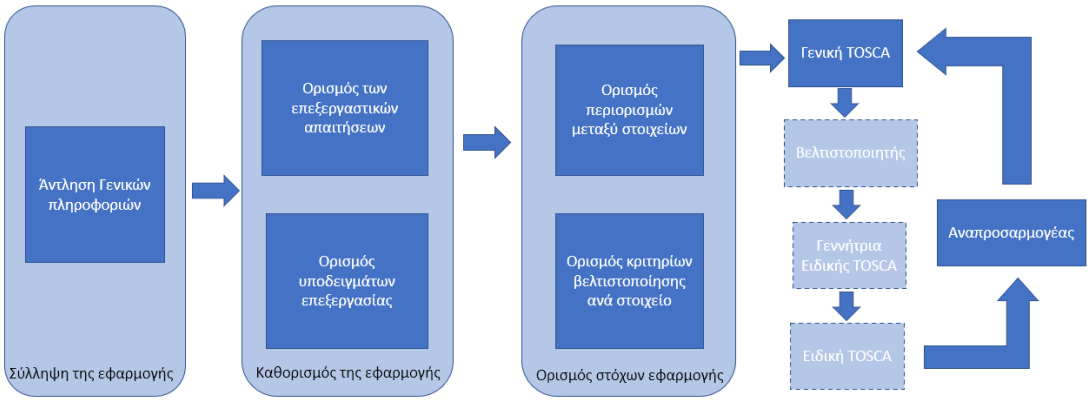
Σε ό,τι αφορά την αναπροσαρμογή μιας εφαρμογής FaaS, παρατίθενται ορισμένες από τις σημαντικότερες κατά τη κρίση του γράφοντος προσεγγίσεις ως προς την αναπροσαρμογή εφαρμογών γενικότερα στο νέφος, καθώς ο τομέας των εφαρμογών FaaS είναι νεότερος και λιγότερο διερευνημένος.

Οι περισσότερες προσεγγίσεις για την αναπροσαρμογή εφαρμογών Cloud βασίζονται είτε στον έλεγχο διαμέσου κανόνων, είτε στη Θεωρία ελέγχου, είτε στη βελτιστοποίηση διαμέσου της αναζήτησης [16]. Οι προσεγγίσεις που βασίζονται στους κανόνες είναι οι απλούστερες και πιο διαισθητικές. Η απόδοσή τους εξαρτάται από την ικανότητα του διαχειριστή της εφαρμογής να ορίζει τις κατάλληλες μεταβλητές προς παρακολούθηση και τα σωστά όρια λειτουργίας. Οι προσεγγίσεις που βασίζονται στη θεωρία ελέγχου (π.χ [17–19]) χαρακτηρίζονται από την αμεσότητα της απόκρισης και το δυναμισμό τους, ωστόσο χρειάζονται προσεκτικό σχεδιασμό

της ανατροφοδότησης προκειμένου το σύστημα να συγκλίνει γρήγορα και να μην οδηγείται σε αστάθεια. Διάφορες προσεγγίσεις έχουν επίσης αναπτυχθεί βασισμένες στη θεωρία των δικτύων αναμονής, όπου το φορτίο και η επεξεργαστική τοπολογία μοντελοποιείται εκ των προτέρων. Στην εργασία [20] σχολιάζεται η ακαμψία αυτών των μοντέλων, διότι όταν αλλάζει η εφαρμογή ή το υπολογιστικό φορτίο, χρειάζεται να επανυπολογιστούν. Άλλες προσεγγίσεις συμπεριλαμβάνουν τους αλγορίθμους αναζήτησης της βέλτιστης λύσης, οι οποίες ακολουθούν μεταξύ άλλων τεχνικές δυναμικού προγραμματισμού, γενετικών αλγορίθμων, ενισχυτικής μάθησης και (ακέραιου) γραμμικού προγραμματισμού. Ο χρόνος ωστόσο που απαιτούν αυτές οι τεχνικές προκειμένου να εκπαιδευτούν, ή/και να εκτελεστούν είναι για ορισμένες εφαρμογές μη αποδεκτός. Επιπλέον, αν ο τρόπος λειτουργίας τους δεν μπορεί να μεταφραστεί σε ισοδύναμους κανόνες (ή να ανιστοιχιστεί σε χρήσιμες διατυπώσεις που θα τον εξηγούν), η μεταφορά της γνώσης σε ένα νέο, παρεμφερές πρόβλημα είναι αδύνατη.

1.2.Η προσέγγιση της διατριβής για την αναπροσαρμογή FaaS εφαρμογών

Στη διατριβή αυτή ακολουθείται μια προσέγγιση βασισμένη σε μοντέλα, και πιο συγκεκριμένα στην γλώσσα TOSCA. Η γενική εικόνα της προσέγγισης φαίνεται στην Εικόνα 1:



Εικόνα 1. Γενική προσέγγιση για τον ορισμό και αναπροσαρμογή εφαρμογών FaaS

Το πρώτο βήμα αφορά τη σύλληψη της εφαρμογής και συμπεριλαμβάνει την άντληση γενικών πληροφοριών που την αφορούν. Σε αυτό καθορίζεται η αρχιτεκτονική της εφαρμόγης καθώς και οι είσοδοι και οι έξοδοι των στοιχείων που θα χρησιμοποιηθούν σε αυτή. Ως ‘στοιχείο’

νοείται κάθε υπολογιστικό πρόγραμμα που μπορεί να εκτελεστεί αυτοτελώς. Επίσης, καθορίζονται λεπτομέρειες σχετικά με τους περιέκτες Docker (containers) που θα χρησιμοποιηθούν για κάθε στοιχείο της εφαρμογής.

Στο δεύτερο βήμα καθορίζονται οι επεξεργαστικές απαιτήσεις της εφαρμογής και τα υποδείγματα επεξεργασίας που πρόκειται να χρησιμοποιηθούν σε αυτή. Μεταξύ των απαιτήσεων συμπεριλαμβάνονται και οι επιθυμητοί επεξεργαστές, η απαιτούμενη μνήμη, και ο αριθμός των στιγμιοτύπων του στοιχείου. Τα υποδείγματα επεξεργασίας μπορεί να είναι ξεχωριστά για κάθε στοιχείο και επιλέγονται μεταξύ FaaS ή none – στη δεύτερη περίπτωση υποθέτουμε ότι ο αριθμός των στοιχείων προσδιορίζεται κατηγορηματικά στην TOSCA και δεν αλλάζει. Ο τρόπος με τον οποίο δίνονται οι απαιτήσεις αυτές στο πρότυπο λογισμικό που έχει αναπτυχθεί προκειμένου να υποστηρίξει τα βήματα της προσέγγισης μέχρι τη δημιουργία της Γενικής TOSCA, είναι είτε με τη χρήση επισημάνσεων σε Java (annotations) είτε με τη χρήση ενός πρότυπου UI.

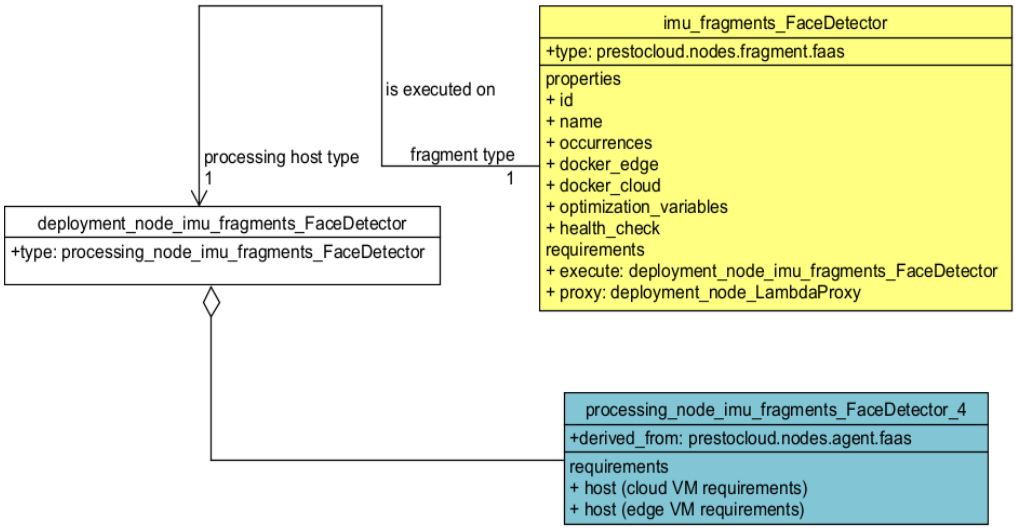
Στο τρίτο βήμα καθορίζονται οι περιορισμοί και οι στόχοι που αφορούν κάθε στοιχείο της εφαρμογής. Οι στόχοι αυτοί μπορούν να προσδιορίζουν απαιτήσεις συνύπαρξης, απομάκρυνσης, χρονικής προτεραιότητας, αλλά και το σχετικό βάρος που καλούνται να έχουν το κόστος, η απόσταση από τους χρήστες και οι προσωπικές προτιμήσεις.

Με βάση όσα προδιαγράφηκαν στα τρία πρώτα βήματα, μία γεννήτρια Γενικής TOSCA (π.χ, [21]) θα πρέπει να δημιουργεί μια γενική προδιαγραφή TOSCA. Στη συνέχεια, αφού επεξεργαστεί τη προδιαγραφή αυτή ένας βελτιστοποιητής θα μπορεί να παραχθεί η Ειδική TOSCA η οποία θα συμπεριλαμβάνει επιπλέον λεπτομέρειες για τη πραγματική εγκατάσταση της εφαρμογής – κατ'ελάχιστο τους χρησιμοποιούμενους παρόχους υπηρεσιών, τη τοποθεσία (άκρο του δικτύου ή νέφος) την υπολογιστική ζώνη (του κάθε παρόχου) και το είδος των εικονικών πόρων που πρόκειται να χρησιμοποιηθούν. Η Ειδική TOSCA ή πληροφορίες που βρίσκονται σε αυτήν χρησιμοποιούνται ως είσοδος από τον αναπροσαρμογέα προκειμένου να παράξει πληροφορία με την οποία να δημιουργείται το νέο μοντέλο Γενικής TOSCA.

1.3.Επεκτάσεις στη TOSCA για την υποστήριξη FaaS εφαρμογών

Προκειμένου να δοθεί η δυνατότητα της περιγραφής μικτών (FaaS) εφαρμογών στη TOSCA, χρειάζεται αφενός να περιγραφεί ο τρόπος με τον οποίο θα γίνεται ο διαχωρισμός των πόρων στο άκρο του δικτύου και στο νέφος (αλλά ταυτόχρονα θα είναι δυνατό να γίνεται η αναπροσαρμογή τους ως ένα στοιχείο της εφαρμογής) και αφετέρου να περιγραφεί ο τρόπος με τον οποίο θα αναπαρίσταται η αρχιτεκτονική μιας εφαρμογής που θα χρησιμοποιεί στοιχεία FaaS.

Ως προς τη πρώτη ανάγκη, υποστηρίζεται καταρχήν η ανάγκη να αποσυνδέεται η προδιαγραφή του εκάστοτε στοιχείου της εφαρμογής από την επεξεργαστική τοπολογία. Για το λόγο αυτό διαιρούμε τις δομές της TOSCA που αφορούν τη προδιαγραφή των στοιχείων σε κόμβους επεξεργασίας (processing nodes), κόμβους στοιχείων(fragment nodes) και κόμβους αντιστοίχησης (mapping/deployment nodes). Ένα παράδειγμα συσχέτισης αυτών των κόμβων δίνεται στην Εικόνα 2.



Εικόνα 2. Η σχέση μεταξύ κόμβων στοιχείων (κίτρινο) με κόμβους επεξεργασίας (μπλε) και κόμβους αντιστοίχισης (άσπρο)

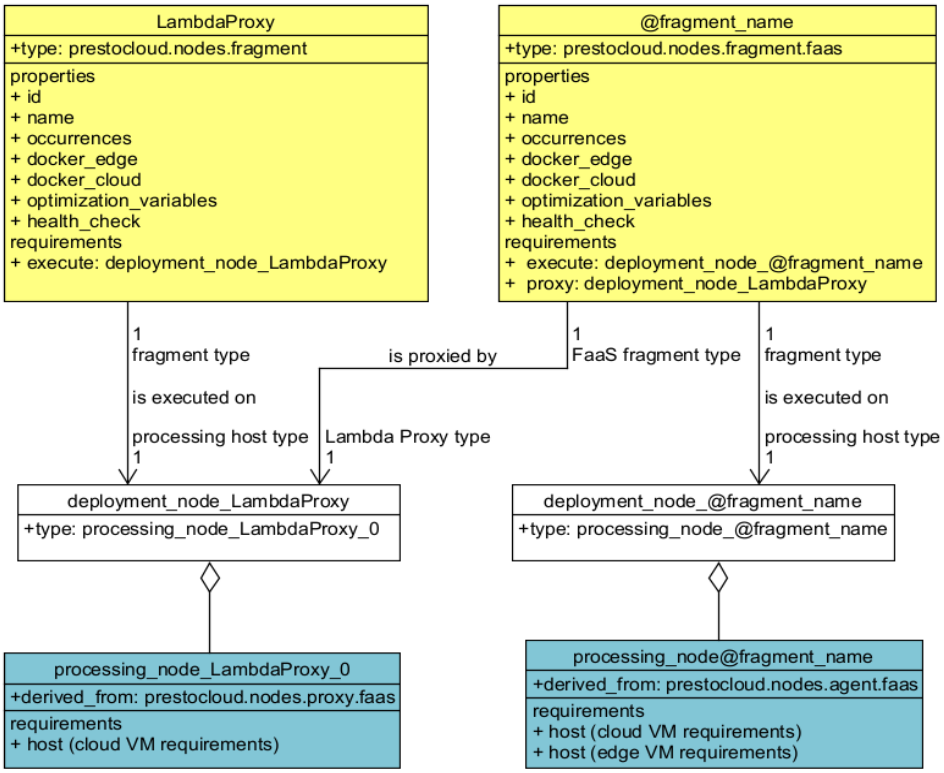
Οι κόμβοι στοιχείων περιλαμβάνουν όλες τις λεπτομέρειες που χρειάζονται για τη σωστή εκτέλεση ενός στοιχείου. Οι κόμβοι επεξεργασίας δίνουν τις τεχνικές προδιαγραφές που πρέπει να ικανοποιούν οι πόροι (εικονικές μηχανές στο παράδειγμα) που πρόκειται να χρησιμοποιηθούν είτε στο νέφος είτε στο άκρο του δικτύου. Τέλος, οι κόμβοι αντιστοίχισης συνδέουν τους κόμβους επεξεργασίας με τους κόμβους στοιχείων.

Στη περίπτωση των κόμβων στοιχείων, η πλειονότητα των πεδίων που εισάγονται αφορούν τη παροχή λεπτομερειών για το περιέκτη Docker του εκάστοτε στοιχείου. Καθώς μπορούν να χρησιμοποιηθούν διαφορετικές εικόνες Docker για την εκτέλεση σε διαφορετικές τοποθεσίες, όπως επίσης να οριστούν και διαφορετικές μεταβλητές περιβάλλοντος, παρέχεται η δυνατότητα σε όποιον μοντελοποιεί το σύστημα να προσαρμόσει τη συμπεριφορά του ανάλογα με τους πόρους και τις δυνατότητες επεξεργασίας που αυτοί έχουν. Ακόμη, οι κόμβοι στοιχείων περιέχουν τις προδιαγραφές για τη βελτιστοποίηση οι οποίες χρειάζεται να φτάσουν στο βελτιστοποιητή.

Στη περίπτωση των κόμβων επεξεργασίας, παρέχεται η δυνατότητα να οριστούν διαφορετικές προδιαγραφές για την εκτέλεση της εφαρμογής στο νέφος, και διαφορετικές προδιαγραφές για την εκτέλεση στο άκρο του δικτύου. Με αυτό το τρόπο μπορεί να επιτευχθεί σε ορισμένες περιπτώσεις – για παράδειγμα όπου υπάρχει η δυνατότητα παραλληλίας – μια εξισορρόπηση των πόρων προκειμένου να υπάρχει παρόμοια απόδοση στο νέφος και στο άκρο του δικτύου. Στο παράδειγμα της παραλληλίας που δόθηκε προηγουμένως, αν μια συσκευή στο άκρο του δικτύου έχει ένα ασθενέστερο υπολογιστικά επεξεργαστή από μια εικονική μηχανή, μπορούν να απαιτηθούν περισσότεροι πυρήνες όταν η εκτέλεση γίνεται στο άκρο του δικτύου.

Αναφορικά με το δεύτερο ερώτημα που τέθηκε στην εισαγωγή αυτού του κεφαλαίου, χρειάζεται να δοθεί ο τρόπος ορισμού των εφαρμογών FaaS. Κάθε εφαρμογή FaaS θεωρούμε ότι αποτελείται από ένα ή περισσότερα στοιχεία FaaS, το καθένα εκ των οποίων έχει το δικό του αυτόνομο τρόπο λειτουργίας. Η πρόσβαση σε κάποιο στοιχείο FaaS θεωρούμε ότι γίνεται μέσα από τη χρήση ενός τμήματος λογισμικού στο οποίο αναφερόμαστε ως Lambda-πληρεξούσιο (Lambda proxy) – από το όνομα της υπηρεσίας FaaS της Amazon και την έννοια του πληρεξούσιου στον πραγματικό κόσμο ο οποίος αναλαμβάνει να διεκπεραιώνει υποθέσεις στο όνομα κάποιου τρίτου.

Η συσχέτιση του Lambda πληρεξούσιου με κάποιο στοιχείο της εφαρμογής απεικονίζεται στην Εικόνα 3:



Εικόνα 3. Οι σχέσεις μεταξύ στοιχείου FaaS και Lambda-πληρεξούσιου (Lambda proxy)

Όπως φαίνεται στη παραπάνω εικόνα, ένα στοιχείο FaaS (τρία δεξιά κουτιά) συσχετίζεται με την απαίτηση proxy με τον Lambda-πληρεξούσιό του. Αυτό συμβαίνει με τη δήλωση της απαίτησης (requirement) *proxy* στο κόμβο στοιχείου, που το συσχετίζει με το κόμβο αντιστοίχισης του Lambda-πληρεξούσιου.

1.4.Επεκτάσεις στη TOSCA για τη βελτιστοποίηση FaaS εφαρμογών

Ήδη η υφιστάμενη προδιαγραφή της γλώσσας TOSCA [7] επιτρέπει τον ορισμό μεταδεδομένων στο αρχείο TOSCA. Προτείνεται λοιπόν η επέκταση αυτής της ήδη υπάρχουσας δυνατότητας προκειμένου να δίνεται η δυνατότητα να καθορίζονται γενικές προδιαγραφές για όλη την εφαρμογή – τις βασικές προτιμήσεις της εφαρμογής (π.χ περιορισμός κόστους), τους παρόχους που πρέπει να χρησιμοποιηθούν ή πρέπει να αποκλειστούν και τους οικονομικούς περιορισμούς που χρειάζεται να τηρηθούν. Οι γενικοί αυτοί περιορισμοί θα μπορούν να χρησιμοποιηθούν από το βελτιστοποιητή προκειμένου να αποφασίσει για τους πόρους και την ενδεδειγμένη τοποθεσία εκτέλεσης των στοιχείων.

Παράλληλα με τους γενικούς περιορισμούς, συστήνεται και ο προσδιορισμός ειδικών περιορισμών για κάθε στοιχείο. Οι περιορισμοί αυτοί μπορούν να αφορούν απαιτήσεις για την συνύπαρξη στον ίδιο πάροχο (τουλάχιστον) δύο στοιχείων, απαιτήσεις για την απομάκρυνση μεταξύ τους (τουλάχιστον) δύο στοιχείων και την εκτέλεσή τους σε άλλους παρόχους, και απαιτήσεις χρονικής προτεραιότητας (που θα ρυθμίζουν τη σειρά έναυσης των στοιχείων). Επίσης, μπορούν να οριστούν και απαιτήσεις εξαίρεσης συγκεκριμένων συσκευών (ή πόρων, γενικότερα) από την εκτέλεση, λόγω της παλαιότερης μη ικανοποιητικής τους απόδοσης.

Επιπλέον, το κάθε στοιχείο μπορεί να ορίζει και βάρη στις ειδικές προτιμήσεις που θα επηρεάζουν τη τοποθέτησή του - στο κόστος, στην απόσταση (των οντοτήτων που αλληλεπιδρούν με ένα στοιχείο, από το στοιχείο αυτό) και στις ανεξάρτητες προτιμήσεις (που μπορούν να εκφράζουν τη προτίμηση ενός παρόχου νέφους έναντι άλλων για οποιονδήποτε λόγο – π.χ επειδή βρίσκεται κοντά στη παραγωγή των δεδομένων). Τα κριτήρια αυτά είναι ενδεικτικά, και μπορούν να προστεθούν και άλλα ή να αντικατασταθούν.

Ο βελτιστοποιητής θα μπορεί να λαμβάνει όλα αυτά τα κριτήρια υπόψιν του (εφαρμόζοντας πρώτα τους περιορισμούς και στη συνέχεια επιλέγοντας να δημιουργήσει μια υπερέχουσα – έναντι των άλλων πιθανών, βάσει των ειδικών προτιμήσεων – τοπολογία).

1.5. Ελαστικότητα εφαρμογών FaaS χρησιμοποιώντας την έννοια της Σοβαρότητας

Όπως αναφέρθηκε παραπάνω, ο αναπροσαρμογέας είναι ένα αναπόσπαστο τμήμα της προσέγγισής μας καθώς παράγει τον αριθμό εικονικών μηχανών ή περιεκτών που πρέπει να προστεθούν ή να αφαιρεθούν για κάθε στοιχείο για την ανανεωμένη Γενική TOSCA. Για να το πετύχει αυτό βασίζεται στη χρήση κανόνων ελαστικότητας. Ορίζουμε τους κανόνες ελαστικότητας ως οδηγίες που αφενός δείχνουν τα επιτρεπτά όρια κανονικής λειτουργίας για μια εφαρμογή, και αφετέρου περιέχουν την ενέργεια οριζόντιας ελαστικότητας που είναι αναγκαία για να καλυφθούν οι ανάγκες της εφαρμογής, προσθέτοντας ή αφαιρώντας πόρους.

Σε αντίθεση με τη συνηθέστερη όμως χρήση των κανόνων ελαστικότητας, προτείνουμε τον απλό ορισμό των ορίων λειτουργίας και την αυτόματη απόφαση για το μέγεθος της αναπροσαρμογής (η οποία συνίσταται στη προσθήκη ή την αφαίρεση επεξεργαστικών

κόμβων). Η απόφαση αυτή λαμβάνεται με βάση τη 'Σοβαρότητα' μιας κατάστασης εκτός των ορίων λειτουργίας. Ορίζουμε τη Σοβαρότητα στην Εξίσωση 1.

$$Σοβαρότητα(\boldsymbol{V}_{παραβίασης}) = \sqrt{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} w_i \cdot (Κανονικοποιημένο(v_i))^2}$$

Εξίσωση 1. Ορισμός της Σοβαρότητας

Στη παραπάνω εξίσωση v_i είναι οι τιμές των επιμέρους μετρικών (κατανάλωσης CPU, μνήμης κλπ.) του διανύσματος τιμών $\boldsymbol{V}$ που παραβιάζει την επιθυμητή ποιότητα υπηρεσίας, ενώ w_i είναι τα βάρη που θέλουμε να αποδώσουμε σε κάθε μετρική.

Ορίζουμε τις κανονικοποιημένες τιμές των v_i με τις παρακάτω εξισώσεις (για κανόνες μεγαλύτερου-από και μικρότερου-από αντίστοιχα)

$$Κανονικοποιημένο(v_i) = \frac{απόλυτη\_τιμή(v_i - t_i)}{απόλυτη\_τιμή(μέγιστο(μετρική_i) - t_i)}$$

Εξίσωση 2. Κανονικοποιημένη τιμή στη περίπτωση κανόνα 'μεγαλύτερου-από'

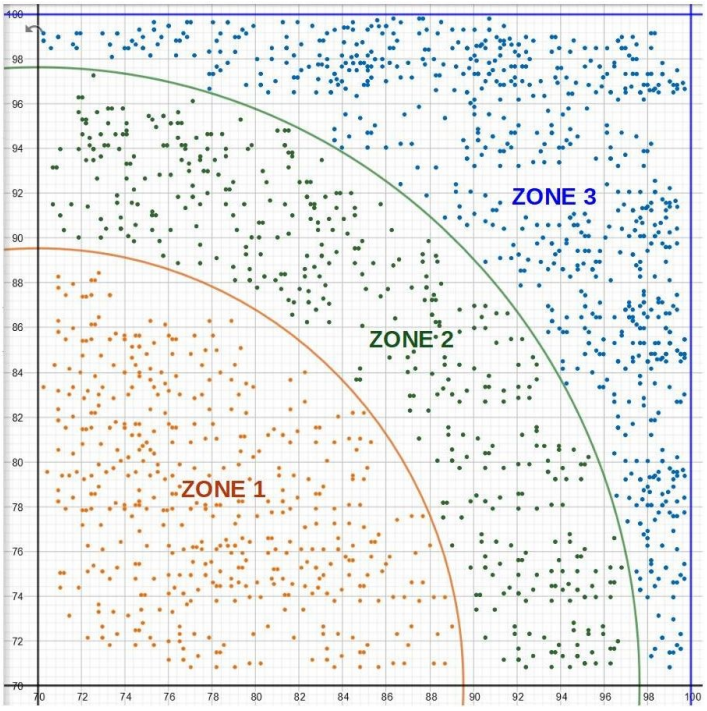
$$Κανονικοποιημένο(v_i) = \frac{απόλυτη\_τιμή(v_i - t_i)}{απόλυτη\_τιμή(t_i - ελάχιστο(μετρική_i))}$$

Εξίσωση 3. Κανονικοποιημένη τιμή στη περίπτωση κανόνα 'μικρότερου-από'

Στη περίπτωση χρήσης μετρικών που δεν έχουν εγγενώς περιορισμό στις τιμές κάποιου (ή και των δύο) άκρων λειτουργίας – όπως για παράδειγμα ο χρόνος απόκρισης δεν έχει άνω φράγμα – προτείνεται είτε η χρήση του 90ου εκατοστημορίου των τιμών, είτε η χρήση του τύπου του Chebyshev για το προσδιορισμό των τιμών που βρίσκονται μέσα σε ένα επιθυμητό ποσοστό, είτε ο απευθείας προσδιορισμός μιας τιμής κάτω ή άνω της οποίας όλες οι τιμές θεωρούνται 'ελάχιστες' ή 'μέγιστες', αντίστοιχα.

Με βάση τη τιμή της σοβαρότητας, μπορούν να δημιουργηθούν διάφορες τεχνικές αναπροσαρμογής οι οποίες θα προσθέτουν ένα σταθερό ή μεταβλητό πλήθος πόρων. Μία από τις τεχνικές που πέτυχαν τα καλύτερα αποτελέσματα στη πειραματική τους επαλήθευση είναι η ΑΠΣ (απλή - με περιοχές σοβαρότητας) η οποία χωρίζει το διανυσματικό χώρο στον οποίο εντάσσονται οι πιθανές τιμές σε ένα προσαρμόσιμο αριθμό περιοχών. Στη περίπτωση του

κανόνα «Αν η κατανάλωση CPU>70% και η κατανάλωση μνήμης >70% τότε πρόσθεσε επεξεργαστικούς πόρους», χρησιμοποιούνται οι περιοχές που φαίνονται στην Εικόνα 4.



Εικόνα 4. Περιοχές Σοβαρότητας (με τρεις περιοχές και δύο μετρικές)

Στη μέθοδο αναπροσαρμογής ΑΠΣ όσες καταστάσεις βρίσκονται εντός της ζώνης 1 οδηγούν στη προσθήκη ή αφαίρεση ενός επεξεργαστικού κόμβου, όσες βρίσκονται στη ζώνη 2 δύο κόμβων και όσες βρίσκονται στη ζώνη 3 τριών κόμβων. Η αντιστοιχία ζώνης με αριθμό κόμβων είναι προσαρμόσιμη αν αυτό είναι επιθυμητό.

Στο πλαίσιο της πειραματικής επαλήθευσης της Σοβαρότητας, συγκρίθηκε η απόδοση της τεχνικής αυτής με δύο άλλους ευρέως γνωστούς αλγορίθμους, τη μεταβολή των επεξεργαστικών κόμβων με προσθήκη ή αφαίρεση ενός σταθερού αριθμού (μέθοδος MK - 4 κόμβοι προστίθενταν/αφαιρούνταν) και τη μεταβολή των επεξεργαστικών κόμβων με τη τεχνική μέγιστης μετρικής (ΜΜ), εμπνευσμένη από τον αλγόριθμό HPA του Kubernetes. Ο νέος αριθμός κόμβων δίνεται στην Εξίσωση 4.

$$ΝέοιΚόμβοι = ΤρέχοντεςΚόμβοι \cdot \frac{μέγιστη\_μετρική}{όριο\_μέγιστης\_μετρικής}$$

Εξίσωση 4. Νέος αριθμός κόμβων σύμφωνα με τη τεχνική Μέγιστης μετρικής

Η μέγιστη μετρική ορίζεται ως η μετρική εκείνη που αποκλίνει περισσότερο από όλες από το όριο λειτουργίας της, και το όριο μέγιστης μετρικής είναι το όριο λειτουργίας που έχει οριστεί για αυτή τη μετρική.

1.6.Εκτίμηση τεχνικών αναπροσαρμογής σε συνθήκες πραγματικού φορτίου

Για την εκτίμηση των τεχνικών αναπροσαρμογής σε συνθήκες πραγματικού φορτίου δημιουργήθηκε εγκατάσταση της πλατφόρμας Faas OpenFaas σε Kubernetes, σε υπολογιστική υποδομή από 3 εικονικές μηχανές μοιρασμένου επεξεργαστή χαμηλών προδιαγραφών (1vcpu, 1GB ram), 1 εικονική μηχανή μοιρασμένου επεξεργαστή υψηλών προδιαγραφών (4vcpu, 8GB RAM) – αυτές οι 4 εικονικές μηχανές τοποθετήθηκαν στο δημόσιο cloud DigitalOcean Φραγκφούρτης – και 1 εικονική μηχανή μέσων προδιαγραφών (1vcpu, 4GB RAM). Σε αυτούς τους πόρους δημιουργούνταν περιέκτες με όριο χρήσης 0.2vcpu και 120Mib RAM, που ο καθένας βασικά εκτελούσε κλήσεις προς μια συνάρτηση υπολογισμού πρώτων αριθμών (η οποία παράλληλα δέσμευε και μνήμη). Οι κλήσεις προς τους περιέκτες γίνονταν με χρήση του λογισμικού Locust, ενώ χρησιμοποιήθηκαν έξι καμπύλες φορτίου. Για την δημιουργία και αποδέσμευση των περιεκτών χρησιμοποιήθηκαν δύο κανόνες λειτουργίας - ο πρώτος προέβλεπε τη χρήση μέχρι 10% του επεξεργαστή (από το 20% που ήταν διαθέσιμο σε κάθε περιέκτη) και μέχρι 10 αιτήματα το δευτερόλεπτο, και ο δεύτερος προέβλεπε τη χρήση τουλάχιστον 5% του επεξεργαστή και τουλάχιστον ενός αιτήματος το δευτερόλεπτο. Στη περίπτωση που παραβιαζόταν ο πρώτος κανόνας η τοπολογία επεκτεινόταν με τη δημιουργία νέων περιεκτών, ενώ στη περίπτωση που παραβιαζόταν ο δεύτερος κανόνας η τοπολογία συστελλόταν με τη κατάργηση περιεκτών.

Κατά τη πειραματική αξιολόγηση εξετάστηκαν η ΑΠΣ (SSZ), η ΜΜ (MACL) και η ΜΚ (ST). Για την αξιολόγηση των μεθόδων χρησιμοποιήθηκε συνάρτηση χρησιμότητας, η οποία ορίζεται με βάση το άθροισμα τον αριθμό των επιτυχημένων αιτημάτων, των αποτυχημένων αιτημάτων, το μέσο χρόνο απόκρισης του συστήματος, τον αριθμό των επιπλέον περιεκτο-δευτερολέπτων, το συνολικό αριθμό περιεκτο-δευτερολέπτων και το συνολικό αριθμό αναπροσαρμογών. Η

συνάρτηση χρησιμότητας έδινε κατά πρώτο λόγο βάρος στην απόδοση και κατά δεύτερο λόγο στην χαμηλή χρήση περιεκτών.

Όπως φαίνεται από τη τελευταία στήλη στους πίνακες Table 19 - Table 34, η τεχνική ΑΠΣ (SSZ στους πίνακες) παράγει εν γένει αποτελέσματα τα οποία είναι καλύτερα από τις άλλες μεθόδους που εξετάστηκαν.

Συνολικά η τεχνική ΑΠΣ επιτρέπει για την επιλεγμένη συνάρτηση χρησιμότητας καλύτερη απόδοση. Στις περιπτώσεις που δεν έχει τη καλύτερη απόδοση, αυτή μπορεί να βελτιωθεί είτε με την αλλαγή του χρονικού ορίζοντα τον οποίο χρησιμοποιεί η μέθοδος, είτε με την αλλαγή των ορίων χρήσης. Ουσιαστική βελτίωση της μεθόδου ΑΠΣ παρατηρήθηκε ότι μπορεί να επιτευχθεί με την προσαρμογή των βαρών της. Συγκεκριμένα, θέτοντας στον ορισμό της Σοβαρότητας βάρος ίσο με 1 στο κριτήριο των αιτημάτων ανά δευτερόλεπτο και 0 στη κατανάλωση του επεξεργαστή και της μνήμης, επιτυγχάνεται σημαντική βελτίωση και στις επιδόσεις χωρίς να επιβαρύνεται δυσανάλογα η κατανάλωση σύμφωνα με την επιλεχθείσα συνάρτηση χρησιμότητας.

1.7.Συμπεράσματα

Στις προηγούμενες ενότητες παρουσιάστηκε περιληπτικά η προτεινόμενη προσέγγιση για τη περιγραφή και την αναπροσαρμογή επεξεργαστικών τοπολογιών FaaS με τη χρήση μοντέλων. Η χρήση της γλώσσας TOSCA επιτρέπει στη προσέγγιση που αναπτύχθηκε να αξιοποιεί την ερευνητική εργασία που πραγματοποιείται στη πιο δημοφιλές πρότυπο για τη περιγραφή εφαρμογών Cloud. Ωστόσο, η προσέγγιση δεν περιορίζεται στη περιγραφή μόνο πόρων στο Cloud, αλλά προτείνει παράλληλα και τη περιγραφή συσκευών στο άκρο του δικτύου, και τη περιγραφή των παραμέτρων βελτιστοποίησης που αντιστοιχούν στη κάθε εφαρμογή. Επιπλεόν, παρουσιάζεται και η έννοια της Σοβαρότητας, που επεκτείνει τους απλούς κανόνες προκειμένου να λαμβάνεται υπόψιν η πληροφορία από τις διάφορες μετρικές αναπροσαρμογής που περιλαμβάνονται σε ένα κανόνα. Τεχνικές που βασίζονται στη σοβαρότητα όπως η ΑΠΣ

βελτιώνουν την συνάρτηση χρησιμότητας που χρησιμοποιήθηκε και μπορούν να επωφεληθούν και από πρακτικές μηχανικής μάθησης. Με την υλοποίηση πρότυπων συστημάτων (της γεννήτριας Γενικής TOSCA και του Αναπροσαρμογέα) αλλά και τη προδιαγραφή των λειτουργιών των άλλων συστημάτων, καταδεικνύεται ο ρεαλισμός της προσέγγισης.

2. Introduction

Cloud computing has become a widely valued commodity with constantly increasing popularity among large and small to mid-sized enterprises (SMEs), as well as public organizations. According to the report by Gartner, it is estimated that the value of public cloud spending will increase about 22%, surpassing $480 billion in 2022 [22]. Furthermore, the infrastructure-as-a-service segment of cloud computing is forecasted to experience the highest growth.

As more applications make use of the cloud and more cloud providers appear, “vendor lock-in” becomes an increasingly important issue. The lack of common standards and the heterogeneity of cloud provider solutions put at risk the portability of data and applications, as moving to a technology supported by a different provider may be associated with high costs [23]. The need for a common and interoperable standard is further augmented due to the appearance of new trends and paradigms, such as Functions-as-a-Service (FaaS). The Functions-as-a-Service (or “serverless”) paradigm involves the use of computing resources in a fine-grained manner, while also reducing the maintenance overhead which is typically associated with cloud applications. A serverless computing platform is defined in [24] as “a platform that hides server usage from developers and runs code on-demand, automatically scaled and billed only for the time the code is running”. Serverless computing has given rise to numerous function-as-a-service (FaaS) platforms, some of which have been successfully coupled with deployments on edge resources, e.g., OpenWhisk [25] and OpenWhisk Lean [26]. To correctly fulfill the promises which it brings however, serverless requires proper modelling and deployment capabilities.

