Integrating Quantum Computation into Software Production Environments

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

Master of Science

in the Department of Computer Science

by

Juan Giraldo

B.Eng. in Software Systems Engineering, Universidad Icesi, Colombia, 2021

We acknowledge and respect the lək^{w}əŋən peoples on whose traditional territory the university stands, and the Songhees, Esquimalt and W̱SÁNEĆ peoples whose historical relationships with the land continue to this day.

Integrating Quantum Computation into Software Production Environments

by

Juan Giraldo

B.Eng. in Software Systems Engineering, Universidad Icesi, Colombia, 2021

Supervisory Committee:

Dr. Hausi A. Müller, Supervisor
Department of Computer Science, University of Victoria

Dr. Norha M. Villegas, Supervisor
Department of Computer Science, University of Victoria

Dr. Nikitas J. Dimopoulos, Outside Member
Department of Electrical and Computer Engineering, University of Victoria

Abstract

Increasingly complex dynamics in the software operations pose formidable software evolution challenges to the software industry. Examples of these dynamics include the globalization of software markets, the massive increase of interconnected devices worldwide with the internet of things, and the digital transformation to large-scale cyber-physical systems. To tackle these challenges, researchers and practitioners have developed impressive bodies of knowledge, including adaptive and autonomic systems, run-time models, continuous software engineering, and the practice of combining software development and operations (*i.e.*, DevOps). Despite the tremendous strides the software engineering community has made toward managing highly dynamic systems, software-intensive industries face major challenges to match the ever-increasing pace. To cope with this rapid rate at which operational contexts for software systems change, organizations are required to automate and expedite software evolution on both the development and operations sides.

The aim of our research is to develop continuous and autonomic methods, infrastructures, and tools to realize software evolution holistically. In this dissertation, we shift the prevalent autonomic computing paradigm and provide new perspectives and foci on integrating autonomic computing techniques into continuous software engineering practices, such as DevOps. Our methods and approaches are based on online experimentation and evolutionary optimization. Experimentation allows autonomic managers to make informed data-driven and explainable decisions and present evidence to stakeholders. As a result, autonomic managers contribute to the continuous and holistic evolution of design, configuration and deployment artifacts, providing guarantees on the validity, quality and effectiveness of enacted changes. Ultimately, our approach turns autonomic managers into online stakeholders whose contributions are subject to quality control.

Our contributions are threefold. We focus on effecting long-lasting software changes through self-management, self-improvement, and self-regulation. First, we propose a framework for continuous software evolution pipelines for bridging offline and online evolution processes. Our framework's infrastructure captures run-time changes and turns them into configuration and deployment code updates. Our functional validation on cloud infrastructure management demonstrates its feasibility and soundness. It effectively contributes to eliminate technical debt from the Infrastructure-as-Code (IAC) life cycle, allowing development teams to embrace the benefits of IAC without sacrificing existing automation. Second, we provide a comprehensive implementation for the continuous IAC evolution pipeline. Third, we design a feedback loop to conduct experimentation-driven continuous exploration of design, configuration and deployment alternatives. Our experimental validation demonstrates its capacity to enrich the software architecture with additional

components, and to optimize the computing cluster's configuration, both aiming to reduce service latency. Our feedback loop frees DevOps engineers from incremental improvements, and allows them to focus on long-term mission-critical software evolution changes. Fourth, we define a reference architecture to support short-lived and long-lasting evolution actions at run-time. Our architecture incorporates short-term and long-term evolution as alternating autonomic operational modes. This approach keeps internal models relevant over prolonged system operation, thus reducing the need for additional maintenance. We demonstrate the usefulness of our research in case studies that guide the designs of cloud management systems and a Colombian city transportation system with historical data.

In summary, this dissertation presents a new approach on how to manage software continuity and continuous software improvement effectively. Our methods, infrastructures, and tools constitute a new platform for short-term and long-term continuous integration and software evolution strategies and processes for large-scale intelligent cyber-physical systems. This research is a significant contribution to the long-standing challenges of easing continuous integration and evolution tasks across the development-time and run-time boundary. Thus, we expand the vision of autonomic computing to support software engineering processes from development to production and back. This dissertation constitutes a new holistic approach to the challenges of continuous integration and evolution that strengthens the causalities in current processes and practices, especially from execution back to planning, design, and development.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This dissertation represents more than just a piece of writing, it is a milestone in years of work at the University of Victoria. This work was possible thanks to the collaboration and support of many people, whom I want to acknowledge and thank.

I would first like to express my utmost gratitude to my supervisors, professors Hausi A. Müller and Gabriel Tamura. Their continuous guidance and support throughout this journey, and their motivation and inspiration to endure made this work possible. I will be forever grateful for all their teachings during this time. Thank you, Hausi and Gabriel, for your patience during our meetings and brainstorming sessions. Thank you for your ingenious suggestions to improve my manuscripts. I was lucky to have you both as my supervisors. Your distinct perspectives were always insightful. I will not forget the words of encouragement, the *have fun* and the Fridays to think big thoughts.

I would also like to extend my deepest gratitude to professor Norha M. Villegas, who provided invaluable advice and contributions into my work. Her extensive knowledge, constructive criticism and guidance had a great impact on my publications and on my formative process. Norha, I greatly appreciate the practical and insightful suggestions you provided. I will not forget the happy faces and encouragement phrases you always added to your annotated reviews.

I would like to express my deepest appreciation to my committee, professors Neil Ernst and Issa Traoré, for their insightful comments and hard questions. I also thank Dr. Grace Lewis for acting as the external examiner of my dissertation, and professor Richard Marcy, for serving as the chair of my final oral examination.

To the members of the Rigi-Pita team, Ulrike, Lorena, Felipe, Priya, Ivan, Kirti, Giovanni, Sunil, Karan and Alvi, thank you for the meaningful, and meaningless, conversations we had in the lab. I appreciate and cherish my memories of us spending time and bonding at CASCON, CASTLE and CSER, and will not forget the sleepless nights we worked together before deadlines. I believe the Rigi lab will have the perfect coffee machine someday. Felipe, it has been a pleasure working with you.

I would like to acknowledge the institutions that sponsored my research. This dissertation, and the publications derived from it, were possible thanks to the funding provided by the University of Victoria, the National Sciences and Engineering Research Council (NSERC) of Canada, IBM Corporation, and Universidad Icesi (Colombia). I am specially thankful to the Center for Advance Studies (CAS) at IBM for the opportunity of being a CAS student in multiple projects. Particularly helpful to me during this time were Joe

Wigglesworth, Ian Watts, Hugh Hockett, Joanna W. Ng, and Vio Onut for their valuable feedback, guidance and support.

I would like to express my gratitude to Joe Wigglesworth, Ian Watts, and Hugh Hockett. Thank you for helping shape our contributions and for your guidance and patience during the patenting process of our invention. Each of you contributed in many ways that I appreciate.

I would also like to extend my sincere thanks to the faculty and administrative staff of the University of Victoria. Thank you Kath Milinazzo, Nancy Chan, Wendy Beggs, and Erin Robinson, you have contributed in many ways to the completion of my research. Nancy, your patience cannot be overestimated. I would also like to extend my gratitude to the faculty of the computer science department, especially professors Ulrike Stege, Sudhakar Ganti and Daniela Damian. Thank you for giving me advice, reading manuscripts and enriching my academic experience. Ulrike, I appreciate your constant support at our group meetings. You were one of the first people with whom I worked when I started my PhD, I remember you were very supportive, kind and patient.

I also wish to thank all the researchers whom I have met many times during academic engagements. Among many others, I would like to thank professors Kelly Lyons, Kostas Kontogiannis, Marin Litoiu, Marios-Eleftherios Fokaefs, and Nelly Bencomo for the stimulating discussions we held offline and online, and their feedback at conferences and other academic events. I am grateful for the training, networking and participation events of the Dependable Internet of Things Applications (DITA) program. Many thanks to Shabnam Nikfar for coordinating the DITA training sessions and student conferences. Lastly, I would like to acknowledge students from across Canada with whom I shared joyful moments during long conference days: Andrés Paz, Cristiano Politowski, Fabio Petrillo, Hao Jiang, Manel Grichi, Marios-Stavros Grigoriou, Mohab Aly, Mouna Abidi, Zeinab kermansaravi, and many others. I cherish these memories and thank you all for being a part of my student life in Canada.

The completion of this thesis would not have been possible without the unparalleled support and nurturing of Juanita. Thank you, for always believing in me. Thank you for lifting me up when the going was rough. Your company through this journey kept me strong. I am deeply indebted to you.

I am lucky to have a group of wonderful friends. I want to thank you, Lorena, Carlos, David, Geo, Maryi, and Felipe, for the best moments I had in Victoria. I hold dear every memory of us spending time, and wish we can repeat them sometime. I cannot think of my friends without mentioning Jessica and Fabián. From a distance, you accompanied me through every step of this journey, and helped me stay focused. I treasure your friendship, and very much appreciate our long-distance calls, our plans, and everything. To my friends in Cali, Diego, Cristiam, Jeffer, Alexis and Danny, I thank you for the good times, and for always keeping an eye on me.

I end this section by talking about Colombia, where the people I love and miss the most reside: my family. Their support has been unconditional all these years. They have been by

my side from a distance, each in their own unique ways. I thank them in Spanish now.

¡Gracias a todos! A mi abuelita Marta, quien se nos fue mientras hacía el doctorado, le agradezco por enseñarme el valor de la humildad. A mi mamá y a mi papá, Amelia y Nelson, les agradezco el apoyo incondicional y la confianza que desde siempre han depositado en mi. A mi hermano Andrés le agradezco por darme motivos para terminar. A mi padrino Fredy le agradezco las llamadas a larga distancia, siempre cargadas de noticias y hechos científicos; me las he gozado de principio a fin. A mi tía Teresa le agradezco por acompañarme desde Francia; su compañía ha sido una guía para mis experiencias en Canadá. Siempre estaré agradecido con mi tío Helbert por haberme introducido al mundo de los sistemas; a pesar de nuestra distancia de un tiempo a esta parte, sus enseñanzas no caducan. A mis tías y a mis tíos les agradezco los buenos deseos y las bendiciones cada vez que hablamos.

Dedication

Para la próxima generación
mi sobrino, mis primas y mis primos
Espero que mi esfuerzo en esta tesis
les inspire a superarse todos los días
y a alcanzar sus metas

Part I

Motivation and Context

Chapter 1

Introduction

Quantum Computing (QC) is a new computing field that aims to perform calculations by manipulating quantum bits (i.e., qubits) through the use of quantum mechanical properties such as superposition, entanglement and interference [1]. Given the nature of these quantum operations, QC has the potential to surpass the capabilities of current classical computers when solving complex problems, including drug discovery, material design and financial optimization. The concept of a quantum computer was first introduced in the 1980s by Richard P. Feyman with the goal of developing a tool that could solve one of the most important problems in physics: simulating quantum mechanical systems [2]. During the following forty years scientists and engineers greatly advanced the field of quantum computing by inventing novel quantum algorithms [3], [4], [5] and building quantum computers capable of demonstrating an advantage in specific tasks when compared to their classical counterparts [6], [7].

On the quest of finding the most optimal way of building a quantum computer, multiple hardware implementations were developed, each with their own unique characteristics and potential advantages. These implementations referrer specifically to the method by which qubits are represented within the quantum processing unit (QPU). The three most common types of qubit implementations are: superconducting qubits, ion traps, photonic qubits and topological qubits. As the development of quantum computers continued, researchers began to explore different ways of executing quantum algorithms efficiently. This led to the emergence of two main QC paradigms: gate-based QC and quantum annealing. Both models of computing make use of quantum mechanical principles and have the potential to solve very complex problems, nevertheless they differ in the type of problems they can solve and in the way qubits are manipulated to perform quantum operations.

In present day there are still many technical challenges to overcome before quantum computers can be widely used. As a matter of fact the current state of the field has been characterized as the Noisy Intermediate-Scale Quantum (NISQ) era [8], which refers to the limitations in the applicability and functionality of contemporary quantum devices due to their small number of qubits and considerable amount of errors caused by noise. Generally, the execution of most quantum algorithms requires far better hardware resources than what is provided by NISQ devices. Nonetheless, the emerging field of hybrid quantum-classical computing offers a promising solution to these challenges. By making use of these limited quantum computers and incorporating classical resources into their execution, this approach effectively addresses some of the shortcomings of the quantum device allowing

the production of useful results. Hybrid approaches are the prime candidates for demonstrating quantum advantage for real-world applications, such as molecular simulation [9] and various instances of optimization [10].

The development of hybrid quantum-classical solutions that effectively make use of computing resources and are able to tackle wide-scale real-world problems is one of the main goals of Quantum Software Engineering (QSE). This developing area of research aims to use knowledge from classical software engineering to bootstrap the transition from theoretical QC to practical quantum applications. Many crucial aspects of implementing hybrid systems have begun to be addressed by QSE. These include the impact of QC on the classical software development lifecycle and the new stages to be considered for such applications [11], effective techniques for testing quantum software [12], and the evolution of classical development and operations practices to support quantum applications [13]. However, QSE is still in a very early stage, offering immense opportunities for innovative solutions and pivotal advancements. We believe that continued development of quantum software engineering solutions is crucial in order to fully realize the potential benefits of hybrid quantum-classical computing applications in the near future.

1.1. Motivation

Over the last ten years, a large number of QC use cases have been explored for solving intractable computational problems. These use cases are aimed at providing practical and efficient near-term solutions for industries such as finance, logistics, and manufacturing. For instance, hybrid quantum-classical algorithms have been used as a tool to find an optimal combination of assets to include in an investment portfolio over a given period, while considering market impact costs [14]. They have also been employed to optimize the routing of delivery vehicles over different time windows for various logistics scenarios [15]. Additionally, some QC approaches have shown to be an efficient option for modeling and distributing complex large-scale energy supply chains, which are crucial for a range of industrial and civil operations [16]. This type of projects, and the significant investment in research and development in quantum computing applications by leading companies, such as Bosch[1], Airbus[2], and BMW[3], has sparked a widespread interest in this technology. According to Zapata Computing's Annual Report on Quantum Computing Adoption[4], approximately 74% of enterprises have adopted or were planning to adopt QC in 2022, highlighting the growing recognition of its potential to transform a variety of industries.

With the imminent incorporation of quantum computing into most companies' technology stack, it is only natural that hybrid quantum-classical software applications begin to be developed in-house to tackle industry-specific use cases. However there are multiple limitations within the field that make the implementation of quantum software solutions

[1] https://www.bosch.com/stories/future-of-quantum-computing/
[2] https://www.airbus.com/en/innovation/disruptive-concepts/quantum-technologies
[3] https://challenge.quantum.bmw.cloud/
[4] https://www.zapatacomputing.com/enterprise-quantum-adoption-2022/

challenging for early adopters. One of the primary factors contributing to this situation is that QSE is a relatively new field of research, having only been officially established in 2020 with the publication of The Talavera Manifesto for Quantum Software Engineering and Programming [17]. This seminal work provides a collection of principles, commitments, and calls to action for advancing the field and addressing the challenges faced by quantum software developers. Given that only a few years have passed since, Quantum Software Engineering hasn't had enough time to develop well-defined standards, good practices, and guidelines for the development of quantum software applications. Access to this set of tools is essential to help breach the gap between non-quantum developers and the rapidly expanding array of quantum computing resources, including, libraries, frameworks, development kits, algorithms, and hardware providers. Building on these insights, this thesis works as a guide on how to develop hybrid quantum-classical applications that successfully integrate quantum computing into traditional software systems. More specifically, this work presents guidelines and a flexible structure on how to address the implementation of hybrid software solutions.

1.2. Problem Definition and Research Questions

It is possible to draw a parallel between the current state of quantum software development and the software engineering crisis back in the 60s. As quantum computers become larger and more capable to solve harder problems, the complexity of creating software programs that efficiently use these resources increases. Current practices that work for small quantum solutions and proofs of concepts are not scalable for more intricate and complex quantum software applications with strict functional and non-functional requirements. A well defined methodology for the development of hybrid quantum-classical applications is needed for the advancement of the field and the wide-spread adoption of QC in the software industry. Our work constitutes the first steps towards achieving this ambitious goal. This research aims to answer the following questions:

- Q1: How can different computationally challenging problems be formulated in order to be solved using quantum computing?
- Q2: What are the benefits and limitations of different quantum computing providers?
- Q3: How can classical software engineering practices be applied to design and develop hybrid quantum-classical applications?

1.3. Contributions

The following are the contributions of this thesis.

- C1: Three different QUBO formulations for hard computational problems commonly found in the industry.

C2: A review of benefits and limitations of different quantum computing providers.

C3: A library for the seamless execution of optimization problems on different quantum computing hardware.

C3: A comprehensive guide for implementing hybrid quantum-classical applications.

1.4. Research Methodology

In order to answer the research questions proposed in this thesis, we aim to develop three different hybrid quantum-classical software solutions for three different use-cases in key application areas of QC (i.e. simulation, optimization and machine learning). We go through the main stages of building a software application, starting from the problem analysis and formulation, moving to the selection of the most appropriate technology and finalizing with the implementation of the solution.

To begin this work we perform an assessment of the three aforementioned impact fields of QC and describe one computationally complex, and relevant, use case for each. We analyze the chosen problems and proceed to realize formulations amenable to a quantum computer, taking into account variables, constraints and the overall objective. This stage helps us answer the first research question.

For the second research question we proceed to review and characterize different quantum computing providers and their respective software development kits (SDK) in order to understand the current state of quantum hardware. We consider three different QC vendors, IBM, D-wave and IonQ. With this selection we cover both QC paradigms (i.e., gate-based and annealing) and two different hardware implementations, superconducting and ion trap. The characterization is performed based on different aspects, including the size of their quantum computers, gate-fidelity, ease-of-use of their SDKs and the level of access to their devices. With this information we are able to clearly identify the benefits and limitations of each provider.

To answer the third research question, we take into account our findings and results from R1 and R2 in conjunction with tools and knowledge from classical software engineering to develop a hybrid quantum-classical solution that effectively integrates QC into classical software systems while addressing the proposed use cases. To accomplish this goal we identify clear requirements for these types of systems and proceed to design a solution that addresses all of them. Finally we proceed to implement our design and report on the trials and tribulations of developing such an application.

1.5. Thesis Outline

This chapter presented our motivation, outlined the research questions to be approached and highlighted the contributions of our work. The remaining chapters of this thesis are

organized as follows.

Chapter 2: formalizes key concepts of this research, describes the background and the state-of-the-art in the literature for this thesis.

Chapter 3: describes the key application areas for QC and presents the selected use cases.

Chapter 4: presents the findings of the review of three different QC providers.

Chapter 5: elucidates the requirements, design and implementation of the proposed solution.

Chapter 6: presents the trials and tribulations of implementing a hybrid software application.

Chapter 7: concludes with a summary of this research and discusses ideas for future work.

Chapter 2

Context and State-of-the-Art Background

Contents

The research problem we address in this dissertation intersects autonomic computing, continuous software engineering, and run-time software evolution. Hence, this chapter presents fundamental concepts from these areas, key research challenges and approaches, as related to the scope of this dissertation.

This chapter is organized as follows. In Section 2.1, we describe practices of continuous software engineering relevant for our research, namely Continuous Integration, Continuous Delivery, Continuous Experimentation, and DevOps. Section 2.2 introduces the fundamental concepts of autonomic computing and its foundational ideas. Moreover, we identify run-time processes of autonomic software systems and describe current approaches to integrate them into the Software Development Life Cycle (SDLC). Section 2.3 introduces self-adaptation and self-evolution, and contrast relevant definitions and approaches from the state of the art. Finally, Section 2.4 introduces Infrastructure-as-Code and its life cycle.

2.1. Continuous Software Engineering

Software-intensive industries have experienced a radical transformation in the way they deliver services [**?**], led by agile and continuous software engineering practices. Recent trends have helped organizations transform their culture and practices to deliver software more rapidly and with increased quality. Moreover, high levels of automation in the delivery process have enabled them to shift software development activities from development to execution. Therein, recent methodological and technological progress help software development teams break the rigid transition from development to production [**?**, **?**]. Consequently, companies have reduced time to market [**?**] and significantly boosted their productivity [**?**, **?**]. Nevertheless, discontinuities among business goals, software development and operations lead to challenges and open questions regarding the adoption of a continuous engineering mindset in the software industry [**?**]. Further research is required to facilitate the continuous evolution and maintenance of software systems and their corresponding architectures, prominently in the run-time context. A more holistic approach is necessary rather than one which is merely focused on continuous integration of software [**?**].

This section introduces key practices for contextualizing our work in this dissertation, namely continuous integration, delivery and experimentation. We describe these practices in the following sections as follows. Section 2.1.1 introduces the concept of Continuous Integration (CI). Section 2.1.2 describe the practice of continuous delivery. Section 2.1.3 contextualizes and describes the practice of continuous experimentation. Finally, Section 2.1.4 introduces DevOps by describing its main principles and adoption paths.

2.1.1. Continuous Integration

Continuous Integration (CI) is an agile software engineering practice that allows developers to frequently merge work to a shared mainline multiple times per day [**?**, **?**]. It includes frequent automated building and testing of the software in response to code modifications. A typical implementation of this practice includes a CI server that pulls code from a version control repository and executes interconnected steps to compile the code, run unit tests, check quality and build deployable artifacts. Even though automating the integration process is important for adoption, the relevance of CI lies in the frequency of integration. It has to be regular enough to provide quick feedback to developers, thereby improving their productivity and the software quality [**?**]. CI has the effect of producing shorter release cycles.

2.1.2. Continuous Delivery

Continuous delivery is a software engineering approach—aligned with the DevOps principles and the *deployment* adoption path—that promotes to deliver added value to end-users as soon and as frequently as possible, by deploying successful releases of a subject software

system [**?**]. The major benefits of this approach are the empowerment of teamwork between development and operations, the injection of fewer bugs (therefore reducing costs and risks), generation of less pre-release team stress, and a more flexible deployment process. To achieve these benefits, a software provider must promote a culture of collaboration between all teams involved in the delivery process, the sharing of knowledge and tools among participants, the establishment of measurement metrics, and the gathering of regular feedback for continuous improvement. That is, software providers must subscribe to DevOps principles to acquire continuous delivery benefits [**?**] and guarantee a repeatable and reliable process for releasing software, the automation of deployment and operation activities, the automation of integration, testing, and release processes, and the definition of an effective quality assurance process [**?**].

Although not every organization is able perform these processes as such, a similar concept to continuous delivery, popularized by Timothy Fitz,[1] named *Continuous Deployment*, has emerged as an alternative to constantly provide added value to end-users. The only difference between these two approaches lies in the fact that a software provider must be able to deploy every change that passes the corresponding automated test to production for enabling continuous deployment [**?**]. Both methods have now a widespread diffusion in the largest and most influential software-based companies in the world such as Amazon, Facebook, Flickr, Google, and Netflix, where new (atomic) features or bug fixes are deployed in short periods of time [**?**].

2.1.3. Continuous Experimentation

In the field of software engineering, experimentation refers to reducing software construction suppositions, assumptions, speculations and beliefs to concrete facts [**?**]. Experimentation leads organizations to gain understanding of quality properties of *good* software, as well as the process to make software well [**?**]. Software engineering is more than building software products. Engineers follow a design and development process to build products with quality. Thereby intellectual investigation to what produces the best software is desirable. Software engineering experimentation has been extensively used in empirical studies by researchers. Nevertheless, practitioners seem not to gain valuable outcomes, as there is and has been a lack of experimentation in the industry. The lack of conducting controlled experiments has been pointed out as one of the reasons of software engineering immaturity [**?**] (as cited by Juristo and Moreno [**?**]).

Systematic experimentation is a valuable technique for practitioners. They can gain insights about new methods, techniques or tools before introducing them into the organization [**?**]. When confronted with an array of options for building software, software engineers should seek proof that a particular option is better than another; they need reliable evidence and facts rather than assumptions [**?**].

Basili points out that experimentation in software engineering should be viewed from two perspectives, namely research and business [**?**]. In the first case, researchers need

[1] http://timothyfitz.com/2009/02/08/continuous-deployment/

industry-based facilities for understanding the software engineering processes and products, and build descriptive models accordingly. Nevertheless, Basili questions the direct benefits of such models for the software development teams. In the second case, software development teams need products and processes to help them build quality systems productively and profitably. Aligned with this perspective, a growing trend over the last decade consists of using experimentation to increase user satisfaction from a functional standpoint, while maximizing business profitability.

Organizations in software-intensive industries are moving away from value delivery based on guesswork to a more systematic and continuous approach [**?**]. In this systematic approach, customers and customer usage behavior play a central role in driving the development process and, therefore, customer satisfaction and revenue growth [**?**, **?**]. The continuous testing of business hypotheses and assumptions based on customer feedback is known as continuous experimentation [**?**, **?**]. This practice is based on the coupling between rapid development and an experimental framework to evaluate the impact of changes on usage behavior [**?**].

While so far mainly applied in the context of business decisions and regression testing, continuous experimentation can be used to validate assumptions and make more informed decisions in the evolution of software systems [**?**, **?**]. Recent research [**?**] shows that regression-driven experiments are more widespread than business-driven experiments. Obstacles, such as the lack of expertise and uncertain conditions, hamper the adoption of continuous experimentation practices, especially those associated with business concerns. We describe these two categories below.

Business-driven experiments aim to evaluate the business impact of software changes to guide the development of new features. Development teams formulate hypotheses about customer usage behavior in response to a software change. Then, the team defines the metrics needed to verify the hypotheses and instrument the associated software components [**?**]. This is continuously performed to learn how users react to the experiments and decide whether the changes should be deployed to all users or be abandoned. Fagerholm *et al.* point out that a suitable experimentation system requires the ability to release a minimum set of viable features with appropriate instrumentation, as well as capabilities to design and manage experiment plans [**?**].

Regression-driven experiments aim to identify and mitigate the impact of software regressions. As described by Schermann, this kind of experimentation corresponds to the application of DevOps' shift right concept [**?**]. This approach to experimentation validates the software quality through live testing techniques, such as A/B tests, dark launches, canary releases, and gradual rollouts [**?**].

Schermann *et al.* surveyed 187 software professionals and found two implementation techniques used in industry [**?**], namely feature toggles and run-time traffic routing. We describe each one of these implementation techniques below.

Feature toggles. According to Schermann *et al.*, 36% of the survey respondents realize feature experiments at the code level, creating multiple versions of the features in the code

base. This means that certain feature may be enabled or disabled for a given user or user group. This technique is also known as feature switches, feature flags, feature flippers, and conditional features.

Runtime traffic routing. Schermann *et al.* report that 30% of the survey respondents deploy and operate multiple service or component instances running the original feature and its variants. In this case, the control group is routed to the original instance while the treatment groups are routed to the variants. The routing can be done by using filtering criteria based on the users' request information or at the network level.

Several practices exist to deploy customer experiments, most importantly canary releases, dark launches, gradual rollouts and A/B testing. A *Canary release* allows to test the experimental code under real production conditions safely. The new release is exposed to a subset of users only, aiming at finding bugs on a small sample of the user population [**?**]. In a *Dark launch*, the pilot code is deployed to the production environment silently (*i.e.*, without being visible for any user) to test performance regressions using real user traffic. This is achieved by duplicating the traffic from the stable version of the application [**?**]. *Gradual rollout* is a technique to incrementally roll out the experimental code to production [**?**]. The new release is exposed to a percentage of the user base. This percentage increases over time until it reaches 100%. *A/B testing* is a technique to test two alternative implementations of the same software feature. In this case, two user groups access the two alternatives.

Significant work has been done in live testing and customer experiments. There are many research and industry opportunities in automation and tool support to facilitate planning, conducting and analyzing experiments. Nevertheless, Research on this topic is mainly focused on business- and regression-driven experiments. Other kinds of experiments could bring benefits to the development process itself, without looking at the business value proposition but the quality built into the software product. Specifically, software engineering experiments and exploratory activities could support non-functional decisions, promoting collaborative work between architects, developers and operators. This combination of experimentation and quality assurance has not been explored at large.

Mattos *et al.* identified architecture qualities and key research challenges to support automated experiments in software systems [**?**]. They propose an architectural framework for automated experimentation to coordinate experiment executions and system adaptations through monitor probes and effectors. This work, however, does not address experimenting with software patterns to create architecture variants, adaptations to the experimentation itself nor integration with delivery pipelines.

Gerostathopoulos *et al.* propose the realization of systematic experiments expressed using declarative specifications to allow developers confirm design assumptions [**?**]. In this approach, automated experiments are conducted by adjusting configuration properties of the system, avoiding the use of system replicas. The experimentation process is a closed loop that continually assesses the effectiveness of experiment executions by monitoring their effects on the system, and halts experiments when certain variable reaches

a threshold. This approach does not consider architecture variants of the system. Conversely, Porter *et al.* propose *TESS*, an experimentation testbed that facilitates evaluating self-healing and self-adaptive distributed systems [**?**]. *TESS* enables the automatic generation and deployment of random architecture variants by conducting different types of experiments (*e.g.*, self-healing of component and node failures, and self-adaptation experiments), in order to collect evaluation metrics. These measures are used to analyze the effectiveness of recovery and adaptation frameworks. Although this approach considers experimenting with architectural configurations, it is aimed at evaluating and testing recovery and adaptation frameworks only.

2.1.4. DevOps

There is a lack of consensus on the definition of DevOps [**?**]. Some researchers and practitioners refer to it as a paradigm, a method, or a set of principles and practices, whose main purpose is *closing the gap* between software development and Information Technology (IT) infrastructure operations. The truth is that DevOps emerged out of the disconnection between development and operations teams. As simply put by Bass *et al.*, after years of *siloed* software development and operation, a series of approaches have been flourishing between the two teams [**?**]. Therefore, much of the cultural and organizational changes associated with the adoption of DevOps relate to communication and collaboration. Nevertheless, adopting DevOps has technical and socio-technical implications [**?**, **?**]. For example, it may be necessary to re-architect an existing software system to apply software engineering practices to infrastructure code.

While no common definition of DevOps exists, IBM consolidated the main principles developed in the evolution of the DevOps movement [**?**]. Humble *et al.* played an influential role in advocating the practices supporting these principles [**?**, **?**, **?**, **?**]. These principles are: (i) develop and test against production-like systems, the main premise of the shift-left concept moving operations toward development; (ii) deploy with repeatable and reliable processes, for which automation is essential; (iii) monitor and validate operational quality, based on functional and non-functional software characteristics; (iv) amplify feedback loops, reacting and producing changes more rapidly. Similarly, IBM proposed four paths and respective foci of concerns for adopting DevOps: (i) *steer*: continuous business planning; (ii) *develop/test*: continuous integration and testing; (iii) *deploy*: continuous release and deployment; and (iv) *operate*: continuous monitoring [**?**].

Two prominent DevOps practices are shift left and shift right. They are usually applied in the context of testing, primarily on shift left testing. In this case, the focus is on testing as early as possible in the SDLC. In contrast, shift-right promotes testing software updates with live production traffic [**?**]. However, these practices apply in any context, and consist of moving concerns from development later to the operations side, and from operations earlier, toward development. The first principle described above, for example, stems from the shift-left concept. In spite of these concepts being beneficial for producing quality software products, there is a lack of attention on the topic, especially in areas outside testing.

Identified Challenges on Continuous Software Engineering

Shifting operations left by automating software experiments. Many organizations adopt DevOps from a traditional point of view, that is, focusing on deployment as a checkpoint going forward from development to operations—considering it as an end for that purpose. However, from a wider DevOps perspective, automated deployment is a crucial phase, for instance, to explore different design, configuration and deployment implementations in the operations setting, enabling the collection of data efficiently. This data, used backwards, is key to improving development artifacts, and in this process, deployment serves as a medium rather than as an end. Achieving DevOps requires to find ways of traversing development and operations processes in both directions, and the shift-left concept enforces especially its backward application.

2.2. Autonomic Computing

In 2001, IBM released a manifesto alerting the software engineering community about a looming software complexity crisis [**?**]. The motivation behind the manifesto was the increasing complexity of installing, configuring, tuning, and maintaining computing systems. Emerging computing systems at the time underwent an unprecedented success, thus growing beyond company and geographic boundaries into the internet [**?**]. Resulting levels of complexity became unmanageable, even by the most skilled IT professionals. In fact, the manifesto pointed out that managing such systems was well beyond the administration of individual software environments [**?**]. Declared initially by IBM's senior vice president of research Paul Horn, a consensus grew that self-management was the only viable alternative to cope with the anticipated crisis [**?**, **?**].

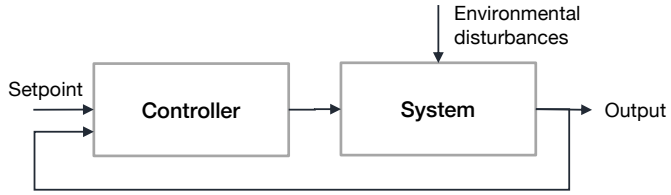
The complexity stemmed from factors internal and external to the software systems, causing uncertainties that were difficult to anticipate during design, development, configuration, and deployment. Key factors include the heterogeneity of the underlying communication and computing infrastructure, changes to the availability of resources, and continuously evolving requirements and environments [**?**]. The need for autonomic computing, then and now, is rooted in the ability of software systems to manage themselves autonomously. In this regard, autonomic computing is conceptually closer to automating the day-to-day operation of computing systems than it is to building "thinking machines" that embody the popular conception of artificial intelligence [**?**].

Self-management refers to computing systems that can adapt autonomously to achieve their high-level goals. Such computing systems are usually called self-adaptive systems, and are closely related to, and are often even referred to as, other types of software systems. The most prominent are autonomic and self-managing systems. Many researchers, in fact, use these terms interchangeably [**?**]. In this regard, self-management capabilities pertain to any of these systems insofar their goals require it.

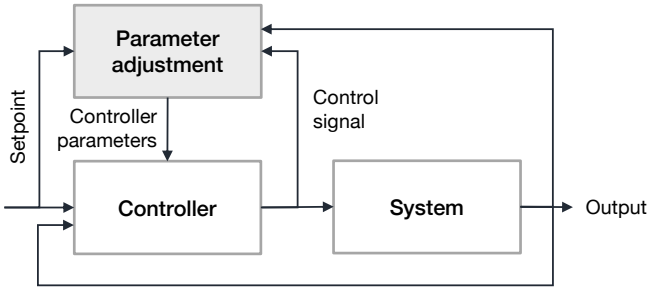
The rest of this section introduces relevant concepts from autonomic computing. Section 2.2.1 describes control-based software adaptation, including feedback loops and adaptive control. Section 2.2.2 defines self-management, self-regulation and self-improvement, as needed in this dissertation. Section 2.2.3 describes run-time processes present in prominent models for architecting self-adaptive software systems. Finally, Section 2.2.4 discusses approaches in autonomic computing that consider integrating self-adaptation processes into the SDLC.

2.2.1. Control-based Software Adaptation

Feedback loops are a fundamental concept of control theory [**?**]. They automate system control by analyzing the behavior of the controlled system and continuously producing corrective actions. Their main objective is to ensure the adherence of the system to expected or anticipated behavior. Consequently, feedback loops manage uncertainty, which allows the controlled system to deal with environmental disturbances [**?**, **?**]. As depicted in Figure 2.1a, feedback loops are closed loops in which control actions adjust the system's inputs to achieve a desired output (*cf.* setpoint in Figure 2.1a). The controller's job is to compare reference inputs and measured outputs to decide on the application of control actions to the controlled system.



(a) Block diagram of a control system
Adapted from [**?**].



(b) Block diagram of an adaptive control loop
Adapted from [**?**].

Figure 2.1.|: Feedback and control loops

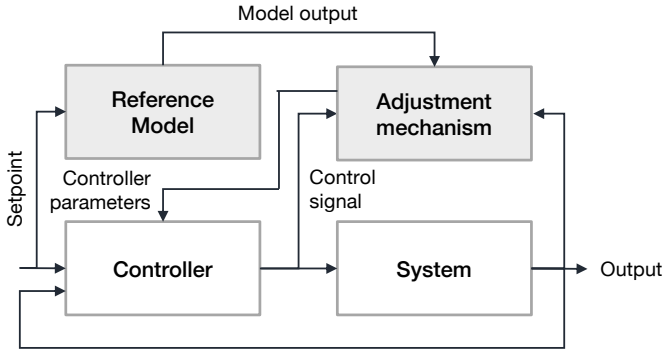
For systems with uncertain or varying parameters, additional control structures are necessary either to adjust the control actions according to run-time disturbances, or to compensate for possible inaccuracies in the initial system model [**?**]. This is known as adaptive

control, and refers to the ability of a system controller to adjust its working parameters over time. This is usually achieved through the definition of one or more supplementary feedback loops dedicated to parameter adjustment, providing flexibility to the controlling mechanism [**?**]. Adaptive control provides an additional mechanism for managing uncertainty, thus contributing to the resiliency of controlled systems (*e.g.*, Cyber-Physical Systems (CPSs)). Figure 2.1b depicts the structure of a classic adaptive control system where control operations performed over a controlled system are synthesized from parameter adjustments. Adaptive control mechanisms have been implemented successfully in different domains, including industrial plants, flight control systems, ship steering, and automobile control [**?**]. There exist several reference models for adaptive control. We concentrate on three of them, namely, Model Reference Adaptive Controller (MRAC), Model Identification Adaptive Controller (MIAC), and Multi-Model Adaptive Control (MMAC). We describe these models below.

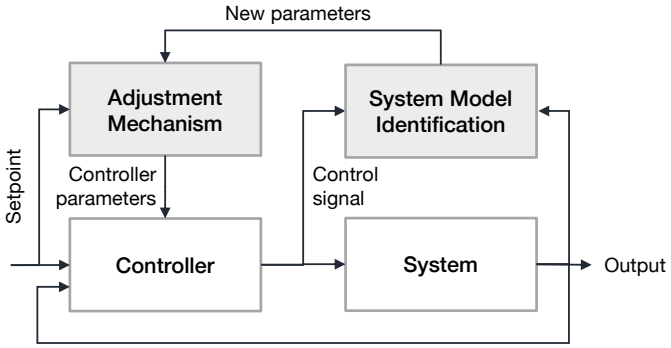
Model Reference Adaptive Controller (MRAC) is also known as Model Reference Adaptive System (MRAS). It refers to the use of a reference model, established in advance, to adjust the way a controller defines proper control actions to achieve the desired output on the controlled system. In this regard, anticipated—or predicted—behavior delineates tolerance intervals for the dynamics of the controlled system, and adjustment actions take place when discrepancies occur. Figure 2.2a depicts the main components of MRAC. The adjustment mechanism acts as a high-level controller over the feedback loop's controller. In this case, the controlled input refers to the control parameters, and the measured output is a control signal. The adjustment mechanism relies on a reference model, which is analogous to the controller's setpoint.

Model Identification Adaptive Controller (MIAC) establishes that control parameters are obtained from the identification or inference of a model characterizing the behavior of the controlled system. Run-time system identification methods allow the identification of this model. Figure 2.2b depicts the main components of MIAC. The convergence between the control signal (control actions) and the measured outputs enables the identification of a reference model. The usage of this outcome is twofold. First, it is used to determine how the adjustment mechanism calculates the control parameters. Second, it also determines how a controller directs a controlled system to achieve a desired output.

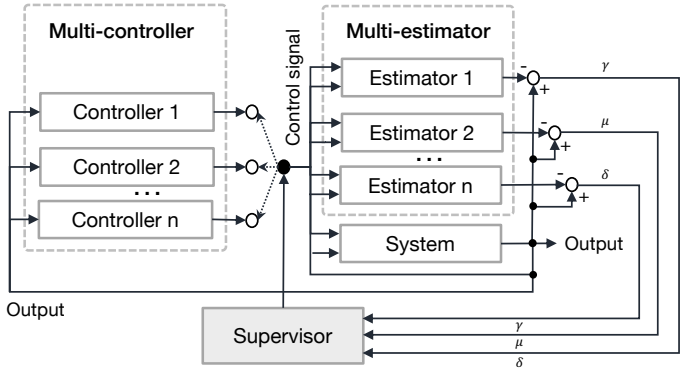
Multi-Model Adaptive Control (MMAC) combines multiple controllers aiming to increase the accuracy of the control actions. Figure 2.2c depicts the main components of MMAC with switching. MMAC defines four major blocks for realizing adaptive control. The multi-estimator block predicts (identifies) possible outputs of a controlled system using a set of estimators (models). Each of them uses the reference inputs and measured outputs of the controlled system to calculate the estimates. Moreover, each estimator is associated with a controller of the multi-controller block. Controllers stabilize and satisfy the required performance for estimators. Following predefined criteria, the supervisor block selects, via the switches, the estimator with the smallest error. Consequently, the control output applied to the controlled system at each instant is selected by the supervisor. The selection is based on switching logic and a function of the estimation error to indicate the best estimator at each time [**?**].



(a) Block diagram for MRAC
Adapted from [?].



(b) Block diagram for MIAC
Adapted from [?].



(c) Block diagram for MMAC
Adapted from [?].

Figure 2.2.|: Reference models for adaptive control

2.2.2. Self-Management, Self-Regulation, and Self-Improvement

During the past two decades, much effort has been devoted to instrumenting software systems with feedback loops. The objective is to provide software systems with improved intelligence and autonomy, aiming to reduce their operational complexity and increase the value offered to external systems and final users [**?**]. This goal has so far been predominantly researched in the context of self-management. A set of well-known properties of self-management, as introduced by IBM, are self-configuration, self-optimization, self-healing, and self-protection [**?**, **?**]. These properties are often referred to as self-* (*i.e.*, self-star), self-X, or adaptivity properties [**?**, **?**]. We describe them as follows, based on Kephart and Chess, Bantz *et al.*, and Salehie and Tahvildari.

Self-configuration refers to a system's capability to re-configure itself as a response to changes in its environment, or in its high-level goals. Self-configuration is conducted dynamically, during execution, by configuring, installing, updating, upgrading, integrating, composing and decomposing software elements seamlessly and automatically.

Self-optimization refers to the capability of managing software qualities and computing resources to improve their own performance and efficiency. It is also known as self-tuning or self-adjustment [**?**].

Self-healing refers to the capability of discovering, diagnosing, and reacting to unanticipated disruptions. Moreover, it also refers to detecting problems and counteracting errors, faults, and failures. It is related to self-diagnosing and self-repair [**?**], which focus on problem diagnosis and recovery, respectively.

Self-protection refers to the capability of detecting and defending against security threats and breaches, malicious attacks, and cascading failures automatically. Moreover, protection actions can concentrate on proactive measures to avoid security problems or mitigate them to prevent further damage.

Salehie and Tahvildari present a hierarchical view of the self-* properties composed of three levels. At the bottom of the hierarchy they place the *primitive* level, containing self-awareness, self-monitoring, self-situated, and context-awareness. This level generally refers to the system being aware of its self states and behaviors, as well as its context. In the middle, Salehie and Tahvildari place the *major* level, containing the self-* properties introduced by IBM. And at the top of the hierarchy, they place the *general* level, containing the properties of self-adaptiveness and self-organization. They define self-adaptiveness in terms of sub-properties self-management, self-governance, self-maintenance, and self-evaluation. They define self-organization as the capacity of a system to support emergent and decentralized functionality.

Insaurralde and Vassev approach self-regulation from a biological perspective [**?**]. In this sense, self-regulation helps achieving continuous adaptation to the environment by changing internal conditions. It is a physiological response of an organism to control its

internal state. According to Merriam-Webster's dictionary, self-regulation refers to controlling or supervising from within instead of by an external entity or authority.[2] In the case of a self-adaptive system, we define this property as the system's capability of keeping internal conditions in an appropriate state according to its environment. More specifically, internal conditions include state models, analytical and predictive functions, and other types of context-dependent knowledge artifacts. Appropriateness may vary by domain, but generally refers to stability, accuracy, fidelity, and properties alike.

Krupitzer *et al.* define self-improvement with respect to a self-adaptive system as *the adjustment of the adaptation logic to handle former unknown circumstances or changes in the environment or the managed resources* [**?**]. They define improvement in terms of self-adjustment of the adaptation logic, which generally refers to adaptive control from the perspective of control theory (*cf.* Fig. 2.1b). Similarly, from the perspective of autonomic computing, this definition fits well with hierarchical autonomic systems (*e.g.*, IBM's Autonomic Computing Reference Architecture (ACRA) [**?**]). Therefore, We define self-improvement from the perspective of continuous process improvement, commonly used in Lean philosophy as kaizen (*i.e.*, continuous and incremental improvement). Although self-optimization is closely related to self-improvement, by our definition, its application is commonly scoped to the run-time context (*i.e.*, as part of self-management) and involves performance factors as the primary focus. In contrast, self-improvement refers to any software quality and may be conducted at any time in the software life cycle.

2.2.3. Run-Time Processes of Self-Adaptive Software Systems

Various self-management feedback loop models have been proposed in the autonomic computing literature. Among them, there are approaches leaning toward the architecture, focusing on high-level components and their interconnections (*i.e.*, [**?**, **?**, **?**]). Other models tend to focus more on explicitly defining phased cycles, clearly delineating functions present in the feedback loops (*i.e.*, [**?**, **?**]). In both cases, there are common activities shared across the models, namely measurement, requirements management, analysis, change management, and evolution.

Oreizy *et al.* [**?**] presented two management cycles, one for ephemeral adaptations and another one for persistent evolution actions. The adaptation management cycle plans and deploys architectural updates at run-time. The evolution management cycle maintains the consistency and integrity between a software architecture model and its implementation. These two cycles aim to realize a comprehensive methodology to bridge adaptation in the small to adaptation in the large.

Kephart and Chess introduced the Monitor-Analyzer-Planner-Executor–Knowledge (MAPE-K) loop for monitoring and analyzing a managed system's context, and then planning and executing actions to counteract potential disturbances [**?**]. The combination of monitoring, analysis, planning, execution, and knowledge management in this loop is usually known as an autonomic manager (*cf.* Figure 2.3). The MAPE-K loop has played

[2] https://www.merriam-webster.com/dictionary/self-regulation

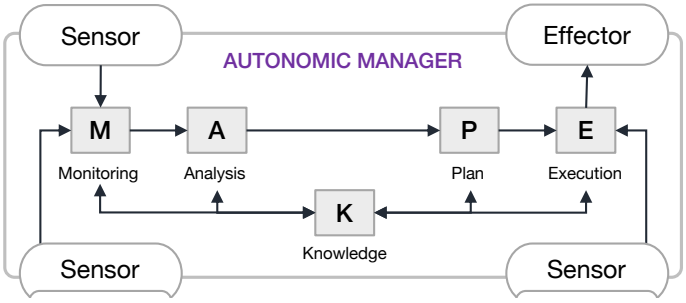


Figure 2.3.|: The MAPE-K loop
Adapted from [**?**].

a significant role in providing software systems with improved intelligence and autonomy. IBM later refined its feedback loop and introduced ACRA, a reference architecture for autonomic computing. It features the four elements composing an autonomic element, including standard interfaces Sensor and Effector as interaction touch points between the managing and managed systems. However, in ACRA, autonomic managers may be managed as well, thus facilitating building hierarchical autonomic systems. Similar to the MAPE-K elements, Dobson *et al.* [**?**] defined an autonomic control loop composed of activities Collect, Analyze, Decide and Act.

Kramer and Magee [**?**] proposed a three-layer reference model to self-managed systems. The bottom most layer corresponds to a component control layer, which includes an architectural model similar to that of Oreizy *et al.*'s work. The control layer provides operations over such a model to create, delete, bind, and unbind components at run-time, as well as set properties of their inner state. The middle and top layers are responsible for change and goal management, respectively.

Villegas *et al.* [**?**] also presented a three-layer reference model for governing control objectives (*i.e.*, adaptation goals) and context relevance in self-adaptive software systems. The top most layer monitors the reference control objectives and synthesizes context inputs to adapt the monitoring infrastructure eventually. The middle layer is based on IBM's MAPE-K elements and represents the application adaptation feedback loop. And the bottom most layer monitors the system's context to guarantee that the monitoring infrastructure remains relevant in the face of context changes.

All the models above rely on run-time representations of the managed system for various reasons. These representations are Models at Run-Time (MARTs)[3]. That is, abstract constructs intended to represent up-to-date information about a system's structure, behavior or goals, at run-time. MARTs can be causally connected with the system they represent, allowing run-time modification of a managed system through its models [**?**]. They have played a significant role in research advances toward self-adaptive software systems by enabling run-time reasoning of domain semantics, system architectures, goals, requirements, software components, context, source code, among many other concerns [**?**].

[3]In the literature often referred to as models@run.time [**?**, **?**]

We now describe further details on how the aforementioned models define each of the identified run-time processes.

Measurement Oreizy *et al.*'s management cycles define two activities for collecting and monitoring observations from the managed system. IBM's reference architecture follows the same approach and opts for a similar nomenclature, namely sensing and monitoring. Dobson *et al.* and Kramer and Magee mention the use of system and environmental sensors. Villegas *et al.* present a layered approach based on IBM's architecture, therefore their model uses sensors and monitors as well. However, they make emphasis on dynamically monitoring control objectives, the target system and its context.

Requirements management Although the models do not explicitly account for managing requirements at run-time, their run-time representation is fundamental to identify adaptation symptoms and guide the change management activity. Because of this, there is a clear need for specifying control requirements in one way or another at design-time. Oreizy *et al.*'s model requires the formulation of behavioral requirements or environmental assumptions. IBM's MAPE-K, Dobson *et al.*'s control loop and Kramer and Magee's architectural model rely on the specification of high-level adaptation goals. Villegas *et al.*'s DYNAMICO explicitly requires reference control objectives, which are specified based on adaptation properties of the managed system.

Analysis All the models evaluate whether the managed system satisfices the specified control requirements—or high-level goals. They analyze observations, rules and policies to determine whether an adaptation is necessary. Kramer and Magee's three-layer model analyzes state changes coming from the software components or the management goals with the same purpose. Villegas *et al.*'s DYNAMICO analyzes changes in the reference control objectives, their satisfaction, and the system's context.

Change management Similarly to the analysis activity, change management is rather uniform across all the models. The managing system is expected to synthesize an adaptation plan to adapt the managed system's behavior through its structure or operating parameters. A common strategy to realize this is to devise a set of adaptation tactics at design-time, defining criteria to select them at run-time. Out of these models, DYNAMICO performs additional changes to keep its monitoring infrastructure relevant to the system's context and the reference control objectives.

Evolution Although all the models follow a similar execution process of the synthesized adaptation plan, some perform additional tasks. Oreizy *et al.*'s management cycles include an activity to maintain the consistency between an architectural model and its implementation. This activity bridges the execution and development of the managed system. Dobson *et al.*'s control loop includes a final step to inform stakeholders about the enacted changes. This additional step also bridges the execution and development contexts, in this case for communication purposes.

None of the aforementioned models explicitly considers activities to validate or verify properties of the managing and managed systems during the adaptation cycle. Notwithstanding, Tamura *et al.* [**?**] extended IBM's MAPE-K model with run-time validation and

verification (V&V) tasks and enablers. They proposed to augment the Planner component with run-time validator and verifier elements to verify adaptation plans either before or after their execution. Additionally, they define V&V monitors for monitoring and enforcing the V&V tasks performed by the validator and verifier elements. Tamura *et al.* consider models at run-time as enablers for V&V. They highlight the relevance of having requirement specifications at run-time for meeting control objectives and realizing dynamic context monitoring.

2.2.4. Autonomic Computing in Software Development

Most works on self-adaptive and self-managing systems focus on run-time aspects only (*e.g.*, [**?**, **?**, **?**, **?**, **?**, **?**]). Therefore, self-management capabilities have been traditionally relegated to the production environment. However, some researchers have proposed various considerations to integrate the offline and online parts of the SDLC. A smooth transition between these two sides demands the integration and coordination of corresponding activities from both sides.

Qureshi and Perini argue that design-time uncertainty can only be addressed through Requirements Engineering (RE) at run-time, and differentiate between design-time and run-time activities [**?**]. On the one hand, at design-time, stakeholders conduct elicitation and analysis activities to produce a goal-oriented specification containing goals, soft goals, preferences, as well as monitoring specifications, evaluation criteria and adaptation actions. On other hand, at run-time, the system uses such a specification to guide the adaptation process. The differentiation proposed by Qureshi and Perini effectively realizes a RE process on each side (*i.e.*, design-time and run-time) connected through software artifacts.

Sawyer *et al.* take a similar position to that of Qureshi and Perini, arguing that distinct operational contexts may demand different requirements trade-offs [**?**]. They propose the use of run-time models to represent requirements at run-time, thus enabling managing systems to reason about them.

Ghezzi *et al.* identifies strong similarities between the automated process of self-adaptation, and the existing semi-manual process of Change Management of Information Technology Infrastructure Library (ITIL) [**?**]. They even analyze the potential for automation, ultimately, focusing on the design and enactment of change at design-time and run-time. Furthermore, Ghezzi *et al.* pose that supporting online software evolution and reconciling it with dependability require revisiting the entire software development process in terms of methods, techniques, languages, and tools.

In line with Ghezzi *et al.*, Gacek *et al.* and Andersson *et al.* address the offline and online integration beyond RE and analyze the relationship between design-time and run-time evolution processes [**?**, **?**]. Gacek *et al.* discuss the co-existence of self-adaptation and traditional change management. However, they do not discuss the implications on software engineering processes. Andersson *et al.* argue that responsibilities for development activities

shift from stakeholders to the system, thus causing the traditional boundary between development and execution to blur; an idea also proposed by Baresi and Ghezzi [**?**]. Furthermore, Andersson *et al.* characterize the running system as a stakeholder with a specific role in the development process, as it actively affects software development and maintenance. They propose an integration between offline and online activities through interaction points to synchronize the activities themselves or corresponding software artifacts. Their contribution focuses on mapping offline activities with online activities, thereby making explicit how to engineer self-adaptation and self-management capabilities at design-time, as well as how to integrate them with the SDLC.

