Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Applying Control Theory to the Orchestration of Data Stream Processing Systems

Lília Rodrigues Sampaio

Campina Grande, Paraíba, Brasil

09/2022

Universidade Federal de Campina Grande

Centro de Engenharia Elétrica e Informática

Coordenação de Pós-Graduação em Ciência da Computação

Applying Control Theory to the Orchestration of Data Stream Processing Systems

Lília Rodrigues Sampaio

Doctoral Dissertation submitted to the Postgraduate Course Coordination of Computer Science at Universidade Federal de Campina Grande - Campus I as part of the requirements to acquire the Doctor degree in Computer Science.

Main Topic: Computer Science

Research Line: Control Theory and Resource Provisioning

Andrey Elísio Monteiro Brito

(Advisor)

Campina Grande, Paraíba, Brasil

S192a Sampaio, Lília Rodrigues.
Applying control theory to the orchestration of data stream processing systems / Lília Rodrigues Sampaio. - Campina Grande, 2022.
180 f. : il. color.

Tese (Doutorado em Ciência da Computação) - Universidade Federal de Campina Grande, Centro de Engenharia Elétrica e Informática, 2022.
"Orientação: Prof. Dr. Andrey Elísio Monteiro Brito."
Referências.

1. Processamento de Dados. 2. Escalonamento de Recursos. 3. Teoria do Controle. I. Brito, Andrey Elísio Monteiro. II. Título.

CDU 004.451.7.031.43(043)

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECÁRIA ITAPUANA SOARES DIAS GONÇALVES CRB-15/93

MINISTÉRIO DA EDUCAÇÃO
UNIVERSIDADE FEDERAL DE CAMPINA GRANDE
POS-GRADUACAO CIENCIAS DA COMPUTACAO
Rua Aprigio Veloso, 882, - Bairro Universitario, Campina Grande/PB, CEP 58429-900

FOLHA DE ASSINATURA PARA TESES E DISSERTAÇÕES

LÍLIA RODRIGUES SAMPAIO

APPLYING CONTROL THEORY TO THE ORCHESTRATION OF DATA STREAM PROCESSING SYSTEMS

Tese apresentada ao Programa de Pós-Graduação em Ciência da Computação como pré-requisito para obtenção do título de Doutor em Ciência da Computação.

Aprovada em: 22/09/2022

Prof. Dr. ANDREY ELÍSIO MONTEIRO BRITO, Orientador, UFCG

Prof. Dr. FÁBIO JORGE ALMEIDA MORAIS, Examinador Interno, UFCG

Prof. Dra. RAQUEL VIGOLVINO LOPES, Examinadora Interna, UFPB

Prof. Dra. PRISCILA AMÉRICA SOLÍS MENDEZ BARRETO, Examinadora Externa, UnB

Dr. ANDRÉ MARTIN, Examinador Externo, TU-Dresden

sei! assinatura eletrônica

Documento assinado eletronicamente por **ANDREY ELISIO MONTEIRO BRITO**, **PROFESSOR 3 GRAU**, em 22/09/2022, às 15:41, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **FABIO JORGE ALMEIDA MORAIS**, **PROFESSOR(A) DO MAGISTERIO SUPERIOR**, em 22/09/2022, às 19:20, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **RAQUEL VIGOLVINO LOPES**, **Usuário Externo**, em 23/09/2022, às 09:13, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

Documento assinado eletronicamente por **André Martin**, **Usuário Externo**, em 27/09/2022, às 08:01, conforme horário oficial de Brasília, com fundamento no art. 8º, caput, da Portaria SEI nº 002, de 25 de outubro de 2018.

A autenticidade deste documento pode ser conferida no site https://sei.ufcg.edu.br/autenticidade, informando o código verificador **2743817** e o código CRC **0D5355F3**.

Referência: Processo nº 23096.064474/2022-87 SEI nº 2743817

Resumo

A capacidade de processar eficientemente grandes quantidades de dados, como os advindos de sensores IoT, é um objetivo desejado por variados sistemas, especialmente porque o valor desses dados pode cair rapidamente após o momento de sua coleta. Demandas de processamento desse tipo levaram ao desenvolvimento do paradigma *Data Stream Processing*, onde dados chegam continuamente e precisam ser processados em tempo real. Tais aplicações estão sujeitas a variadas condições de operação, sendo importante se adaptar bem a diferentes cenários enquanto mantém metas de Qualidade de Serviço. Abordagens tradicionais sugerem soluções voltadas ao escalonamento automático dos recursos, que apresentam desafios como definir boas métricas de interesse para os objetivos de QoS, determinar o intervalo de coleta desses dados e estimar a quantidade de recursos que devem ser provisionados.