Based on the review by Bergmayr et al. [1], one of the most prominent modelling languages for handling principally cloud deployments is TOSCA (Topology and Orchestration Specification for Cloud Applications) [7], which is an Organization for the Advancement of Structured Information Standards (OASIS) standard that sets the tone for all other cloud modelling languages that try to be compatible with it. There are numerous [2–4] implementations based on this standard that have been regularly maintained.

Notwithstanding the efforts towards its adoption, TOSCA orchestration is impeded by several limitations preventing its direct usage within a hybrid cloud and edge environment. Firstly, TOSCA lacks a native mechanism to track run time deployments and factor-in any reconfiguration decisions [5]. For example, TOSCA's most prominent implementations Alien4Cloud [2], Cloudify [3], and OpenTOSCA [4] focus on the design of the application topology—the modification of the initial template requires the manual intervention of Development and Operations (DevOps) or else risks a discrepancy between the model and the real deployment. Especially due to this first issue, the adoption of TOSCA to address the needs of cloud application deployments in production has been met by solutions such as HashiCorp's Terraform[1] and Red Hat's Ansible[2]. Secondly, a TOSCA template created based on the approaches in [2–4] is required to explicitly define the properties of the application and configure the usage of different cloud providers, along with the characteristics of the VMs that are needed. The need to specify this information limits the dynamicity in deployments using these platforms, which is a prerequisite in edge and fog environments. Finally, even when the application is manually reconfigured, optimization cannot be supported [1]. Thus, it becomes difficult to maintain a model of a real-world application featuring a dynamic topology, which may not scale predictably. These last two issues are also evident in the Terraform and Ansible solutions. These open-source platforms use declarative and pre-defined run time configuration languages to state the desired final state of the cloud application deployment. As a consequence, this channels the responsibilities of DevOps engineers towards maintaining a cloud application topology that must address contradicting requirements, such as low cost and high quality of service. Therefore, such solutions may be popular at the moment since they provide a straightforward way of deploying applications in several cloud vendors, however they still lack real and automatic cross-cloud optimization capabilities. Additionally, their ability to cope with new paradigms, such as edge computing and other distributed execution approaches, may rely on tedious procedures that will result in bespoke solutions that may endanger their portability.

[1] https://www.terraform.io

[2] https://www.ansible.com

Especially when considering the available TOSCA specification [7], it is known that it is very extensible but also quite generic and lacking explicit modelling artefacts for fog applications or serverless approaches. Thus, any custom solutions that are developed pose a barrier to wider adoption of a reference TOSCA-based methodology to handle basic concepts in these fields.The existence of a reference TOSCA-based methodology on the other hand, can not only contribute towards provider-agnostic orchestration, but also fill gaps in the management of application assets in FaaS environments.

Closely related – and an indicative measure of its importance – to service modelling, is the problem of optimal service elasticity. Service elasticity harnesses the capability of an application to use multiple instances to respond to increasing traffic. The cloud, modelled as an infinite source of resources, is the ideal grounding to build service elasticity on. The optimal balance between low cost and high performance shapes the optimal elasticity problem. A wide range of methods – from simple rules to machine learning and control theory – have been developed, aiming to provide timely application adaptations at low cost. The prominence of software architecture paradigms which emphasize on scalability and adaptivity, culminating to the Functions as a Service (FaaS) paradigm coupled by the surge in popularity of cloud applications, have created a pressing need to properly cope with workload fluctuations and efficiently handle workload fluctuations.

The mechanisms supporting cloud elasticity which are available today, can be broadly divided in three categories: i) manual decision making; ii) automatic adaptations; and iii) hybrid solutions. To the best of the author's knowledge, most of the developed mechanisms either require some input from the DevOps in the form of event-condition-action rules – and do little more than applying them – or rely on complex solutions inspired from the fields of control-theory, queuing-theory and machine-learning [27]. The knowledge acquired from machine-learning – based algorithms is not easily transferable to other domains (the transfer learning problem [28]) and they inherently lack transparency in the way the decision-making takes place, a fact that may hinder their adoption in production systems at this stage. Mechanisms requiring manual input are the most popular solution among the available providers. While simplistic in

some cases, this last category of systems guarantees a stable and predictable adaptation action in highly dynamic environments.

Based on the experience accumulated from work for the PrEstoCloud framework [6], it is argued that in such dynamic environments there is an urgent need to shift some of the core responsibilities of DevOps to an appropriate middleware. This middleware shall automatically support the separation of resources between the design time and run time and will provide appropriate error-free maintenance of dynamic topologies. Therefore, the DevOps should be able to model the deployment requirements as generically as possible, without having to make concrete decisions on the initial deployment, optimization, and reconfiguration of cross-clouds and fog computing applications. This is needed in modern organizations that need to make sure they optimally use their resources. To achieve this, first elaborated extensions to TOSCA – which is the current cloud modelling standard – are provided, and secondly a theoretical description of a middleware that can exploit the extended TOSCA is presented. The focus of this work is on 'custom' FaaS applications – cloud applications which do not use commercial FaaS services but rather implement all relevant functionality (albeit possibly using existing software primitives - e.g a load-balancer). This allows the definition of custom scaling policies, and the creation of optimizers which can determine the appropriate placement of application components. It has been decided to propose extensions to TOSCA, since they can be made a part of this standard, which then can be used to extend in a bespoke manner any other solution that is used now or in the future for cloud applications in production (e.g., Terraform, Ansible). Additionally, continuous cloud modelling support (through the use of standards and TOSCA specifically) has been argued before [1] as desirable for aligning existing and potential cloud modelling languages, and therefore achieving interoperability.

Complementing the modelling effort, in this work 'Severity' is presented, a novel algorithmic approach aiding the adaptation of cloud applications. Based on the input of the DevOps, situations are detected, Severity is calculated and adaptations are proposed, leading to better application performance for a range of workloads. The theoretical definition of Severity is complemented by a prototype software system, which uses it to characterize the current load

and produce the necessary adaptation actions. Unlike other similar approaches, the triggering of adaptation actions, as well as the adaptation actions themselves are provided in an easily understandable form and it is possible to log precisely the cause and the effect of each adaptation recommendation of the application. This part of the work focuses on the usage of horizontal scaling adaptation.

As such this work intends to answer the following research questions:

- What semantic enhancements should be made to TOSCA to describe and enact FaaS deployments, which can make use of cloud and edge resources?
- How can optimization aspects regarding the deployment be included in the modelling artefacts?
- How does the suggested elasticity methodology compare to other methodologies enabling application elasticity?
- How does Severity compare to well-known elasticity approaches in commercial products, in terms of satisfaction of QoS attributes?

To address the above research questions, a modelling approach for the definition of applications using TOSCA is firstly proposed. It is argued that TOSCA should support generic use-case patterns and deployments using serverless and other distributed execution paradigms by providing a set of relevant generic constructs. To this end, a series of custom approaches is proposed, addressing deployments in hybrid clouds (i.e., combined use of private and public cloud resources), multi-clouds (i.e., combined use of public cloud resources from different vendors), edge-based applications, and FaaS-based applications, extending the base YAML Ain't Markup Language (YAML) TOSCA specification [7]. To allow the optimization of the topology, it is suggested that two versions of the TOSCA model should be used—initially an abstract version focusing on the structure of the topology and subsequently a more concrete version, which would include more specific details of the actual deployment.

Having made the modelling extensions to TOSCA, an analysis of the suggested adaptation methodology is presented. Then, Severity is presented along with some adaptation techniques

which have been derived from it. Finally, simulation and experimental results are presented and discussed.

The remainder of this work is structured as follows. Section 3, provides a background for some of the core scientific concepts which are used in this work, as well as a FaaS application scenario which aims to improve its understanding. In Section 4, an analysis of works from the state of the art is provided. Section 5 presents a model-driven approach to guide the definition of the TOSCA semantic enhancements and artefacts to address the research questions which were raised. In Section 6, the extensions to TOSCA are detailed, in order to support FaaS applications. Section 7 includes details on the new TOSCA structures, which support optimization factors and placement constraints. Section 8 includes an evaluation of the modelling suggested in Section 4 (modelling – deployment), by means of a comparison with one of the most prominent commercial offerings. Section 8 presents the concept of Severity, its relationship to elasticity rules, indicative algorithms and their implementations. Section 9 includes an evaluation of the suggested approach, from the perspective of simulations and also based on a series of experiments on realistic FaaS infrastructure. Finally, in Section 10 a discussion of the results of this work is presented, while Section 11 concludes this work.

3. Background

3.1.Model-driven engineering for cloud applications

Applications which can utilize the benefits of cloud computing are typically complex, having interdependencies among their subcomponents and precise processing requirements which should be met. Unsurprisingly, application-specific architectural constraints pose a barrier to the refactoring of some applications, which renders the use of cloud computing resources difficult. Even if they can be appropriately refactored however, it is still very difficult to use the features offered by cloud computing (e.g. elasticity) using only the help of high-level programming languages. In an effort to handle language/platform complexity, and to express application domain concepts effectively, model-driven engineering technologies have been suggested [29].

Bringing model-driven engineering techniques to the cloud through the use of various templating systems has been providing considerable help to cloud application developers. Using current modelling tools, it is possible to specify many characteristics of the cloud processing topology and the application itself, while also the use of parameters allows a clear distinction between the business logic of application components and their configuration. Most importantly, modelling has allowed to raise the abstraction level, to use graphical tools to model the cloud infrastructure, and even validate the cloud application [30,31]. Further, the advantages brought by a modelling proposal are not confined to the aspect of the cloud application (e.g specification) which is modelled. Instead, when complementary, it can be combined with the other modelling proposals (e.g targeting the deployment process) to yield a unified model-driven approach. This was suggested in [31], where the authors used two cloud standards (TOSCA and OCCI - Open Cloud Computing Interface) to drive the deployment of cloud applications.

Notwithstanding, the use of models can be associated with extraneous development effort, when the features provided by the model-driven approach are more sophisticated than the features used by the cloud applications. Modelling can be helpful, but it should not intimidate developers to use it; Unfortunately, the effort to include ‘everything’ in a model, can prove an obstacle for its adoption.

3.2. FaaS applications

The popularity of decoupled software installations, and microservice based applications is undeniable. Technologies such as virtual machines, and containers, have certainly contributed to the success of this schema, allowing for fast and easy application deployments. However, the need for a large number of specialized, independent components often increases the burden of maintenance and fine-tuning of the system. The scaling of platform components, their compliance with security demands, as well as the monitoring for correct health and availability are necessary in many cloud application deployment scenarios. ***Function-as-a-Service (FaaS)' (or 'Serverless') is an architecture (model) for cloud-based software that focuses on executing arbitrary functions without much server- and resource-management burden put on the cloud developer or customer***[32]. FaaS aims to alleviate the aforementioned issues by decoupling functionality from maintenance and allowing software developers to concentrate their effort on specifying the application functionality. Serverless computing is being increasingly adopted by industry and studied by academics [33]. It is supported by numerous open-source and commercial platforms (such as OpenFaaS, OpenWhisk, Amazon Web Services (AWS) Lambda, Microsoft functions etc). Each of these platforms provides a framework which allows multiple application functionality units ('functions') to be executed and scaled.

These application units should be functionally independent (i.e it should be possible to execute them independently) but they can be coupled with one another to create chains and use a common backend for persistence. The FaaS developer can write functions by providing snippets of code using one of the supported platform languages (commonly Javascript, Python, Java etc.) – some platforms even support the execution of arbitrary functions packaged in a container.

FaaS implementations are not flawless (in [34] the authors mention among other shortcomings the limited execution time offered, the use of slow storage, the i/o bottlenecks and the lack of support for specialized hardware). Moreover, there are concrete examples in which serverless infrastructure is outperformed by conventional, VM-based infrastructure [34]. However, the design efficiency which can be achieved with FaaS for particular application types, the potential

cost savings for particular workloads and the speed of development [33] are significant assets for system architects – and this work aims to support their use.

The execution environment of FaaS functions is different for each FaaS application. Since FaaS applications are used both in cases where latency is important and in cases where no latency requirements are set, the appropriate choice to host FaaS processors can be either cloud, "fog" or even "edge" resources. The use of FaaS in these resources is possible either through accessing provider-specific application programming interfaces (APIs) which allow using the FaaS infrastructure of a provider (or vendor), or instead using appropriate FaaS platforms installed on VMs/PMs to emulate analogous functionality. Unsurprisingly, the execution of functions on a Cloud environment may incur a latency penalty which is unacceptable, while it can also be associated with privacy concerns (private Cloud environments are an exception). In these cases, it is advisable to use edge resources which possess adequate compute resources to perform the processing which is needed. Therefore, to take advantage of the computational capacity which is offered by edge devices – generally offering inferior elasticity support and performance compared to cloud resources – a need for efficient FaaS modelling is evident.

3.3.Motivating Scenario: Fog Surveillance Application

In this section, a motivating scenario is presented in order to assist the reader in better understanding this work. Below, an application deployment is described based on the need of a surveillance company to deploy a number of processing components both in the edge and the cloud. For this scenario, it is assumed that the testbench used includes a number of Raspberry Pi devices equipped with cameras and connected to the Internet, some ARM-based servers situated near the edge, and some VM assets in public or private clouds, conforming to the budget allocation. The processing components – or 'fragments' – which are considered in the scenario are described in Table 1.

Table 1. Fog surveillance application fragments

Fragment Name	Description
VideoStreamer	The fragment is responsible for the transmission of video from the edge to the host of the VideoTranscoder fragment
VideoTranscoder	The fragment is responsible for changing the format of a video
FaceDetector	The fragment is responsible for face detection in a captured video scene
AudioCaptor	The fragment is responsible for continuously capturing audio
PercussionDetection	The fragment is responsible of detecting any captured percussion sounds, and triggering the FaceDetector component
MultimediaManager	The fragment hosts various necessary assets to perform the detection of suspects and present alerts to a user of the platform

Hereafter, the term 'processing component' will be used interchangeably with the term 'fragment', as it is considered that the 'whole' of an application consists of one or more fragments. Throughout this work, fragments are assumed to be containerized, using the de-facto standard of containerization, Docker.

For all fragments which can be executed on edge resources, priority is given to the deployment on suitable edge hosts. However, if a fragment cannot be executed on edge resources (or none are available) then deployment criteria are used to govern its deployment on one or more cloud providers. For the VideoStreamer, and MultimediaManager fragments, the primary objective which should govern the cloud deployment is the reduction of latency. The secondary objective preferred is the usage of the AWS cloud provider, and finally the reduction of cost is considered an additional business goal. For the FaceDetector fragment the first priority is the usage of the AWS provider, as a stringent agreement has been reached with the particular provider on the handling of sensitive data.

Since the VideoTranscoder, FaceDetector and PercussionDetector fragments are assumed to perform stateless operations, all of them can be attached to a common FaaS Proxy which will balance and redirect incoming requests appropriately.

The operating system for all fragments is defined to be Ubuntu Linux. In addition, the Google Cloud Compute provider is required to be excluded. Moreover, the budget available for cloud

deployments is set equal to 1000€, and the time-frame for which it will be available is set to 1 month (720 hours). Fine-grained optimization criteria (explained in detail in Section 7.2) have been set for the VideoStreamer, FaceDetector, and MultimediaManager fragments, according to the requirements of the fragments described. The overall optimization objective is the reduction of the cost, through the reduction of fragment instances scheduled for execution on the cloud.

To illustrate the data flow dependencies, a deployment graph was created in a prototype UI - corresponding to a deployment using the above fragments. The application graph is shown in Figure 1. The arrows indicate (from right to left), that the MultimediaManager fragment depends on data from the VideoTranscoder, FaceDetector and PercussionDetector fragments, which in turn depend on data from the VideoStreamer, VideoTranscoder and AudioCaptor fragments, respectively. The scenario and the respective application graph, were created as part of work [35].

The requirements for each fragment are listed in Table 2:

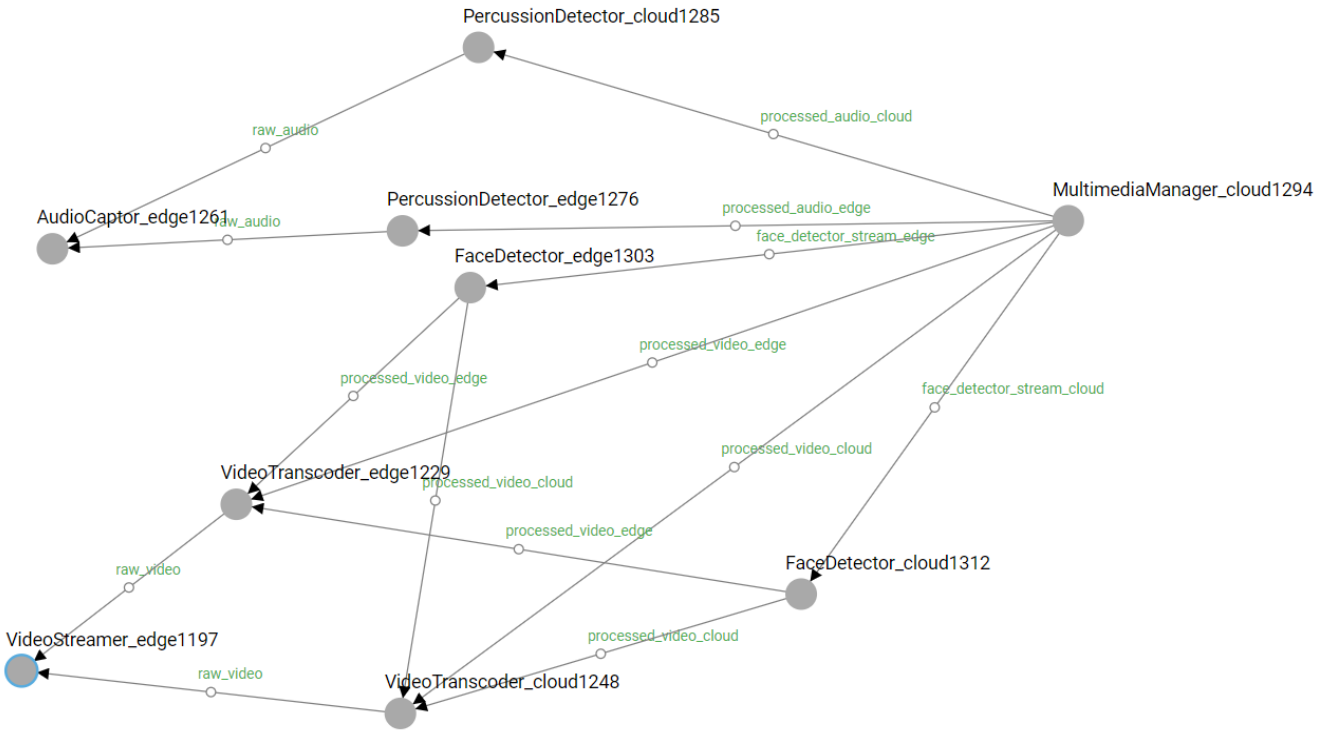


Figure 1. The deployment graph for the illustrative example

Table 2. Fragment Processing Requirements and Constraints

Fragment Name	Hosting Requirements (CPU cores – RAM GB's – Free Disk GB's)	Acceptable Hosting Resource	Processing Architectures	Collocation dependencies /Anti-Affinity requirements	Precedence Dependencies	Optimization Criteria (Cost-Distance-Friendliness)	Elasticity Mechanism
VideoStreamer	1-1-4	Edge	arm64, armel, armhf	VideoTranscoder (collocation)	-	2-8-{aws:5, gce:0, azure:1}	None
VideoTranscoder	2-4-4	Edge /Cloud	arm64, armel, armhf, x86_64, i386	VideoStreamer (collocation)	VideoStreamer	1-1-(1, implied)	FaaS (Lambda) Proxy
FaceDetector	1-1-4	Edge/ Cloud	arm64, armel, armhf, x86_64, i386		VideoTranscoder	1-1-{aws:5, gce:0, azure:1}	FaaS (Lambda) Proxy
AudioCaptor	1-1-4	Edge	arm64, armel, armhf	PercussionDetector (collocation)	-	1-1-(1,implied)	None
PercussionDetector	1-1-4	Edge/Cloud	arm64, armel, armhf	AudioCaptor (collocation)	AudioCaptor	1-1-(1,implied)	FaaS (Lambda) Proxy
MultimediaManager	2-4-128	Cloud	x86_64, i386	-	FaceDetector, VideoTranscoder, PercussionDetector	2-8-{aws:5, gce:0, azure:1}	None

4. State of the art analysis

4.1.Model-driven cloud application deployment

The OASIS TOSCA standard [7] is based on the definition of a cloud application through the usage of templates or blueprints. There are several implementations (although none are officially endorsed), some of which support more features and are more actively maintained than others. Indicative examples include Alien4Cloud (also used by the Apache Brooklyn project) [2], Cloudify [3], and OpenTOSCA [4]. All of these implementations allow the definition of new node types, the generation of TOSCA deployment templates, and the orchestration of the deployment. They are designed for single deployments of a cloud application and do not incorporate any optimization capabilities. Moreover, the high level of detail in these TOSCA templates provides a complete view of the application, however imposes difficulties in terms of the comprehension of its overall structure. Conversely, the description of the model of an application is difficult in these platforms, without first describing a complete proof-of-concept.

Starting from the above observation, this work proposes a clear distinction between an initial, modelling-oriented (and more abstract), "type-level" flavor and a final, "instance-level" flavor of TOSCA. The blueprints that are generated by the type-level TOSCA generator which has been developed [21] are vendor-neutral and can be deployed on any (combination of) cloud(s) and edge resources. In doing so, consistency is maintained with the intent-based design, which is endorsed by TOSCA. Using TOSCA, the power of expression of policy-based approaches (i.e approaches focusing on the constraints and rules that application parameters should conform to) can be combined with the modelling power of steady-state approaches (i.e approaches which specify the service state that an infrastructure should maintain) [36]. In this

work the maintenance of relationships between components is emphasized by defining appropriate TOSCA relationships and capabilities.

In [8], Wurster et al. proposed an extension of OpenTOSCA to describe FaaS-based applications. In addition to purely FaaS-focused applications, their work is also valid for mixed architectures consisting of FaaS-based and VM-based solutions. Their modelling scheme is specifically applied on an AWS-Lambda-based application. The proposed approach in this work can be used alongside such modelling, as it does not focus on the support of provider-specific components or services.

In [37], Yussupov et al. discuss a methodology to translate serverless function orchestrations from BPMN to multiple vendor-specific TOSCA models. Their work is illustrated using three different FaaS providers, on a non-trivial FaaS application. However, since translation is automatically done from BPMN to concrete TOSCA types, using the facilities offered by each provider, room for optimization is limited.

RADON [9] is another approach that is based on TOSCA. RADON extensively uses TOSCA inheritance to define abstract and derived concrete, deployable entities. Overall, RADON is considered closer to the concept of instance-level TOSCA (see Section 6.3), as it contains detailed information related to deployment parameters of particular cloud application types and serverless functions. Similar to this work, the modelling specification of RADON allows the definition of a custom FaaS architecture. VM deployment is mentioned in the RADON reference technologies [38], however no reference was found for edge deployment. Without edge deployment support, the low cost and high data processing locality offered by edge nodes is impossible to exploit (at least automatically). Moreover, the appropriateness of particular edge nodes for particular fragments cannot be modelled.

CAMEL [10,11] is described as a domain-specific language (DSL), which enables dynamic and cross-cloud deployments. The authors support that while TOSCA and CAMEL are similar, the latter can also be used not only during design, but also at run time because it can specify the instances to be used. CAMEL relies on multiple

specialized DSLs, each focusing on a particular aspect. It emphasizes the creation of UML-based metamodels, enriched with additional domain semantics. The models that are created are always synchronized with the actual topology that is deployed at the time. Both direct (manual) and programmatic access to these models is allowed, enabling self-adaptive cross-cloud applications. Furthermore, CAMEL has already been extended to support the specification of commercial FaaS services [5]. While CAMEL provides some advanced features and can already manage cross-cloud deployment and adaptivity, there are significant aspects requiring improvement. First, no language features specifically target edge devices (for example to account for the volatility of the devices or the migration of components from the cloud to the edge). Thus, the topology can only partially be optimized to consider the benefits of fog computing. Also, FaaS services are modelled from the perspective of a FaaS framework consumer (i.e., user of already-existing commercial offerings such as AWS Lambda), rather than a FaaS framework designer (i.e., creator of any FaaS service).

Another significant modelling effort is OCCI, which according to [39] is a protocol and API for all kinds of management tasks. It is also stated that the main focus of OCCI is to create a remote management API for IaaS model-based services allowing for the development of interoperable tools for common tasks, including deployment, autonomic scaling, and monitoring. In [40], the authors support the idea that the focus of OCCI is to provide a standardized API and that it does not define concepts to handle reusability, composability, and scalability. Conversely, TOSCA offers means to express reusability, composability, and scalability. These advantages grant TOSCA a superiority in its modelling capabilities over OCCI. Moreover, TOSCA can be used alongside OCCI [31,40] to achieve full-standard-based deployments [31].

Terraform is a popular declarative language oriented towards cloud deployments, also supporting FaaS services, backed by open-source implementation. Different plugins exist to instantiate nodes on different cloud providers and interact with external

services providing content delivery network (CDN) and domain name service (DNS) facilities using a unified syntax. All of these features grant versatility and robustness to Terraform. Pulumi[3] is another open-source framework similar to Terraform, which provides the additional advantage of using programming languages to express cloud topologies rather than requiring the use of a specific cloud application language. Similar to Terraform, it is also capable of handling FaaS deployments. However, when considering model-driven deployments, Terraform and Pulumi present certain disadvantages compared to TOSCA. These disadvantages originate from the fact that while resources are properly declared, there are no language features offered that can be used to generalize the relationships among application components. Thus, no means are offered to (i) extract generic, reusable blueprints and (ii) optimize the deployment of components, taking into account any other dependent components. Additionally, the computing resources and their roles are very specific and detailed, which while providing a concrete view of the state of the deployment, also obstructs the higher-level understanding of the model of the cloud application. The use of edge devices is possible, but it requires manual configuration of the details of the topology, as illustrated in Section 8. TOSCA, on the other hand, excels in its capability for modelling and abstraction of an application, while also being capable of specifying concrete actions that should be considered when instantiating a topology (through its workflows feature). In this work, the modelling capabilities of TOSCA are enhanced, as a set of new constructs is introduced to assist the representation of hybrid clouds, multi-clouds, edge, VM, and FaaS-based applications.

In addition, proprietary software systems such as Amazon Cloud Development Kit (Amazon CDK) have been developed, providing capabilities traditionally offered by Cloud DSLs. Using Amazon CDK, a DevOps can model the application directly from the integrated development environment (IDE) used and specify the requirements of the application using a preferred programming language (TypeScript, JavaScript,

[3] https://www.pulumi.com

Python, Java, and C#/.Net are currently available). Although such an offering allows deep integration with the existing Amazon constructs and offers a good abstraction over the Amazon services that are used, it is nevertheless difficult to introduce it without the prior expertise of the DevOps with the specific technology products offered by Amazon. Furthermore, services can only be developed in connection with the AWS cloud computing provider, threatening vendor lock-in.

The vendor lock-in problem also emerges when considering other DSL-based solutions, such as Azure Resource Manager, and CloudFormation AWS templates. In some cases such as Openstack Heat templates and Google Cloud Deployment Manager templates, support for FaaS is obscure if not absent. Moreover, all of the templates cited above contain too many technical details that are associated with the solution offered by a particular provider. On the other hand, this work aims to simplify the model of the application, providing two views: a model view before the deployment of the application featuring the least amount of technical details and an instance view after the instantiation of the topology (type-level and instance-level TOSCA, respectively).

Wurster et al. [41] reviewed prominent deployment automation approaches to derive the essential deployment metamodel. The metamodel refers to a technology-independent baseline, containing the core parts of deployment automation technologies such as Chef, Puppet, Ansible, Kubernetes, and OpenStack Heat. The authors state in their work that the generated metamodel uses only a subset of the entities described in TOSCA. Approaches similar to the metamodel can further be used to introduce or map other technologies to the terminology of TOSCA, and vice versa.

In an effort to reduce the processing latency incurred by using a FaaS framework, the creation of edge-based FaaS systems has been announced in the past few years. Applications that use processing nodes at the edge of the network can attain considerably better performance for applications that are either response-time-

sensitive or privacy-oriented or that aim to minimize energy consumption [42]. Representative FaaS systems include Amazon GreenGrass [12] and Azure IoT Edge [13].

4.2.Model-driven Fog application deployment

The study by van Lingen et al. [14] extended the YANG language [15] with support for fog nodes. A similar approach using TOSCA is also followed in this work, as TOSCA is directly aimed at cloud deployments and is already an OASIS standard. Noghabi et al. [43] worked on Steel, a high-level abstraction for the development and deployment of edge–cloud applications. Their work emphasizes the ability to migrate services from the cloud to the edge, and the ability to optimize the placement of services while respecting constraints. Mortazavi et al. [44] proposed CloudPath, a multitier computing framework, in which the location and REST path of a FaaS system running on fog resources were configured using web.xml Java deployment descriptors. The semantic enhancements of the TOSCA standard – which are introduced in this work – can also support the definition of placement constraints, while the definition of conflicting optimization criteria is also allowed. Since there are plenty of deployment automation tools built on top of it, extending TOSCA would make sense from the perspective of reusability, as the implied extensions to the TOSCA-based deployment tools should be manageable.

4.3.Cloud application Elasticity

Decision-making approaches for cloud autoscaling systems are based in general either on rule-based control, or Control Theory and Search-based optimization [16]. In the following subsections this work is positioned in relation to other works from these fields. It is assumed that a single virtualization layer is used, and that no

synchronization issues as those investigated in [45] appear. Moreover, unlike some works which consider resource contention between cloud components (e.g [46]), the adaptation approach proposed in this work through Severity, is targeted on generic contexts, on which resource contention may not necessarily appear and resource contention is not considered.

4.3.1. Rule-based and Control-theoretic adaptation approaches

The rule-based adaptation approach is one of the simplest and more intuitive approaches which can be followed to scale a cloud application. Rule-based adaptations rely on the expertise of a DevOps to define the variables which should be monitored, and the thresholds which should be respected (a priori knowledge). The rules should be carefully tuned in order to include all variables which can influence the deployment. While some adaptation systems only use some adaptation attributes for input (as indicated in [16]) such as CPU or the response time, the system which is proposed in this work can work with any number and type of measurable attributes.

Control theoretic approaches (e.g [17–19]) are based on traditional control theory but are occasionally enhanced with extensions. They are characterized by their dynamicity and low latency. However, the configuration of the control loop should be performed by an expert in order to prevent waiting for the system to stabilize after multiple iterations. Rule-based approaches hold an advantage over pure control-theoretic approaches on simplicity and clarity, and thus domain experts can more easily transfer their knowledge to the systems.

Gandhi et al. [27] describe a technique based on Kalman filtering, which estimates the parameters of a queuing model representing the application. The estimated values are used to create scaling directives, providing auto-scaling capabilities to the platform. When an abrupt change in its time-series representation is detected, a scaling event is transmitted. The authors evaluated the performance of their algorithm and found it superior to threshold-based rules working with static percentages, adding or removing

one VM instance when the threshold (upper or lower respectively) has been violated. Using the suggested approach, scaling actions can be more varying and detailed in their response, than simply adding or removing a single VM instance. Besides response time, any number of attributes to feed the suggested Severity-based techniques. Lorido-Botran et al [20] comment on queuing theory models used to horizontally scale an application, that they suffer from being tightly bound to the workload, the application and usually the processing infrastructure for which they have been created. As a result, they need to be recalculated when these change.

Arkian et al. [47] propose Gesscale, a control-theory inspired autoscaling approach, based on the measurement of the maximum sustainable throughput. The estimation of the results of a scaling action using in Gesscale is based on the existence of a performance model. When the performance is better than expected, multiple processing instances can be removed, while when it is worse, a single processing instance is added. They use a single composite metric (maximum sustainable throughput - MST) to guide their autoscaling model. MST is calculated based on the maximum network delay between nodes, the throughput of a single node, and the parallelization inefficiency. Using their methodology, they demonstrate superior performance compared to algorithms which are latency-unaware, and/or use only the cpu consumption as an indication of the intensity of the workload. In practice this approach still uses one strictly defined (albeit composite) metric to guide scaling. Instead, Severity combines any number of arbitrary metrics to obtain better results rather than using individual metric values to scale. Moreover, the Severity scaling algorithms which are defined, allow more than one instances to be added as necessary which reduces the number of reconfigurations.

In [48] the approaches of Amazon and Google concerning scaling are described, Target Tracking and Step Scaling, and Multiple Zones and Horizontal-Pod Autoscaling (Kubernetes) respectively. These tools can be divided into two algorithmic categories, the first containing Multiple Zones and Step scaling, and the

second containing the Kubernetes Horizontal-Pod Autoscaling and Target Tracking. In the first category of tools (which is also encountered in other major providers, such as Microsoft Azure [49] and Oracle Cloud Infrastructure [50]), the DevOps should either enter a number of rules that scale out/in the application by a predefined number of instances (or a percentage of the active instances). Unfortunately, while this approach is simple, it requires considerable input from the DevOps. Tools belonging in the second category are more sophisticated, requiring the creation of a control-loop that will perform scaling automatically to attain a specific threshold value. Amazon Target Tracking supports only one metric in the Control Loop (but it can be composite, and multiple parallel scaling directives may exist) and the Kubernetes horizontal pod auto-scaler (HPA) can also support multiple metrics.

In the same work [48], a custom approach to scaling using the 'dynamic-multi level' (DM) method is outlined, combining predictive elements with a control loop to direct the scaling of the platform. Using a variety of workloads and benchmarking metrics, an evaluation against a real system was carried out, and their approach was found to be better than approaches which are used by leading cloud vendors in many scenarios. However, only one threshold value was used for all algorithms, and a default VM instantiation delay of 30 seconds was assumed. The adaptation techniques proposed in this work were evaluated in a realistic setting using two cooldown intervals, and in simulations using four VM spawn delay intervals, as well as six combinations of thresholds (in simulations) to detect variation in their performance. Additionally, the workload patterns which are used in simulations are more radically changing compared to those provided in [48] (in terms of the rate of change of the absolute values of the workload), stress-testing the performance of all techniques.

In [51], an extension to the Kubernetes HPA algorithm is discussed, evaluating the use of a constant absorbing small fluctuations of the workload. In cases where scaling is performed using multiple metrics, it is supported by this work that one or more of the performance criteria of the application can be improved, when all of the available

values of the monitoring metrics are used (rather than only the maximum value or only one metric value as is the case in Kubernetes HPA and Amazon TTS, respectively). Severity also allows creating hybrid algorithms – for example a control loop activated by rule thresholds as illustrated in section 8.4.5.

Another interesting approach is followed by Lorido-Botran et al. [52], who thoroughly describe the idea of modifying the thresholds which are employed in rule-based systems to obtain a better response. They support that when no service-level objective violations are detected within a time frame, the scale-in and scale-out rule thresholds should converge to higher and lower values respectively to improve the responsiveness of the system when high workload is encountered. The evaluation of their algorithm is performed using a single, highly variable workload trace, and two benchmarking criteria (service-level objective violation and cost). Approaches similar to those introduced in [48,51,52] are viewed as complementary to the use of Severity and can be combined to possibly provide an enhanced yet more complex system.

Vaquero et al [53], Galante et al. [54]and Copil et al. [55,56] have proposed rule-based frameworks, which either rely on user input to calibrate the adaptation actions by manually setting the scaling action as in [53,54] or always using the same adaptation event (e.g add one VM instance) to keep the desired monitoring attributes to acceptable levels as in [56]. Using these frameworks, the user should manually detail all the situations for which adaptation will be required. However, this process is error-prone and nevertheless requires the constant attention of the DevOps. In the case of Ferretti et al. [57], the ability to implement scaling decisions adding or removing more than one instances is supported, however no information is provided on whether the number of instances (de)allocated can change at runtime without the intervention of the DevOps. Severity-based adaptation techniques aim to waive the requirement from a user to frequently change the response of the system, as the user needs only to specify basic thresholds with a generic action once. Then, the violation of these thresholds can be measured and an adaptation action automatically be derived.

In [58], Trihinas et al enhance the basic adaptation support offered by the previous rule-based systems, by offering AdaFrame, a library to support resource-based elasticity controllers. AdaFrame improves the results of rule-based systems by adapting a cooling-down period between successive adaptations, through the analysis of the statistical properties of a monitoring metric stream, e.g., CPU utilization. Thus, scaling out and scaling in actions are less likely to occur on sudden bursts, and occur faster in the case of increased 'regular' workload. This approach is complementary to the one suggested in this work, as it improves the triggering of the autoscaling loop.