Goltz *et al.* provide insights into the need for co-evolution of software systems along with their environments [**?**]. They recognize the significance of integrating development, operation, adaptation, monitoring, and maintenance throughout software evolution. In a similar vein, Iftikhar and Weyns contemplate integrating self-adaptation with operation activities in the DevOps life cycle [**?**]. In their approach, development activities are triggered by unanticipated change stemming from the system operation, whereas self-adaption refers to anticipated change under run-time conditions.

2.3. Run-Time Software Evolution

Software evolution refers to the application of maintenance actions to a software system aiming to generate a new operational version of the system. These actions are expected to guarantee the system's functionalities and qualities according to changes in its requirements and environments [**?**, **?**]. Software systems that run on highly changing environments, or whose requirements are subject to change frequently, pose additional requirements on their maintenance and evolution [**?**]. They often need updates while they are running, without causing them to pause, or even to shut down. These requirements give rise to run-time software evolution.

Oreizy *et al.* conceive run-time software evolution as a way to support run-time change [**?**]. Their visionary work focuses on quality assurance during system execution, such as high system availability. In this context, they identify the need for providing guarantees about the consistency and correctness of run-time change in a systematic way. Andersson *et al.* classifies run-time software evolution as an online process, meaning that its activities are conducted by self-adaptive software systems [**?**]. This also means that some stakeholders' responsibilities on the offline side shift toward the execution, thereby connecting the two sides of software evolution. Similarly, Müller and Villegas differentiate between offline and run-time software evolution [**?**]. They define the former as the process of modifying a software system relying on intensive human intervention and the interruption of the system operation. In contrast, according to their definition, the latter is conducted while the system is running with minimum human intervention. This differentiation is compatible with Andersson *et al.*'s offline and online process reconceptualization, in which offline evolution is performed in tandem with run-time adaptation coordinately [**?**].

Even though Oreizy *et al.* consider changes at the implementation level, they view run-time software evolution as a complement to stakeholders' work. That is, systems provide themselves important updates during execution, while taking advantage of preconceived invariants for preserving run-time qualities. This vision is compatible with that of Bosch and Olsson, who advocate for a balance between autonomic and manual work [**?**]. More specifically, in their vision, development teams build the software functionality and set guardrails, while smart systems experiment and adjust their behavior autonomously. Gacek *et al.*'s vision aligns well too. They explore the integration between ITIL's Change Management process and self-adaptation [**?**]. They identify a need for integrating change management with run-time activities in terms of roles, responsibilities, metrics, artifacts, and outcomes. Andersson *et al.* go a step further and propose a more holistic process approach in which offline and online activities interact throughout the software life cycle [**?**], thus contributing to blur what has been traditionally a rigid boundary [**?**]. They, as well as Müller and Villegas [**?**], envision the planning of software engineering processes according to their associated cost. Evolution activities are accommodated on the offline or online side according to certain properties, such as the level of uncertainty and required change frequency. In this regard, run-time software evolution has implications beyond system execution, as it potentially produces long-lasting effects on the planning, development and operation of the software product. These effects are nonetheless planned and decided offline by stakeholders in the described approaches.

Oreizy *et al.* pioneered a comprehensive methodology to balance software evolution on the development and execution sides [**?**]. Their approach synchronizes run-time adaptations to an architecture model (*i.e.*, a MART) with its corresponding implementation. Therefore, it combines quality assurance concerns stemming from offline and online quality attributes (*e.g.*, maintainability and availability, respectively). Modern work on run-time software evolution intersects with advances in deployment, monitoring, and management technology, especially in the context of DevOps. Iftikhar and Weyns contemplate the integration of self-adaptation with operation activities in the DevOps feedback loop [**?**]. In their conception of software evolution, under dynamic conditions, development activities are triggered by unanticipated change stemming from system operation. Conversely, self-adaption refers to anticipated change under run-time conditions. Following the same line of argument, Goltz *et al.* provide high-level insights into the need for co-evolution of software systems along with their environments [**?**]. They recognize the significance of integrating development, operation, adaptation, monitoring, and maintenance throughout software evolution. Weyns *et al.* conceive the coordination of run-time adaptation and evolution as a key enabler for sustainable software systems resilient to changing surrounding conditions [**?**]. Their work extends Oreizy *et al.*'s architecture-based approach [**?**] by augmenting their evolution management cycles with various types of uncertainties.

So far in this section, we have described run-time software evolution as an autonomic process that manages software changes at run-time. However, such a process can also be described from a time perspective, including the time horizon and the time of change. Section 2.3.1 discusses the differences between self-adaptation and self-evolution. And Section 2.3.2 describes the time of change timeline and how it relates to run-time software

evolution.

2.3.1. Self-Adaptation and Self-Evolution

Run-time software evolution and adaptation are often used interchangeably (*e.g.*, [**?**, **?**, **?**]). However, there seems to be a consensus on their specific meaning when it is relevant to separate them. Gacek *et al.* establish the difference between these terms with respect to the system's goals [**?**]. They present an approach comprising two iteratively interacting and concentric cycles. The outermost cycle concentrates on evolution and is based on Kramer and Magee's goal management architecture layer [**?**]. The innermost cycle focuses on adaptation, and addresses control and change management. In this regard, Gacek *et al.*'s work is aligned with Oreizy *et al.*'s change management process for run-time evolution. Their assumption is that change management and self-adaptation will co-exist synergistically and will improve incrementally. Their position is that run-time evolution tackles uncertainty in the long term, in contrast to software adaptation's short-term focus. Weyns *et al.* defines run-time software evolution and adaptation similarly [**?**]. For them, adaptation refers to the ability of mitigating uncertainty to keep satisfying the system's goals, whereas evolution refers to the ability of accommodating uncertainty to handle goal changes. Contrary to consensus, for Baresi and Ghezzi, evolution refers to the offline process, while adaptation refers to the online process [**?**].

Mens *et al.* define software evolution's time horizon as the effort required to achieve a particular result (*e.g.*, short, medium or long-term effort) [**?**]. Although this definition does not directly apply to the duality of evolution and adaptation, it is useful for describing the expected time horizon of a maintenance task. In this regard, a software change is intended to be either short or long lasting. In other words, whether the change addresses a low-level control objective or a high-level governing goal, respectively. Weyns *et al.* approach the time horizon from a sustainability standpoint [**?**]. According to Weyns and Becker *et al.*, sustainability refers to the longevity of a software system, its infrastructure and their adequate evolution under changing environmental conditions [**?**]. Software evolution then applies to high-level goals, whereas adaptation refers to low-level control objectives. Therefore, evolution and adaptation entail distinct design considerations.

Since control objectives are subject to changing environmental conditions, software adaptations are generally ephemeral. It is enough that changes occur in the context of the system for it to change again. Not all contextual conditions change rapidly nonetheless. In this case, the effect of corresponding adaptation actions may last days, weeks or even months, and therefore can be considered persistent evolution actions. Based on this distinction, this dissertation defines self-adaptation and self-evolution as autonomic capabilities of a software system to conduct software evolution and adaption actions autonomously, as a response to emerging behavior, regardless of whether it was anticipated or not.

Since evolution actions can be conducted by a self-adaptive software system, and persisted on the development or the execution side, it is possible to define self-evolution further. From a software engineering perspective, self-evolution is the process of producing

a new operative version of a software product, with minimum human intervention. From a control theory perspective, self-evolution is the process of changing the underlying management rules or procedures governing a self-adaptive system, with minimum human intervention and while the system is running. The work by Oreizy *et al.* [**?**] fits well in the first perspective. Their management cycle synchronizes a run-time model of the software architecture with its implementation. Thus, it makes adaptation actions persistent through changes in the source code. The evolution process presented by Gacek *et al.* [**?**] fits well in the control theory perspective. Their outermost cycle realizes a goal management layer that adapts production rules in the innermost (adaptation) cycle.

Self-evolution and self-adaptation entail distinct processes, design considerations and quality concerns. The latter has been explored multiple times, as described in this section. However, the design and quality implications of self-evolution have been largely unexplored, especially with respect to the software engineering perspective.

2.3.2. The Time of Change Timeline

According to the taxonomy of software change proposed by Buckley *et al.*, software changes are subject to multiple themes and dimensions, regarding what changes are made, when, where and how [**?**]. They classify them into the following groups. System properties, such as availability and safety, indicate *what* is being changed. Temporal properties, such as time and frequency, refer to *when* changes are made. The object of change, such as an artifact, relates to *where* changes are made. Finally, change support, such as the degree of automation, refers to *how* changes are accomplished.

To define the time of change, Buckley *et al.* focused on the development life cycle from the programming language's perspective. They identified three categories based on when changes are incorporated into a software system, namely static, load-time, and dynamic. *Static* refers to changes added to the source code, thus requiring recompiling the software for them to become available. *Load-time* refers to changes that occur while software elements are being loaded into an executable environment. Finally, *dynamic* refers to changes introduced while the software system is running. In light of these categories, the resulting time of change comprises three stages, namely compile-time, load-time and run-time.

Andersson *et al.* build on the identified change times, broadening the timeline to include development-time and deployment-time [**?**]. Moreover, they argue that the timeline is missing a more fine-grained perspective to include self-adaptive systems' concerns. According to Andersson *et al.*, software changes can be enacted offline by stakeholders, or online by the software system [**?**]. These new categories arise from a software engineering process-oriented perspective. That is, what activities are carried out by whom and where, and how do they intertwine. They argue that such a reconceptualization of the time of change blurs the boundary between development and execution. Thus, it is necessary to reassess the software engineering process in light of said new timeline. Müller and Villegas

define equivalent categories offline and run-time to denote the degree of human intervention [**?**], but emphasize software evolution concerns, such as frequency of change and level of uncertainty, as opposed to the whole development life cycle.

Out of Buckley *et al.*'s dimensions and themes, the time and the object of change have particularly grown outdated over time. Some of the key factors causing this include the emergence and widespread adoption of DevOps [**?**, **?**, **?**], and the accelerated progress in software engineering automation [**?**]. Because of this, the proposed taxonomy falls short in describing when and what changes are made in modern development settings. Even when including Andersson *et al.*'s proposed times and offline and online categories, the timeline fails to include relevant times associated with computing environments other than development and production. This is because the development and execution duality of the software engineering life cycle has been the predominant paradigm in run-time software evolution (*e.g.*, [**?**, **?**, **?**]). In recent research on machine learning [**?**], automated testing and program repair can potentially introduce changes as part of the delivery pipeline. For example, SapFix and Sapienz, tested at scale by Facebook, are able to cover the entire repair life cycle, from designing the test cases to fixing and retesting detected bugs [**?**, **?**]. Moreover, new artifact life cycles have been introduced to software engineering. Therefore, new perspectives require consideration beyond the programming language's one studied by Buckley *et al.*, and the process' one proposed by Andersson *et al.*. Examples of this include containers [**?**] and machine learning models [**?**]. In both cases, new times need to be considered for quality attribute evaluation, such as security for containers, and transparency, fairness and explainability for machine learning models, as well as offline activities that are now being conducted online. Grounded in the aforementioned reasons, the time of change timeline ought to be reconciled under the light of advances in adaptive systems and deployment technologies.

2.4. Infrastructure-as-Code

Infrastructure-as-Code (IAC) is an approach to provisioning and managing dynamic infrastructures through machine-readable specifications [**?**, **?**]. It is also referred to as programmable infrastructure in reference to the application of practices and tools from software engineering to IT infrastructure management. Changes to the infrastructure are made in a structured way, by means of reliable and established processes. The main enabler for IAC has been the advent of cloud computing technology, such as virtualization, which allow the provisioning, configuration and management of computational resources [**?**]. The benefits of IAC include repeatability of creating and configuring execution environments, management automation, development agility and infrastructure scalability [**?**].

The state of the practice for testing IAC is based on static analysis and functional tests [**?**, **?**, **?**]. The former provides quick feedback on minor programming mistakes, such as syntax errors. The latter consists of deploying the infrastructure and executing unit, integration and system tests to determine if the deployed resources and their configuration

Identified Challenges on Run-Time Software Evolution

Linking development and run-time software concepts. Systematic approaches to maintain the correspondence between distinct views of a particular artifact are rarely used in practice (*e.g.*, design artifacts and source code [**?**]). However, software and its deployment evolve independently over time. The mapping between development and run-time artifacts is specified across various specification notations, with no explicit references among them, which makes it loose and overall implicit. Mens *et al.* refers to this challenge as supporting co-evolution of artifacts [**?**], and Goltz *et al.* as supporting co-evolution of software systems and their environment [**?**]. Therefore, mapping run-time changes with source code contributions is a challenging problem. From left to right, this has been successfully managed through the delivery pipeline. However, these mappings are non-existent from right to left.

Adaptivity and configuration management at run-time. The dynamic nature of the cloud computing paradigm enables architectural agility that allows applications to evolve along with application requirements dynamically [**?**]. Cloud native software requires mechanisms for seamlessly integrating new components and updating the ones already running as part of its normal operation. Furthermore, the flexibility of the cloud enables today's systems to update infrastructure resources at execution time. Regardless of the characteristics of these mechanisms, deployment and configuration specifications should remain updated with respect to the actual system deployment as it adapts. The challenge here is to depict the deployment and configuration evolution, not only by performing isolated architectural changes in the infrastructure but especially tracking and visualizing them in the specification effectively.

Continuous integration infrastructure for self-evolving systems. Self-management capabilities rarely cross the boundary to the development side. Evidence of this, in part, is denoted by the lack of attention to models at the code level [**?**]. Therefore, CI infrastructure is necessary to integrate software changes produced online, and to guarantee minimum levels of acceptable quality [**?**].

are adequate. Deploying and re-deploying the system and its infrastructure to sandbox environments is resource and time consuming nonetheless.

Multiple implementations of IaC tools have been widely adopted by development teams, especially those following DevOps practices. Among those, Terraform[4] is one of the most popular tools for infrastructure provisioning [**?**]. Terraform is an open source software tool that provides a consistent workflow to manage cloud services. It manages source specifications written in the HashiCorp Configuration Language (HCL), which allows the specification of resources in a declarative way. That is, the specification contains the desired state of the infrastructure as opposed to the procedures to deploy it. Each specification contains a collection of resources, such as Virtual Machines (VMs), according

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

to the target cloud. Although syntactic rules are the same regardless of the resource and cloud provider, resource names, identifiers, and attributes vary by provider. In fact, anyone can develop modules and make them publicly available to the open source community by publishing the module to the Terraform Registry.

```
1  variable "storage_mount_target_path" {
2    default = "/sharedfs"
3  }
4  provider "oci" {
5    tenancy_ocid     = var.tenancy_ocid
6    user_ocid        = var.user_ocid
7    fingerprint      = var.fingerprint
8    private_key_path = var.private_key_path
9    region           = var.region
10 }
11 data "oci_identity_availability_domains" "availability_domains" {
12   compartment_id = var.tenancy_ocid
13 }
14 resource "oci_file_storage_file_system" "file_system" {
15   availability_domain = var.availablity_domain_name
16   compartment_id      = oci_identity_compartment.Compartment.id
17   display_name        = "P-FL-FileSystem"
18 }
19 output "mount_target_ocid" {
20   value = oci_file_storage_mount_target.MountTarget.id
21 }
```

Listing 2.1: A Terraform template containing Oracle resources

Listing 2.1 depicts an example HCL template containing resource definitions for Oracle cloud, as follows. Variables (*cf.* Line 1-3) are input parameters. Lines 5-9 exemplify how variables are used to specify attribute values. Providers (*cf.* Lines 4-10) are collections of authentication and authorization data attributes. Requests to the cloud API are secured using this information. Data sources (*i.e.*, Lines 11-13) query information about existing resources, such as identifiers and attributes. Resources (*cf.* Lines 14-18) are elements from the target cloud that can be deployed through Terraform. In this case, the resource in the listing is a file system object (*i.e.*, non-volatile storage). Lastly, outputs are data that can be queried once the deployment finishes, such as IP addresses, configuration files, and data alike. Typically, outputs are only known to Terraform after the deployment successfully finishes.

Figure 2.4 depicts the IaC life cycle according to Terraform. DevOps practitioners develop Terraform templates and store them in a source code repository. Every time a new software change is pushed to the repository, a CI server triggers a set of jobs for functionally testing the template, as well as checking the quality statically. In this regard, IaC source code goes through the same integration process as application source code. The deployment of a Terraform template requires running at least three commands, namely init, plan, and apply. Init downloads required modules to setup the Terraform project. Plan queries the target cloud based on the specified resources' names to find out which resources already exist, and compare them with the desired infrastructure state. It then presents a

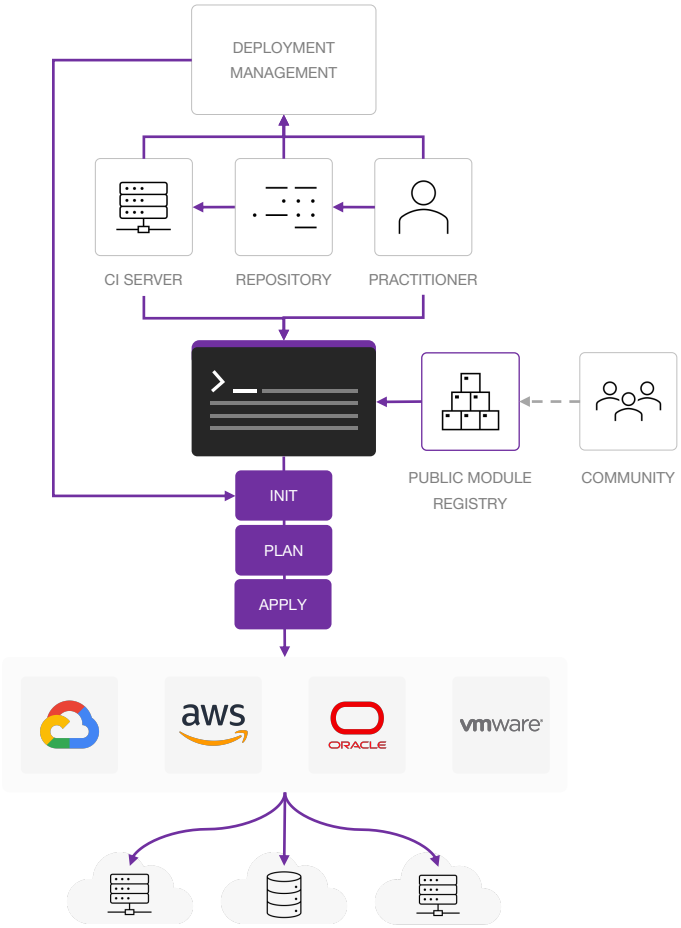


Figure 2.4.|: The Infrastructure-as-Code life cycle according to Terraform

deployment plan to the user, including actions to delete, update and create cloud resources. Finally, apply executes the deployment plan. In enterprise settings, it is common to manage Terraform templates through a deployment management software, such as IBM Cloud Automation Manager (CAM) and Terraform Cloud. The DevOps team publishes the templates, and makes them accessible throughout the organization. Other business units can then take advantage of pre-configured templates to deploy and manage their own instances independently from other teams, following security guidelines, compliance regulation, and governance policies setup transversely to all teams and departments.

2.5. Chapter Summary

In this chapter, we introduced foundational concepts of continuous software engineering, autonomic computing, run-time software evolution, and infrastructure as code. Furthermore, we have analyzed current concerns, approaches and key challenges with respect to the scope of our work in this dissertation.

We summarize below the most relevant challenges we identified, which correspond to the challenges addressed by this dissertation at different levels, as follows:

- In continuous software engineering:
  - The capacity of shifting operations left by automating the design and execution of software experiments, assisting stakeholders to make data-driven decisions, and providing autonomic managers with a foundation for contributing to the evolution of software artifacts on the development side.
- In run-time software evolution:
  - The linkage between development and run-time software concepts to explicitly connect run-time variability with software evolution on the development side.
  - The update of computing infrastructure deployment specifications as a direct response to adaptivity and configuration management at run-time.
  - The continuous integration infrastructure for enabling self-evolving systems to persist software changes on the development side, providing guarantees on their effectiveness and quality.

The next chapter provides an overview of our contributions. It identifies two remaining discontinuities in the SDLC, reassesses self-management capabilities, and proposes the use of self-evolution to tackle these discontinuities.

Part II

Contributions

Chapter 3

Contributions Overview

Contents

Correspondences in This Chapter

Addressed Research Question(s): Q1—How do autonomic managers fit into continuous software engineering practices, and corresponding computing environments, for the development phase and for long term evolution? Q2—How can self-improvement and self-management capabilities be integrated with quality assurance in continuous software engineering? Q4—How can development-time and run-time autonomic mechanisms be integrated reusing knowledge artifacts from both sides (*i.e.*, development and execution)? Q5—How can autonomic managers produce architectural and deployment variants? Q7—How can autonomic managers help reduce maintenance work stemming from emerging behavior?

3.1. Rethinking the Software Evolution Life Cycle

Recent software engineering practices are intended to reduce the impact of episodic and expensive activities during development. Development teams invest resources in automating their delivery processes and adapting their software services to take advantage of technological advances. A major factor motivating such an investment is removing discontinuities

from the development process. In spite of these efforts, two major discontinuities remain. On the one hand, run-time quality attributes are usually evaluated as part of the delivery pipeline, but the software design, configuration and deployment are rarely adjusted with the same frequency (*cf.* A in Figure 3.1). On the other hand, run-time changes and incidents on the operations side do no explicitly affect the evolution of development artifacts (*cf.* B in Figure 3.1). In both cases, the lack of continuity in the evolution process tends to cause disruption, either in the form of unsatisfied quality requirements or technical debt.

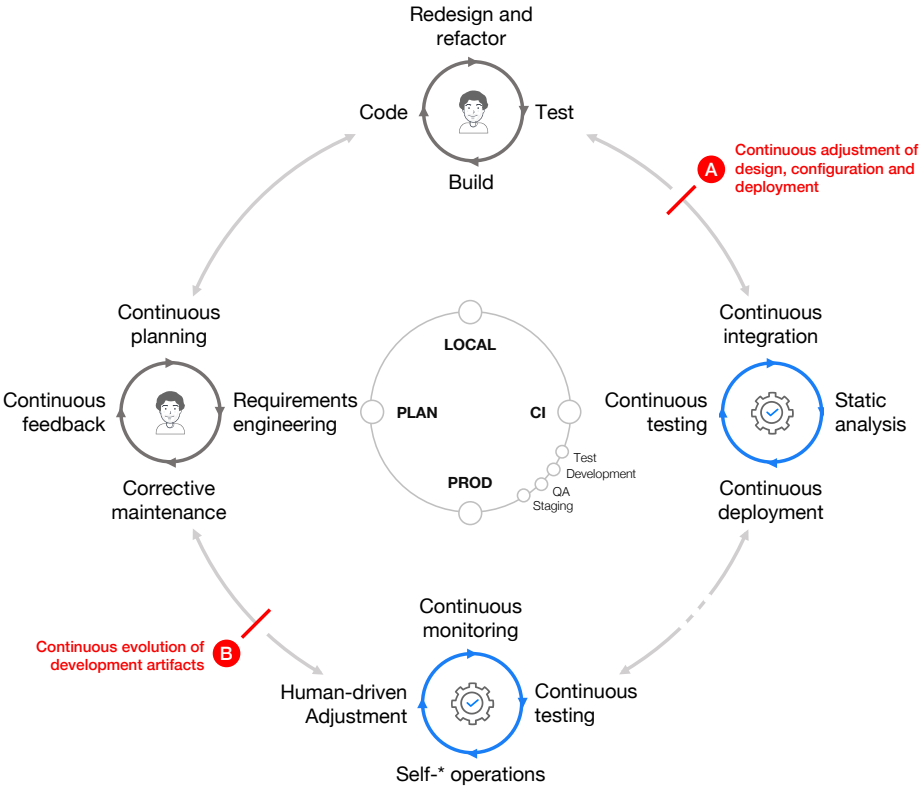


Figure 3.1.|: Discontinuities in the development process

From a software engineering perspective, software evolution is largely a product of human work. Indeed, software changes are mainly produced by stakeholders.[1] Developers introduce changes and integrate them into an automated delivery pipeline. However, regardless of its degree of automation, no stage of such a pipeline is explicitly concerned with introducing software changes into the source repository, for example to tune the software configuration. Its main role has been checking whether software products meet expected quality levels. One could argue that self-* capabilities do contribute to software evolution. In fact, they are usually considered a form of run-time software evolution [**?**]. Nevertheless, self-management actions rarely cross the boundary to the development side, hence, the discontinuities (*cf.* A and B). This is consistent with the lack of attention to changes at

[1]A recent practice consists of using software bots to free developers from tedious development tasks [**?**]. Among these, bots have proven useful for introducing minor software patches, such as upgrading outdated dependencies [**?**]. However, developing features, correcting defects, configuring and deploying software systems, among other major activities are still largely human endeavors.

the code level in the models-at-run-time community [**?**]. Thus, we identify an opportunity to integrate autonomic computing with continuous software engineering.

In light of the aforementioned opportunity, we reassess the time of change timeline and repurpose the delivery pipeline. We extend the pipeline's role to produce meaningful software changes. Section 3.1.1 describes the concrete updates we make to the time of change timeline. And Section 3.1.2 explains how we rethink the role of self-management to automate software evolution during development.

3.1.1. Reconceptualizing The Time of Change Timeline

Figure 3.2 depicts our offline and online reconceptualization. We split the timeline into offline and online based on whether the software evolution is human or automation driven (*cf.* A). Accordingly, we favor the use of offline and online over development-side and execution-side. Such a nomenclature is commonly used in reference to pre-production and production environments (*cf.* E), respectively. Nevertheless, as depicted in Figure 3.2 (*cf.* B), the production environment is no longer exclusively used for execution. The delivery pipeline requires executing the software system in non-production environments to perform various types of assessments. Therefore, neither development nor execution are exclusively bound to a particular environment.

In addition to traditional times of change (*cf.* C), we propose evolution-time to characterize when the system undergoes software changes. On the offline side of the timeline, this would typically correspond to the *coding* activity. On the online side, however, it corresponds to autonomic processes (*i.e.*, self-evolution jobs). We deliberately locate evolution-time in the `development` environment for two reasons: First, it is stable enough to avoid the waste of computational resources; And second, it is not as critical as subsequent environments for introducing software changes automatically. Previous environments `integration` and `test` typically run automated jobs for every commit pushed to the source repository. Thus, running self-evolution jobs in these environments would mean running time-consuming processes too frequently. Furthermore, changes between consecutive commits may not be significant enough to justify such use of the computational resources. Thus, self-evolution jobs can be run in the development environment, after having released an alpha or beta version,[2] and before performing quality assurance (*i.e.*, before the QA environment in Figure 3.2).

Evolution-time takes place at run-time as well. After all, it is during execution where new usage patterns emerge. Some form of self-evolution is necessary at run-time to ensure that the system is always operating based on relevant models, policies and assumptions.

[2]The software release life cycle varies by development team. Generally, the cycle starts with a pre-alpha release in the integration environment, through alpha, beta, and release candidate, to a stable version in staging. One example of such a life cycle, excluding the pre-alpha version, is Mozilla Firefox, which provides a Nightly release (*i.e.*, alpha), Aurora release (*i.e.*, beta), Beta candidate (*i.e.*, release candidate), and Main release (*i.e.*, stable) [**?**].

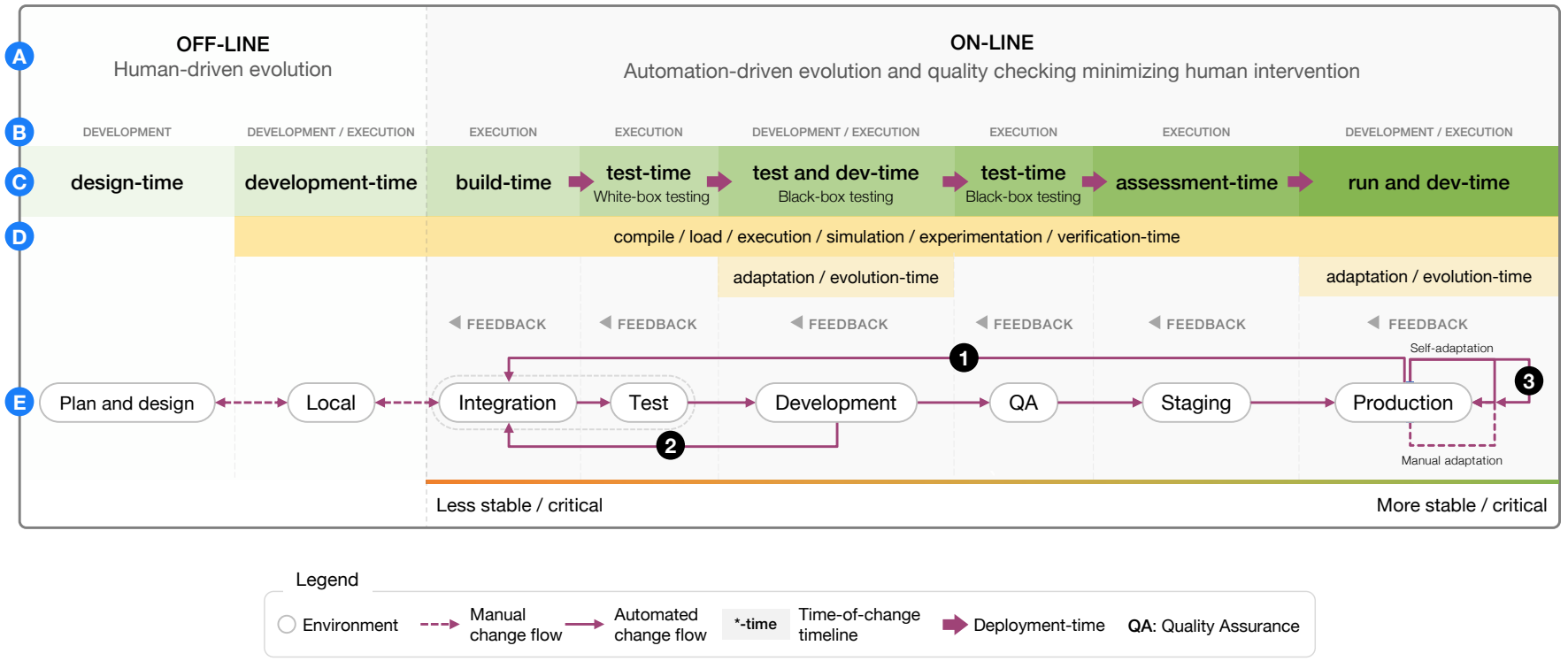


Figure 3.2.|: Offline and online reconceptualization

Nevertheless, evolving the system at run-time, without any kind of prior work (*e.g.*, exploration, analysis, reasoning and comparison of configuration alternatives) may be counterproductive. Consider self-tuning as an example. Sudden increases in service demand can rapidly change and behave erratically. If the execution conditions change often, it is likely that autonomic managers spend their time chasing new types of workloads without finding appropriate configurations timely. If more computing resources are allocated to decrease the search time, the cost-effectiveness may be negatively impacted. If the workload changes before paying off the cost of the evolution time, the cost will accumulate over time, yielding no value overall at the end of the billing period.

If we consider the iterative nature of the software engineering process, it is not completely necessary to anticipate the system's operating conditions. Autonomic managers can collect and learn from production data over time. This means that the interaction between pre-production and production autonomic managers is analogous to the interaction between development and IT operations teams. As in traditional DevOps settings, autonomic managers from both sides have to work cooperatively rather than independently. Therefore, evolution-time does have a place at run-time, where it can complement evolution strategies discovered autonomically at development-time.

We propose assessment-time to characterize the manual and automated validation process performed before deploying a release candidate to production. The staging environment provides an appropriate level of stability to justify the time invested on additional assessment procedures. Furthermore, assessment-time could be potentially used to evaluate other aspects of the system that could not be evaluated otherwise through testing. Examples of such an evaluation include user acceptance checks and extra-functional assessment of machine learning models (*e.g.*, fairness and explainability). Since the staging environment is also known as *acceptance*, acceptance-time would be acceptable too. However we consider assessment-time more semantically accurate.

We also propose a list of more granular phases that we call checkpoints. Unlike most times of change, they usually occur more than once through the timeline (*cf.* D). Checkpoints may introduce software changes by assisting stakeholders in evaluating configuration alternatives (*e.g.*, experimentation-time); they may assess the system's quality by observing its behavior (*e.g.*, verification-time); or they may predict Key Performance Indicators (KPIs) by running the system in a controlled environment (*e.g.*, simulation-time). These checkpoints enable new opportunities for exploiting autonomic computing during development. First and foremost, self-management capabilities are no longer exclusively applied at run-time, on the production environment. This means that their purpose can be adapted to the needs of other environments from the delivery pipeline. Therefore, offline activities that could not be automated for production can now be accommodated somewhere else (*e.g.*, experimentation). Second, autonomic managers can conduct time-consuming activities on pre-production environments to reduce processing time at run-time. Finally, these checkpoints enable further integration between autonomic computing and continuous software engineering. Efforts can be dedicated to reduce remaining discontinuities by shifting offline activities to the online side, thus contributing to the continuity of the software evolution process.

3.1.2. Repurposing Self-Management With Respect to Software Evolution

We propose three ways in which self-management capabilities can be repurposed to automate software evolution. The three added purposes emerge from how we conceive self-evolution (*cf.* Section 2.3.1). That is, evolving a software system from a software engineering perspective, as well as from a control theory perspective. The first proposed way is intended to help reduce operative development work caused by run-time variability. The new purpose consists of producing various types of persistent updates to development artifacts based on run-time changes (*cf.* ❶ in Figure 3.2). A persistent update refers to a software change intended to create a new operative version of a software artifact. Examples of persistent updates include updating a deployment specification in a source control system, and updating a template instance in a deployment management software. Since run-time variability increases the technical debt (*e.g.*, configuration drift, *snowflake* servers and infrastructure erosion [**?**]), persistent updates can be seen as technical debt payments. The second proposed way is intended to improve development artifacts in various ways actively (*cf.* ❷ in Figure 3.2). Through experimentation and optimization techniques, self-management capabilities can be used to explore design, deployment, and configuration alternatives aiming to improve KPIs. Once an autonomic manager has found an improvement, it proposes the respective changes as persistent updates. And the third way is intended to update run-time artifacts to keep them relevant with respect to the system's operation context and long-term objectives—that is, self-regulation (*cf.* ❸ in Figure 3.2). In this case, the run-time updates are created in response to emergent behavior. For example, whenever the usage demand pattern changes for a particular software service, the demand model associated with the service must be updated. In this case, the demand model is a reference control input that should remain relevant for correctly controlling the system's operation.

We conceive self-regulation as an autonomic capability with two levels of control. The first level is concerned with short term adaptation goals, while the second one focuses on knowledge updates to prolong the relevance of the first one. Such a hierarchical control structure is based on adaptive control, more specifically on MIAC and MRAC, and the ACRA [**?**].

Figure 3.3 represents the relationship between self-management, self-improvement, and self-regulation in the context of the delivery pipeline. The relationships depicted in Figure 3.3 (*cf.* ❶, ❷ and ❸) represent each a form of self-evolution. We extend the delivery pipeline to support paths ❶ and ❷ for delivering persistent updates autonomically. Conceptually, this implies realizing the practice of two-way continuous delivery. From left to right (*i.e.*, Dev $\rightarrow$ Ops), the pipeline realizes continuous delivery as it is usually put in practice. And from right to left (*i.e.*, Dev $\leftarrow$ Ops), it delivers value to the development team. Technically, however, some changes are produced in the development environment (*cf.* ❷), not on the operations side. Nevertheless, these changes are the product of controlled execution scenarios that reflect realistic operation conditions. In other words, the software changes are enacted in production-like environments.

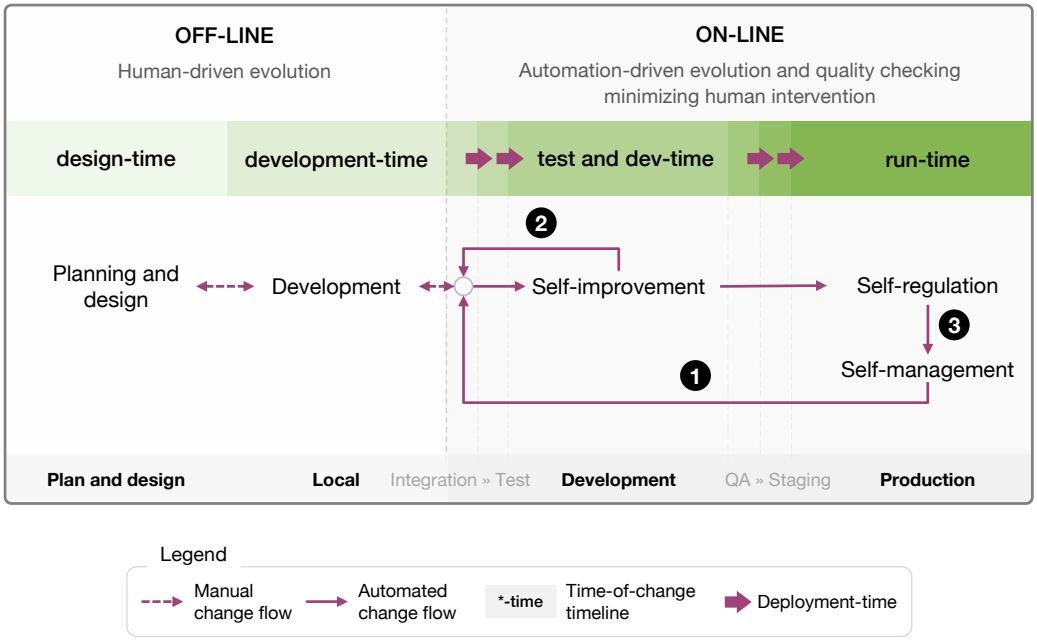


Figure 3.3.|: Self-evolution through self-improvement, self-regulation and self-management

Figure 3.4 illustrates the proposed evolution paths in the context of DevOps. On the left, we depict human- and automation-driven evolution as two engineering cycles rotating in opposite directions (*i.e.*, Dev $\rightarrow$ Ops and Dev $\leftarrow$ Ops). On the right, we depict how the two cycles integrate, producing a continuous process in terms of software evolution. By connecting the two cycles, we aim to reduce the discontinuities previously discussed (*cf.* A and B in Figure 3.1).

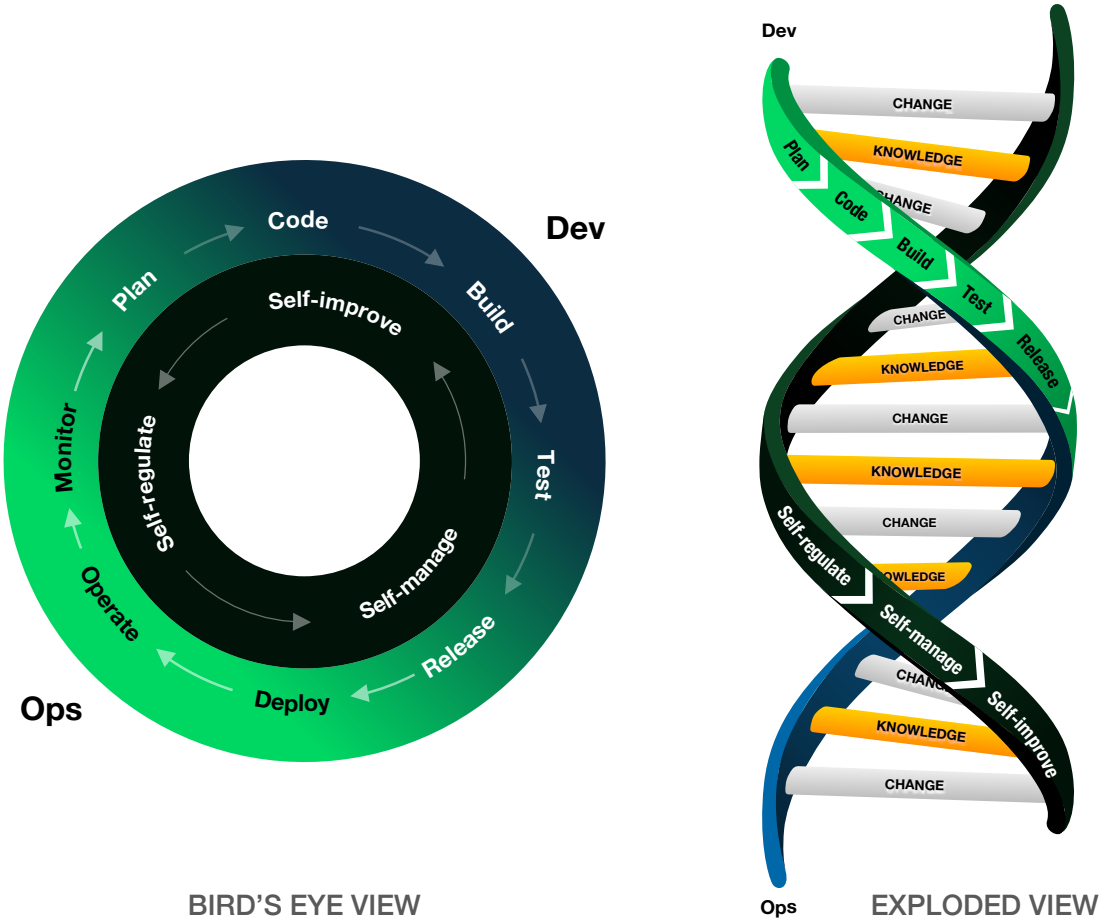


Figure 3.4.|: Autonomic and continuous software evolution process

3.2. Our Contributions

This section summarizes the four general contributions of this dissertation.

Framework for Continuous Software Evolution Pipelines. We propose a framework for continuous software evolution pipelines to bridge offline and online evolution processes. We take advantage of existing infrastructure and automation intended for software delivery, and put it into service of autonomic managers. Our pipeline extends the concept of CI to complete the evolution loop (*i.e.*, Dev ← Ops). That is, software changes stem not only from offline processes—driven by stakeholders, but also from online activities—driven by autonomic managers. Therefore, it enables feedback loops to capture run-time changes and integrate them *safely* into source specifications. This means that live modifications to source code, originated on the Ops side, go through existing testing procedures to guarantee minimum and acceptable levels of quality and confidence.

Our software evolution pipeline considers two evolution workflows: with and without quality evaluation support. In the first case, an autonomic manager adapts a MART that represents a specific aspect of the system, such as the Operational Service Plan (OSP) of a transportation system. As a result, the pipeline triggers a model transformation chain that ends up in the update of source specifications, or knowledge represented in models. This process is based on current software engineering practices, including, for example, using a code repository and following a branching model. In the second case, an autonomic manager adapts the system through the underlying computing platform. This time, since changes are applied immediately to the running system, changes to the source code specifications optionally skip quality control. By separating the run-time updates from the change enactment in the specifications, we aim at unifying the software evolution process. Thereby, our software evolution pipeline realizes a holistic and continuous process that connects software execution and development, while providing quality control to run-time changes. Furthermore, since run-time changes are integrated into source code repositories, run-time variability becomes traceable in existing processes and corresponding tools.

This contribution realizes self-evolution through self-management, as described in Section 3.1.2 (*cf.* ❶ in Figure 3.3).

Quality-driven Self-Improvement Feedback Loop. We propose online self-improvement as a way to address the rapid pace of software change. More specifically, we propose the use of a feedback loop for exploring design and configuration alternatives during development. The main purpose of our self-improvement feedback loop is to improve KPIs continuously before releasing a stable version of the software. That is, we propose taking advantage of non-production computing environments in the delivery pipeline. We connect our feedback loop with our continuous software evolution pipeline through various modeling layers, ranging from the physical computing infrastructure to the software application. This means that our feedback loop focuses on exploring configuration variants and finding possible improvements. Once the feedback loop identifies a variant that outperforms the baseline configuration, it delegates to the evolution pipeline the corresponding code updates.

Our self-improvement feedback loop relies on existing quality assessment procedures present in the delivery pipeline. By extending quality assessment with continuous improvement, our feedback loop frees stakeholders from maintenance work and expedites the implementation of incremental software changes. We follow the separation of concerns principle by splitting the online experiment management into three internal feedback loops. First, the Experimentation Feedback Loop (E-FL) guides high-level self-evolution while satisfying high-level goals through online experiment design. Second, the Provisioning Feedback Loop (P-FL) derives, deploys and monitors infrastructure configuration variants. Finally, the Configuration Feedback Loop (C-FL) derives, deploys and monitors architectural design variants.

Both the P-FL and the C-FL make use of combinatorial techniques to derive system variants. In the first case, the P-FL explores configuration parameters of declared virtual resources. In the second case, the C-FL explores design variants by applying domain-specific design patterns to the managed system's architecture. The C-FL relies on its capability to measure the impact of design patterns on KPIs.

This contribution realizes self-evolution through self-improvement, as described in Section 3.1.2 (*cf.* ❷ in Figure 3.3).

Run-Time Evolution Reference Architecture. We propose a reference architecture for guiding the design of dependable and resilient CPSs. The proposed architecture realizes a continuous engineering cycle where adaptation and evolution work cooperatively to achieve the system goals. Evolution plays both a reactive and a proactive role explicitly in this engineering cycle. Its main purpose is to regulate the reference models used for controlling the system's operation, thus, ultimately contributing to the managed system's long-term evolution. To do so, our architecture takes advantage of a hierarchy of autonomic control structures based on adaptive control and the ACRA. Moreover, it achieves the continuous cycle between short-term adaptation and long-term evolution by keeping up-to-date representations of relevant aspects of the system (*i.e.*, MARTs).

Our run-time evolution reference architecture realizes our envisioned engineering cycle as follows. First, it uses evolutionary optimization and online experimentation to find suitable models to guide the adaptation of the managed system. Second, it uses online experimentation, supported by parameter optimization, to gather evidence of statistically significant improvements over the system's baseline design, thus generating knowledge of possible configuration states and their respective performances. Lastly, our reference architecture accommodates self-evolution for reflecting persistent changes in the physical infrastructure of execution as well as improving the use of resources. Self-adaptation is used to ensure that the managed system operates within acceptable and viable boundaries. Together, these characteristics contribute to achieving dependability and ensuring resiliency for SCPSs at run-time.

This contribution realizes self-evolution through self-regulation, as described in Section 3.1.2 (*cf.* ❸ in Figure 3.3).

An Implementation for the Continuous IAC Evolution Pipeline. We provide a comprehensive implementation for the continuous evolution pipeline for IAC specifications. Our implementation covers the full life cycle of Terraform[3] (*i.e.*, an IAC tool) templates and instances as follows. First, it assists DevOps engineers in acquiring knowledge from an infrastructure deployed to a VMWare cloud. Our evolution pipeline creates the Terraform templates and stores them in a git repository. These templates represent the structural features of the deployment, including resource definitions and their interconnections with resources existing outside of the scope of the deployment (*e.g.*, data stores and networks). Furthermore, our implementation creates the corresponding template and template instance in IBM Cloud Automation Manager.[4] In this case, the template instance represents the actual parameter values used for the deployed resources (*e.g.*, The amount of storage associated with a VM). And second, our evolution pipeline prevents configuration drift between the Terraform templates, the IBM CAM instance, and the VMWare deployment by counteracting live deployment updates with changes on the development side continuously and incrementally. More details about this contribution are included in Appendix A.

Our implementation of the evolution pipeline integrates well with our implementation of the self-improvement feedback loop. Modifications to the deployment and configuration specifications are proposed as code contributions through our pipeline implementation.

3.3. Cloud Infrastructure Management Case Study

IAC is the practice of configuring system dependencies and provisioning local and remote compute instances automatically. One requirement of the IAC life cycle is that specifications need be the only source of change. Otherwise, IAC tools could not guarantee idempotency, at least without removing already deployed resources. That is, the deployment process could not be executed multiple times expecting the same result every time. Therefore, IAC tools can provide guarantees on the repeatability, convergence and predictability of the deployment process [**?**, **?**]. Modifying both the specification and the generated infrastructure leads to configuration inconsistencies, thereby increasing the technical debt[5] and possibly affecting these guarantees.[6] Nevertheless, computing infrastructures are subject to change during development and execution.

Hybrid cloud deployments illustrate the need for run-time modifications. Large organizations make extensive use of automation to conduct management operations, such as cost

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

[4] https://www.ibm.com/ca-en/marketplace/cognitive-automation

[5] In IAC, the best practice for modifying a deployed infrastructure consists of changing the source specification, thereby evaluating the changes throughout the delivery pipeline. In addition to this being a time-consuming process, changes are typically deployed according to a release schedule—unless they are critical bugs. In cases where the changes are required immediately, a system administrator may decide to bypass the delivery pipeline, applying the changes directly to the target infrastructure.

[6] IAC tools may guarantee these properties by destroying and recreating existing resources (*e.g.*, a virtual machine). Therefore, even if the resource is modified outside the IAC life cycle, the tool can guarantee the desired state. Often, this limitation comes from the cloud vendor, which may provide limited support for run-time updates.

reduction, load balancing, workload migration during maintenance periods, resource upgrades, and policy compliance enforcement. Cloud software management performs these updates live. Moreover, there is a growing concern in the enterprise world for *Day 2+ operations* [**?**]. This concept refers to the phase posterior to initial software deployment on *Day 1* into real-world execution conditions. On Day 2, and the days following, IT operations teams begin to stress software and infrastructure resilience, scale, data flow management, security, and governance [**?**]. Because of this, IT operations teams may use scripts developed in-house and third party software to manage the deployed infrastructure. Both cloud automation and Day 2+ operations are necessary for organizations to manage their hybrid cloud successfully. Nevertheless, they are both excluded from the IAC life cycle. Run-time modifications are an irreconcilable requirement that make the hybrid cloud incompatible with the current IAC life cycle.

3.3.1. Problem Definition

Even if IT operations teams decide to migrate their ad-hoc deployments to IAC, three problems emerge, namely: knowledge acquisition of deployed infrastructures, configuration drift, and continuous improvement.

3.3.1.1. Knowledge Acquisition

The team may have been managing their computing resources through scripts, third-party command-line applications and administration portals. The cost of such a migration can be significant. It can include hundreds or even thousands of resources. Furthermore, there is a constant trade-off between spending time migrating the infrastructure deployment and developing bug fixes and features. This is primarily true when the changes do not produce any direct benefit for the users. Moreover, manually writing the IAC specifications could result in many functional bugs, which may only become visible once the infrastructure is in production.

3.3.1.2. Configuration Drift

Deployed infrastructures will continue to be updated live for the aforementioned reasons. This causes an incremental drift between the IAC specifications and the deployed resources. Eventually, the discrepancy is such that specifications can no longer be used [**?**]. Since they contain outdated information, they will likely disrupt running services when redeployed. Thus, IAC specifications become obsolete over time, losing the initial investment in effort, time and money.

In an enterprise environment, IAC is usually managed through cloud management software, such as IBM CAM and Terraform Cloud. The IT department provides self-serve facilities to deploy pre-configured services. This means that other members of the organization tune and deploy the services they need, while the IT department focuses on maintaining

the IAC specifications, also referred to as templates. This separation between templates and instances makes the configuration drift problem even more challenging. This is because run-time changes can affect both, making it necessary to coordinate the evolution of independent instances and the template they share. From here onward, we refer to this pair of template-instances as specifications.

3.3.1.3. Continuous Improvement

DevOps engineers usually design and configure the initial computing infrastructure based on service demand assumptions and historical data. Once the software system goes live, they spend time adjusting the initial configuration based on the actual service demand (*i.e.*, Day 2+ operations). During this time, users of the software application often encounter errors and briefly experience limited functionality.[7] Reducing errors and risks during Day 2+ operations is critical to the business success nonetheless. However, IAC specifications are prone to errors as much as application code. With a 28% change ratio, IAC files are equally changed as other types of production files, even more so than software tests [**?**]. Since change of source code is a predictor of bug proneness [**?**, **?**] or risk [**?**], infrastructure code is vulnerable to similar issues [**?**]. Therefore, manual modification of IAC files may hinder the adjustment and improvement process.

Continuous improvement of IAC specifications is also regularly conducted during development. Automated performance testing is usually one of the quality checks in continuous delivery. Software development teams setup automated reporting to identify performance regressions or service degradation in general. The idea is to identify performance problems early in the development process, and correct them if possible before deploying the software to production. Ideally, these problems are solved by refactoring the software system, possibly impacting the overall design, configuration and deployment. Nevertheless, over-provisioning the computing infrastructure is seen as an alternative. However, because new code can add additional inefficiencies, a vicious cycle sets in (*i.e.*, over provisioning as a default response to new inefficiencies) while the cost of operation augments silently until it is too high, or adding more computing power stops being effective.

The problem that continuous improvement posits is twofold. On the one hand, development and operations teams ought to coordinate efforts for exploring configuration alternatives and adjusting the IAC specifications accordingly. On the other hand, this process should be conducted frequently to ensure meeting service level objectives in the face of changing execution conditions, while reducing error proneness in the adjustment process.

[7] This is an agile practice where the fail-and-fix-fast philosophy applies [**?**]

3.4. Smart Urban Transit System Case Study

In the context of Intelligent Transportation Systems, Smart Urban Transit Systems (SUTS) enable the optimization of public transportation services through the continuous exploitation of operational data from multiple context sources. SUTS make extensive use of Internet of Things (IoT) sensors and effectors to collect valuable data about the physical components of the system. Such data help characterize, predict, and control the system's behavior. Moreover, given the scale of SUTS, optimizing their operation is crucial to prevent negative effects on the city's overall transit. As the performance of routes affects not only the citizens who use them but also the mobility of the whole city from a global perspective, their effectiveness is crucial for the city's quality of life.