Apesar de novas técnicas para o monitoramento e adaptação de sistemas de processamento de dados em fluxo estarem continuamente evoluindo, muitas das soluções propostas não possuem a base teórica necessária para garantir níveis altos de acurácia em suas execuções. Dada sua abordagem analítica, a teoria do controle pode ser uma boa alternativa para este fim. Entretanto, aplicar técnicas de controle em sistemas de computação ainda se apresenta como um desafio, principalmente pela dificuldade em abstrair o comportamento complexo de *software* em uma forma matemática adequada para o design de um controlador, de forma a diminuir o atraso do sistema, gerar ações corretivas adequadas e minimizar o erro de estado estável.

Considerando isso, este trabalho propõe aplicar e avaliar metodologias da teoria do controle em sistemas de processamento de micro-lotes de dados em fluxo. Métodos de identificação de sistemas são utilizados para modelagem do Asperathos, um *framework* para automação de diferentes aplicações de processamento de dados mantendo metas de QoS customizáveis. Com base nisso, é proposto um controlador Proporcional-Integral que rastreia métricas de desempenho, além de uma demonstração de sintonização de seus ganhos. Ainda é proposto um controlador de múltiplos objetivos do tipo SIMO, baseado em métricas de desempenho e custo. Para validação da solução, tarefas de desagregação de dados de energia são executadas em um *cluster* Kubernetes orquestrado pelo Asperathos.

Abstract

The ability to efficiently process large amounts of data, such as that from IoT sensors, is a desired goal for many systems, especially since the value of this data can quickly drop after the moment it is collected. Processing demands of this kind led to the development of the Data Stream Processing (DSP) paradigm, where data arrives continuously and needs to be processed in real time. Such applications are subject to varying operating conditions, and it is important to adapt well to different scenarios while maintaining Quality of Service (QoS) goals. Traditional approaches suggest solutions aimed at the automatic scaling of resources, which presents challenges such as defining good metrics of interest for QoS objectives, determining the interval for collecting this data and estimating the amount of resources that must be provisioned.

Although new techniques for monitoring and adapting DSP systems are continuously evolving, many of the proposed solutions do not have the necessary theoretical basis to guarantee high levels of accuracy in their execution. On the other hand, given its analytical approach, Control Theory can be a good alternative for this purpose. However, applying control techniques in computer systems still presents itself as a challenge, mainly due to the difficulty in abstracting the complex behavior of software in a mathematical form suitable for the design of a controller, in order to reduce the system delay, generate appropriate corrective actions, and minimize steady-state error.

Considering this, this work proposes to apply and evaluate control theory methodologies in micro-batch DSP systems. System identification methods are used to generate a model representation of Asperathos, a framework for automating different data processing applications while maintaining customizable QoS goals. Based on this, a Proportional-Integral controller that tracks performance metrics is proposed, as well as a demonstration of its tuning. A SIMO-type multi-objective controller is also proposed, based on performance and cost metrics. For the validation of the solution, energy data disaggregation tasks are performed in a Kubernetes cluster orchestrated by Asperathos.

Acknowledgements

À Deus, por me mostrar os caminhos que sempre me fizeram perseverar.

Ao meu pai (*in memoriam*) e à minha mãe, palavras não conseguem expressar. Sei que se estivesse aqui, a rua inteira estaria sabendo que todos os seus filhos são doutores, pai. Mas sei também que daí você me guiou e celebrou comigo essa conquista. Mãe, obrigada por desde pequena me ensinar o poder e o valor da educação. Ter uma mãe professora, que sempre lutou para que amássemos a educação como a senhora ama, fez toda a diferença em quem eu sou hoje. Sem meus pais eu não seria nada, e por eles sempre fiz e farei tudo.

Aos meus irmãos, por me ensinarem no companheirismo de sempre o poder da determinação e da dedicação. Por mais sonhos nossos realizados, juntos, sempre. Agora posso entrar para o clubinho dos doutores :)

À toda minha família, que me enche de amor todos os dias.

Ao meu namorado e meus amigos, por serem comigo, companheiros e verdadeiros. Grata por ter vocês sempre por perto, ao meu lado.