Dutreilh et al. [59] have explored both threshold-based rules and Q-learning, concluding that Q-learning is superior, given enough training. Two of the techniques which are examined in Sections 8.4.6 and 8.4.7 are simplifications of Q-learning, with the absence of feedback. Unlike Q-learning though, the suggested Severity-based techniques benefit from being usable without extensive training or requiring the definition of a complex reward function. Besides, a mechanism is described to ascertain the Severity of a situation, similar to the reward function employed in Q-learning, which can be used as input for a multitude of algorithms, one of which can also be Q-learning.

Ali-Eldin et al. discuss in works [60,61] elasticity controllers based on a generic model of queuing theory, the G/G/N queue. In work [61] they consider workloads which can be queued and then be appropriately serviced by tuning the number of VMs according to the requests which should be serviced. Their approach allows a service to remain operational even under heavy load, by limiting the queued requests. The availability of a buffer to queue requests is not present in their previous work [60]. The principle behind the scaling of the application in their approach is similar to the algorithms which are proposed. Moreover, since Severity provides a means to scalarize a set of metric values one should be able to use a single-metric based controller - such as the one which is proposed in their work – with the value of Severity as its input. While Severity focuses on the ability to extract more information

from rules involving multiple metrics, this does not preclude the use of the advanced techniques presented in these works (e.g different combinations of reactive and/or proactive scaling-up and scaling-down).

In [62] the authors propose Chameleon, a hybrid, proactive autoscaling mechanism, evaluating its performance using realistic workloads and works suggested in the state of the art. CPU utilization and request rate are mainly used to estimate the workload of an application and guide autoscaling. Chameleon combines forecasting methods and realtime monitoring to enable proactive and reactive scaling decisions. It uses thresholds for both reactive and proactive scaling decisions, however the service demand estimation component also uses Kalman filter, regression and optimization estimators (among others) to estimate the time required for a request. Chameleon is extensively evaluated against other approaches suggested in the autoscaling literature and found to outperform them by a large margin. Severity enables the use of more metrics if necessary, including custom metrics. As such, it is argued that it can enrich approaches such as Chameleon to consider additional context factors (metrics) for their autoscaling algorithms.

4.3.2. Search-based optimization adaptation approaches

Search-based optimization approaches comprise another main category of decision-making approaches used by Self-aware and Self-adaptive Cloud Application Systems (SSCAS) [16]. Using the classification of Chen et al. [16], search-based techniques include dynamic programming, genetic algorithms, reinforcement learning and integer linear programming among others. By definition, all of these techniques are based on traversing the search space of solutions using a specific algorithm, attempting to optimize one or more criteria. However, the exponential number of solutions which should be explored when considering a number of attributes and actions which can be optimized results in training or execution times (or compute power requirements) which can be unacceptable. Also, while these techniques require less work from the

side of the domain expert – as a lot of information is learnt at runtime – they need more time to converge. Heuristics help with the aforementioned problems, yet unless the actions learnt can be translated to a set of rules/statements, no knowledge can be transferred in a case of a different instance of the problem.

The work of Ramirez et al. [63] describes an autoscaling mechanism which considers two virtualization layers (VMs and containers) to deliver the required quality of service. Quality of service is calculated based on the number of requests which can be serviced. Five different techniques to determine the number of VMs and containers for a workload are evaluated, three of which traverse the configuration space (number and type of containers in VMs) to find an appropriate solution (the others use heuristics). They demonstrate that appropriately handling scaling using two virtualization layers results in reduced cost. The techniques described below can be used in parallel with such approaches although the focus of this work is on applications which exploit a single virtualization layer (or use the assumption of one container instance in one virtual/physical machine). Moreover, multiple metrics are allowed to influence the decision of scaling.

Zhu et al. [64] presented a queuing-network-based approach to optimize the allocation of AWS Lambda resources, based on the requirements stated in a TOSCA model. Using their approach, a layered queuing network firstly predicts the performance of a FaaS-based application, and then the optimal configuration (appropriate memory reservation and concurrency) is found using genetic algorithms. In this work, the use of additional metrics through Severity is proposed and a generalized approach which can work over a multicloud infrastructure is offered.

In [65] the authors compare two functionality modes of the Kubernetes Cluster Autoscaler. The Kubernetes Cluster Autoscaler is a component responsible to allocate new processing nodes to host Kubernetes pods when this is necessary. The first functionality involves using nodes from a single node pool (identical nodes - CA) while in the second multiple node pools are used (allowing differently-sized nodes to

be spawned – CA-NAP). They conclude using standard autoscaling metrics that CA-NAP is overall superior to the CA, although no significant cost benefits are observed. In the evaluation of this work, nodes with a similar processing capacity are considered. However, Severity-based techniques can be generalized to use processing nodes offering a fraction of the performance of a normative processing node.

GKE Autopilot [66] offers an advanced autoscaling approach, capable of vertical and horizontal autoscaling. The main emphasis of this work is on vertical scaling, setting the appropriate resource limits for each processing node. Autopilot can set these limits, even if no user input is provided. Autopilot manages to greatly reduce slack (unused resources) using either statistical or machine learning techniques. However, the configuration of statistical recommendations is tuned for long running services, which might not be optimal. Besides, its machine learning recommender has the advantage that it can output easily explainable recommendations on the resource limits of a processing node. The part related to horizontal scaling resembles the algorithm which is used by the Kubernetes Horizontal Pod Autoscaler.

AWS [67] offers another autoscaling approach which is based on predicted data about the application. It tries to attain a target utilization level, based on monitoring metrics (it does not however currently support custom metrics). It uses machine learning models trained in Amazon, based on billions of data points. These machine learning models are not known though to be publicly available. Notwithstanding, it is difficult to train ML models of a comparable size without access to the data, algorithms and processing infrastructure used by Amazon. The approach suggested in this work is independent from the use of a particular algorithm. Nevertheless, the Simple Severity Zones algorithm which is most thoroughly explored, does not need any training, the adaptations created by it can be easily traced to the original monitoring observations and can it can be used in addition to the presence of a forecasting mechanism (e.g using predicted metrics in the definition of rules). Severity-based techniques extract added value from domain expert knowledge while retaining the simplicity of

threshold rules. A thoroughly documented, open and modular approach is introduced which - thanks to the scalarization realized through Severity - can use ideas present in any existing horizontal scaling adaptation technique using metrics (as Severity itself can be considered a metric).

5. Suggested approach for the definition and adaptation of FaaS applications

In accordance with most efforts analyzed in Section 4.1, a model-driven engineering approach is followed, because it offers portability and reusability, which are important considerations for all cloud applications. Specifically, an extension of the TOSCA modelling standard is proposed adopting two views of the topology—type-level and instance-level TOSCA. Each of these is used to create a document reflecting the initial abstract view and the processed deployment view of the topology, respectively. The decoupling of the information contained in type-level TOSCA and instance-level TOSCA is meant to aid modelling and allow for the optimization of applications across the cloud computing continuum. These models are not bound to a specific provider, and given a TOSCA orchestrator can be used at any time.

Type-level TOSCA encapsulates the user's requirements, preferences, and business goals and provides a high-level overview of the topology to be deployed. This model can subsequently be optimized, finalized with provider-specific characteristics (e.g., network parameters), and deployed. During this process, a final "instance-level" model of the application can be created. External monitoring mechanisms, e.g., Prometheus[4], can be used to create an updated "type-level" deployment, which will trigger a reconfiguration of the platform. This workflow is in line with the challenges and research directions for cloud adaptations that were described in work [68].

The definition of a fog application (e.g., similar to the one depicted in Figure 1) can be seen in the steps presented in Figure 2.

[4] https://prometheus.io

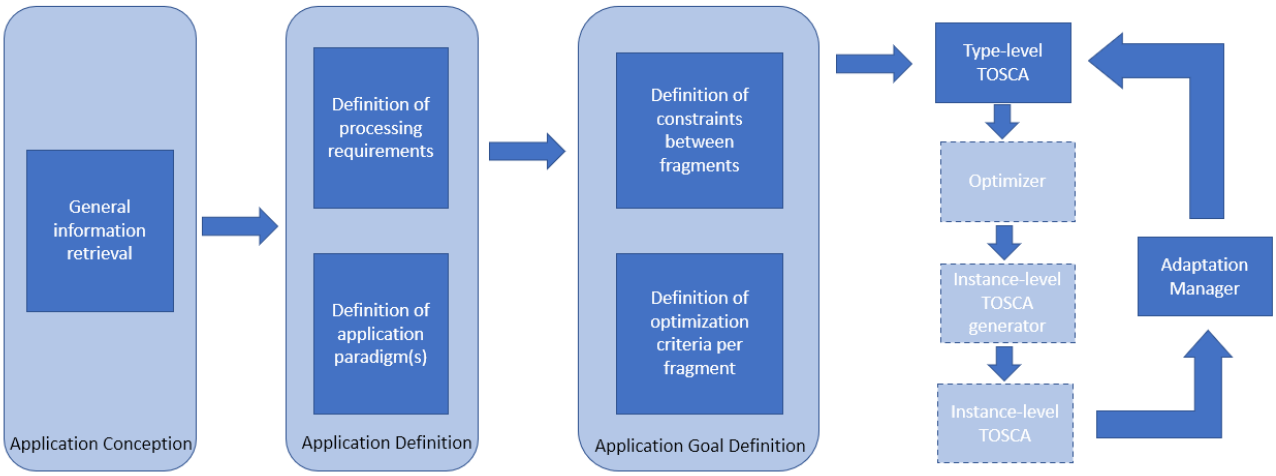


Figure 2. Overview of the suggested approach. Although the optimization of the type-level TOSCA template and the generation of the instance-level TOSCA are essential constituents of the suggested approach, an implementation is not provided for an optimizer or an instance-level TOSCA generator.

To facilitate the use of the modelling extensions, a tool [21] is provided to aid application definition and application goal definition as a proof-of-concept for the approach discussed in this work. The tool was developed as part of the PrEstoCloud framework [6], and is able to parse the input requirements, optimization criteria, and constraints, generating type-level TOSCA as its output. In the following subsections, more details are provided for each of the steps illustrated in Figure 2.

5.1.Application Conception

In the application conception stage, the base for the proposed modelling approach is established, as the DevOps structures the fragments that will comprise the application. Firstly, the architecture of the application is determined, as well as its input(s) and output(s) and the components that will be used in it. Following, the DevOps examines the fragments used in the application to determine how is the FaaS application paradigm used in it, and the number of instances necessary for the processing of each fragment. In addition, generic information related to the fragments themselves (e.g., a known functional configuration, as well as the interfaces that are provided and the interfaces that are needed) should be collected. Then, the DevOps should determine

for each fragment the Docker image and Docker registry that should be used, the respective environmental variables and their port mappings, and the proxy which should be used (see Section 6 for details). All of these attributes should later be mapped to the respective TOSCA constructs.

Finally, global constraints and preferences can be provided by the DevOps, specifying the providers that are preferred or should be excluded and the budget constraints that should be respected for this application.

5.2.Application Definition

During this step, the DevOps should enter more precise information describing the processing context of each of the fragments, as well as any application paradigm that is followed by the fragment. The information provided for the processing instances and the processing resources for each fragment are critical, as all further deployments and optimization steps will be primarily based on these factors. The DevOps is required to provide data for the fragment attributes, which are described in Table 3.

Table 3**.** Fragment attributes that should be defined by the DevOps.

Fragment Processing Attribute	Description
Number of cores	The minimum and maximum numbers of CPU cores needed to process the particular fragment
Processor architecture	The architecture of the CPU processor
Memory requirements	The minimum and maximum amounts of main memory required by a fragment
Disk capacity requirements	The minimum and maximum amounts of disk space required by a fragment
Sensors required	The input or output sensors required for the processing of this fragment
Operating system	The operating system and version that should be used for the processing of this fragment

Number of instances	The minimum and maximum numbers of instances of a particular fragment
Execution zone	The eligible locations for the processing of a particular fragment (cloud, edge)
Elasticity mechanism—paradigm	The elasticity mechanism that is required for a fragment implementing a particular paradigm (can be FaaS or none, although a mechanism to additionally handle load-balanced and JPPF applications has been suggested in work **[35]**)

The approach described in this work is agnostic to the medium used to define these properties (as well as those mentioned in Section 5.3). However, tool [21] supports both the usage of an external prototype UI, as well as code-level annotations coupled with a policy file. In the first case, informative application graphs similar to the one illustrated in Figure 1 can be retrieved and the input is provided to the component through graphical forms in a machine-readable json format. In the second case, annotations can capture details concerning the deployment requirements of each fragment, while the policy file contains global constraints and preferences that guide the deployment of the topology. The handling and specification of annotations is an integral part of the TOSCA generator, allowing the software to be used without depending on an external component. The main advantage of using an annotation-driven deployment lies in the potential to combine annotations with code introspection techniques to influence the behavior of the application. In [56], information present in annotations was used to govern the addition of new workers and removal of existing workers or to modify their behavior.

An annotation is expected firstly to include information on the memory, CPU, and storage load that the particular application fragment will use, which is translated into a range of values using information from the policy file. The translation is performed using configurable static mappings (e.g., for CPU or memory, VERY_LOW = 1 core/1Gb of ram, LOW = 2 cores/2Gb of ram, MEDIUM = 4 cores/4Gb of ram,

HIGH = 8 cores/8Gb of ram and VERY_HIGH = 16 cores/16Gb of ram). Each annotation on the workload of a particular resource reflects the least amount of resources required to carry out the processing needed by the particular fragment.

Then, information on the possible processing zone should be provided—whether the fragment is onloadable (i.e., can be deployed on edge devices), offloadable (i.e., can be deployed on cloud VMs), or both. Afterward, information on the Docker configuration (here using the edge_docker_registry and edge_docker_image fields) of the component should be provided. Additionally, the application paradigm followed by the particular fragment can be specified using the elasticity_mechanism annotation. It should be noted that each of the fragments can follow a different paradigm and multiple paradigms may co-exist in the same application.

The annotations scheme suggested also allows one to provide the minimum and maximum numbers of instances that should be provided for a fragment. While one may argue that it is difficult to enforce a particular number of instances on a volatile edge topology, this capability is included as it is a business requirement that is very common and that provides to the DevOps the capability to precisely define the requirements of the topology. Further, it is assumed that the information that is input by the DevOps in a type-level TOSCA template is not static but subject to (automatic) updates based on the state of the processing topology, as illustrated in work [68].

Listing 1 contains an example annotation for a Java class representing the percussion detector fragment (it can either be a placeholder or its actual implementation), which is defined in the illustrative scenario. In the case of an annotation-driven deployment, one annotation needs to exist over each fragment that can be autonomously executed. As mentioned before, the developed tool can also accept input from a prototype user interface.

```
@PrestoFragmentation(
 memoryLoad = PrestoFragmentation.MemoryLoad.LOW,
 cpuLoad = PrestoFragmentation.CPULoad.VERY_LOW,
 storageLoad = PrestoFragmentation.StorageLoad.LOW,
 onloadable = true,
 offloadable = true,
 edge_docker_registry = "prestocloud.test.eu",
 edge_docker_image = "percussion_detector:latest",
 elasticity_mechanism = faas,
 min_instances = 1,
 max_instances = 5,
 dependencyOn = {"imu_fragments.AudioCaptor"},
 precededBy = {"imu_fragments.AudioCaptor"},
 optimization_cost_weight = 1,
 optimization_distance_weight = 1,
 optimization_providerFriendliness_weights = {"aws", "1",
 "gce", "1",
 "azure", "1"
 }
)
public class PercussionDetector {
}
```

Listing 1. Annotations example for the PercussionDetector fragment of the illustrative scenario.

Listing 1 also contains (for completeness) certain annotations that are not mapped to the requirements, which should be set for each fragment at this stage—these are discussed in Section 5.3.

5.3.Application Goal Definition

During this step—with the architecture of the application determined and the application components fully described—the DevOps can finally specify the application goals in terms of the optimization criteria that should be used for the deployment of the fragments, as well as any constraints that should be applied. The available fragment optimization criteria and constraints appear in Table 4.

Table 4. Fragment goals (optimization criteria and constraints) available to be defined by the DevOps.

Application Goal	Type	Definition Level
Precedence	Constraint	Defined for groups of two or more fragments
Collocation	Constraint	Defined for groups of two or more fragments
Anti-Affinity	Constraint	Defined for groups of two or more fragments
Distance	Optimization criterion	Defined for each fragment
Cost	Optimization criterion	Defined for each fragment
Friendliness	Optimization criterion	Defined for each fragment

If no constraints are chosen, the application deployment will be guided only by the automatic device exclusion constraint(s) (see Section 7.2), the optimization criteria specified for each fragment (where these exist), and the overall business goal that has been set for the application (e.g., the minimization of cost). If no optimization criteria are specified for one or more fragments, the application deployment will be performed based on the overall business goal, fulfilling any placement or precedence constraints. Naturally, the set of constraints and optimization values, which will be adopted for the fragments of the application, can lead to completely different deployments.

Using the example of the illustrative scenario, based on Table 2 it can be determined that a collocation constraint is required between the VideoStreamer and VideoTranscoder fragments and the PercussionDetector and AudioCaptor fragments. Moreover, the prevalent optimization criterion that will govern the processing zone (cloud or edge) and processing location (provider data center or edge device) of the VideoStreamer instances will be “distance” (for more details see Section 7.2). For annotation-driven deployments (as in Listing 1), the precedence constraints are indicated using the precededBy annotation, while the collocation constraints are indicated using the dependsOn annotation. Individual optimization criteria are indicated using the optimization_cost_weight, optimization_distance_weight, and optimization_providerFriendliness_weights annotations, respectively.

As soon as the requirements of the application have been provided, the initial type-level TOSCA model should be created. This procedure is undertaken by a TOSCA generator (e.g., [21]) which receives the input gathered at the first stage of the processing and converts it to TOSCA format, producing a type-level model of the topology. This process was completed for the application described in Section 3.3 and the type-level TOSCA model corresponding to it was created using a prototypical, open-source TOSCA generator (the full type-level document is included in Appendix A). This model file should then be sent to a TOSCA orchestrator—for example by uploading to a repository node, which will enable file artefacts to be communicated and stored.

5.4.Processing and Deployment of Requirements

During this step, the application requirements and structure that have been defined in the previous steps are received in the form of a type-level TOSCA template.

Then, a dedicated optimizer component (e.g., developed using Choco Solver [69] or BtrPlace [70]) is required to parse the received type-level TOSCA and solve the constraint programming problem associated with the optimization goals and placement constraints in it. The output of this process should include as a minimum the zone of the processing instances (cloud or edge), the provider(s) to be used, the optimal number of instances (per zone or even overall), and their flavor in a machine-readable format (e.g., json). Then, the network details can be specified and the Docker environmental variables can be updated. Finally, the instance-level TOSCA file reflecting the deployment can be generated. It is considered that the optimization process and instance-level TOSCA generation process should be triggered automatically once a new type-level TOSCA document is published to provide a fully automated deployment process.

5.5.TOSCA FaaS Application Definition Algorithm

To aid the comprehension of the proposed approach, the algorithmic steps that should be performed by a DevOps prior to the automatic generation of the TOSCA template are provided in Listing 2.

```
Input
F:Fragments
FPA:Fragment processing attributes//Defined in Table 3
FG:Fragment goals//Defined in Table 4

Algorithm
//Application Concept Definition
AC ← determine_coarse_grained_application _constraints
F ← determine_application _fragments

for fragment in F
  fragment.determine_application_paradigm
  fragment.determine_interfaces
  fragment.determine_environmental_variables
  fragment.determine_docker_properties
  fragment.determine_necessary_number_of_processing_instances

//Application Definition
for fragment in F
  for processing_attribute in FPA
    fragment.assign_value(processing_attribute,DevOps_value)

//Application Goal Definition
for fragment in F
  for fragment_goal in FG
    fragment.assign_value(fragment_goal,DevOps_value)

type_level_TOSCA_document ← TOSCAgenerator(AC,F)
return type_level_TOSCA_document
```

Listing 2. Algorithmic steps necessary for the definition of a fog application in type-level TOSCA.

5.6.Creation of updated type-level TOSCA

In the previous sections, the initial deployment flow is described illustrating the steps which should be taken to create a type-level TOSCA document. However, the model-driven nature of the approach allows it to be generalized to potentially handle any required adaptations. Indeed, when during the operation of the FaaS application it is determined that its deployment should be updated (e.g. that the container replicas of a function should be increased), it should be possible using the result of the processing of an adaptation manager (e.g. add two container instances) to update the model. Thus, depending on the intelligence of the adaptation manager, a lot of manual (and therefore error-prone) changes can be saved, and the burden of the DevOps which is related to the maintenance of the application can be decreased.

In the next sections, answers will be provided to each of the research questions set in the introduction of this work. First, a description of the type-level TOSCA semantic enhancements to allow the use of both cloud and edge resources will be performed. Then, details will be provided on the specific changes introduced to support FaaS applications. Following, the support provided for placement constraints and optimization criteria will be presented. Also, the methodology which can be followed to handle adaptations of the platform, leading to the generation of the new type-level TOSCA, is described, leading to the final sections which discuss these adaptations and the contribution of this work.

6. Improvements to TOSCA to model FaaS applications

The official TOSCA specification [7] provides sample configurations of processing nodes and cloud application topologies, however no reference is made to fog topologies, which commonly need processing nodes both on the edge and the cloud. While a fog topology could be implemented using TOSCA or any other DSL, an important issue that would not be solved would be the ability to use a different configuration for a fragment depending on its processing zone (cloud or edge), while still qualifying as the "same" fragment type for scaling purposes. A second challenge encountered when creating a service template is to accurately describe the model of the service while still allowing for its optimization.

To resolve the first issue and be able to seamlessly describe cloud-only, edge-only, and fog applications a modelling schema is introduced, decoupling the software from the hardware it is installed on but still maintaining their relationship. The second challenge is mainly tackled by the separation of concerns between type-level and instance-level TOSCA approaches.

Although the extensions of TOSCA involve numerous aspects of the deployment, the proposed changes to the language are non-intrusive and can be used within the original language features. This means that the core logic of TOSCA templates is not modified and that the model maintains the traditional structure of a TOSCA application. A summary of these changes is presented in Table 5.

Table 5. Overview of the extensions to core TOSCA concepts.

TOSCA Feature Extended	Summary of Changes	(Indicative) Extension(s)	Related Section(s)
Metadata	Introduction of new fields relating to optimization support	TimePeriod, CostThreshold	Section 7.1
Node_types	New node types are introduced to denote particular processing characteristics that are desired on a processing node	prestocloud.nodes.agent.faas	Section 6.4
Node_templates	New node templates are presented to allow Docker support, optimization support, as well as the expression of edge-related attributes and coordination paradigms	prestocloud.nodes.fragment.faas	Section 6.1
Policies	New policies are added to indicate the manner in which application deployment should be managed	prestocloud.placement.Gather, prestocloud.placement.Spread	Section 7.2
Relationships	A new TOSCA relationship indicates the relationship of the processing between components	prestocloud.relationships.executedBy.faas	Section 6.4
Capabilities	A new TOSCA capability indicates the special processing capabilities offered by some devices	prestocloud.capabilities.proxying.faas	Section 6.4

An important consideration related to the implemented improvements is the small learning curve of the type-level TOSCA model, which allows a DevOps to quickly become familiar with and inspect the structure of a cloud application.

6.1.Fragment and Processing Host Decoupling

A crucial aspect of the approach described in this work is the introduction of optimization capabilities to the TOSCA template. Unfortunately, as most of the existing approaches follow an analytical approach by specifying the topology in detail (i.e., specifying the provider to be used at the time of model formulation and the exact VM details), there is little room left for optimization. However, using the approach suggested as part of this work, the optimizer determines the exact processing zone and processing location. Furthermore, the number of fragment instances can be easily changed from components external to those involved in the TOSCA generation process (e.g., an adaptation Manager —see Figure 2) without adding unnecessary complexity to the type-level model.

These advantages are only possible if a clear distinction between the software components of the application from the hardware that they are installed on is made using distinct TOSCA structures. Each application, therefore, consists of fragment nodes (reflecting the software components), which are each related to a processing node (reflecting the hosting hardware). Fragment nodes contain a description of a fragment, which will run independently within the context of the application, while the hardware and operating-system level requirements that are imposed by each fragment are modelled on so-called “processing nodes”. Each processing node defines a relevant TOSCA node type and each “fragment node” corresponds to an instance of a TOSCA node template. While processing nodes are generic (and could possibly each be used by many fragments), fragment nodes are tightly coupled to the fragment that they describe—hence the naming of the nodes. Fragment nodes are mapped to

processing nodes, using “mapping nodes” (minimal TOSCA node templates). The definitions of both processing and fragment nodes are based on the hosting requirements expressed either through the annotations mechanism or the UI. Processing, fragment, and mapping nodes themselves are each defined in a new TOSCA node type, respectively. An example of the relationships between processing, fragment, and mapping nodes is illustrated in Figure 3.

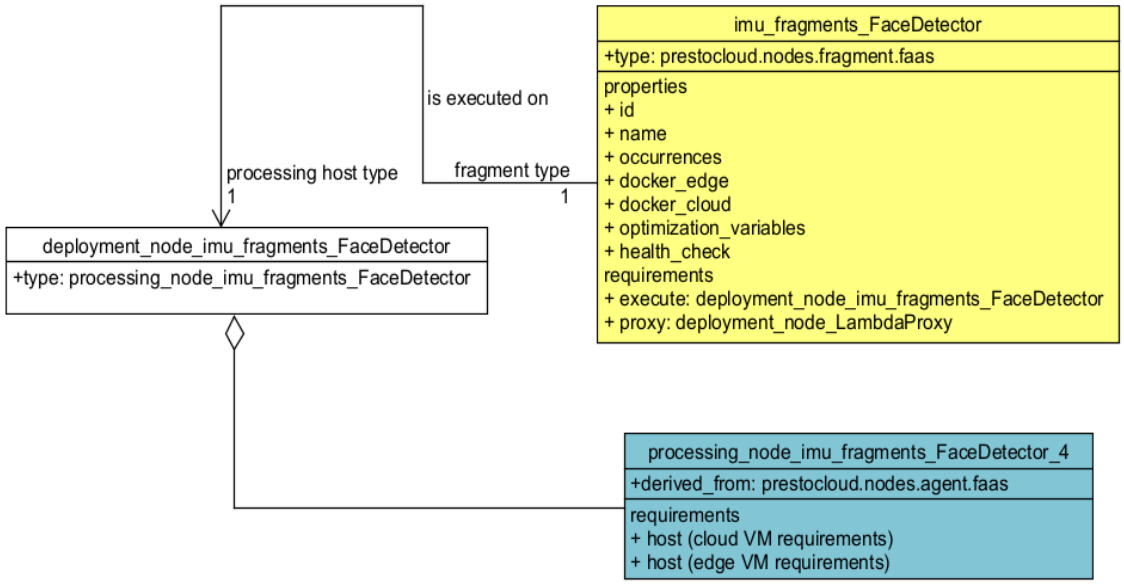


Figure 3. The relationships between fragment, processing, and mapping nodes.

An excerpt from the type-level TOSCA showing the sections related to the connection of application fragments (according to the motivating scenario of Section 3.3) with their respective processing nodes is shown in Listing 3.

```
deployment_node_imu_fragments_FaceDetector:
  type: processing_node_imu_fragments_FaceDetector_4

imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  ..................
  requirements:
    - execute: deployment_node_imu_fragments_FaceDetector
    - proxy: deployment_node_LambdaProxy
```

Listing 3. The connection of fragments with processing nodes.

In the example in Listing 3, the “deployment_node_imu_fragments_FaceDetector” is the mapping node, while the “imu_fragments_FaceDetector” is the fragment node. The “proxy” field of the fragment node indicates the mapping node that will host the Lambda Proxy of the topology (this node is not part of Listing 3—see Section 6.4 for more details). When the TOSCA blueprint is parsed, the TOSCA orchestrator will deploy a Lambda Proxy, which will enable the deployment of the FaceDetector fragment as a serverless function.

The processing node defining the type of mapping node (processing_node_imu_fragments_FaceDetector_4) that will be used to deploy the above fragment (imu_fragments_FaceDetector) is specified in Listing 4.

```
processing_node_imu_fragments_FaceDetector_4:
 description: A TOSCA representation of a processing node
 derived_from: prestocloud.nodes.agent
 requirements:
  - host:
    capability: tosca.capabilities.Container
    node: prestocloud.nodes.compute
    relationship: tosca.relationships.HostedOn
    node_filter:
     capabilities:
      - host:
        properties:
         - num_cpus: { in_range: [1, 2] }
         - mem_size: { in_range: [1024 MB, 2048 MB] }
         - storage_size: { in_range: [4 GB, 32 GB] }
      - os:
        properties:
         - architecture: { valid_values: [x86_64, i386] }
         - type: { equal: linux }
         - distribution: { equal: ubuntu }
      - resource:
        properties:
         - type: { equal: cloud }
  - host:
    capability: tosca.capabilities.Container
    node: prestocloud.nodes.compute
    relationship: tosca.relationships.HostedOn
    node_filter:
     capabilities:
      - host:
        properties:
```

```
      - num_cpus: { in_range: [1, 2] }
      - mem_size: { in_range: [1024 MB, 2048 MB] }
      - storage_size: { in_range: [4 GB, 32 GB] }
    - os:
      properties:
      - architecture: { valid_values: [ arm64, armel, armhf ] }
      - type: { equal: linux }
      - distribution: { equal: raspbian }
    - resource:
      properties:
      - type: { equal: edge }
```

Listing 4. The description of a processing node.

As can be noted in Listing 4, the processing node definition allows for different requirements for the cloud and the edge version of a fragment. This facility can be used to adjust the processing requirements on edge devices, to account for the disparity in performance between them and cloud instances.

Furthermore, in the case of fragments that should be only executed on edge devices, a further constraint can be specified to ensure that candidate devices possess the necessary sensors to acquire input. To represent this constraint, a sensors property was introduced in the extended TOSCA compute node specification, which includes a list of all required sensors. Any device that does not possess one or more of these sensors is not eligible to host the particular fragment.

An example of the sensors property used to limit (using the TOSCA nodefilter structure) possible hosts to those possessing a microphone and camera is shown in Listing 5.

```
- sensors:
  properties:
    - microphone: { equal: "/dev/snd/mic0" }
    - camera: { equal: "/dev/video/camera0" }
```

Listing 5. Example of the "sensors" property.

6.2. TOSCA Specification of Fragment Nodes

Fragments are the central elements in the extended TOSCA document, as they contain the actual business logic that will be carried out by the cloud application. Fragments

can represent software components that run on the edge, on the cloud, or both—using the same or a different set of properties. The connection of fragments with processing nodes and with their processing requirements is achieved using a new TOSCA relationship.

The definition of a fragment node begins with a declaration of the type to which it belongs. Immediately afterward follows the property segment, which begins with generic information on the id and the name of the fragment. This information is followed by the scalable and occurrences fields, which indicate if the fragment is scalable and the number of instances that are needed for it to operate, respectively. The specification of the number of occurrences at the fragment level drastically reduces the size of type-level TOSCA files (the alternative would be to copy large, identical specification blocks) and aids their comprehensibility. It is considered a necessary step to allow the DevOps to define the number of instances available to a fragment as a starting point to deploy the topology.

The major share of the fragment specification belongs to the definition of Docker properties, either for edge or cloud versions of the fragment, or both. These properties include the Docker image, the registry, the environmental variables, the specification of port forwarding within Docker, and a custom Docker command line that can be executed by the fragment (if needed). The ability to specify different properties on the cloud and edge versions of a fragment provides the means for applications to adapt their execution according to the resources of the host (generally inferior in an edge device compared to a cloud VM). This is exemplified in Listing 6, where lower precision is used for the edge version of FaceDetection and for more iterations in order to diminish the probability of an edge device becoming overloaded (it is presumed here that the face detection algorithm can either iterate more times using a coarse model to detect faces in the image or fewer times by running calculations with higher precision). On the other hand, the cloud version can operate at full effectiveness, requiring a much lower number of iterations to verify the result.

Further, it is important to mention that environmental variables may be dynamic, using the get_property function available to TOSCA. Thus, IP addresses that are unknown at the time of the deployment of the fragment may be denoted by a variable, which is later replaced in instance-level TOSCA by an appropriate IP address. Following the specification of Docker properties, the optimization variables section contains the weights for the cost and distance criteria, while the friendliness criterion accepts a list of providers and the weight that is assigned to each of them. The optimization criteria for each fragment of the motivating scenario are specified in Table 2. For example, the DevOps has set for the VideoStreamer fragment a weight value of 8 for distance, a weight value of 2 for cost, a weight value of 1 for the friendliness of the "Azure" provider, and a weight value of 5 for the friendliness of the "AWS" provider (meaning that providers favoring low latency should be favored, then the AWS provider, then providers offering low cost, and then any provider—higher weights imply a higher preference for this criterion). This segment is concluded by the definition of a custom health check command line and an integer interval between two successive health check commands.

The last elements in the specification of a fragment are the mapping node that will execute the particular fragment and the Lambda proxy which is related to the particular fragment (if it is a FaaS fragment - for more details see Section 6).

Listing 6 provides the full specifications for an application fragment.

```
imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  properties:
    id: 3
    name: imu_fragments.FaceDetector
    scalable: true
    occurrences: 1
    docker_edge:
      image: "face_detector_edge:latest"
      registry: "local.prestocloud.test.eu"
      variables: { "PRECISION": "50", "ITERATIONS": "10" }
    docker_cloud:
      image: "face_detector_cloud:latest"
```

```
    registry: "prestocloud.test.eu"
    variables: { "PRECISION": "100", "ITERATIONS": "2" }
  optimization_variables:
    cost: 1
    distance: 1
    friendliness: { "aws": "5", "gce": "0", "azure": "1" }
  health_check:
    interval: 1
    cmd: "curl health.prestocloud.test.eu FaceDetector"
requirements:
  - execute: deployment_node_imu_fragments_FaceDetector
  - proxy: deployment_node_LambdaProxy
```

Listing 6. Full application fragment specifications for the FaceDetector fragment.

6.3. Description of Instance-Level TOSCA

Although this work emphasizes the modelling capabilities offered by the introduction of type-level TOSCA, instance-level TOSCA is also a major asset for the reconfiguration of the application topology. As the instance-level document contains the processing zone selected for each fragment, this information can be consumed by components external to the generation of TOSCA templates to understand the mixture of fragment instances that were deployed on edge devices and cloud VMs and to improve the quality of the updated blueprint (e.g., by adding one more instance for a component that is chiefly deployed on edge devices, which typically have lower processing power).

The creation of the instance-level TOSCA document should be automatically triggered each time a new type-level document is produced. This involves as a first step the extraction of the information contained in the type-level TOSCA document by the optimizer. The cost constraints and the related time interval included in type-level TOSCA are evaluated to create an average cost that is admissible for the topology. Then, the collocation and precedence constraints are evaluated, as well as the optimization preferences provided in each fragment, to determine the final deployment of the application (for more details on the optimization process see Section 7.3). It is clear that the policies segment contained in type-level TOSCA is not

necessary in instance-level TOSCA, as the constraints included there are taken into account during the allocation of resources. If the optimizer can produce a valid configuration satisfying the constraints of the topology described in the type-level document, this configuration should be sent to the instance-level TOSCA generator, which should produce the final instance-level TOSCA document.

The average cost of the suggested deployment is evaluated against the maximum admissible cost threshold of the DevOps. If it is lower, it is admitted and the deployment can be implemented. The processing resources, networking configuration details, as well as the cost of the VM or edge device used are then added for each fragment to new "node_type" definitions of the processing nodes that will host them. The TOSCA type of each processing node is different to represent different edge devices and cloud providers. Thus, provider-specific information can be abstracted in the definition of certain normative TOSCA "provider types".

Besides, the instance-level TOSCA template shares with the type-level TOSCA template the definitions of fragments, as well as the relationships between fragment nodes and mapping nodes.

An example description of a processing node in instance-level TOSCA is described in Listing 7.

```
processing_node_fragments_FaceDetector_1:
  type: prestocloud.nodes.compute.cloud.amazon
  properties:
    type: cloud
    network:
      network_id: s-gbdpnc4s
      network_name: subnet1
      addresses:
        - 192.168.1.1
  capabilities:
    resource:
      properties:
        type: cloud
        cloud:
          cloud_name: amazon_public1
          cloud_type: amazon
```

```
    cloud_region: us-east-1
host:
  properties:
    num_cpus: 2
    mem_size: 4.0 GB
    disk_size: 50 GB
    price: 0.120000
```

Listing 7. Processing node specifications for instance-level TOSCA.

In instance-level processing nodes, the information described in the form of constraints in type-level TOSCA should be concretized into specific details. The number of CPUs, the memory, and the disk size are all fixed values; the cloud provider and the cloud region are also chosen. These fixed values come from the solving process of the optimizer (see Figure 2), which considers the available hosting candidates with respect to the pre-defined optimization goals, as detailed in Section 7. Moreover, networking information is available for the particular instance.