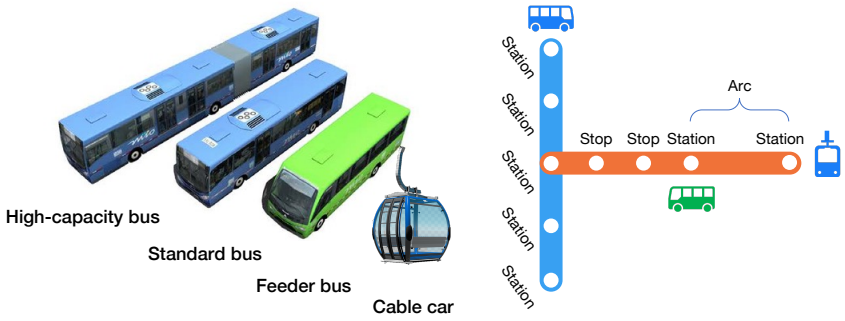


Figure 3.5.|: Conceptual model of SUTS case study

Figure 3.5 depicts common elements of a transportation system. Although there are various types of vehicles, we simplify the case study by considering them as buses. The elemental notion of a bus route is described by a set of *arcs* that connect stops—freestanding or as part of a station—and which buses traverse at predefined frequencies (*i.e.*, *headways*) to complete required trips. Upon arrival at a bus stop, a bus completes a *service time* defined by the time required for onboard passenger disembarkation and boarding of passengers. The latter is based on identified probability distributions describing passenger arrivals at the stop. There exist several measures of effectiveness that quantify and describe the performance of system routes from both a control and user (*e.g.*, headway design and waiting times, respectively) perspectives [**?**, **?**, **?**]. Two relevant measures are part of the KPIs of our SUTS case study: the Headway Coefficient of Variation (HCoV) and the Excess Waiting Time at Bus Stops (EWTaBS). The former captures the variability among observed headways from a route controller perspective [**?**, **?**]. The latter is a measure from the users' perspective that reflects the unjustified waiting time imposed by service irregularity [**?**, **?**]. As depicted in Figure 3.6, these measures of effectiveness can be used to describe the resiliency of SUTS in terms of its reliability (*i.e.*, low headway variability) and availability (*i.e.*, readiness of bus service when required) during changing conditions. The element hierarchy within the green tree associates target qualities of the system, namely dependability and resiliency, with system capabilities (*i.e.*, Self-evolution and self-adaptation) and metrics (*i.e.*, HCoV and EWTaBS). Darker elements in the figure represent the concepts addressed in the scope of this case study. This decomposition extends the one introduced by [**?**].

3.4.1. Problem Definition

Inevitably, the operational complexity of an intricate system such as our SUTS case study potentially exceeds the reasoning capabilities of human planners and operators. This results in far from acceptable system performance characterized by undesirable values of HCoV and EWTABS defined in the OSP. Currently, human planners and controllers face two major challenges. First, there is a lack of guarantees about the effectiveness of short-term control actions performed on the system when its operation oversteps its viability zone. This refers to human controllers currently not being able to anticipate the potential impact of changes made to a particular route (*e.g.*, increased buses or departure frequency) because of unforeseen conditions requiring immediate attention. Second, and similarly to the previous challenge, the suitability of long-term evolution adjustments to the OSP defined by the planners is uncertain and mostly empirical. Thus, they are unable to fulfill the plans during operation. Consequently, there exist many opportunities to smarten transportation CPS. New and innovative mechanisms are required to augment the capabilities of constituent components to make it more resilient, efficient, autonomous, and self-managed, thus contributing to the system's advancement towards SUTS.

Figure 3.6.|: Feature-based classification scheme for SCPS.

3.5. Chapter Summary

This chapter presented an overview of the contributions achieved in the development of this dissertation. We started by identifying two remaining discontinuities in the software development process. Then, we described our reconceptualization of the time of change timeline focusing on the delivery pipeline and its constituents. Based on this, we introduced and briefly described our contributions, namely: A framework for continuous software evolution pipelines; An implementation for the continuous IaC evolution pipeline; A quality-driven self-improvement feedback loop; And a run-time evolution reference architecture. Finally, this chapter presented the two case studies we used to evaluate our contributions, namely cloud infrastructure management and smart urban transportation.

Chapter 4

A Framework for Continuous Software Evolution Pipelines

Contents

In this dissertation, we aim to develop self-evolution through self-management, self-improvement and self-regulation. This chapter addresses self-management, aiming to eliminate technical debt caused by run-time variability. It focuses on producing persistent updates to development artifacts based on run-time changes (*cf.* Figure 4.1). Research on autonomic computing has so far approached run-time changes aiming to provide quality guarantees in the face of uncertainty. This means that self-managing systems, as well as the adaptations they produce, are bound to the run-time context. Given our goal of automating operative development tasks, we leverage self-management capabilities on the development side by mapping run-time adaptations with persistent updates. That is, the contribution we present in this chapter monitors the execution environment, identifies run-time changes, and realizes and persists software changes on the development side. In this context, it is important to apply the principle of separation of concerns, meaning that run-time management should not be concerned with development-time updates. Otherwise, autonomic managers would have to adhere to continuous software engineering practices and tools. For example, they would have to be equipped with functions to interact with source code repositories, code review tools, automated tests, and other tools alike from the delivery pipeline. Additionally, not all persistent updates are intended to update code

artifacts. For example, in a smart transportation system, service improvements must be recorded in an operation plan that usually rests in a database. In this case, even though the operation plan can be represented as a MART, related updates would require the data to be persisted in a database instead of source code. Other types of persistent updates may require additional steps to complete the update. This includes tasks such as messaging stakeholders, or even asking them for confirmation to proceed with a particular course of action. Therefore, by separating the run-time adaptations from the evolution process of development-time artifacts, the task of evolving the system is simplified, allowing autonomic managers to focus on the self-management capabilities.

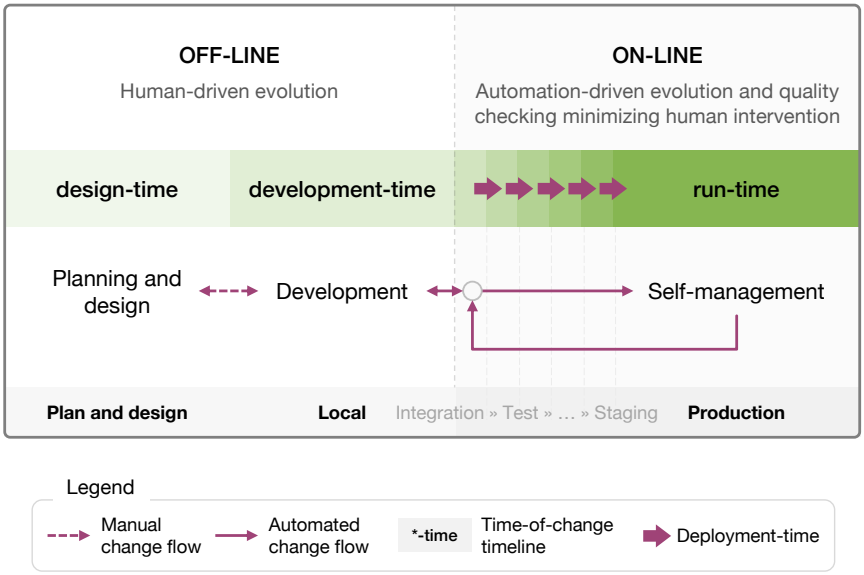


Figure 4.1.|: Self-evolution through self-management

This chapter focuses on the integration of system development and operation. This topic has been discussed and motivated before [**?**, **?**, **?**], and some works have been developed [**?**, **?**]. Nevertheless, the state of the art still lacks concrete contributions to bridge the gap between current engineering practices and run-time adaptations, not in the forward direction, but precisely from execution back to development. Furthermore, proposed contributions do not take advantage of execution environments other than production to reify predictive models and devise configuration alternatives. Ultimately, we propose an analogous relationship to that of development and operations teams, in which the autonomic managers on each side cooperatively contribute to the continuous improvement of the system and its artifacts. This chapter lays the foundation for exploiting this relationship.

This chapter presents our framework for continuous software evolution pipeline from two perspectives. First, Section 4.1 focuses on the evolution process from the autonomic manager's viewpoint. That is, the chapter presents how autonomic managers fit into the continuous delivery process through two-way CI. We propose two evolution workflows. First, an autonomic manager adapts a MART that represents a specific aspect of the system. For example, a model of the computing cluster configuration can be adapted to scale the cluster's computing power. Second, an autonomic manager adapts the system through

the underlying computing platform. In this case, the change is applied directly and immediately, so the changes do not go through the quality procedures provided by the continuous delivery pipeline. Section 4.2 concentrates on the evolution process from the artifact's viewpoint. That is, the chapter presents how development artifacts are updated from modifications to a MART. This includes the transformation chain from the model to textual form, and then the update to the source code repository and other tools. In this way, both viewpoints constitute a holistic and continuous evolution process that connects the execution with the development of the system. However, not every change in a MART implies a change in the software code of some artifact, but in the knowledge representation structure.

Correspondences in This Chapter

Addressed Challenge(s): CH1—Self-evolution mechanisms should be compatible with practices of continuous software engineering for quality assurance; CH2—Autonomic managers should provide the criteria and decision-making process behind a change; and CH3—Self-evolution mechanisms should capture live infrastructure modifications and propose corresponding software changes; *Question(s)*: Q2—How can self-improvement and self-management capabilities be integrated with quality assurance in continuous software engineering? Q3—What are the modeling requirements to map live modifications to a computing infrastructure with changes to its deployment specification? and Q4—How can development-time and run-time autonomic mechanisms be integrated reusing knowledge artifacts from both sides? *Goal(s)*: G1—Establish a general solution to integrate development-time and run-time autonomic work; and G2—Develop a self-evolution mechanism to update development and knowledge artifacts whenever the execution environment changes. *Contribution(s)*: C1—Continuous Software Evolution Pipeline.

4.1. Two-Way Continuous Integration: The Autonomic Manager's Viewpoint

This section describes a support layer for realizing self-evolution from a software engineering perspective (*cf.* ❶ and ❷ in Figure 3.2). It realizes the separation of concerns principle by allowing autonomic managers to evolve development artifacts without mixing management and development concerns. That is, they focus on updating run-time artifacts, avoiding the need to implement source code updates. On a conceptual level, we achieve this by extending CI to consider software changes produced autonomically. Typically in CI, a software developer integrates software changes into a shared source code repository. From there, a build server constructs executable artifacts and runs various types of tests. In this way, the developer gets quick feedback on the quality of their changes. Our extension to CI consists of introducing changes enacted online by an autonomic manager. On a practical level, we achieve this by introducing a set of components to coordinate and

realize evolution actions. Software changes enacted at run-time go through all the stages of testing and quality assessment composing the delivery pipeline.

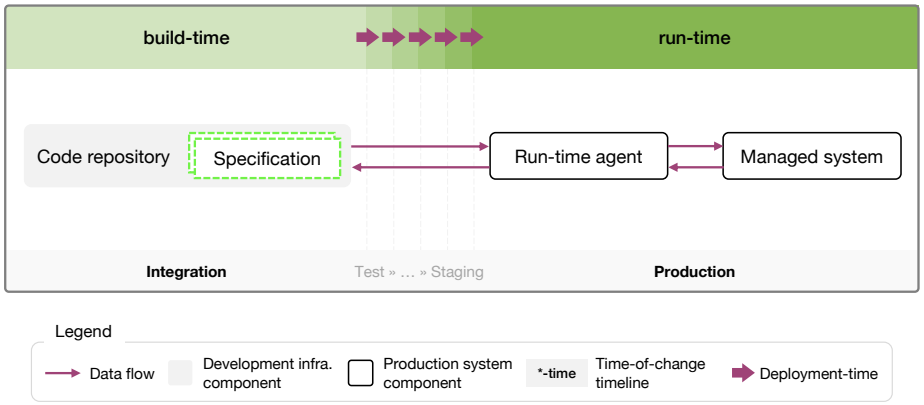


Figure 4.2.|: Overview of our self-evolution support layer

Our support layer reconciles IaC specifications with their corresponding elements from the managed system. It integrates run-time changes applied directly to the computing infrastructure into the source code repository. We call this support layer a run-time agent. Its main objective is to keep an up-to-date MART representing the managed system's state to transform it to text form eventually, in order to track the modifications throughout versions. This is used to determine what parts of the corresponding specification need to be updated. Figure 4.2 depicts a high-level overview of our self-evolution support layer that shows how information flows from both sides (*i.e.*, the source code repository and the managed system).

4.1.1. The CI-Aware and Direct Self-Evolution Workflows

We conceive two self-evolution workflows and illustrate them in Figure 4.3, as follows.

CI-aware Evolution This evolution workflow starts when an autonomic manager or a human in the loop requests a run-time agent to update a MART. This update causes an immediate code contribution but an eventual deployment of the change, meaning that the change will go through the delivery pipeline before it reaches the running system. Once the autonomic manager has caused the MART update, the run-time agent produces an evolution action to update the corresponding IaC specifications in the source code repository. As expected, the build server notifies the delivery server about the test and quality assessment results. If the change passed the quality checks, the delivery server opens an update transaction in the run-time agent before re-deploying the system. It does so to avoid inconsistent MART updates during the deployment. Once the update is done, the delivery server updates the MART and closes the transaction.

Direct Evolution This evolution workflow starts when an autonomic manager updates the running system directly through the underlying computing platform. In this case, the

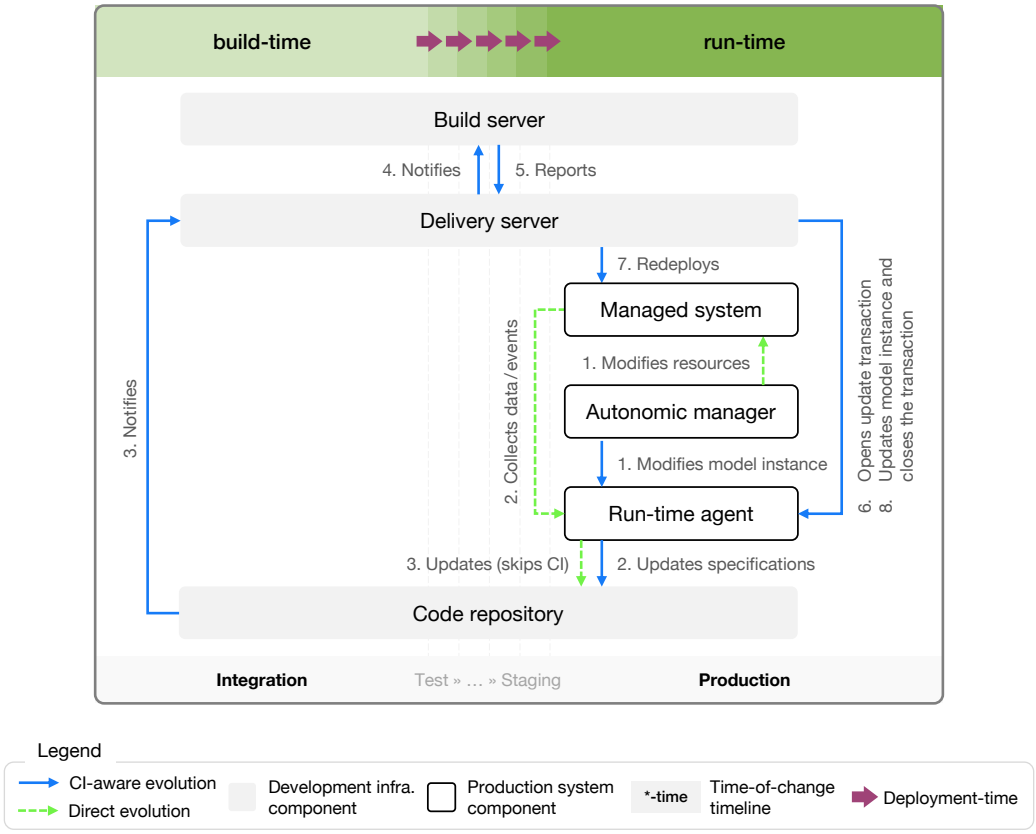


Figure 4.3.|: Direct and CI-aware self-evolution workflows

change would typically skip the delivery pipeline because it was already applied in the production infrastructure. Nevertheless, skipping the delivery pipeline is optional. Provided the change does not meet established quality criteria, the run-time agent could undo it. This evolution flow may be necessary to avoid delays in the update process. A typical example that requires an immediate intervention in the infrastructure is sudden increases in demand that affect service quality.

4.1.2. Technical Considerations

This section discusses three considerations regarding the implementation and adoption of our self-evolution workflows. We outline concerns that could potentially affect the development workflow and propose alternative solutions.

4.1.2.1. Contribution Strategy

Although Continuous Delivery (CD) considers mechanisms to guarantee the software quality, unsupervised changes can produce adverse effects. For example, a defect in the MART update process can cause side effects in the system's operation. We identify two strategies to update the IaC specifications: the committer strategy, in which run-time agents are

collaborators to the source code repository (*i.e.*, they have writing access); and the contributor strategy, in which the agents propose code modifications as merge requests rather than directly committing the code. A similar contribution model can be applied to IAC management software. On the one hand, directly updating an IAC template instance is analogous to committing changes to a source code repository. In this case, effecting the modification refers to updating parameters of the template instance. On the other hand, duplicating the template instance and updating the parameters is analogous to creating a new repository branch. In this case, there is no concept of merge request. Therefore, stakeholders must review the new instance and replace the old one manually.

The committer strategy ensures no delay in reflecting the changes to the specifications. Because of this, updates under this strategy could produce fewer merge conflicts. However, it does not mitigate the risk of unwanted effects, such as an inconsistent behavior of the system as a consequence of flawed MART semantics. In contrast, the contributor strategy completely avoids such a risk. This benefit comes at the cost of engineers allocating additional time to review the merge requests, delaying the updates and increasing the possibility of merge conflicts. While a merge request remains open, the MART is inconsistent concerning either the specification or the running environment. One solution to this issue is to put the update on hold while the merge request is still open. When the code contribution is merged, the changes will go through the build server and both the MART and the managed system will be updated.

It is common today that computing platforms and autonomic managers make decisions to affect a running system. Therefore, it is acceptable, at least in some cases, to grant commit access to run-time agents. We believe that combining both contribution alternatives is the best option. Furthermore, since the software changes are directly linked to configuration changes in the infrastructure, the need for an extensive review is unlikely. First, the run-time changes likely come from automation, which means that possible adaptations have been tested thoroughly. Second, if the changes were enacted by a human in the loop, they are likely reviewing the corresponding software changes. Finally, if the changes were already applied at run-time, they must be enacted on the development side to maintain consistency. Software changes in the repository can be rather seen as an opportunity to store the history of run-time changes, their impact on the source code, and the execution conditions that triggered the change. This is in itself a knowledge base for training and reifying predictive models.

4.1.2.2. Conflict Resolution

Conflict resolution is not a trivial task. It requires spending time inspecting the code and making informed decisions about the merge conflicts. Therefore, automating conflict resolution likely requires simplifying the problem. We identify two strategies to do so. The first strategy aims to avoid conflicts related to source code formatting. The transformation from a MART to its textual form must follow a standard process, which always generates statements in the same order, case and format (*e.g.*, spacing and indentation). The stakeholders working with the corresponding IAC specifications must follow the same formatting rules.

To facilitate doing so, they can use a formatting utility before committing changes.[1] The second strategy is to either raise or lower the run-time agent's priority in the merge process. In case of merge conflicts, the agent can decide to either drop the local changes or replace the remote ones, according to its assigned priority level. The former requires to roll back the latest changes to keep the system and the MART consistent with the specification.

4.1.2.3. Quality Assurance

One of the most important parts of continuous delivery is the application of quality control. We propose the use of pre- and post-validation rules to ensure invariants and quality validations before and after modifying the MART. There could also be concerns on the MART itself, rather than its state concerning a certain operation. For instance, consider an IaC specification where an IP address is specified outside of the network IP range. The state itself is erroneous, making it necessary to check the MART quality before deployment. Another type of concern is business restrictions; validations on the MART allow, for example, limiting what can be deployed by the tenant or even how. These business restrictions can be implemented as semantic restrictions on the MART, as the MART itself represents entities from the domain in which these rules live.

It is worth noting that the build and delivery servers are in charge of several tasks in addition to the regular testing and deployment workflow. Checking the quality of the MART, as suggested in this section, would likely happen in the build server as a testing step. Opening and closing an update transaction and updating the MART are tasks of the delivery server, as shown in Figure 4.3.

4.2. Round-Trip Engineering: The Artifact's Viewpoint

Section 4.1 introduced two workflows for conducting self-evolution: *direct evolution* and *CI-aware evolution*. In the first case, IaC specifications need to be updated right after the running system is evolved. This is necessary to keep the specifications consistent with the managed system. In the second case, the specifications need be updated as well, but in this case the source of change is the MART. This section describes how we perform round-trip engineering to keep the artifacts from both sides consistent. Section 4.2.2 introduces the main components of our solution, and explains how their interactions realize the specification update process. Section 4.2.3 concentrates on how direct evolution actions are mapped to MART updates automatically, thus connecting the two self-evolution workflows. Finally, Section 4.2.4 describes how the main components of our solution coordinate the evolution of IaC templates and instances.

[1] Nowadays, these formatting utilities are integrated with source code management tools, such as Git, through a *hooks* API. This way, formatting is executed automatically whenever changes are committed.

4.2.1. A Running Example

This section introduces a running example based on our IAC cloud management case study (*cf.* Section 3.3). We will reference such an example in the rest of the chapter. A rather simple but common software deployment includes a set of VMs connected to a private virtual network. For the sake of simplicity, we reduce this example to a single virtual machine running on VMware's hybrid cloud.[2] This kind of deployment is commonly used for small data science projects, internal services, and demos. Notwithstanding, since the VM is deployed to a hybrid cloud, it is subject to automatic migrations and policy conformance procedures. Therefore, even with such a small example, there may be configuration drift issues. In this example, we use the IAC tool Terraform[3] together with a template management software.

Below are the variables required for the bare minimum configuration of the deployment instance.

- **Datastore:** Data store where the disk will be stored
- **Disk size:** Hard drive size
- **Folder:** Server folder where the VM will be located
- **Name:** Unique name associated with the VM
- **Network:** Existing network to which the machines will be connected
- **Cores per socket:** Number of cores per socket
- **vCPUs:** Number of virtual CPUs
- **Operating system:** Base operating system
- **MART:** Amount of volatile memory
- **Resource pool:** Resource pool associated with the VM
- **Virtual machine template:** Custom configuration on top of the base operating system

4.2.2. Continuous Integration Loop

Figure 4.4 shows the elements of a run-time agent and their relation with the subject IAC specification. A run-time agent comprises five main components: (1) a MART that represents a particular aspect of the managed system. Examples of this are models representing the physical and virtual hardware infrastructure. In the case of our running example, this MART represents VMware resources that can be deployed with Terraform; (2) a model-to-text transformation to generate the textual representation of the MART (*e.g.*, Terraform templates); (3) a model transformation to create a map of current configuration parameters and their values based on the MART; (4) the run-time semantics from the application domain to test the MART as part of the delivery pipeline (*cf.* the aforementioned pre- and post-validation rules); and (5) causal links, which are dependencies with other MARTs, allowing the propagation of changes from one model to its dependents, as well as the

[2] https://www.vmware.com
[3] https://www.terraform.io

tracking of changes. For example, re-configuring the managed system's physical infrastructure should cause a change propagation to the virtual hardware infrastructure model, assuming the latter depends on the former.

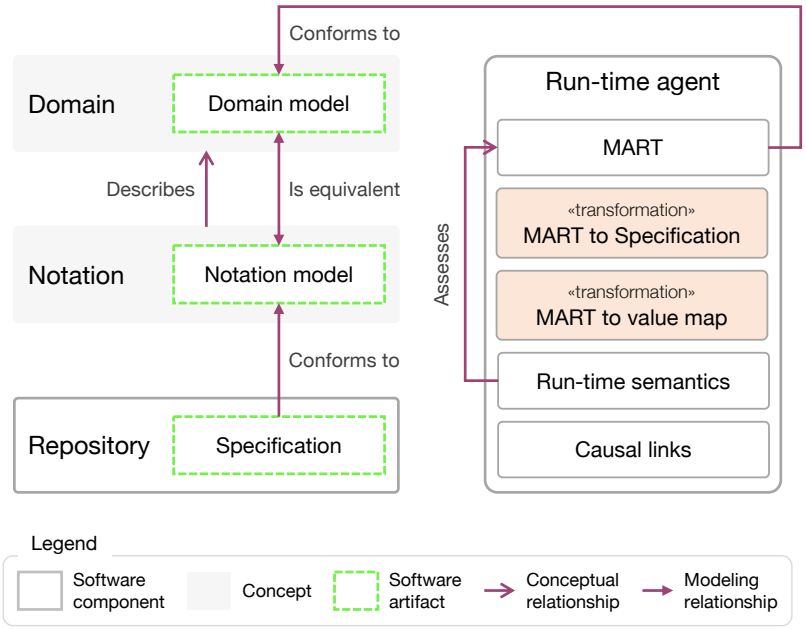


Figure 4.4.|: Domain and notation modeling relationships

A MART conforms to a domain model, and must be equivalent to a notation. That is, a one-to-one relationship is expected between the specification notation and the domain model. However, achieving such a relationship is often difficult; it may be necessary to limit the facts that can be expressed to guarantee said equivalence. To map the concepts from one model to the other, the pair domain-notation is associated with a set of transformations. For example, a MART representing the networking domain can be set up to work with OpenStack Heat Orchestration Template (HOT)[4] and/or HCL[5] (*i.e.*, Terraform templates' notation). Each of these configurations contains the necessary information to update the model instance and the specification, given a change in either of them.

Figure 4.5 represents the main components of the CI loop. That is, the necessary components to realize round-trip engineering of the IAC specifications. From left to right in the transformation chain (*i.e.*, forward engineering), the IAC specifications are used to instantiate a local MART instance inside the build and delivery servers. The specification parser reads the text files from the repository and returns an instance of the notation model (*i.e.*, `Specification`). The build server runs a job to validate the local MART, and the delivery server updates the MART hosted by the run-time agent. Other deployment tasks are conducted as established in the delivery pipeline. From right to left in the transformation chain (*i.e.*, reverse engineering), changes may arise from the managed system or the run-time agent. In the first case, a *Historian* monitor periodically collects information from the cloud platform's API. It maintains an up-to-date representation of the managed system's

[4] https://docs.openstack.org/heat/latest/template_guide/
[5] https://www.terraform.io/docs/language/syntax/configuration.html

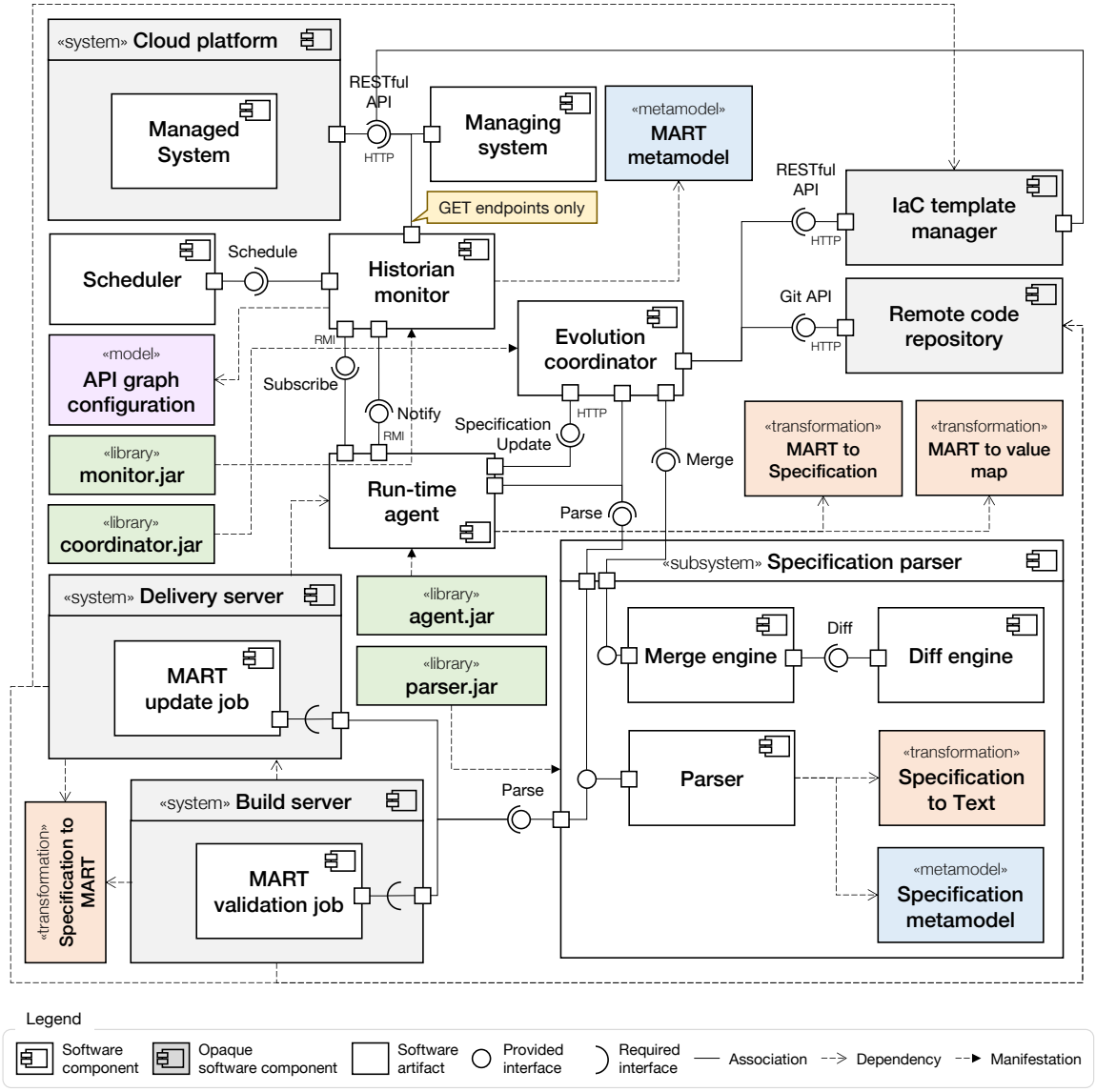


Figure 4.5.|: Components of the CI loop

state based on the responses from the API. Changes to the state detected by Historian are passed to the run-time agent. Listing 4.6 depicts a simplified view from our running example. In this case, Historian collected relevant identifiers as well as properties of the deployed VM. In the second case, an autonomic manager directly passes desired state changes to the run-time agent. In any case, the run-time agent passes template and instance updates to the evolution coordinator. This component then proposes the corresponding changes to the source code repository and the IaC template manager. The nature of these changes depends on the contribution model implemented. The code contribution and parameter value updates are devised by transforming the MART into the corresponding notation model, and then to text form and value map.

Table 4.1 exemplifies the variable names and values collected for our running example. The model transformation creates variables according to the deployed resources, matching

those present in the template and its instances. The information displayed in the table represents the value map used to update the template instance.

4.2.3. Continuous Models at Run-Time (MARTs) Evolution

Synchronizing a state model with the managed system is not a trivial task. Model evolution is typically implemented following the publish/subscribe design pattern.[6] In the case of a changing cloud infrastructure, the publisher is the cloud platform and the events are change notifications. The latter is usually triggered for every state the system experiences, including before, during, and after a modification takes place. It also includes myriad events unrelated to the updates of interest, such as authentication, role management, and billing notifications. Model synchronization can quickly become an architecturally significant requirement for the CI loop. The volume of functionality stemming from implementing hundreds of event handlers and mapping them with model updates is very large, especially if several cloud platforms need to be supported. Moreover, cloud providers usually use distinct notification communication technology. To make matters worse, the level of granularity and the information included in the events vary per vendor and product. To reduce the implementation effort, we decided to adopt a different strategy to continuous MART evolution.

This section introduces Historian, the monitoring component depicted in Figure 4.5. Once Historian is notified by the Scheduler, it periodically collects resource data from the cloud platform's API to update the MART as described in Section 4.2.2. It creates a simple view of the computing infrastructure by requesting data of interest through the cloud's API. This view is then passed to a model-to-model transformation, which creates an instance of the run-time agent's MART. Since the view only contains concepts of interest (*cf.* Listing 4.6), the transformation is simple to implement.

Historian relies on the OpenAPI standard to derive relevant monitoring information related to the cloud platform's API. This allows Historian to target any cloud API. Given an OpenAPI specification, Historian generates a Java program that will interface with the run-time agent. The initial step in the project generation consists of transforming the input (OpenAPI) specification into an instance of Historian's monitoring model. Next, Historian generates the necessary configuration data to setup its run-time library. This includes configuration parameters such as data collection periodicity, authentication information, endpoint dependencies, among others.

Figure 4.7 depicts Historian's metamodel for monitoring REST APIs. Besides authentication information, a monitoring model (*cf.* Concept `Root`) contains a set of polling monitors (*cf.* Concept `Monitor`). Each of them is associated with a data model (*cf.* Concept `Schema`) and an endpoint that returns instances of the data model. A data model is expressed as a collection of properties.

[6] Two popular implementations are the Viatra framework (`https://www.eclipse.org/viatra`) and EMF Cloud's model server (`https://www.eclipse.org/emfcloud`)

```
{
  "listVcenterHost": [
  "host-112"
  ],
  "listVmFilteredByHost": {
    "host-112": [
    "vm-123"
    ]
  },
  "listVcenterFolder": [
  {
    "folder": "folder-1",
    "name": "My-Deployment",
    "type": "VIRTUAL_MACHINE"
  }
  ],
  "listVcenterDatacenter": [
  "datacenter-1",
  "datacenter-2"
  ],
  "listVcenterVmFilteredByDatacenter": {
    "datacenter-1": [
    "vm-123"
    ],
    "datacenter-2": []
  },
  "listVcenterVm": [
  "vm-123"
  ],
  "getVcenterVm": [
  {
    "vm":"vm-123",
    ...
    "cpu": { ... },
    "disks": [ ... ],
    "memory": { ... },
    "name": "my-vm-123",
    ...
  }
  ],
  "listVcenterResourcePool": {
    "value": [
    {
      "name": "Resources",
      "resource_pool": "resgroup-1"
    }
    ]
  },
  ...
}
```

Figure 4.6.|: Historian's output summary for our running example

Table 4.1.: The run-time agent's value map for our running example

Variable Name	Value
datastore_1_name	CAM02-RSX6-002
vm_1_disk_1_label	camc-vis232c-vm-123/camc-vis232c-vm-123.vmdk
vm_1_disk_1_size	45
vm_1_disk_1_unit_number	0
vm_1_folder	folder-1
vm_1_name	my-vm-123
network_1_interface_1_label	VIS232
vm_1_num_of_cores_per_socket	1
vm_1_number_cpus	2
vm_1_guest_os_id	UBUNTU_64
vm_1_memory	1024
resource_pool_1_name	Resources

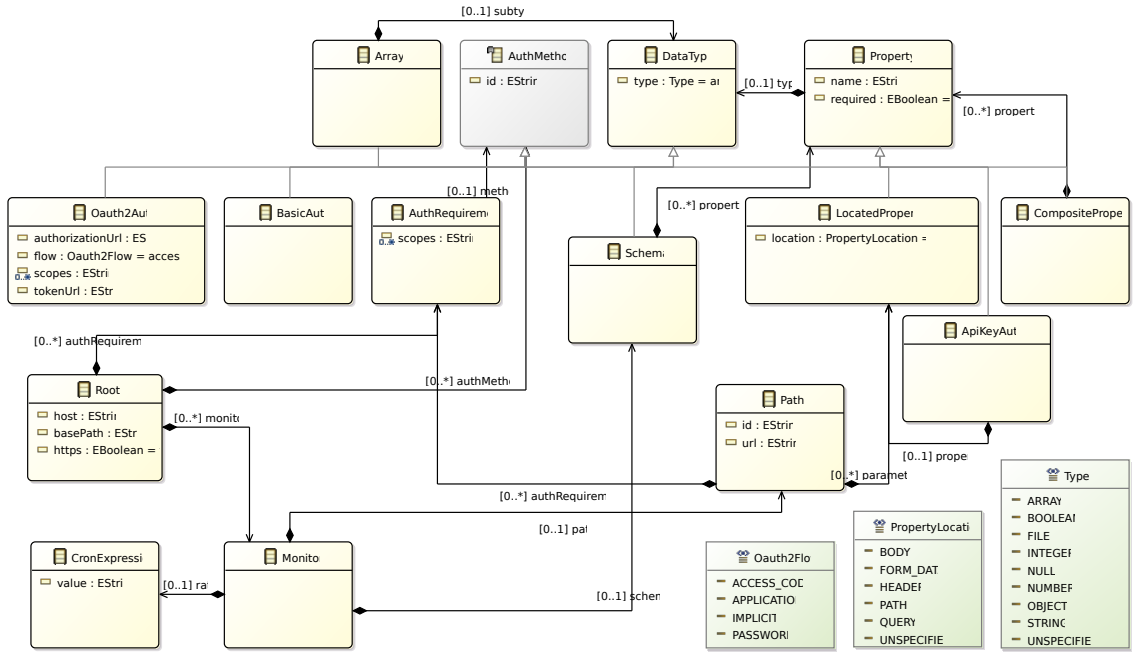


Figure 4.7.|: A metamodel for monitoring REST APIs

Historian's run-time library contains classes to perform HTTP requests and handle the result. The most relevant part of the library is an implementation of the FORK AND COLLECT algorithm. Our Fork and Collect resolves data dependencies between the API endpoints. For example, consider VMware's vCenter[7] API; to request details of a VM, the request must include the unique identifier of the VM. Provided the identifier, the corresponding endpoint will provide general details of the VM itself, but not about its associated resources, such as attached disks and CD-ROMs. Such details must be requested from other endpoints that also provide the resource identifier. If requests are performed in a certain order, some endpoints can provide data required by others. We consider these implicit links as data dependencies among endpoints. Although they must be manually configured, this process is done only once per cloud provider. It can be reused afterwards. Moreover, Historian assists in configuring them by generating the list of endpoints along with their required inputs.

Because data dependencies are present in any API, Historian expects the target API represented as a dependency graph. That is, a Directed Acyclic Graph (DAG) in which the edges are explicit dependency declarations in the form "endpoint A provides value O, which sources input I from endpoint B." As an example, Figure 4.8 depicts VMware's vCenter API as a dependency graph. Green nodes represent endpoints with no dependencies, which means they will be collected first. Blue nodes will be collected after their only dependency is released. Finally, all red nodes are collected, as they have at least two dependencies.

[7] vCenter is the centralized management utility for VMware, and is used to manage virtual machines, hosts, and all dependent components from a single centralized location.

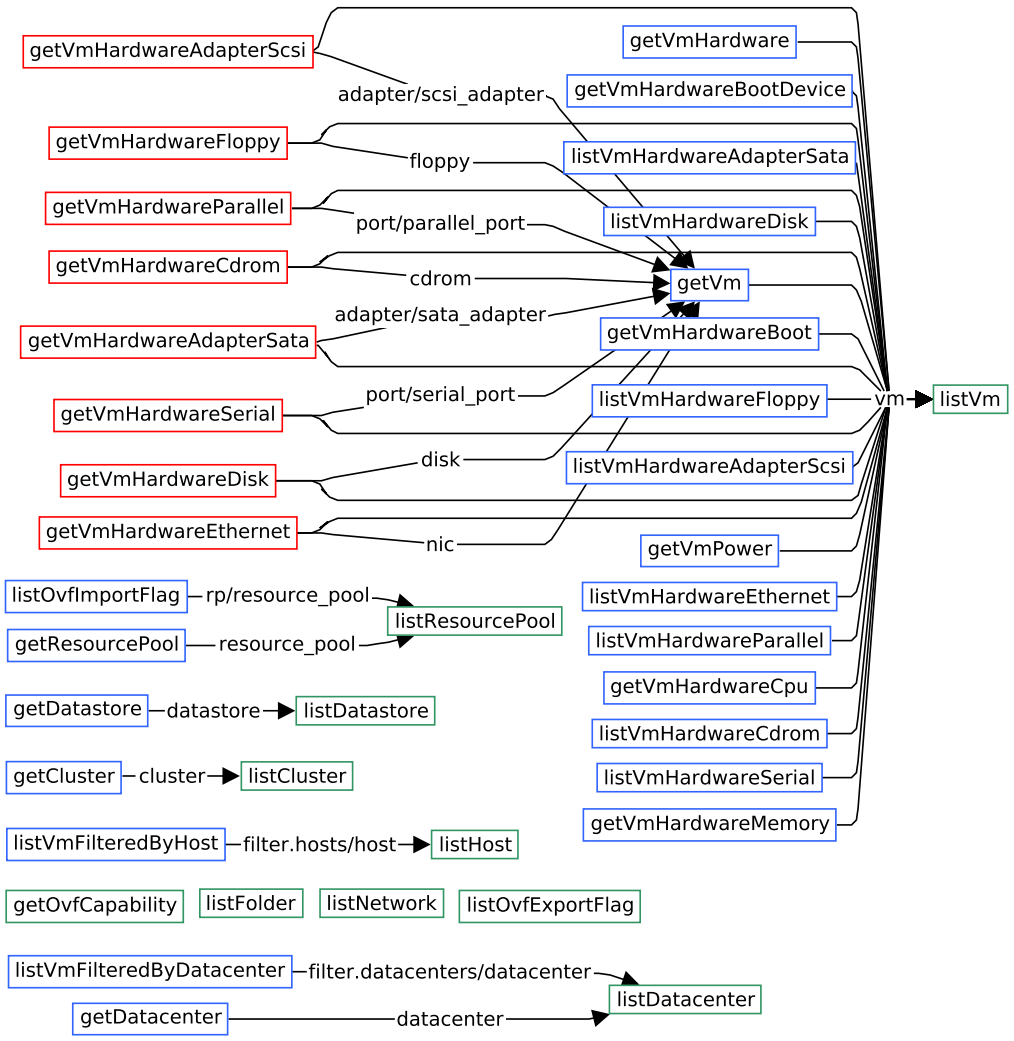


Figure 4.8.|: VMWare's vCenter API as a dependency graph

4.2.3.1. The Fork and Collect Algorithm

Algorithm 1 lists FORK AND COLLECT. Our algorithm aims to collect and aggregate resource information based on the aforementioned dependency graph. The algorithm starts by identifying the set of vertices (*i.e.*, endpoints) without dependencies (*cf.* Line 2). Each vertex is considered to be a branch. That is, an endpoint that will fork into many HTTP requests based on its inputs and collected values. For example, consider the endpoints `listVm` and `getVm` from Figure 4.8. The first one lists existing VMs and the second one gets the details of a specific VM. There is a dependency on the VM identifier from `getVm` to `listVm`. Suppose `listVm` collects identifiers `vm-123`, `vm-124` and `vm-125`. This means that `getVm` forks into `getVm(vm-123)`, `getVm(vm-124)` and `getVm(vm-125)`.

For each branch, the algorithm collects the resource data, marks the branch as released and optionally alters the collected data (*cf.* Lines 8, 9 and 10). Altering the data means transforming the result by either augmenting it with an input value, selecting a single value, or grouping a set of attributes by an input value. These transformations aim to simplify the

Algorithm 1: Fork and Collect

```
1  Function State(G)
       input  : a DAG G composed of vertices V and edges E
       output: the current state of the monitored resources
       // Initial fork step (vertices without dependencies)
2      B ⟵ {v ∈ V | ∀u ∈ V.(v, u) ∉ E}
3      D ⟵ ForkAndCollect(B, G, ∅)
4  return D
5  Function ForkAndCollect(B, G, R)
       input  : a set of vertices B; a DAG G composed of vertices V and edges E; A set of released
                dependencies R (vertices)
       output: a document D containing collected content from the API endpoints
6      D ⟵ ∅
7      for branch ∈ B do
8          content ⟵ Collect(branch)
9          R ⟵ R ∪ {branch}
10         D ⟵ D ∪ f(content)
11         X ⟵ ∅
12         for output ∈ branch.outputs do
13             values ⟵ Extract(content, output.selector)
14             X ⟵ X ∪ {(output.name, values)}
15         end
16         for values ∈ X do
17             N ⟵ Fork(branch, values)
18             D ⟵ D ∪ ForkAndCollect(N, G, R)
19         end
20     end
21 return D
```

resulting model.

Figure 4.9 displays an example of a group-by transformation. The document on the left shows a list of VMs for each host as returned by vCenter's API. The document on the right displays the VM identifiers grouped by their containing host. Since the `getVm` endpoint already provides details of a specific VM, only the identifier is necessary when using the group by transformation. Figure 4.10 displays an example of a value select transformation. The document on the left lists two hosts as returned by vCenter's API, including four attributes per host. The document on the right lists the same hosts, including only the identifier. Figure 4.11 displays an example of a value augmentation transformation. The document on the left shows a VM as returned by vCenter's API. The document on the right contains the same data, but includes the VM identifier.

Lines 11 to 15 of the algorithm extract values from the collected data to source the inputs from dependent endpoints. The values are represented as XML Xpath selectors. Based on these values, the algorithm forks the current branch into the branches for the next iteration. That is, the algorithm finds the vertices whose dependencies have already been released. Then, for each of them, it creates one branch per input value, as described previously in the `getVm` example. In Line 17, new branches are created by forking the current branch based on the collected input values. Finally, in Line 18, the new branches are passed as an argument to a recursive invocation of Fork and Collect. The resulting data

```
{
  "listVmFilteredByHost": [
    {
      "value": [
        {"vm": "vm-123", ...},
        {"vm": "vm-124", ...},
      ]
    },
    {
      "value": [
        {"vm": "vm-125", ...},
        {"vm": "vm-126", ...},
      ]
    }
  ]
}
```

```
{
  "listVmFilteredByHost": {
    "host-112": [
      "vm-123",
      "vm-124"
    ],
    "host-113": [
      "vm-125",
      "vm-126"
    ]
  }
}
```

Figure 4.9.|: Before and after applying a group by transformation

is added to the current document.

4.2.4. Run-time State Synchronization and Specification Update Workflows

Our reverse engineering strategy comprises two main components: run-time state synchronization and automatic source specification update. We separate these workflows based on their primary concern: MART evolution and specification update. Figure 4.12 extends the configuration previously presented in Figure 4.3. It includes an evolution coordinator component as well as an IAC template manager as part of the evolution life cycle of the specification. The former coordinates various types of updates to the specification based on the template and instance notions discussed in Section 3.3—*Infrastructure Management Case Study*. The latter manages template instances and their parameters. Notice that the deployment of infrastructure resources is no longer performed by the delivery server but the template manager.

The run-time agent no longer interfaces with the code repository directly but through the evolution coordinator. Its job consists of maintaining an up-to-date specification model, and sending it to the coordinator periodically. Whenever there is a change, the coordinator updates the source code as well as the parameter values in the template manager. These updates can be performed directly or by duplicating the update target—it depends on the contribution strategy.

```
{
  "listHost": {
    "value": [
      {
        "connection_state": "CONNECTED",
        "host": "host-112",
        "name": "h112.domain.com",
        "power_state": "POWER_ON"
      },
      {
        "connection_state": "CONNECTED",
        "host": "host-113",
        "name": "h113.domain.com",
        "power_state": "POWER_ON"
      }
    ]
  }
}
```

```
{
  "listHost": [
    "host-112",
    "host-113"
  ]
}
```

Figure 4.10.|: Before and after selecting a single value

```
{
  "getVm": [
    {
      "value": {
        "boot": {
          "delay": 0,
          "setup_mode": false,
          "retry": false,
          "retry_delay": 10000,
          "type": "BIOS"
        }
        ...
      }
    }
  ]
}
```

```
{
  "getVm": [
    {
      "vm": "vm-123",
      "value": {
        "boot": {
          "delay": 0,
          "setup_mode": false,
          "retry": false,
          "retry_delay": 10000,
          "type": "BIOS"
        }
        ...
      }
    }
  ]
}
```

Figure 4.11.|: Before and after augmenting the document with an input value

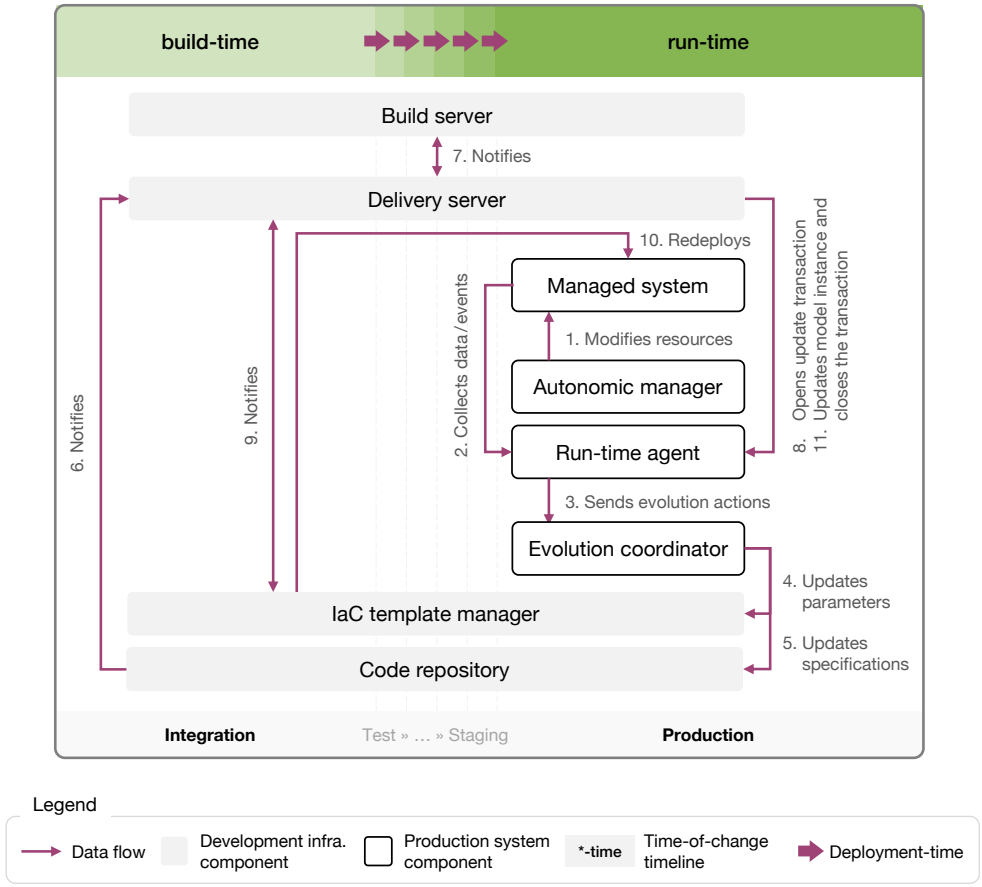


Figure 4.12.|: Evolution coordination for template and instance updates

Figure 4.13 illustrates the aforementioned workflows as an activity diagram. Once FORK AND COLLECT has collected the resource data, the run-time agent updates its MART and creates a map of parameter values. Then, it sends both elements to the evolution coordinator. It first compares the updated and current model. This ensures that elements contributed by developers are not removed in the update. This is the case of inline documentation (*i.e.*, comments), which are not preserved or used in the run-time context. To conduct this comparison, the template in the local repository is instantiated as a model. If the models are found structurally different, they are merged into a new model. The new model is converted into a textual representation that conforms to the notation model. These files are used to update the local clone of the template's repository. If no merge conflict is found and the template was in fact updated, the changes are pushed to the remote repository. The coordinator then compares the collected parameter values with the current ones. If there was a change, it updates the template instance or creates a new version in the template management system.