Ao meu orientador, Andrey, por sempre acreditar no meu melhor e me inspirar todos os dias a ser segura, motivada e ambiciosa. Obrigada por todos os trinta segundinhos de conversas cheias de sabedoria, por me compreender e ajudar quando precisei.

À Maxwell, membro do meu grupo de pesquisa, por contribuir diretamente com este trabalho na geração dos modelos aqui propostos, e nas muitas (longas!) discussões que partilhamos durante esse tempo. Sem dúvida sou grata por toda experiência e aprendizado trocado entre nós. Também à Diego, Ignacio e Armstrong, pela participação no artigo que é fruto desse trabalho, contribuindo junto ao Asperathos e na execução de experimentos. Em especial durante a pandemia, vocês me ajudaram a manter o foco e me aprofundar cada vez mais na temática dessa tese.

Aos meus amigos nos vários projetos em que pude participar ao longo do doutorado, em especial Fabinho, Rodolfo, Marcus, Vinha, Clenimar, Fellype e Igor, por me enriquecerem todos os dias com conhecimento e discussões proveitosas que fizeram de mim uma profissional melhor. Além disso, pelo ombro amigo para chorar os medos e angústias, e comemorar as (muitas!) alegrias e sucessos.

Ao Laboratório de Sistemas Distribuídos, por todo amparo profissional e tecnológico de qualidade, mas em especial por sempre me fazer sentir em casa. Quando graduanda, ainda no terceiro período, ser LSD foi sem dúvidas a melhor escolha que eu poderia ter feito.

Por fim, agradeço à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), e aos projetos ATMOSPHERE e LiteCampus pelo financiamento desta pesquisa.

Contents

1 Introduction **1**

1.1 Context and motivation 1

1.2 Objectives 5

1.2.1 Methodology 6

1.3 Contributions 7

1.4 Summary 8

2 Background **9**

2.1 Quality of Service 9

2.2 Container orchestration 10

2.3 Asperathos 12

2.3.1 Architecture 12

2.3.2 Custom plugins 14

2.4 Use case: Energy data processing in real time 15

2.4.1 The workload 17

3 Related work **20**

3.1 Quality of service 20

3.1.1 Web applications 20

3.1.2 Data stream processing 22

3.2 Provisioning and scaling of resources 25

3.2.1 Overview 25

3.2.2 Control theory 27

3.2.3 Other approaches 32

4 Control systems **34**

4.1 Feedback control 34

4.1.1 Choosing control variables 37

4.2 Controller types 38

4.2.1 Proportional control 39

4.2.2 Integral control 40

4.2.3 Derivative control 40

4.2.4 PID control: proportional, integral, derivative 41

4.3 Considerations when implementing a controller 42

4.4 Challenges for computer systems 44

5 Applying First-Order Plus Dead Time models to DSP systems **46**

5.1 System identification methods 46

5.2 Application use case: Asperathos 50

5.3 Using filters 55

5.3.1 Inserting a low-pass filter on Asperathos 57

6 Adaptive control of DSP systems **59**

6.1 Context and motivation 59

6.2 Selecting control variables 61

6.3 Control approaches 64

6.3.1 Fixed Action control 64

6.3.2 Proportional-Integral control 65

6.4 Evaluation 69

6.4.1 Experimental design 69

6.4.2 Scenarios I and II: Analytic PI x Fixed Action 76

6.4.3 Scenario III: Analytic PI x Manual PI 82

6.4.4 Scenario IV: PI - Runtime x PI - Estimated 86

7 A multiple-objective control approach **91**

7.1 Context and motivation 91

7.2 Definition of a controller for a cost variable 92

7.2.1 Business model 93
7.2.2 Gain scheduling control 93
7.3 Definition of a multiple objective controller 95
7.4 Evaluation 97
7.4.1 Experimental design 98
7.4.2 Scenario V: SIMO PI x Independent control 101

8 Conclusions **109**

A Descriptive analysis of experimentation data **135**
A.1 Scenarios I and II: Proportional-Integral x Fixed Action 135
A.1.1 Tracked error 135
A.1.2 Replica allocation 139
A.1.3 Response time 143
A.1.4 Data variability 143
A.2 Scenario III: Analytic PI x Manual PI 144
A.2.1 Tracked error 144
A.2.2 Replica allocation 147
A.2.3 Response time 150
A.2.4 Data variability 151
A.3 Scenario IV: PI - Runtime x PI - Estimated 151
A.3.1 Tracked error 152
A.3.2 Replica allocation 154
A.3.3 Response time 156
A.3.4 Data variability 157
A.4 Scenario V: SIMO x Independent control 158
A.4.1 Tracked error 158
A.4.2 Replica allocation 161
A.4.3 Response time 163
A.4.4 Data variability 164