6.4.FaaS Paradigm architectural elements definitions

The new node types proposed for TOSCA both allow the representation of FaaS and other distributed software paradigms, in which there is a coordinator of execution that handles a number of workers. It is assumed that these distributed software paradigms are followed by one or more of the application components. Although here the definition of FaaS node types is proposed – so that they can also be exploited by other TOSCA applications – the definition of suitable node types for other distributed execution paradigms is feasible as illustrated in [35].

FaaS-based applications are assumed to consist of a set of application fragments that are independent of other application components and have a self-contained execution flow. Fragments are assumed to be hosted inside Docker containers, which are in turn hosted inside VMs. Access to fragments is allowed through REST calls, which are managed by a publicly-facing load-balancer component. If more than one fragment type are managed by the load-balancer, the component is referred to as a Lambda

Proxy, since it serves as a proxy for AWS-Lambda-like, serverless functions. Unlike some serverless platforms, which limit the processing time and the languages that can be used to develop functions, the proposed approach supports all fragment types that can be dockerized, running for any desired amount of time.

Fragments following the FaaS paradigm use the custom prestocloud.nodes.fragment.faas fragment type. The “proxy” field of the fragment type accepts the name of the Lambda Proxy mapping node that will manage requests to this fragment. FaaS fragments are installed on FaaS agents (workers), which are modelled using the prestocloud.nodes.agent.faas type. FaaS agents satisfy by definition the prestocloud.relationships.executedBy.faas relationship required by FaaS fragments. FaaS Lambda Proxies are modelled with the prestocloud.nodes.proxy.faas type and possess the prestocloud.capabilities.proxying.faas capability, which allows them to coordinate worker agents hosting a FaaS fragment. The relationships between FaaS fragments, their executing processing nodes, and the Lambda Proxy can be seen in Figure 4.

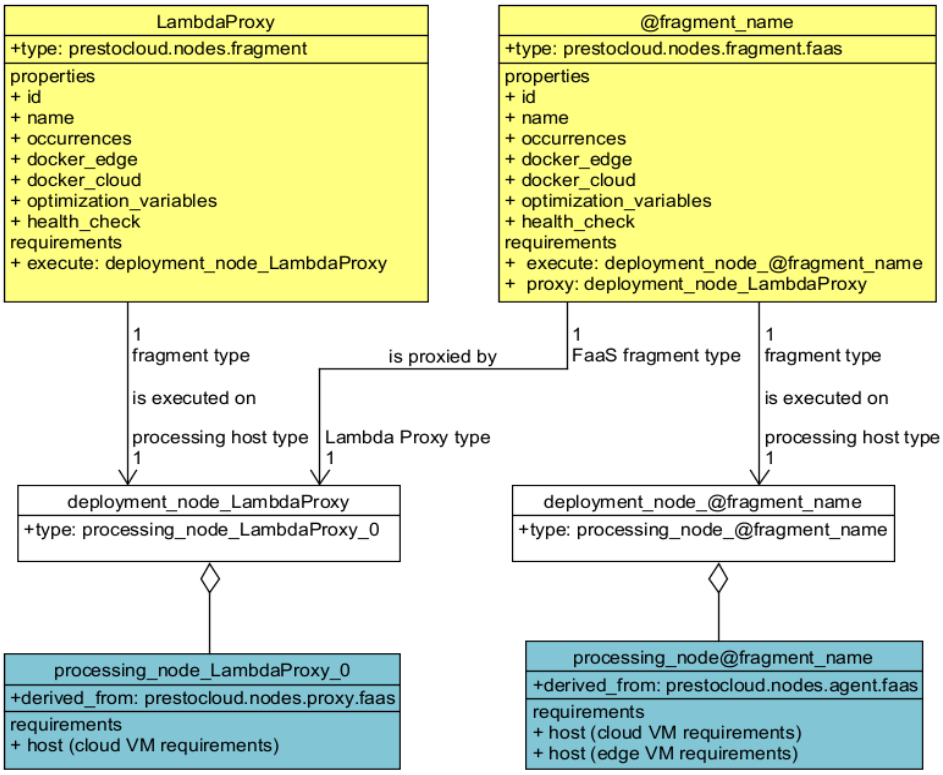


Figure 4. The relationships between the fragment, processing, and mapping nodes of the Lambda Proxy and the proxied FaaS fragments

As mentioned before, additional software paradigms can be defined using this approach. Moreover, more than one of these software paradigms can co-exist in the same FaaS application and each may function independently from the others. As long as each of these software paradigms will have a coordinator node, which will handle many agents, the relationships between coordinator, processing, and fragment nodes within an application topology are depicted in Figure 5.

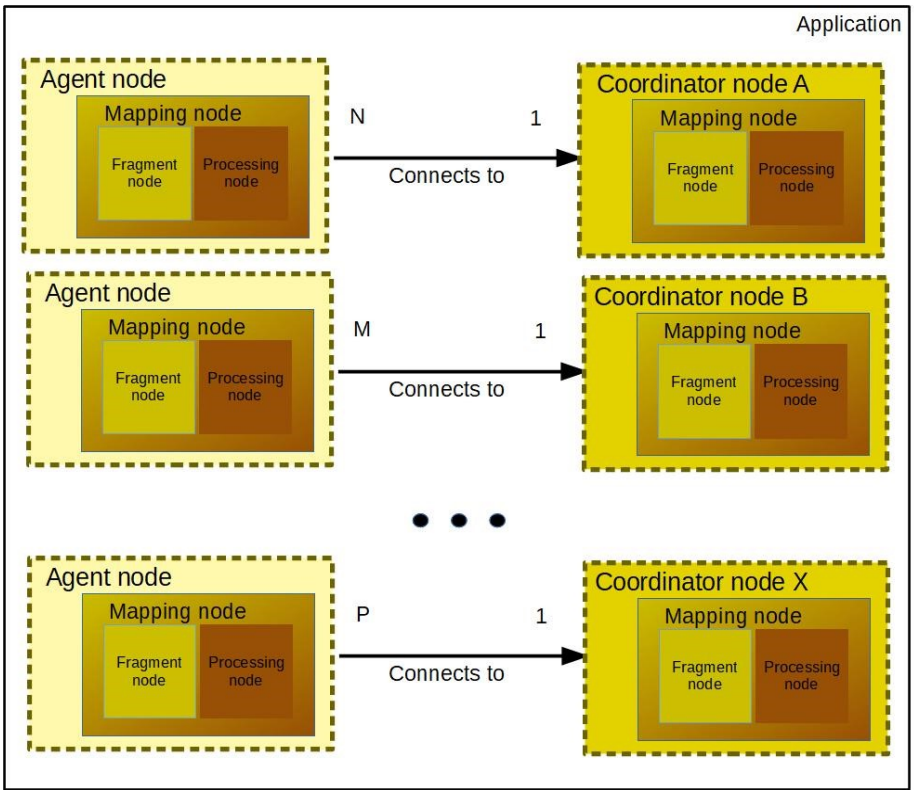


Figure 5. The structure of the topology template segment in a coordinator-driven, type-level TOSCA blueprint. Multiple agent nodes may be connected to one coordinator node and there can be an arbitrary number of coordinator nodes.

7. Optimization and Application Constraints in FaaS applications

The TOSCA extensions that are introduced include support for the optimization of the deployment of fragments. Type-level TOSCA processing nodes specify ranges of satisfactory values for most of their attributes (e.g., cpu cores, available ram and disk space). This permits a reasonable number of alternative providers to be researched for the availability of similar VMs (in resources), while ensuring a minimum performance standard. The final selection of provider resources and edge devices should obey the coarse-grained and fine-grained constraints set for the application, as well as any placement policies set for one or more fragments.

7.1.Coarse-Grained Application Constraints

The vanilla TOSCA language specification [7] already permits the definition of a metadata segment inside the TOSCA file. In this work, this native construct will be used to hold extended constraints and preferences of the application. Its key-value body contains: the data relevant to the business goals pursued by the application; the providers that are preferred or excluded; and the budget constraints that should be respected. These constraints can be used by the optimization engine to select the most appropriate flavor and locations for the processing nodes.

An example of the usage of metadata fields to denote some of the constraints outlined above is presented in Listing 8:

metadata:
template_name: IMU generated types definition
template_author: IMU
template_version: 1.0.0-SNAPSHOT
CostThreshold: 1000
TimePeriod: 720
ProviderName_0: OpenStack_local

ProviderRequired_0: false
ProviderExcluded_0: true
MetricToMinimize: Cost

Listing 8. Example of metadata fields used for application-level constraints.

In the above example, aside from some generic informative fields concerning the particular template version, the name, and the author, definitions exist for the business goals, provider, and budget requirements related to the application as a whole. The budget available is set to 1000 monetary units (e.g., euros), which should be used over a time period of 720 h. The "OpenStack_local" provider is set to be excluded, and the primary objective to be minimized is set to "cost".

7.2.Fine-Grained Constraints and Optimization Criteria

As mentioned in Section 6.2, in each fragment definition, a number of optimization criteria are identified, namely the cost, distance, and friendliness. The cost optimization criterion reflects the monetary cost of choosing a particular hosting VM for a time period. The distance optimization criterion reflects the distance of the host of the fragment from the centroid of a user group related this fragment, e.g. the edge devices contacting it. The friendliness criterion reflects a preference towards a particular cloud provider (for any reason, e.g., data locality). While these criteria are modelled in a specific construct, more optimization criteria can of course be defined to supplement or replace the above. An example of the specification of optimization criteria appears in Listing 6.

Additionally, these application constraints can be coupled with a set of tools that enables the DevOps to guide the deployment of fragments by considering their relationships. It is suggested to implement these using a set of proper placement optimization policies. In the context of TOSCA, three different optimization policies are introduced: collocation policies, anti-affinity policies, and precedence policies. The formal definitions of these policies appear in Listing 9.

Input

x_i:Cloud providers

f_i:Fragments

d_i:Devices

Provider: f → x Function mapping from fragments to providers

Hosting: d →f Function mapping from hosting (edge) devices to fragments

DeploymentTime: f → $\mathbb{R}^+$ Function mapping from fragments to the positive real numbers

Policy definitions

Collocated (f_i,f_j) → Provider(f_i) = Provider (f_j)

Collocated(f_i,f_j,f_k,…,f_{n-1},f_n) = **Collocated**(f_i,f_j) and **Collocated**(f_i,f_k) **and** … **and Collocated**(f_i,f_n)

and Collocated(f_j,f_k) **and**…**and Collocated**(f_j,f_n) **and** … **and Collocated**(f_{n-1},f_n)

Antiaffinity(f_i,f_j)→Provider(f_i) ≠ Provider(f_j)

Antiaffinity (f_i,f_j,f_k,…,f_{n-1},f_n) = **Antiaffinity**(f_i,f_j) **and Antiaffinity**(f_i,f_k) **and** … **and Antiaffinity**(f_i,f_n) and **Antiaffinity**(f_j,f_k) **and** … **and Antiaffinity**(f_j,f_n) **and** … **and Antiaffinity**(f_{n-1},f_n)

Precedence(f_i,f_j) → DeploymentTime(f_i) < DeploymentTime(f_j)

Precedence(f_i,f_j,…,f_n) → DeploymentTime(f_i) < DeploymentTime(f_j) < ⋯ < DeploymentTime(f_n)

Excluded(f_i,(d_1,d_2,…,d_n)) → **not** Hosting(d_1,f_i) **and not** Hosting(d_2,f_i) **and** … **and not** Hosting(d_n,f_i)

Listing 9. Formal definitions of optimization policies.

The collocation policies which are introduced as part of this work indicate that a fragment should be collocated with other fragments (using the same cloud provider), unlike the collocation policies that are briefly mentioned in the TOSCA specifications, which imply the use of the same compute node. This allows low-latency communication and results in improved compatibility and communication between processing nodes. However, the optimization component cannot consider the option of using different cloud providers for the fragments to lower the total costs. In

the motivating example, a collocation policy is needed for the VideoStreamer and VideoTranscoder fragments (Table 2).

Anti-affinity policies specify that a fragment should not be collocated with other fragments. This results in the placement of this fragment and all target fragments in different cloud providers (and is, thus, different from the anti-collocation policies briefly mentioned in the TOSCA specifications). Using this policy can enhance the security of a critical information system that is communicating with a potentially vulnerable component (as it is easier to isolate systems in case of a breach) or can ensure that processing can be decoupled, location-wise. However, this also means that the optimization component cannot request instances from the same provider for the fragments, and as a result some of the lower-cost options might be lost. In the motivating example (Table 2), an anti-affinity policy is needed for the AudioCaptor and PercussionDetector to ensure that (violent percussion) detection happens reliably and quickly (i.e., away from edge nodes on which AudioCaptor fragments are hosted).

Precedence policies describe that fragments should be instantiated and deployed in the order that is mentioned. An advantage of precedence policies is that all required interfaces—indicating data needed from a data flow for each component—are automatically satisfied by the time they are instantiated. Precedence policies guarantee the satisfaction of interfaces, however increased deployment time is required in return, as Docker containers should be spawned sequentially.

Device exclusion policies ensure that fragments are optimally scheduled for processing at the edge. They enhance the response of the system by marking a certain set of devices as unsuitable for deployment—therefore being excluded from the scheduling of instances of a particular fragment. The suitability of a device for a fragment depends on historical data processing results and the availability status, which can be detected and analyzed using machine learning techniques (for more details see [71]). The details of such a component are not detailed here, since this is considered out of the scope of this article.

The placement policies are modelled at the level of TOSCA using the fragment nodes. They offer a significant benefit over the usage of native TOSCA relationships, in that they permit the easy visualization of the most important constraints associated with the deployed application. In addition, the implementation of their enforcement is more straightforward compared to the resolution of TOSCA relationships between fragments.

An example of the four deployment policies based on the motivating example is included in Listing 10.

```
Topology_template:
  policies:
    - collocation_policy_group_0:
      type: prestocloud.placement.Gather
      targets: [ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder ]

    - anti_affinity_policy_group_0:
      type: prestocloud.placement.Spread
      targets: [ imu_fragments_PercussionDetector, imu_fragments_AudioCaptor ]

    - precedence_policy_group_0:
      type: prestocloud.placement.Precedence
      targets: [
imu_fragments_VideoStreamer,imu_fragments_VideoTranscoder,imu_fragments_FaceDetect
or,imu_fragments_MultimediaManager,imu_fragments_AudioCaptor,imu_fragments_Percus
sionDetector ]

    - exclude_fragment_from_devices_0:
      type: prestocloud.placement.Ban
      properties:
        excluded_devices:
          - "a6f2:d8bd:bf45a:de2a:d1e8:5f58:c256:0492",
          - "c2c1:def1:2c38:c83b:6b0d:b7bd:a0d2:95c2",
          - "b3ef:58d8:39d0:86ce:81d2:6e93:5f7d:23cd",
          - "b28e:9f32:2076:3599:39c3:6cc5:794a:5140"
      targets: [ imu_fragments_VideoStreamer ]
```

Listing 10. Full example of the available deployment policies.

7.3.Constraints and Optimization Handling

The enforcement of the optimization policies presented in this work is delegated to the optimization engine that consumes the type-level TOSCA. Additionally, in order to provide a clearer understanding of the effects of each optimization policy, the necessary steps to be performed by any optimizer implementation process are described.

It is considered that the optimizer first retrieves the available cloud provider VM types and edge devices that can be used for the deployment. Then, any device exclusion policies are applied and the excluded edge devices are removed from the candidate hosts. If there are not any contradictory policies (e.g., an anti-affinity policy and a collocation policy set for the same set of fragments, or cyclic precedence policies), a list of valid configurations that satisfy all collocation and anti-affinity policies is proposed; otherwise, the list of valid configurations is set as empty. In the creation of valid configurations, precedence is given to the assignment of fragment instances to edge processing hosts satisfying the requirements of a fragment. If the list of valid configurations is not empty, the configurations are sorted according to the optimization criteria that have been defined, and the best configuration that satisfies the global constraints should be chosen to be translated to instance-level TOSCA.

Otherwise, if no configuration is found to satisfy the constraints and deploy all fragment instances, the deployment fails and the DevOps should resubmit a new type-level template. Finally, at deployment time, the containers of different fragments should be started according to the priority, which is set in one or more precedence policies.

The process described above appears in pseudocode in Listing 11. No optimization of the data structures and algorithmic logic is performed, as the intention is to provide an easy-to-follow overview of the proposed process.

```
Input

X:Cloud providers
F:Fragments
D:Candidate Edge Devices

CLP:collocation policies
AAP:anti-affinity policies
PRP:precedence policies
DEP:device exclusion policies

Algorithm

for fragment in F
  for collocation-policy in fragment_CLP
    for anti-affinity policy in fragment_AAP
      if collocation-policy contradicts anti-affinity-policy then return
FAILED_DEPLOYMENT

for precedence-policy in fragment_PRP
  for other-precedence-policy in fragment_PRP
    if precedence-policy contradicts other-precedence-policy then return
FAILED_DEPLOYMENT

for fragment in F
  for device in fragment_DEP
    D ← D-{device}//remove the device from the eligible hosts

Configurations ← find_eligible_configurations (F,D,X)
Configurations ← apply_coarse_grained_application_constraints (Configurations)

if Configurations is not Empty
  maximum_utility ← -∞
  best_configuration ← None
  for configuration in Configurations
    configuration.utility ← calculate_utility_from_optimization_criteria (configuration)
    if configuration.utility > maximum_utility then
      maximum_utility ← configuration.utility
      best_configuration ← configuration
  return best_configuration
else return FAILED_DEPLOYMENT
```

Listing 11. Optimization process pseudocode.

8. FaaS application elasticity with Severity-based elasticity rules

8.1. Elasticity Rules

In this work, elasticity rules are defined as directives which indicate firstly the QoS limits of normal operation of an application, and secondly the horizontal elasticity action which should be taken to accommodate the needs of the application when these limits are trespassed – scaling in or scaling out. The QoS limits can be specified in terms of any measurable metric, including custom metrics. Rules are assumed to be entered by a DevOps who possesses significant experience and knowledge on the application which is deployed and monitored. Contrary to static rules which specify one concrete set of conditions, and one concrete set of actions, the proposed elasticity rules require less input for their definition. The DevOps should specify the QoS conditions which trigger the rule, but in the action part of the rule, only the scaling direction is required and not the number of instances which should be added/removed. This allows a flexible response action, which may be decided using a variety of techniques as demonstrated in Section 8.4. With elasticity rules, the triggering conditions of a rule and the adaptation – the concrete actions – are separated conceptually; the suggested approach assumes that a DevOps is primarily interested in defining the criteria indicating that the application functions correctly, rather than the exact adaptation action which will be followed.

The format of an elasticity rule can be found in Listing 12.

$$
\begin{aligned}
&\textbf{For component\_id} = component\_id \\
&\textbf{if } (attribute\_1) \leqslant attribute\_1\ value\ \textbf{and}\ (attribute\_2) \\
&\qquad\qquad \leqslant attribute\_2\ value\ \textbf{and} \ldots \textbf{and}\ (attribute\_n) \\
&\qquad\qquad \leqslant attribute\_n\ value \\
&\textbf{within Timewindow} \\
&= Timewindow \textbf{and}\ Cooldownperiod\ \textbf{has passed from previous adaptation} \\
&\textbf{then}\ Scale\_out\ /\ Scale\_in
\end{aligned}
$$

Listing 12. Elasticity rule format.

Any number of QoS attributes connected with the "AND" logical operator can be entered. While "OR" logical operators are not allowed to be used alongside "AND" logical operators, multiple elasticity rules can be enforced in parallel. Additionally, non-bounded attributes, e.g., response time, are supported, provided that a threshold is set by the DevOps. Furthermore, the DevOps defines the time-window over which this rule is calculated (e.g., 10 minutes), and the cooldown period which should elapse between two triggerings of the rule.

An instantiated example of an elasticity rule appears in Listing 13:

$$\textbf{For component\_id} = VideoTranscoder$$
$$\textbf{if } AverageCPU_{cluster} > 70\% \textbf{ and } AverageRAM_{cluster} > 70\%$$
$$\textbf{within Timewindow} = 10\ minutes$$
$$\textbf{and } 30\ minutes \textbf{ have passed from previous adaptations then } Scale\_out$$

Listing 13. Elasticity Rule example.

When the thresholds set by the DevOps for the monitoring attributes expressed in an elasticity rule are violated, a violating situation is detected, and an elasticity rule is triggered. Onwards, a violating situation shall be referred to simply as a 'situation'.

8.2.Situation Severity

Once a situation is detected, the adaptation manager should be invoked, to determine the adaptation action which should be taken. To assist this decision (horizontal scaling in this work), the 'Severity' of the situation is assessed. Severity quantifies the rough magnitude of the violation of the thresholds of the attributes used in the elasticity rule. The values of all violating attributes are used, and weights are assigned to each of them to indicate their relative importance. Higher values of Severity indicate that more pronounced changes to the application should be made (i.e., more VMs/containers hosting a function should be added/removed).

The Severity of any detected situation $V_{violating} = (v_1, v_2, \ldots, v_n)$ is determined as shown in Equation 1.

$$Severity(\boldsymbol{V}_{violating}) = \sqrt{\sum_{\boldsymbol{i=1}}^{\boldsymbol{n}} w_i \cdot (Normalized(v_i))^2}$$

Equation 1. Calculation of the Severity of a situation.

In Equation 1, v_i are the individual, threshold violating QoS attribute values comprising the particular situation, w_i are their respective weights and n is the number of attributes included in the triggered elasticity rule. For each of the v_i values it is assumed that $v_i \in [0,1]$.

While the definition of Severity allows for the usage of different weights for each of the attributes being evaluated, in the remainder of this work it is assumed for simplicity that all weights are equal to 1. Following the definition of Severity (Equation 1), the maximum Severity value for a situation is observed when all attributes have reached their maximum normalized values, i.e., 1 and is equal to $\sqrt{n}$.

Having chosen the weight values for each attribute, the calculation of Severity relies on obtaining the normalized values for each of the attributes. For each attribute v_i - threshold t_i pair in the rule, the normalization formula in Equation 2 is used in cases of attributes that need to be greater than their threshold and Equation 3 is used in case of attributes that need to be less than their threshold.

$$Normalized(v_i) = \frac{abs(v_i - t_i)}{abs(\text{maximum}(\text{attribute}_\text{i}) - t_i)}$$

Equation 2. Variable normalization in the greater-than case.

$$Normalized(v_i) = \frac{abs(v_i - t_i)}{abs(t_i - \text{minimum}(\text{attribute}_\text{i}))}$$

Equation 3. Variable normalization in the less-than case.

Equations 2 and 3 are applicable in the case of attributes which are bounded. In the case of unbounded attributes – for example response time, the denominator of Equation 2 and Equation 3 is unknown, and therefore the normalized value is not computable. In such cases, the unknown or unavailable bounds of the attribute can be estimated using past observations. For the estimation of the bounds Chebyshev's equation [72] can be used, or a custom percentile value (e.g 90^{th} percentile) or a 'sufficiently' high or low value under or over which all values will also be considered to be minimum or maximum respectively. As the last two estimation modes are straightforward, the first estimation mode is elaborated on below.

To estimate bounds using Chebyshev's equation it is assumed that each attribute follows an arbitrary distribution and that the attribute – random variable is integrable, has a finite expected value μ, a finite non-zero variance σ^2 and a standard deviation σ. It is considered that determining that 96% of the samples of the attribute are within an upper and a lower bound, provides an adequate estimation of the maximum and the minimum value respectively. In this case, only 4% of the samples will be outside these boundaries. Substituting this probability value in the left handside of [72], and solving for k it is determined that k=5. The conclusion is then that the contrapositive argument, i.e., that all samples of a distribution will be contained inside the boundaries with a probability of 96%, is true as long as the samples are within 5 standard deviations of its mean value.

To illustrate, using the example of response time – which does not have an upper bound – let it be assumed that current observations for this attribute indicate an expected value μ = 200msec, with a standard deviation of 30msec. Then, the probability of measuring a response time X, being retarded more than $5 \cdot 30\text{msec} = 150\text{msec}$ from the expected value (200msec) is less than 4%. The upper bound of the distribution can then be estimated to be 200+150=350msec with 96% probability.

The expected value and the standard deviation of the distribution is calculated as the arithmetic mean over a window of the last z samples of the distribution – a number

which can be configurable. The greater the value of z, the more the arithmetic mean will approach the expected value of the distribution (provided that the distribution is unchanged). The smaller the value of z, the more susceptible are the bounds to changes in the distribution of unbounded variables. 'Upper' and 'lower' bounds are updated dynamically using a sliding event window.

8.3.Severity Zone Calculation

Although Severity provides an assessment of the Severity of a situation, a means is needed to group situations by their Severity. Considering that a detected situation with p attributes is represented as a point in p-dimensional space by $V_{violating}$, situations having similar Severity values form circular annuli, spherical shells or hyper-spherical shells (depending on whether $p = 2, p = 3$, or $p \geq 4$ respectively). These regions are called 'Severity zones' and are used by the Simple severity zones and Relative severity zones techniques presented below.

The rationale behind both of these techniques is that similar Situations in terms of Severity should result in same adaptation actions. For this reason, 'Severity zones' are introduced. The real number interval $[0, \sqrt{n}]$ reflecting all possible Severity values for a given set of metrics is divided into m equal sub-intervals. Each such sub-interval is a Severity zone (Table 6).

Table 6. Severity zones bounds.

Severity Zone	Sub-interval lower bound	Sub-interval upper bound
1	**0**	$\frac{\sqrt{n}}{m}$
2	$\frac{\sqrt{n}}{m}$	$\frac{2\sqrt{n}}{m}$
3	$\frac{2\sqrt{n}}{m}$	$\frac{3\sqrt{n}}{m}$
…		
m	$\frac{(m-1)\sqrt{n}}{m}$	$\sqrt{n}$

Zones containing situations with Severity values with numbers closer to 0 will result in milder adaptation actions, while Severity zones closer to $\sqrt{n}$ (the maximum value of Severity) will result in more instances being added to/removed from the application. Choosing higher values for m indicates that finer-grained adaptation actions are required. On the other hand, choosing lower values for m increases the amount of historical data available for each adaptation action (if Severity values are logged), which might be beneficial if zone adaptation is considered.

Adaptation decisions are made under the assumption that the Severity value calculated from a random situation can belong to each zone with equal probability – in order not to bias the triggering of a particular adaptation decision – and thus obtain results which are relevant to the situations included in the particular Severity zone. To satisfy this requirement, it is needed to define all Severity zones to have equal area (or volume, or hypervolume, in the case of 3 and more attributes-dimensions). Furthermore, it is required that equal Severity values should trigger the same adaptation actions. In the case of two attributes, finding an analytical expression to determine the splitting of a square area zone to three equal zones, also satisfying the requirement for equal Severity values, is a difficult but nevertheless achievable task. However, as the number of dimensions increases to three or more, the problem becomes greatly exaggerated. This means that a solution based on an alternative mathematical principle should be found.

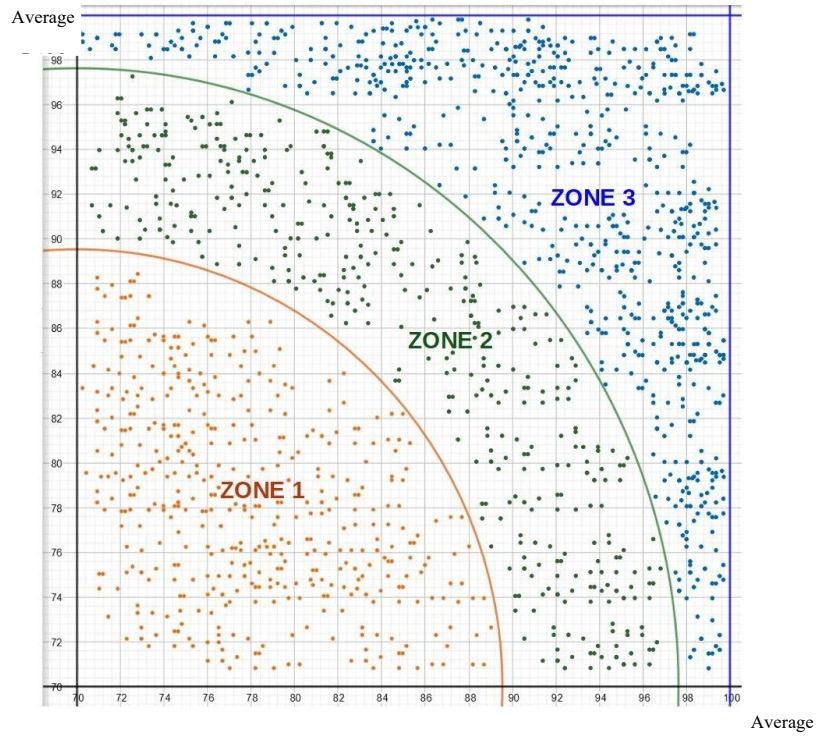


Figure 6. The situation space of the example Elasticity rule of Listing 13, split into 3 equal Severity Zones.

Such a solution is possible, if a solution based on a random simulation is considered. For this purpose, the normalized monitoring attribute values for situations having k normalized attributes are simulated, by retrieving random points s_i from the Cartesian product of possible normalized values of each of the k attributes. Since normalization converts the values of attributes to percentages, it is required that $s_i \in [0,1]x[0,1]x...x[0,1]$. Each point s_i reflects the values of the monitoring attributes of a possible detected situation. The choice of each s_i is uniformly random, so it can be assumed that the number of points which should belong to each of the Severity zones will be equal, if their volume (i.e., event space) is equal. Thus, a number of p random points is chosen and sorted. Then, supposing that there exist m zones (areas) which should be determined, the ratio $z = \lfloor p/m \rfloor$ is determined, where z is the number of points per area. Finally, the maximum Severity values for the first m-1 zones are calculated (the last Severity zone always has the value of $\sqrt{n}$ as already stated above) by calculating the Severity value of the $(i \cdot z)$th element, where $1 \leq i \leq m-1$.

When these values are known, the Severity zone of a detected situation can be determined by comparing the Severity value calculated to the Severity values of each Severity zone.

The complex calculations outlined above, are based on a simple Markov Chain Monte Carlo simulation which is a well-established technique in the field of engineering. Although the calculation of complex integrals which is needed in this – and in more complex cases – is difficult to perform in an analytic fashion, the Markov Chain Monte Carlo simulation provides a satisfactory approximation which can readily be used.

8.4.Cloud adaptation techniques

In this Section different techniques based on Severity are presented, each of which guide application adaptation in a different manner. Further and in order to highlight the novelty of this approach, adaptation techniques based on the commercial offerings of major cloud vendors are also presented. The latter are used as a baseline in Section 9.2 and Section 9.3 to aid evaluating the usefulness of Severity. Sections 8.4.1 and 8.4.2 describe two techniques which are based on commercial offerings discussed in the state-of-the-art analysis (Section 4). In Sections 8.4.3 through 8.4.9 seven Severity-based techniques are defined, which can each govern the scaling of an application. Each of these techniques serves adaptation using a different way to spawn and deallocate instances. The efficiency of each design based on simulations is portrayed in Section 9.2, and for some of these techniques in a realistic setting, in Section 9.3.

While designing any technique using the Severity value, one should be prepared to balance the detail of the response between small load fluctuations and the need to handle sudden workload peaks or troughs, which theoretically can be several times bigger/smaller than the current workload. Here, seven basic flavours are discerned: Absolute severity value, Normalized absolute severity value, Normalized absolute

severity control loop, Simple severity zones, Relative severity zones, Severity value and Normalized severity value. In the equations presented below, it is assumed that the absolute Severity value is **as**, Severity value of a situation is **s**, the Severity zone of the situation is **sz**, the Severity value of the threshold is **ts** and that the maximum Severity value possible is **ms**. The current number of instances of an application is assumed to be **ci** and the new number of instances after the adaptation is assumed to be **ni**.

With the exception of the Maximum attribute control loop technique, the adaptation instances which are determined by each technique in an adaptation action, are rounded to the nearest integer. The Maximum attribute control loop technique uses as an exception the ceiling value of the calculated number of adaptation instances.

Table 7 summarizes the design traits of the methods to be discussed in the next subsections. Each method is characterized by its origin (whether it attempts to simulate related commercial offerings) and its dynamic behaviour. If a method uses more than one metric value present in an SLO rule, to establish the suggested new number of instances, a positive indication appears in the third column. Similarly, if it can spawn or deallocate a non-predefined number of instances, a positive indication appears in the third column. Finally, the fourth column provides a measure of the relative number of instances the method is expected to change in a scaling event. Methods exhibiting similar characteristics use input data (the threshold, the metric values) in a different manner, so they are proposed as possible alternatives which can be more effective under different circumstances.

Table 7. Characteristics of adaptation techniques.

Technique	Inspired from commercial offerings	Uses individual values from multiple metrics	Dynamic resource (de)allocation	Aggressive (de)allocation of instances
Simple threshold	✓	✗	✗	Very Low (Simulations), Low (Realistic tests)
Maximum attribute control loop	✓	Partly	✓	Medium – also depends on the thresholds set
Absolute severity value	✗	✓	✓	Very High
Normalized absolute severity	✗	✓	✓	Medium
Normalized absolute severity control loop	✗	✓	✓	Low – also depends on the thresholds set
Simple severity zones	✗	✓	✗	Custom (In tests, was configured as Low)
Relative severity zones	✗	✓	✓	Custom (In tests, was configured as Low)
Severity value	✗	✓	✓	High
Normalized severity value	✗	✓	✓	Low

Techniques which feature higher dynamicity than others can potentially respond quite well to sudden workload changes; however, they might also respond too aggressively when a small workload change occurs, and thus be unstable. Still, techniques which have lower dynamicity, might not be efficient in handling workload spikes, but can be more stable when small adaptations are needed.

8.4.1. Simple threshold

The Simple threshold technique is inspired by the offerings of major cloud providers, and the THRES technique described in [48]. It adds or removes one instance for as

long as the thresholds of a rule are violated. The new number of instances after an adaptation is ni = ci±k (the plus sign is for a scale out rule and the minus sign for a scale-in rule). The value of k was chosen to be 1 in the case of simulations and 4 in the case of the realistic evaluation. Using lower k values, the aim of this technique is to allow fine-grained adaptations (in this case it cannot efficiently handle sudden workload changes) while using higher k values this technique tries to respond quickly to workload fluctuations.

8.4.2. Maximum attribute control loop

The Maximum attribute control loop technique is inspired by the offering of the Kubernetes HPA. The technique adds or removes processing instances, trying to keep a number of monitoring attributes close to their thresholds and choosing the greatest adaptation, i.e., the maximum number of instances which should be added/removed. Its dynamicity renders it is suitable for both small and greater workload changes. The attribute which triggers the greatest adaptation is referred to as the 'maximum_attribute' and its threshold as 'maximum_attribute_threshold'. The new number of instances after a scale-out or a scale-in adaptation appears in Equation 4.

$$ni = ci \cdot \frac{maximum\ attribute}{maximum\ attribute\ threshold}$$

Equation 4. New instances determined for a scale out rule using the Maximum attribute control technique.

8.4.3. Absolute severity value

The Absolute severity value technique uses the absolute Severity value from a situation, which is calculated by assuming that t_i=0 in Equation 2 and Equation 3. In techniques using the absolute Severity value, the values of the thresholds of each

metric are only used to trigger the rule, but do not affect the new number of instances. The new number of instances after an adaptation using this technique is ni = ci(1±**as**) As the maximum possible value of Severity increases linearly with the number of attributes which are involved in a situation, this technique is oriented to handle sudden spikes which are caused by a precise combination of multiple metrics. However, in the case of smaller workload fluctuations it can introduce unnecessarily large reconfigurations. As mentioned above, **as** reflects the absolute Severity value.

8.4.4. Normalized absolute severity

The Normalized absolute severity value technique tries to stabilize the instances of the application using the normalized absolute Severity value – which is obtained by dividing the absolute Severity value with the maximum possible Severity value. This technique allows a reaction which is proportional to the actual metric values (and does not use the threshold values except for its triggering). The new number of instances after an adaptation using this technique is shown in Equation 5.

$$ni = ci\left(1 \pm \frac{as}{ms}\right)$$

Equation 5. New instances determined for a scale out (plus sign) and a scale in (minus sign) rule using the Normalized absolute severity technique.

As an example, if the absolute Severity of a scale in rule was calculated to be 1.2, the maximum Severity for this rule is 2 and the current number of instances is 10, from Equation 5 the new number of instances will be 4, meaning that 6 of the instances will be deactivated.

8.4.5. Normalized absolute severity control loop

The Normalized absolute severity control loop technique tries to stabilize the normalized absolute Severity value around the Severity value of the threshold. To avoid continuous adaptations, an upper threshold and a lower threshold are used, separated by a customizable margin inside which no adaptation is triggered. This technique can be seen as a generalization of the Maximum Attribute Control Loop technique of Section 8.4.2 to use Severity (also not using only the maximum value). Depending on the thresholds set the technique can be very conservative (and stable) or quite liberal in its recommendations. The new number of instances after an adaptation using this technique is shown in Equation 6:

$$ni = ci \pm ci \cdot \left(\frac{as}{ts} - 1\right)$$

Equation 6. New instances determined for a scale out (plus sign) or a scale in (minus sign) rule using the Normalized absolute severity control loop technique.

where the ***ts*** value corresponds to either the upper or the lower threshold for a scale out and a scale in rule, respectively.