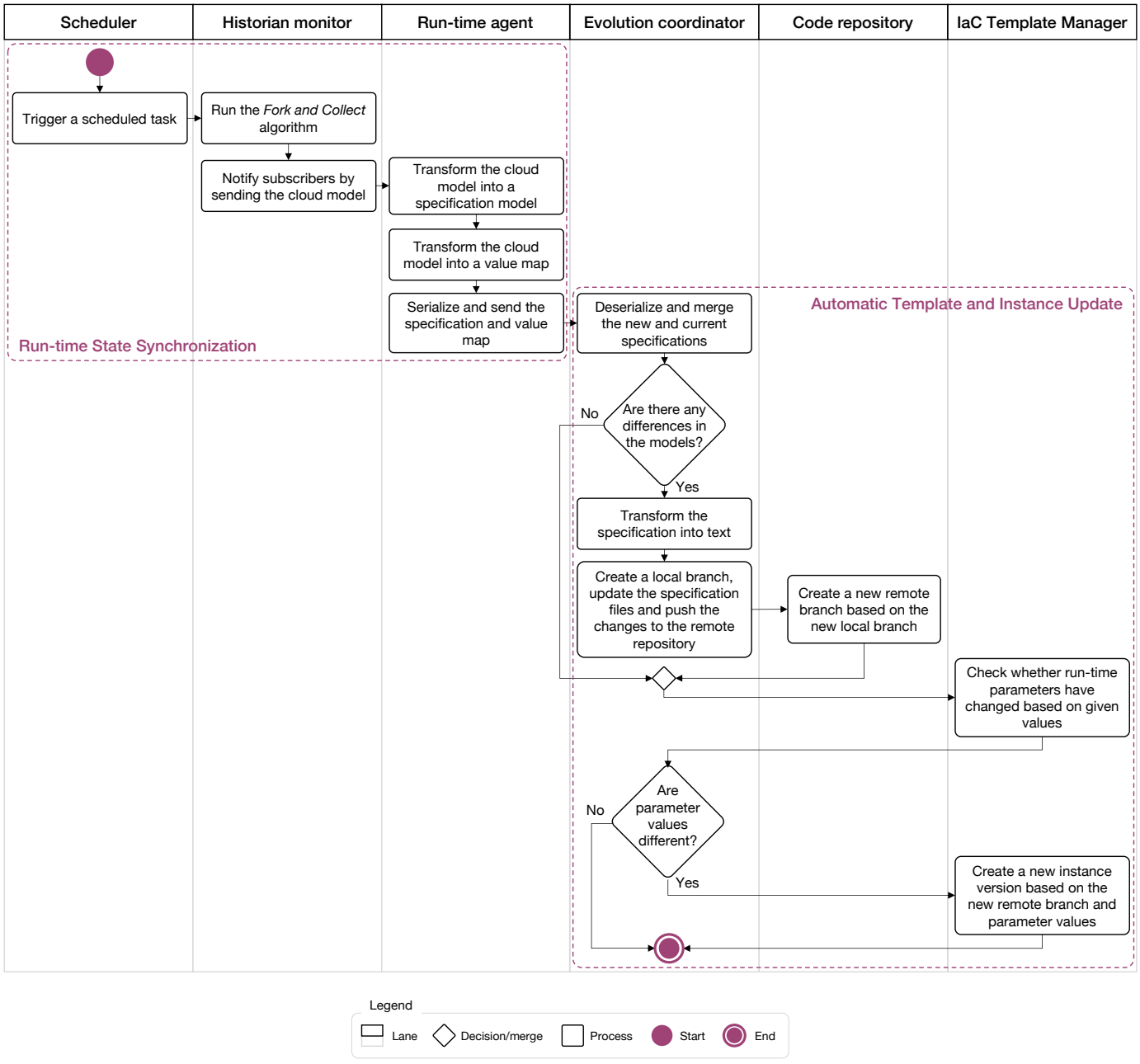


Figure 4.13.|: Run-time state synchronization and automatic template and instance update

4.3. Chapter Summary

An implicit assumption in software engineering and autonomic computing has been that persistent changes exclusively originate during development. This approach to software evolution is a legacy of traditional software engineering, where changes would typically flow from requirements elicitation to deployment, overlooking run-time management. In the last decade, an industry shift has put the focus on run-time technology, effectively changing software development and system administration practices. Operational concerns are now part of software planning and design, and development methods have been adopted on the infrastructure side (*e.g.*, Infrastructure- and Configuration-as-Code). Nevertheless, a more revolutionary change is yet to occur.

The implicit connection between operational requirements and changes on the development side remains largely unexplored. In spite of recent works contributing to making such a connection explicit, the focus tends to be on one side, still requiring stakeholders to conduct the evolution process. Further integration between development and operation requires a holistic approach to software evolution. Ideally, changes on either side (*i.e.*, development and operation) would cause the corresponding changes on the other side.

This chapter explored the integration between system development and operation from a software evolution perspective. We proposed a support layer that connects development-time and run-time artifacts. We leverage existing techniques and practices of continuous software engineering and autonomic computing. As a result, run-time changes are turned into traceable code contributions. By producing persistent updates from run-time changes, we aim to complete the evolution cycle. That is, make the connection between software changes from either side explicit.

Our contribution makes use of the already established delivery pipeline. It extends the concept of CI to consider run-time changes as part of the development process. Therefore, our support layer effectively turns autonomic managers into contributors to the software evolution. By leveraging the delivery pipeline, code updates go through the same quality assurance process applied to changes that originate during development. Moreover, this chapter presented an alternative evolution workflow that allows momentarily bypassing quality controls when context conditions require an immediate application of the changes. In this case, the autonomic manager adapts the system directly. The support layer monitors changes to the system and enforces quality criteria through the delivery pipeline, if required.

We used round-trip engineering to realize the CI loop. From left to right (*i.e.*, forward engineering), we rely on the delivery pipeline as it is with a few alterations. From right to left (*i.e.*, reverse engineering), we put in place a model-driven system to monitor and transform run-time changes into various types of persistent updates. It is worth noting that implementing the monitoring system can quickly become an architecturally significant requirement. To reduce the investment on development effort and time, we developed an algorithm to monitor changes to the running system.

The next chapter presents our proposal for realizing a second type of self-evolution, which relies on the contributions from this chapter. It consists of actively improving the system artifacts during development.

Chapter 5

Quality-driven Self-Improvement Feedback Loop

Contents

Increasing uncertainty demands organizations to react and adapt to change more quickly than ever before. At the same time, customers expect the continuous delivery of first-rate service quality. Architecture design together with dynamic infrastructure provisioning are critical in achieving the required agility and quality while keeping operational costs under control. Indeed, just-in-time elastic infrastructures allow reducing time to market while enabling the system to respond to ongoing and evolving environmental factors by adding resources on demand. Of course, adding computing power to a system is possible and worthwhile to the extent permitted by the architecture. In other words, power alone is not the only answer [**?**]. Abusing the cloud's elasticity to make up for the lack of adequate design structures could lead to infrastructure over-provisioning, thus, increasing the total cost of software ownership.

Finding a cost-effective software architecture and hardware configuration that satisfy the quality requirements is challenging, especially under uncertain execution conditions. First and foremost because actual workload requirements can change, unexpectedly. For instance, before the launch of Niantic's virtual-reality mobile game *Pokémon Go*, the engineering team tested the load capacity of the system to process up to five times their most

optimistic traffic estimate. Nonetheless in fact, the launch requests per second rate was nearly fifty times larger [**?**]. In settings like this, a reasonable development process requires consecutive cycles of configuration and deployment. On the one hand, it is necessary to characterize the behavior of the software system under different architecture configuration and execution scenarios. On the other hand, the team must also determine an adequate configuration for infrastructure resources. Second, there is a constant trade-off between spending time finding adequate configurations and developing bug fixes and features. This occurs primarily when the configuration changes do not produce any direct benefit for the users. Oftentimes, over-provisioning the infrastructure is seen as an alternative to meet the agreed quality of service in the face of uncertainty. However, because new code can add additional inefficiencies, a vicious cycle sets in while the cost of operation augments silently until it is too high, or adding more resources stops being effective (*e.g.*, the workload is transferred to the database).

In software-intensive industries, where the talent shortage only seems to grow [**?**, **?**], companies must find innovative solutions to the uncertainty problem. We advocate for the application of autonomic computing to development practices, not only to expedite software delivery, but especially to adjust the software based on changes in its execution environment. That is, in addition to continuous assessment of quality requirements as part of the delivery pipeline, we propose performing continuous improvement and adjustment as well. Thereby, self-management capabilities help remove what we consider to be a remaining discontinuity in the development process. Applying self-management as part of the delivery pipeline enables a connection between development and execution beyond the software. Knowledge developed by autonomic managers at development-time can be exploited and augmented at run-time. This, combined with the framework for continuous software evolution pipelines (*cf.* Chapter 4), realizes a continuous evolution process, thus mitigating run-time uncertainty.

This chapter presents a self-improvement feedback loop for exploring suitable architecture and infrastructure variants during development as a way of instilling self-evolution. Its main objective is to improve KPIs continuously. Such a feedback loop is based on what we call *quality-driven experimentation*. That is, a best-effort approach to quality improvement driven by experiment modeling and statistical analysis. The remainder of this chapter is organized as follows. First, Section 5.1 contextualizes our self-improvement feedback loop into the overall solution strategy presented in this dissertation. Section 5.2 explores modeling aspects of online experiment design. Section 5.3 follows the separation of concerns principle by decomposing the proposed feedback loop into experimentation, provisioning and configuration components. Moreover, it details how our feedback loop fits into the framework for continuous software evolution pipelines we presented in Chapter 4. Finally, Section 5.4 describes our proposed alternatives to devise architecture and infrastructure variants autonomically, even though our framework can accommodate any other.

Correspondences in This Chapter

Addressed Challenge(s): CH4—Self-improvement mechanisms should use design and

analysis of experiments to explore system alternatives for the dimensions of interest; and CH5—Self-improvement mechanisms should factor execution conditions into the experiment design; *Question(s)*: Q5—How can autonomic managers produce architectural and deployment variants? and Q6—How to reproduce typical execution scenarios in a controlled environment automatically? *Goal(s)*: G3—Develop an experimentation-driven self-improvement mechanism to improve KPIs during development; *Contribution(s)*: C2—Quality-driven self-improvement feedback loop.

5.1. Overview of the Feedback Loop in Our Solution Strategy

In this dissertation, we distinguish between traditional experimentation in software engineering and quality-driven experimentation. We make this distinction to emphasize the reduced scope of the latter with respect to the former. Classic experimentation has been conducted either by large organizations or research institutions with the purpose of evaluating development processes or practices. It is mainly presented as a research activity, thereby its return on investment may not be perceived as valuable by small companies. Oftentimes, this is the case because they are focused on delivering customer value to survive in increasingly competitive markets. This may be a plausible reason as to why they mostly conduct business-driven experiments. In contrast, we propose quality-driven experimentation as a best-effort approach to continuous improvement. This technique focuses on the qualities that make the subject software better, especially those most directly experienced by end users. Application examples are systematic design pattern selection and configuration, quality attribute optimization, and configuration tuning. It is not necessary to engage in all experimentation activities nonetheless, but only the ones adding quality to the product.

Two common arguments to refuse undertaking experimentation are that it slows progress and that experiments cost too much [**?**]. Quality-driven experiments are not only about trying out innovative solutions or embarking on large-scale studies, but exploring design, configuration and deployment alternatives with a reduced scope. In this chapter, we propose automating the design and execution of experiments, as well as the exploitation of resulting insights to improve performance factors (*e.g.*, service latency and throughput). To this end, Section 5.1.1 describes how the proposed feedback loop fits into our overall goal of automating software evolution during development. Section 5.1.2 explains how our approach contributes to removing one of the remaining discontinuities in the development process. And Section 5.1.3 describes the modeling layers to realize the feedback loop, including the physical and virtual infrastructure, as well as the software application.

5.1.1. Self-Managed Evolution of Development Artifacts

Chapter 4 introduced a support layer for synchronizing development-time and run-time artifacts. When used in production, as described throughout the chapter, it frees autonomic managers and stakeholders from producing additional updates to development artifacts,

thus contributing to reduce the technical debt. Nevertheless, it fits well in other environments as well. The self-improvement feedback loop, we present in this chapter, takes advantage of the framework for continuous evolution pipelines. It aims to produce persistent updates during development by exploring possible modifications to development artifacts. The end goal, in this case, is to find potential improvements for KPIs.

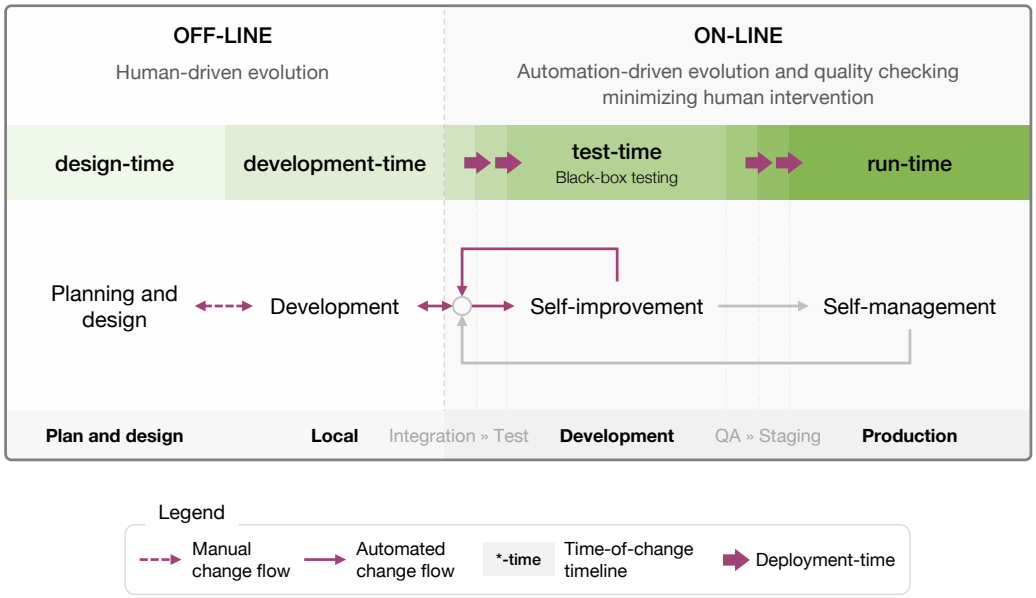


Figure 5.1.|: Self-evolution through self-improvement

Figure 5.1 depicts how the self-improvement feedback loop fits into the time-of-change timeline. As discussed in Section 3.1.1, we choose the development environment for exploring configuration alternatives for two reasons. First, it is stable enough to avoid wasting computational resources, and second, it is not as critical as subsequent environments for introducing changes. The self-improvement process starts once the development team creates a new alpha release. The build server triggers the self-improvement feedback loop, which generates and deploys system variants to an ephemeral controlled environment. Improvements are implemented using the CI-aware evolution workflow introduced in Section 4.1.1. Thereby, our feedback loop realizes self-evolution from a software engineering viewpoint (*cf.* Section 3.1.2).

5.1.2. Continuous Quality Assessment and Improvement

As mentioned in Section 3.1, a remaining discontinuity in the development process concerns the continuous adjustment of design, configuration and deployment. Despite development teams automating the reporting of performance inefficiencies, adjustment of design, configuration and deployment artifacts is not as frequent as required. We address this discontinuity by taking advantage of existing performance assessment facilities to improve the run-time quality. That is, from load, stress, spike, soak and scalability testing artifacts, our self-improvement feedback loop measures the response to variations in the

software architecture and configuration parameters. It is through continuous quality evaluation that our feedback loop establishes a quality baseline against which system variants are compared.

The experimentation process is independent from developers' work. Since it takes the latest alpha release as a starting point, daily software changes are unlikely to invalidate previously found improvements. In essence, the feedback loop acts as another developer whose main goal is to adjust the computing infrastructure and the software architecture. Once it finds a better alternative, it proposes the changes explaining the modifications and perceptual gains per KPI. Thereby, the self-improvement feedback loop aims to keep the system performant regardless of its evolution over time. In case neither the system baseline nor its variants meet the quality requirements, the feedback loop can alert the team.

Our self-improvement feedback loop reduces the risk of deciding too early in the design process. By providing an initial architecture with fewer assumptions (*e.g.*, not choosing a particular strategy to balance or distribute the workload), the team delays architectural decisions that will impact run-time qualities of the software. The feedback loop can then determine the best possible combination of design variants that will produce the expected quality. The benefit of such an approach is that the architecture will evolve along with the execution context (*e.g.*, user traffic over time), freeing the team from the related maintenance and development activities, at least at a high level. Moreover, the feedback loop can confirm whether initial assumptions of the execution context still hold and explore possible improvements if necessary.

5.1.3. Multi-Layer System Modeling and Evolution

Computing systems typically encompass multiple levels of specification and configuration. Accordingly, the self-improvement feedback loop can focus on exploring various aspects of the software architecture and the deployment configuration across these levels. Figure 5.2 illustrates a high-level view of our feedback loop applied to four modeling layers, namely: Physical Infrastructure, which comprises Configuration-as-Code (CaC) specifications and refers to operating system packages, services, policies and similar artifacts; Virtual Infrastructure, which comprises IaC specifications and refers to virtual machines, volumes, networks and similar resources; Software Deployment, which comprises any type of software configuration external to the application; and Software Application, which comprises functional elements, typically in the form of component or service definitions. These modeling layers represent aspects of the managed system that can be manipulated by the self-improvement feedback loop. Ultimately, a system variant that outperforms the baseline configuration will be used to update the specifications as described in Chapter 4. That is, a run-time agent (*cf.* ❶ in Figure 5.2) will derive the code contributions from its MARTs.

The modeling layer is the basis for generating system variants autonomically. For example, given a VM with attributes Random Access Memory (RAM) and number of Virtual Central Processing Units (vCPUs), the self-improvement feedback loop can generate variants by selecting a different attribute value each time. It can do so because the models

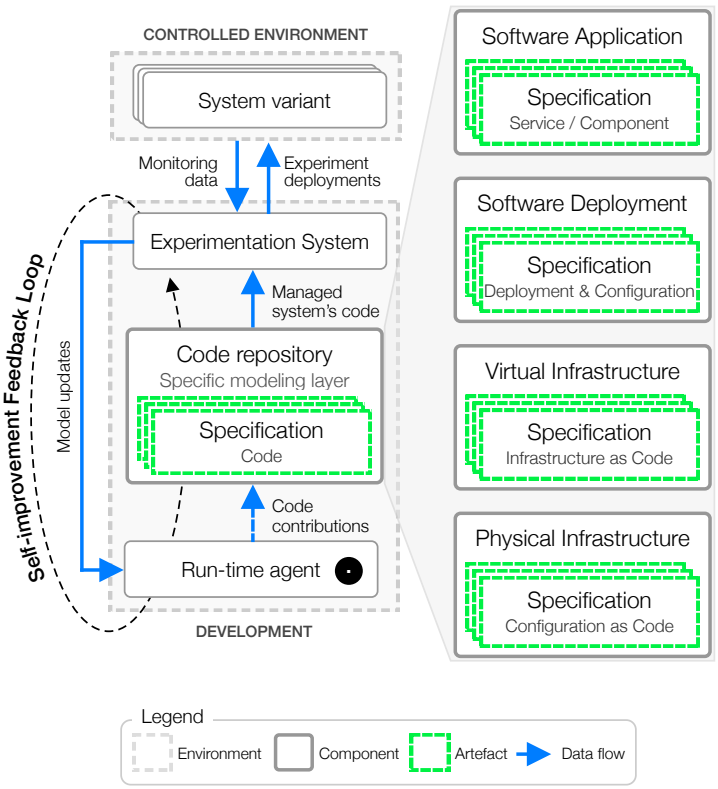


Figure 5.2.|: Levels of system specification and configuration

contain domain-specific information, such as valid memory sizes. A more advanced strategy consists in inserting new services into the software deployment to better handle the service demand. For example, a system variant may include a proxy cache to reduce network usage; or a rate-limiting proxy to continue serving requests while new computing nodes are deployed to handle a surge in demand; or a rate-limiting proxy to mitigate a denial of service attack.

In addition to the modeling layers presented in Figure 5.2, we identify an additional documentation layer.[1] It contains models to coordinate and document the software evolution, namely: a Quality of Service (QoS) model to represent KPIs and their relationship with the application services (*e.g.*, Kritikos *et al.*'s ideal quality of service metamodel [**?**]); a software changes model to specify the model changes included in a code contribution. This model is highly relevant for humans in the loop since they will likely need to evaluate whether a contribution is valuable and trustworthy, thus needing to understand the changes at an adequate level of abstraction. Furthermore, autonomic managers could use this model to create reconfiguration plans, and run-time agents could use it to propagate changes via their causal connections; and a decision documentation model for explaining to humans in the loop the rationale behind a code contribution and support its traceability from the software requirements (*e.g.*, Hesse and Paech's decision model [**?**]).

[1] In this dissertation, we concentrate on the modeling layer nonetheless.

5.2. Online Experiment Modeling

We guide the design of online experiments based on the concept map depicted in Figure 5.3. A feedback loop controls the planning and execution of experiment trials based on the concepts and relationships included in the concept map. An *experiment* (*cf.* Ⓐ) is first scoped by defining what is studied (*e.g.*, the virtual infrastructure), the particular focus (*e.g.*, effectiveness and cost of the computing cluster's size) and perspective (*e.g.*, the end user), the context of execution (*e.g.*, the managed system), and the intention behind it (*e.g.*, to predict the number of erroneous requests) [**?**, **?**]. Each experiment states a *hypothesis* in the form of an expression that operates on *independent* variables to obtain *dependent* variables across *trials*, performed through observed system executions. Each variable (*cf.* Ⓑ) is measured within a sampling rate and a measuring instrument (*e.g.*, through a performance monitor). An experiment consists of a set of trials whose *object* is typically a managed system and *subject* is an autonomic manager. The trials can be executed sequentially or concurrently (*cf.* attribute `execMode` of concept *experiment*). Each trial entails a *treatment* that considers a particular factor (*i.e.*, an independent variable), such as the number of computing nodes in a cluster, and a set of confoundings (*i.e.*, extraneous variables), such as whether the computing cluster runs on a shared physical infrastructure.

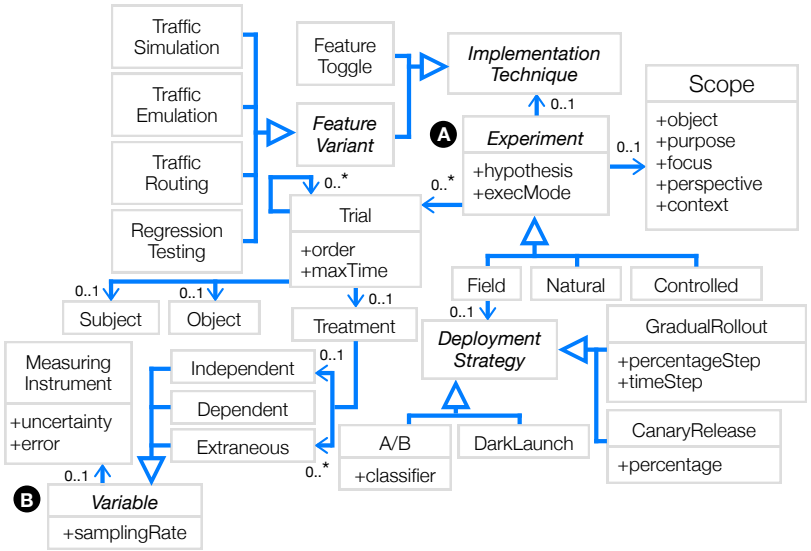


Figure 5.3.|: Our concept map to guide the online design of experiments

An experiment can be a *controlled experiment*—conducted in a lab or a replica of the production environment; a *field experiment*—conducted in production, along with the object of study; or a *natural experiment*—its object of study is determined by factors outside the control of the subject (*e.g.*, an experiment-agnostic human in the loop). A field experiment can be deployed selecting among various strategies, namely: *canary release*—the experimental code is exposed to a subset of users only, aiming at finding bugs on a small sample of the user population [**?**]; *gradual rollout*—the system variant is exposed to a percentage of the user base, which increases over time until it reaches 100% [**?**]; *A/B testing*—two user groups access the system baseline and variant; and *dark launch*—the system variant is

deployed to the production environment silently (*i.e.*, without being visible to any user) to test performance regressions using real user traffic. Since we chose the development environment for conducting self-improvement, this chapter focuses on controlled experiments only.

For experiments conducted on managed systems, each experiment can be implemented through a feature *toggle* or *variant*. Since a feature toggle is shipped together with the original object, the effect on dependent variables can be measured based on actual user traffic. For feature variants, the outputs can be measured based on various strategies, namely: *traffic routing*, meaning that user traffic is routed to the variant as it happens; *traffic simulation*, consisting of statistical simulation of user behavior (*e.g.*, queuing theory); *traffic emulation*, consisting of reproducing historical user traffic; and *regression testing*, consisting of automated load, stress, spike, soak and scalability tests.

In Figure 5.3 we omitted the definition of attribute types to simplify the concepts. The complete metamodel definition is depicted in Figure 5.4.

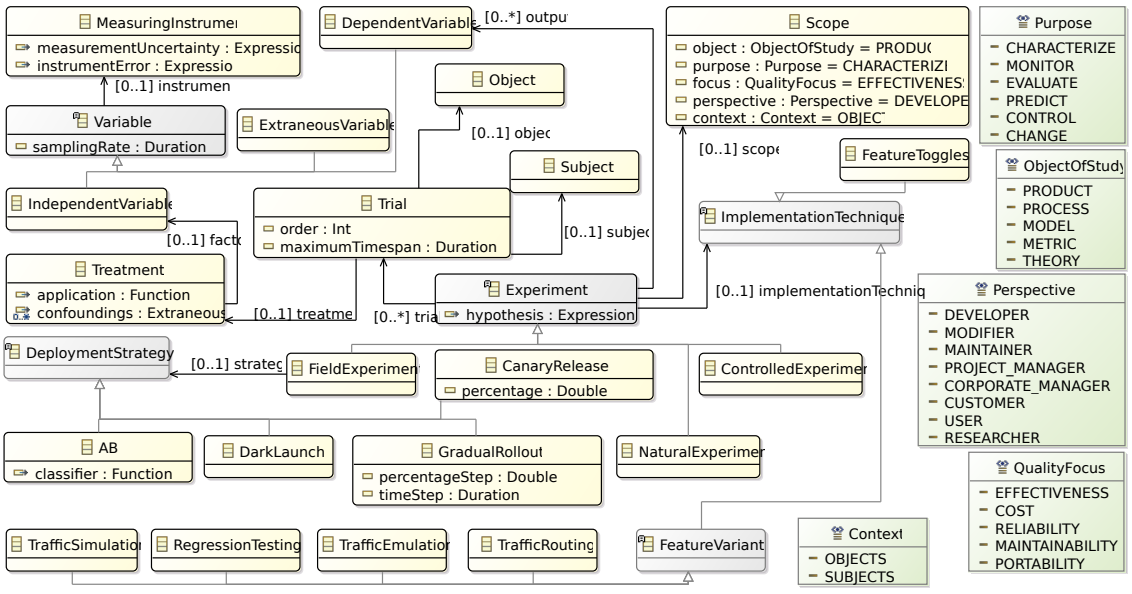


Figure 5.4.|: Our metamodel for online design of experiments

5.3. Online Experiment Management

Our feedback loop for planning and conducting online experiments is based on the ACRA [?] and the DYNAMICO reference model [?]. We reuse the control hierarchy from the former, and the goal management from the latter. Moreover, we address the separation of concerns in the experimentation process by decomposing the self-improvement feedback loop into three feedback loops with a reduced scope. Each of them is based on the MAPE-K reference model [?]. Figure 5.5 illustrates them and their high-level interactions. The

Experimentation Feedback Loop (E-FL) controls the satisfaction of high-level experimentation goals through online experiment design (*cf.* component Ⓐ). The *Provisioning Feedback Loop* (P-FL) derives, deploys and monitors infrastructure configuration variants (*cf.* component Ⓑ) in the form of IaC; and the *Configuration Feedback Loop* (C-FL) derives, deploys and monitors architectural design variants (*cf.* component Ⓒ). The deployment of system variants is done through the continuous evolution pipeline. Components Run-time agent, Evolution coordinator and Deployment template manager in Figure 5.5 reference the workflow depicted previously in Chapter 4, Section 4.2.4. Once an experiment is finished, the E-FL performs a persistent update through the run-time agent based on the system variant with the highest performance, if any.

The P-FL derives system variants by applying combinatorial techniques to configuration parameters of declared virtual resources. In contrast, the C-FL derives system variants by applying domain-specific design patterns to the managed system's architecture. The C-FL relies on its capability to measure the impact of design patterns on KPIs.

Although the impact of design patterns is quantifiable under controlled conditions, their application during software design is generally not data-driven. In practice, there is a vast body of knowledge to understand the motivating forces, structure and behavior of design patterns. However, there is a lack of quantitative data and analysis of their impact at runtime. We claim that our online experiments can help on two fronts. First, they help deciding which combination of patterns is best for specific workloads and quality requirements (*i.e.*, the experimentation goal). Second, our experiments help in tuning configuration parameters of each pattern. For example, setting up the number of worker connections and the limit on worker processes for a load balancer. Furthermore, the number of proxy workers must be determined according to the workload, taking into account that too many workers will negatively impact the database performance, or worse, will not contribute at all but will consume resources nonetheless. These are tasks for which development and operation teams have to account, relying on informal communications, metrics, logs and, oftentimes, trial and error.

Figure 5.6 illustrates the elements composing each feedback loop and their interactions. We describe these elements in detail in Sections 5.3.1, 5.3.3 and 5.3.4.

5.3.1. The Experimentation Feedback Loop (E-FL)

The E-FL synthesizes experiments based on a factorial design with 5 factors: i) the execution scenario modeled by the testing artifacts; ii) the arrangement of architectural elements; iii) the amount of RAM per computing node; iv) the number of Virtual Central Processing Units (vCPUs) per node; and v) the number of nodes in the computing cluster. The execution scenario is either constant traffic, linear growth of service demand, exponential growth of service demand, or quick surge of user requests (*i.e.*, a spike). The E-FL explores the search space aiming to meet high-level Quality of Service (QoS) goals. An example QoS goal is minimizing the latency for a particular software service.

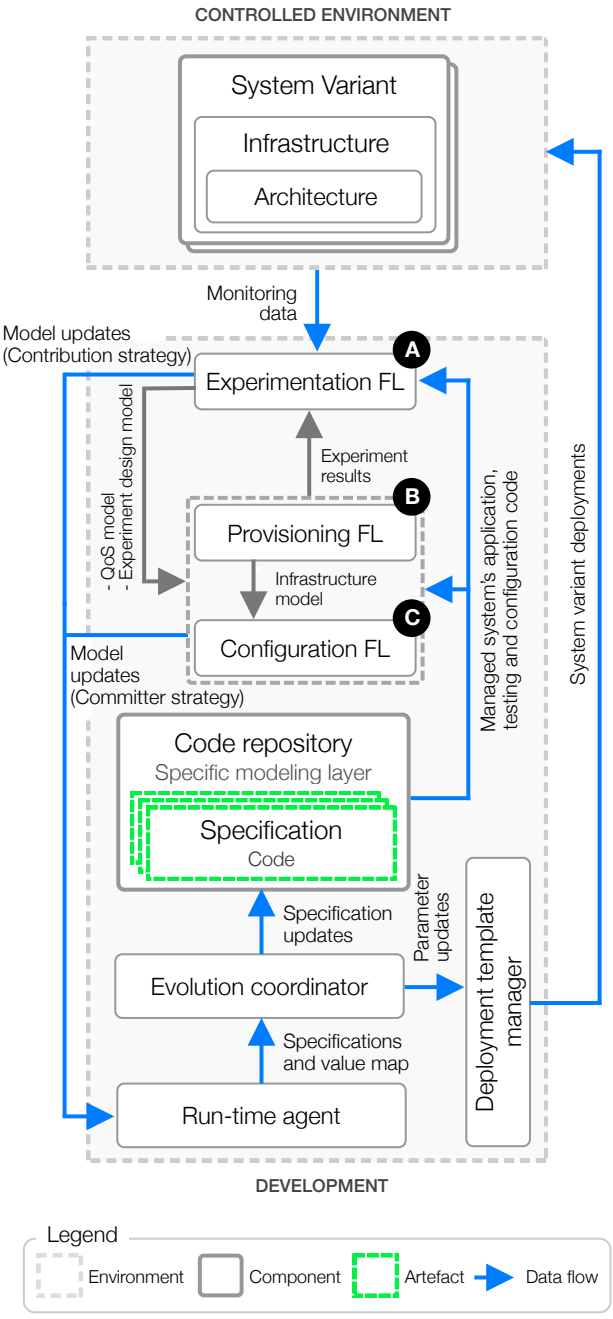


Figure 5.5.|: Structural view of the self-improvement feedback loop (FL stands for Feedback Loop)

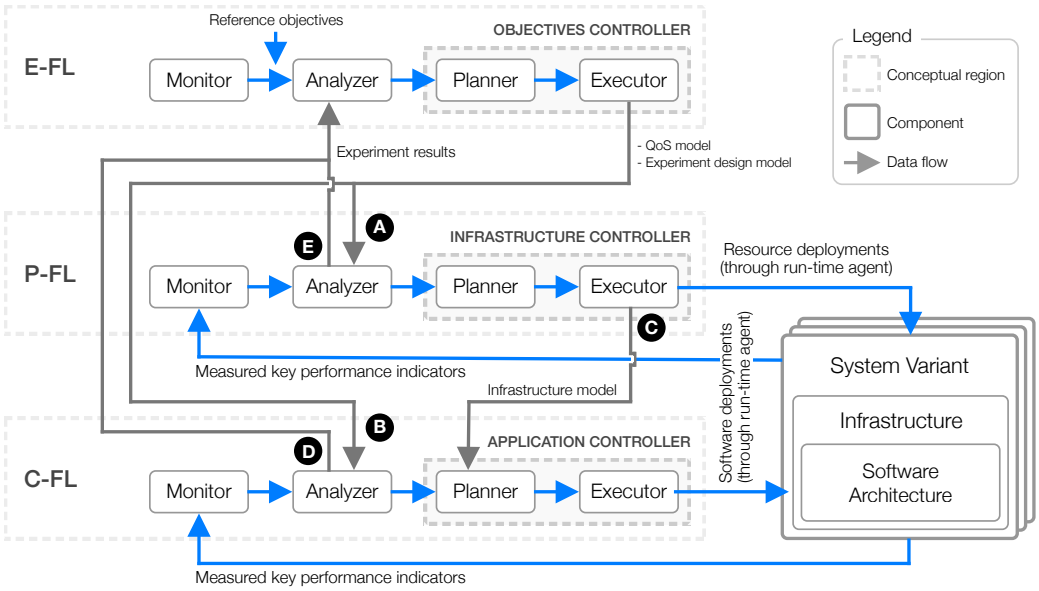


Figure 5.6.|: Internal feedback loops for experimentation, provisioning and configuration
We omitted the Knowledge element of the MAPE-K model.

Since the experiments exhaustively explore the search space, a first assessment of the approach may be that it takes too long to be useful or too expensive to be worth it. Nevertheless, new alpha releases of the software system are unlikely to affect past explorations heavily and frequently. This means that memoization techniques, along with a reassessment strategy,[2] can considerably reduce the experimentation time in subsequent releases. Moreover, even though experiments are conducted in the development environment, the search space exploration can be tremendously useful at run-time. The combination of factor values and measured KPIs serves as a data set to predict the behavior of the system during known execution scenarios. Run-time autonomic managers can bypass the exploration phase, perform pruning on the search space, or rather augment the data set, thus collaborating across environments.

The *Analyzer* uses partial and complete experiment results, if any, collected by the *Monitor* (*cf.* interactions Ⓓ and Ⓔ in Figure 5.6) to determine whether the *Planner* needs to update the current experiment design or create a new one. In the first case, the Planner will stop underperforming trials or discard those considered as less effective by the *Analyzer*. In the second case, the Planner fetches deployment and configuration patterns from the *Knowledge* element, and synthesizes one experiment design for the infrastructure and another for the software architecture. Both designs describe a controlled experiment implemented with feature variants according to our metamodel (*cf.* Figure 5.4). The design contains at least one trial per variant, but can include more for increased accuracy. The *Executor* component instantiates the P-FL and C-FL, passing as arguments the experiment designs and the QoS reference values. In case the QoS goal changes, this Executor updates the reference values in the P-FL and C-FL (*cf.* interactions Ⓐ and Ⓑ).

[2]For example, testing various combinations of factor values randomly to verify whether past explorations yield a similar result.

5.3.2. Statistical Analysis of Experiment Results

Depending on the type of experiment and the quality attribute under improvement, the target system is instrumented for data gathering and action enactment. Data gathering is performed by monitors deployed in the system. Once the experiment finishes, a set of statistical analyzers is launched to compute the selected KPIs of interest.

To select the system variant with the highest performance, the E-FL processes collected samples—included in the experiment results—from two fronts. On the one hand, the E-FL generates multiple charts per variant as well as summary charts comparing the mean, standard deviation, standard error and confidence interval of the data. These charts are relevant for humans in the loop who want to further understand why the E-FL selected a particular variant. The charts are accessible through the merge request created by the Evolution Coordinator component of the framework for continuous software evolution pipelines (*cf.* Section 4.2.4). On the other hand, the E-FL runs various tests on the data to determine which variant performed best and whether there is a significant difference in the measured KPI with respect to the other variants. These tests are conducted per application service because the results can significantly vary. The statistical tests are executed as follows.

1. Consolidate the samples into a single data set and filter out erroneous requests.
2. Use the Shapiro-Wilk test to determine whether the samples are normal.
3. Determine if the mean value is significantly different across variants using an analysis of variance (ANOVA) or a Kruskal-Wallis test depending on whether the samples are normal. In case more than one KPI is being studied, ANCOVA must be used instead.
4. If the mean values are found not significantly different, select any variant as the best one. In the opposite case, perform Dunn's test or Tukey HSD to cluster similar variants, depending on whether the samples are normal.
5. Combine the results for all the application services to select a single best variant. The selection is made by minimizing or maximizing the KPI under analysis, depending on each case.

5.3.3. The Provisioning Feedback Loop (P-FL)

The P-FL targets the experiment design part that is related to the virtual infrastructure where the configuration variants will be executed (*e.g.*, in cloud-provisioned virtual machines). The *Planner* synthesizes a provisioning plan based on the experiment trials involving the infrastructure. Using the provisioning plan, the *Executor* deploys and configures the corresponding resources accordingly and provides the C-FL with infrastructure data (*cf.* interaction Ⓒ). The *Monitor* collects measurements from the deployed resources, and the *Analyzer* aggregates and classifies them into partial and complete experiment results to feed them back to the E-FL (*cf.* interaction Ⓔ). In case the E-FL provides a new experiment

design or updates the current one, the *Analyzer* determines whether the provisioning plan should be resynthesized, thus triggering the redeployment of computing resources.

5.3.4. The Configuration Feedback Loop (C-FL)

The C-FL deals with software architecture design variants based on the factors specified in the design supplied by the E-FL. The *Planner* derives system variants based on the software architecture part of the experiment design. Next, the *Executor* deploys the variant being tried. Once it is running, the *Executor* deploys the necessary *Monitor*(s) for collecting the software QoS metrics and starts the testing strategy defined in the experiment design (*e.g.*, traffic emulation). This assumes that the baseline code includes the necessary testing artifacts to ensure these actions. Since there can be significant variability in the monitoring requirements, the deployment of the monitors might be non-trivial [**?**]. For instance, measuring throughput may require a particular data aggregation strategy across various services. Once the monitors have been deployed, the *Analyzer* aggregates and classifies the collected metrics into partial and complete experiment results to feed them back to the E-FL (*cf.* interaction ⓓ). It also determines whether it is necessary to derive the variants again, and therefore redeploy the software variants when the experiment design is updated by the E-FL.

5.4. System Variant Generation

As mentioned before, this chapter focuses on system variants at the virtual infrastructure and software deployment levels. This section details what variants are included and how they are derived. Sections 5.4.1–2 describe infrastructure and architectural variants, respectively.

5.4.1. Infrastructure Variant Generation

The P-FL derives system variants by applying combinatorial techniques to the managed system's Virtual Infrastructure modeling layer (*cf.* Section 5.1.3). It generates numerical and nominal values corresponding to virtual resources' attributes, thus producing distinct configurations of them. We focus on cluster configuration by evaluating the number of virtual machines (*i.e.*, computing nodes), the RAM size and number of vCPUs per node.

Infrastructure variants generation follows the standard implementation of a genetic algorithm with integer genes. Each possible configuration is represented as an array of integer values, where each element refers to a concrete value for a particular attribute of the cluster. For example, the configuration $[1, 2, 3]$ represents a cluster with 1 node, 4GB of RAM per node, and 6 vCPUs per node. The number of nodes range from 1 to 6; RAM belongs to the set $2, 4, 6, 8$; and vCPUs belongs to the set $2, 4, 6, 8$. The algorithm uses two alterers: a single point crossover and a mutator. The goal is to minimize the fitness

score, which comprises KPIs of interest measured both in the infrastructure (*e.g.*, resource utilization) and the software application (*e.g.*, service latency).

Infrastructure variants generation can either affect the IAC template, the template instance, or both. In the first case, selecting a concrete parameter value may require updating the declaration of resources in the IAC template. This requirement is technology- and vendor-specific.[3] In the second case, the P-FL will update parameter values of the template instance. In both cases, the P-FL performs these persistent updates through the run-time agent (*cf.* Figure 5.5), following the direct evolution workflow (*cf.* Section 4.1.1).

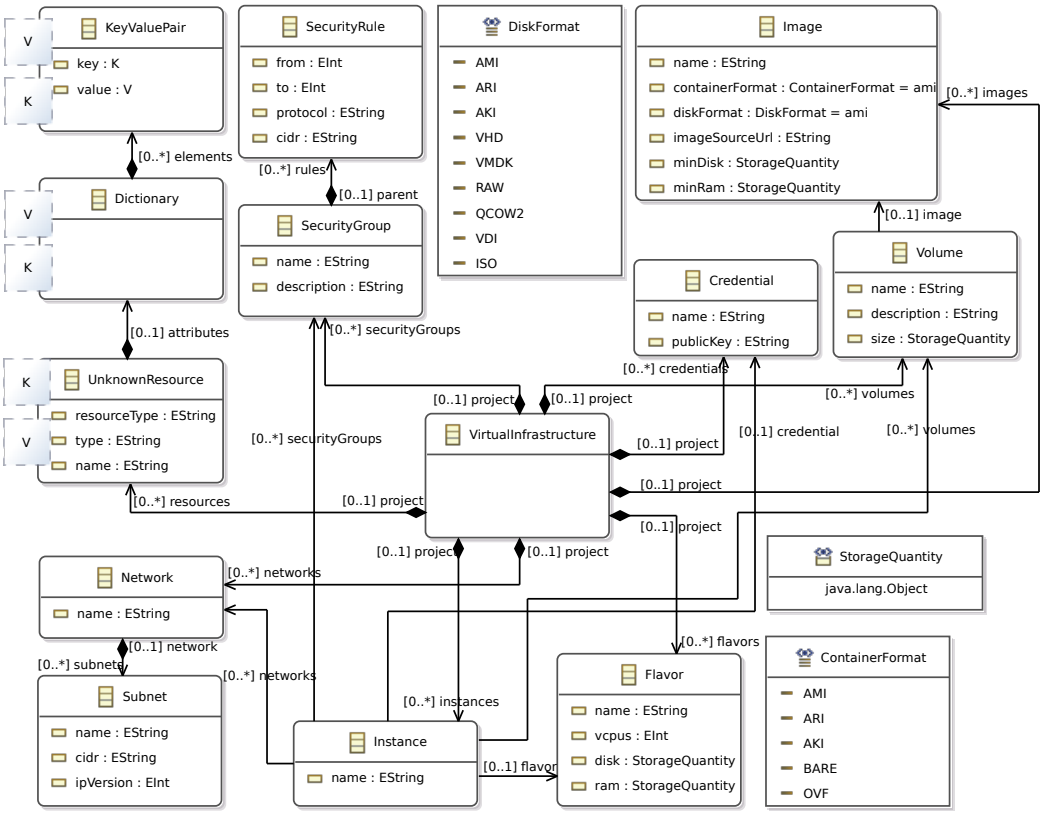


Figure 5.7.|: Our metamodel for virtual resources supported by OpenStack

In the public cloud, vendors usually set the compute and memory capacity of virtual machines to a concrete list of combinations. This is known as *t-shirt sizing*. This limitation greatly reduces the search space based on available sizes. Consider OpenStack[4] as an example. Figure 5.7 depicts our metamodel for representing the computing resources of OpenStack. An *Instance* represents a virtual machine running on OpenStack. All instances are created based on a particular *Flavor*—a *t-shirt* size. For the attribute values presented at the beginning of this section, there are 96 possible infrastructure variants (*i.e.*, $6 * 4 * 4$), including only 4 options for RAM and vCPUs. T-shirt sizing considerably reduces the

[3]For example, vendor A may provide an abstraction to declare computing clusters, including the number of nodes as a parameter, whereas vendor B requires declaring the computing nodes as independent virtual machines.

[4]An open source cloud software solution (`https://www.openstack.org`).

experimentation time, but at the same time prevents the P-FL from executing fine-grained control actions.

Figure 5.8 depicts a high-level workflow of the infrastructure experimentation process. This figure details, step by step, how the P-FL conducts each deployment. The P-FL interfaces with various third-party utilities, which in Figure 5.8 are exemplified by git, jMeter, Kubernetes, R, and Terraform. Git is used to create a local clone of the managed system's source code. jMeter allows specifying and running performance testing plans. An outcome of these plans is performance data for each service request. R scripts could be used to automate the analysis of the collected data, and the generation of statistical information.

5.4.2. Architectural Variant Generation

The C-FL derives system variants by applying domain-specific design patterns to the Software Deployment modeling layer (*cf.* Section 5.1.3). More specifically, it applies deployment and configuration patterns for cloud software [**?**, **?**], which include design strategies for load balancing, caching, routing, rate limiting, and job handling. Patterns such as proxies, interceptors, and decorators are plugged-in directly into the deployment specification. A generic pattern implementation contains the logic to declare and connect third-party components (*e.g.*, A Web Application Firewall) with application components. This is possible because new architectural elements are encapsulated in containers, which greatly facilitates their integration with the managed system through port bindings. For example, adding a SQL proxy to a specified database server requires: i) declaring the proxy in the specification exposing the database server's original connection port; ii) modifying the database server's connection port to another one that is available; and iii) configuring the origin server's port in the proxy to be the new database server's port. More advanced patterns can also be implemented as long as they and the software application are prepared beforehand, as in works such as QoS-CARE [**?**]. For example, the Master element of the Master/Worker design pattern is generally application-specific. This limitation comes from the strategy to split and merge each request, which depends upon the application's functional requirements. Furthermore, the service provider and consumer to which Master/Worker is applied must define an explicit service contract using language-agnostic interfaces.[5] Based on the interfaces, the C-FL can generate adapter components required to apply the pattern. Other patterns with similar requirements are Consumer/Producer and Reactor. Adding support for these patterns requires an additional investment of time and effort. Nevertheless, the architectural agility they provide pays off over time as less maintenance and tuning is necessary. Moreover, the extra effort is only required for developing adapter components, not complex business logic.

Figure 5.9 illustrates the Load Balancer pattern applied to a simple client/server application. The left-hand side of the figure depicts Linux containers `API` and `Worker` assembled into one *pod*—a group of one or more containers with shared storage and network. The

[5] This can be done using an interface definition language such as Protocol buffers (https://developers.google.com/protocol-buffers) or Apache Thrift (https://thrift.apache.org)

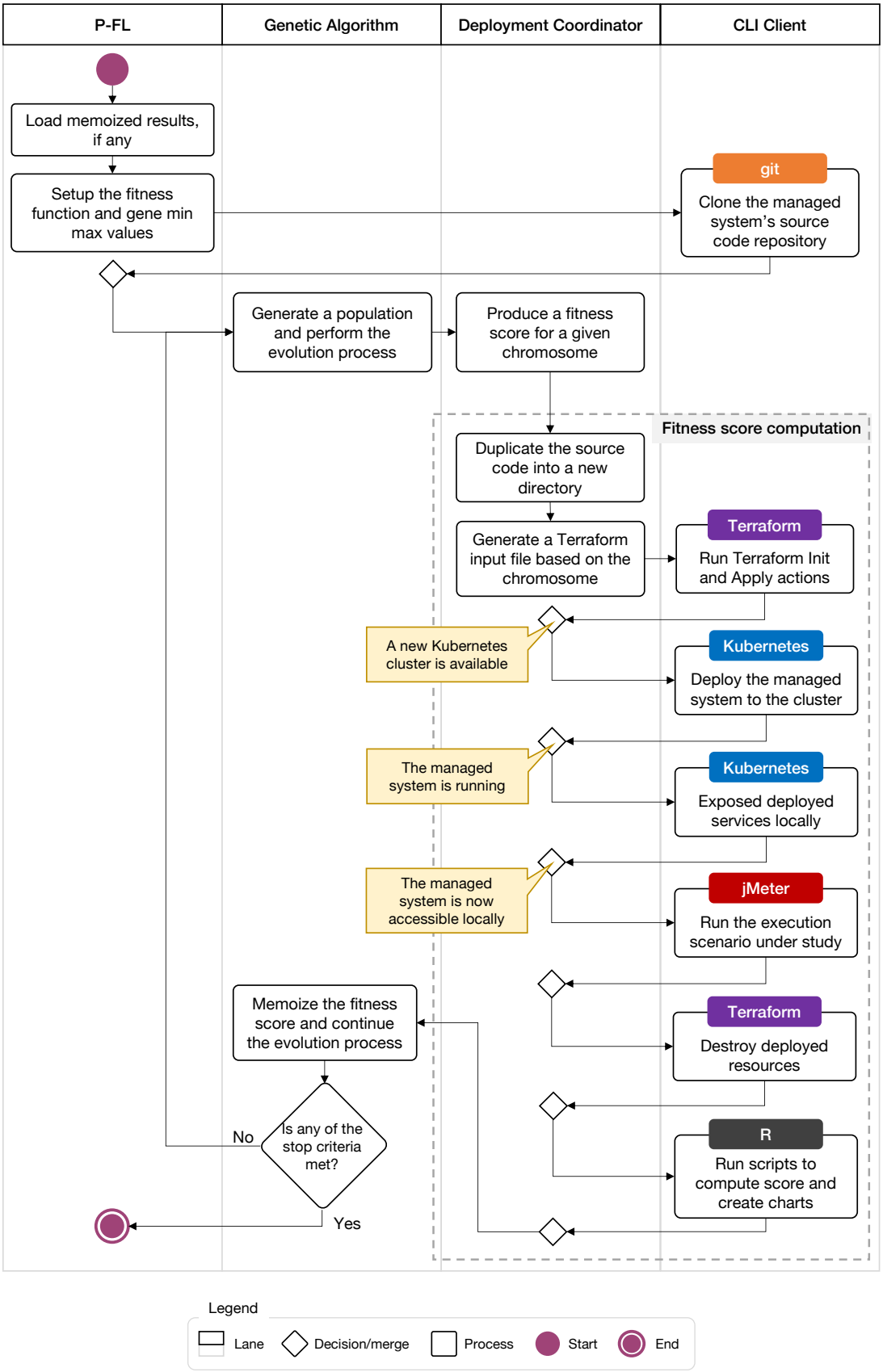


Figure 5.8.|: Experimentation workflow to devise, deploy and select infrastructure variants

right-hand side of the figure shows the same containers, now exposed through a service each (*cf.* frontend-svc and backend-svc). This allows exposing a single port while balancing requests through multiple pod instances (*cf.* frontend and backend). Figure 5.10 shows the same example application after having applied both Load Balancer and Producer/Consumer. The latter's implementation creates two adapter components based on the service interface. The first one takes requests produced by `API` and enqueues them into `queue`. The second one dequeues the requests and passes them to `Worker`. An implementation of Producer/Consumer without Load Balancer would group `API` and `Worker` with the corresponding adapter into a pod.

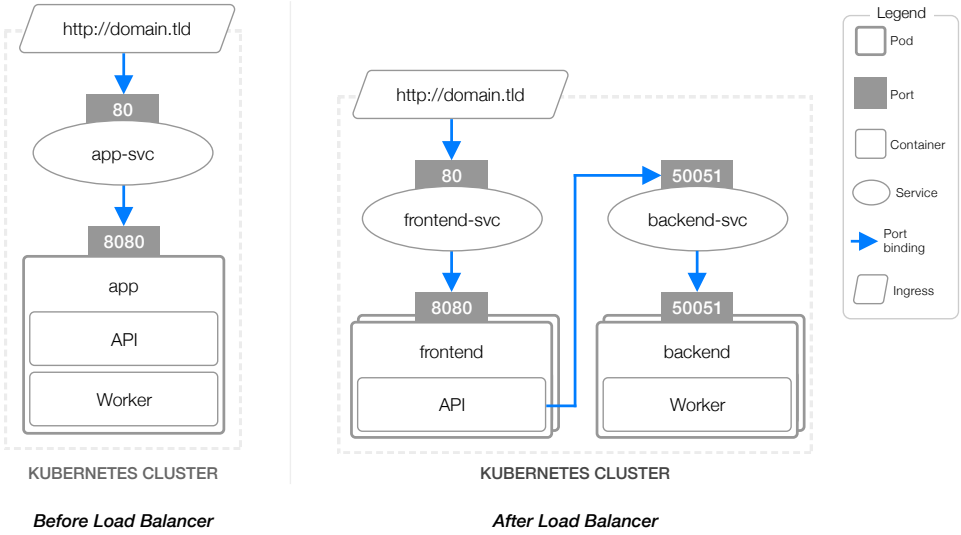


Figure 5.9.|: Deployment configuration before and after applying a Load Balancer

Our variant generation algorithm, developed as a possible strategy in this dissertation, is inspired by evolutionary computation. It defines design variants modeling important features as chromosomes (X) comprised of connection points (*i.e.*, genes (Γ)) between architectural elements (E). The left-hand side of a connection plays the role of a client and the right-hand side plays the role of a server. These two roles match those present in common design patterns (Φ). For example, for Producer/Consumer, Master/Worker and Reactor, the service providers (*i.e.*, Consumer, Worker and Handler) are the servers and the elements sending requests to them are the clients (*i.e.*, Producer, Master and Client); for a Load Balancer or any kind of Proxy, Decorator and Interceptor, the original client and server elements take the same role when the pattern is applied. Patterns may introduce new roles according to their structure, thus adding new architectural elements (*e.g.*, `queue` in Figure 5.10).