As an example, if the absolute Severity of a scale in rule was calculated to be 0.9, the upper threshold Severity for this rule is 0.6 and the current number of instances is 10, from Equation 6 the new number of instances will be 5, meaning that 5 instances will be deactivated.

8.4.6. Simple severity zones

The Simple severity zones technique uses the concept of Severity zones to find the number of instances which should be added to the infrastructure which is currently used. The flavour of the technique which was tested during the experiments used 3 Severity zones, which resulted in 1, 2 or 3 instances being added or removed from the current infrastructure as appropriate. It can be viewed as a generalization of the

Simple threshold technique of Section 8.4.1 (being more aggressive when k<3, k being the instances modified using Simple Threshold), using Severity. As the number of instances it can spawn or deallocate are constant, this method is ideal for situations in which the platform is stable and the workload is only gradually modified. The new number of instances after an adaptation using this technique is shown in Equation 7:

$$ni = ci \pm sz$$

Equation 7. New instances determined for a scale out (plus sign) or a scale in (minus sign) rule using the Simple severity zones technique.

As an example, if the processing infrastructure currently has 4 instances and the Severity zone of a situation triggered by a scale out rule was found to be 2, from Equation 7 the new number of instances will be 4+2=6.

8.4.7. Relative severity zones

The Relative severity zones technique uses the concept of Severity zones to add (or remove) a percentage of the current number of instances to the processing infrastructure. Hence it is more dynamic (in general) than the Simple severity zones technique described in Section 8.4.6. Higher values of the k constant indicate more pronounced adaptations. It is recommended that $k \cdot \max(sz) \leq 1$, otherwise in extreme scale-in adaptations all available instances will be deactivated (if no other specific handling of this issue occurs). The flavour of the technique which was tested in simulations used 3 Severity zones, and k=0.1. The updated number of instances based on this technique is shown in Equation 8:

$$ni = ci\,(1 \pm k \cdot sz)$$

Equation 8. New instances determined for a scale out (plus sign) or a scale in (minus sign) rule using the Relative severity zones technique.

To illustrate, let it be assumed that a scale out rule has been triggered and the resulting situation is in the second Severity zone while k=0.25. Then, if the current number of instances is 4, two instances will be removed bringing the total number of instances to 2.

8.4.8. Severity value

The Severity value technique uses the value of the Severity which is calculated using the value of each metric threshold as t_i in Equation 2 and Equation 3. It is suitable both for small and large workload changes. In the case though that the thresholds are not tuned to the workload, it can be unstable and introduce a large number of reconfigurations. The updated number of instances based on this technique is shown in Equation 9:

$$ni = ci\,(1 \pm s)$$

Equation 9. New instances determined for a scale out (plus sign) or scale in rule using the Severity value technique.

For example, if a scale in rule has been triggered and its Severity is 0.5 while the current number of instances is 4, two instances will be removed bringing the total number of instances to 2.

8.4.9. Normalized severity value

The Normalized severity value technique uses the value of Severity calculated as in the case of the Severity value technique, divided by the maximum Severity possible to obtain a normalized (and smaller overall) result. This technique can be used to obtain more conservative adaptation results when the thresholds are not known to be tuned to the workloads, and also on workloads of smaller variability. The new number of instances after the application of this technique appears in Equation 10, for scale out and scale in rules respectively.

$$ni = ci\ (1 \pm \frac{s}{ms})$$

Equation 10. New instances determined for a scale out (plus sign) or scale in (minus sign) rule using the Severity value technique.

To illustrate, if the Severity value of a scale out rule was found to be 1.0, the maximum Severity value is 2.0 and the current number of instances is 10, the new number of instances will be 15, meaning that 5 new instances will be added.

8.5.Illustrative scenario

In this section, a walkthrough of the situation detection process is provided, from the monitoring data published by instances comprising the cloud application to the adaptation which is decided. The focus is on the calculation of the Severity value, and how this can be translated to an adaptation using the Simple severity zones technique. Assuming that the elasticity rule illustrated in Listing 13 (repeated here for convenience) is used to process incoming events:

$$\textbf{For } \textbf{fragid} = VideoTranscoder$$
$$\textbf{if } AverageCPU_{cluster} > 70\% \textbf{ and } AverageRAM_{cluster} > 70\%$$
$$\textbf{within Timewindow} = 10\ minutes$$
$$\textbf{and } 30\ minutes \textbf{ have passed from previous adaptations then } Scale\_out$$

Listing 14. Elasticity Rule example

The above rule states that if the average CPU and memory usage on all devices hosting the component 'VideoTranscoder', surpasses 70% in a time window of 10 minutes, a new scale out adaptation action decision should be issued – provided that no previous adaptation event has occurred in the last 30 minutes (cooldown period).

8.5.1. Situation Detection

It is considered that at a certain time point, a new observation is detected, indicating that over the last 10 minutes, the average values for CPU and RAM were 92% and 71% respectively. Moreover, no adaptation event has occurred in the last 30 minutes. As the average CPU and RAM values are trespassing both thresholds which have been set to 70%, the rule will be triggered, and a situation will be detected. To calculate the Severity of the situation, the detected values should be normalized:

$$Normalized(CPU_1) = \frac{abs(92-70)}{abs(100-70)} = \frac{22}{30} \sim 73.3\%$$
$$Normalized(RAM_1) = \frac{abs(71-70)}{abs(100-70)} = \frac{1}{30} \sim 3.3\%$$

In some of the techniques which were presented in Section 8.4, the absolute Severity value is used. Had such a technique been considered, the detected attribute values would not be normalized, and the original metric values would be used instead.
Once the final metric values to be used are known, the Severity of the situation can be calculated:

$$Severity(CPU, RAM) = \sqrt{1 \cdot 0.733^2 + 1 \cdot 0.033^2} = 0.806$$

Listing 15. The calculated Severity of the situation.

8.5.2. Using Severity zones – based techniques

To accurately find the Severity zone for a detected situation, the number of Severity zones be used is needed to be known, as well as the number of attributes which are monitored in each situation. To satisfy the first need, throughout this work it is assumed that three Severity zones will be used. To satisfy the second need, it can be seen from the active rule (Listing 14), that there are two attributes which are monitored (AverageCPUcluster and AverageRAMcluster). Following the random

point generation and sorting, the Severity zones are determined to have the upper bounds indicated in Table 8.

Table 8. Upper bounds for three Severity zones with two attributes.

Severity zone	Calculated upper bound
1	0.651
2	0.921
3	1.414

As already stated, the upper bound of each rule refers to the deviation from the threshold values, which when normalized are equal to zero. The highest upper bound of Severity zone 3 reflects the situation which has monitoring attribute values with the maximum deviation from the thresholds set by the DevOps observed when all monitoring attributes reach their maximum value, and is equal to $\sqrt{n}$ – in this case $\sqrt{2}$. The Severity of the situation calculated in Table 8 is greater than the first upper bound, and as a result the situation is marked as Severity zone 2. This can be visualized in the following illustration, depicting the Severity classification of all points which indicate a detected situation. The reported average CPU consumption is indicated in the vertical axis, while the reported average RAM consumption is indicated in the horizontal axis. Following the upper bound calculation method presented above, the three Severity zones are depicted in Figure 7.

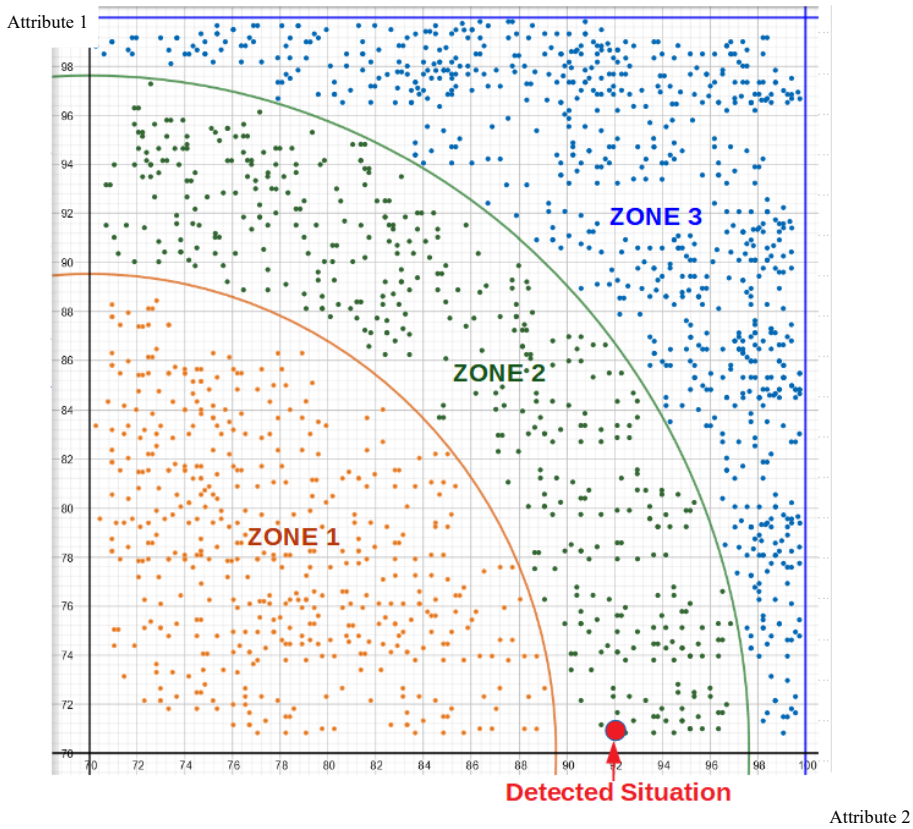


Figure 7. Severity zones in the case of two metrics and three zones.

As expected, the situation initially observed (average CPU = 92%, average RAM = 71%) is located inside the second Severity zone.

In the case of a more complex rule with three attributes and four Severity zones, situations would appear as points in a three-dimensional space, and three spherical shells would be needed to mark the boundaries of the zones. In Figure 8, red points indicate possible situations, and the three shells indicate the limits of each Severity zone. Situations which are 'outside' all shells belong to zone 4, while situations which are 'inside' all shells belong to zone 1.

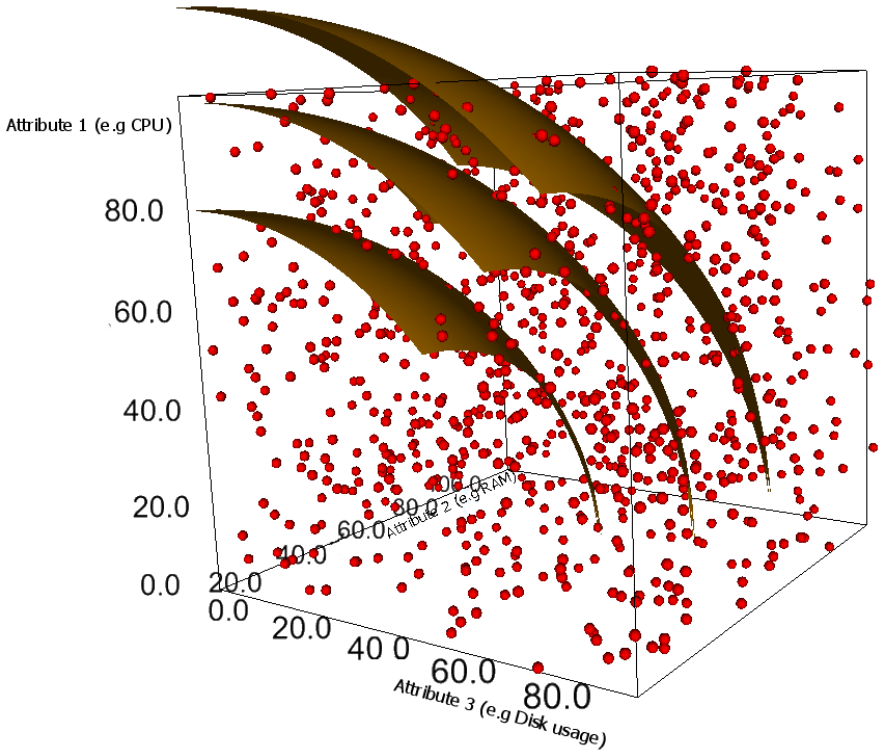


Figure 8. Severity zones in the case of three metrics and four zones.

8.6. Prototype implementations

In the process of validating the usefulness of Severity, three prototype software implementations using it, have emerged. First, a realistic application adaptation manager – offering the functionality of the Situation Detection Mechanism (SDM), using the Simple severity zones technique – was implemented. Then, a simulator was created allowing the evaluation of the performance of Severity-based techniques under 'ideal' monitoring and evaluation conditions. Finally, a lightweight adaptation manager was created in order to evaluate the performance of the Simple severity zones technique against other options. Evaluation results reported in Section 9 are based on the use of the second and third (simulator, and lightweight) adaptation manager implementations.

Below, details are provided for the first implementation – the 'Situation Detection Mechanism' (SDM), which proves the feasibility of the suggested adaptation approach. The source code of this software is publicly available in Gitlab [73]. An

overview of the subcomponents involved in the process of the situation detection and the subsequent platform adaptation is provided in Figure 9 (which was created as part of work [74]):

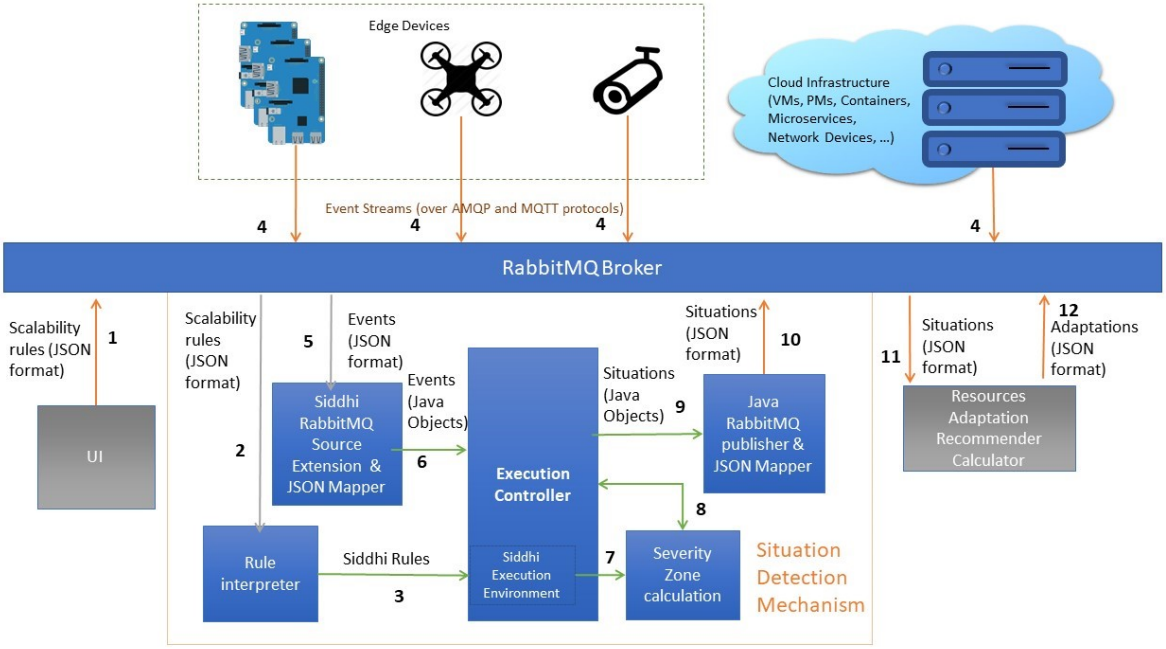


Figure 9. Architecture of prototype situation detection implementation. The numbers in the figure indicate a conceptual flow of information.

The Situation Detection Mechanism is assumed to be part of a platform which manages the adaptation of a cloud application by issuing appropriate scaling directives. Therefore, the subcomponents illustrated in Figure 9 are not part of the cloud application, but instead form the internal architecture of the SDM. The architecture of the Situation Detection Mechanism is structured around the usage of a common message bus, in this case the RabbitMQ[5] broker. This allows not only the decoupling of subcomponents, but also an abstraction layer over the monitoring data which is sent by monitored devices. The mechanism which is used to retrieve monitoring data from the application is agnostic to the Situation Detection Mechanism. The SDM can handle the monitoring of a processing infrastructure

[5] https://www.rabbitmq.com/

composed of any kind of processing machines (VMs, Physical Machines PMs, Containers, Network and edge devices), and receiving any number of processing attributes values, with the proper configuration by the DevOps. Information on the situations which are detected by the SDM is sent through the message bus to an external component, the Resources Adaptation Recommender (RARecom), which issues the actual scaling directives.

The input from the DevOps which triggers the monitoring cycle are the elasticity Rules which are created using an appropriate user interface (UI). This user interface allows the DevOps to select the monitoring metric(s) which should be monitored for a particular component of the deployed cloud application, and the threshold(s) which need to be set. Once an elasticity rule is received by the SDM, it subscribes to the RabbitMQ broker which is directly connected with the infrastructure instances. Then, it can start consuming monitoring events which are related to the metrics of the rule. The flow of monitoring events appears over the message bus in the uppermost part of the figure.

The *Rule Interpreter* module undertakes the conversion of elasticity rules to queries understandable by the Siddhi streaming input processor [75]. These queries use monitoring data to detect a new situation. To translate an elasticity rule, the monitoring metrics which are specified in it are first determined. Then the source (Siddhi stream) for these values is determined, and a suitable Siddhi query is programmatically constructed to retrieve the values. In Listing 13 above, an example elasticity rule which can be used to govern the scaling of an application outwards is described. The representation of the generated Siddhi query for this rule can be seen in Listing 16. It is assumed that cpu_perc, mem_perc, fragid and res_inst are monitoring attributes contained in a Siddhi monitoring stream named ‘serverStream’ (fragid is the component id and res_inst is a unique identifier of the monitored resource).

```
define stream scalability_rule_stream (avg_cpu_perc double, avg_mem_perc double, fragid string,
res_inst string);

from  serverStream#window.timeBatch(600 sec)  [fragid == '1cc5Fragment']
select  avg(cpu_perc) as avg_cpu_perc, avg(mem_perc) as avg_mem_perc, fragid, res_inst
        having  avg_cpu_perc >  70  and  avg_mem_perc >  70
insert  into  scalability_rule_stream;
```

Listing 16. Siddhi query created for example elasticity rule.

The check for a possible previous adaptation which could have happened within the cooldown period is implemented by Java code which is external to Siddhi. Using Siddhi, the SDM can detect when the monitoring attribute thresholds set in a rule are surpassed in which case the elasticity rule is violated, and a situation is detected. Once a situation has been detected it is associated with a Severity zone by the *Severity Zone Calculation* module which performs the calculations which are required. Subsequently, the detected situation and information related with its Severity is published to the broker using an appropriate situation event for further processing by the *RARecom* which will consume it. The RARecom can then process these events to determine the number of instances that will be needed for the scaling adaptation (according to the flavour of the Severity zones technique which is desired, or even perform a new and independent assessment of the situation using another technique).

9. Evaluation

9.1. Comparative Assessment of the TOSCA modelling extensions

In order to evaluate the modelling extensions of the suggested approach, a comparison is conducted between the extended TOSCA and one of the most prominent commercial solutions, Terraform. Terraform was chosen as the most representative of the other approaches in terms of features that are offered. Following this, a definition of the example application presented in Section 3.3 is provided using both approaches; Then, the advantages of each are highlighted. It is considered that the application deployment should try to respect the optimization criteria specified in Table 2.

The first question that should be answered concerns the choice of the cloud provider that should be used. To create the Terraform template, it is assumed that the DevOps of the application invests a thorough amount of time balancing the pros and cons of deployment on a particular cloud provider, while also factoring in the requirements of a particular fragment in order to choose the VMs that have the lowest price for a satisfying deployment. This process is difficult, error-prone, and time-consuming. On the other hand, the approach proposed in this work depends on an initial investment of time to create a component able to solve a constraint programming problem. Subsequently, the optimizer component will be able to automatically evaluate the available cloud offerings and provide the most appropriate processing location for each fragment instance.

The creation of the topology template of the fog application will now be considered, firstly using Terraform. A major disadvantage of Terraform – and the majority of the approaches listed in Table 9 – is that the mixture of edge and cloud devices should be known beforehand and be static in order to accurately describe the topology. However, the assumption of a static topology is very difficult to make, as it requires considerable expertise on the offerings of cloud providers, while more importantly the

available edge resources are opportunistic. On the other hand, if the topology is not static (and a tool similar to Kubernetes is used to abstract this), then there is a risk the instances will be deployed on a possibly suboptimal location (in the case that the cluster consists of both edge devices and cloud VMs, a component may be assigned to a VM when an edge device could support it). Furthermore, in the case that two processing clusters are used, one for edge devices and one for cloud VMs, it is possible that the topology will not be deployable at all (for example, if a component is set to be deployed on the edge cluster but there are insufficient resources in it).

For the purposes of this comparison, an assumption can be made (without loss of generality) that aside from the public cloud resources, the application can also use Raspberry Pi devices to host application fragments.

Based on these requirements, a simplified topology template (omitting most of the network-related details), which can be created using Terraform, appears in Appendix B. Listing 17 contains excerpts from this template, which will aid the comparison with the suggested approach.

```
provider “aws” {
  profile = “default”
  region = “us-east-1”
}
# Network configuration …
resource “aws_instance” “FaceDetector” {
  ami = “ami-2757f631”
  instance_type = “t2.micro”
  key_name = “${aws_key_pair.auth.id}”
  vpc_security_group_ids = [“${aws_security_group.default.id}”]
  subnet_id = “${aws_subnet.default.id}”
  depends_on = [aws_instance.VideoTranscoder]
}
# Other cloud components…
resource “aws_instance” “MultimediaManager” {
  ami = “ami-2757f611”
  instance_type = “t3a.medium “
  key_name = “${aws_key_pair.auth.id}”
  vpc_security_group_ids = [“${aws_security_group.default.id}”]
```

```
    subnet_id = "${aws_subnet.default.id}"
    depends_on =
[aws_instance.FaceDetector,aws_instance.VideoTranscoder,docker_container.percussion_det
ector]
  }
  # Configure the Docker providers
  provider "docker" {
    host = "tcp://192.168.1.2:2375/"
  }
  provider "docker" {
    alias = "worker_2"
    host = "tcp://192.168.1.3:2375/"
  }
    provider "docker" {
    alias = "worker_3"
    host = "tcp://192.168.1.4:2375/"
  }
  # Create a container
  resource "docker_container" "video_streamer" {
    image = docker_image.vs_image.latest
    name = "vs_cont"
  }
  resource "docker_container" "audio_captor" {
    provider = docker.worker_2
    image = docker_image.ac_image.latest
    name = "ac_cont"
  }
  resource "docker_container" "percussion_detector" {
    provider = docker.worker_3
    image = docker_image.pd_image.latest
    name = "pd_cont"
    depends_on = [docker_container.audio_captor]
  }
  resource "docker_image" "vs_image" {
    name = "video_streamer:latest"
  }
  resource "docker_image" "ac_image" {
    provider = docker.worker_2
    name = "audio_captor:latest"
  }
  resource "docker_image" "pd_image" {
    provider = docker.worker_3
    name = "percussion_detector:latest"
  }
```

Listing 17. Sample deployment using a Terraform template.

Similar to what is proposed in this work, this template can be used for repeated deployments of the application relieving the DevOps from the need to manually provision new processing nodes and instantiate software on them. Additionally, Terraform (and other similar approaches) provides a mature, industry-backed, domain-specific language—unlike the suggestion of this work, which is based on a well-recognized standard but is a research effort. Furthermore, Terraform allows the use of variables, which is not exploited in the approach proposed in this work. As a result, components that are used in one use-case can also be used in a similar but different setting by changing only a few values (for example by changing a variable holding the deployment region for a resource or changing a variable holding the Amazon Machine Image (AMI) that will be used by some resources).

However, the explicit nature of Terraform templates also means that they are not easily adaptable. It is also difficult to define relationships between components, which is a native characteristic of TOSCA. As a result, the detailed Terraform templates need to be cautiously inspected by a DevOps to reveal any possible relationships between components (if the DevOps was not involved in creating these templates). In Terraform, expressing software architecture paradigms in a provider-independent manner is a rather difficult task, as the only relevant tool that can be used is the "depends_on" statement. In contrast, building on the ability of TOSCA to create relationships and capabilities, the approach suggested in this work renders a template that is much simpler and easier to understand (see Appendix A). This is especially relevant in the case of function-as-a-service applications, as one can readily understand the architecture of an application.

Continuing this evaluation, a scenario is considered during which the topology should be adapted due to increased load and that two more processing nodes are required for the VideoTranscoder fragment. This leads to a need for a template update to depict the two new nodes that should be added to the topology. However, where should these

nodes be physically instantiated? Even if the unrealistic assumption that the DevOps can know the most appropriate cloud site (e.g., in terms of cost and performance) is made, choosing a cloud-based VM may be a suboptimal solution if one or more edge devices could handle the processing of a fragment. Thus, the DevOps should be additionally burdened with the knowledge of all edge devices that are available for processing if any degree of optimization is sought. Clearly, while this approach is inefficient with a small number of devices, it is totally inapplicable when a large number of edge devices are used. The same argument applies to the knowledge of all VM instance types offered by the cloud vendors. Even considering using automated helper services (e.g., a script calculating a DevOps-defined utility value over all nodes, also providing the best edge candidate nodes), the final confirmation of the DevOps will be needed for any reconfiguration of the platform, which is impractical if large-scale applications are considered. Moreover, in the case of small-scale applications, there will always be a "man-in-the-loop", devoting non-negligible amounts of time and effort to implementing topology adaptation actions. For a Devops, the handling of collocation and anti-affinity constraints is tedious in a processing topology with a large number of fragments, but it is almost impossible when the optimization of costs is also sought. The approach suggested in this work however, paves the way for the usage of multi-objective scheduling based on approaches similar to those in [76–78], which can handle multiple conflicting optimization criteria that should be implemented by the optimizer component.

The current state-of-the-art in cloud application deployment is summarized in Table 9, listing the most prominent approaches. The first column contains the names of each approach, while the second column contains the type, which can be a programming framework, a DSL, or an API. The third and fourth columns discuss the availability of an abstract and an instance model view, which allow an overview of the application and a more precise view of the topology, respectively. The TOSCA-based approach which is described in this work is the only one aside from the CAMEL-based

approach providing a type-level model and instance-level model. Unlike CAMEL, however, it also supports the modelling of execution on edge devices in a direct manner. The fifth column discusses the ability of the approach to deploy a topology utilizing multiple clouds. The sixth column indicates whether certain steps have been taken by approaches to support the modelling topologies using both the cloud and edge. In this context, if a documented methodology to handle edge devices as part of the native language, API, or programming facilities (alongside cloud VMs) is natively offered by an approach, it is considered to fully support cloud and edge deployments. On the other hand, approaches that allow cloud deployment and permit deployment on edge devices (although with manual modelling steps or limited optimization opportunities) are considered to offer partial support for cloud and edge deployments. The seventh column indicates whether the approach has the semantic enhancements required to represent cloud-only, edge-only, and hybrid edge–cloud fragments in a unified way. The eighth column indicates the ability to support the definition of optimization criteria. The ninth column indicates whether there is support for the representation of serverless functions. While the support of commercially available function-as-a-service platforms (already offered by some of the existing approaches) could be considered a reasonable further step for this work, its main focus is to allow the automated creation and update of appropriate modelling constructs for custom FaaS applications.

Table 9. Notable cloud application deployment approaches

Name	Type	Abstract Model View	Instance Model View	Multi-Cloud Topology Support	Cloud and Edge Modelling	Semantic Enhancements	Optimization Readiness	FaaS Support	Comments
Cloudify	DSL (TOSCA-Based)	No	Yes	Yes	Partial	No	No	AWS Lambda	-
Alien4Cloud	DSL (TOSCA-Based)	No	Yes	No	Partial	No	No	No	Partial edge deployment support could be implemented using the concept of "bring your own node" (BYON) for hosting applications
OpenTOSCA	DSL (TOSCA-Based	No	Yes	Yes	Partial	No	No	Yes	FaaS support can be implemented as in [8]
CAMEL	DSL	Yes	Yes	Yes	Partial	No	Yes	Yes	Multi-DSL language built for multi-clouds deployment and recently extended for FaaS support
OCCI	API	No	No	No	Partial	No	No	No	Only few providers are actively backed by an OCCI implementation
Provider-specific languages/ tools	DSL	No	Yes	No	Provider-Dependent	Provider-Dependent	No	Provider-Dependent	e.g., OpenStack Heat, Azure Resource Manager, etc.
Terraform	DSL	No	Yes	Yes	Partial	No	No	Yes	Partial edge deployment support could be manually implemented using a Docker provider
Pulumi	Programming-language-based	No	Yes	Yes	Partial	No	No	Yes	Partial edge deployment support could be manually implemented using a Docker provider
This approach	DSL	Yes	Yes	Yes	Full	Yes	Yes	Yes	Modelling of a custom FaaS architecture is possible

9.2.Simulation-based Evaluation of Severity techniques

In order to evaluate the performance of adaptation techniques using Severity, a series of experiments was made, varying both regarding the input of the DevOps (i.e., the elasticity rules), and the characteristics of the incoming workload. This Section focuses on the experiments which were carried out using a prototype workload simulator.

9.2.1. Benchmark & error metric choice

The rule approaches which have been adopted by some of the main commercial Cloud vendors are based on the use of one or more static rules, which should be created by the DevOps [48]. Using a Severity-based approach, the DevOps can opt to setup a single rule for scale out, and a single rule for scaling in (per metric combination). Then, this rule can be used as an input for a multitude of techniques based on the concept of Severity. In this section the techniques which were discussed in Section 8.4 are evaluated through simulations, along with the common approaches which are found in commercial rule-based systems – using either Simple threshold (ST), or the Maximum attribute control loop (MACL). Only one rule is used for scaling up and one rule for scaling down (using 2 attributes in the case of 2-metric workloads, 3 attributes in the case of 3-metric workloads and 4 attributes in the case of 4-metric workloads). Specifically, the threshold pairs which were tested appear in Table 10:

Table 10. The upper and lower thresholds of the elasticity rules which were used in Simulations.

ID	Scale Out Rule Threshold (greater than operator)				Scale In Rule Threshold (less than operator)			
	Attribute 1	Attribute 2	Attribute 3	Attribute 4	Attribute 1	Attribute 2	Attribute 3	Attribute 4
1	70	70	70	70	30	30	30	30
2	65	65	65	65	55	55	55	55
3	80	80	80	80	70	70	70	70
4	80	80	80	80	55	55	55	55
5	90	90	90	90	80	80	80	80
6	90	90	90	90	10	10	10	10

The maximum number of attribute values which were examined simultaneously was 4, although the concept of Severity can handle an arbitrary number of metrics. Also, the upper threshold for both attributes was chosen to be the same for both attributes, for reasons of simplicity. At this point it is assumed that the nature of the workload is only roughly known to the DevOps – which in turn does not allow the fine-tuning of the rules and permits only a simple selection of the thresholds. Attributes 1 to 4 may reflect any metric (e.g., CPU, RAM, Disk usage, Network bandwidth utilization etc.). The starting point for these experiments is the definition of elasticity rules by the DevOps. These rules dictate the scaling in or scaling out of the platform – also referred to as an adaptation action – to accommodate the load which is induced by the cloud application. Then each pair of rules in Table 10 was applied to handle the workloads presented in Figure 10, Figure 11, Figure 12 and Figure 13, reconfiguring the application and collecting metrics on its performance. The time window of the rules was set equal to one-fifth of the VM spawn duration with a minimum duration of 3 seconds (i.e., 3, 3, 6 and 12 seconds for the 0, 15, 30 and 60 second spawn intervals which were tested). A cooldown period of 10 seconds was required between successive adaptations for all rules.

Optimally, it would be more preferable to use the available resources to their maximum capacity, while also serving the traffic appropriately and having maximum stability. However, to attain this ideal goal, it is required to be able to accurately know the current and future demand of the service so, unavoidably some deviation (error) will exist in at least one of the above-mentioned goals. Thus, the thresholds of the rules should be created by a field expert considered able to balance the risk of service unavailability, with the number of resources which are overprovisioned and application stability. To evaluate the proposed techniques based on Severity, various benchmarking metrics have been considered, and are described in Table 11.

Table 11. Description of the benchmarking metrics used for the evaluation of Severity-based techniques.

Benchmarking metric	Description
Availability	The percentage of time for which the workload pattern for a particular metric was not more the processing capacity of the infrastructure. No distinctions are made for the cases that the processing capacity was exceeded by a small or large margin – in both cases the service is considered unavailable for the purpose of these experiments. Moreover, it was considered that the lack of availability of the service at a particular instance of time does not influence the ability of the service to handle the workload correctly as soon as it receives the resources which are required.
Overprovisioning	The product of the extraneous VM instances which were used (compared to the optimal) with the percentage of simulation time for which they were spawned.
Rigidness	The time percentage of a simulation, for which the application was working either above or even below the rule thresholds set. For example, if a rule on a metric states that a scale out action should happen when the value of the metric surpasses 70%, while a scale in action should happen when the value of the metric drops below 30%, the system is considered to be exhibiting 'rigidness' when the value of the metric is greater than 70% or lower than 30%.
Number of scaling adaptations	The total number of scaling adaptations (associated with an addition or removal of a number of VMs) which were performed by the platform. The first deployment is also counted in this number.

Other approaches have also been using similar metrics in their experiments. For example [48] and [52] examined the number of containers and VMs respectively (which can be related to cost) and the response time for different techniques, while [79] examined cost and execution time (which can be converted to a question between cost and availability). In a thorough review of cloud elasticity [80], the authors mention eagerness, sensitivity and plasticity (related to the 'rigidness' metric), quality of service (indirectly related to the 'availability' metric), cost, oscillatory behaviour/thrashing (related to the 'adaptations' metric) and precision (related to the 'overprovisioning' metric) as principal aspects of service elasticity which should be considered in an elastic system.

In these experiments, four workload types were used, each including two, three or four workload patterns, one for each of the metrics. A separate pattern per processing metric was used in all four workloads (the green line reflects values of the first metric, while the

red line reflects values of the second metric, the blue line the values of the third metric and the yellow line the fourth metric). In experiments using two metrics the red and green lines were used, in experiments using three metrics the red, green and blue lines were used, and in experiments with four metrics all four lines were used. Unlike other approaches (e.g., [48]) workload values are normalized against the processing capacity of a single VM, which is assumed to be 100% per processing metric resource. The x-axis of all workload types represents the time in seconds which has elapsed since the start of the experiment.

4-metric workloads

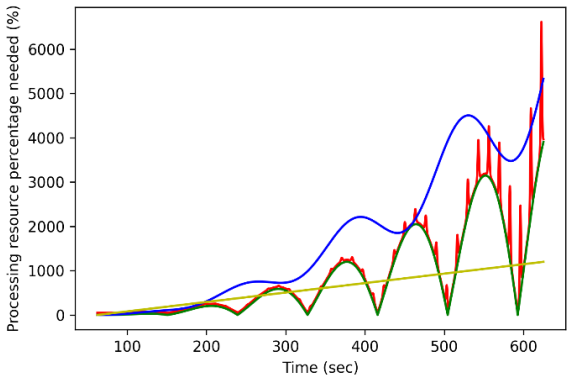


Figure 10. Periodically increasing workload, with spikes.

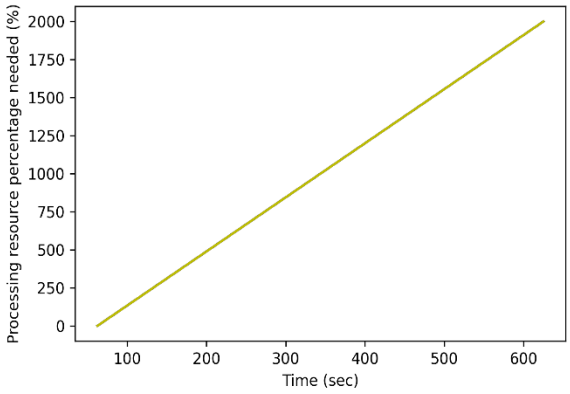


Figure 11. Linearly increasing workload. Due to overlapping of the values, only the yellow line is visible.