The variants generation is as follows (*cf.* Algorithm 2). First, an initial population of chromosomes is generated by applying each pattern to the connection points defined between the architectural elements (*cf.* line 1). Then, a chromosome crossover phase takes place following a single-point crossover strategy [**?**]. In this phase, two chromosomes are combined to create a new one (*i.e.*, a new variant). The new chromosome adopts the client from one of the originating chromosomes and the server from the other

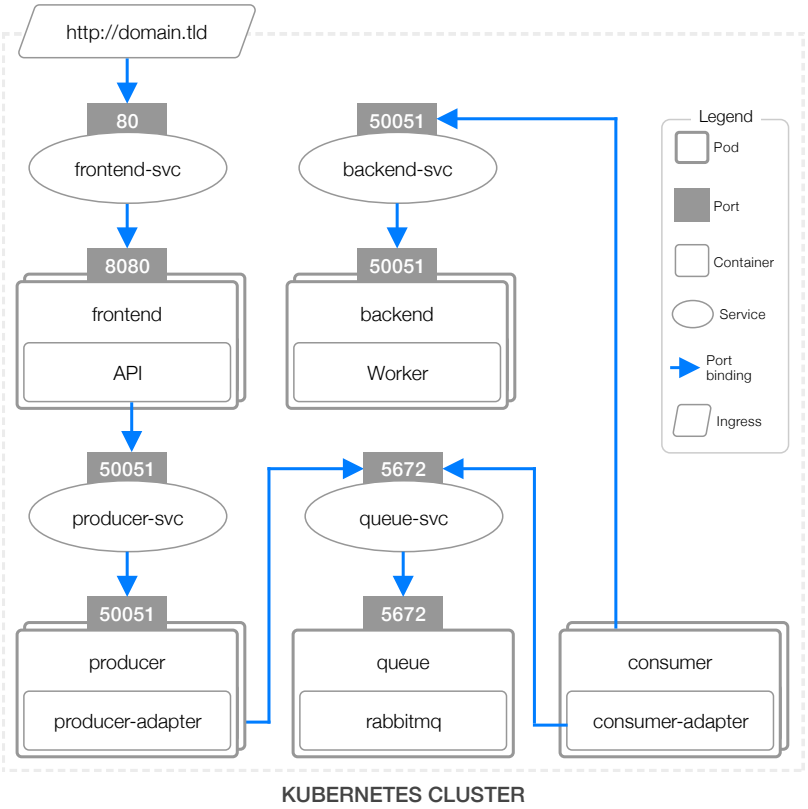


Figure 5.10.|: Deployment configuration using Producer/Consumer

one (*cf.* line 3). Subsequently, the algorithm mutates the chromosomes by introducing a new pattern in one of the genes (*cf.* line 13). For example, given the chromosome Client$\circ\rightarrow\circ$Server, a Proxy Cache can be added immediately after the client, thus resulting in (Client$\rightarrow$Proxy Cache)$\rightarrow\circ$Server. In a subsequent phase, the parenthesized elements represent a single connection point (*i.e.*, a gene), thus allowing to compose the patterns. Once the mutation phase is completed and also after each crossover, the algorithm evaluates the *fitness* of each chromosome (*i.e.*, the validity of each variant, *cf.* lines 4 and 14). The fitness function is a predicate that determines whether a particular variant is worth running. The predicate evaluates the following constraint types: *Exclusion*, which prevents equivalent patterns from being combined; *Precedence*, which prevents invalid pattern combinations; *Uniqueness*, which prevents a particular pattern from being applied multiple times; And *Location*, which constraints the position of a pattern in the chromosome.

The list below describes example constraints to evaluate whether a chromosome is valid.

Exclusion Either Master/Worker or Producer/Consumer can be included in a chromosome but not both.

Uniqueness Master/Worker, Producer/Consumer and Rate Limiting Proxy must be included at most once in a chromosome.

Location Rate Limiting Proxy can only be included right after Client.

Precedence Master/Worker and Producer/Consumer must not be load-balanced. That is, Any preceding Load Balancer must have a distance of at least 2.

Precedence Load Balancer must not be immediately preceded by Master/Worker or Producer/Consumer.

Algorithm 2: Design Variants Generation Algorithm

Data: $\Phi = \{\varphi_i \mid \varphi_i \text{ is a pattern}\}, E = \{\varepsilon_i \mid \varepsilon_i \text{ is an architectural element}\}, \delta \in \mathbb{N}$
Result: $X = \{\chi_i \mid \chi_i \text{ is a chromosome}; \chi_i \ni \{\gamma_{i,1}, \gamma_{i,2} \mid \gamma_{i,1}, \gamma_{i,2} \text{ are genes}; \gamma_{i,1}, \gamma_{i,2} \in \Gamma; i, j \in \mathbb{N}\}\}$

```
// Initialize the population
X = {χ_i | ∀χ_i ∃γ_{i,j}; PatternApplied(γ_{i,j}, φ_i)}
// Perform the crossover on the population
foreach χ_i, χ_j ∈ X; i ≠ j do
    childChromosome = {γ_{i,1}, γ_{j,2} | γ_{i,1} ∈ χ_i; γ_{j,2} ∈ χ_j}
    if IsFit(childChromosome) then
        X = X ∪ {childChromosome};
    end
end
// Apply mutations (combinations)
counter = 0
while counter < δ do
    foreach φ_i ∈ Φ do
        foreach χ_i ∈ X do
            foreach γ_{i,j} ∈ χ_i; j < |χ_i| do
                newChromosome = ApplyPattern(χ_i, γ_{i,j}, φ_i)
                if IsFit(newChromosome) then
                    X = X ∪ {newChromosome};
                end
            end
        end
    end
    counter++;
end
```

As an example, consider the deployment depicted in Figure 5.9. In this case, the generation algorithm was invoked with components Client, API and Worker. Client is a component we injected synthetically that represents a user of the API, thus materializing the connection between them. The variant in the figure represents the output (Client→LB)∘→API→∘(LB→Worker). Such a variant succeeds variants (Client→LB)∘→API→Worker and Client→API→∘(LB→Worker).

5.5. Chapter Summary

The rate at which operational contexts and software requirements change increasingly requires accelerating the evolution of development artifacts. However, software systems grow at an exponential rate [**?**, **?**], thus making it increasingly harder to match the pace of change. In practice, software changes emerging from the system execution are usually delayed in favor of fixing bugs and creating new software features, usually falling back to

infrastructure over-provisioning as an alternative. To make matters worse, the talent shortage in software-intensive industries only seems to grow [**?**, **?**]. The answer to this intricate problem is to balance *manual* and *autonomic* development work [**?**].

This chapter presented our self-improvement feedback loop for continuous architecture and infrastructure adjustment. Our feedback loop aims to eliminate a remaining discontinuity from the development process. By extending quality assessment with continuous improvement, it frees stakeholders from maintenance work and expedites the implementation of software changes. In other words, it realizes self-improvement at development-time. The exploration and analysis of potential improvements (*i.e.*, system variants) is based on what we call quality-driven experiments. This is a best-effort approach to eventual quality improvement. When a system variant outperforms the managed system, the self-improvement feedback loop proposes the corresponding code contribution through a continuous software evolution pipeline (*cf.* Chapter 4).

Following the separation of concerns principle, we decomposed our self-improvement feedback loop into three feedback loops with a reduced scope. The *Experimentation Feedback Loop* (E-FL) controls the satisfaction of high-level experimentation goals through online experiment design. The *Provisioning Feedback Loop* (P-FL) derives, deploys and monitors infrastructure configuration variants. And the *Configuration Feedback Loop* (C-FL) derives, deploys and monitors architectural design variants. Both the P-FL and the C-FL derive system variants based on evolutionary computing. In the first case, the P-FL experiments with configuration parameters of declared virtual resources. In the second case, the C-FL experiments with domain-specific design patterns to create distinct arrangements of architectural components. Overall, the design of experiments, the generation of system variants and the update of software artifacts are driven by MARTs.

Now that we have presented our approach for realizing self-evolution from a software engineering point of view, we shift the perspective to control theory. The next chapter presents our run-time evolution reference architecture for dependable autonomy and operational resiliency.

Chapter 6

Run-Time Evolution Reference Architecture

Contents

This chapter addresses self-regulation as introduced in Chapter 3. We focus on realizing self-evolution from a control theory perspective (*cf.* Section 3.1.2). Although the challenges discussed in this chapter manifest in various types of systems, including cloud software applications, they are best exemplified in the context of large-scale CPS. Given their complexity and, more importantly, the natural properties of real-world entities and the implications of run-time adaptations on them, CPS pose quality considerations critical for their operational success. Therefore, we base our contribution on CPS.

The proliferation of SCPS is increasingly blurring the boundary between physical and digital properties of real-world entities. SCPS are engineered systems with augmented reasoning, learning, control, and autonomic capabilities to manage entities and processes in deeply intertwined virtual and physical environments [**?**]. As these systems continue

permeating the whole human activity spectrum, their operational contexts and interactions with users and surroundings become increasingly hard to anticipate fully [**?**, **?**, **?**]. SCPSs's inherently dynamic nature demands abilities to accommodate to the physical environment's run-time complexity and uncertainty in the short and long term [**?**, **?**]. This refers to the ability to self-adapt and self-evolve to deal with unpredictable change [**?**]. However, the potential impact of such self-driven actions on uncertain environments calls for their evaluation prior to materialization. Consequently, the design of SCPS must strive for dependable autonomic capabilities, while ensuring operational resiliency.

To provide guarantees about potential adaptations and evolution actions, SCPS require robust and accurate representations of controlled physical entities and their interactions with the environment. However, the intrinsic highly dynamic nature of physical entities challenges SCPSs' competence to mirror and predict significant characteristics effectively throughout prolonged operation. Inevitably, physical entities associated with SCPS are subject to evolutionary and emergent behavior [**?**] from their interactions with their environment, resulting in transformations (*i.e.*, evolution) of the dynamics of the entities themselves. Therefore, SCPS designers ought to incorporate mechanisms to manage the dynamicity of associated entities and processes, with at least a minimum of operational guarantees.

We envision a continuous engineering cycle where adaptation and evolution work cooperatively to achieve the system goals. In this engineering cycle, continuous evolution plays both a reactive and proactive role explicitly. Its main purpose is to improve the reference models used for controlling the system's operation, thus, ultimately contributing to the managed system's long-term evolution. Therefore, in this chapter, we present our contribution to run-time evolution conceived as a reference architecture for guiding the design of dependable and resilient CPS. The proposed architecture realizes our envisioned engineering cycle as follows. First, it uses evolutionary optimization and online experimentation, building on the premises presented in Chapter 5, to find suitable models to guide the adaptation of the managed system. Second, it uses online experimentation, supported by parameter optimization, to gather evidence of statistically significant improvements over the system's baseline design, thus generating knowledge of possible configuration states and their respective performances. Lastly, our reference architecture accommodates self-evolution for reflecting persistent changes in the physical environment as well as improving the use of resources. Self-adaptation is used to ensure that the managed system operates within acceptable and viable boundaries. Together, these characteristics contribute to achieving dependability and ensuring resiliency for SCPS at run-time.

Dependability and resiliency of SCPS are commonly approached from a cyber-security standpoint. However, an aspect often overlooked is the natural evolution of physical entities external to the system that affect its day-to-day operation. Thus, changes on these entities eventually render adaptation mechanisms irrelevant. Our contribution addresses dependability and resiliency through the design of run-time evolution mechanisms that deal with uncertain operation conditions.

The remainder of this chapter is structured as follows. First, Section 6.1 contextualizes

our reference architecture into the overall solution strategy presented in this dissertation. Second, Section 6.2 introduces our reference architecture for designing SCPS enabled for dependable and resilient run-time evolution.

Correspondences in This Chapter

Addressed Challenge(s): CH6—Autonomic managers are required to keep their internal models relevant, aiming to reduce the need for additional maintenance; *Question(s)*: Q7—How can autonomic managers help reduce maintenance work stemming from emerging behavior? *Goal(s)*: G4—Define required design elements to expedite run-time changes when facing emerging behavior; *Contribution(s)*: C3—Run-time evolution reference architecture.

6.1. Overview of the Reference Architecture in Our Solution Strategy

Although sometimes used interchangeably among application domains, system adaptation and evolution are worth exploring and analyzing individually in the context of SCPS. On the one hand, system adaptations are usually considered as control changes to correct deviations from anticipated behavior or to achieve operational goals. Largely, system adaptations are reactive and concern short-term changes during sudden periods of manageable uncertainty. On the other hand, system evolution involves protracted changes that are usually discovered through proactive exploration of alternative system structures and behaviors. Evolution actions usually aim to optimize the system's operation, reflect permanent changes in the system's usage patterns or mitigate potential adverse conditions.

In this chapter, we propose the combination of adaptation and evolution as way of handling short-term and long-term changes in the system's environment. Evolution actions are intended to regulate the MARTs guiding the overall operation, whereas adaptations focus on meeting the control objectives in the short-term. Section 6.1.1 describes the relevance of evolution and adaptation in CPS in terms of dependable autonomic behavior, as well as resilient system operation. Section 6.1.2 explains the need for a more relaxed contract between managed and managing systems, leading to self-evolution as a way to regulate and improve the system. Finally, Section 6.1.3 describes how the mechanisms for self-regulation fit into our offline and online reconceptualization.

6.1.1. Dependability and Resiliency in Autonomic Cyber-Physical Systems

Increased autonomy in CPS results in more uncertainty and thereby more risks. First, internal structures and behavioral models of the managing system may become unsuitable: since elements of the managed system evolve over time, at some point adaptation plans could be synthesized based on inaccurate estimations or outdated MARTs. Consider our

SUTS case study, introduced in Chapter 3, as an example. User demand (*i.e.*, daily passengers) and usage patterns (*i.e.*, demand levels for a particular line at specific day times) may change temporarily and even permanently. Although many of these changes can be anticipated, such as scheduled events in the city, national holidays or construction projects, many others are fortuitous, such as traffic accidents, protests or bad weather conditions. However, even if these changes can be anticipated, there are no guarantees about their effects or progression over time. Second, managing systems can only guarantee their effectiveness over supported context attributes. Therefore, even if the system is under regular operation, adaptation plans can negatively impact the system's operation. This is because there can be confounding factors unknown to the managing system. Third, small changes can have a great impact on complex systems. Predicting the outcome of an adaptation plan using theoretical models can be futile on a complex system. A first reason is that it is virtually impossible to perform an exhaustive exploration of all possible scenarios; a second one is the impossibility of models to capture the complexity of the managed system and its environment reasonably.

One mitigation strategy for these risks is to involve humans in the adaptation planning process. Human operators are knowledgeable on the different parts of the system, being skillful in assessing adaptation alternatives. Nevertheless, involving humans is costly and introduces the constant possibility of errors, therefore justifying the need for engineering reliable *autonomic* managers. Moreover, human expertise may not be enough in the face of complex system dynamics. For these reasons, we consider the aforementioned risks as relevant quality concerns. Thus, we delegate the task of accounting for these concerns as architecturally significant quality attributes to the managing system.

We group the aforementioned concerns into two quality attributes: *dependability* of the autonomic behavior, and *resiliency* of the overall operation. For the first attribute, we consider necessary to have a trustworthy, evidence-based adaptation planning process. We aim to ensure continuity and readiness of service (*i.e.*, reliability and availability, respectively). For the second attribute, it is important to provide the system with the necessary means to recognize when it is operating outside of the control objectives' viability zones. That way, the managing system can either optimize control parameters, or take preemptive measures to increase the likelihood of goal fulfillment. Additionally, a run-time verification and validation technique might be necessary to reduce the effect of unidentified confounding factors.

6.1.2. Adaptation and Evolution as Autonomic Operational Modes

In classical control theory, a controller synthesizes control actions from, and instruments them into, a controlled system for achieving its operational goals. The controlled system lacks any notion of adjustment review, acceptance or rejection because, by design, it is unaware of the controller (*e.g.*, a thermostat). Autonomic managers perform a similar task in the case of software systems, with the difference that they not only control but manage the system's operation [**?**]. Examples of such management capabilities are self-healing, self-protection, self-configuration and self-optimization [**?**]. Although they are

conceptually more advanced than classical controllers, their relationship with the managed system is also based on controlling its execution. Both classical controllers and autonomic managers interface with the managed system through sensing and effecting elements only. In both cases, their objective is to measure and adapt the managed system's behavior.

Short-term and frequent adaptations, typical of classical control systems, are not suitable for all CPS. Consider our case study as an example. Adjusting the headway design or the operating fleet size in short periods of time (*e.g.*, within a few minutes) can be unproductive, ineffective and expensive. Moreover, frequent adaptations may lead to instability, especially if the frequency of the changes causes a long settling time (*i.e.*, the time required for an adaptive system to achieve a desired and steady state) [**?**]. Since the managing system is likely to consider a limited set of context attributes, unforeseen events may destabilize the managed system further (*e.g.*, road blocks caused by protests or traffic accidents). Moreover, an adaptation plan may include manual tasks, thereby rendering control mechanisms unfit. Oftentimes, humans in the loop must evaluate, or at least review, the adaptation plan to prevent expensive mistakes. Therefore, the managing system must be able to provide evidence for the predicted outcome to instill confidence in the adaptation plan [**?**, **?**, **?**]. These situations do not disregard the need for autonomic control, but evidence the need for a more relaxed contract between managing and managed systems.

While classical controllers are focused on short-term and reactive adaptations, quality properties of large-scale CPS demand management mechanisms for long-term evolution as well. We conceive this duality—short-term adaptation and long-term evolution—as two operational modes of an autonomic manager. On the one hand, the adaptation operational mode retains the control relationship described previously. On the other hand, the evolution operational mode relaxes the frequency and mandatory nature of the adaptations in favor of eventual evolution. That is, the process of regulating and improving the managed system as opposed to controlling its operation. Moreover, such a process shifts the focus from short-term, non-exploratory and prescriptive adaptations to long-term, exploratory and dynamic adaptations.

Beneficial discoveries found through the evolution operational mode eventually manifest in the managed system, thus completing the evolution cycle between managed and managing systems (*i.e.*, managed system $\rightleftarrows$ managing system). As the managed system evolves naturally in the real world, so do the managing system's internal models (*i.e.*, evolution in the forward direction). And whenever the managing system synthesizes an improvement, the adaptation operational mode will enforce the necessary changes for its managed system to accept or reject them (*i.e.*, evolution in the reverse direction). Furthermore, associated causal connections will also trigger updates in the managed system. This evolution cycle enables continuous engineering throughout the entire system life cycle, even at different levels of granularity. From the managed to the managing system, humans in the loop frequently deliver updates. In the opposite direction, the managing system exploits data that is usually unavailable offline to optimize the use of system resources. This includes, for instance, data from the actual system operation. Furthermore, it is the combination of data devised during development, stakeholder knowledge, and run-time data that would allow CPS reach other—richer—levels of autonomic behavior.

6.1.3. Self-Regulated Evolution of Run-Time Artifacts

The reference architecture we present in this chapter relies on the self-improvement feedback loop we presented in Chapter 5. This chapter concentrates on the use of dynamic simulations to try out distinct configuration scenarios as a response to emerging behavior. The purpose of using the self-improvement feedback loop at run-time is to reify long-lasting descriptive and predictive models to guide short-term adaptations. For example, whenever the usage demand pattern changes for a particular system, the managing system's internal demand model must be updated. In this case, the demand model is a reference control input that should remain relevant for correctly controlling the system's operation.

Figure 6.1 depicts how self-regulation fits into the time-of-change timeline. Configuration explorations based on historical data are executed at development-time, and then complemented with data collected during the operation of the managed system. Descriptive and predictive models reified during development are updated in response to emergent behavior. We achieve this through adaptive control. More specifically, we realize the two operational modes described in Section 6.1.2 through architectural components inspired by MIAC and MRAC.

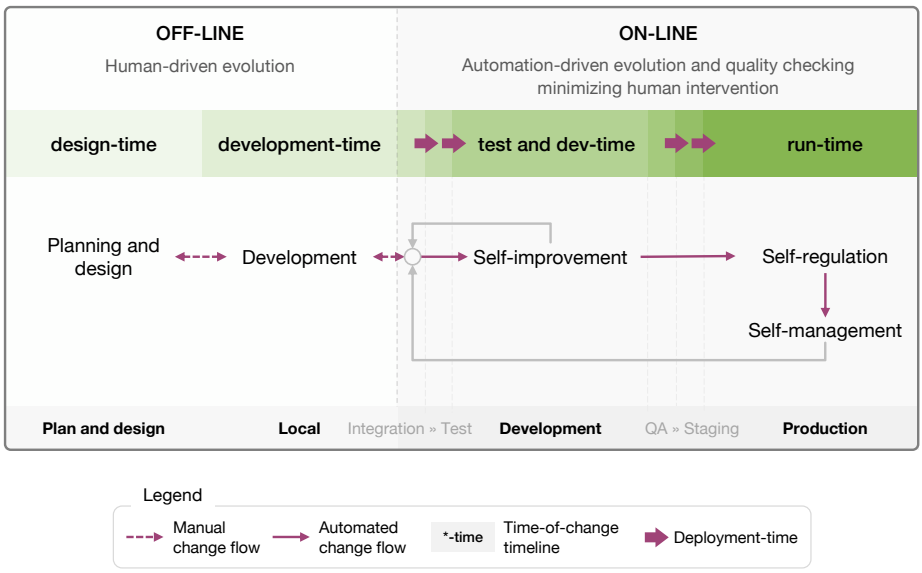


Figure 6.1.|: Self-regulation based on adaptive control

6.2. Architecture Overview

This section introduces our reference architecture for dependable and resilient CPS. We follow the *Views and Beyond* documentation approach [**?**] to describe the elements of our architecture and their relationships. We chose to depict these elements with decomposition views (*cf.* Figures 6.2, 6.3 and 6.5) and a behavioral view (*cf.* Figure 6.6), using an informal notation.

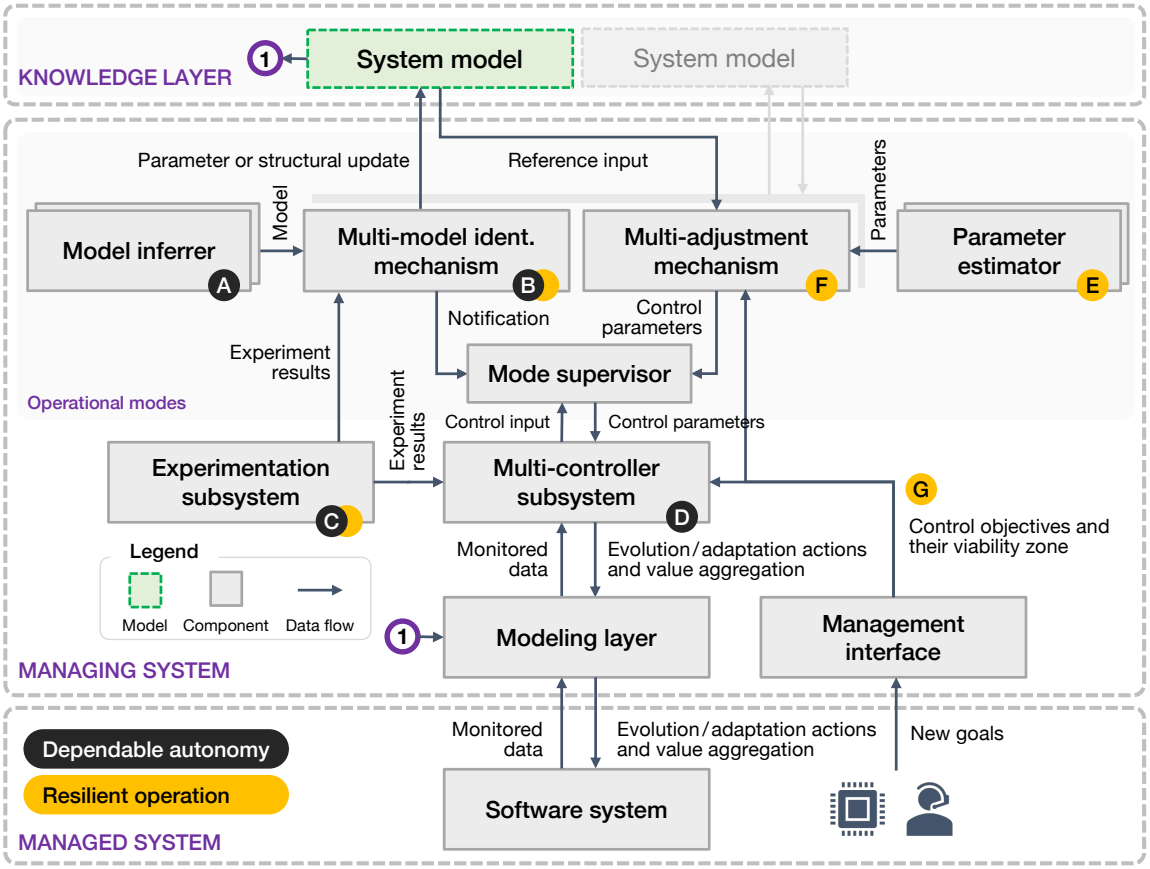


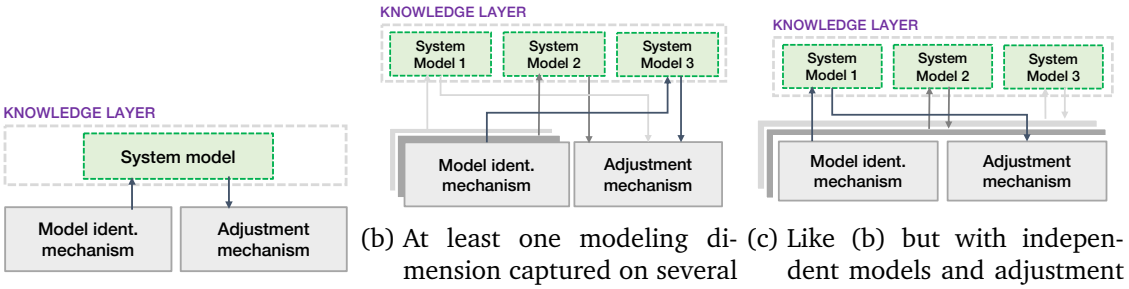
Figure 6.2.|: The proposed architecture
Labels A through G mark the components that provide support for dependable autonomy (●) and resilient operation (●). Labels B● and C● support both quality attributes.

Figure 6.2 is split into three main subsystems: a common knowledge layer, which comprises data sets and MARTs; a managing system, which comprises elements of an autonomic manager; and a managed system, which refers to the CPS. Information flowing between the managing and managed systems passes through a modeling layer—an interface to managed resources, regardless of whether they are physical, virtual or conceptual (*e.g.*, a bus line). This component is responsible for both mirroring the managed system's state to its managing system, and also realizing causal connections from MARTs to the managed system. Effector components connected with our modeling layer translate changes in the models into updates to the managed system. We call this evolution in the reverse direction (*i.e.*, managed system ← managing system). Conversely, We call evolution in the forward direction (*i.e.*, managed system → managing system) to collecting, filtering, aggregating and analyzing sensed data to keep the MARTs up to date with respect to the current managed system's state.

The concerns described in Section 6.1.1 manifest themselves in three main subsystems: A multi-controller subsystem, a model identification mechanism (Model Identification Mechanism (MIM)) with support for multiple model inferrers, and an adjustment mechanism (Adjustment Mechanism (AM)) supporting multiple parameter estimators (*cf.*

D, B, and F, respectively in Fig. 6.2). The last two form what we call a dimensional control adapter.

Figure 6.3 shows three possible configurations of a dimensional control adapter. The first configuration considers one modeling dimension captured in a single MART (*cf.* Figure 6.3a). Examples of modeling dimensions include user demand, bus arrivals, and topology of lines and stations. The AM uses the MART as a reference input to adjust the controller's parameters, while the MIM updates it when necessary. In the context of our case study, a MIM updates a probability distribution function of bus arrivals for a particular bus line and specific bus stop, whereas an AM uses it for calculating an appropriate headway design. The second configuration considers at least one modeling dimension captured on several MARTs (*cf.* Figure 6.3b). In this case, an AM uses them as reference inputs, while several MIMs update them when necessary. The third configuration also considers at least one modeling dimension captured on several MARTs (*cf.* Figure 6.3c). In this case, there is a dimensional control adapter for each of them.



(a) One modeling dimension

(b) At least one modeling dimension captured on several models

(c) Like (b) but with independent models and adjustment mechanisms

Figure 6.3.|: The dimensional control adapter

Sections 6.2.1 and 6.2.2 further describe the elements of our architecture under the light of quality attributes dependability and resiliency. And Section 6.2.3 details relevant aspects of the knowledge layer we propose.

6.2.1. Achieving a Dependable Autonomy

Our architecture addresses dependable autonomy in multiple ways (*cf.* labels A through D in Figure 6.2) as follows.

6.2.1.1. Error mitigation through multi-model identification (A)

Model identification, inference or estimation are useful techniques to obtain a model given an initial data set of historical data. Nevertheless, while the generated data is useful, it is not always accurate. By implementing several techniques, the MIM can either select the model with the smallest estimated error, or combine them as shown with the multi-controller approach (*cf.* Section 2.2.1). Examples of model inference are function approximation using interpolation or symbolic regression. Regarding our SUTS case study, by

using a queuing network model of a Bus Rapid Transit (BRT) system, a MIM could simulate many headway design values, thus obtaining their corresponding excess waiting times. Then, it could use symbolic regression to find a function using the known data points.

6.2.1.2. Reliable models through model inference (B)

The MIM realizes the evolution operational mode. The `mode supervisor` component monitors the control objectives and their viability zones, looking for a drift between reference and measured values. Whenever a variable of interest goes from an inaccurate state (*i.e.*, mild drift from the expected value) to an unknown state (*i.e.*, the model cannot reliably predict future values), the MIM is activated to synthesize an improved model. By evolving the MARTs, the MIM keeps the control parameters relevant, thus contributing to the dependability of the autonomic manager.

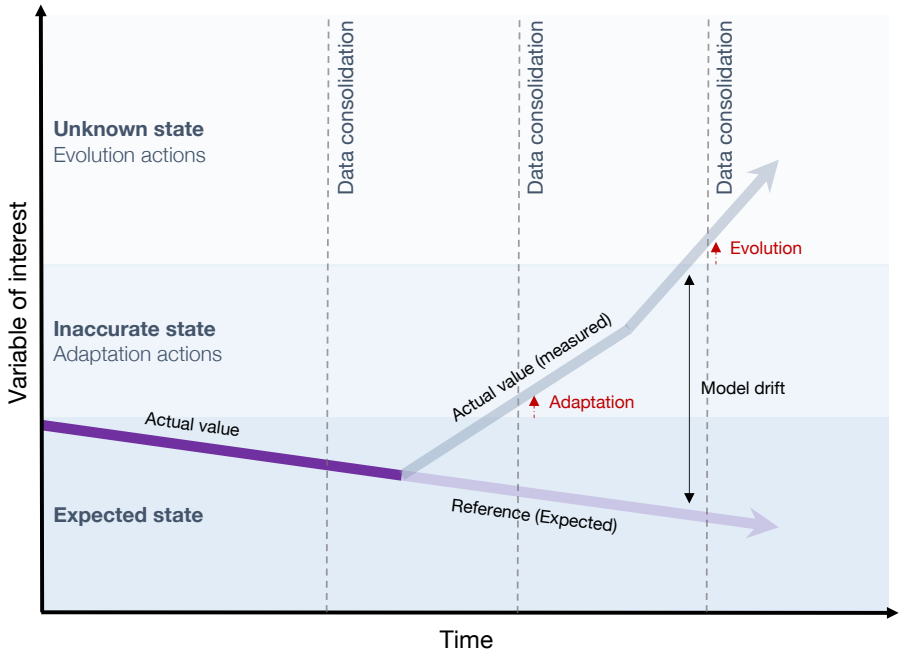


Figure 6.4.|: Viability zone for a particular variable of interest

Figure 6.4 illustrates the relationship between referenced and measured values with respect to the system state. The chart depicts the progression of a variable of interest, such as the user demand after a peak hour for a particular bus line and station. Right before the second data consolidation, the mode supervisor identifies a mild drift between the expected and actual values. When there is enough data available, the mode supervisor predicts further drift based on historical data and domain knowledge. Whenever the drift between these values is unacceptably large, the number of control actions increases (*i.e.*, adaptations). For example, when the value is expected to be strictly decreasing but it starts to increase. However, when the measured value enters an unknown state, that is, the reference value is no longer reliable, the mode supervisor switches from the adaptation to the evolution operational mode.

6.2.1.3. Evidence collection through experimentation (C)

Evidence-based decision making is an effective way of mitigating the risk of unwanted effects. The MIM uses online experimentation to design and conduct experiments. This allows the MIM to test multiple models by varying input parameters of the particular method being used. This is known as hyperparameter optimization, and is used in techniques such as gradient-based and evolutionary optimization. Figure 6.5 depicts the experimentation subsystem, and Figure 6.6 illustrates a detailed view of the data flow. A client component requests to the experimentation subsystem to design and conduct an experiment, passing a set of treatment functions and an experimentation goal as parameters. The `experimentation feedback loop` creates an experiment following the workflow presented in Chapter 5. Then, the `configuration feedback loop` uses the functions to devise different groups (*e.g.*, control and treatment groups), and deploys them to the `experiment execution environment` according to the design. The target subjects may reside in various types of environments, including a simulation, a computing platform or the real world. Each deployed artifact is considered to be an independent system instance, therefore it provides sensing and predicting capabilities. The configuration feedback loop measures and aggregates metrics of the variable of interest. The experimentation feedback loop then uses these metrics to conduct statistical tests (*e.g.*, normality and analysis of variance) and hypothesis testing. Then, it can decide whether the experiment has achieved its goal or needs to be adapted. Once the experiment is terminated, the client receives a list of subject groups sorted by a particular metric of interest, and clustered based on similarity.

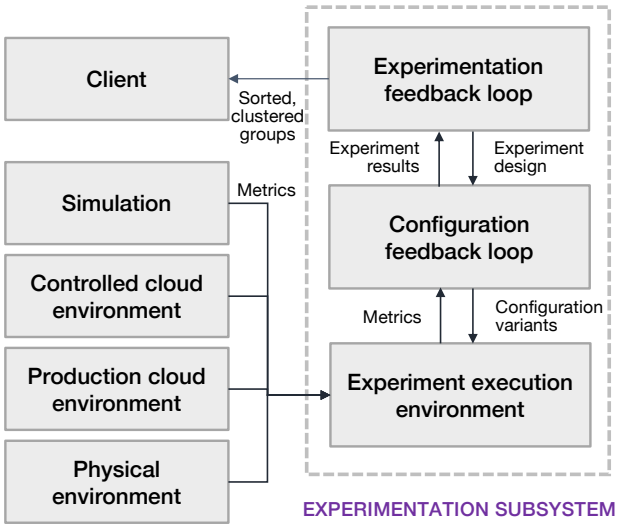


Figure 6.5.|: The experimentation subsystem

6.2.1.4. Autonomic behavior through adaptive control (D)

The `multi-controller subsystem` component in Figure 6.2 is inspired by multi-model adaptive control. Similarly to the MIM, its main purpose is to minimize the estimated error

when adapting the managed system. Since this subsystem is application-specific, we do not provide further details about it in this section.

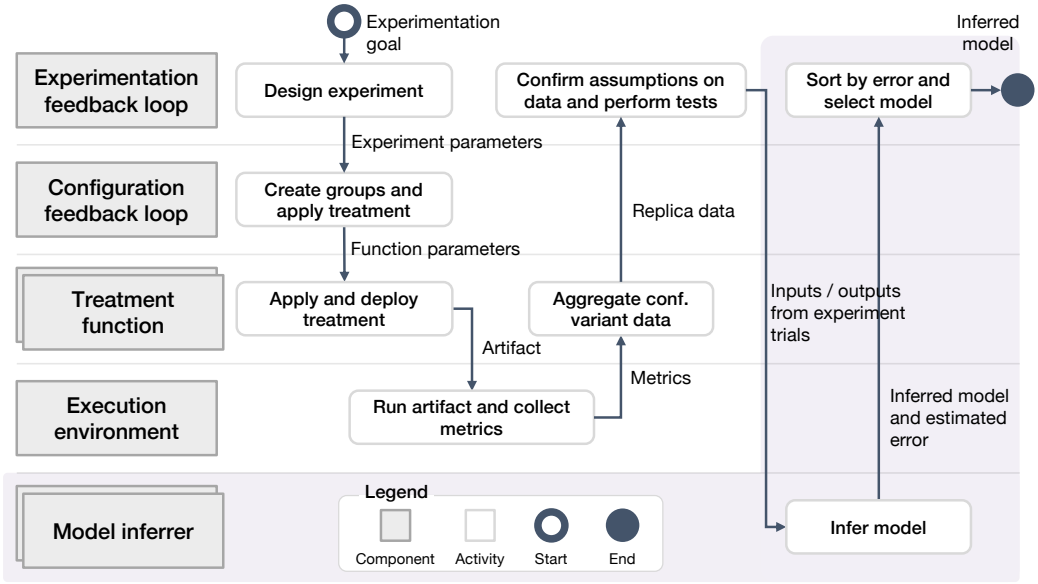


Figure 6.6.|: Model identification workflow

Figure 6.6 depicts interactions between the experimentation subsystem and the model identification mechanism. The shadowed area contains components and activities that exclusively belong to the latter. The rest of the figure shows an alternative view of the experimentation subsystem (*cf.* Figure 6.5).

6.2.2. Achieving Resilient Operations

Our architecture ensures resilient operation through multiple techniques (*cf.* labels B, C, E, F and G in Figure 6.2) as follows.

6.2.2.1. Predictable adaptation through reliable models (B)

Up-to-date models allow the AM to plan adaptations based on both relevant inputs and criteria, thus increasing its resiliency in the face of uncertainty. On the one hand, the AM optimizes control parameters based on reference inputs, thereby ensuring that the controller will work within a known scope and will be able to satisfice the control objectives. On the other hand, the multi-controller subsystem adapts the managed system's behavior based on viability zones that reflect changing conditions instead of fixed constraints. Consider the example in Figure 6.4. If the corresponding MART is updated right after the third data consolidation, the operation could be optimized for the current conditions. In case the control objectives are not met, the managing system would desist of the task and delegate it to human stakeholders. In any case, the system is aware enough to decide which path to take.

6.2.2.2. Run-time validation through evidence collection (C)

Run-time validation and verification are required to reduce the likelihood of adverse effects when adapting the managed system. Our architecture considers a form of run-time validation through continuous experimentation. Instead of deploying a complete adaptation plan, the multi-controller subsystem coordinates with the experimentation subsystem to adapt the managed system partly and assess whether it behaves as expected. For example, instead of updating the headway design for all the bus lines in the system, only a representative group of lines (*i.e.*, the treatment group) is updated. The experimentation subsystem will run statistical tests to confirm that the adaptation plan yields the predicted outcome. The multi-controller subsystem will gradually roll out the adaptation plan over time until it has been completely deployed.

6.2.2.3. Error mitigation through parameter estimation (E)

By considering multiple parameter estimators, the `multi-adjustment mechanism` mitigates various risks: non-convergent optimizations, inaccurate estimations, and erroneous estimations. Similarly to the MIM, the multi-adjustment mechanism can either select the estimator with the smallest error or combine them (*cf.* Section 2.2.1). The effect on the overall adaptation process is that the AM is less likely to fail in estimating a control parameter, thereby making the estimation more resilient to uncertain conditions. Examples of a parameter estimator are Bayesian, gradient-based and evolutionary optimization.

6.2.2.4. Goal achievement through hyperparameter optimization (F)

The AM aims at estimating control parameters based on reference inputs. Its main purpose is to aid in achieving the control objectives. For example, a gradient-based optimization method, such as gradient descent, can minimize the distance between a given setpoint (*i.e.*, the goal, as defined in the control objective) and a point evaluated in a known function. For example, an excess waiting time of 5 minutes for a particular line and stop would be possible with a headway design of 3 minutes, for a fixed number of buses. In this case, the method found that when the headway design is 3 minutes, the distance between the actual excess waiting time and the setpoint is minimal. Nonetheless, such a headway is of course highly costly, and it would be affordable only at highly peak times of service.

6.2.2.5. Assurance through viability zones and control objectives (G)

Viability zones are usually conceived as zones whose borders define the threshold between the regular operation of a system and when an adaptation is necessary (*e.g.*, [**?**]). Our reference architecture extends the notion of a viability zone with an additional threshold, illustrated in Figure 6.4. This new threshold delimits the managing system's effectiveness scope with respect to a particular variable of interest. Such a scope may be defined in

terms of known contextual situations, availability of adaptation strategies, or limitations of the reference models. For example, consider a machine learning model trained on data for which the current value of a variable would be considered an outlier. Predictions produced by such a model for the current value are not reliable. Similarly, if the reference model being used was inferred by function interpolation and new values of the variable are outside the domain of the original data set, the model cannot produce reliable predictions. The new threshold separates the need for (short-term) adaptation and (long-term) evolution. Therefore, our notion of a viability zone, together with the MIM and AM, provides run-time assurance even when the controller falls short at certain contextual situations. Consider our SUTS case study as an example. After a peak hour, the reference model may be expecting a strict decline in user demand. If there is a sudden increase in demand, the measured variable crosses a first threshold, thus assigning the system an inaccurate state (*i.e.*, there is a drift between the expected and actual values). If the demand continues to increase, the variable will eventually cross a second boundary, assigning the system an unknown state (*i.e.*, the reference model is no longer reliable). In this case, a demand identification mechanism performs the corresponding statistical tests to identify the new probability distribution, ultimately updating the reference model. At this point, the AM along with the controller will enforce a new operational state, thus returning the system to an expected state.

6.2.3. Realizing the Knowledge Layer

Our reference architecture realizes self-evolution from a control theory perspective. This means that MARTs are inferred at run-time, in response to emerging behavior in the environment. A perspective missing in the chapter so far is the integration of our architecture with our software evolution pipeline and our self-improvement feedback loop. Without this, autonomic managers built according to our architecture would be isolated from other computing environments in the delivery pipeline. We identify two key advantages justifying such an integration. On the one hand, exploring configuration scenarios can be a time consuming activity, depending on the target environment to where configuration alternatives are deployed. If the search space exploration is conducted entirely at run-time, the MIM may not be cost-effective or adaptations may not be timely. On the other hand, integrating our reference architecture with our evolution pipeline assists autonomic managers in realizing evolution in the reverse direction (*i.e.*, managed system $\leftarrow$ managing system). That is, optimized operation parameters can be used to update MARTs, and in turn, trigger persistent updates on the development side—or physical side, for CPSs.

The use of knowledge is indeed a common aspect between our continuous software evolution pipeline, our self-improvement feedback loop, and the reference architecture we present in this chapter. In Chapter 4, managing systems use MARTs to mirror the managed system's state and synchronize development artifacts. In Chapter 5, feedback loops take advantage of causally-connected MARTs to produce system variants as a way to improve the managed system. And in this chapter, managing systems reify predictive models to regulate control actions. Together, these contributions embody a holistic framework thus

far based on change. Despite knowledge playing a relevant role in evolving, improving and regulating the managed system, its use remains scoped to each specific environment. Knowledge plays an equally relevant—and sometimes overlooked—role in orchestrating feedback loops. In fact, it is through both change and knowledge that the offline and online parts of our envisioned evolution process interface (*cf.* Figure 3.4).

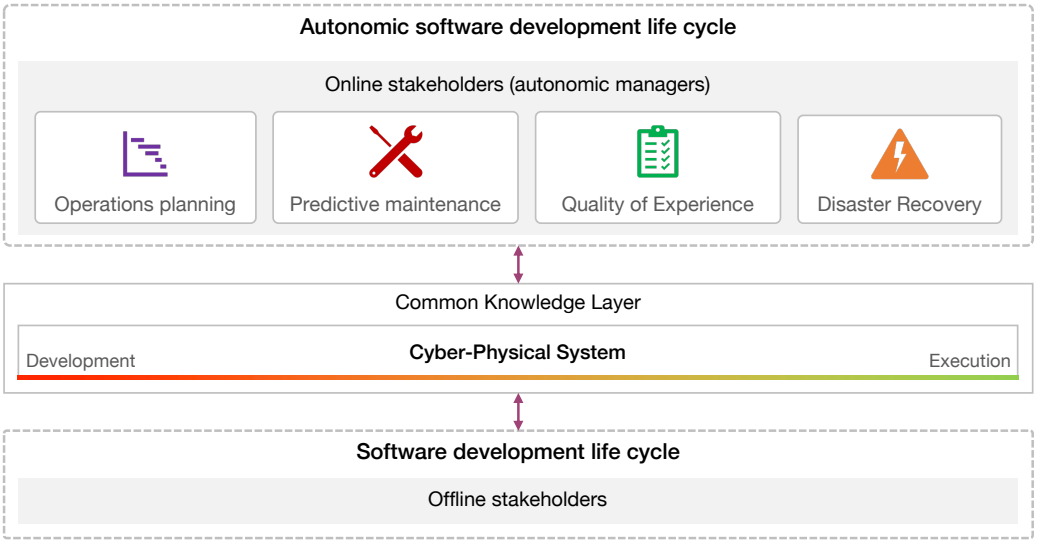


Figure 6.7.|: Knowledge layer to support stakeholder interactions beyond change

This section unifies knowledge aspects of our reference architecture through knowledge modeling and reification. We create a separate layer for storing relevant measurements and MARTs. In terms of the MAPE-K model, this means that we separate the knowledge element from the rest—much like cognitive processes in the human brain (*e.g.*, memory and reasoning). Thus, we create a common knowledge layer, external to the managing system. In general terms, we aim to promote collaboration between humans in the loop and autonomic managers, which we consider offline and online stakeholders, respectively. Figure 6.7 illustrates our vision. Each autonomic manager at the top of the figure concentrates on a specific aspect of the managed system. Both offline and online stakeholders obtain and augment data sets from the knowledge layer, throughout the delivery pipeline. Offline stakeholders can benefit from knowledge produced online, for example, by augmenting their tools with relevant information (*e.g.*, Monitoring- and performance-aware Integrated Development Environments (IDEs) [**?**, **?**]). By doing so, they shorten the feedback cycle from development to production on execution issues (*e.g.*, predicting relevant metrics based on changes to the OSP). Online stakeholders can benefit from knowledge produced offline, as well as by knowledge produced by other online stakeholders. In the first case, offline stakeholders can clean collected data, identify data relationships and create behavioral models. In the second case, each autonomic manager explores a portion of the evolution space, based on the concerns they address. At the same time, online stakeholders can focus on a single task, and rely on others to perform associated tasks. For example, an autonomic manager focused on operations planning can update the managed system at run-time, while another one focused on online evolution handles corresponding persistent updates. This integration is based on causal relationships among the MARTs rather than

dependencies among the autonomic managers.

The rest of this section is organized as follows. Section 6.2.3.1 details how the common knowledge layer fits into the delivery pipeline. Section 6.2.3.2 concentrates on the continuous engineering cycle we realize in our reference architecture for synthesizing predictive knowledge online.

6.2.3.1. Implicit and Explicit Knowledge Processing

We conceive the knowledge layer as a data store and processing infrastructure for MARTs. It spans throughout the delivery pipeline, starting from the local development environment and expanding to production. Along the pipeline, offline and online stakeholders get data and MARTs from the knowledge layer to augment their tools and own datasets. Moreover, since each stakeholder is concerned about distinct requirements, they will feed back valuable knowledge that can later complement another stakeholder's work. For example, an autonomic manager focused on achieving quality of experience goals will monitor passenger arrival times and associated metrics. This information is tremendously useful for a planning agent working from the operations control center. Similarly, an autonomic manager focused on operations planning updates bus schedules according to changes in the OSP. Changes to these data are input to an autonomic manager focused on predictive bus maintenance, which takes advantage of the updates to book necessary revisions without impacting overall system performance.

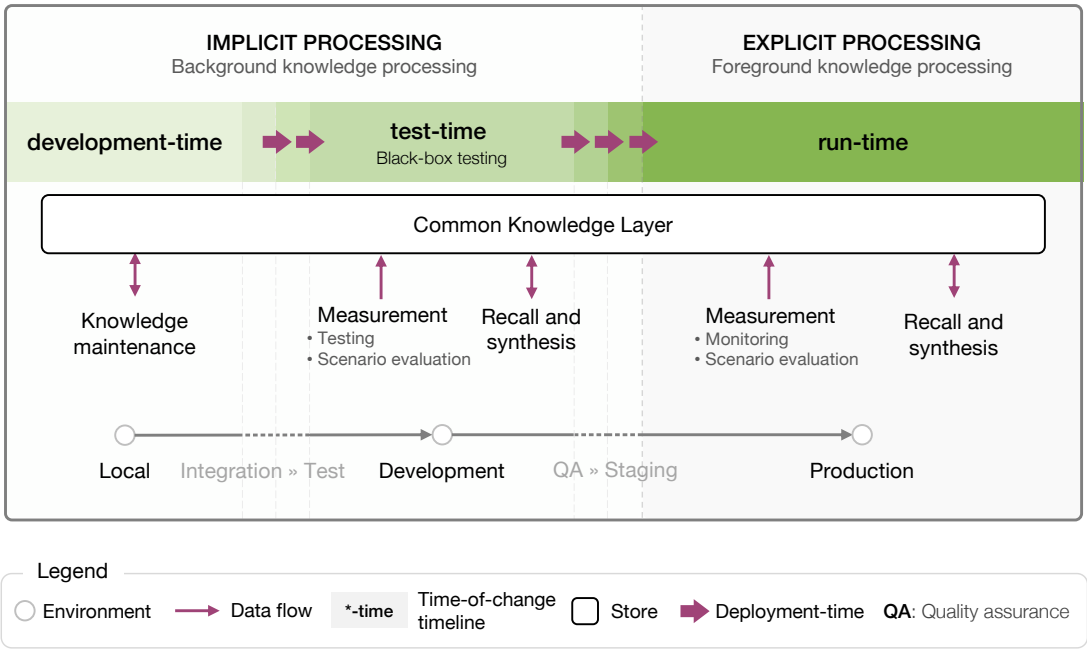


Figure 6.8.|: Implicit and explicit knowledge processing

Figure 6.8 depicts the knowledge layer in the context of the delivery pipeline. Although we do not impose an implementation strategy, it is important to note that this layer is not necessarily a database management system accessible from all environments. Although

this is one possibility, the knowledge layer can be conceptually anything that transports knowledge from one environment to others following standard methods and formats. The pipeline-wide access can be implemented based on practices from database administration or machine learning, such as database migrations and data versioning.

Conceptually, we split knowledge processing activities into implicit and explicit, from the autonomic managers' perspective. At run-time, all activities conducted to recall or update knowledge are considered as foreground knowledge processing. Conversely, similar activities conducted in any other environment are considered as background knowledge processing. We do this classification to highlight the relevance of choosing the right environment for each activity, always considering that each environment has its own limitations. For example, exploration time is limited at run-time; At development-time, where this is not an issue, execution conditions do not reflect the actual user demand. Moreover, this classification may motivate the creation of new environments. Consider, for example, temporary environments to conduct time-consuming and compute-intensive knowledge processing activities.

We now describe each of the processing activities present in Figure 6.8.

Knowledge Maintenance. Offline stakeholders maintain data sets and MARTs in their local development environment. Data science engineers can clean up existing data by removing abnormal behavior captured at run-time, or tag it accordingly to prevent skewing analytical models. Moreover, they can augment existing data with special and edge cases to consider unlikely but relevant regions of the search space. A product manager can recall collected metrics and combine them with source code elements to prioritize support tickets based on occurrence or relevance. Moreover, this information can be used to create dynamic reports and make decisions with up to date information. An operations planner can obtain monitoring data from a transportation system to analyze how social and economic factors affect usage patterns to make decisions based on the target population. For example, late-shift workers are more likely to take the bus in the evening, late-night and early-morning in specific parts of the city. In a second example, software developers and roles alike can augment their tools with production information to shorten the feedback cycle for their local changes to the managed system (*e.g.*, Production performance feedback in the IDE [**?**]). The main purpose of the knowledge layer is to support data-driven decisions on the development and planning side. Moreover, since autonomic managers access the same data sets and MARTs, it promotes improving online decisions over time, as well as increasing systems' autonomy.

Measurement. Measuring metrics about the system's operation is critical to understand how changes in the environment affect the system's behavior. We propose exploiting three sources of measurements. First, during development, autonomic managers can take advantage of existing testing procedures to characterize the evolution of the system over time. Moreover, as discussed in Chapter 5, testing procedures can be used to evaluate the system's response to changes in its configuration. Second, autonomic managers can use historical data to evaluate what-if scenarios. Action plans can be tried out as simulations prior to materializing them into the managed system. In this case, past measurements

embody the current system's behavior under specific operation conditions. This is useful to contrast results from hypothetical scenarios. And third, run-time monitoring provides unique insights into actual usage patterns, system performance, management cost, among many others. This information is key to measure the impact of changes on the development and planning side.

Recall and Synthesis. Recall refers to a system's ability to obtain relevant information from the knowledge layer. Ideally, autonomic managers can focus their attention on the particular subset of the knowledge that will increase prediction accuracy. The synthesis process refers to autonomic managers' ability to reify and optimize knowledge artifacts, and make them available to other managing systems, as well as offline stakeholders.

6.2.3.2. Online Knowledge Synthesis

As demonstrated in Chapter 4, run-time monitoring is a key enabler for realizing the continuous evolution cycle. Monitoring information is consolidated into data repositories and run-time models that reflect an up-to-date snapshot of the system's behavior and structure. High levels of representation fidelity enable accurate predictions, further analysis and improvement of the system. Nevertheless, the evolution operational mode poses knowledge requirements beyond collecting data.

Data collection is a common industry practice that leads to a better understanding of usage patterns and emerging behaviors. This is even more prevalent in IoT and CPS, where connected devices, such as buses and traffic sensors, are constantly sending measurements to data warehouses or data lakes (*cf.* Measurement in Figure 6.8). We posit that this is passive knowledge: data collections waiting to be processed by human operators through data mining or machine learning techniques. In contrast, we define active knowledge as any type of artifact that captures domain- and system-specific patterns, insights and dynamics from data. They can be exploited by managing systems to predict future states, support run-time decision making, and populate the evolution (and adaptation) space.