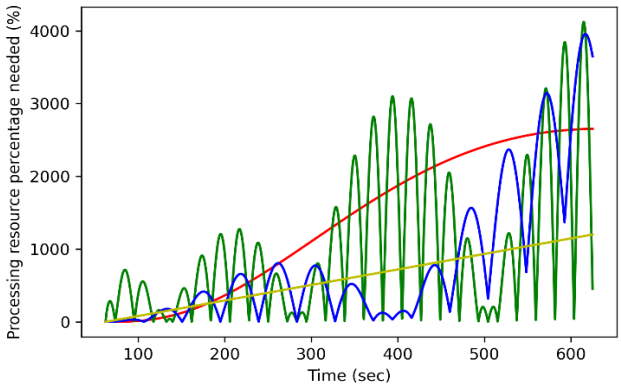


Figure 12. Periodically increasing workload, with fluctuations.

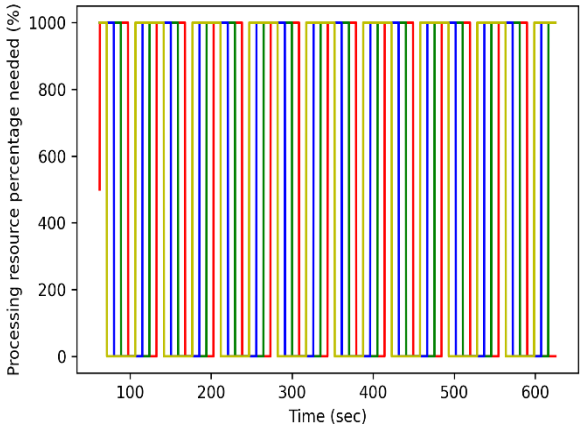


Figure 13. Polarized workload.

It can be readily observed that the four workload types have values which surpass 100% for each of the attributes which are involved. These values are interpreted as a need for more resources which would require additional VMs. Specifically, a value pair of (1%, 200%) in workloads having two attributes means that at least two VMs are needed for the handling of this workload type even though the first attribute only consumes 1% of the resources. In this work, all VMs are considered to have the same processing capacity, i.e. to belong to the same VM type. Moreover, the processing capacity of a VM does not correspond to the processing capacity which is offered by a particular VM flavour of a cloud vendor.

9.2.2. Evaluation results

In this section, the results of applying each proposed technique in different representative workloads are discussed, with respect to the benchmarking metrics discussed in Section 9.2.1. The results were gathered by means of a Python simulator which was developed specifically for the needs of this work. The simulator produced an output file containing the simulated timestamp – a constant time interval was added between successive timestamps, and information on the current real load on the application, the optimal number of instances which could handle the workload, and whether the workload is over the operational thresholds set by the DevOps. Both the code as well as all of the workload traces which were used are available upon request.

In Table 12, the minimum and maximum value for each combination of benchmarking metric, VM spawn delay and 4-metric workload, are presented regarding the Maximum attribute control loop and Severity value techniques. The Simple severity zones and Severity value techniques exhibited the best performance overall in the four criteria which were defined. Data for the Maximum attribute control loop technique was also included as it had better performance than the Simple threshold technique. Therefore, it represents the best of the two techniques which are inspired from commercial offerings and are considered in this work.

Ranges were calculated from the output of the simulations which used the rule pairs which are included in Table 10. The complete dataset with the exact performance of all techniques on every rule pair, every spawn delay and every workload examined is

available as part of this work [81]. All values in Table 12 reflect percentages, except for the number of scaling adaptations which is an integer. Positive values in availability reflect the percentage of the time that the platform could satisfy the load with the existing resources. Positive values of rigidness indicate the percentage of the time that the metric values of the platform were outside the acceptable range set by the greater-than and less-than thresholds. Positive values in overprovisioning indicate a percentage of superfluous VMs which were commissioned by a technique to handle the load, compared to the number of VMs which would exactly match the workload, utilizing their resources up to 100%. As discussed above this is an unrealistic case, as the DevOps needs to set thresholds below 100%, but it is nevertheless the optimal case concerning overprovisioning. Negative values in overprovisioning portray a usage of a smaller number of VMs than the optimal, which by definition impacts availability.

Table 12. Techniques comparison matrix (4 metric workloads).

Technique	Workload	Overprovisioning				Availability				Rigidness				# of Adaptations			
		0-delay	15-delay	30-delay	60-delay	0-delay	15-delay	30-delay	60-delay	0-delay	15-delay	30-delay	60-delay	0-delay	15-delay	30-delay	60-delay
Severity value	Periodically increasing with spikes	-46.2 -9.1	-50.5 15.9	-43.4 31.8	-61.2 9.3	19.0 47.1	25.0 51.8	23.2 52.6	20.0 41.2	3.9 50.3	17.9 75.3	33.3 75.5	24.2 68.2	16 63	16 48	27 46	15 38
	Linearly increasing	10.1 49.9	0.9 42.5	22.5 128.1	19.8 92.8	61.2 100.0	46.6 98.9	47.2 95.9	41.6 86.7	0.2 82.1	13.4 90.5	13.4 92.3	15.3 93.1	23 118	11 65	11 52	11 46
	Periodically increasing with fluctuations	-9.4 28.6	-24.3 45.0	-36.4 57.7	-30.5 14.7	30.3 73.0	28.7 72.3	25.3 66.1	28.4 57.4	0.2 22.7	5.6 50.5	10.4 65.7	15.4 71.2	8 43	8 43	9 42	12 31
	Polarized	-61.9 -57.8	-75.8 -74.0	-79.2 -72.9	-48.2 -23.5	30.6 32.9	12.5 12.5	12.5 12.5	28.1 46.4	29.4 36.3	54.1 57.5	63.5 74.3	33.1 52.9	32 33	32 34	38 40	27 39
Maximum attribute control loop	Periodically increasing with spikes	-73.6 -34.9	-75.4 -42.9	-72.0 -36.3	-65.3 -18.3	9.2 33.0	9.2 20.4	8.3 24.4	9.9 24.6	33.5 65.0	39.9 75.5	34.7 74.2	36.8 70.0	31 47	30 48	27 46	25 43

	Linearly increasing	10.1 50.5	8.1 48.3	8.8 43.5	-16.3 66.0	100.0 100.0	96.6 99.1	61.7 95.9	46.0 87.7	0.2 12.1	22.2 40.0	27.7 89.8	30.1 91.5	23 31	23 31	21 52	21 45
	Periodically increasing with fluctuations	-11.6 31.0	-13.2 26.3	-35.7 26.2	-38.7 40.6	34.7 73.1	32.7 66.5	14.8 63.2	23.1 60.3	3.1 10.5	12.3 24.1	15.3 40.8	24.6 55.0	12 25	13 23	12 32	20 30
	Polarized	-79.7 -70.8	-79.7 -76.5	-79.2 -77.4	-82.2 -44.8	12.5 12.5	12.5 12.5	12.5 12.5	12.5 29.4	34.9 36.7	54.1 57.5	63.5 74.3	30.1 51.3	32 33	32 34	38 40	31 51

For brevity, for the cases of 2 metric and 3 metric workloads we provide a quick summary of the results and two indicative figures. Similar to Table 12, the minimum and maximum values for each of the benchmarking metrics are provided – the top edge of each bar reflects the maximum value and the lower edge the minimum value. Percentage values are used for the measurement of the benchmarking metrics with the exception of the number of scaling adaptations metric which is an integer. Figure 14 illustrates an example case comparing the performance of the SV algorithm with the MACL algorithm, in a 3-metric workload setting. Similarly, Figure 15 illustrates a comparison of the performance of the SSZ algorithm vs the MACL algorithm in an example case using a 2-metric workload.

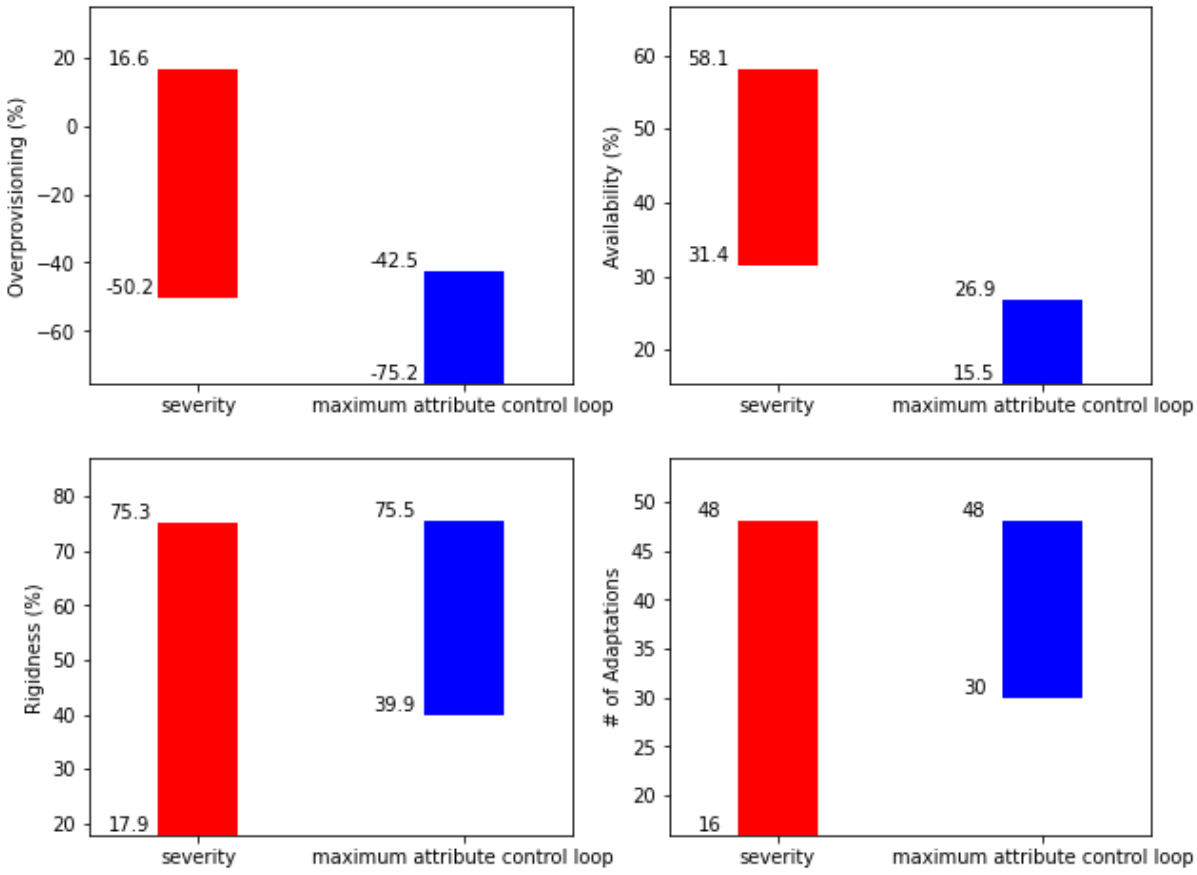


Figure 14. An example case (3 metrics workload, 15 seconds delay and 'Periodically increasing with spikes' workload) in which the severity technique outperforms the maximum attribute control loop technique.

Figure 14 illustrates an extreme case in which the severity technique completely outperforms the maximum attribute control loop technique. It is reminded that the range of observed values for each attribute of each technique is the result of experimenting with multiple scale-in and scale-out threshold pairs. The severity technique both allows a wider range of choices and can also obtain the best values in each of the four measured criteria. The prioritization of these criteria should be performed by the DevOps, who can then choose the most appropriate scaling technique among the suggested techniques. In this particular example, using the severity technique one may choose whether to avoid the underprovisioning of the service to obtain higher availability, lower rigidness and a lower number of adaptations. Such a choice is not available when using the maximum attribute control loop technique.

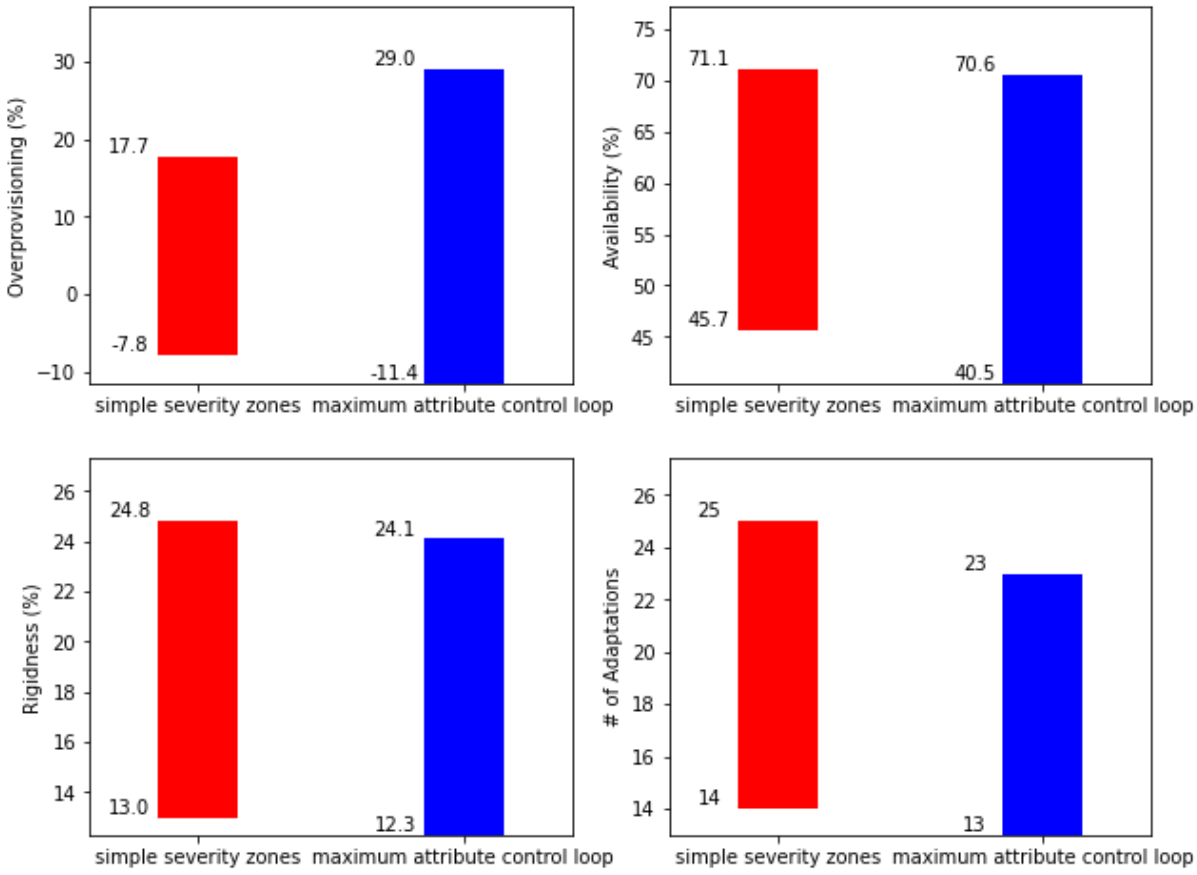


Figure 15. An example case (2 metrics workload, 15 seconds delay and 'Periodically increasing with fluctuations' workload) in which the maximum attribute control loop technique outperforms the simple severity zones technique.

Figure 15 illustrates a case typical of the relative performance of the simple severity zones and maximum attribute control loop techniques in 2 metric workloads. In this case the maximum attribute control loop technique slightly outperforms the simple severity zones technique. However, based on the experimental data the simple severity zones technique can offer better availability and smaller overprovisioning, provided that a much greater number of reconfigurations and increased amount of rigidness can be tolerated by the application.

Table 12 as well as the data collected during the experiments with 2 metric and 3 metric workloads illustrate numerous cases in which Severity-based techniques obtain better results than other techniques which are currently used by cloud vendors. To further

illustrate a comparison between the different techniques, a use-case which emphasizes availability (**av**), low overprovisioning (**op**), low rigidness (**rg**), and a low number of scaling adaptations in turn (**ad**) was assumed. The coefficients pertaining to each of the evaluation criteria appearing in the utility function of the particular use-case were chosen in order to assign more weight to high availability and low overprovisioning than low rigidness and a low number of scaling adaptations. The utility function using the benchmarking metrics, as well as their weights appear in bold in Equation 11. Apart from the weights, the constant terms of the utility function are chosen to normalize the values of the evaluation criteria. Higher utility values indicate a more preferable performance.

$$U = \frac{\mathbf{3} * \boldsymbol{av} + \mathbf{2} * \left(\frac{1 - \boldsymbol{op}}{2}\right) + \mathbf{1} * (1 - \boldsymbol{rg}) + \mathbf{1} * (1 - (\boldsymbol{ad} - 5) * 0.00877)}{7}$$

Equation 11. Example Utility Function.

Table 13, Table 14 and Table 15 illustrate the performance of each technique against each of the four workloads which were used, in the experiments conducted using 2, 3 and 4 metrics respectively. Each cell contains four values which from top to bottom reflect Utility values with a varying delay of 0,15,30 and 60 seconds, respectively. The following abbreviations are used: Absolute severity value (ASV); Maximum attribute control loop (MACL); Normalized absolute severity (NAS); Normalized absolute severity control loop (NASCL); Normalized severity value (NSV), Severity value (SV), Simple severity zones (SSZ), Relative severity zones (RSZ); and Simple threshold (ST). Underlined values indicate an (approximate) tie between algorithms, while bold values indicate the superiority of an algorithm.

Table 13. The performance of each technique using 2-dimensional workloads and a variable spawn delay.

Workload	ASV	MACL	NAS	NASCL	NSV	SV	RSZ	SSZ	ST
Linearly increasing	0.78	0.82	0.78	0.82	0.80	0.82	0.82	0.82	0.82
	0.66	0.77	0.74	0.77	0.76	0.78	0.77	**0.78**	0.77
	0.42	0.74	0.55	0.74	0.69	0.69	0.74	**0.75**	0.74
	0.64	0.66	**0.69**	0.68	0.68	0.67	0.65	0.68	0.68
Periodically increasing with spikes	0.65	0.63	0.64	0.67	0.72	0.72	0.72	0.72	0.63
	0.59	0.59	0.56	0.60	0.62	0.61	0.61	**0.65**	0.60
	0.52	0.61	0.51	0.61	0.56	0.49	0.61	**0.64**	0.61
	0.57	0.61	0.59	0.61	0.60	0.58	0.62	**0.63**	0.60
Periodically increasing with fluctuations	0.70	0.69	0.69	0.69	0.69	**0.71**	0.69	0.67	0.59
	0.63	0.66	0.67	0.65	0.69	0.69	0.67	0.65	0.56
	0.54	0.65	0.67	0.63	0.60	0.67	0.63	0.66	0.56
	0.63	0.59	0.64	0.57	0.62	0.65	0.57	0.65	0.53
Polarized	0.65	0.60	0.58	0.60	0.56	**0.65**	0.54	0.57	0.54
	0.57	0.57	0.57	0.57	0.57	0.57	0.53	0.48	0.53
	0.56	0.55	0.55	0.55	0.54	0.55	0.54	0.53	0.54
	0.60	0.62	0.63	0.62	**0.64**	0.61	0.62	0.61	0.62

Table 14. The performance of each technique using 3-dimensional workloads, and a variable spawn delay.

Workload	ASV	MACL	NAS	NASCL	NSV	SV	RSZ	SSZ	ST
Linearly increasing	0.78	0.82	0.78	0.82	0.80	0.82	0.82	0.82	0.82
	0.66	0.77	0.74	0.77	0.76	0.78	0.77	0.78	0.77
	0.42	0.74	0.55	0.74	0.69	0.69	0.74	**0.75**	0.74
	0.64	0.66	**0.69**	0.68	0.68	0.67	0.65	0.68	0.68
Periodically increasing with spikes	0.63	0.53	0.61	0.55	0.62	**0.66**	0.54	0.53	0.49
	0.55	0.47	0.55	0.49	0.60	**0.61**	0.52	0.54	0.48
	0.50	0.48	0.51	0.47	0.59	0.57	0.50	0.59	0.47
	0.55	0.49	0.59	0.50	**0.60**	0.57	0.54	0.58	0.50
Periodically increasing with fluctuations	0.69	0.68	0.68	0.68	0.67	**0.70**	0.68	0.66	0.57
	0.62	0.65	0.66	0.64	**0.68**	0.67	0.66	0.64	0.54
	0.53	0.63	0.65	0.62	0.59	0.65	0.62	0.64	0.53
	0.62	0.58	0.62	0.56	0.61	0.63	0.56	0.63	0.51
Polarized	0.59	0.53	0.52	0.53	0.49	0.59	0.48	0.51	0.48
	0.50	0.51	0.50	0.51	0.50	0.50	0.47	0.45	0.47
	0.50	0.49	0.48	0.49	0.48	0.48	0.48	**0.53**	0.48
	0.57	0.56	**0.60**	0.56	0.59	0.58	0.55	0.56	0.55

Table 15. The performance of each technique using 4-dimensional workloads, and a variable spawn delay.

Workload name	ASV	MACL	NAS	NASCL	NSV	SV	RSZ	SSZ	ST
Linearly increasing	0.78	0.82	0.78	0.82	0.80	0.82	0.82	0.82	0.82
	0.66	0.77	0.74	0.77	0.76	0.78	0.77	0.78	0.77
	0.42	0.74	0.55	0.74	0.69	0.69	0.74	**0.75**	0.74
	0.64	0.66	**0.69**	0.68	0.68	0.67	0.65	0.68	0.68
Periodically increasing with spikes	0.59	0.50	0.57	0.51	0.58	**0.62**	0.51	0.50	0.45
	0.52	0.45	0.52	0.46	0.58	**0.59**	0.50	0.52	0.45
	0.47	0.46	0.48	0.44	0.56	0.54	0.47	**0.57**	0.45
	0.53	0.47	0.56	0.48	**0.57**	0.54	0.51	0.55	0.47
Periodically increasing with fluctuations	0.69	0.68	0.68	0.68	0.67	**0.70**	0.68	0.66	0.56
	0.62	0.65	0.66	0.64	0.67	0.67	0.66	0.64	0.53
	0.53	0.63	0.65	0.62	0.59	0.65	0.61	0.64	0.53
	0.62	0.57	0.61	0.56	0.60	0.62	0.56	**0.63**	0.50
Polarized	0.57	0.51	0.50	0.51	0.47	0.57	0.45	0.49	0.45
	0.48	0.48	0.48	0.48	0.48	0.48	0.44	0.44	0.44
	0.47	0.46	0.46	0.46	0.45	0.46	0.45	**0.50**	0.45
	0.58	0.52	0.55	0.53	**0.60**	0.58	0.51	0.56	0.51

From the data which is presented in Table 13, Table 14 and Table 15, it can be observed that Severity-based techniques obtain the best results in the majority of the test-cases, whether 2,3 or 4 dimensions are used. The number of metrics used does not significantly influence the utility values which are obtained, although when fewer metrics are used the utility values are in general slightly increased. Also, the increase in the delay to spawn or deallocate instances in general results in lower utility function values for the same technique.

Further, it is apparent that some techniques are more suitable for particular cloud application settings, while others are consistently outperformed. The Simple threshold technique does not perform well under any workload, indicating that either a different number of instances should be added/removed in challenging workloads, or it should not be used. The Maximum attribute control loop technique is a better contender, but it is not so effective when the spawn delay is increased to 60 seconds. This can be attributed to its control-loop character which tries to stabilize the values of the metrics around a desired threshold. However as most workloads change rapidly and its decisions are enforced with a delay, the stabilization is obsolete – resulting in either overprovisioning, or underprovisioning and loss of availability. This characteristic is shared with the Normalized absolute severity control loop method. On the contrary, the Simple severity

zones technique consistently attains the best results when the spawn delay is 30 and 60 seconds and is excellent in handling linear-like workloads. Moreover, the Severity value technique attains excellent results when the delay is zero (and in 10 of the 12 cases when the delay is 15), has consistently the best performance in the 'periodically increasing with fluctuations' workload, and is the technique which attains the best value more often.
Table 16 contains the average improvement of the utility function from the usage of Severity-based techniques.

Table 16. Improvement of utility function values from Severity-based techniques.

	2-metric workloads	**3-metric workloads**	**4-metric workloads**
Maximum improvement over best commercial technique evaluated	4.09%	7.86%	8.54%
Best single-technique improvement over best single commercial technique evaluated	1.26%	5.88%	6.53%
Best Severity-based technique	Simple severity zones	Severity value	Severity value

To determine the values of Table 16 it is needed to determine the best Severity-based and commercially inspired techniques. The sole criterion which is used to find the best technique is the higher utility function value it attains using the most favourable thresholds (for it).
For the first row of Table 16 the improvement of the best Severity-based adaptation technique against the best commercially inspired adaptation technique is calculated per workload and VM spawn delay (16 combinations). The average of this improvement is used to determine the values of this row. To fill the values of the second row, the Severity-based technique which has the highest average utility value across all workloads and VM spawn delays (for the particular workloads) is determined. Then the average improvement when using the best Severity-based technique against the best of the commercially inspired techniques is calculated. The third row contains the best of the Severity-based techniques, in terms of the highest average utility value, calculated over the best choice of thresholds for each technique in each experiment setting (i.e., workload and spawn delay).

It is important to note that in all cases the highest values of the utility function are produced using Severity-based techniques. In the three experiment sets (with 2, 3 and 4 metrics), a total of 48 combinations of workloads and VM spawn delays were evaluated. The Maximum attribute control loop technique was only thrice able to equal this exact maximum value – which besides was attained by the Normalized absolute severity control loop technique. In Table 13, Table 14 and Table 15 above this appears to happen more often due to rounding.

From the evaluation of the scaling methods based on simulations, the following directions on the use of the approach can be established:

- Using data from additional metrics (even partially, as the Maximum Attribute Control Loop algorithm does) in general leads to better estimations. Therefore, the research on algorithms which use data from multiple metrics – all related to the need for scaling – is encouraged.
- Some "non-functional" criteria when choosing a particular algorithm are its adaptability, its extensibility and its explainability. Severity as a concept lends itself to many extensions; in addition, the algorithms which are based on it are both adaptable and explainable.
- Algorithms which have a very high or very low spawn/deallocation behaviour, are not recommended for rapidly changing workloads, when threshold-based rules are to be used.
- By consulting the full results of the simulations, it can be observed that even for single evaluation metrics (e.g., Overprovisioning) Severity-based techniques attain the best (in this case the lowest) value in the greater majority of test cases, independent of the number of metrics used, or the workload or the spawn delay. Moreover, in many of the cases that commercial-based scaling techniques (Simple threshold, Maximum attribute control loop) attain the best value, this value is also produced by a Severity-based technique.

9.3.Cloud Adaptation Evaluation

9.3.1. Introduction

In order to acquire a realistic view of the capabilities which are offered using Severity, a series of experiments was also made on real cloud infrastructure, using a real FaaS platform to complement the software-based simulations,. Specifically, the Digital Ocean public cloud provider (Frankfurt region) and Okeanos was used to create a 5-node Kubernetes cluster, hosting one of the most popular Faas distribution, OpenFaas. [6]OpenFaas was chosen as it offered a modular architecture which was relatively easy to modify, and also allowed custom scaling policies to be enforced.

Concerning the structure of the cluster, three of the five nodes were 'small' VMs, utilizing 1 shared vCPU and 1 GB of RAM, one 'big' VM used 4 shared vCPUs and 8GB of RAM, and the Okeanos VM used 1 vCPU and 4GB of RAM (shared with other applications). The latter two machines were used to deploy all of the components which were required for the operation of the Kubernetes cluster (itself created using K3s[7]) as they had more processing capacity which however was not identical. Moreover, they hosted all components which were required for the operation of OpenFaas, except for application containers which were installed on the small droplets.

In order for the experiments to be run, some configuration changes were needed. Firstly, the AlertManager component of Openfaas was scaled down to 0 replicas, as it interfered with custom scaling. Then, the metrics-collector component was reconfigured to collect metrics every 15 seconds. In addition to the components which were used by OpenFaas a custom Docker registry was created, to include the definitions of any functions which were needed during experimentation. Moreover, Linkerd[8] was installed to handle the enforcement and allow inspection of the parallelization of function calls.

During the course of experiments, a number of 'instances' of a Python function were created to support a processing load. These instances were Kubernetes pods, which included a linkerd agent container and the main processing container. The main processing container was constrained to use only 20% of the CPU and and 120Mib of

[6] https://openfaas.com/
[7] https://k3s.io/
[8] https://linkerd.io/

RAM. Therefore, each worker could host approximately up to 15 instances (3workers*100%(capacity per worker)/20%(capacity per instance)) working at full capacity (considering the processing requirements of system services and the Linkerd agents negligible). In practice, the workloads could be supported with even 8 workers. Using more workers – tests were made purposedly using 30 workers – resulted in the processing topology becoming more unstable, however it was possible to service all workloads, with a very low number of failures and a very low response time.

However, before proceeding to present and discuss the results of the experimentation it should be noted that while in a simulation setting the appropriate modelling of parallelism can be guaranteed, in a realistic processing setting this is not necessarily the case. Thus, the first challenge is to perform a 'sanity check', verifying that parallel processing is meaningful, and that a benefit can be observed from using parallel instances (at all). To this end, an installation of OpenFaas was performed on Kubernetes, on an independent dedicated processing node, with 6 CPUs and 12 processing threads (VCPUs) to determine parallelism benefits of the architecture – ruling out any performance inconsistencies owing to the use of shared vCPUs. In this evaluation, a very simple processing function which created and returned a Python list was used.

Table 17. Response time per number of instances

Instances	Average response time (sec)	Instances	Average response time (sec)
1	0.133	**2**	0.0735
1	0.1546	**2**	0.075
1	0.1555	**3**	0.0594
1	0.1515	**3**	0.0603
1	0.1498	**3**	0.0598
1	0.1475	**4**	0.0481
1	0.1375	**4**	0.0483
1	0.1375	**4**	0.0419
1	0.1371	**4**	0.043
1	0.1334	**4**	0.0435
1	0.1352	**4**	0.0429
1	0.1365	**4**	0.0414
1	0.1401	**4**	0.0423
1	0.1331	**8**	0.0226
2	0.0737	**8**	0.0231
2	0.0833	**8**	0.0232
2	0.0831	**8**	0.0227
2	0.0831	**8**	0.0232
2	0.0739	**8**	0.0234
2	0.0748	**8**	0.0206

A linear regression model was fitted on the data of Table 17, and the constant processing time was estimated to be 9.9 msec, while the proportional processing time was estimated to be 132.4 msec. The r-squared value of this model was 98.4%, which combined with the inspection of Figure 16 indicates a good fit of the model to the data.

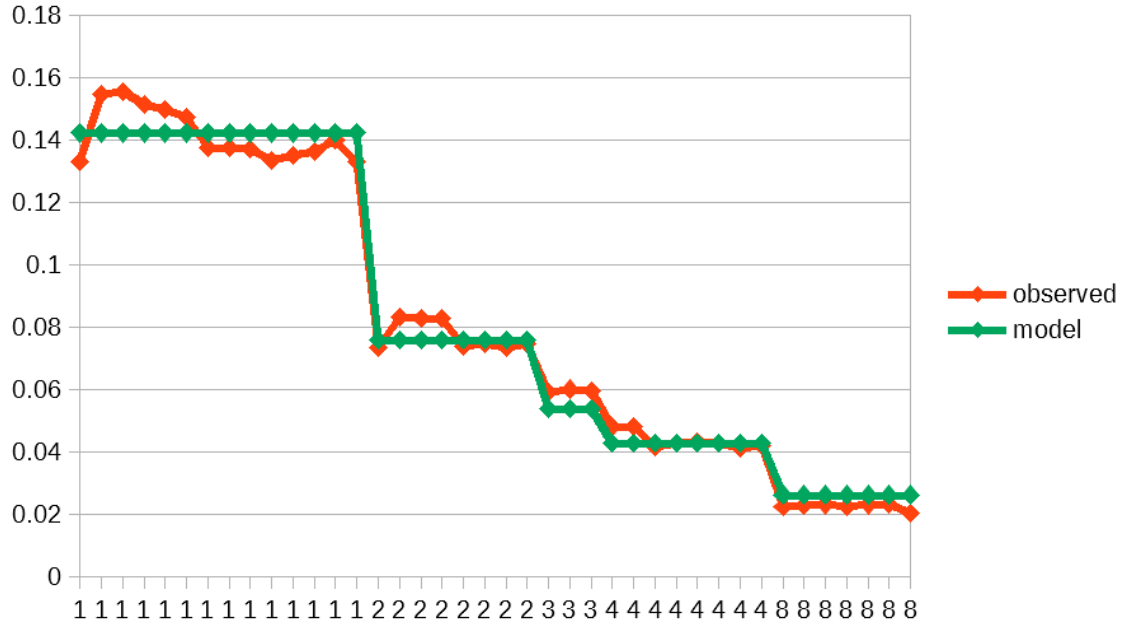


Figure 16. Performance improvements owing to function parallelism

The observed behaviour is consistent with Amdahl's law [82] which determines the limit to which any parallel program or process can benefit from horizontal scaling.

9.3.2. Experiment Design

The basic building block of the experiments, and the main processing consumer inside the function container, was a Python function – named *primecalc3* – which calculated prime numbers up to a given limit (or 100 if no limit was given). Artificial memory usage (unrelated to finding prime numbers) was also incorporated to simulate an intense CPU and RAM workload. The code of the function appears in Listing 18.

```
def handle(req):
 if (req is None or req==""):
  req = 100
 memory_list=[]
 count=0
 primes=[]
 for number in range(1,int(req)+1):
  for i in range(2,number):
   memory_list.append(i)
   memory_list.append(i)
   memory_list.append(i)
   memory_list.append(i)
   memory_list.append(i)
   memory_list.append(i)
   memory_list.append(i)
   if number%i==0:
    count+=1
    break
   i+=1
  if count==0:
   primes.append(number)
  count=0
  number+=1
 #print(primes)
 return str(primes)
```

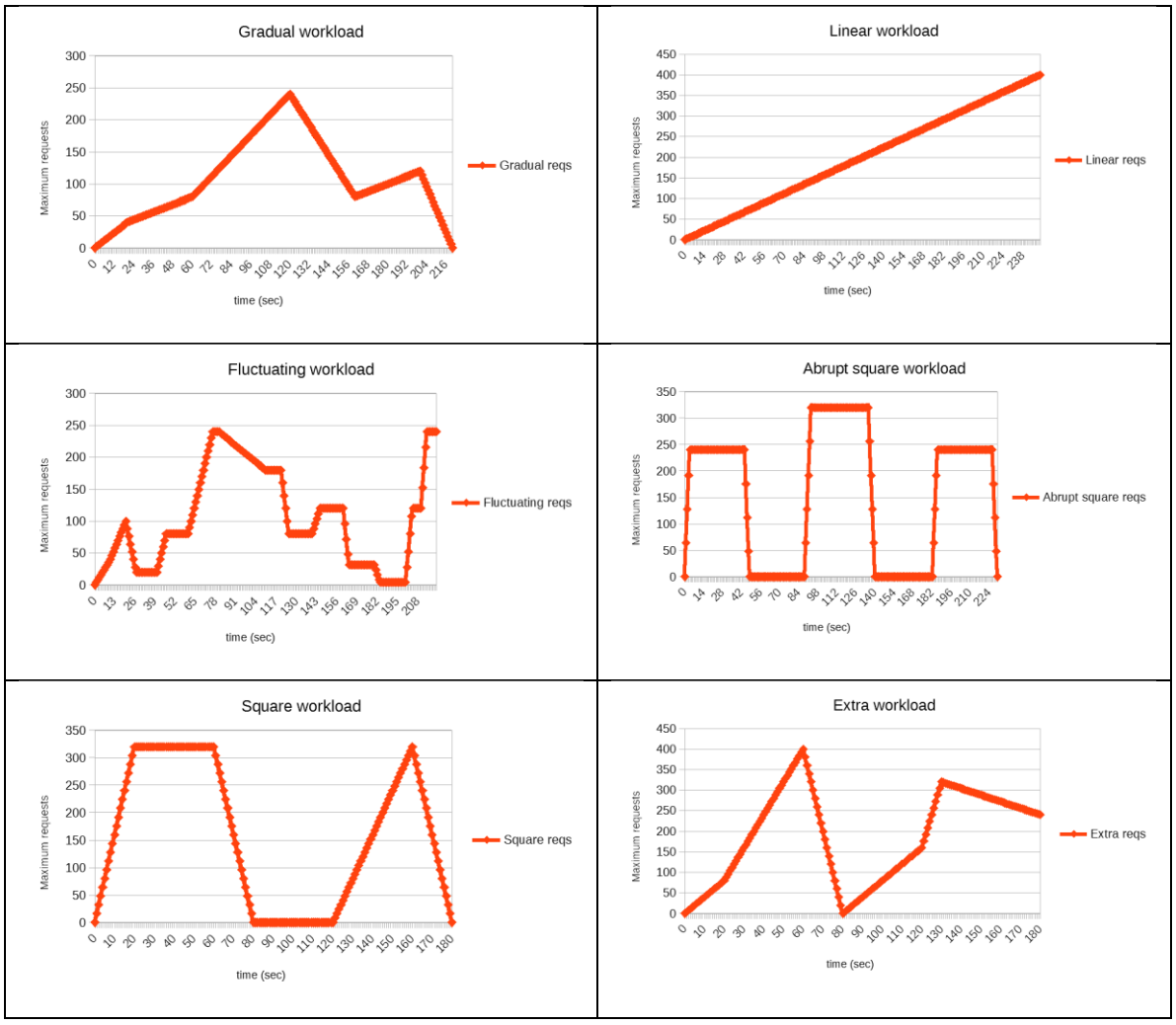
Listing 18. Openfaas handler code for the primecalc3 function

Each function could be served by one or more instances (pods), although when each experiment started there was only one instance. A programmable number of identical calls were performed based on appropriate configuration of the Locust load generator. Then, each of the four examined scaling methods – Simple Threshold, Triple Threshold, Simple Severity Zones and Maximum Attribute Control Loop was used to support a number of requests under a particular cooldown constraint – during which it was not possible to issue further adaptations.

The generator was configured through the use of appropriate Python files, detailing a number of users which should be simulated, and the maximum throughput of each user. When adequate processing capacity was available, the workload generator could issue the maximum permissible amount of requests, while when less processing capacity was available the number of requests which were made dropped.

Six processing workloads were used to evaluate the performance of the examined scaling methods. The (theoretically) maximum permissible amount of requests for each of these appears in Table 18.

Table 18. Processing workloads used to evaluate scaling methods



The scaling threshold is common to all scaling methods – to scale up it was required to surpass (on average) 10 requests per second, 30Mb for RAM consumption and 10% (out of 20% maximum) of CPU usage. To scale down, it was necessary to observe (on average) less than 1 request per second, less than 60Mb of RAM usage and less than 5% of CPU usage. The required monitoring metrics were retrieved i) by exposing the cAdvisor native monitoring component of Kubernetes, tuned to provide monitoring

metrics every 15 seconds (the highest possible frequency) and ii) using the OpenFaas gateway API. Apart from trespassing the thresholds, to perform a scaling action it was also required that a cooldown delay had passed from the previous scaling action. The cooldown delays which were used were 15 seconds (in order to take into account updated monitoring data) and 30 seconds.

The Maximum Attribute Scaling Control Loop scaling technique was the same as defined in Section 8.4.2, as was the Simple Severity Zones technique (defined in Section 8.4.6). The Simple Threshold scaling technique was modified – inspired from the default scaling action of Openfaas – to add or remove four processing instances instead of one (which was used in simulations), allowing a more rapid (and expensive) reaction to workload changes.

For each workload, algorithm, cooldown delay and scaling technique, 15 experiments were carried out. The exception to this rule were the experiments conducted using the Gradual workload using a cooldown delay of 15 seconds. In this case, 20 experiments were requested. Experiments were carried out manually, and automatically through the use of scipts. In the case of automatic executions, experiments were started in the infrastructure based on a schedule. The local adaptation controller was running and whenever a relatively large amount of time passed without adaptation (60 seconds) the next experiment started to be logged. Additionally, a reasonable amount of time was allowed between experiments to allow the processing topology to return to 1 processing instance. Therefore the speed at which the topology achieved the required scale-in using a particular adaptation technique was important during experiments.

The processing topology which was used during experiments appears in Figure 17 below.

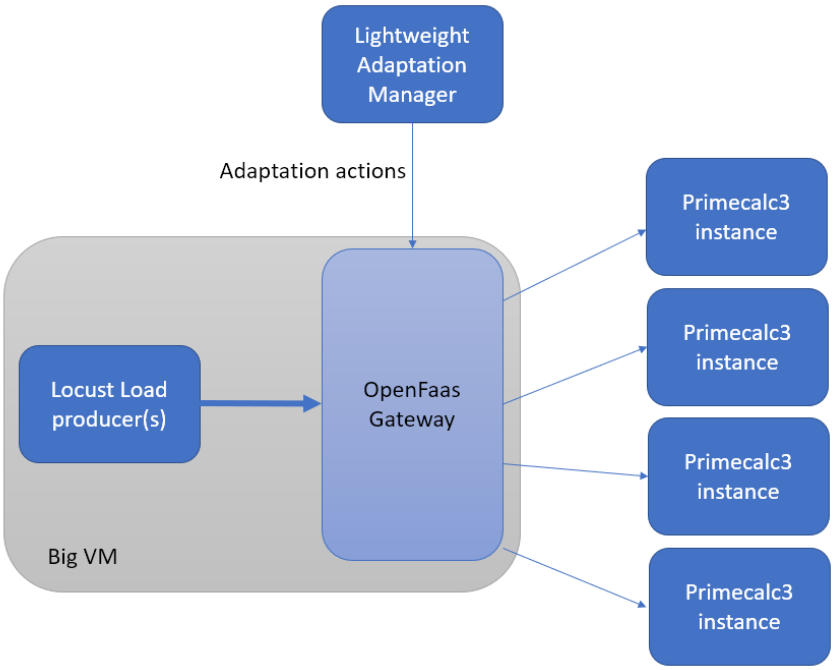


Figure 17. Processing topology used in the evaluation using a realistic topology

The adaptation evaluation metrics which were obtained are closely related to the metrics which were obtained from simulation data. The successful number of invocations was calculated, as well as the unsuccessful number of invocations, the average response time, the number of reconfigurations (adaptation decisions), the average number of function instances (hosts) which were used, the number of 'extra' host-seconds (the number of seconds in which more than one function instance were used, multiplied by the number of function instances which were used at that time) and the total number of host-seconds.
The utility function which was created, appears in Equation 12:

$$U = \mathbf{2} * \boldsymbol{sr} + \mathbf{1} * (\mathbf{1} - \boldsymbol{fr}) + \mathbf{1} * \boldsymbol{rt} + \mathbf{1} * (1 - \boldsymbol{ehs}) + \mathbf{1} * (1 - \boldsymbol{ths}) + \mathbf{1} * \boldsymbol{ad}$$

Equation 12. Utility function for the evaluation of adaptation methods on real infrastructure

In Equation 12, **sr** is the normalized number of successful requests, **fr** is the normalized number of unsuccessful requests, **rt** is the normalized average response time per request, **ehs** is the normalized number of extra host-seconds, **ths** is the normalized number of total host-seconds and **ad** is the normalized number of adaptations (application

reconfigurations through horizontal elasticity). Similar to the utility function defined in the case of simulated workload (see Section 9.2.2), the utility function of Equation 12 assigns a weight of 3 to metrics related to the availability of the application (2 to successful requests plus one to unsuccessful requests), a weight of 1 to the rigidness-like metric of response-time, a weight of 2 to metrics related to the overprovisioning of the application (1 to total host-seconds and 1 to extra host-seconds) and a weight of 1 to the number of adaptations.

The indicators of performance of an adaptation method were defined to be the number of successful requests (desired to be high), the number of unsuccesful requests (desired to be low) and the response time (desired to be low), while the indicators of resource consumption and adaptation flexibility were extra host-seconds, total host-seconds and the number of adaptations.

In the case of successful requests, normalization was made against the maximum effective number of requests which was calculated by executing each workload with a constant number of 15 available function instances. The values which were obtained for the performance metrics in this way are indicated by the 'BEST' method. Expectedly, reconfigurations and host-seconds were not meaningful to be measured for the 'BEST' method, as no adaptations were taking place.

Normalization was also performed for the average response time metric, and the number of unsuccessful requests, although in the last case the best case was simplified to 0 failed requests in calculations (as the number of failed requests was less than 2 in all workloads). In these cases the metric was normalized as seen in Equation 13.

$$normalized\_metric \ = \ 1 - \frac{(metric\_value \ - best\_value)}{(worst\_value - best\_value)}$$

Equation 13. Normalization of utility function metrics

Regarding the resource consumption metrics, normalization was performed against the best case – that is the use of 0 extra host-seconds and 0 total host-seconds, as well as the use of 0 adaptations, also using Equation 13.

At the end of the normalization process, higher values close to 1 indicated better results, while values close to 0 indicated worse results on the particular metric. Since utility function values were based on normalized values, and these normalized values used only

the data available in each data table, it is not meaningful to compare the utility function values across experiments. Moreover, utility function values were calculated also taking into account the performance of the improved Simple Severity Zones method which is discussed in Section 9.3.9.

The results which were obtained on each of the workloads are presented discussed in the following sections.

9.3.3. Gradual workload

The performance of the four evaluated methods appears in Table 18 (using a 15 second cooldown period) and Table 19 (using a 30 second cooldown period):

Table 19. Comparison of adaptation methods handling the Gradual workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
MACL	27,280.20	536.55	394.80	11.91	2,584.53	2,820.69	3.90	2.83
ST	27,403.10	271.55	299.70	9.76	2,371.53	2,641.09	6.80	3.56
SSZ	27,272.40	89.25	449.30	6.15	1,387.13	1,656.46	9.45	3.86
BEST	27,446.13	1.87	163.33					

Table 20. Comparison of adaptation methods handling the Gradual workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	27,312.33	325.07	385.53	7.78	2,127.36	2,441.11	6.00	4.10
MACL	27,027.07	1,582.00	916.00	11.00	2,736.44	3,010.21	3.80	2.47
SSZ	27,303.60	186.20	511.20	4.80	1,220.67	1,541.28	7.20	4.50
BEST	27,446.13	1.87	163.33					

The Gradual workload was one of the easiest workloads for adaptation methods, as can be understood by the performance levels which were high for all methods – although noticeably less for the MACL method.

The Simple Severity Zones technique attains the best Utility value in both experiments, as it offers considerably lower cost while keeping a high number of successful requests, and a low number of unsuccessful requests.

9.3.4. Fluctuating workload

The performance of the four evaluated methods appears in Table 21 (using a 15 second cooldown period) and Table 22 (using a 30 second cooldown period):

Table 21. Comparison of adaptation methods handling the Fluctuating workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	23,134.20	61.20	147.73	7.44	1,700.96	1,962.90	5.07	4.38
SSZ	22,711.60	74.73	341.33	6.03	1,380.14	1,654.56	8.47	3.53
MACL	22,194.00	382.73	338.87	9.73	2,219.12	2,474.86	4.53	2.40
BEST	23498.53	1.13	58					

Table 22. Comparison of adaptation methods handling the Fluctuating workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	22,951.47	166.87	263.20	6.70	1,608.48	1,890.17	5.73	4.25
MACL	20,895.60	986.27	664.60	10.18	2,345.52	2,602.64	4.27	2.20
SSZ	21,972.87	142.67	530.13	4.65	1,125.39	1,433.67	6.80	3.99
BEST	23498.53	1.13	58					

The Fluctuating workload was another case of a workload in which adaptation techniques delivered a high number of successful requests close to the BEST value. Yet, response time values were much higher than the values attained in the BEST case.

In the Fluctuating workload, the Simple Threshold technique attains the best Utility value in both experiments, due to its exceptional performance. The SSZ technique also performs quite well, yet it does not offer the performance attainted by ST.

Besides, the Simple Severity Zones technique is capable of benefiting from further configuration, to obtain better performance (in this case the assumption of a DevOps who only roughly knows workload details does not hold). Another test was made therefore using the same workload, and a smaller scale-out threshold for the number of invocations per second (5 instead of 10). The adaptation metrics which were captured for the adaptation methods appear in the following Table 23 and Table 24:

Table 23. Comparison of adaptation methods using a low scale-out threshold handling the Fluctuating workload, and a 15-second cooldown delay

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	23,073.87	375.67	223.27	8.72	2,018.40	2,279.42	6.27	3.23
MACL	21,500.93	1,267.87	396.67	11.38	2,624.52	2,878.56	4.27	1.69
SSZ	23,023.47	150.53	206.67	6.23	1,473.18	1,755.09	8.73	3.56
BEST	36424.33	1.07	71.47					

Table 24. Comparison of adaptation methods using a low scale-out threshold handling the Fluctuating workload, and a 30-second cooldown delay

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF value
ST	23,128.67	341.47	229.73	6.10	1,388.25	1,659.19	4.53	4.08
MACL	20,109.93	1,784.33	620.60	13.28	3,167.35	3,425.71	4.00	1.41
SSZ	22,846.53	170.67	283.13	4.57	1,079.70	1,382.41	5.73	4.03
BEST	36424.33	1.07	71.47					

The best performance for all methods was generally observed when the cooldown period was smaller. However, it should be noted that in the case of the Simple Threshold technique, the observed values of the adaptation criteria were improved almost everywhere when the cooldown period was increased. These values, appear to be

marginally more desirable than the values obtained by the Simple Severity Zones technique, yet when the utility function is evaluated only against these two the SSZ technique actually achieves a better score (2.86 vs 2.46).

Allowing a longer cooldown delay reduced the performance criteria for all techniques, yet it allowed for a considerable decrease of the host-seconds. As a result, the UF values obtained by these two methods using a 30-second cooldown delay, would be better if the values could be directly compared.

Therefore, using a lower threshold can enable the Simple Severity Zones method to benefit from its ability to divide the load into broad categories and attain similar if not better utility values than those of the Simple Threshold technique.

On the other hand, using a higher threshold was not beneficial for any adaptation method. When the threshold was changed to 15% cpu usage and 25 invocations per second for a scale out (memory thresholds stayed the same) the data appearing in Table 25 and Table 26 was gathered:

Table 25. Comparison of adaptation methods using a high scale-out threshold handling the Fluctuating workload, and a 15-second cooldown delay

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations
ST	22,954.47	162.20	246.47	5.96	1,266.72	1,521.06	4.27
MACL	21,780.67	361.47	585.00	6.04	1,295.71	1,553.17	5.00
SSZ	22,171.87	185.53	486.93	4.78	1,011.09	1,278.58	6.20
BEST	36424.33	1.07	71.47				

Table 26. Comparison of adaptation methods using a high scale-out threshold handling the Fluctuating workload, and a 30-second cooldown delay

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations
ST	22,940.60	193.27	251.40	5.33	1,156.60	1,424.06	4.00
MACL	20,919.93	294.27	667.40	4.04	842.05	1,119.25	4.47
SSZ	22,259.87	100.40	454.53	4.10	936.18	1,238.06	5.20
BEST	36424.33	1.07	71.47				

9.3.5. Square workload

The performance of the four evaluated methods appears in
Table **27** (using a 15 second cooldown period) and Table 28 (using a 30 second cooldown period):

Table 27. Comparison of adaptation methods handling the Square workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
SSZ	20,597.07	876.67	1,042.13	5.24	966.79	1,194.29	9.20	3.58
ST	19,748.87	1,253.73	1,154.27	6.72	1,325.13	1,555.15	8.80	3.03
MACL	14,574.47	2,850.67	1,775.33	10.88	2,042.78	2,250.78	6.73	1.31
BEST	29,132.80	0.20	52.20					

Table 28. Comparison of adaptation methods handling the Square workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	21,647.53	813.47	1,035.73	5.28	1,117.44	1,377.39	5.87	2.42
SSZ	20,540.53	720.13	1,237.40	4.12	811.36	1,070.57	6.07	2.64
MACL	18,518.87	1,471.47	1,077.27	6.88	1,324.06	1,549.01	5.00	1.60
BEST	29,132.80	0.20	52.20					

It can be observed that the Simple Severity Zones technique attains better utility function values than other techniques, both in the case of 15 second cooldown delay and 30 second cooldown delay. In the case of the 15-second cooldown delay, it is remarkable that it achieves both better performance and less resource consumption than alternative methods.

9.3.6. Extra workload

The performance of the four evaluated methods appears in Table 29 (using a 15 second cooldown period) and Table 30 (using a 30 second cooldown period):

Table 29. Comparison of adaptation methods handling the Extra workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	30,139.13	445.33	524.67	9.24	2,105.56	2,361.29	7.33	3.70
SSZ	29,970.53	456.80	475.73	6.74	1,489.60	1,749.21	9.20	3.99
MACL	25,521.93	2,030.87	791.07	14.81	2,825.50	3,030.43	4.87	2.03
BEST	34,462.27	0.20	76.80					

Table 30. Comparison of adaptation methods handling the Extra workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	30,620.40	258.73	536.87	7.22	1,848.01	2,144.85	6.00	3.12
SSZ	29,261.80	258.00	603.27	5.05	1,200.28	1,496.52	6.67	3.30
MACL	31,651.67	433.27	340.87	13.16	2,672.33	2,892.38	3.13	2.94
BEST	34,462.27	0.20	76.80					

As can be observed from the experimental data which was collected, the performance of all methods was quite good, and much closer to the optimal values than what was observed in other workloads. The Simple Severity Zones algorithm attained the highest Utility function scores, sacrificing optimal performance to decrease the number of host-seconds used by a large margin.

9.3.7. Linear workload

The performance of the four evaluated methods appears in Table 31 (using a 15 second cooldown period) and Table 32 (using a 30 second cooldown period):

Table 31. Comparison of adaptation methods handling the Linear workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	48,992.87	257.00	247.20	10.00	2,989.58	3,321.82	8.00	3.49
SSZ	48,526.67	105.47	323.93	7.36	2,172.04	2,513.37	15.00	3.46
MACL	47,785.53	688.40	267.13	12.07	3,101.20	3,381.72	3.87	2.92
BEST	50,166.13	1.33	88.97					

Table 32. Comparison of adaptation methods handling the Linear workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	49,182.80	436.73	234.73	8.69	3,060.06	3,457.60	7.87	2.91
SSZ	49,886.07	181.67	193.20	5.53	1,723.90	2,104.41	9.80	4.11
MACL	48,972.00	860.60	182.07	12.15	3,367.44	3,669.66	3.00	3.05
BEST	50,166.13	1.33	88.97					

In the case of the Linear workload, while the Simple Threshold was determined to offer a slightly more desirable response when using a 15-second delay, the best performance overall was observed when using a 30-second delay, and was achieved by the Simple Severity Zones technique. Moreover, when pairwise comparing the two techniques, Simple Severity Zones achieves higher utility function scores.

9.3.8. Abrupt Square workload

The performance of the four evaluated methods appears in Table 33 (using a 15 second cooldown period) and Table 34 (using a 30 second cooldown period):

Table 33. Comparison of adaptation methods handling the Abrupt Square workload using a 15-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	22,448.27	1,692.20	1,573.27	5.94	1,395.61	1,676.63	9.73	2.81
MACL	14,297.40	2,971.47	2,544.07	8.55	1,990.99	2,255.41	8.60	1.08
SSZ	19,860.27	1,615.53	2,076.27	4.32	960.86	1,247.96	10.13	2.87
BEST	36424.33	1.07	71.47					

Table 34. Comparison of adaptation methods handling the Abrupt Square workload using a 30-second cooldown period

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations	UF Value
ST	14,700.73	2,533.07	2,614.52	3.00	560.75	840.53	6.00	1.70
MACL	15,354.47	2,698.73	2,359.87	4.40	954.90	1,236.52	6.07	1.03
SSZ	16,171.80	2,315.20	2,366.09	2.62	461.54	746.51	6.07	2.13
BEST	36424.33	1.07	71.47					

The abrupt workload illustrates the worst performance case for all of the examined adaptation methods. Not only was the number of successful requests only slightly over 50% of the possible successful requests, but also the number of unsuccessful requests was high, as was also the response time. Nevertheless, the Simple Severity Zones achieved the best utility function values in both cooldown delays, although by a very small margin in the case of the 15-second delay.

9.3.9. Improving the performance of Simple Severity Zones

In Equation 1, the definition of Severity allows the specification of weights on the criteria which are involved in its calculation. Throughout this work, for simplicity it was assumed that all weights were equal to each other, and equal to 1. However, this choice might not be satisfactory in all cases. In general, the Severity-based approach can be enhanced with machine learning in order to establish the best weights which should be assigned and obtain a better response.

In the case of the experiments described in the previous subsections, better results could be obtained for a Severity-based approach if the invocations per second metric was solely used (equivalent to setting w_i=0 for CPU and RAM and w=1 to invocations per second). Relevant performance data for this flavor of the SSZ method appears in Table 35:

Table 35. Performance of an extreme, simplified yet improved instance of the Simple Severity Zones method on the workloads described in previous sections.

	Successful requests	Unsuccessful requests	Average response time	Average hosts	Extra Host-seconds	Total Host-seconds	Adaptations
Gradual 15	27,432.35	27.45	162.50	7.49	1,842.50	2,126.39	11.25
Gradual 30	27,395.53	21.27	191.67	5.89	1,562.66	1,881.90	7.53
Fluctuating 15	23,272.60	22.33	97.87	6.31	1,471.53	1,748.33	9.20
Fluctuating 30	23,283.67	25.33	114.07	5.44	1,407.56	1,723.96	7.33
Square 15	21,815.67	712.93	875.27	5.45	1,045.21	1,278.99	9.73
Square 30	22,371.20	526.87	968.80	4.35	862.64	1,119.88	6.20
Extra 15	32,083.00	140.80	370.47	7.47	1,711.72	1,975.91	10.73
Extra 30	31,620.33	115.87	401.00	5.86	1,548.00	1,865.26	8.00
Abrupt Square 15	27,626.73	945.80	892.67	5.63	1,357.43	1,649.62	12.20
Abrupt Square 30	20,177.07	1,624.33	1,773.67	3.09	614.92	906.00	6.67
Linear 15	49,965.07	64.07	156.60	8.79	2,789.52	3,147.40	17.07
Linear 30	50,191.00	79.40	104.27	7.18	2,638.99	3,065.76	11.33

To rely always on one metric to derive the scaling response is not recommended however, as the mixture of requests is not always guaranteed to need the same processing capacity. For example, if the number of requests is falling yet their difficulty (i.e cpu consumption) is increasing, a scale out action might be necessary to retain an acceptable

response time. Therefore, cpu usage and possibly other performance metrics should also be used in the formulation of rules.

9.3.10. Remarks on the evaluation using realistic workloads

Although the results of the realistic workloads did show significant differences from the results of the simulations, all of the points of the discussion on simulations (with the exception of the absolute superiority of Severity-based techniques) continue to hold. A major difference between the two evaluation modes are the workloads. In the case of simulations these were synthetic and quite independent from one another (except for the trend which was positive in three of the four workloads). On the contrary, realistic workloads were based on the variation of a single metric (invocations) which indirectly influenced other metrics. A second important difference was the monitoring interval which was quite high at 15 seconds compared to 1 second in simulations. A third difference involves the smaller scale of the experiments, both workload-wise (the processing requirements of the workload, and its duration were smaller) and variability-wise (one rule threshold pair was mainly analyzed). Finally, while monitoring, scaling and resource use were ideally handled in the simulator, in the case of the realistic environment load distribution and processing they were not. Notwithstanding these differences, the Simple Severity Zones technique demonstrated its value in a realistic setting.

The second observation involves the use of multiple monitoring metrics. Comparing the performance of the Maximum Attribute Control Loop, the Simple Threshold and Simple Severity zones, SSZ generally attains better utility function scores. This implies that it is actually more useful to use the values from multiple monitoring metrics to guide adaptation and not only to trigger it. In the case of the SSZ flavor using only the invocations per second metric, better performance was enabled using only a single metric. However, i) it is not always true that a higher number of requests necessitates a larger amount of workers (as requests may have a varying difficulty) and ii) the invocations per second metric was updated at a higher frequency compared to other metrics, therefore allowing for more relevant scaling actions (as all requests had the same computing difficulty).

10. Discussion

10.1. Modelling Discussion

If the approach which was introduced above is followed, a major advantage related to the installation of a custom FaaS application is brought to the DevOps, as it will be possible to generate provider-agnostic and standards-based models of the application, not only making use both of cloud and edge devices, but also still being easy to review. The prototype type-level TOSCA generator developed [21] can handle both Java annotations and JSON input reflecting an application graph. Existing, widely-used approaches such as Terraform can benefit from being used in conjunction with the proposed modelling approach to deploy the concrete topology (e.g, extending the translation approach proposed in [83]) yet when used independently expressivity and/or optimization opportunities are missed.

The new language structures suggested in this work use the existing TOSCA syntax and each one targets a specific enhanced behaviour. The extended TOSCA specification enables the modelling of self-managed FaaS-based applications alongside more traditional VM-based applications. Applications comprised of coordinator-driven and more traditional architectures may coexist or can be completely separated. It is known [84] that Cloud functions as a part of a FaaS deployment enjoy a much lower startup times and offer more fine-grained cost execution options. These advantages render FaaS-based deployments more preferable to conventional VM-based deployments in some use-case scenarios. In other cases, though, the stability and predictability of VMs is preferred over FaaS. The new language structures which are suggested enable (manual) gradual migrations from VM-based deployments to FaaS-based deployments and vice-versa, using the expressiveness of TOSCA.

In addition, the extensions in the TOSCA specification also target the correct modelling of edge devices. Both fog and edge-only deployments are targeted, supporting mobility from the edge to the cloud and inversely where this can be implemented. Naturally, fog

architectures can be combined in modelling with FaaS-based applications to describe FaaS fog deployments.

Also, irrespectively of the topology which is actually deployed, support for placement constraints was introduced, and optimization factors at the level of each software component (fragment) or the whole of the topology. As part of the illustrative scenario, example optimization factors were presented both at topology and at fragment-level, permitting the runtime optimization of the topology as required.

An alternative modelling approach which is often pursued when designing language extensions would be to create a series of new types which target particular applications. While this brings modelling very close to the actual infrastructure and also provides concrete artifacts which can be readily used, it also diminishes its genericity, decreases the understandability of the model and eventually results in heavier models. Accordingly, judicious creation of specialized but generic types very close to the orchestration software is proposed. This work aims to align to the mentality of recent work on the Essential Deployment Metamodel (EDMM) [85] (and TOSCA-Light [86] which introduces an EDMM-compatible TOSCA subset), which aim to simplify modelling and allow the convergence of modelling efforts. The emphasis is on simplicity, while still addressing some important challenges. The exact manner in which this topology will be later managed and revised is outside the scope of this work, however it has been demonstrated above that type-level and instance-level TOSCA can cope with dynamic setups. A suggestion for a hierarchical optimization scheme was made in Section 7.3, yet more advanced optimization techniques can also be used.

Concerning the relationship of the TOSCA approach outlined in this work with existing modelling solutions, in Section 9.1 this work was compared with one of the currently most successful commercial solutions, Terraform. It was shown that Terraform is unable to provide an abstract view of a processing topology, and is therefore unable to enforce cross-cloud optimization policies, without significant manual intervention. Secondly, the exact knowledge of the edge devices which are available, as well as the instance types which are offered by each provider appears to be the only path to optimize cost while retaining performance to an acceptable level for each application fragment. Moreover,

Terraform and other vendor-specific template-based approaches require the meticulous configuration of all fine-grained networking parameters on behalf of the DevOps.

Instead, the approach proposed in this work isolates the specification of application components from their deployment, can handle both small-scale and large-scale applications, can provide for different optimization criteria and is based on the open standard of TOSCA. Moreover, when coupled by a proper TOSCA interpreter and topology reconfiguration tools as those described in [6] it provides a fully automated approach to application deployment and reconfiguration in a mixed edge/cloud infrastructure. Using type-level and instance-level TOSCA, the application description is not bound to any cloud provider – which allows the application to be preserved for long-term deployments.

In the case of larger topologies, it is expected that the type-level TOSCA templates can become very lengthy. Even in this case, since the type and relative location of nodes which are related to a particular fragment are known beforehand (processing, fragment and mapping nodes), it is easy to understand the purpose of TOSCA elements in a custom FaaS application. When more complex relationships are involved, additional software paradigms should be defined to streamline the understanding of the application topology.

As part of this work, a prototype type-level TOSCA generator was provided, able to create type-level TOSCA for an application topology. However, it was shown that other components, such as the Optimizer and the instance-level TOSCA generator, along with a TOSCA Orchestrator are needed, in order to allow the full exploitation of this approach. A definite future research direction involves the development of components which can create the required output, and further extend the capabilities of the TOSCA ecosystem.

Another interesting research direction involves the modelling and the optimization of services – complete topologies of FaaS microservices– as entities in TOSCA, which will be especially useful for organizations handling hundreds of microservices and tens of services. In this case, software architects will be given the opportunity to select the desired topology among a selection of topologies, choosing the tools and deployment methods that are most relevant for each case.

Finally, applications which are defined using the suggested methodology in TOSCA can easily profit from software components able to deploy and scale application topologies

based on monitoring data. Using the suggested approach – and provided all relevant components are implemented – scaling in and out, up and down is very easy to model and users can readily understand, even from a terminal window (template diff), the changes which the platform has undergone.

10.2. Adaptation Discussion

Related to the adaptation of the FaaS application to cope with workload changes, a Severity-based approach was presented in this work to improve the response of rule-based system. Severity can minimize the input necessary for devising elasticity rules and help the DevOps to guide the operation of a cloud application in a more effective manner. Familiar concepts found in traditional rule-based systems, such as aggregations, thresholds and cooldown intervals are still the basic building blocks. Therefore, it also reaps the benefits associated with rule-based adaptivity, such as lower updating overhead, relative genericity with respect to the workload managed and lower computational complexity when compared to other approaches [68]. Moreover, it can automatically use information from a multitude of monitoring attributes which can be provided in each elasticity rule and not only from a limited, hard-coded selection between average CPU, response time and number of requests.

In contrast to traditional rule-based approaches, the choice of the metrics which should be monitored, and the appropriate threshold values, needs to be complemented by the choice of an appropriate scaling technique. Depending on the nature of the workload, the application goals and the time required to spawn new processing instances, the DevOps should choose the technique which is the most appropriate. The data from the simulations and the realistic experiments can help in this decision. Still, if this is not desirable, the Simple Severity Zones technique can be a sufficiently good candidate for many workloads.

The value of the proposed approach was demonstrated using a utility function which prioritizes the correct provisioning and service availability while also favouring application stability. The evaluation metrics generated by the Simple severity zones and the Severity Value techniques as part of the simulations (and – for SSZ – in many cases in the realistic experiments) yield a better result (higher utility function values) overall,

compared to any of the other commercially inspired techniques. Moreover, in simulations, other Severity-based techniques also demonstrate at least equal and in general better results (than the evaluated commercially inspired techniques) in the isolated testcases which are not covered by the two best techniques. These advantages are considered a direct contribution to the elasticity capabilities of any cloud application facing challenging workloads.

The superiority of some techniques over others which is underlined above is attributed to their design, and the choice of the utility function. For example, if a technique sacrifices availability or performance to reduce overprovisioning it will have a reduced utility function score if any of the presented utility functions is used. In turn, the performance of each technique in each of the performance metrics is directly related to its decisions to spawn or deallocate processing instances.

Notwithstanding, the definition of the Severity value can provide an aggregate view of the current situation, for a number of metrics which violate a given elasticity rule. It also decouples the detection of a situation, from the adaptation which will be triggered. This enables the creation of hybrids which can exploit threshold-based rules as a first stage before triggering another adaptation method, e.g., control loops or machine learning as discussed in Section 9.3.9. When the number of n is relatively small (e.g., $n<10$), the calculation of Severity, is very fast. The most important advantage of calculating the Severity value, however, is that it provides a uniform way to measure the importance of a situation, when multiple metrics are involved.

In Section 9.2 and 9.3 several techniques were evaluated, using a minimal input of the threshold value and the direction of the scaling. Note that different options could have been considered for some techniques. For example, a flavour of the Simple Threshold technique adding or removing 3 instances could have been examined, or a flavour of the Relative severity zones technique using $k=0.2$ rather than $k=0.1$. Hence, the tables which are included in the relevant section serve not as an exhaustive comparison, but rather as an indicator of the suitability of each technique for intense workloads.

Overall, the results appearing in Sections 9.2 and 9.3 are dependent on the datasets which were used and on the design of the experiments, however an effort was made to consider indicative cases by using multiple workloads, spawn delays and threshold pairs. Related

to the time units which were assumed to be used in simulations (as the code implementation was property-driven and not time-driven), it is underlined that although seconds were used for the purposes of comprehensibility, the results would have been unchanged if the time unit used in the workloads and the formulation of rules was changed to minutes, hours, or by any other proportionate factor. Consequently, even if a workload was less intense but followed the same pattern the same results would still be observed if the windows of the rules were also changed proportionately.

Experiments using a realistic platform suggest three main findings: Firstly, Severity is useful in many cases off-the shelf and this is manifested also in quite challenging workloads, while it is also tunable when better performance is needed. It should be admitted however that this needs time and effort from the part of the domain expert, and therefore could benefit from machine learning. Secondly, simplicity is preferable when the nature of the workload changes dramatically and frequently (with respect to the monitoring interval). Thirdly, allowing a platform to operate on monitoring data faster (with a smaller cooldown period) is in general preferable to having a longer cooldown period, which is better when the load is changing very frequently.

11. Conclusions

In the previous sections, the modelling extensions to support custom FaaS applications were discussed, along with a novel approach which can guide the adaptation of their processing topology. The modelling extensions allow the use of resources at the edge and the cloud, and provide an understandable view of the processing topology while allowing optimization to be carried out subsequently. Moreover, a novel approach to adapt these custom FaaS applications was described, using a small amount of input from elasticity rules. FaaS is one of the most influential technologies which have appeared in the context of Cloud computing over the past few years. This proposal involves using a two-flavoured TOSCA scheme which allows the DevOps to unlock the potential of optimized hybrid cloud/edge deployments and easily configure the criteria governing the deployment of components. The expression of constraints is built-in the topology template, and complete configuration over each fragment of the topology is available. Moreover, each node template and node type have the same structure and follow the same conventions, therefore improving the understanding of the TOSCA document, whether at the instance or the type-level. This work is backed by a software implementation of the most important modelling stage, the type-level TOSCA generation, and also an implementation of an adaptation manager based on a Severity technique. The presented approach along with the appropriate fine-tuning of the open-source type-level TOSCA generator and Adapter, coupled to proper orchestration and optimization capabilities which need to be externally implemented (possibly interfacing with successful commercial products) can offer model-driven adaptation for FaaS applications in a fully functional platform.

12. References

1. Bergmayr, A.; Breitenbücher, U.; Ferry, N.; Rossini, A.; Solberg, A.; Wimmer, M.; Kappel, G.; Leymann, F. A Systematic Review of Cloud Modeling Languages. *ACM Comput. Surv.* **2018**, *51*, 1–38, doi:10.1145/3150227.
2. Alien4cloud/Alien4cloud Available online: https://github.com/alien4cloud/alien4cloud (accessed on 12 February 2021).
3. Cloudify-Cosmo Available online: https://github.com/cloudify-cosmo (accessed on 12 February 2021).
4. Binz, T.; Breitenbücher, U.; Haupt, F.; Kopp, O.; Leymann, F.; Nowak, A.; Wagner, S. OpenTOSCA – A Runtime for TOSCA-Based Cloud Applications. In Proceedings of the Service-Oriented Computing; Basu, S., Pautasso, C., Zhang, L., Fu, X., Eds.; Springer: Berlin, Heidelberg, 2013; pp. 692–695.
5. Kritikos, K.; Skrzypek, P.; Moga, A.; Matei, O. Towards the Modelling of Hybrid Cloud Applications. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD); IEEE: Milan, Italy, July 2019; pp. 291–295.
6. Verginadis, Y.; Apostolou, D.; Taherizadeh, S.; Ledakis, I.; Mentzas, G.; Tsagkaropoulos, A.; Papageorgiou, N.; Paraskevopoulos, F. PrEstoCloud: A Novel Framework for Data-Intensive Multi-Cloud, Fog, and Edge Function-as-a-Service Applications. *Information Resources Management Journal* **2021**, *34*, 66–85, doi:10.4018/IRMJ.2021010104.
7. TOSCA Simple Profile in YAML Version 2.0 Available online: https://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html (accessed on 19 July 2021).
8. Wurster, M.; Breitenbucher, U.; Kepes, K.; Leymann, F.; Yussupov, V. Modeling and Automated Deployment of Serverless Applications Using TOSCA. In Proceedings of the 2018 IEEE 11th Conference on Service-Oriented Computing and Applications (SOCA); IEEE: Paris, November 2018; pp. 73–80.
9. Casale, G.; Artač, M.; van den Heuvel, W.-J.; van Hoorn, A.; Jakovits, P.; Leymann, F.; Long, M.; Papanikolaou, V.; Presenza, D.; Russo, A.; et al. RADON: Rational Decomposition and Orchestration for Serverless Computing. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2020**, *35*, 77–87, doi:10.1007/s00450-019-00413-w.
10. Paasage Public Deliverables - D2.1.3 Camel Documentation Available online: https://paasage.ercim.eu/images/documents/docs/D2.1.3_CAMEL_Documentation.pdf (accessed on 12 February 2021).
11. Achilleos, A.P.; Kritikos, K.; Rossini, A.; Kapitsaki, G.M.; Domaschka, J.; Orzechowski, M.; Seybold, D.; Griesinger, F.; Nikolov, N.; Romero, D.; et al. The Cloud Application Modelling and Execution Language. *J Cloud Comp* **2019**, *8*, 20, doi:10.1186/s13677-019-0138-7.
12. AWS IoT Greengrass - Amazon Web Services Available online: https://aws.amazon.com/greengrass/ (accessed on 22 July 2021).
13. IoT Edge | Microsoft Azure Available online: https://azure.microsoft.com/en-us/services/iot-edge/ (accessed on 22 July 2021).
14. van Lingen, F.; Yannuzzi, M.; Jain, A.; Irons-Mclean, R.; Lluch, O.; Carrera, D.; Perez, J.L.; Gutierrez, A.; Montero, D.; Marti, J.; et al. The Unavoidable Convergence of NFV, 5G, and Fog: A Model-Driven Approach to Bridge Cloud and Edge. *IEEE Commun. Mag.* **2017**, *55*, 28–35, doi:10.1109/MCOM.2017.1600907.
15. Bjorklund, M. The YANG 1.1 Data Modeling Language Available online: https://www.rfc-editor.org/info/rfc7950 (accessed on 12 February 2021).