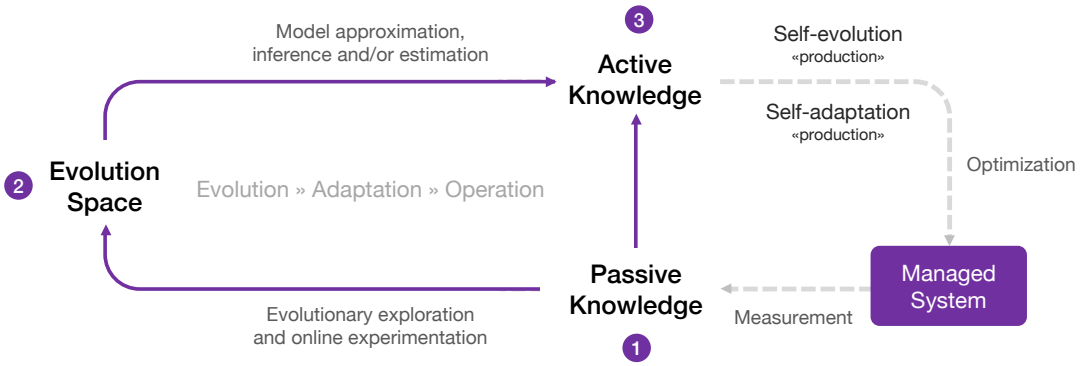


Figure 6.9.|: Conceptual continuous engineering cycle for run-time evolution

The concepts of active knowledge and evolution cycle converge into our envisioned continuous engineering cycle, depicted in Figure 6.9. This cycle comprises three main con-

cepts: passive knowledge (*cf.* ❶), evolution space (*cf.* ❷), and active knowledge (*cf.* ❸). We argue that there is a natural transition from collecting and storing data (*i.e.*, passive knowledge gathered from the managed system) to populate and explore future configuration states (*i.e.*, evolution space), ultimately to synthesize executable models (*i.e.*, active knowledge). In terms of control, these transitions can be seen as the dynamics between evolution, adaptation and operation. Changes in the evolution space affect the possible adaptation scenarios, thus determining the actual operational states. As shown in Figure 6.9, key techniques for realizing our continuous engineering cycle are evolutionary algorithms, online experimentation, model approximation, estimation or inference methods, as well as optimization methods (*e.g.*, hyperparameter optimization).

There are two ways of synthesizing active knowledge. On the one hand, based on data collected from deployed sensors, managing systems can fit statistical models to describe the current service demand (*e.g.*, passenger arrivals at each bus station), the current bus frequency per line and station, and other behaviors related to the system's operation. This process is represented by the arrow connecting labels ❶ and ❸ in Figure 6.9. On the other hand, synthesizing predictive models based on collected data may be too limiting in terms of the capacity to accommodate uncertainty. Since the models would be synthesized based on known execution conditions, their usefulness in handling unknown situations is very limited. Therefore, an intermediate exploration step is required to augment the data. Its objective is to generate enough information about the search space to be able to capture the dynamics between configuration parameters and the metrics. By exploring the search space of potential configuration alternatives, managing systems can reify knowledge artifacts that enable managed systems to respond to emerging situations, even if they have not occurred before. This process is represented by label ❷ in Figure 6.9. The full cycle, represented by labels ❶–❸, realizes MIAC through model identification (*i.e.*, model estimation). As shown in Figure 6.9, estimated models are then used as control inputs to adapt the managed system.

The purpose of exploring the search space is twofold. First, collected data can be augmented as previously explained. Second, managing systems can try out distinct configurations of the managed system, as explained in Chapter 5, to find possible solutions to an ongoing situation. As an example, consider our SUTS case study. In the event that a street becomes inaccessible, the managing system may run scenarios to determine a routing plan quickly, and continue the operation while minimizing the impact. In any case, a genetic algorithm is used to generate chromosomes (*i.e.*, configurations) and evaluate their fitness based on metrics of interest, such as the excess waiting time and the headway coefficient of variation in our SUTS case study. Since field experiments would be too expensive, time demanding, or simply damaging to the system's operation, the experiments are conducted in a controlled environment. For example, the experimentation system can derive simulation models based on the chromosomes.

Figure 6.10 illustrates an iterative process combining the techniques mentioned above: function reification, experimentation and evolutionary optimization. The main purpose of this process is to devise optimized operation parameters, as explained in Section 6.2.2.4. Figure 6.10 combines knowledge outcomes produced by the MIM and AM components.

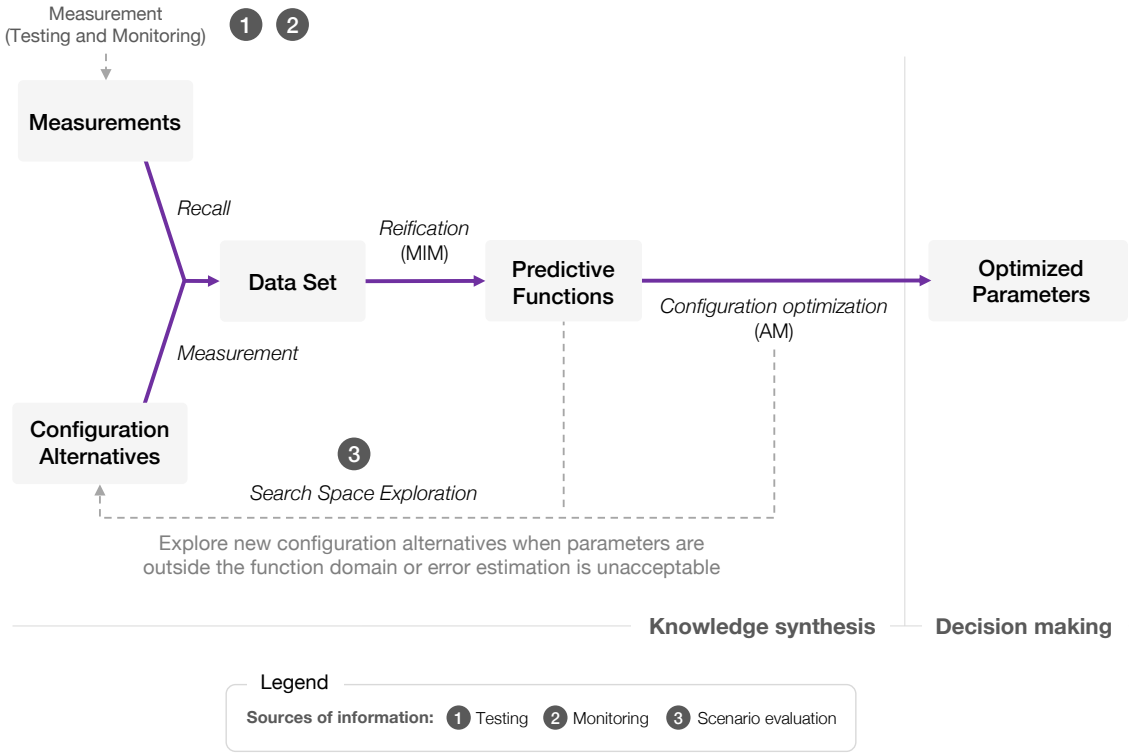


Figure 6.10.|: Iterative process to optimize configuration parameters

There are three sources of information feeding the reification process. The first two sources are non-functional testing and monitoring, which create metrics measurements at test-time and run-time, respectively (*cf.* Figure 6.8). The third source of information is the evaluation of configuration scenarios. This process is triggered by either the MIM or the AM at run-time, if the estimated error is unacceptable, or from another autonomic manager running in a non-production environment (*e.g.*, the self-improvement feedback loop presented in Chapter 5). In either case, a combined data set is passed to the experimentation subsystem. Once the functions are reified, and the estimated error is good enough, each parameter is optimized based on the corresponding viability zones.

6.3. Chapter Summary

The proliferation of Smart Cyber-Physical Systems (SCPS) and technological advances in Artificial Intelligence, Big Data, Cloud Computing and the Internet of Things are increasingly blending physical entities with their digital counterparts. This blending, however, challenges researchers to design reliable and dependable self-management capabilities, able of adapting and evolving autonomously to deal with unanticipated and emerging behavior at run-time.

This chapter presented our run-time evolution reference architecture for guiding the design of dependable and resilient CPS. Our architecture constitutes a blueprint for ar-

chitecting SCPS where self-evolution and self-adaptation work in harmony to achieve the system's operational goals autonomously. Self-adaptation and self-evolution constitute two operational modes of an autonomic manager for realizing a continuous evolution loop. In the forward direction, managing systems keep internal models and other forms of knowledge in sync with the natural evolution of the CPS. In the reverse direction, managing systems deliver short-term and long-term updates, such as operational corrections and optimizations. Thereby, our architecture addresses the dependability of run-time decisions, as well as the overall resiliency of the operation.

Increased autonomy in complex CPS raises various research challenges regarding the dependability and resiliency of CPS. On the one hand, CPS designers ought to introduce mechanisms to guarantee predicted outcomes of the adaptations and evolution actions planned and conducted at run-time. Failing to do so may cause a faulty behavior, increased operational costs or even safety issues. Our reference architecture addresses this challenge by accounting for the design and execution of online experiments. Experimentation plays a two-fold role in our architecture: exploration of the evolution and adaptation space, as well as run-time assurance. More specifically, the factors contributing to dependability are: Error mitigation through multi-model identification; Reliable models through model inference; Evidence collection through experimentation; And autonomic behavior through adaptive control. On the other hand, CPS must exhibit, and work under, resilient operation even in the presence of uncertainty. Our reference architecture extends the notion of a viability zone and includes it as a supervisor component. Whenever a variable of interest starts behaving in an unexpected way, the supervisor component will switch between the adaptation and the evolution operational modes as needed. Moreover, the major components realizing these two modes of operation will ensure that the controller has the most appropriate parameters to plan corrective actions. The factors contributing to resiliency are: Predictable adaptation through reliable models; Run-time validation through evidence collection; Error mitigation through parameter estimation; Goal achievement through hyperparameter optimization; And Assurance at run-time through viability zones and control objectives.

The proposed reference architecture comprises two main components. On the one hand, a model identification mechanism (MIM) approximates MARTs to describe system metrics, such as the excess waiting time. These models serve as a reference input to guide the adaptation and operation of the managed system. The MIM uses various types of execution environments to simulate the managed system under specific context conditions, thus generating key performance indicators, such as the excess waiting time and the headway coefficient of variation. A genetic algorithm guides the evolution space exploration, running the simulated system as part of the fitness value calculation. Online experimentation is a key technique to optimize the exploration process as well as clustering results based on similarity. Moreover, we detail the process for synthesizing insights at run-time through active and passive (*i.e.*, declarative) knowledge, a base requirement for value aggregation. Furthermore, the MIM exemplifies the kind of updates autonomic managers can deliver as part of the evolution cycle. On the other hand, an execution adjustment mechanism (AM) finds appropriate control parameters so that the controller can enforce the control

objectives in the CPS. The AM constitutes an example of how to exploit active knowledge to realize self-adaptation. It uses reified functions to optimize the control parameters with respect to a given setpoint (*i.e.*, the associated control objective). Moreover, this component exemplifies a way to associate the evolution, adaptation and operational spaces. In this case, the approximated function, constrained by the setpoint, determines the possible configuration parameters that guide the adaptation and eventually the operation.

This chapter concludes our focus on self-evolution through self-management, self-improvement, and self-regulation. Next chapter presents the evaluation of our contributions based on the two case studies introduced in Chapter 3, namely cloud infrastructure management and smart urban transportation.

Part III

Evaluation

Chapter 7

Evaluation

Contents

This chapter presents the evaluation of our contributions. We conducted functional, experimental and qualitative validation of representative elements from each contribution. We do so based on the case studies introduced in Sections 3.3 and 3.4—*Infrastructure Management Case Study* and *Smart Urban Transit System Case study*. For the infrastructure management part, we present three application scenarios, namely: 1) assistance with acquisition of knowledge from deployed resources; 2) prevention of configuration drift for IaC templates; and 3) exploration of software architecture variants. Scenarios 1) and 2) (*cf.* Section 7.1.2) aim to validate the soundness and feasibility of our continuous software evolution pipeline. Scenario 3) aims to validate the capacity of our self-improvement feedback loop to contribute to the managed system's long-term evolution. For the smart transportation part, we present an application scenario focused on model identification and

run-time adjustment (*cf.* Section 7.2.2). Such a scenario aims to validate the soundness and feasibility of our run-time evolution reference architecture.

Our qualitative validation focuses on aspects not directly addressed by our infrastructure management case study. In Section 7.1.4, we address the following concerns: the holistic and continuous nature of our software evolution process; and our contribution to reduce remaining discontinuities in the SDLC.

Since we introduced the case studies from the problem point of view, we state two concrete cases of application to conduct the evaluation. In the first case, we describe a Cloud Media Encoding Service (CMES) based on the problem definition from Section 3.3.1. Similarly, in the second case, we follow the problem definition from Section 3.4.1 to describe a transportation system. We model this system and conduct related experiments based on operation data from a Colombian BRT system.

7.1. Infrastructure Management Case Study

This section contains the validation of our contributions based on the infrastructure management case study. It is organized as follows. Section 7.1.1 introduces our concrete case of application. Section 7.1.2 presents the application scenarios concerning functional validation. Section 7.1.3 presents the application scenarios concerning experimental validation. Finally, Section 7.1.4 addresses the qualitative validation of various aspects of our contributions.

7.1.1. Concrete Case of Application: A Cloud Media Encoding Service

The CMES features a REST API to convert digital media files from one standard format to another. Given an address of a website and a desired format, the CMES will download any audio or video files present in the website and will convert them, if necessary. The catalog of supported websites contains more than a thousand news and media-sharing sites, including BBC, CBS, YouTube, Vimeo and SoundCloud. In case the site contains a playlist, the CMES will collect the items and merge them into a single file. Requests to this service tend to be time consuming because they include establishing the connection, downloading the files, and compiling and encoding them if necessary. Users worldwide use similar services to download media files from social networks and websites, therefore, keeping the latency as low as possible is a priority, as well as maximizing the request-handling capacity.

The CMES comprises two services: a gRPC-based[1] worker component that downloads and converts the media files; and an HTTP front-end service (*i.e.*, an API). We decided to deploy the CMES using Kubernetes[2] because it greatly facilitates realizing and enforcing deployment specifications. Moreover, it enables integrating third party components that

[1] High-performance Remote Procedure Call (RPC) framework, available at `https://grpc.io`

[2] Open-source system for automating deployment, scaling, and management of containerized applications, available at `https://kubernetes.io`

are commonly required in a cloud environment, such as load balancing, caching, routing, rate limiting, and job handling.

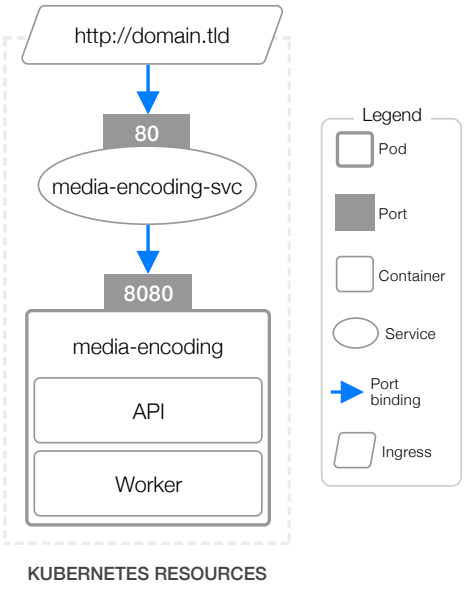


Figure 7.1.|: Deployment diagram for the bare minimum configuration of the CMES

We decided to start from a very simple architecture, suppressing any assumption regarding potential user traffic. The overall goal of our second contribution is that the software architecture and the computing infrastructure change along with the service demand. Figure 7.1 depicts the bare minimum configuration for the concrete application. `API` and `Worker` are Linux *containers* assembled into one *pod*—a group of one or more containers with shared storage and network. Service `media-encoding-svc` routes user requests from `http://domain.tld` to API through port 8080, and in turn to Worker through port 50051.

Resources of the computing infrastructure have been manually managed through the cloud provider's web dashboard. Nevertheless, it would be highly beneficial to manage them through IaC templates, using Terraform's HCL notation. As expected, the CMES is prone to the issues we discussed in Section 3.3.1, namely: migration from ad-hoc management to IaC, configuration drift (*i.e.*, technical debt), and the need for continuous improvement. The initial cluster configuration of the CMES features a Kubernetes cluster with one control plane (*i.e.*, a Kubernetes master node) and three computing nodes. Each of them has been allocated 8 GB of RAM and 4 vCPUs.

7.1.2. Functional Validation

This section presents two application scenarios addressing the functional validation of our evolution pipeline. Section 7.1.2.2 focuses on creating an IaC template and its corresponding instance from resources deployed to a target cloud environment. This scenario validates the feasibility of our model transformation chain, our Fork and Collect algorithm, and

the integration of multiple software tools from the IaC life cycle. Section 7.1.2.3 focuses on demonstrating how the pipeline handles changes on either side of the integration loop. This scenario validates the effectiveness of the direct and CI-aware evolution workflows.

7.1.2.1. Implementation Details

Our implementation of the continuous evolution pipeline is driven by run-time manipulation of models. We developed Historian's monitoring metamodel and Terraform's HCL metamodel using Xcore[3]—a metamodeling Domain-Specific Language (DSL) based on Eclipse EMF.[4] On top of the HCL metamodel, we developed an interpreter based on Eclipse Xtext.[5] The interpreter allows loading IaC templates and transforming them into HCL model instances. Moreover, our interpreter's public API allows comparing and merging HCL models, as well as transforming them back to their textual representation. We implemented diff and merge functionalities based on EMF Compare.[6]

We also developed a graph metamodel based on Java's XML API. We use this metamodel to configure Historian's endpoint dependencies. Listing 7.1 depicts VMware's API configuration tailored to our case of application. Each monitor declaration represents an Hypertext Transfer Protocol (HTTP) GET endpoint. Their name references a unique identifier taken from VMware's OpenAPI model. To help developers specify the dependency graph, we developed a Command Line Interface (CLI) tool for generating a base monitoring project, including authentication, logging and dependency configuration. This project is setup to use Historian's run-time library, thus requiring minimal effort from developers.

```
1 <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
2 <monitors xmlns="http://www.rigiresearch.com/middleware/graph/1.0.0">
3   <!-- VM details -->
4   <monitor name="getVcenterVm">
5     <input name="vm" source="listVcenterVm">vm</input>
6     <mappings>
7       <transformation selector="value"/>
8       <augmentation inputs="vm"/>
9     </mappings>
10  </monitor>
11  <!-- Datacenters details -->
12  <monitor name="getVcenterDatacenter">
13    <input name="datacenter" source="listVcenterDatacenter">datacenter</input>
14    <mappings>
15      <transformation selector="value"/>
16      <augmentation inputs="datacenter"/>
17    </mappings>
18  </monitor>
19  <!-- Inventory of datacenters -->
20  <monitor name="listVcenterDatacenter">
21    <output name="datacenter" selector="//datacenter" multivalued="true"/>
```

[3] https://wiki.eclipse.org/Xcore
[4] https://www.eclipse.org/modeling/emf
[5] http://www.eclipse.org/Xtext
[6] https://www.eclipse.org/emf/compare

```
22      <mappings>
23        <transformation selector="//datacenter" multivalued="true"/>
24      </mappings>
25    </monitor>
26    <!-- Inventory of VMs -->
27    <monitor name="listVcenterVm">
28      <input name="filter.vms">vm-22922</input>
29      <output name="vm" selector="//vm" multivalued="true"/>
30      <mappings>
31        <transformation selector="//vm" multivalued="true"/>
32      </mappings>
33    </monitor>
34    ...
35    <!-- Inventory of VMs per datacenter -->
36    <monitor name="listVcenterVmFilteredByDatacenter" template="listVcenterVm">
37      <input name="filter.datacenters" source="listVcenterDatacenter">datacenter</input>
38      <mappings>
39        <transformation selector="//vm" multivalued="true"
              groupByInput="filter.datacenters"/>
40      </mappings>
41    </monitor>
42  </monitors>
```

Listing 7.1: Historian's configuration file for monitoring VMware resources

We managed the IAC template instances of our case of application through IBM CAM. The communication between the evolution coordinator component from our pipeline and CAM was implemented by consuming CAM's public REST API. This includes functionality to initialize, plan, refresh and update Terraform template instances. Similarly, the evolution coordinator consumed Github's Git API for creating a local copy of the target repository (*i.e.*, a clone), and publishing new code branches.

We implemented the evolution coordinator and the run-time agent as independent executable components. The former publishes a POST endpoint to receive specification updates from the latter.

7.1.2.2. Application Scenario 1: Assistance With Knowledge Acquisition

Since the infrastructure resources are already running on VMware, we focus on the generation of the corresponding IAC template files, as well as the creation of the instance on IBM CAM. The first step consists of executing the evolution coordinator. As shown in the logs below, the coordinator clones the existing—but so far empty—repository where the template files will be stored (*cf.* Line 2). Then, the coordinator collects an authentication token from CAM for executing future requests. Finally, it starts an evolution service on port 5050 to receive specification updates from the run-time agent.

```
1  $ ./coordinator-0.1.0/bin/coordinator
2  TerraformRepository - Cloned repository https://github.com/company/cmes
```

```
3   CamClient - Collected authentication token (CAM)
4   Server - Started the evolution service (port 5050)
```

The second step consists of executing the run-time agent. In this case, we have implemented a VMware agent that knows how to map changes from the cloud resources to the current HCL model instance—so far non-existent. As shown in the listing below, once executed, the agent instantiates a Historian monitor, thus scheduling a task for running the FORK AND COLLECT algorithm periodically.

```
1   $ ./vmware.hcl.agent-0.1.0/bin/vmware.hcl.agent
2   RuntimeAgent - Started the Historian Monitor
```

The third step is automated and consists of running FORK AND COLLECT to create a snapshot of the resources running on VMWare. The algorithm starts exploring the API endpoints based on the dependency graph (*cf.* Listing 7.1). As shown in Line 1 of the listing below, the first endpoints inventory the running system. That is, they list existing cloud resources. Subsequent endpoints collect details for each listed entity (*i.e.*, Lines 4, 8 and 9).

```
1   ForkAndCollectAlgorithm - Branches: 5 Endpoints: listVcenterHost,
        listVcenterDatacenter, listVcenterFolder, listVcenterResourcePool,
        listVcenterVm, listVcenterCluster
2   ForkAndCollectAlgorithm - Branches: 1 Endpoints: listVcenterVmFilteredByHost
3   ForkAndCollectAlgorithm - Branches: 3 Endpoints: listVcenterVmFilteredByDatacenter
4   ForkAndCollectAlgorithm - Branches: 3 Endpoints: getVcenterDatacenter
5   ForkAndCollectAlgorithm - Branches: 1 Endpoints: listVcenterVmFilteredByFolder
6   ForkAndCollectAlgorithm - Branches: 5 Endpoints: listVcenterVmFilteredByResourcePool
7   ForkAndCollectAlgorithm - Branches: 5 Endpoints: getVcenterResourcePool
8   ForkAndCollectAlgorithm - Branches: 1 Endpoints: getVcenterVm
9   ForkAndCollectAlgorithm - Branches: 3 Endpoints: getVcenterCluster
```

The listing below depicts the JSON document created by FORK AND COLLECT. It contains the minimum amount of information necessary to instantiate or update the IAC model. The property names match those from the dependency graph, and in turn, the endpoint identifiers from VMware's OpenAPI specification.

```
1 {
2   "listVcenterHost":[
3     "host-112"
4   ],
5   "listVmFilteredByHost":{
6     "host-112":[
7       "vm-22922"
8     ]
9   },
```

```
  ...
  "listVcenterDatacenter":[
    "CAMDC1",
    "CAMDC2"
  ],
  "listVcenterVmFilteredByDatacenter":{
    "CAMDC1":[
      "vm-22922"
    ],
    "CAMDC2":[]
  },
  "listVcenterVm":[
    "vm-22922"
  ],
  "getVcenterVm":[
    {
      "vm":"vm-22922",
      ...
      "cpu":{ ... },
      "disks":[ ... ],
      "memory":{ ... },
      "name":"camc-vis232c-vm-121"
    }
  ],
  ...
  "listVcenterResourcePool":{
    "value":[
      {
        "name":"CAM02/Resources",
        "resource_pool":"resgroup-1"
      }
    ]
  }
}
```

Right after exploring VMware's API, the agent maps the JSON document to HCL resources, thus instantiating a model that represents the current deployment. It also extracts parameter information from the document, including their name and current value on VMware. The agent finishes the scheduled task by sending both elements, the specification and the parameters, to the evolution coordinator. The parameters are created to match those present in the Terraform template, as well as those included in CAM's instance.

```
RuntimeAgent - Sent the current specification to the evolution coordinator
RuntimeAgent - The specification values are as follows
RuntimeAgent - resource_pool_1_name = CAM02/Resources
RuntimeAgent - vm_1_num_cores_per_socket = 1
RuntimeAgent - vm_1_disk_1_unit_number = 0
RuntimeAgent - vm_1_name = camc-vis232c-vm-121
RuntimeAgent - vm_1_number_of_cpus = 2
RuntimeAgent - vm_1_adapter_1_type = VMXNET3
RuntimeAgent - vm_1_scsi_type = lsilogic
```

```
10  RuntimeAgent - vm_1_guest_os_id = ubuntu64Guest
11  RuntimeAgent - network_1_interface_1_label = VIS232
12  RuntimeAgent - vm_1_disk_1_label = camc-vis232c-vm-121/camc-vis232c-vm-121.vdmk
13  RuntimeAgent - datastore_1_name = CAM02-RSX6-002
14  RuntimeAgent - vm_1_memory = 2048
15  RuntimeAgent - datacenter_1_name = CAMDC2
16  RuntimeAgent - vm_1_disk_1_size = 85
17  RuntimeAgent - vm_1_folder = ContentRunTimes
18  ...
```

The evolution coordinator uses the HCL interpreter for merging the current model with the new one. In this case, since there is no current model, it just stores it. Next, it transforms the model into its textual representation and updates the local repository. In addition to the Terraform files, the coordinator also generates `camtemplate.json` and `camvariables.json`, two JSON templates required to import the repository into CAM. These files contain information about the Terraform template, the repository, and declared parameters. Changes to the remote repository are included in a new branch by default.

```
1  TerraformRepository - Pushed changes to the remote repository
```

Having published the new template, the coordinator proceeds to import the template if it does not exist, specifying the repository and the new branch. Next, it creates an instance and performs various Terraform actions, as shown in the listing below. First, it performs a Terraform plan (*cf.* Line 1), which uses the VMware provider to analyze the deployed resources with respect to the template and the parameter values. As the status indicates, Terraform determines that some changes are necessary (*cf.* Line 2). This is because Terraform's current state is unaware that the specified resources are the same ones running on VMware. The coordinator then imports the resource identifiers into Terraform's state (*cf.* Line 3), and then confirms that the changes are no longer necessary (*cf.* Line 6). At the time our prototype was implemented, CAM did not support the import action. Therefore, we performed this action manually.

```
1  CamClient - Started a Terraform plan
2  CamClient - PLAN action ended with status SUCCESS_WITH_CHANGES
3  CamClient - Started a Terraform import
4  CamClient - IMPORT action ended with status SUCCESS
5  CamClient - Started a Terraform plan
6  CamClient - PLAN action ended with status SUCCESS
```

Listing 7.2 shows the contents of the Terraform template. It contains i) a list of variables declared from the collected resource attributes (*cf.* Lines 1-4); ii) provider and data declarations necessary to link the new Terraform resources with those outside the IaC life

cycle (*cf.* Lines 5-12), such as data centers and hosts;[7] and iii) the imported resources (*cf.* Lines 13-49).

```
1 variable "datacenter_1_name" {
2   type = "string"
3 }
4 ...
5 provider "camc" {
6   version = "~> 0.2"
7 }
8 ...
9 data "vsphere_datacenter" "datacenter_1" {
10   name = "${var.datacenter_1_name}"
11 }
12 ...
13 resource "vsphere_virtual_machine" "vm_1" {
14   datastore_id         = "${data.vsphere_datastore.datastore_1.id}"
15   folder               = "${var.vm_1_folder}"
16   guest_id             = "${var.vm_1_guest_os_id}"
17   memory               = "${var.vm_1_memory}"
18   name                 = "${var.vm_1_name}"
19   num_cores_per_socket = "${var.vm_1_num_cores_per_socket}"
20   num_cpus             = "${var.vm_1_number_of_cpus}"
21   resource_pool_id     = "${data.vsphere_resource_pool.resource_pool_1.id}"
22   scsi_type            = "${var.vm_1_scsi_type}"
23   clone {
24     template_uuid = "${data.vsphere_virtual_machine.vm_1_template.id}"
25     customize {
26       dns_server_list = "${var.vm_1_dns_servers}"
27       dns_suffix_list = "${var.vm_1_dns_suffixes}"
28       ipv4_gateway    = "${var.vm_1_ipv4_gateway}"
29       linux_options {
30         domain    = "${var.vm_1_domain}"
31         host_name = "${var.vm_1_name}"
32       }
33       network_interface {
34         ipv4_address = "${var.vm_1_ipv4_address}"
35         ipv4_netmask = "${var.vm_1_ipv4_prefix_length}"
36       }
37     }
38   }
39   disk {
40     label       = "${var.vm_1_disk_1_label}"
41     size        = "${var.vm_1_disk_1_size}"
42     unit_number = "${var.vm_1_disk_1_unit_number}"
43   }
44   network_interface {
45     adapter_type = "${var.vm_1_adapter_1_type}"
46     network_id   = "${data.vsphere_network.network_1.id}"
47   }
48 }
49 ...
```

[7] These resources are configured by the cloud provider, therefore they are not managed along with the CMES's resources.

Listing 7.2: Declaration of VMware resources in the Terraform template

In this section, we did a walkthrough of the first application scenario step by step. We showed how our prototype collects information from the target cloud and successfully creates the corresponding Terraform and CAM templates. Moreover, we describe additional interactions with CAM to instantiate the new template, thus closing the integration loop. Therefore, our prototype was able to acquire the necessary knowledge to migrate the CMES from ad-hoc infrastructure management to IaC.

7.1.2.3. Application Scenario 2: Prevention of Configuration Drift

This application scenario focuses on demonstrating that our integration loop prevents configuration drift. That is, we show that our loop keeps the artifacts from both sides (*i.e.*, Dev and Ops) synchronized. This includes the Terraform template, the CAM instance, and the deployed resources. In Section 7.1.2.2, we already showed the mapping process from Ops to Dev (*i.e.*, from VMware to Terraform and CAM). Therefore, in this section we focus on the updates stemming from changes on either side.

From left to right, changes can be either structural or parametric. In the first case, changes would occur in the Terraform specification. Nevertheless, for this change to be eventually deployed, the template instance needs to be updated first on CAM. Changing the template is a conscious and intentional effort. This means that changing the corresponding instance is expected to follow. In any case, modifying the template without reflecting the changes on CAM would have no effect on the instance or the deployed resources. Therefore, we do not consider this case as a potential drift in the configuration. In the second case, changes would occur on the template instance. Once the parameter values are updated, CAM performs a Terraform plan action followed up by an apply action. As a result, the cloud resources are synchronized with the template instance. Eventually, the Historian monitor will notify the run-time agent about the changes, and in turn, the agent will notify the evolution coordinator. However, since the template did not change, it only updates the values, as shown in the logs below.

```
RuntimeAgent - The specification did not change
RuntimeAgent - The specification values are as follows
RuntimeAgent - resource_pool_1_name = CAM02/Resources
RuntimeAgent - vm_1_num_cores_per_socket = 1
RuntimeAgent - vm_1_disk_1_unit_number = 0
RuntimeAgent - vm_1_name = camc-vis232c-vm-121
RuntimeAgent - vm_1_number_of_cpus = 2
RuntimeAgent - vm_1_adapter_1_type = VMXNET3
RuntimeAgent - vm_1_scsi_type = lsilogic
RuntimeAgent - vm_1_guest_os_id = ubuntu64Guest
RuntimeAgent - network_1_interface_1_label = VIS232
RuntimeAgent - vm_1_disk_1_label = camc-vis232c-vm-121/camc-vis232c-vm-121.vdmk
```

```
13  RuntimeAgent - datastore_1_name = CAM02-RSX6-002
14  RuntimeAgent - vm_1_memory = 2048 -> 4096
15  RuntimeAgent - datacenter_1_name = CAMDC2
16  RuntimeAgent - vm_1_disk_1_size = 85 -> 120
17  RuntimeAgent - vm_1_folder = ContentRunTimes
18  ...
```

The coordinator starts a new Terraform plan, which ends successfully without requiring further actions. At this point, both sides are effectively synchronized.

```
1  CamClient - Started a Terraform plan
2  CamClient - PLAN action ended with status SUCCESS
```

From right to left, changes can originate from multiple sources, such as a web management dashboard, a CLI tool, a script, or the cloud infrastructure itself (*e.g.*, an automated migration). In any case, it is very likely that the change will be ultimately performed through the cloud's public API. Therefore, for testing our prototype, it is enough with choosing only one source. We performed a storage migration using VMware's migration tool from the web dashboard. Consequently, the run-time agent identifies the migration and sends the new data store to the evolution coordinator. The listing below shows the output logs from the run-time agent. As in the previous case, the evolution coordinator updates the parameter on CAM, and the transaction ends with a successful Terraform plan.

```
1  RuntimeAgent - The specification did not change
2  RuntimeAgent - The specification values are as follows
3  RuntimeAgent - resource_pool_1_name = CAM02/Resources
4  ...
5  RuntimeAgent - vm_1_guest_os_id = ubuntu64Guest
6  RuntimeAgent - network_1_interface_1_label = VIS232
7  RuntimeAgent - vm_1_disk_1_label = camc-vis232c-vm-121/camc-vis232c-vm-121.vdmk
8  RuntimeAgent - datastore_1_name = CAM02-RSX6-002 -> visidpxer18_local
9  RuntimeAgent - vm_1_memory = 4096
10 ...
```

There is a last type of run-time change that may cause configuration drift. In the event that a cloud resource is deleted, or added to Historian's monitoring configuration,[8] both the template and its instances need be reassessed. As shown in Section 7.1.2.3, the evolution coordinator will add new resources to the Terraform template, as well as their corresponding variables, providers, and data declarations. Since this functionality is based on model-to-text transformations, deleting resources would work as well. The coordinator creates a new version for each instance pointing to the new repository branch. This means that a member of the team needs to approve the new versions manually.

[8]This is possible by updating the VMware XML configuration. More specifically, the endpoint filters need be updated, and the run-time agent must be re-executed.

In this section, we have covered various evolution scenarios. We discussed how our integration loop, through round-trip engineering, contributes to reduce configuration drift for deployments managed with Terraform and CAM. In the next section, we concentrate on the experimental validation of our contributions.

7.1.3. Experimental Validation of Our Proof of Concept

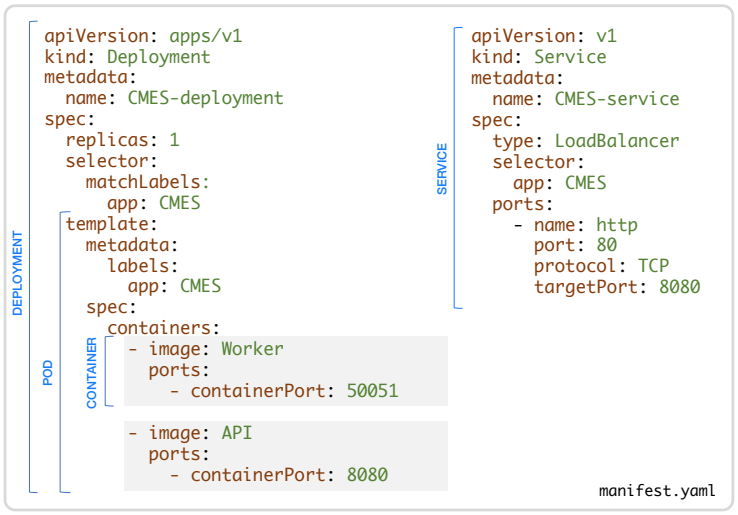
The experimental validation of our proof of concept, for the infrastructure management case study, consists of a semi-automated exploration of software architecture variants (*i.e.*, Our implementation of the C-FL). We aim to show how architecture variants perform under distinct execution scenarios. Thereby, we demonstrate the feasibility of our experimentation feedback loop. Moreover, we show the relevance of recognizing what configurations perform best according to specific software qualities. In this case, we focus on service latency. By performing this exploration during development, autonomic decisions at run-time are timely, but also cost-effective. Moreover, our prototype implementation found performant variants that meet the required service latency. Therefore, they are able to contribute long-lasting changes to the evolution of the managed system, in a continuous way.

This section is structured as follows. We first detail some aspects of our implementation in Section 7.1.3.1. We then describe the experimentation setup in Section 7.1.3.2. Finally, we present and analyze the results in Section 7.1.3.3. In this section, we omit some of the charts depicting the behavior of each architecture variant. These additional resources are included in Appendix B.

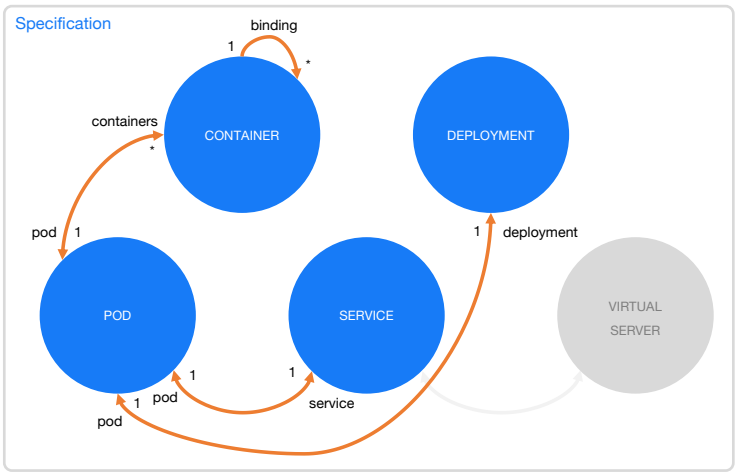
7.1.3.1. Implementation Details

Our implementation of the C-FL takes advantage of Kubernetes's declarative configuration for devising architecture variants. We built a library of design patterns by manipulating manifest files—that is, Kubernetes's specifications. Figure 7.2a displays a simplified version of the Kubernetes manifest for the bare minimum configuration of the CMES. The constituent elements of the manifest file match those present in Figure 7.2b. Such a figure depicts the relationships between Kubernetes resources declared in the manifest. Each of these relationships represents an axis for navigating forward and backward from any given resource. For example, given any container, it is possible to navigate to its containing pod and, in turn, navigate to the associated service or deployment. We used these navigation axes to implement our pattern library. Figure 7.3 contains a sample implementation of the Load Balancer pattern. Lines 8 and 9 refer to parameters associated with the pattern, whereas Lines 8-18 implement the actual pattern application. In this case, the implementation takes advantage of Kubernetes's built in load balancer for deployments, thus, setting the number of pod instances suffices for applying the pattern. The C-FL uses our library for loading and transforming manifest files before deploying the experiment trials.

To simplify the implementation, and because some design patterns contain application-specific code (*i.e.*, Master/Worker and Producer/Consumer), we tested 6 of the generated



(a) Resource definitions



(b) Abstract navigation axes

Circles represent Kubernetes resources. Arcs are navigation links for traversing the specification.

Figure 7.2.|: Fluent interface for navigating Kubernetes resources

configurations. Although a complete run of the experiment would provide more accurate results, we believe that comparing these trials is representative of the overall results. Furthermore, these design variants are commonly found in production-ready applications with moderate workload. The variants are as follows.

V1 Client→Proxy-Cache→LB→API→LB→Worker
V2 Client→LB→Proxy-Cache→LB→API→LB→Worker
V3 Client→LB→Proxy-Cache (sharded)→LB→API→LB→Worker
V4 Client→Rate-Limiting-Proxy→LB→API→LB→Worker
V5 Client→API→Master/Worker→Worker
V6 Client→API→Producer/Consumer→Worker

All variants load-balance requests to API and Worker. Variants V5 and V6 do so with an

```
@FunctionalInterface
public interface Pattern {
    Specification apply(Specification spec, Map<String, Object> config)
}

class LoadBalancer implements Pattern {
    Specification apply(Specification spec, Map<String, Object> config) {
        val transformed = spec.copy()
        val containers = <List<Container>>option(config, "image")
        val replicas = <Integer>option(config, "replicas").orElse(3)
        containers.forEach(container -> {
            val deployment = container.pod().deployment()
            val updated = new Deployment(
                deployment.name(),
                replicas,
                deployment.pod()
            )
            transformed.replace(deployment, updated)
        })
        return transformed
    }
}
```

Parameters

Navigation

Figure 7.3.|: Sample pattern definition using our fluent interface

implicit load balancer. Variant V1 has a single Proxy Cache instance. Variant V2 is similar but it contains a cache cluster. We duplicated this variant into V3 to measure the effect of sharding the cache. All Proxy Cache instances implemented the same caching policy: The request URL was used as the cache key; Revalidation was set to off, and a minimum use of 0 requests was required to store responses in the cache; And cache lock was enabled with an age of 30 seconds and timeout of 60 seconds. Similarly, all instances of Load Balancer used the same number of container instances: 10 for API, 15 for Worker, and 10 for Proxy Cache. In the case of Master/Worker there were 10 instances of the Master component. And for Producer/Consumer, there were 10 producer and consumer adapters.

7.1.3.2. Experimentation Setup

We used load testing to simulate predictable user traffic according to three execution scenarios, namely: Regular, Linear and Spike. The Regular scenario represents a stable user traffic with a target throughput of 450 requests distributed across 30 users (*i.e.*, threads). The Spike scenario was set up similarly targeting 150 requests with an additional spike of 90 requests. And the Linear scenario represents an increasing user traffic of 160 users with a ramp-up period of 1 second. Experiment results ended up containing fewer requests due to the hosting platform (*i.e.*, YouTube) restricting the number of consecutive requests. Nevertheless, sample sizes are large enough to guarantee sound analyses. Each scenario was modeled using jMeter,[9] which allowed us to have repeatable executions and measurements per request. The scenarios were executed on a single machine with 8 GB of RAM

[9] https://jmeter.apache.org

and a 1.6 GHz Dual-Core Intel i5 processor. The Kubernetes cluster was setup based on the CMES's initial configuration (*i.e.*, 3 computing nodes with 8 GB of RAM and 4 vCPUs). Load balancing for HTTP was done with nginx,[10] and linkerd[11] for gRPC. Each scenario ran for at least 5 minutes.

The test data was created based on a catalog of 80 videos and 80 playlists, which were uploaded to YouTube. The latter contains 10 videos each from the former. Although there were 80 different URLs, it is worth noting that all of them corresponded to the same 2 minute and 2.6 MB video. We made this decision aiming to reduce the variability in the results.

7.1.3.3. Experimentation Results

The Shapiro-Wilk test confirmed that none of the variants' samples are normal. The Kruskal-Wallis test found a significant difference in the mean latencies for all scenarios. According to Dunn's test, variants V1, V2 and V3 are not significantly different for any of the scenarios, regardless of the request type. For the Regular scenario and video requests, Dunn's test found that V4 and V6 are different by chance. In the case of playlist requests, the test found that V4, V5 and V6 are not significantly different. Out of these variants, V3 was selected as the best one[12] with a mean latency of 1.46 seconds for videos and 1.62 seconds for playlists. We present various descriptive statistics for video and playlist requests in Table 7.1. Table 7.2 displays pairwise comparisons from Dunn's test for video and playlist requests. The Z statistic represents how many standard deviations, above or below, is the z-test from the mean population from which the score derived. That is, it describes each latency value's relationship to the mean of the group. In a multiple comparison testing, the adjusted P value is the smallest (familywise) significance level at which a particular comparison will be declared statistically significant. Figure 7.4 shows the latency by request type for the Regular scenario.

A similar analysis for the Spike scenario reveals that variants V2 and V4 are different by chance. Nevertheless, the number of erroneous requests for V4 clearly separates them. V4 rejected the requests causing the spike—a surge of at least 44 requests, which would be considered an abusive use of the CMES. Consequently, V4 was selected as the best variant with a mean latency of 4.45 seconds for videos and 28.9 for playlists. Figure 7.5 shows the latency by request type for the Spike scenario. Figure 7.5d shows that V4 completely mitigated the traffic spike, while the rest struggled to keep the latency low, even when caching the responses.

For the Linear scenario and video requests, variants V5 and V6 were found not significantly different. For playlist requests, variants V1, V2, V3 and V4 were found different by chance. These results are shown in Figure 7.6. V3 was selected as the best variant with a mean latency of 2.80 seconds for videos and 20.7 for playlists.

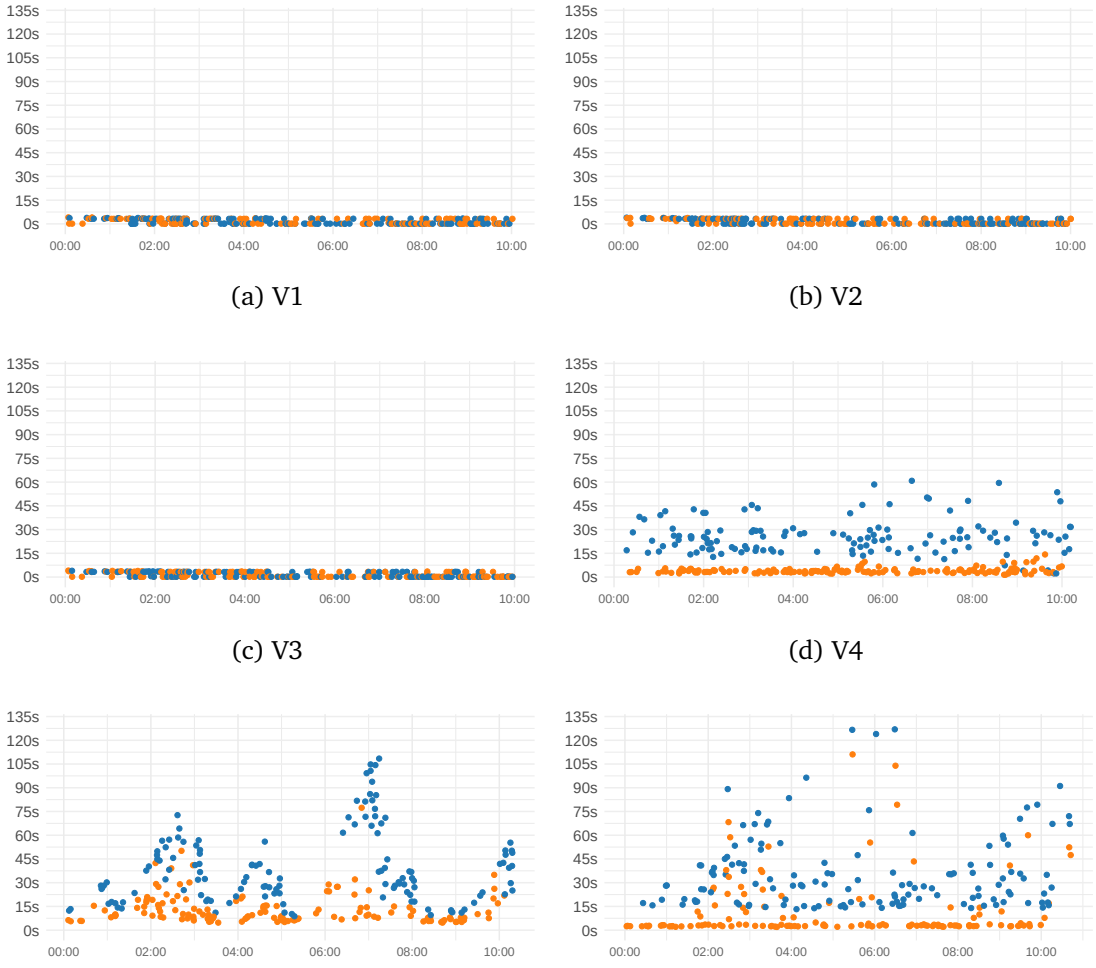
[10] https://nginx.org
[11] https://linkerd.io
[12] We averaged video and playlist requests and chose the smallest out of the cluster containing V1, V2 and V3.

Table 7.1.: Descriptive statistics for video and playlist requests
n represents the sample size, excluding erroneous requests, which appear in parenthesis; *SD* and *SE* stand for standard deviation and error; *LCL* and *UCL* stand for lower and upper control limits; *Med* stands for median; *Min* and *Max* stand for minimum and maximum. Latency values are in seconds.

	Video Requests						**Playlist Requests**					
	V1	V2	V3	V4	V5	V6	V1	V2	V3	V4	V5	V6
Regular												
n	148	156	163	163	144	130	160	152	145	132 (13)	133	136
Mean	1.65	1.61	1.46	4.03	13.5	12.2	1.55	1.59	1.62	25.6	39.5	35.9
SD	1.55	1.53	1.54	1.82	10.1	19.7	1.52	1.54	1.56	11.5	23.2	23.7
SE	0.127	0.123	0.121	0.143	0.842	1.73	0.120	0.125	0.129	1.00	2.01	2.03
LCL	1.40	1.37	1.22	3.75	11.8	8.79	1.31	1.35	1.36	23.6	35.5	31.9
UCL	1.91	1.85	1.70	4.31	15.1	15.6	1.79	1.84	1.87	27.6	43.4	39.9
Med	0.224	0.186	0.179	3.58	9.55	3.11	0.182	0.181	0.199	24.6	32.9	28.4
Min	0.153	0.152	0.158	1.46	4.78	2.05	0.151	0.154	0.159	2.29	8.77	13.3
Max	4.13	3.81	4.10	14.3	77.4	111	3.82	3.97	3.97	60.8	108	127
Spike												
n	101	84	93	72 (46)	95 (4)	84	77	96 (4)	85 (1)	75 (44)	59 (16)	77
Mean	9.86	6.95	9.87	4.45	98.2	63.8	42.5	39.8	45.2	28.9	182	84.1
SD	7.58	5.38	7.51	1.40	64.6	56.5	25.1	29.5	34.2	11.9	80.7	50.8
SE	0.754	0.587	0.779	0.165	6.62	6.16	2.86	3.01	3.71	1.37	10.5	5.79
LCL	8.36	5.78	8.32	4.12	85.0	51.6	36.8	33.8	37.8	26.2	161	72.6
UCL	11.4	8.11	11.4	4.78	111	76.1	48.2	45.7	52.6	31.7	203	95.7
Med	8.39	6.59	9.16	4.20	100	55.7	46.1	38.9	40.6	27.5	200	80.4
Min	0.241	0.238	0.241	2.19	4.11	2.58	0.235	0.227	0.203	11.9	8.70	14.1
Max	27.7	16.4	25.3	7.23	262	188	90.5	121	121	59.8	293	196
Linear												
n	84 (1)	78	77	77	78	72	72 (3)	82	82 (1)	79 (4)	82	88
Mean	2.77	2.64	2.80	4.61	47.8	50.5	22.5	22.7	20.7	27.7	148	81.9
SD	2.08	1.85	1.96	1.66	47.6	42.0	17.2	18.2	16.4	10.9	55.8	43.6
SE	0.227	0.210	0.223	0.190	5.39	4.95	2.02	2.01	1.81	1.22	6.17	4.65
LCL	2.32	2.22	2.36	4.23	37.1	40.6	18.5	18.7	17.1	25.3	136	72.7
UCL	3.22	3.06	3.24	4.99	58.5	60.4	26.6	26.7	24.3	30.1	160	91.2
Med	3.29	3.08	3.26	4.16	21.8	42.9	24.8	23.6	20.0	24.6	156	71.9
Min	0.228	0.237	0.246	2.15	4.18	2.86	0.253	0.267	0.235	14.2	9.44	21.2
Max	7.48	7.32	7.22	9.5	191	165	89.9	59.6	60.4	59.1	271	176

Table 7.2.: Pairwise comparisons from Dunn's test (Hochberg)
Comparisons marked with a circle denote variants different by chance.

Regular			**Spike**			**Linear**		
Variants	Z Statistic	Adj. P-value	Variants	Z Statistic	Adj. P-value	Variants	Z Statistic	Adj. P-value
Video Requests								
• V1 - V3	-0.137624	0.4453	• V1 - V3	-0.100347	0.4600	• V1 - V3	-0.191831	0.4239
• V1 - V2	0.809712	0.6272	• V1 - V2	1.838233	0.0990	• V1 - V2	0.588189	0.8346
• V3 - V2	0.969048	0.6650	• V3 - V2	1.899160	0.1151	• V3 - V2	0.764121	0.8896
V1 - V5	-16.16428	0.0000*	V1 - V5	-9.168365	0.0000*	V1 - V5	-11.08871	0.0000*
V3 - V5	-16.40741	0.0000*	V3 - V5	-8.884146	0.0000*	V3 - V5	-10.66532	0.0000*
V2 - V5	-17.17664	0.0000*	V2 - V5	-10.56171	0.0000*	V2 - V5	-11.46649	0.0000*
V1 - V6	-7.520797	0.0000*	V1 - V6	-5.815485	0.0000*	V1 - V6	-11.19946	0.0000*
V3 - V6	-7.555149	0.0000*	V3 - V6	-5.609327	0.0000*	V3 - V6	-10.78703	0.0000*
V2 - V6	-8.395024	0.0000*	V2 - V6	-7.324578	0.0000*	V2 - V6	-11.57171	0.0000*
V5 - V6	8.166741	0.0000*	V5 - V6	3.015430	0.0064*	• V5 - V6	-0.336892	0.7362
V1 - V4	-8.222174	0.0000*	V1 - V4	3.210086	0.0040*	V1 - V4	-3.517963	0.0013*
V3 - V4	-8.286863	0.0000*	V3 - V4	3.245992	0.0041*	V3 - V4	-3.256100	0.0028*
V2 - V4	-9.164485	0.0000*	• V2 - V4	1.392711	0.1637	V2 - V4	-4.030708	0.0002*
V5 - V4	8.381079	0.0000*	V5 - V4	11.55495	0.0000*	V5 - V4	7.398736	0.0000*
• V6 - V4	-0.251111	0.8017	V6 - V4	8.429938	0.0000*	V6 - V4	7.586034	0.0000*
Playlist Requests								
• V1 - V3	0.465367	0.6417	• V1 - V3	0.464571	0.3211	• V1 - V3	0.465367	0.6417
• V1 - V2	-0.076362	0.4696	• V1 - V2	1.310816	0.2849	• V1 - V2	-0.076362	0.4696
• V3 - V2	-0.560224	0.8630	• V3 - V2	0.855698	0.3922	• V3 - V2	-0.560224	0.8630
V1 - V5	-11.21401	0.0000*	V1 - V5	-7.574920	0.0000*	V1 - V5	-11.21401	0.0000*
V3 - V5	-12.07811	0.0000*	V3 - V5	-8.165797	0.0000*	V3 - V5	-12.07811	0.0000*
V2 - V5	-11.51788	0.0000*	V2 - V5	-9.134892	0.0000*	V2 - V5	-11.51788	0.0000*
V1 - V6	-8.186793	0.0000*	V1 - V6	-3.867309	0.0004*	V1 - V6	-8.186793	0.0000*
V3 - V6	-8.965656	0.0000*	V3 - V6	-4.426219	0.0000*	V3 - V6	-8.965656	0.0000*
V2 - V6	-8.395631	0.0000*	V2 - V6	-5.384961	0.0000*	V2 - V6	-8.395631	0.0000*
V5 - V6	3.323748	0.0031*	V5 - V6	3.972616	0.0003*	V5 - V6	3.323748	0.0031*
• V1 - V4	-1.119305	0.6575	V1 - V4	3.336781	0.0025*	• V1 - V4	-1.119305	0.6575
• V3 - V4	-1.633566	0.3070	V3 - V4	2.955713	0.0078*	• V3 - V4	-1.633566	0.3070
• V2 - V4	-1.078586	0.5615	• V2 - V4	2.211481	0.0540	• V2 - V4	-1.078586	0.5615
V5 - V4	10.33148	0.0000*	V5 - V4	10.64232	0.0000*	V5 - V4	10.33148	0.0000*
V6 - V4	7.217215	0.0000*	V6 - V4	7.178564	0.0000*	V6 - V4	7.217215	0.0000*



(a) V1

(b) V2

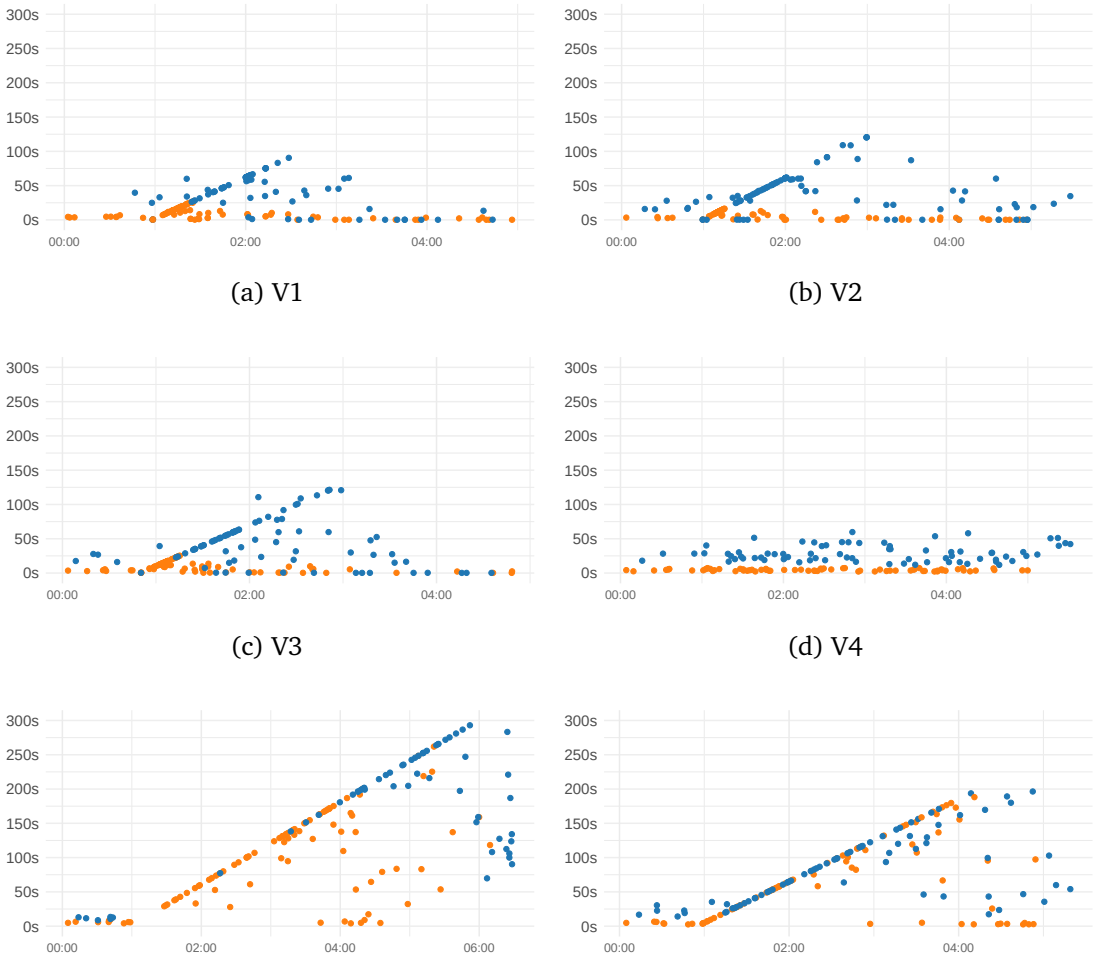
(c) V3

(d) V4

(e) V5

(f) V6

Figure 7.4.|: Latency for the Regular scenario
The orange and blue dots correspond to video and playlist requests, respectively.

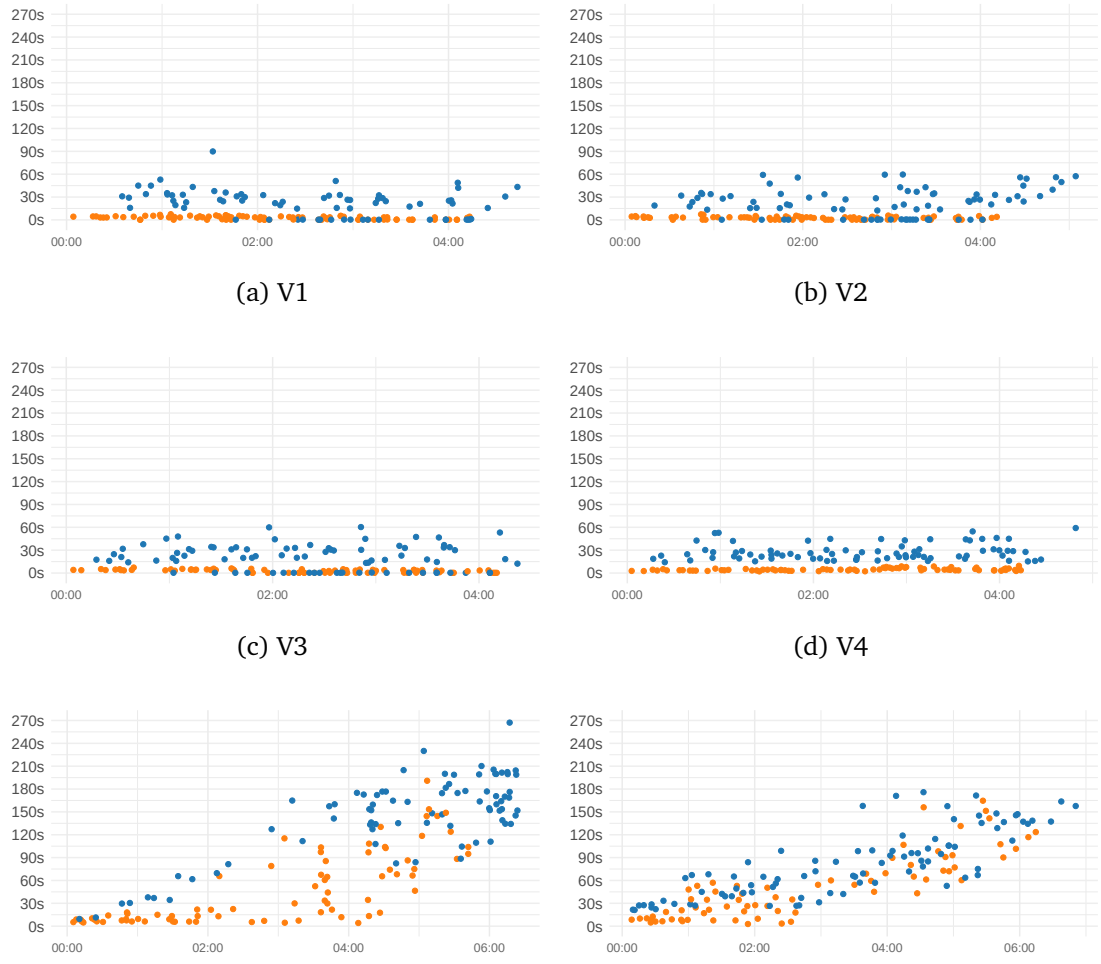


(a) V1 (b) V2

(c) V3 (d) V4

(e) V5 (f) V6

Figure 7.5.|: Latency for the Spike scenario
The orange and blue dots correspond to video and playlist requests, respectively.



(a) V1

(b) V2

(c) V3

(d) V4

(e) V5

(f) V6

Figure 7.6.|: Latency for the Linear scenario
The orange and blue dots correspond to video and playlist requests, respectively.

As shown in Figure 7.4c, the latency for videos and playlists in the Regular scenario is very similar. This does not happen in any other scenario, as shown in Figures 7.5c and 7.6c. A plausible explanation for this is that variants V1, V2 and V3 are not shielded against an increasing number of requests. Even though, on average, 56% of them are cached, the Spike and Linear scenarios put an additional burden.

Figure 7.7 summarizes the results above. The bar charts display the same latency range to allow comparing the variants across scenarios. Despite the number of variants being limited to 6, Figure 7.7 evidences how different architectural configurations of the same application potentially respond better under different execution scenarios.

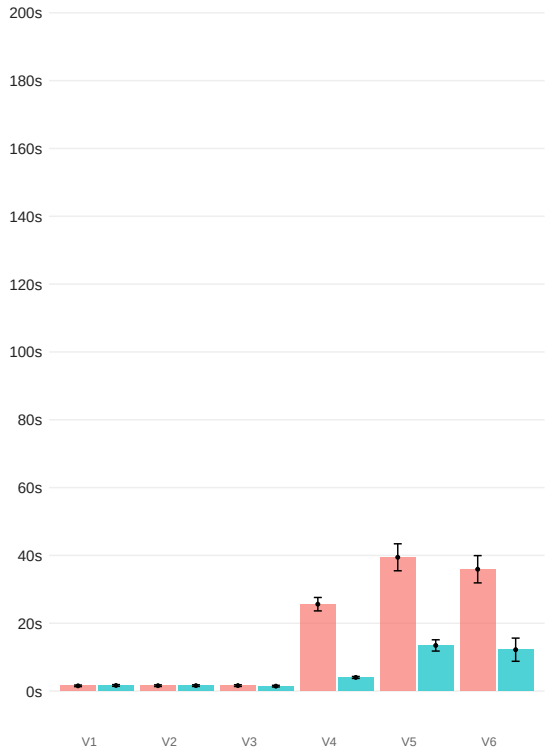
The results presented in this section aim to evaluate the soundness and feasibility of our self-improvement feedback loop. Regardless of the limitations of our implementation, we provide enough evidence to support our long term vision. Autonomic managers can potentially update the managed system at the software level to maximize the use of resources based on current execution conditions, as well as reduce the cost of operation. Thereby, they contribute to the managed system's long-term evolution. Although a complete run of the experiment would provide more accurate results, we believe that comparing these trials is representative of the overall results.

7.1.4. Qualitative Validation

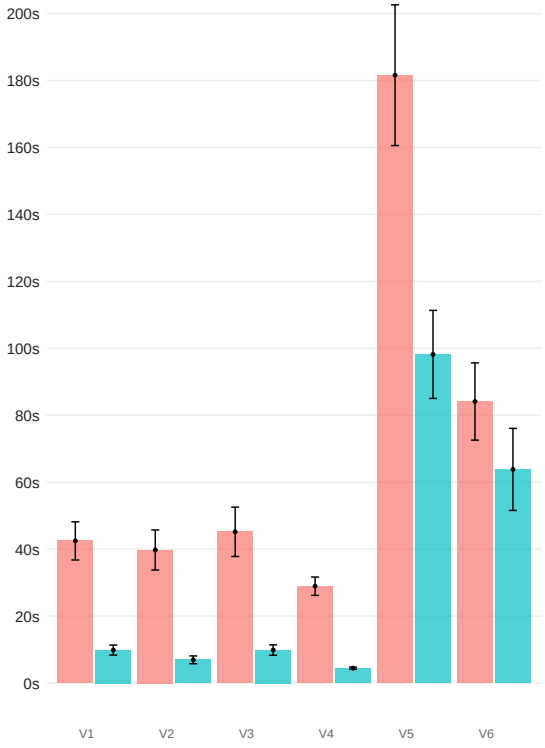
This section discusses two concerns regarding our continuous evolution process. First, Section 7.1.4.1 addresses the continuity and holisticness of our evolution process. And Section 7.1.4.2 addresses our contribution toward reducing the identified discontinuities in the SDLC.

7.1.4.1. Holistic and Continuous Software Evolution Process

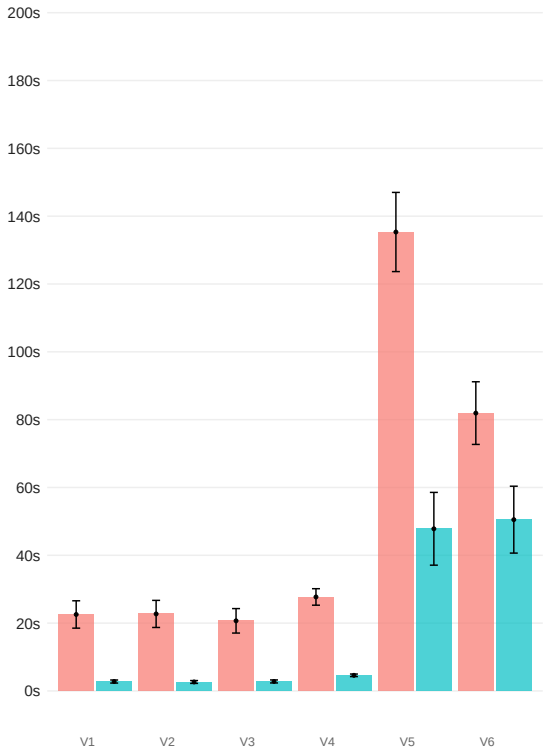
Continuity in continuous software engineering lies in the frequency and scope of development processes. For example, continuous testing reduces mean time to repair by increasing the frequency of test execution. In this and other cases, the outcome would be less effective if the frequency was decreased. Imagine, for example, doing "*continuous*" testing every week, or executing tests per committed code but delaying the fixes until the end of the sprint. Of course, increased frequency and reduced scope have brought myriad benefits to software engineering in practice. However, there is a missing property in continuous engineering, namely explicit causality. By this we mean linking the evolution of artifacts across the delivery pipeline explicitly. This property provides traceability to online changes, as well as automated mapping between online and offline changes. Therefore, as we have shown, explicit causality reduces potential technical debt and has the effect of decreasing friction in the evolution process.



(a) Regular scenario



(b) Spike scenario



(c) Linear scenario

Figure 7.7.|: Mean latency and confidence interval (95%) per variant
The orange and green bars correspond to video and playlist requests, respectively.

The continuity we built into the evolution process is twofold. First, by integrating autonomic evolution actions into CI, we bring autonomic managers closer to software engineering processes, such as continuous testing and quality checking. Therefore, autonomic managers can start contributing to the software evolution with increased trustworthiness—thereby, requiring less supervision. Second, we integrate autonomic managers themselves into the development process and related tooling. This means that autonomic managers can further contribute to reduce other discontinuities. Since our evolution process integrates software and knowledge changes, we extend the explicit causality property to local development environments. Run-time changes and knowledge discoveries are immediately put in service of stakeholders through the tools they used. For example, based on search space explorations conducted in the development environment, and models reified on production, developers can get immediate feedback on their IDE. Therefore, we believe explicit causality can reduce feedback time without affecting the frequency and scope of existing processes.

Explicit linkage across the delivery pipeline expands the scope of the evolution process. Our approach to software evolution goes beyond a bipartite process (*i.e.*, development and execution)—an approach prevalent in autonomic computing. By repurposing self-management capabilities, we contribute to fade away the boundary between development-time and run-time. Traditionally, such boundary has kept autonomic managers' actions from escaping the run-time context and the production environment. Furthermore, by making the entire delivery pipeline available to autonomic managers, we enable them to contribute to the evolution of managed systems at various stages of their development. We add additional concerns rooted in the evolution of development artifacts, not just their execution and management. Therefore, the evolution process we realize is in itself a holistic approach to evolving software systems.

7.1.4.2. Contribution to Reduce Remaining Discontinuities

We identified two remaining discontinuities in the SDLC (*cf.* Section 3.1). In the first case, we pointed out the frequency gap between the evaluation and adjustment of performance inefficiencies. Testing procedures are generally available for detecting potential service degradation in a continuous way. However, its continuous adjustment faces friction for various reasons, such as the trade-offs of implementing business-critical functionality compared to making incremental adjustments without added benefit to users. In the second case, we noticed a disconnect between the run-time and development evolution processes. Despite some development changes happening as a consequence of run-time changes, there is no explicit linkage between the two evolution processes. Our contributions tackle these discontinuities.

Our self-improvement feedback loop found an architecture and infrastructure configuration suitable for a required service latency. From a very simple starting point, our feedback loop was able to introduce new software components, modify configuration parameters, and assess the system based on KPIs. Moreover, such an assessment was conducted as part of the delivery pipeline, meaning that the adjustment our feedback loop provides is

of a continuous nature. Since our self-improvement feedback loop is built on top of the evolution pipeline, it reduces human intervention by creating merge requests. Therefore, developers only need analyze the evidence presented in the repository management tool to implement the proposed changes. Thus, our feedback loop reduces not only experimentation friction, but also implementation friction.

Our continuous software evolution pipeline effectively synchronized artifacts on the development and execution sides. It provides a consistent and explicit way of connecting run-time adaptations with software changes. Moreover, since the MARTs it uses are accessible by autonomic managers, our evolution pipeline separates short-term and long-term evolution concerns. Thus, it allows them focusing on domain-specific updates while the corresponding software changes are introduced automatically. At the same time, it contributes to closing the gap between the development-time and run-time evolution processes. Thereby, our pipeline contributes to reduce evolution friction, which presents itself in the form of technical debt.

7.2. Smart Urban Transit System Case Study

This section contains the proof of concept validation of our contributions based on the smart urban transportation case study. It is organized in two parts, as follows. First Section 7.2.1 introduces our concrete case of application. And second, Section 7.2.2 presents the application scenarios concerning functional validation.

7.2.1. Concrete Case of Application: The MIO Transportation System

In this research, we employ historical data from the BRT system of Cali—the third-largest city of Colombia with a population of nearly 2.2 million. Cali's transportation system is called MIO, for its acronym in Spanish. At the end of 2019, the MIO system had 98 bus lines (*i.e.*, routes) and mobilized around 455,000 passengers on a business day, using a fleet of 828 vehicles. The MIO system is considered as a developing CPS in which physical entities and actions (*e.g.*, buses, routes, bus drivers and bus schedules) are managed by human planners and controllers from Metro Cali—the company operating the system. Ideally, these operators would perform control actions supported or predicted by systems in the cyberspace. By simulating our application scenario with real data, we aim to demonstrate the effectiveness of our design approach for CPSs.

As expected, the MIO system is affected by the problem we described in our SUTS case study. On the one hand, Metro Cali is unable to make any guarantees about the effectiveness of daily control actions aimed to meet Service Level Objectives (SLOs). On the other hand, planners cannot reliably anticipate the long-term effects of adjustments to the OSP. Modifications are uncertain and mostly empirical. As a result, Metro Cali is unable to fulfill operation plans.

7.2.2. Functional Validation

We evaluated our reference architecture by using the proposed models and following its guidelines for modeling and implementing prototypes of the main components for our SUTS case study. With this, we show the consistency and feasibility of our reference architecture for guiding software architects on how to apply it for engineering SCPSs. We validate the prototypes using concrete dependability and resiliency domain-specific concerns, such as the cost associated with a particular operating fleet size, the average EWTABS per passenger, and the HCoV [**?**, **?**, **?**] (*cf.* Figure 3.6). Following the separation of concerns principle, we split the evaluation into two parts. In the first part, Sections 7.2.2.1 and 7.2.2.2 describe the artifacts modeling the physical system; Section 7.2.2.3 presents the components we developed for the MIM; And Section 7.2.2.4 presents our implementation of the AM. In the second part, Section 7.2.2.5 presents the application scenario and discusses the results.

7.2.2.1. The Managing System's Modeling Layer

Our proposal of the modeling layer comprises two model types. The first one is a graph representing the topology of stations, stops and lines. Figure 7.8 depicts these elements using a graphical notation. The graph contains four stations, represented by white circles, each containing two stops—one for each direction. The second model type is the probability distributions of bus service times and bus and passenger arrivals per line and station. Out of the 98 lines, we selected the most crucial one for the passengers of MIO, namely the *T31*. Crossing the city's heart in a north-south direction, this route interconnects some of the most important and emblematic sectors for locals and visitors. It plays a fundamental role in the dynamics of the city from an economic, mobility, tourism, and social perspective.

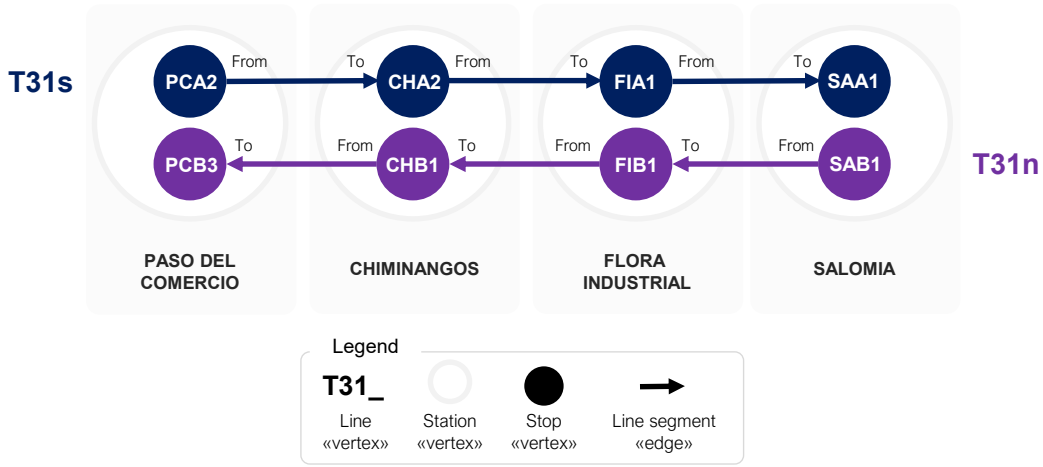


Figure 7.8.|: Topology of stations, stops and lines of the MIO system

To obtain the probability distributions of service time and arrival of passengers and buses at each stop selected for this evaluation, we followed the data preparation process

depicted in Figure 7.9. First, data provided by Metro Cali was available in the form of periodic data frames generated by buses in operation about every 200ms (*cf.* ❶ in Figure 7.9). Each data frame contains information about the date, the line being served, the bus coordinates, the task being fulfilled, among other data produced by the onboard bus sensors. We stored one year of data from 2019 in a local Hadoop cluster (*cf.* ❷ in Figure 7.9). Then, we selected a representative day of operation for the MIO system and used Apache Hive to filter data frames accordingly (*cf.* ❸ in Figure 7.9). Next, we used Python to reformat the data set and generate a new one with data frames filtered by line *T31s* and by the operation hours selected for the simulations (*cf.* ❹ in Figure 7.9). We chose a time frame from 5:00 to 10:00, as these are critical operation hours in the system, where our research could have the biggest impact. Subsequently, we developed a simple Java program to synthesize a data set with bus service and interarrival times for each of the selected stops (*cf.* ❺ in Figure 7.9). The program uses the GPS coordinates reported by the buses to estimate whether they are in the close vicinity of the bus stop (*i.e.*, a bus arrival). Then, it calculates the time it takes a bus to provide service (*e.g.*, passenger disembarkation and boarding) to leave the stop toward its next destination. Finally, we used *Flexim*'s *ExpertFit*[13] module to fit probability distributions from interarrival and service times data, and identify an appropriate distribution for simulation and experimentation modeling. To obtain the distributions of passenger interarrivals, we used estimated data from the stations provided by Metro Cali. In accordance with available probability distribution implementations, we selected distributions that best fitted the data (according to ExpertFit's reports) and used them in the current implementation of our developing simulation framework. Table 7.3 presents the distributions identified with their corresponding parameters.

Table 7.3.: Reference probability distributions used for simulations

Line	Stop	Bus Arrival			Passenger Arrival					Service Time		
		Distribution	Shape	Scale	Distribution	Alpha1	Alpha2	Min	Max	Distribution	Shape	Scale
T31s	PCA2	Constant (mean = 335)	N/A	N/A	Johnson SB	1.70271	0.74959	23.89094	547.41560	Log-Logistic	7.60901	32.47336
	CHA2	Log-Logistic	2.77931	423.89955	Johnson SB	1.90231	0.90182	39.87878	771.46872			
	FIA1	Gamma	2.14470	223.92435	Johnson SB	1.78292	0.83046	49.94927	1012.29896			
	SAA1	Weibull	1.66242	562.48870	Johnson SB	1.83560	0.85613	51.89543	1119.29601			

Figure 7.10 shows the density-histogram plots generated by ExpertFit for the identified distributions of bus interarrival times for each of the bus stops selected in this study. To have control over the buses injected into the line during the intended simulations, we avoid using a bus interarrival distribution for the first stop in the line. Instead, it is configured as an input parameter. To facilitate the service time modeling in our simulation framework, we used a single distribution (Log-Logistic) that represents the service time be-

[13] https://www.flexsim.com/expertfit

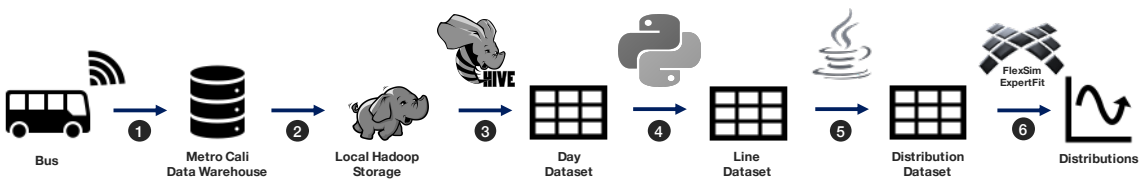
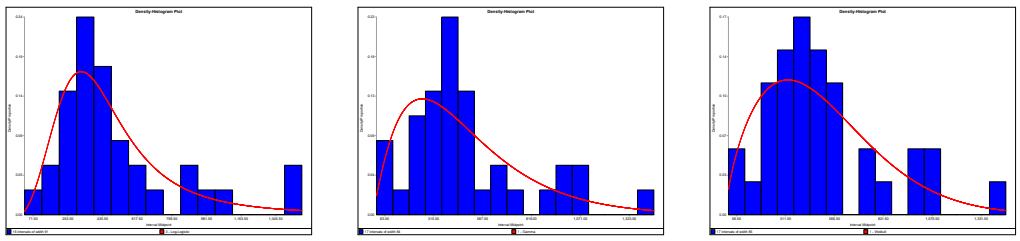


Figure 7.9.|: Data pre-processing workflow



(a) CHA2 stop (b) FIA1 stop (c) SAA1 stop

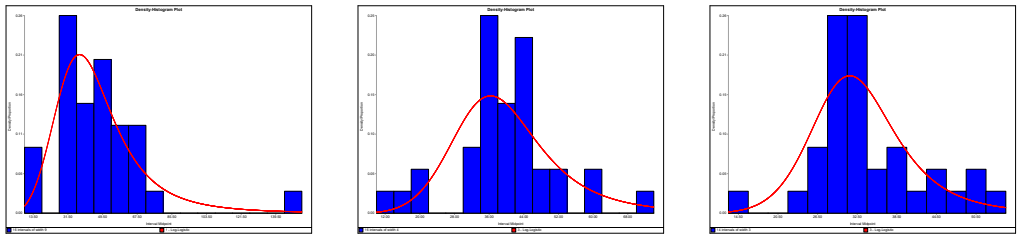
Figure 7.10.|: Density-histogram plots for reference bus inter-arrival times

havior reasonably well in the city sector containing the selected bus stops (*cf.* Figure 7.11). Figure 7.12 shows the density-histogram plots for the identified distributions of passenger interarrival times.

7.2.2.2. Model Identification Through A Topology-Conforming Simulation

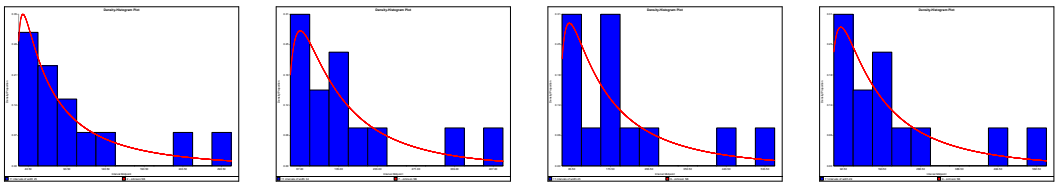
From the modeling layer definition given in the previous section, we developed a conforming simulation using a queuing network to support run-time decision making and assurance. The queuing network captures essential constraints and behaviors of the physical system, allowing to compute KPIs, such as the average bus and passenger queue length, observed line headway, observed EWTABS per line, and any other metric that can be computed from them, such as the HCoV. The queuing network enables the managing system to configure and simulate intrinsic characteristics of the managed system. In this evaluation, the intrinsic characteristics are related to elements from the OSP: the headway design and the number of buses per line, for a particular time range (*i.e.*, from 05:00 to 10:00).

Figure 7.13 depicts the queuing network configuration. A `station` groups a set of `stops`, by which `buses` stop to pick up `passengers`. Notice that we did not model passenger drop off because Metro Cali currently does not track this information. A line starts and ends at different stations, and has a frequency described by the headway design. When a `bus` arrives at a stop, there may be a queue of buses waiting for their turn to pick up passengers. The time a bus stays at each stop is described by the service time associated with the bus's



(a) CHA2 stop (b) FIA1 stop (c) SAA1 stop

Figure 7.11.|: Density-histogram plots for reference service times



(a) PCA2 stop (b) CHA2 stop (c) FIA1 stop (d) SAA1 stop

Figure 7.12.|: Density-Histogram plots for reference passenger inter-arrival times

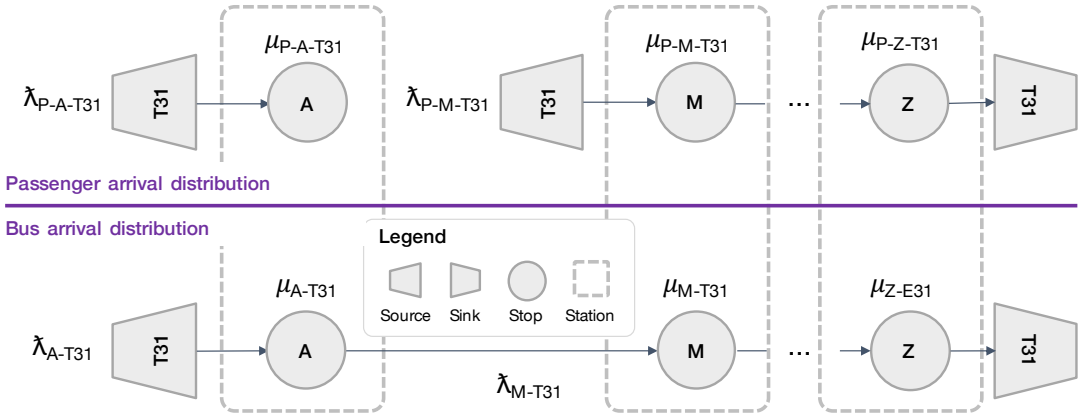


Figure 7.13.|: Queuing network configuration
This figure exemplifies the queuing network configuration for bus and passenger arrival distributions. The top half of the figure shows passenger arrivals (*e.g.*, $\lambda_{P-A-T31}$) and waiting times (*e.g.*, $\mu_{P-A-T31}$) for station A and line T31. The bottom half of the figure shows bus arrivals (*e.g.*, λ_{A-T31}) and service times (*e.g.*, μ_{A-T31}) for the same station and line.

line at that particular station. Each stop processes the arrival of passengers for a particular line. Since we do not model passenger drop off, the queuing network assumes that all passengers head to the last stop.

The queuing network is instantiated based on the graph model described in Section 7.2.2.1. All random variables and constants are input parameters, therefore they are subject to change according to hyper-parameter optimization and the probabilistic distributions from the modeling layer. Our approach to modeling the transportation system differs from others mainly in its dynamicity and run-time modifiability.

7.2.2.3. Multi-Model Identification Mechanism (MIM)

Our implementation of the MIM uses evolutionary optimization and function approximation to infer the EWTABS function. We use a genetic algorithm to find and evaluate values for the headway design and the operating fleet size for each line. The evaluation uses the simulation described in Section 7.2.2.2. Since the simulation conforms to the current system topology model, the MIM will always find an appropriate model for the system's

current contextual situation. Although our evaluation concentrates on estimating a function, another implementation of the MIM could optimize the topology model itself, aiming at improving the user experience (*i.e.*, reducing the EWTABS by modifying the lines' configuration).

Next, we describe the implementation approach to each of the techniques we used.

7.2.2.3.1. Evolutionary optimization We implemented a genetic algorithm using the *Jenetics* library,[14] initially configured as follows: an initial population of 30 chromosomes; a mutation probability of 1%; and a single-point crossover with a probability of 50%. Moreover, we configured the stop condition of the algorithm based on two conditions: a steady fitness is found over a desired number of generations, or the algorithm reaches a maximum number of pre-established generations.

Each chromosome contains two numeric genes, namely the number of buses (integer) and the headway (double) for a particular line at a specific time range. Based on these two numbers, the fitness function instantiates and runs 10 replicas of the queuing network simulation. Then, it computes the fitness value and stores the collected metrics. The fitness function comprises five components, using three different functions as follows.

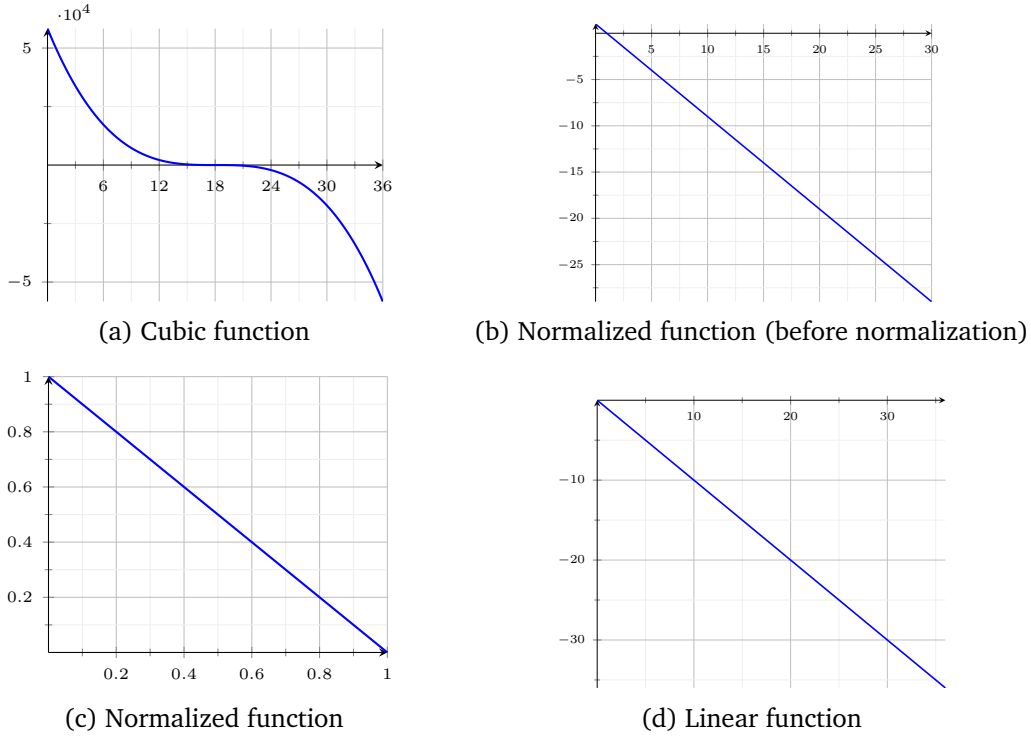
Cubic function The first function is $f(x) = -10(x-b)^3$ with domain $(a,\ c)$ where $a \leq b \leq c$. This function translates $-x^3$ to the right and stretches it so that $f(x)$ is a positive number and is greater when x tends to a, and is a negative number and smaller when it tends to c. We use this function to penalize chromosomes with a number of buses greater than the ones in the OSP, and to reward those using fewer buses. If x is outside the domain, the function returns positive infinity. Figure 7.14a shows an example of this function with a minimum number of buses of 0 (*i.e.*, there is no demand of a particular line at the specified hours), a planned number of buses of 18, and a maximum number of buses of 36 (*e.g.*, the fleet size for that particular line).

Normalized function The second function is $g(x) = -x + 1$, where $d \leq x \leq e$. The result is greater when x tends to d, and is smaller when it tends to e. As the name implies, the function returns a normalized value using the inverse domain $(d,\ e)$. If $x < d$ or $x > e$, the function returns -1, meaning that d and e are the minimum and maximum acceptable values. Figure 7.14b shows an example of this function with $d = 0$ and $e = 30$ before applying normalization. Figure 7.14c shows the same example after normalizing the output. We use this function for two components of the fitness value. In the first case, we use it to reward chromosomes with a headway smaller than the initial headway design. And in the second case, we use it to explicitly reward chromosomes that produce a small EWTABS.

Linear function The third function is $h(x) = -x$, where $x \geq k$. If $x < k$, the function returns negative infinity, meaning that k is the minimum accepted value. Figure 7.14d shows an example of this function, which we use to reward chromosomes with a small headway coefficient of variation, and a small variance of EWTABS.

[14] https://jenetics.io/

The fitness function is defined as $0.2 \cdot f(buses) + 0.3 \cdot g(observed\ headway) + 0.3 \cdot g(\overline{EWTaBS}) + 0.1 \cdot h(HCoV) + 0.1 \cdot h(Var(EWTaBS))$, where $a = 0$, $b = planned\ buses$, $c = fleet\ size$, $d = minimum\ headway$, $e = maximum\ headway$ and $k = 0$. Values b, c, d and e are assigned according to each line. Argument $buses$, is taken directly from the chromosomes, whereas the rest are computed using the simulation. The fitness function rewards maximizing each component, knowing that: using fewer buses reduces the cost of operation but increases the average EWTABS; and at the same time, using a smaller headway decreases the EWTABS but increases the number of buses in operation. Since there are trade-offs among the function arguments, we assigned the weights according to how relevant they could be for Metro Cali.



(a) Cubic function

(b) Normalized function (before normalization)

(c) Normalized function

(d) Linear function

Figure 7.14.|: Components of the fitness function

7.2.2.3.2. Function approximation We implemented two model inferrers to approximate the EWTABS function (*cf.* Figure 6.2). In both cases, the data points used as samples come from the genetic algorithm described in Section 7.2.2.3.1—*Evolutionary Optimization*. Since the genetic algorithm explores the solution space, from an evolutionary viewpoint, the metrics computed as part of the fitness function will provide a more accurate approximation over time. The first inferrer is based on function interpolation and uses natural bicubic splines. The second inferrer uses symbolic regression as follows. The inputs to the algorithm are the collected data points, which are of the form (*number of buses*, *headway design*, *average* EWTABS), and a set of terminals that represent possible operators to apply during the space exploration. Based on trial and error experimentation, we arrived at the following list of operations: addition, subtraction, multiplication, exponentiation and

square root. We configured the genetic algorithm to use the mean square error as loss function with a maximal tree depth (*i.e.*, height) of 5. Additionally, we specified three possible stop conditions: a minimum acceptable estimated error of 0.01, a total number of 10,000 generations, and a maximum execution time of 1 minute. To alter the offspring population we used a single node crossover with a 10% probability and a mutator with a 1% mutation probability.

In our experience, the symbolic regression algorithm will not always converge to a successful result. In most cases this happened because the stop condition (*e.g.*, the number of iterations) was triggered before the algorithm converged, thus, failing to reach an acceptable estimated error. Since this is a risk inherent to evolutionary optimization, we use the interpolated function as a fallback mechanism. This is because function interpolation guarantees termination and its estimated error highly depends on the density of the samples. Therefore, it is likely to improve over time.

7.2.2.4. Multi-Adjustment Mechanism (AM)

Our implementation of the AM finds the argument to the reference function (*i.e.*, the inferred function described in Section 7.2.2.3.2—*Function Approximation*) that best approximates a given target value (*i.e.*, the control objective). For example, given a desired EWTABS of 5 minutes, the AM minimizes the distance between the reference function and the point $(0, 5)$, where the first argument is the headway design. That is, the function $d(x) = \sqrt{(x-0)^2 + (f(x)-5)^2}$, where $f(x)$ is the reference function.

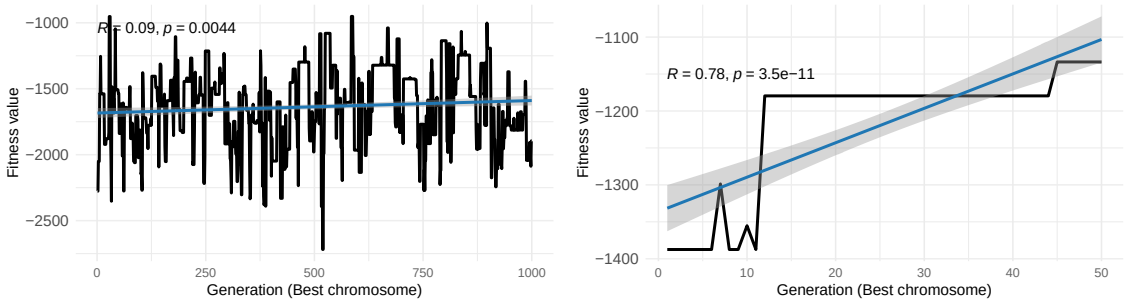
We used the gradient descent method to minimize the aforementioned distance equation. To do so, we implemented automatic differentiation of both the reference and distance functions. We only implemented one parameter estimator. However, other implementations would only differ in the optimization part.

7.2.2.5. Application Scenario: Model Identification and Run-Time Adjustment

The results of the performed evaluation comprise two parts. The first one concerns the effectiveness of the MIM to approximate the EWTABS function. The second one is about the AM's ability to select appropriate parameters for the same function such that the transportation system meets its goal (*i.e.*, to achieve a target EWTABS).

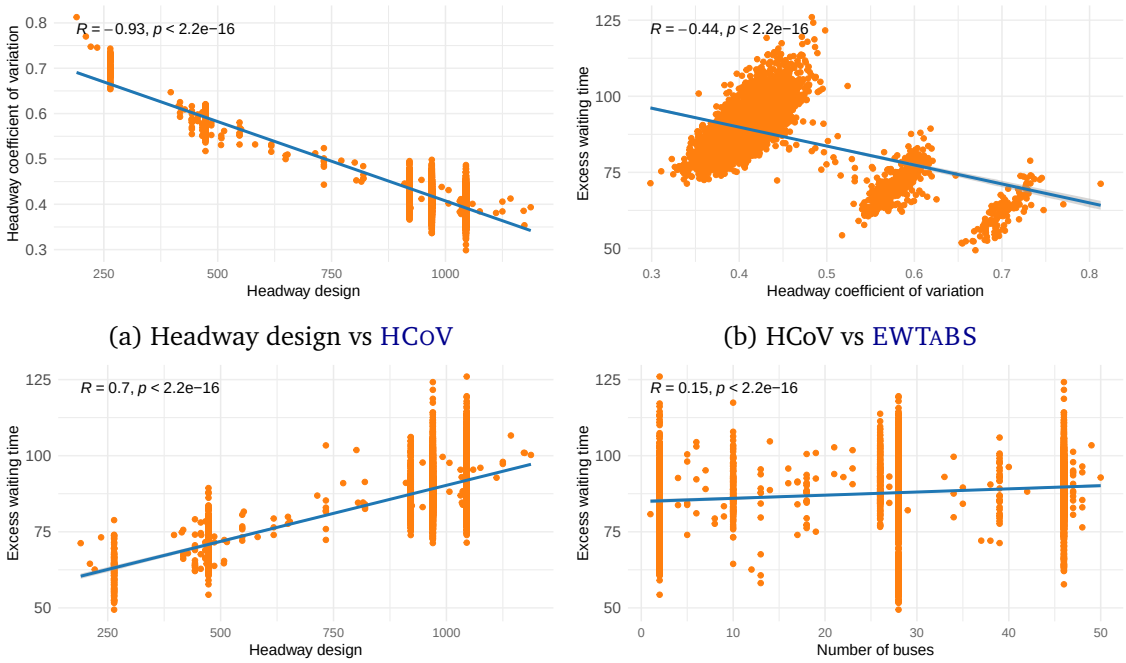
The genetic algorithm was configured to explore the evolution space. As a result, it varies the number of buses and the headway design without describing a particular direction. Figure 7.15 shows the overall fitness performance over (simulated) time. Both figures 7.15a and 7.15b plot the fitness value of the best chromosome for each generation. For comparison, we have included Figure 7.15b, which displays the performance of the genetic algorithm when configured to optimize the fitness value. The chart only displays 50 generations because the algorithm quickly converged. These two modes of operation exemplify the purpose of the MIM and the AM. The first one concentrates on generating

enough information to synthesize active knowledge. The second one is concerned with finding appropriate control parameters regardless of the contextual situation. In fact, the density of the generated data points dictates how reliable are the inferred models.



(a) Exploration mode (b) Optimization mode

Figure 7.15.|: Overall fitness performance over time



(a) Headway design vs HCoV (b) HCoV vs EWTABS

(c) Headway design vs EWTABS (d) Operating fleet size vs EWTABS

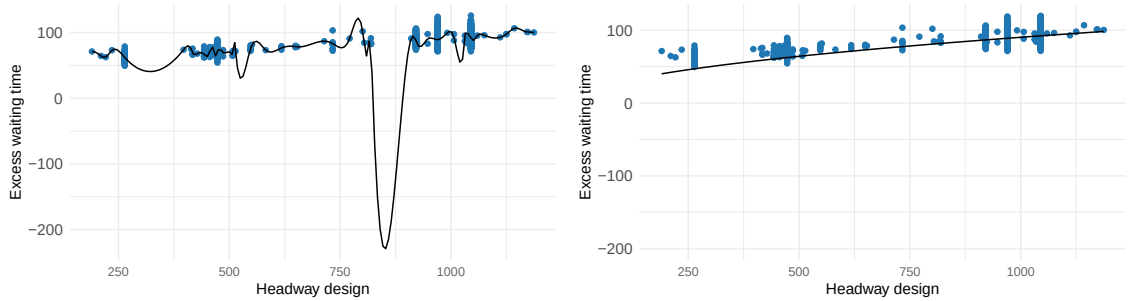
Figure 7.16.|: Correlation and behavior of independent variables and measured metrics with respect to the excess waiting time

The simulation model was crucial to evaluate concrete combinations of parameters and estimate the metrics of interest. Figure 7.16 displays various abstract views depicting the behavior of independent and dependent variables. Figure 7.16 shows several charts depicting the correlation of independent variables and measured metrics with respect to the EWTABS. Figure 7.16a plots the effect of the headway design on the HCoV. As expected, the direction of the regression line, along with the Pearson correlation coefficient, confirms that increasing the headway decreases the variation. In this case, the correlation coefficient and mean square error are valuable metrics to predict how reliable the approximated

function can be. This can be useful when several model inferrers or parameter estimators are available. Figure 7.16b depicts the effect of the HCoV on the EWTABS. This result is consistent with the previous one: a large HCoV, meaning that the headway is actually small, means that passengers will wait less time on average. This kind of linear relationship between the headway design and the EWTABS is also evidenced in Figure 7.16c. The clusters of points in Figure 7.16b are a reason to suspect that the genetic algorithm may need to increase the crossover and mutation probabilities. Hyper-parameter optimization can help spread the generated headway values, thus potentially improving the accuracy of the approximated function. Finally, Figure 7.16d shows that neither increasing nor decreasing the number of buses affects the EWTABS. This means that these two variables are not correlated (*cf.* the Pearson correlation coefficient), therefore the former should not be included in the distance function optimized by the AM.

Once the genetic algorithm has explored the adaptation space, the model inferrers can approximate the function. Figures 7.17a and 7.17b plot the approximated functions. On the one hand, the function approximated by interpolation produces accurate-enough results in areas with enough data density. However, there is a segment of the domain where the function returns a negative excess waiting time. The AM can include validations over basic assumptions such as this one. In this case, since a negative waiting time is impossible, a simple failover strategy may provide a better alternative. On the other hand, the function approximated by symbolic regression describes a more consistent, smooth curve. Such a curve is described by the function $f(x) = sqrt(9 + (((7 + 2x) + (x + sqrt(6 + ((8 + 2x) + ((4 + 2x) + (x + ((4 + 2x) + 9)))))))) + (6 + ((9 + (x + ((6 + 2x) + 6))) + 2x))))$, with a mean square error of 62.37 seconds.

The AM is able to select a headway design correctly for a given a target excess waiting time. For example, an excess waiting time of 1.33 minutes is possible with a headway of 13.17 minutes. Of course, since the CPSs is in constant evolution, available MIMs will be constantly updating the reference models. The AM will always use an up to date reference input, decreasing the likelihood of unintended effects.



(a) Interpolated function using natural cubic splines

(b) Approximated function using symbolic regression

Figure 7.17.|: Approximated functions

We have demonstrated the feasibility of our reference architecture by describing the model and implementation suitability of two prototypes for its major components. These

components contribute to the achievement of dependable autonomy and operational resiliency in the design of SCPSs. Our experimental results show that the MIM was able to approximate a function to describe the EWTABS using two estimators, based on cubic interpolation and symbolic regression (*cf.* Figure 7.17 and Section 7.2.2.3.2). For this, the MIM exploits its exploration and optimization capabilities to synthesize and examine the effectiveness (*cf.* Figure 7.16) of alternative configurations of the system (*i.e.*, through the genetic algorithm described in Section 7.2.2.3.1—*Evolutionary Optimization*). In other words, the MIM is capable of synthesizing active and passive knowledge in the form of data, executable models or functions. These artifacts are useful for considering adaptation scenarios and evaluating their possible outcomes. Consequently, this component constitutes a baseline from which the adaptation and operational spaces are eventually devised and evaluated before their manifestation. We showed how this transition from evolution to adaptation manifests in our reference architecture using our second prototype, the AM. This contributes to the realization of our envisioned continuous engineering cycle.

The results obtained in this study constitute a promising step forward toward the achievement of SUTSs and related types of autonomic CPSs. Our reference architecture promotes the exploration and validation, and exploitation of system adaptations at run-time (*cf.* Section 7.2.2.3). This contributes to increase the dependability of control actions by providing guarantees about their potential effectiveness in managed systems. Moreover, our architecture delineates the way models can evolve at run-time to face the challenges associated with managing dynamic real-world entities (*cf.* Sections 7.2.2.3.2 and 7.2.2.4). With this, we expect to enhance overall system resiliency by enabling the system to mitigate run-time conditions that can not always be anticipated during design time.

7.3. Chapter Summary

This chapter presented the evaluation of our contributions. We conducted functional, experimental and qualitative proof of concept validation of representative elements from each contribution. The two case studies we presented previously in Chapter 3 guide the validation concerns we addressed.

Based on the infrastructure management case study, we validated functional concerns of our continuous software evolution pipeline. We conducted this validation considering two application scenarios, namely: assistance with knowledge acquisition from deployed resources, and prevention of configuration drift for IAC specifications. In both cases, we described implementation details of our functional prototypes and walked through the steps to demonstrate how our implementation contributes to achieve the validation scenario. In the first case, we showed the process to generate the IAC template and instance corresponding to the deployed resources. In the second case, we described the functionalities of our prototype addressing configuration drift. Based on these application scenarios, we validated the soundness and feasibility of our evolution pipeline.

We also evaluated our self-improvement feedback loop based on the infrastructure management case study. We conducted an experimental validation of our proof of concept focused on the exploration of software architecture variants. We described implementation details, experimentation setup, and experimentation results. Our results evidenced the capacity of our self-improvement feedback loop to contribute to the managed system's long-term evolution. Based on the validated application scenario, we demonstrated the feasibility of our approach.