16. Chen, T.; Bahsoon, R.; Yao, X. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Comput. Surv.* **2019**, *51*, 1–40, doi:10.1145/3190507.
17. Lim, H.C.; Babu, S.; Chase, J.S.; Parekh, S.S. Automated Control in Cloud Computing: Challenges and Opportunities. In Proceedings of the Proceedings of the 1st workshop on Automated control for datacenters and clouds; Association for Computing Machinery: New York, NY, USA, June 19 2009; pp. 13–18.
18. Zhu, Q.; Agrawal, G. Resource Provisioning with Budget Constraints for Adaptive Applications in Cloud Environments. *IEEE Transactions on Services Computing* **2012**, *5*, 497–511, doi:10.1109/TSC.2011.61.
19. Ashraf, A.; Byholm, B.; Lehtinen, J.; Porres, I. Feedback Control Algorithms to Deploy and Scale Multiple Web Applications per Virtual Machine. In Proceedings of the 2012 38th Euromicro Conference on Software Engineering and Advanced Applications; September 2012; pp. 431–438.
20. Lorido-Botran, T.; Miguel-Alonso, J.; Lozano, J.A. A Review of Auto-Scaling Techniques for Elastic Applications in Cloud Environments. *J Grid Computing* **2014**, *12*, 559–592, doi:10.1007/s10723-014-9314-7.
21. Prototype TOSCA Generator (Application Fragmentation & Deployment Recommender) Available online: https://gitlab.com/prestocloud-project/application-fragmentation-deployment-recommender (accessed on 12 February 2021).
22. Gartner Says Four Trends Are Shaping the Future of Public Cloud Available online: https://www.gartner.com/en/newsroom/press-releases/2021-08-02-gartner-says-four-trends-are-shaping-the-future-of-public-cloud (accessed on 28 February 2022).
23. Opara-Martins, J.; Sahandi, R.; Tian, F. Critical Analysis of Vendor Lock-in and Its Impact on Cloud Computing Migration: A Business Perspective. *J Cloud Comp* **2016**, *5*, 4, doi:10.1186/s13677-016-0054-z.
24. Castro, P.; Ishakian, V.; Muthusamy, V.; Slominski, A. The Rise of Serverless Computing. *Commun. ACM* **2019**, *62*, 44–54, doi:10.1145/3368454.
25. Baldini, I.; Castro, P.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Suter, P. Cloud-Native, Event-Based Programming for Mobile Applications. In Proceedings of the Proceedings of the International Conference on Mobile Software Engineering and Systems; Association for Computing Machinery: New York, NY, USA, December 14 2016; pp. 287–288.
26. Kpavel/Incubator-Openwhisk Available online: https://github.com/kpavel/incubator-openwhisk (accessed on 12 February 2021).
27. Gandhi, A.; Dube, P.; Karve, A.; Kochut, A.; Zhang, L. Adaptive, Model-Driven Autoscaling for Cloud Applications.; 2014; pp. 57–64.
28. Pan, S.J.; Yang, Q. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* **2010**, *22*, 1345–1359, doi:10.1109/TKDE.2009.191.
29. Schmidt, D.C. Guest Editor's Introduction: Model-Driven Engineering. *Computer* **2006**, *39*, 25–31, doi:10.1109/MC.2006.58.
30. Glaser, F. Domain Model Optimized Deployment and Execution of Cloud Applications with TOSCA. In Proceedings of the System Analysis and Modeling. Technology-Specific Aspects of Models; Grabowski, J., Herbold, S., Eds.; Springer International Publishing: Cham, 2016; pp. 68–83.
31. Challita, S.; Korte, F.; Erbel, J.; Zalila, F.; Grabowski, J.; Merle, P. Model-Based Cloud Resource Management with TOSCA and OCCI. *Softw Syst Model* **2021**, doi:10.1007/s10270-021-00869-y.

32. van Eyk, E.; Iosup, A.; Seif, S.; Thömmes, M. The SPEC Cloud Group's Research Vision on FaaS and Serverless Architectures. In Proceedings of the Proceedings of the 2nd International Workshop on Serverless Computing; Association for Computing Machinery: New York, NY, USA, December 11 2017; pp. 1–4.
33. Eismann, S.; Scheuner, J.; van Eyk, E.; Schwinger, M.; Grohmann, J.; Herbst, N.; Abad, C.L.; Iosup, A. Serverless Applications: Why, When, and How? *IEEE Software* **2021**, *38*, 32–39, doi:10.1109/MS.2020.3023302.
34. Hellerstein, J.M.; Faleiro, J.; Gonzalez, J.E.; Schleier-Smith, J.; Sreekanti, V.; Tumanov, A.; Wu, C. Serverless Computing: One Step Forward, Two Steps Back. *arXiv:1812.03651 [cs]* **2018**.
35. Tsagkaropoulos, A.; Verginadis, Y.; Compastié, M.; Apostolou, D.; Mentzas, G. Extending TOSCA for Edge and Fog Deployment Support. *Electronics* **2021**, *10*, 737, doi:10.3390/electronics10060737.
36. Tamburri, D.A.; Van den Heuvel, W.-J.; Lauwers, C.; Lipton, P.; Palma, D.; Rutkowski, M. TOSCA-Based Intent Modelling: Goal-Modelling for Infrastructure-as-Code. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2019**, *34*, 163–172, doi:10.1007/s00450-019-00404-x.
37. Yussupov, V.; Soldani, J.; Breitenbücher, U.; Leymann, F. Standards-Based Modeling and Deployment of Serverless Function Orchestrations Using BPMN and TOSCA. *Software: Practice and Experience n/a*, doi:10.1002/spe.3073.
38. RADON Public Deliverables - D2.4 Architecture and Integration Plan II Available online: https://radon-h2020.eu/wp-content/uploads/2020/07/D2.4-Architecture-and-integration-plan-II.pdf (accessed on 12 February 2021).
39. Nyrén, R.; Edmonds, A.; Papaspyrou, A.; Metsch, T.; Parák, B. Open Cloud Computing Interface - Core Available online: https://redmine.ogf.org/attachments/242/core.pdf (accessed on 12 February 2021).
40. Glaser, F.; Erbel, J.; Grabowski, J. Model Driven Cloud Orchestration by Combining TOSCA and OCCI: In Proceedings of the Proceedings of the 7th International Conference on Cloud Computing and Services Science; SCITEPRESS - Science and Technology Publications: Porto, Portugal, 2017; pp. 672–678.
41. Wurster, M.; Breitenbücher, U.; Falkenthal, M.; Krieger, C.; Leymann, F.; Saatkamp, K.; Soldani, J. The Essential Deployment Metamodel: A Systematic Review of Deployment Automation Technologies. *SICS Softw.-Inensiv. Cyber-Phys. Syst.* **2020**, *35*, 63–75, doi:10.1007/s00450-019-00412-x.
42. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646, doi:10.1109/JIOT.2016.2579198.
43. Noghabi, S.A.; Kolb, J.; Bodik, P.; Cuervo, E. Steel: Simplified Development and Deployment of Edge-Cloud Applications.; 2018.
44. Mortazavi, S.H.; Salehe, M.; Gomes, C.S.; Phillips, C.; de Lara, E. Cloudpath: A Multi-Tier Cloud Computing Framework. In Proceedings of the Proceedings of the Second ACM/IEEE Symposium on Edge Computing; Association for Computing Machinery: New York, NY, USA, October 12 2017; pp. 1–13.
45. Podolskiy, V.; Jindal, A.; Gerndt, M. IaaS Reactive Autoscaling Performance Challenges. In Proceedings of the 2018 IEEE 11th International Conference on Cloud Computing (CLOUD); July 2018; pp. 954–957.
46. Podolskiy, V.; Mayo, M.; Koay, A.; Gerndt, M.; Patros, P. Maintaining SLOs of Cloud-Native Applications Via Self-Adaptive Resource Sharing. In Proceedings of the 2019 IEEE 13th International Conference on Self-Adaptive and Self-Organizing Systems (SASO); June 2019; pp. 72–81.

47. Arkian, H.; Pierre, G.; Tordsson, J.; Elmroth, E. Model-Based Stream Processing Auto-Scaling in Geo-Distributed Environments. In Proceedings of the 2021 International Conference on Computer Communications and Networks (ICCCN); July 2021; pp. 1–10.
48. Taherizadeh, S.; Stankovski, V. Dynamic Multi-Level Auto-Scaling Rules for Containerized Applications. *The Computer Journal* **2019**, *62*, 174–197, doi:10.1093/comjnl/bxy043.
49. Overview of Autoscale with Azure Virtual Machine Scale Sets - Azure Virtual Machine Scale Sets Available online: https://docs.microsoft.com/en-us/azure/virtual-machine-scale-sets/virtual-machine-scale-sets-autoscale-overview (accessed on 24 February 2022).
50. Oracle Cloud Infrastructure Documentation – Autoscaling Available online: https://docs.oracle.com/en-us/iaas/Content/Compute/Tasks/autoscalinginstancepools.htm (accessed on 24 February 2022).
51. Taherizadeh, S.; Grobelnik, M. Key Influencing Factors of the Kubernetes Auto-Scaler for Computing-Intensive Microservice-Native Cloud-Based Applications. *Advances in Engineering Software* **2020**, *140*, 102734, doi:10.1016/j.advengsoft.2019.102734.
52. Lorido-Botrán, T.; Miguel-Alonso, J.; Lozano, J. Comparison of Auto-Scaling Techniques for Cloud Environments.; January 1 2013.
53. Vaquero, L.M.; Morán, D.; Galán, F.; Alcaraz-Calero, J.M. Towards Runtime Reconfiguration of Application Control Policies in the Cloud. *J Netw Syst Manage* **2012**, *20*, 489–512, doi:10.1007/s10922-012-9251-3.
54. Galante, G.; Bona, L.C.E. Constructing Elastic Scientific Applications Using Elasticity Primitives. In *Computational Science and Its Applications – ICCSA 2013*; Murgante, B., Misra, S., Carlini, M., Torre, C.M., Nguyen, H.-Q., Taniar, D., Apduhan, B.O., Gervasi, O., Eds.; Lecture Notes in Computer Science; Springer Berlin Heidelberg: Berlin, Heidelberg, 2013; Vol. 7975, pp. 281–294 ISBN 978-3-642-39639-7.
55. Copil, G.; Moldovan, D.; Truong, H.-L.; Dustdar, S. Multi-Level Elasticity Control of Cloud Services. In Proceedings of the Service-Oriented Computing; Basu, S., Pautasso, C., Zhang, L., Fu, X., Eds.; Springer: Berlin, Heidelberg, 2013; pp. 429–436.
56. Copil, G.; Moldovan, D.; Truong, H.-L.; Dustdar, S. RSYBL: A Framework for Specifying and Controlling Cloud Services Elasticity. *ACM Trans. Internet Technol.* **2016**, *16*, 18:1-18:20, doi:10.1145/2925990.
57. Ferretti, S.; Ghini, V.; Panzieri, F.; Pellegrini, M.; Turrini, E. QoS–Aware Clouds. In Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing; July 2010; pp. 321–328.
58. Trihinas, D.; Georgiou, Z.; Pallis, G.; Dikaiakos, M.D. Improving Rule-Based Elasticity Control by Adapting the Sensitivity of the Auto-Scaling Decision Timeframe. In Proceedings of the Algorithmic Aspects of Cloud Computing; Alistarh, D., Delis, A., Pallis, G., Eds.; Springer International Publishing: Cham, 2018; pp. 123–137.
59. Dutreilh, X.; Moreau, A.; Malenfant, J.; Rivierre, N.; Truck, I. From Data Center Resource Allocation to Control Theory and Back. In Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing; July 2010; pp. 410–417.
60. Ali-Eldin, A.; Tordsson, J.; Elmroth, E. An Adaptive Hybrid Elasticity Controller for Cloud Infrastructures. In Proceedings of the 2012 IEEE Network Operations and Management Symposium; April 2012; pp. 204–212.
61. Ali-Eldin, A.; Kihl, M.; Tordsson, J.; Elmroth, E. Efficient Provisioning of Bursty Scientific Workloads on the Cloud Using Adaptive Elasticity Control. In Proceedings of the Proceedings of the 3rd workshop on Scientific Cloud Computing; Association for Computing Machinery: New York, NY, USA, June 18 2012; pp. 31–40.

62. Bauer, A.; Herbst, N.; Spinner, S.; Ali-Eldin, A.; Kounev, S. Chameleon: A Hybrid, Proactive Auto-Scaling Mechanism on a Level-Playing Field. *IEEE Transactions on Parallel and Distributed Systems* **2019**, *30*, 800–813, doi:10.1109/TPDS.2018.2870389.
63. Ramirez, Y.M.; Podolskiy, V.; Gerndt, M. Capacity-Driven Scaling Schedules Derivation for Coordinated Elasticity of Containers and Virtual Machines. In Proceedings of the 2019 IEEE International Conference on Autonomic Computing (ICAC); June 2019; pp. 177–186.
64. Zhu, L.; Giotis, G.; Tountopoulos, V.; Casale, G. RDOF: Deployment Optimization for Function as a Service. In Proceedings of the 2021 IEEE 14th International Conference on Cloud Computing (CLOUD); September 2021; pp. 508–514.
65. Tamiru, M.A.; Tordsson, J.; Elmroth, E.; Pierre, G. An Experimental Evaluation of the Kubernetes Cluster Autoscaler in the Cloud. In Proceedings of the 2020 IEEE International Conference on Cloud Computing Technology and Science (CloudCom); December 2020; pp. 17–24.
66. Rzadca, K.; Findeisen, P.; Swiderski, J.; Zych, P.; Broniek, P.; Kusmierek, J.; Nowak, P.; Strack, B.; Witusowski, P.; Hand, S.; et al. Autopilot: Workload Autoscaling at Google. In Proceedings of the Proceedings of the Fifteenth European Conference on Computer Systems; Association for Computing Machinery: New York, NY, USA, April 15 2020; pp. 1–16.
67. Predictive Scaling for EC2, Powered by Machine Learning Available online: https://aws.amazon.com/blogs/aws/new-predictive-scaling-for-ec2-powered-by-machine-learning/ (accessed on 24 February 2022).
68. Tsagkaropoulos, A.; Papageorgiou, N.; Apostolou, D.; Verginadis, Y.; Mentzas, G. Challenges and Research Directions in Big Data-Driven Cloud Adaptivity.; January 1 2018; pp. 190–200.
69. Jussien, N.; Rochart, G.; Lorca, X. Choco: An Open Source Java Constraint Programming Library. In Proceedings of the CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08); Paris, France, France, 2008; pp. 1–10.
70. Hermenier, F.; Lawall, J.; Muller, G. BtrPlace: A Flexible Consolidation Manager for Highly Available Applications. *IEEE Trans. Dependable and Secure Comput.* **2013**, *10*, 273–286, doi:10.1109/TDSC.2013.5.
71. Papageorgiou, N.; Verginadis, Y.; Apostolou, D.; Mentzas, G. Fog Computing Context Analytics. *IEEE Instrumentation Measurement Magazine* **2019**, *22*, 53–59, doi:10.1109/MIM.2019.8917904.
72. Alsmeyer, G. Chebyshev's Inequality. In *International Encyclopedia of Statistical Science*; Lovric, M., Ed.; Springer Berlin Heidelberg: Berlin, Heidelberg, 2011; pp. 239–240 ISBN 978-3-642-04897-5.
73. Severity-Based Situation Detection Mechanism Available online: https://gitlab.com/prestocloud-project/situation-detection-mechanism-v2 (accessed on 24 March 2022).
74. Tsagkaropoulos, A.; Verginadis, Y.; Papageorgiou, N.; Paraskevopoulos, F.; Apostolou, D.; Mentzas, G. Severity: A QoS-Aware Approach to Cloud Application Elasticity. *J Cloud Comp* **2021**, *10*, 45, doi:10.1186/s13677-021-00255-5.
75. Suhothayan, S.; Gajasinghe, K.; Loku Narangoda, I.; Chaturanga, S.; Perera, S.; Nanayakkara, V. Siddhi: A Second Look at Complex Event Processing Architectures. In Proceedings of the Proceedings of the 2011 ACM workshop on Gateway computing environments; Association for Computing Machinery: New York, NY, USA, November 18 2011; pp. 43–50.

76. Sun, Y.; Lin, F.; Xu, H. Multi-Objective Optimization of Resource Scheduling in Fog Computing Using an Improved NSGA-II. *Wireless Pers Commun* **2018**, *102*, 1369–1385, doi:10.1007/s11277-017-5200-5.
77. Zhu, Z.; Zhang, G.; Li, M.; Liu, X. Evolutionary Multi-Objective Workflow Scheduling in Cloud. *IEEE Transactions on Parallel and Distributed Systems* **2016**, *27*, 1344–1357, doi:10.1109/TPDS.2015.2446459.
78. Zhang, F.; Cao, J.; Li, K.; Khan, S.U.; Hwang, K. Multi-Objective Scheduling of Many Tasks in Cloud Platforms. *Future Generation Computer Systems* **2014**, *37*, 309–320, doi:10.1016/j.future.2013.09.006.
79. Simic, V.; Stojanovic, B.; Ivanovic, M. Optimizing the Performance of Optimization in the Cloud Environment–An Intelligent Auto-Scaling Approach. *Future Generation Computer Systems* **2019**, *101*, 909–920, doi:10.1016/j.future.2019.07.042.
80. Barnawi, A.; Sakr, S.; Xiao, W.; Al-Barakati, A. The Views, Measurements and Challenges of Elasticity in the Cloud: A Review. *Computer Communications* **2020**, *154*, 111–117, doi:10.1016/j.comcom.2020.02.010.
81. Adaptation Technique Performance Using 2, 3 and 4-Metric Workloads Available online: http://imu.ntua.gr/static/workloads/ (accessed on 26 February 2022).
82. Krishnaprasad, S. Uses and Abuses of Amdahl's Law. *J. Comput. Sci. Coll.* **2001**, *17*, 288–293.
83. Wurster, M.; Breitenbücher, U.; Harzenetter, L.; Leymann, F.; Soldani, J. TOSCA Lightning: An Integrated Toolchain for Transforming TOSCA Light into Production-Ready Deployment Technologies. In Proceedings of the Advanced Information Systems Engineering; Herbaut, N., La Rosa, M., Eds.; Springer International Publishing: Cham, 2020; pp. 138–146.
84. Jain, A.; Baarzi, A.F.; Alfares, N.; Kesidis, G.; Urgaonkar, B.; Kandemir, M. SpIitServe: Efficiently Splitting Complex Workloads Across FaaS and IaaS. In Proceedings of the Proceedings of the ACM Symposium on Cloud Computing; ACM: Santa Cruz CA USA, November 20 2019; pp. 487–487.
85. Wurster, M.; Breitenbücher, U.; Brogi, A.; Falazi, G.; Harzenetter, L.; Leymann, F.; Soldani, J.; Yussupov, V. The EDMM Modeling and Transformation System. In Proceedings of the Service-Oriented Computing – ICSOC 2019 Workshops; Yangui, S., Bouguettaya, A., Xue, X., Faci, N., Gaaloul, W., Yu, Q., Zhou, Z., Hernandez, N., Nakagawa, E.Y., Eds.; Springer International Publishing: Cham, 2020; pp. 294–298.
86. Wurster, M.; Breitenbücher, U.; Harzenetter, L.; Leymann, F.; Soldani, J.; Yussupov, V. TOSCA Light: Bridging the Gap between the TOSCA Specification and Production-Ready Deployment Technologies: In Proceedings of the Proceedings of the 10th International Conference on Cloud Computing and Services Science; SCITEPRESS - Science and Technology Publications: Prague, Czech Republic, 2020; pp. 216–226.

APPENDIX A – Full Type-level TOSCA template

```
tosca_definitions_version: tosca_prestocloud_mapping_1_2

metadata:
  template_name: IMU generated types definition
  template_author: IMU
  template_version: 1.0.0-SNAPSHOT
  CostThreshold: 1000
  TimePeriod: 720
  ProviderName_0: Google_Cloud_Compute
  ProviderRequired_0: false
  ProviderExcluded_0: true
  MetricToMinimize: Cost

description: Types Description

imports:
  - tosca-normative-types:1.2
  - iccs-normative-types:1.1
  - resource-descriptions:1.0
  - placement-constraints:1.0

node_types:
  #Processing node selection:
  processing_node_LambdaProxy_0:
    description: A TOSCA representation of a processing node
    derived_from: prestocloud.nodes.proxy.faas
    requirements:
      - host:
        capability: tosca.capabilities.Container
        node: prestocloud.nodes.compute
        relationship: tosca.relationships.HostedOn
        node_filter:
          capabilities:
            - host:
              properties:
                - num_cpus: { in_range: [ 2, 4 ] }
                - mem_size: { in_range: [ 2048 MB, 4096 MB] }
                - storage_size: { in_range: [ 10 GB, 50 GB ] }
            - os:
              properties:
                - architecture: { valid_values: [ x86_64, i386 ] }
                - type: { equal: linux }
                - distribution: { equal: ubuntu }
            - resource:
              properties:
```

```
            - type: { equal: cloud }

processing_node_imu_fragments_MultimediaManager_1:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent
  requirements:
    - host:
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
              properties:
                - num_cpus: { in_range: [ 2, 4 ] }
                - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
                - storage_size: { in_range: [ 128 GB, 1024 GB ] }
          - os:
              properties:
                - architecture: { valid_values: [ x86_64, i386 ] }
                - type: { equal: linux }
                - distribution: { equal: ubuntu }
          - resource:
              properties:
                - type: { equal: cloud }

processing_node_imu_fragments_VideoTranscoder_2:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent
  requirements:
    - host:
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
              properties:
                - num_cpus: { in_range: [ 2, 4 ] }
                - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
                - storage_size: { in_range: [ 4 GB, 32 GB ] }
          - os:
              properties:
                - architecture: { valid_values: [ x86_64, i386 ] }
                - type: { equal: linux }
                - distribution: { equal: ubuntu }
          - resource:
              properties:
```

```
            - type: { equal: cloud }
    - host:
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
            properties:
              - num_cpus: { in_range: [ 2, 4 ] }
              - mem_size: { in_range: [ 2048 MB, 4096 MB ] }
              - storage_size: { in_range: [ 4 GB, 32 GB ] }
          - os:
            properties:
              - architecture: { valid_values: [ arm64, armel, armhf ] }
              - type: { equal: linux }
              - distribution: { equal: raspbian }
          - resource:
            properties:
              - type: { equal: edge }

processing_node_imu_fragments_AudioCaptor_3:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent
  requirements:
    - host:
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
            properties:
              - num_cpus: { in_range: [ 1, 2 ] }
              - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
              - storage_size: { in_range: [ 4 GB, 32 GB ] }
          - os:
            properties:
              - architecture: { valid_values: [ arm64, armel, armhf ] }
              - type: { equal: linux }
              - distribution: { equal: raspbian }
          - resource:
            properties:
              - type: { equal: edge }
          - sensors:
            properties:
              - microphone: { equal: "/dev/snd/mic0" }
```

```
processing_node_imu_fragments_FaceDetector_4:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent.faas
  requirements:
    - host:
        capability: tosca.capabilities.Container
        node: prestocloud.nodes.compute
        relationship: tosca.relationships.HostedOn
        node_filter:
          capabilities:
            - host:
                properties:
                  - num_cpus: { in_range: [ 1, 2 ] }
                  - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
                  - storage_size: { in_range: [ 4 GB, 32 GB ] }
            - os:
                properties:
                  - architecture: { valid_values: [ x86_64, i386 ] }
                  - type: { equal: linux }
                  - distribution: { equal: ubuntu }
            - resource:
                properties:
                  - type: { equal: cloud }
    - host:
        capability: tosca.capabilities.Container
        node: prestocloud.nodes.compute
        relationship: tosca.relationships.HostedOn
        node_filter:
          capabilities:
            - host:
                properties:
                  - num_cpus: { in_range: [ 1, 2 ] }
                  - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
                  - storage_size: { in_range: [ 4 GB, 32 GB ] }
            - os:
                properties:
                  - architecture: { valid_values: [ arm64, armel, armhf ] }
                  - type: { equal: linux }
                  - distribution: { equal: raspbian }
            - resource:
                properties:
                  - type: { equal: edge }

processing_node_imu_fragments_PercussionDetector_5:
  description: A TOSCA representation of a processing node
  derived_from: prestocloud.nodes.agent.faas
  requirements:
    - host:
```

```
      capability: tosca.capabilities.Container
      node: prestocloud.nodes.compute
      relationship: tosca.relationships.HostedOn
      node_filter:
        capabilities:
          - host:
              properties:
                - num_cpus: { in_range: [ 1, 2 ] }
                - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
                - storage_size: { in_range: [ 4 GB, 32 GB ] }
          - os:
              properties:
                - architecture: { valid_values: [ arm64, armel, armhf ] }
                - type: { equal: linux }
                - distribution: { equal: raspbian }
          - resource:
              properties:
                - type: { equal: edge }

  processing_node_imu_fragments_VideoStreamer_6:
    description: A TOSCA representation of a processing node
    derived_from: prestocloud.nodes.agent.faas
    requirements:
      - host:
          capability: tosca.capabilities.Container
          node: prestocloud.nodes.compute
          relationship: tosca.relationships.HostedOn
          node_filter:
            capabilities:
              - host:
                  properties:
                    - num_cpus: { in_range: [ 1, 2 ] }
                    - mem_size: { in_range: [ 1024 MB, 2048 MB ] }
                    - storage_size: { in_range: [ 4 GB, 32 GB ] }
              - os:
                  properties:
                    - architecture: { valid_values: [ arm64, armel, armhf ] }
                    - type: { equal: linux }
                    - distribution: { equal: raspbian }
              - resource:
                  properties:
                    - type: { equal: edge }
              - sensors:
                  properties:
                    - video_camera: { equal: "/dev/video/camera0" }


topology_template:
```

```
  policies:
    - collocation_policy_group_0:
      type: prestocloud.placement.Gather
      targets: [ imu_fragments_VideoStreamer, imu_fragments_VideoTranscoder ]

    - collocation_policy_group_1:
      type: prestocloud.placement.Gather
      targets: [ imu_fragments_PercussionDetector, imu_fragments_AudioCaptor ]

    - precedence_policy_group_0:
      type: prestocloud.placement.Precedence
      targets: [
imu_fragments_VideoStreamer,imu_fragments_VideoTranscoder,imu_fragments_FaceDetector,i
mu_fragments_MultimediaManager,imu_fragments_AudioCaptor,imu_fragments_PercussionDe
tector ]

  node_templates:
    deployment_node_LambdaProxy:
      type: processing_node_LambdaProxy_0

    LambdaProxy:
      type: prestocloud.nodes.fragment
      properties:
        id: 6
        name: LambdaProxy
        scalable: false
        occurrences: 1
        docker_cloud:
          image: "traefik:latest"
          registry: "hub.docker.com"
          ports:
            - target: 11111
              published: 11111
              protocol: TCP
            - target: 11198
              published: 11198
              protocol: TCP
        optimization_variables:
          cost: 1
          distance: 1
          friendliness: { }
      requirements:
        - execute: deployment_node_LambdaProxy

    deployment_node_imu_fragments_MultimediaManager:
      type: processing_node_imu_fragments_MultimediaManager_1

    imu_fragments_MultimediaManager:
```

```
    type: prestocloud.nodes.fragment
    properties:
      id: 0
      name: imu_fragments.MultimediaManager
      scalable: false
      occurrences: 1
      docker_cloud:
        image: "multimedia_manager:latest"
        registry: "prestocloud.test.eu"
        variables: { "VIDEO_TRANSCODER_SERVICE": "{ get_property:
[deployment_node_LambdaProxy,host,network,addresses,1] }", "FACE_DETECTOR_SERVICE":
"{ get_property: [deployment_node_LambdaProxy,host,network,addresses,1] }",
"RUNNING_THREADS": "2" }
      optimization_variables:
        cost: 5
        distance: 4
        friendliness: { "aws": "5", "gce": "0", "azure": "1" }
    requirements:
      - execute: deployment_node_imu_fragments_MultimediaManager

  deployment_node_imu_fragments_VideoTranscoder:
    type: processing_node_imu_fragments_VideoTranscoder_2

  imu_fragments_VideoTranscoder:
    type: prestocloud.nodes.fragment
    properties:
      id: 1
      name: imu_fragments.VideoTranscoder
      scalable: true
      occurrences: 1
      docker_edge:
        image: "video_transcoder_edge:latest"
        registry: "prestocloud.edge.test.eu"
        ports:
          - target: 10000
            published: 10000
            protocol: TCP_UDP
      docker_cloud:
        image: "video_transcoder_cloud:latest"
        registry: "prestocloud.test.eu"
        ports:
          - target: 10000
            published: 10000
            protocol: TCP_UDP
      optimization_variables:
        cost: 2
        distance: 8
        friendliness: { "aws": "5", "gce": "0", "azure": "1" }
```

```
  requirements:
    - execute: deployment_node_imu_fragments_VideoTranscoder

deployment_node_imu_fragments_AudioCaptor:
  type: processing_node_imu_fragments_AudioCaptor_3

imu_fragments_AudioCaptor:
  type: prestocloud.nodes.fragment
  properties:
    id: 2
    name: imu_fragments.AudioCaptor
    scalable: false
    occurrences: 1
    docker_edge:
      image: "audiocaptor:latest"
      registry: "prestocloud.test.eu"
      variables: { "SAMPLING_RATE": "22 kHZ" }
    optimization_variables:
      cost: 1
      distance: 1
      friendliness: { }
    health_check:
      interval: 1
      cmd: "cat /proc/meminfo"
  requirements:
    - execute: deployment_node_imu_fragments_AudioCaptor

deployment_node_imu_fragments_FaceDetector:
  type: processing_node_imu_fragments_FaceDetector_4

imu_fragments_FaceDetector:
  type: prestocloud.nodes.fragment.faas
  properties:
    id: 3
    name: imu_fragments.FaceDetector
    scalable: true
    occurrences: 1
    docker_edge:
      image: "face_detector_edge:latest"
      registry: "local.prestocloud.test.eu"
      variables: { "PRECISION": "50", "ITERATIONS": "10" }
    docker_cloud:
      image: "face_detector_cloud:latest"
      registry: "prestocloud.test.eu"
      variables: { "PRECISION": "100", "ITERATIONS": "2" }
    optimization_variables:
      cost: 1
      distance: 1
```

```
        friendliness: { "aws": "5", "gce": "0", "azure": "1" }
      health_check:
        interval: 1
        cmd: "curl health.prestocloud.test.eu FaceDetector"
    requirements:
      - execute: deployment_node_imu_fragments_FaceDetector
      - proxy: deployment_node_LambdaProxy

  deployment_node_imu_fragments_PercussionDetector:
    type: processing_node_imu_fragments_PercussionDetector_5

  imu_fragments_PercussionDetector:
    type: prestocloud.nodes.fragment.faas
    properties:
      id: 4
      name: imu_fragments.PercussionDetector
      scalable: true
      occurrences: 1
      docker_edge:
        image: "percussion_detector_edge:latest"
        registry: "prestocloud.test.eu"
      docker_cloud:
        image: "percussion_detector_cloud:latest"
        registry: "prestocloud.test.eu"
      optimization_variables:
        cost: 1
        distance: 1
        friendliness: { }
    requirements:
      - execute: deployment_node_imu_fragments_PercussionDetector
      - proxy: deployment_node_LambdaProxy

  deployment_node_imu_fragments_VideoStreamer:
    type: processing_node_imu_fragments_VideoStreamer_6

  imu_fragments_VideoStreamer:
    type: prestocloud.nodes.fragment.faas
    properties:
      id: 5
      name: imu_fragments.VideoStreamer
      scalable: true
      occurrences: 3
      docker_edge:
        image: "video_streamer:latest"
        registry: "prestocloud.test.eu"
        variables: { "VIDEO_TRANSCODER_SERVICE": "{ get_property:
[deployment_node_LambdaProxy,host,network,addresses,1] }", "VIDEO_RESOLUTION":
"HD1080p" }
```

```
  optimization_variables:
    cost: 1
    distance: 1
    friendliness: { }
requirements:
  - execute: deployment_node_imu_fragments_VideoStreamer
  - proxy: deployment_node_LambdaProxy
```

APPENDIX B – Full Terraform template

```
provider "aws" {
  profile    = "default"
  region     = "us-east-1"
}

# Network configuration …Create a VPC to launch instances into
resource "aws_vpc" "default" {
  cidr_block = "10.0.0.0/16"
}

# Create an internet gateway to grant to the subnet access to the outside world
resource "aws_internet_gateway" "default" {
  vpc_id = "${aws_vpc.default.id}"
}

# Grant the VPC internet access on its main route table
resource "aws_route" "internet_access" {
  route_table_id         = "${aws_vpc.default.main_route_table_id}"
  destination_cidr_block = "0.0.0.0/0"
  gateway_id             = "${aws_internet_gateway.default.id}"
}

# Create a subnet to launch instances into
resource "aws_subnet" "default" {
  vpc_id                  = "${aws_vpc.default.id}"
  cidr_block              = "10.0.1.0/24"
  map_public_ip_on_launch = true
}

resource "aws_security_group" "default" {
  name        = "terraform_example_lambda_proxy"
  description = "Used in the terraform"
  vpc_id      = "${aws_vpc.default.id}"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

# A security group for the Lambda Proxy
resource "aws_security_group" "lambda_proxy" {
  name        = "terraform_example_lambda_proxy"
```

```
  description = "Used in the terraform"
  vpc_id      = "${aws_vpc.default.id}"

  ingress {
    from_port   = 22
    to_port     = 22
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 11111
    to_port     = 11111
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }

  ingress {
    from_port   = 11198
    to_port     = 11198
    protocol    = "tcp"
    cidr_blocks = ["0.0.0.0/0"]
  }
}

variable "key_name" {}
variable "public_key_path" {}

resource "aws_key_pair" "auth" {
  key_name   = "${var.key_name}"
  public_key = "${file(var.public_key_path)}"
}

resource "aws_instance" "FaceDetector" {
  ami           = "ami-2757f631"
  instance_type = "t2.micro"
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
  depends_on = [aws_instance.VideoTranscoder]
}

resource "aws_instance" "VideoTranscoder" {
  ami           = "ami-2757f621"
  instance_type = "c5.large "
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
```

```
  depends_on = [docker_container.video_streamer]
}

resource "aws_instance" "LambdaProxy" {
  ami           = "ami-2757f622"
  instance_type = "c5.large "
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
  depends_on =
[docker_container.percussion_detector,aws_instance.FaceDetector,aws_instance.VideoTranscode
r]
}

resource "aws_instance" "MultimediaManager" {
  ami           = "ami-2757f611"
  instance_type = "t3a.medium "
  key_name = "${aws_key_pair.auth.id}"
  vpc_security_group_ids = ["${aws_security_group.default.id}"]
  subnet_id = "${aws_subnet.default.id}"
  depends_on =
[aws_instance.FaceDetector,aws_instance.VideoTranscoder,docker_container.percussion_detecto
r]
}

# Configure the Docker providers
provider "docker" {
  host = "tcp://192.168.1.2:2375/"
}

provider "docker" {
  alias = "worker_2"
  host = "tcp://192.168.1.3:2375/"
}

provider "docker" {
  alias = "worker_3"
  host = "tcp://192.168.1.4:2375/"
}

# Create a container
resource "docker_container" "video_streamer" {
  image = docker_image.vs_image.latest
  name  = "vs_cont"
}

resource "docker_container" "audio_captor" {
  provider = docker.worker_2
```

```
  image = docker_image.ac_image.latest
  name  = "ac_cont"
}

resource "docker_container" "percussion_detector" {
  provider = docker.worker_3
  image = docker_image.pd_image.latest
  name  = "pd_cont"
  depends_on = [docker_container.audio_captor]
}

resource "docker_image" "vs_image" {
  name = "video_streamer:latest"
}

resource "docker_image" "ac_image" {
  provider = docker.worker_2
  name = "audio_captor:latest"
}

resource "docker_image" "pd_image" {
  provider = docker.worker_3
  name = "percussion_detector:latest"
}
```