In addition to functional and experimental validation, we validated qualitative concerns of our continuous evolution process, as well as its contribution to reduce the two discontinuities we described in Chapter 3. Based on the SUTS case study, we evaluated our run-time evolution reference architecture as follows. We conducted a functional validation of our reference architecture focused on an application scenario regarding model identification and run-time adjustment. We described implementation details of our prototype, including: the modeling layer, which includes a graph-based topology of the transportation system, and statistical distributions for bus and passenger arrival; a proof of concept implementation of the MIM based on symbolic regression and function interpolation; and a proof of concept implementation of the AM based on gradient descend. Based on this scenario, we showed the consistency and feasibility of our reference architecture for guiding software architects on how to apply it for engineering SCPSs. We validated the prototypes using concrete dependability and resiliency domain-specific concerns, such as the cost associated with a particular operating fleet size, the average EWTABS per passenger, and the HCOV.

The next chapter summarizes the work we presented in this dissertation. It highlights our contributions and discusses potential future work.

Part IV

Summary

Chapter 8

Conclusions

Contents

In this chapter, we present a dissertation summary by revisiting the research challenges and goals we addressed, as well as the corresponding contributions. In light of the latter, this chapter concludes this dissertation with a discussion of future research opportunities.

8.1. Dissertation Summary

In this dissertation, we have addressed and solved several challenges regarding the problem of self-evolution. The fundamental motivation behind our research concerns the extension of autonomic computing from execution to development. We focused on two key aspects of this task. On the one hand, we explored the notion of self-improvement on the development side. We addressed the implicit traditional assumption, in both software engineering and autonomic computing, that, apart from stakeholders' work, nothing else can be done to modify the software while it is being developed. In other words, software changes originate with stakeholders. On the other hand, we identified and addressed software evolution concerns stemming from the automation of development aspects, such as the validity, quality and effectiveness of changes enacted online.

The research problem addressed in this dissertation was to contribute to the incremental improvement of software design, configuration and deployment continuously and autonomously. Our solution strategy is fourfold. First, we devised a framework to integrate development-time and run-time autonomic work based on software changes and knowledge. Second, we developed a feedback loop to monitor and identify run-time variability,

and realize corresponding software changes to configuration source code. Third, we developed an experimentation-driven feedback loop to explore, discover and realize software design and configuration alternatives, aiming to improve KPIs. Finally, we designed a reference architecture for run-time software evolution with two alternating operational modes, namely self-adaptation and self-evolution. Its main goal is to keep knowledge artifacts relevant in the face of emerging behavior, aiming to reduce maintenance work.

8.1.1. Addressed Challenges and Goals

We constrained the research problem by stating a set of research challenges, which we classified into two groups: *self-evolution* and *self-improvement*. In the following, we summarized how we solved the addressed challenges stated for this dissertation.

Self-evolution

CH1: *Self-evolution mechanisms should be compatible with practices of continuous software engineering for quality assurance*. We developed a software evolution pipeline based on the practices of CD (*i.e.*, the delivery pipeline) and CI. Our pipeline allows autonomic managers to realize code updates through run-time domain-specific model modifications. Therefore, autonomic managers exploit existing infrastructure for software testing and quality checking, through the use of repository management and contribution models provided by our evolution pipeline.

CH2: *Autonomic managers should provide the criteria and decision-making process behind a change*. The implementation strategy of our self-improvement feedback loop is based on experimentation and evolutionary optimization. Therefore, autonomic managers can present stakeholders with descriptive statistics of the executed experiment trials, in terms of the subject KPIs, and pair-wise trial comparisons in the form of raw data and charts. This information constitutes statistical evidence that explains the decisions behind a proposed improvement.

CH3: *Self-evolution mechanisms should capture live infrastructure modifications and propose corresponding software changes*. Our FORK AND COLLECT provides a simplified view of the computing infrastructure that a run-time agent uses to trigger a transformation chain. As a result, the agent captures the infrastructure's current state as a MART. Our pipeline's evolution coordinator compares the current and new MARTs, and triggers a source code update, as well as an IAC template instance update, when changes are detected.

Self-improvement

CH4: *Self-improvement mechanisms should use design and analysis of experiments to explore system alternatives for the dimensions of interest*. Our feedback loop for self-improvement separates concerns by dedicating one feedback loop to manage

high-level experimentation goals (*i.e.*, The E-FL), and two others for configuration and architecture variants (*i.e.*, The P-FL and C-FL, respectively). The E-FL directs the search space exploration based on pair-wise trial comparisons and a specific KPI goal.

CH5: *Self-improvement mechanisms should factor execution conditions into the experiment design*. Our self-improvement feedback loop reuses load testing procedures already present in the source repository. By doing this, it ensures that system variants are tested following appropriate quality concerns. That is, variants are assessed based on what the software development team considers relevant.

CH6: *Autonomic managers are required to keep their internal models relevant, aiming to reduce the need for additional maintenance*. Our reference architecture for run-time evolution is based on MRAC and MIAC. The former allows adapting adaptation mechanisms to new reference models. The latter allows identifying, reifying and updating the reference models. Our architecture alternates between these two components, going from short-term adaptations to long-term evolution actions according to changes in the system's environment.

In light of these challenges, we pursued the general goal of creating an autonomic and continuous software evolution process. We reconceptualized and repurposed methods of autonomic computing in light of advances in continuous software engineering. More specifically, we developed self-evolution mechanisms through self-improvement, self-regulation and self-management, and integrated them into the software delivery pipeline. We refined this goal by adopting four specific goals, as follows:

G1: We established a general solution strategy for integrating offline and online software changes and knowledge.

G2: We developed a software evolution pipeline to capture run-time changes in the computing infrastructure and realize corresponding software changes on the development side.

G3: We developed self-improvement driven by experimentation and evolutionary optimization to devise, execute, and monitor architecture and infrastructure variants, aiming to improve KPIs autonomously and continuously.

G4: We designed a reference architecture for run-time software evolution to evolve the reference models guiding adaptations to the managed system.

In the following section, we discuss our achieved contributions with respect to these goals and challenges.

8.1.2. Contributions

We presented our contributions according to their role in our offline and online reconceptualization. First, our continuous software evolution pipeline bridges run-time changes with

corresponding updates on the development side. That is, it connects the production and integration environments from the software delivery pipeline. Second, our quality-driven self-improvement feedback loop explores configuration alternatives to improve KPIs. It does so on the development environment. Finally, our run-time evolution reference architecture guides the design of dependable autonomic behavior and resilient operation. It focuses on run-time aspects of managed systems, thereby occupying the production environment. We summarize these contributions, as follows:

C1: Continuous Software Evolution Pipeline. To address goal G1, we proposed a continuous software evolution pipeline to bridge offline and online evolution processes. We took advantage of existing infrastructure and automation intended for software delivery, and put it into service of autonomic managers. Our pipeline extends the concept of CI to complete the evolution loop in DevOps (*i.e.*, Dev $\leftarrow$ Ops). That is, software changes stem not only from offline processes—driven by stakeholders, but also from online activities—driven by autonomic managers. Therefore, our pipeline enables feedback loops to produce online changes through MARTs, focusing on domain-specific modifications rather than low-level code updates. The integration of these updates into source specifications follows standard practices of software engineering, including repository management, branching and contribution models, among others. This means that online modifications to source code go through existing testing procedures to guarantee minimum and acceptable levels of quality. Conceptually, our evolution pipeline extends CI, thus realizing two-way CI. Thereby, our software evolution pipeline realizes a holistic and continuous process that connects software execution and development, while providing quality control to online changes.

To address goal G2, we devised a round-trip engineering strategy to reconcile the evolution of run-time and development-time artifacts. From left to right (*i.e.*, forward engineering), we rely on the delivery pipeline as it is, considering only a few alterations. From right to left (*i.e.*, reverse engineering), we put in place a model-driven system to monitor and transform run-time changes into various types of persistent updates. Our reverse engineering strategy comprises two main components: run-time state synchronization and automatic source specification update. We separate these workflows based on their primary concern: MART evolution and specification update. The former is based on our FORK AND COLLECT algorithm for API monitoring and MART evolution, while the latter is based on textual and structural model transformations.

To evaluate this contribution, we conducted two application scenarios focused on functional validation. Our validation walks through, step by step, how our prototypes work. The first scenario addressed the creation of an IAC template and its corresponding instance from resources deployed to a target cloud environment. This scenario validated the feasibility of our model transformation chain, our algorithm for run-time cloud monitoring and MART evolution, and the integration of software tools from the IAC life cycle. The second scenario addressed how the pipeline handles changes on either side of the integration loop to prevent configuration drift. We evaluated the effectiveness of the direct and CI-aware evolution workflows, which determine whether run-time changes are integrated following standard practices of quality checking or bypasses them. These application scenarios demonstrate how our software evolution pipeline integrates online changes into the

software evolution process on the development side. Thereby, our pipeline directly contributes to eliminating a remaining discontinuity from the software development process, namely continuous evolution of development artifacts (*cf.* Section 3.1). Furthermore, by connecting the offline and online sides of the software evolution process, we prevent the emergence of technical debt in IAC stemming from run-time variability. This is a significant step toward explicitly connecting other software engineering processes. Thus, we lay a foundation for the exploitation of autonomic managers throughout the delivery pipeline.

C2: Quality-driven Self-Improvement Feedback Loop. To address goal G3, we proposed a feedback loop for exploring design and configuration alternatives during development. The main purpose of this feedback loop is to improve KPIs continuously while the system is running on the development environment. We connected our feedback loop with our continuous software evolution pipeline through two modeling layers, namely: the virtual computing infrastructure and the software architecture. This means that our feedback loop focuses on exploring configuration variants and finding possible improvements. Corresponding source specification updates are delegated to our software evolution pipeline. The feedback loop stops and triggers the source specification update when it identifies a variant that outperforms the baseline configuration (*i.e.*, the system as it runs on production).

Our feedback loop relies on existing quality assessment and testing procedures present in the delivery pipeline. By extending quality assessment with continuous improvement, our feedback loop frees stakeholders from maintenance work and expedites the implementation of incremental software changes. We followed the separation of concerns principle by splitting the online experiment management into three internal feedback loops. The first one addresses the satisfaction of high-level goals through online experiment design. The second one focuses on the derivation, deployment and monitoring of infrastructure configuration variants. Finally, the third one focuses on the derivation, deployment and monitoring of architectural design variants. Both the second and third feedback loops apply combinatorial techniques to derive variants. In the first case, the feedback loop varies configuration parameters of declared virtual resources. In the second case, the feedback loop explores the application of domain-specific design patterns throughout the software architecture.

To evaluate this contribution, we conducted an application scenario focused on experimental validation. The scenario addresses our prototype implementation of the feedback loop for exploring software architecture variants. We provide a simple software architecture to the feedback loop composed of an API component and an GRPC worker. The feedback loop generates hundreds of variants by combining these components with domain-specific design patterns for distributed processing and cloud computing. However, because some of the design patterns we implemented contain application-specific components, this step is semi-automated. Therefore, the experiment ended up containing six architecture variants. Each of them was tested under three execution scenarios, namely constant traffic, linear traffic growth, and traffic spike. These scenarios were used to explore the search space, collecting valuable data to compare the variants in terms of service latency. The results of our proof of concept evaluation provided evidence supporting our long-term vision. On the

one hand, variations of a software architecture can be better equipped to cope with distinct execution scenarios. On the other hand, our self-improvement feedback loop found an appropriate architecture variant that improves the baseline architecture. This means that it can indeed contribute to the long term evolution of the subject software system. Since the feedback loop is a part of the delivery pipeline, improvements are discovered and applied continuously. Thereby, our feedback loop contributes to eliminate a remaining discontinuity from the software development process, namely continuous adjustment of design, configuration and deployment. Moreover, our solution strategy allows exploiting the outcome of the search space exploration in other environments, where it can complement knowledge managed by other autonomic managers. This, in combination with our software evolution pipeline, constitute cooperative self-evolution, a process analogous to how stakeholders contribute to develop software.

C3: Run-Time Evolution Reference Architecture. To address goal G4, we proposed a reference architecture for keeping adaptation decisions relevant in the face of emerging behavior. Our architecture considers temporary as well as long-lasting control actions as a way to countermeasure the evolution of the operation environment over time. In light of this, we chose as architectural drivers the dependability of autonomic decisions, and the resiliency of the managed system's operation. In the first case, we proposed software components based on MIAC, thus making model estimation the cornerstone of our long-term evolution approach. Moreover, we envisioned multiple strategies to contribute to dependability, namely: error mitigation through multi-model identification, reliable models through model inference, evidence collection through experimentation, and autonomic behavior through adaptive control. In the second case, we proposed software components based on MRAC, thus relying on self-adaptation to correct minor and temporary deviations from expected system behavior. The strategies we envisioned to contribute to operation resiliency are: predictable adaptation through reliable models, run-time validation through evidence collection, error mitigation through parameter estimation, goal achievement through hyperparameter optimization, and assurance at run-time through viability zones and control objectives.

The proposed architecture realizes a continuous engineering cycle where adaptation and evolution work cooperatively to achieve the system goals. The main purpose of our engineering cycle is to regulate the reference models used for controlling the system's operation, thus, ultimately contributing to the managed system's long-term evolution. We achieved the envisioned engineering cycle as follows. First, our architecture uses evolutionary optimization and online experimentation to find suitable control reference models. Second, it uses online experimentation, supported by parameter optimization, to gather evidence of statistically significant improvements over the running system, thus generating knowledge of possible configuration states and their respective performances. Lastly, our reference architecture accommodates self-evolution for reflecting persistent changes in the operation environment as well as improving the use of resources. The proposed architecture uses self-adaptation to ensure that the managed system operates within acceptable and viable boundaries. Together, these characteristics contribute to achieving the architectural drivers.

To evaluate this contribution, we used the proposed models and followed the architecture's

guidelines for modeling and implementing prototypes of the main components for our SUTS case study, namely MIM and AM. We conducted an application scenario focused on functional validation. It evaluated our prototype's ability to identify a predictive function for optimizing the excess waiting time, a metric of user experience. Our results show that the MIM was able to approximate a function to describe the user experience metric using two estimators, based on cubic interpolation and symbolic regression. Therefore, this component is capable of synthesizing active and passive knowledge in the form of data and functions. This is a significant step toward synthesizing and evaluating possible adaptation scenarios before manifesting them in the actual system. Moreover, we showed how our proof of concept transitions from evolution to adaptation using our second prototype, the AM. With this, we show the consistency and feasibility of our reference architecture for guiding software architects on how to apply it for engineering SCPS.

8.1.3. Contributions Significance

In recent years, the software industry has been focused on shortening the feedback cycle from post-deployment activities back to development. Great advances in configuration and deployment technology have increased the pace at which software development teams deliver value added services. The success of these efforts has caused the emergence of new challenges that stress the abilities of highly skilled IT personnel to develop and fix mission-critical functionality, maintain existing software and infrastructure implementations, detect and remove technical debt, migrate existing software to modern technologies, refactor design choices, among many others. Various factors call for a radical change to how software is developed and maintained, including increasingly complex software development and operation environments, the anticipated massive increase of interconnected devices [**?**, **?**], and the dependability and resiliency expectations that come with hyper-connectivity.

Future software engineering practices must break out from forward-only life cycle models (*i.e.*, development $\rightarrow$ execution), legacy of waterfall development. Instead, a bidirectional model is necessary, where changes are planned, designed, developed and deployed in both directions (*i.e.*, offline $\rightleftarrows$ online) continuously and relying on manual and automated practices on both sides. Such a process will unlock software's potential to an extent unattainable for the prevalent focus of the software industry—a need for speed through shorter software release cycles [**?**]. In this context, new methodologies, infrastructures and tools are required to realize software evolution online, and fully connect it to its offline counterpart. In this sense, the contributions we present in this dissertation lay down the foundation to establish new foci for software engineering and autonomic computing. We have contributed concrete approaches to realize the required platform to integrate the automated and manual sides of our envisioned software evolution process. We believe that our approaches bring short-term benefits to software development teams. However, we anticipate that our contribution to the state of the art will stimulate further progress in the long term. New products ought to be developed to strengthen information access from every computing environment going from development to production, but especially from production back to planning, design, and development. Such an access to information will

boost a new era of automated tools, thus pushing toward the standardization of online processes,[1] and the emergence of libraries, frameworks and services for self-evolution with increased autonomy.

8.2. Future Work

This dissertation concludes with a presentation of selected future work opportunities emerging from our research.

8.2.1. Long-Term Software Evolution

Our run-time evolution reference architecture concentrates on improving CPSs dependability and resiliency through self-evolution and self-adaptation. They are based on the managing system's capability to represent entities in the managed system's environment accurately. Furthermore, these representations must be kept always up to date and in sync with the entities they model. Since our reference architecture focuses on MARTs, various design concerns remain open research challenges. As elaborated in Section 6.1.2, the regulation relationship we propose differs from the control relationship between managed and managing systems. Since the autonomic systems we propose are not regular autonomic managers or controllers, the interface between them and CPSs is of a different nature. The value delivered goes beyond execution control (*i.e.*, sensing and effecting change). A first open challenge is to define concrete design elements that provide the autonomic system with appropriate interfaces to interact with its CPS in the opportune moments. These elements comprise, for example, the life cycle of persistent evolution actions that may affect physical entities or the policies that govern them (*e.g.*, the OSP). Another design challenge concerns identifying the structures to represent past, present and future states and their interrelationships. A concrete representation of these states would enable the autonomic system to reason about the evolution and adaptation spaces with respect to current viability zones. This type of meta-analysis allows finding unexplored alternatives in a proactive way, thus potentially speeding up the evolution process when it is actually needed. Finally, a third challenge refers to exploring the proactive role of long-term evolution. In the context of our SUTS case study, an autonomic system can investigate the best way of distributing the various types of vehicles based on current passenger demand per line and stop. It can study the placement of new stops, or even propose the need for studying them. This proactive role is contrary to the reactive approach we explored in Chapter 6 (*i.e.*, as a countermeasure to emerging behavior), and is worth exploring independently.

A second group of challenges concerns the control dynamics between run-time self-evolution and self-adaptation. Our architecture establishes a control hierarchy in which the evolution process affects the possible adaptations, and these in turn affect the potential

[1] Possibly following and refining ISO/IEC/IEEE 12207—*Systems and software engineering – Software life cycle processes*

operational states. That is, the evolution space dictates which configurations in the adaptation space are possible, eventually reducing the operational space to a set of concrete instances. A first challenge refers to taking advantage of new adaptation opportunities opened up by a newly explored evolution scenario. For example, imagine that an autonomic manager transforms a software system's architecture from a simple client-server structure to a load-balanced service. The adaptation mechanism can take advantage of the hardware properties as before, but also of the number of service instances and the load balancing algorithm. The adaptation space not only grew in possible adaptations, but is now considering new parameters, thus affecting previous metric calculations. A second challenge consists in predicting the effect that the new parameters have over previous adaptation scenarios, at least while new experiments are conducted. Lastly, as new sensors are deployed and new metrics appear, a third open challenge concerns the reasoning capabilities of the evolution process to increase the accuracy of existing reference models when new data become available. Doing so would decrease the likelihood of unintended effects after system adaptations.

Finally, future advances in dependable autonomy for CPSs to improve system resiliency should address safety and security concerns. Although these properties were out of the scope of this dissertation, they are fundamental to guarantee the stability of the system and trustworthiness of adaptations and evolution actions at run-time.

8.2.2. Knowledge-Preserving Experimentation

The combinatorial explosion of configuration options may produce large search spaces. Since evaluating configuration alternatives requires deploying software to a controlled environment, conducting experiments will likely require much time and computing resources. Moreover, since experiments are being conducted continuously, they may be too expensive, which would increase the total cost of ownership. This situation would defeat the purpose of anticipating adverse conditions or proactively exploring improvements during development. Therefore, self-improvement mechanisms should minimize experimentation time by using resources efficiently. More importantly, they should take advantage of past results to avoid running unnecessary trials. However, even considering these measures, the research challenge may remain open. In addition to reusing past results from whole trials, feedback loops should preserve partial knowledge across trials. That is, granular results from past executions (*e.g.*, knowledge at the gene level when working with chromosomes) to exclude, or include, trials based on a predicted performance. For example, if a gene is known for causing long latency times, chromosomes including the gene can be excluded to prevent under-performing trials.

Failing to address the aforementioned challenge may prevent the adoption of this technology, especially in the context of the delivery pipeline. Imagine, for example, that successive releases to the development environment cause an exhaustive search exploration every time. In contrast, an experimentation mechanism that optimizes its use of resources

would only run a few trials to study KPIs of newly added parts. In this sense, experimentation can be paired with Artifical Intelligence (AI) models, thus realizing an incremental learning strategy.

8.2.3. Software Engineering at Run-Time

In 2001, autonomic computing was deemed as the only alternative to a looming software complexity crisis. It was born to reduce human intervention in the everyday management of computing systems. It was then, and has been, a complement to the work produced by software development teams. In contrast, two decades later, the complexity problem has extended to the left, challenging now the software industry to keep up with the pace of change on the development side.

We draw a parallel between the early days of software engineering and what is emerging now. The so-called software crisis of the 1960s, 1970s and 1980s spurred the early beginning of software engineering. Cost and budget overruns and critical quality issues motivated the later standardization of the discipline into software development processes, methodologies and life cycle models. We believe that the current growth rate of software systems, the increasing complexity of operation environments, and the shortage of technical personnel are significant factors pushing for a similar shift. The advent of AI technology in software development has already started attracting early adopters, especially in open source projects where licensing issues are less restrictive.[2] It's only a matter of time until AI-based tools penetrate the software industry for performing development tasks that are currently being conducted by human stakeholders. We foresee a new era of standardization for software engineering. Our contributions in this dissertation point in this direction. However, in this scenario, autonomic computing represents the early beginning of such a new era. One where processes and life cycle models are digital and programmable. We call this new era *software engineering at run-time*.

[2] Since AI models are being trained on public data sets and code licensed under open source licenses, recommendations and contributions made by AI tools cannot be applied on proprietary source code.

Acronyms

ACRA	Autonomic Computing Reference Architecture
AI	Artifical Intelligence
AM	Adjustment Mechanism
API	Application Programming Interface
BRT	Bus Rapid Transit
CaC	Configuration-as-Code
CAM	Cloud Automation Manager
CD	Continuous Delivery
CD-ROM	Compact Disc Read-Only Memory
C-FL	Configuration Feedback Loop
CI	Continuous Integration
CLI	Command Line Interface
CMES	Cloud Media Encoding Service
CPS	Cyber-Physical Systems
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DSL	Domain-Specific Language
DT	Digital Twin
E-FL	Experimentation Feedback Loop
EWTaBS	Excess Waiting Time at Bus Stops
gRPC	Google Remote Procedure Call
HCL	HashiCorp Configuration Language
HCoV	Headway Coefficient of Variation
HOT	Heat Orchestration Template
HTTP	Hypertext Transfer Protocol

IAC	Infrastructure-as-Code
IDE	Integrated Development Environment
IOT	Internet of Things
IP	Internet Protocol
IT	Information Technology
ITIL	Information Technology Infrastructure Library
JSON	JavaScript Object Notation
KPI	Key Performance Indicator
MAPE-K	Monitor-Analyzer-Planner-Executor–Knowledge
MART	Model at Run-Time
MIAC	Model Identification Adaptive Controller
MIM	Model Identification Mechanism
MIO	Masivo Integrado de Occidente
MMAC	Multi-Model Adaptive Control
MRAC	Model Reference Adaptive Controller
MRAS	Model Reference Adaptive System
OSP	Operational Service Plan
P-FL	Provisioning Feedback Loop
QOS	Quality of Service
RAM	Random Access Memory
RE	Requirements Engineering
REST	Representational State Transfer
RPC	Remote Procedure Call
RT	Real Twin
SCPS	Smart Cyber-Physical Systems
SDLC	Software Development Life Cycle
SLO	Service Level Objective
SLOC	Software Lines of Code
SUTS	Smart Urban Transit System
URL	Uniform Resource Locator

vCPU	Virtual Central Processing Unit
VM	Virtual Machine
XML	Extensible Markup Language

Bibliography

[1] M. A. Nielsen and I. L. Chuang, *Quantum Computation and Quantum Information*, 10th ed. Cambridge, 2011.

[2] R. P. Feynman, "Quantum mechanical computers," *Foundations of Physics*, vol. 16, no. 6, pp. 507–531, 1986. [Online]. Available: https://link.springer.com/article/10.1007/BF01886518

[3] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," 1992.

[4] P. W. Shor, "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Comput.*, vol. 26, no. 5, p. 1484–1509, 1997. [Online]. Available: https://dl.acm.org/doi/10.1137/S0097539795293172

[5] L. K. Grover, "A fast quantum mechanical algorithm for database search," in *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 1996, p. 212–219. [Online]. Available: https://dl.acm.org/doi/10.1145/237814.237866

[6] F. e. a. Arute, "Quantum supremacy using a programmable superconducting processor," *Nature*, vol. 574, no. 7779, pp. 505–510, 2019. [Online]. Available: https://www.nature.com/articles/s41586-019-1666-5

[7] L. S. e. a. Madsen, "Quantum computational advantage with a programmable photonic processor," *Nature*, vol. 606, no. 7912, pp. 75–81, 2022. [Online]. Available: https://www.nature.com/articles/s41586-022-04725-x

[8] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, p. 79, 2018. [Online]. Available: https://quantum-journal.org/papers/q-2018-08-06-79/

[9] A. P. et al., "A variational eigenvalue solver on a photonic quantum processor," *Nature Communications*, vol. 5, no. 1, 2014.

[10] E. Farhi, J. Goldstone, and S. Gutmann, "A quantum approximate optimization algorithm," 2014. [Online]. Available: https://arxiv.org/abs/1411.4028

[11] B. Weder, J. Barzen, F. Leymann, M. Salm, and D. Vietz, "The quantum software lifecycle," p. 2–9, 2020. [Online]. Available: https://doi.org/10.1145/3412451.3428497

[12] A. García de la Barrera, I. García-Rodríguez de Guzmán, M. Polo, and M. Piattini, "Quantum software testing: State of the art," *Journal of Software: Evolution and Process*, vol. 35, no. 4, p. e2419, 2023. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/smr.2419

[13] I.-D. Gheorghe-Pop, N. Tcholtchev, T. Ritter, and M. Hauswirth, "Quantum devops: Towards reliable and applicable nisq quantum computing," pp. 1–6, 2020.

[14] G. Rosenberg, P. Haghnegahdar, P. Goddard, P. Carr, K. Wu, and M. L. de Prado, "Solving the optimal trading trajectory problem using a quantum annealer," in *Proceedings of the 8th Workshop on High Performance Computational Finance*. Association for Computing Machinery, 2015. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2830556.2830563

[15] S. Harwood, C. Gambella, D. Trenev, A. Simonetto, D. Bernal, and D. Greenberg, "Formulating and solving routing problems on a quantum computer," *IEEE Transactions on Quantum Engineering*, vol. 2, pp. 1–17, 2021. [Online]. Available: https://ieeexplore.ieee.org/document/9314905

[16] A. Ajagekar and F. You, "Quantum computing for energy systems optimization: Challenges and opportunities," *Energy*, vol. 179, pp. 76–89, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360544219308254

[17] M. Piattini, G. Peterssen Nodarse, R. Pérez-Castillo, J. L. Hevia Oliver, M. Serrano, G. Hernández González, I. Guzmán, C. Andrés Paradela, M. Polo, E. Murina, L. Jiménez Navajas, J. Marqueño, R. Gallego, J. Tura, F. Phillipson, J. Murillo, A. Niño, and M. Rodríguez, 03 2020. [Online]. Available: https://ceur-ws.org/Vol-2561/paper0.pdf

Part V

Appendices

Appendix A

Implementation Package

A.1. Implementation Projects and Modules

This appendix describes the two projects composing our infrastructure for autonomic and continuous long-term software evolution. The first project encompasses modules related to our infrastructure management case study. Table A.1 describes these modules. This project is publicly hosted on Github,[1] and contains 13,110 Software Lines of Code (SLOC). The second project contains modules related to our SUTS case study. Table A.2 describes these modules. This project is publicly hosted on Github,[2] and contains 8,888 SLOC. Source code information was generated using 'SLOCCount' by David A. Wheeler.[3]

Table A.1.: Source code modules related to our infrastructure management case study

Module	SLOC	Description
notations	4244	Xtext[4] Parser, Match and Diff engine, and Merge strategy for HCL model instances. This module allows instantiating an HCL model from Terraform templates, and comparing HCL models to find structural differences. Furthermore, it contains facilities for merging various models into a single one.
experimentation	3469	Architecture and infrastructure variant generation and realization. This module contains classes for evolutionary exploration of configuration alternatives, and their deployment and monitoring. Moreover, it contains R code for computing fitness scores, and summarizing and reporting experimentation results.
metamodels	1272	Metamodel definitions using Java's XML API (Graph metamodel), and Eclipse's Xcore[5] DSL (HCL and Monitoring metamodels).

[1] https://github.com/RigiResearch/middleware
[2] https://github.com/RigiResearch/rigi-dt-experimentation-cps-code
[3] https://dwheeler.com/sloccount/
[4] A language engineering framework, available at https://www.eclipse.org/Xtext/
[5] A DSL for metamodel definition, available at https://wiki.eclipse.org/Xcore

coordinator	1216	Evolution actions coordination for infrastructure changes. This module exposes a REST endpoint for coordinating evolution actions corresponding to run-time changes. It triggers the model transformation chain to produce code and parameter updates, and then communicates with Git and IBM CAM.
historian.runtime	1205	Runtime library implementing the FORK AND COLLECT algorithm, and associated document transformations introduced in Section 4.2. Moreover, this module contains binding definitions for reusing our Graph metamodel to configure the document transformations.
historian	524	Command-line utility for generating various kinds of artifacts from an OpenAPI specification, including: a base monitoring project that uses the Historian's runtime library; And either a DOT[6] or CXL[7] specification to visualize the cloud endpoints of interest.
vmware.hcl.agent	975	Run-time agent implementation for VMWare's vCenter API. Moreover, this module contains the model transformation code for updating an HCL model from a JSON document collected from vCenter's API.
misc	205	Docker image definition files and configuration code related to Gradle.

Table A.2.: Source code modules related to our SUTS case study

Module	SLOC	Description
simulation	2053	A discrete event simulation framework developed to represent the entities from our SUTS case study, as well as their interrelationships and behavior according to statistical distributions. This project is based on the JSL[8] library. This module also contains implementations for various metrics we used to evaluate configuration variants. Moreover, this module contains binding definitions for reusing our Graph metamodel to configure the simulated system topology.

[6] A graph description language, available at https://graphviz.org/doc/info/lang.html

[7] An XML-based language for describing the content of concept maps, available at https://cmap.ihmc.us/xml/CXL.html

[8] An open source Java Simulation Library (JSL), available at https://github.com/rossetti/JSL

evolution	1869	An evolutionary framework for exploring the configuration space of our SUTS case study, as well as optimizing the corresponding metrics. This module defines a genetic algorithm based on the Jenetics library[9], and components of the fitness function. Furthermore, it uses the `simulation` module to compute the fitness score of chromosomes.
differentiation	1709	Class model and parser implementation for arithmetic function differentiation. The model is completely based on an open source implementation[10], and the parser is based on the exp4j[11] library.
metamodels	1322	Metamodel definitions using Java's XML API (Graph metamodel), and Eclipse's Xcore DSL (Simulation and Scenario metamodels).
pre-processing	1088	Data cleaning, filtering, formatting and aggregation. The source code is written in Java (886) and Python (202).
experimentation	582	Java bridge for R-based implementations of statistical tests.
misc	220	Configuration code related to Gradle.

A.2. Implementation Demos

The following are video demos of some components of our infrastructure. In particular, these videos highlight the IAC evolution pipeline:

- CASCON 2019 Exhibit: Round-Trip Engineering Scenarios for IAC Templates
- Reverse engineering and continuous update of IAC templates:
  - Scenario including the Evolution Coordinator and the Run-time agent
  - Scenario including the Evolution Coordinator, the Run-time agent and IBM CAM
- Two-way CI running on OpenStack and CircleCI
- Historian: A monitoring framework to support the automated evolution of models at run-time (Historian's first version)

[9] A genetic algorithm, evolutionary algorithm, genetic programming, and multi-objective optimization library, available at `https://jenetics.io/`

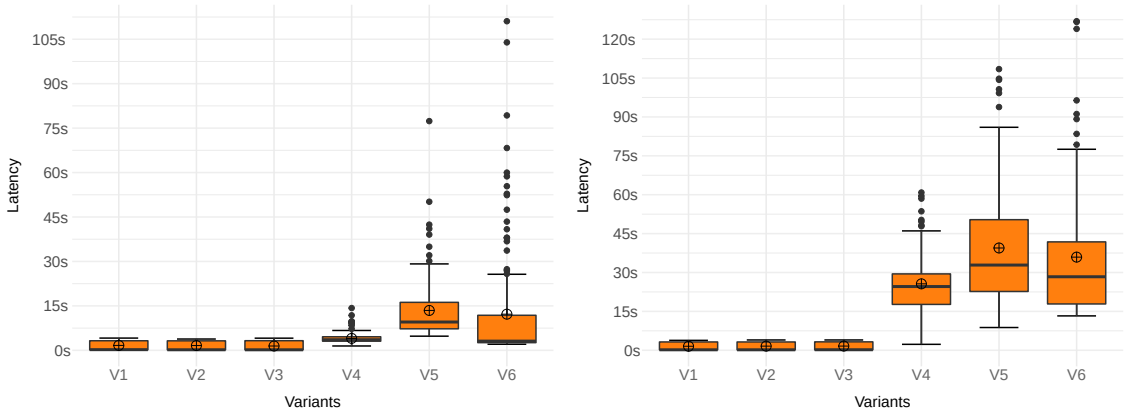
[10] The external code is available at `https://github.com/accelad-com/nilgiri-math`

[11] a library for interpreting arithmetic expressions, available at `https://www.objecthunter.net/exp4j/`

Appendix B

Latency Measurements for the Cloud Media Encoding Service

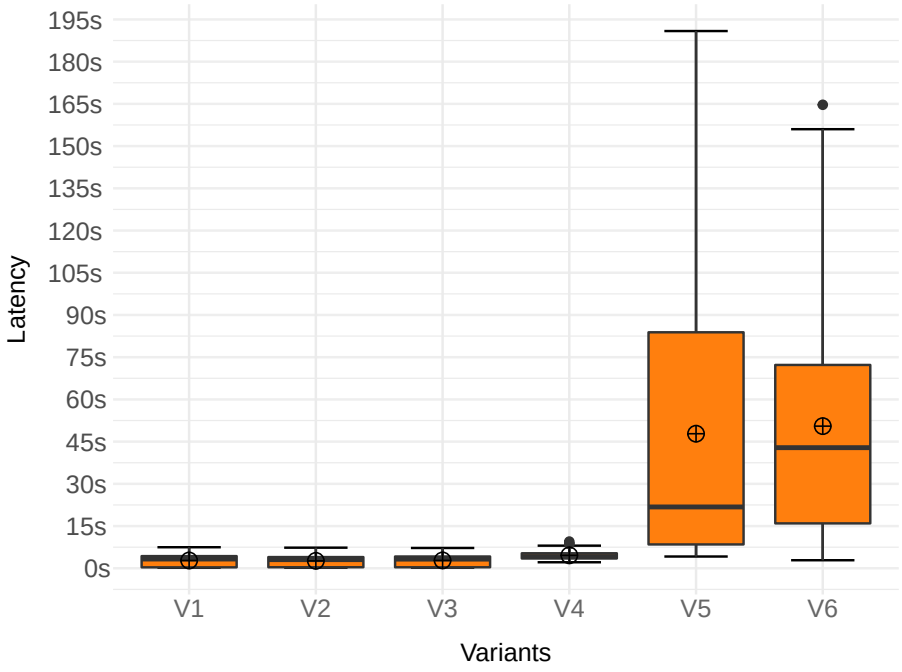
This appendix presents additional charts related to our cloud infrastructure management case study. The measurements presented in these charts were part of the controlled experiments presented in Section 7.1. The latency box plots represent, visually, why the C-FL selected the variants it selected for each execution scenario. These plots are useful for humans in the loop assessing the proposed changes. Figure B.1 displays box plots for video and playlist requests regarding the Regular execution scenario. Similarly, Figs. B.2 and B.3 display box plots for the Linear and Spike execution scenarios, respectively. Figs. B.4 to B.6 display QQ plots based on the same data.



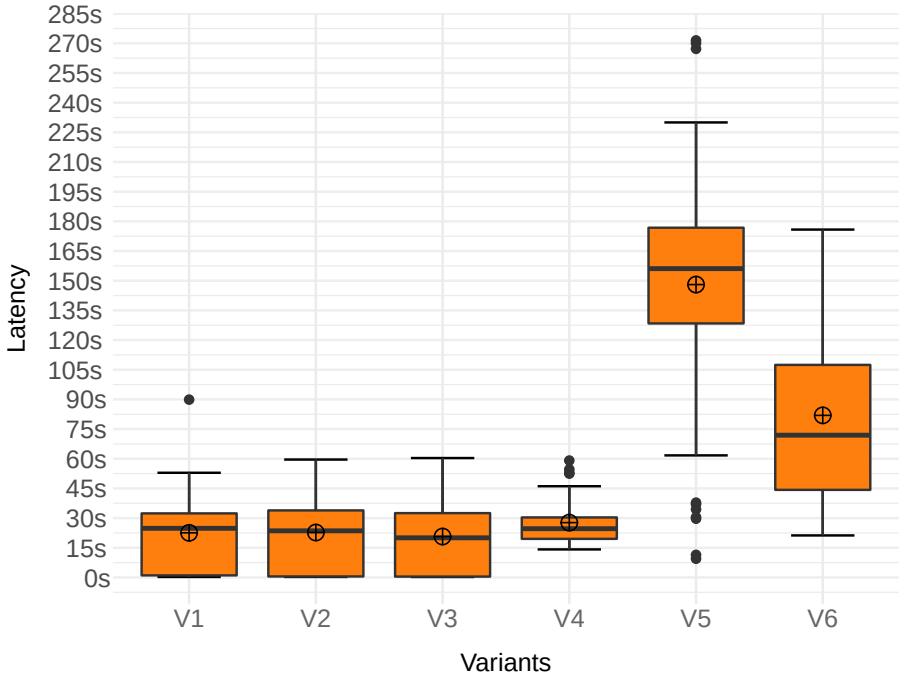
(a) Video requests

(b) Playlist requests

Figure B.1.|: Latency box plots for the Regular scenario

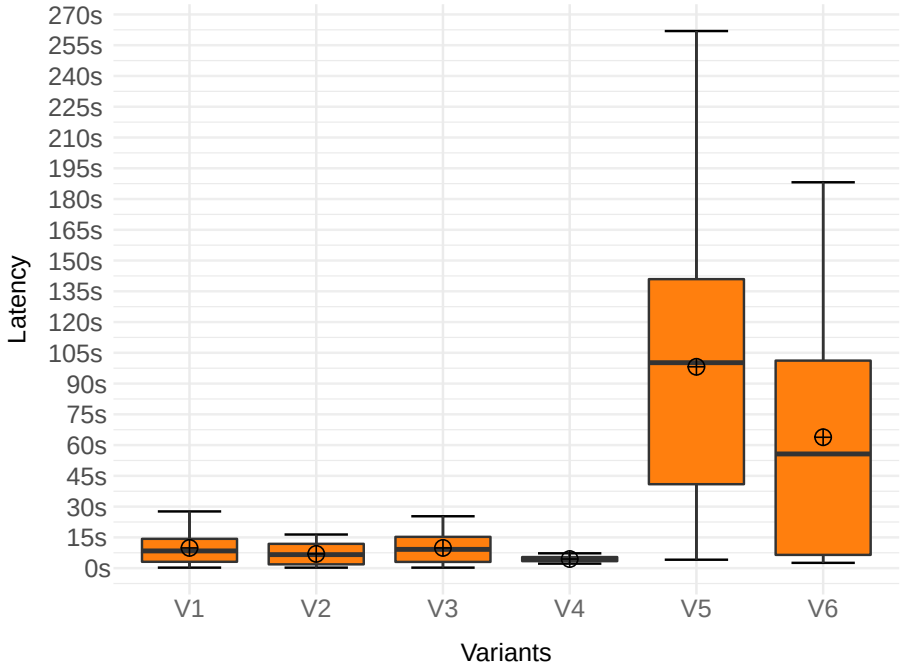


(a) Video requests

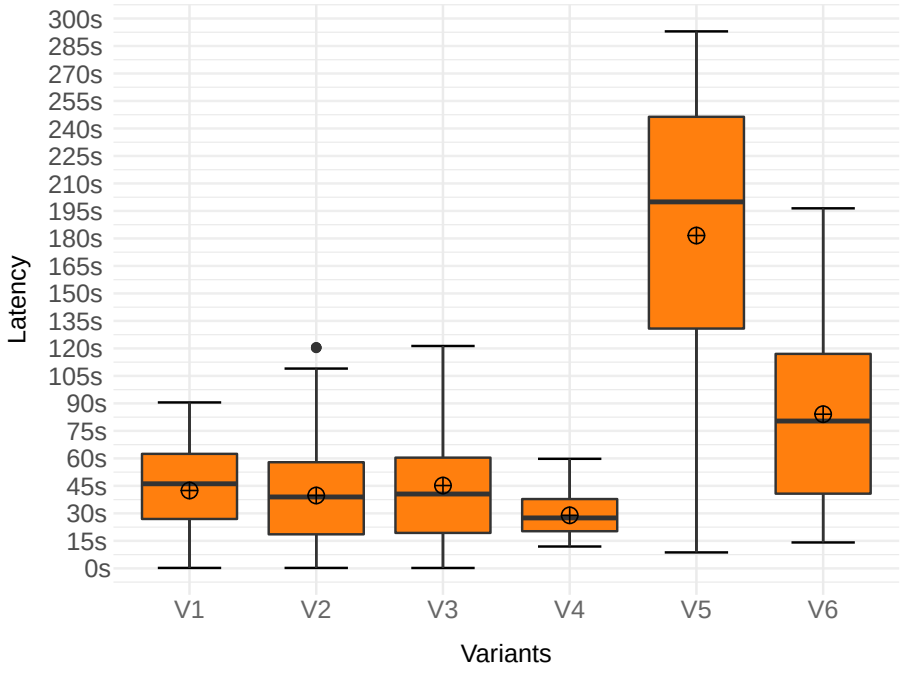


(b) Playlist requests

Figure B.2.|: Latency box plots for the Linear scenario

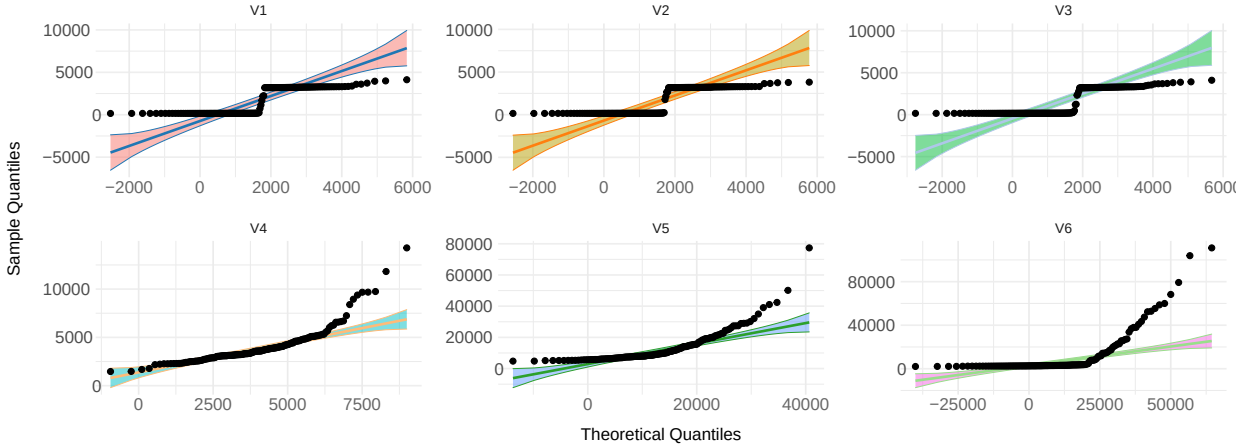


(a) Video requests

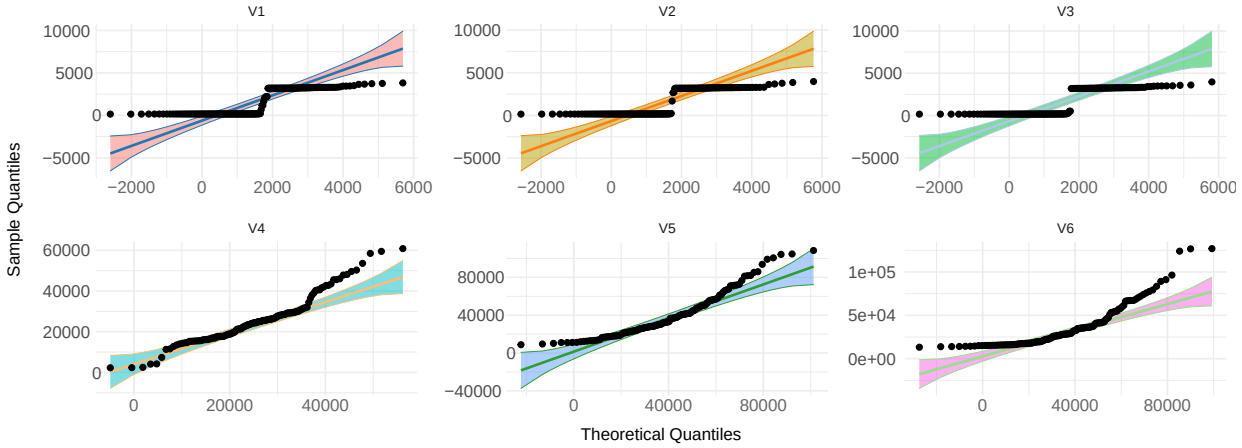


(b) Playlist requests

Figure B.3.|: Latency box plots for the Spike scenario

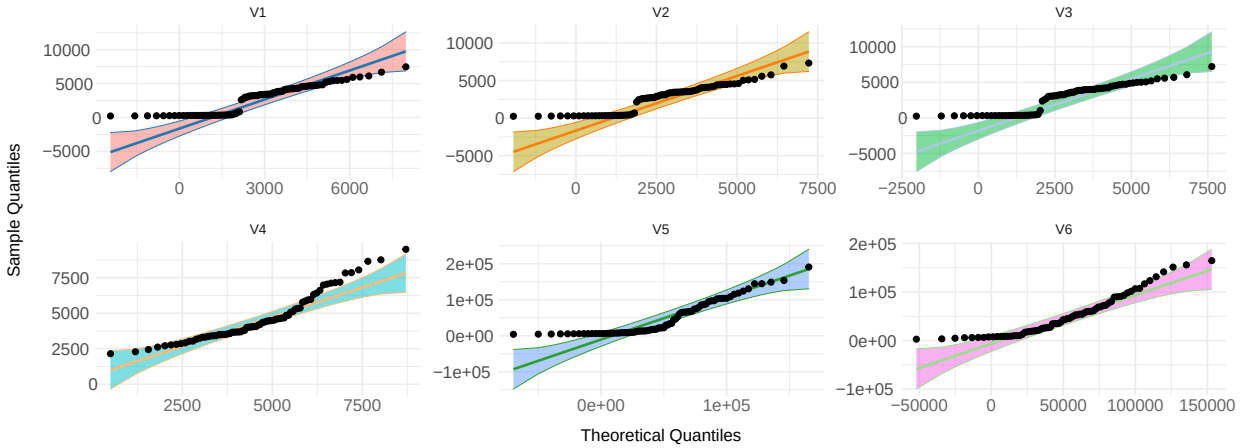


(a) Video requests

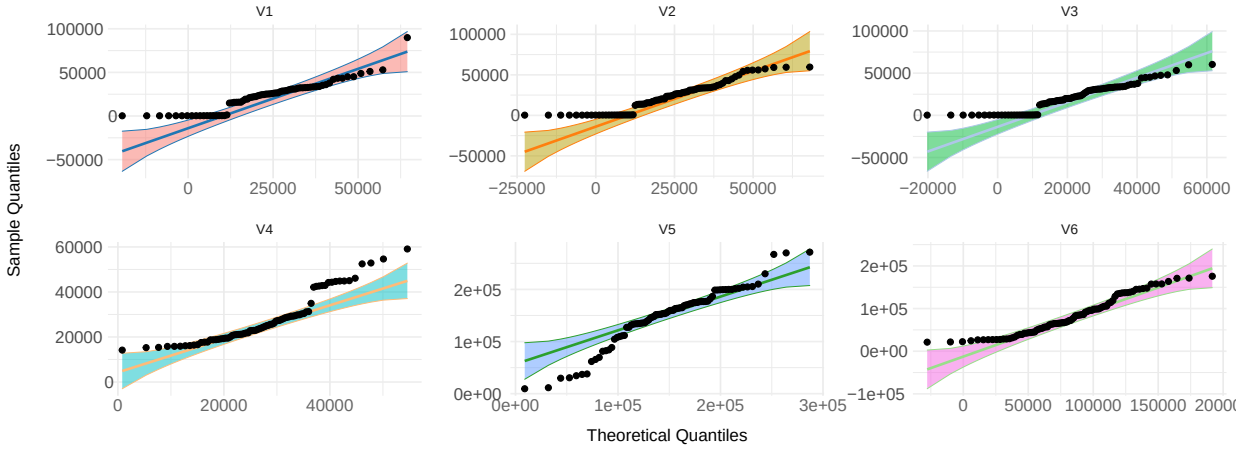


(b) Playlist requests

Figure B.4.|: Latency QQ plots for the Regular scenario

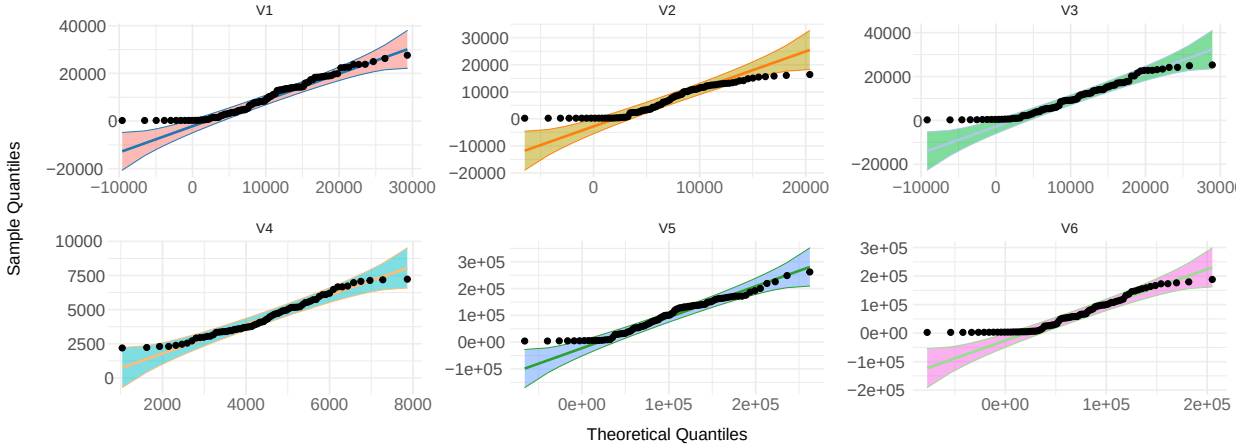


(a) Video requests

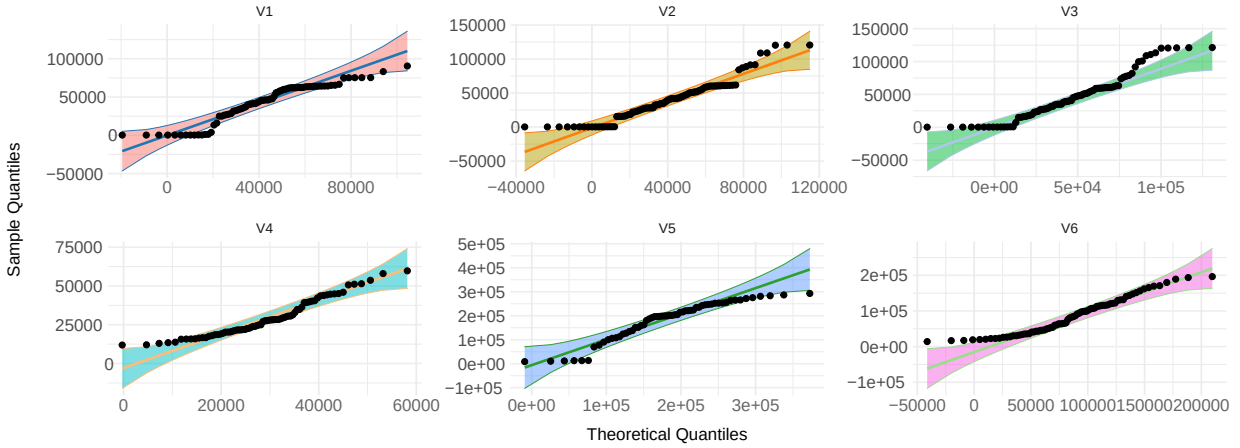


(b) Playlist requests

Figure B.5.|: Latency QQ plots for the Linear scenario



(a) Video requests



(b) Playlist requests

Figure B.6.|: Latency QQ plots for the Spike scenario