B Statistical analysis of experimentation data 165

B.1 Scenarios I and II: Proportional-Integral x Fixed Action 165

B.1.1 Replica allocation 165

B.1.2 Tracking of the reference value 167

B.2 Scenario III: Analytic PI x Manual PI 168

B.2.1 Tracking of the reference value 168

B.2.2 System response time 169

B.3 Scenario IV: PI - Runtime x PI - Estimated 170

B.3.1 Replica allocation 170

B.3.2 System response time 171

B.4 Scenario V: SIMO x Independent control 172

B.4.1 Replica allocation 172

B.4.2 Tracking of the reference value 174

B.4.3 System response time 177

List of Symbols

IoT - *Internet of Things*

DSP - *Data Stream Processing*

QoS - *Quality of Service*

SLA - *Service Level Agreements*

MAPE - *Monitoring, Analysis, Planning and Execution*

PID - *Proportional-Integrative-Derivative*

PI - *Proportional-Integrative*

MIMO - *Multiple-Input, Multiple-Output*

SIMO - *Single-Input, Multiple-Output*

FOPDT - *First-Order Plus Dead Time*

NRMSE - *Normalized Root Mean Square Error*

List of Figures

2.1 Asperathos architecture. 13

4.1 Generic architecture of a feedback control loop. 35

5.1 Model 7 analysis compared with real executions on Asperathos. 53

5.2 Model 8 analysis compared with real executions on Asperathos. 53

5.3 Model 7 highlight: analysis compared with the execution number 7 on Asperathos. 54

5.4 Model 7 analysis compared with filtered executions on Asperathos. 58

5.5 Chosen model highlighted: Model 7 analysis compared with its original and filtered execution on Asperathos. 58

6.1 Tracking of reference value with given PI tuning configurations. 69

6.2 Analytic PI x Fixed Action - 3. 80

6.3 PI Control x Fixed Action - 1. 80

6.4 Analytic PI x Fixed Action: SLA violation ratio. 81

6.5 Analytic PI x Manual PI. 84

6.6 Analytic PI x Manual PI: SLA violation ratio. 85

6.7 PI - Estimated x PI - Runtime - Tracking of reference value. 88

6.8 PI - Estimated x PI - Runtime. 89

6.9 PI - Runtime x PI - Estimated: SLA violation ratio. 90

7.1 Architectural model of the proposed SIMO PI controller. 96

7.2 Tracking of reference value for the SIMO and independent controllers approach. 105

7.3 Tracked error, number of replicas and queue size metrics for the SIMO and independent controllers approach. 105
7.4 Execution time of tasks processed using the SIMO control approach and independent controllers. 106
7.5 Financial impact for the SIMO PI and Independent configuration. 107
7.6 SLA impact for the SIMO PI and Independent configuration. 108

A.1 Analytic PI: Distribution of the tracked error data for 4 random replications. 136
A.2 Fixed Action 3: Distribution of the tracked error data for 4 random replications. 136
A.3 Fixed Action 1: Distribution of the tracked error data for 4 random replications. 137
A.4 Analytic PI X Fixed Action 1 x Fixed Action 3: Histogram distribution of the averaged tracked error data for each treatment. 138
A.5 Analytic PI X Fixed Action 1 x Fixed Action 3: Boxplot distribution of the averaged tracked error data for each treatment. 139
A.6 Analytic PI: Distribution of the replica allocation data for 4 random replications. 140
A.7 Fixed Action 3: Distribution of the replica allocation data for 4 random replications. 140
A.8 Fixed Action 1: Distribution of the replica allocation data for 4 random replications. 141
A.9 Analytic PI X Fixed Action 1 x Fixed Action 3: Histogram distribution of the averaged replica allocation data for each treatment. 142
A.10 Analytic PI X Fixed Action 1 x Fixed Action 3: Boxplot distribution of the averaged replica allocation data for each treatment. 142
A.11 Analytic PI X Fixed Action 1 x Fixed Action 3: Distribution of the system response time data for 4 random replications.. 143
A.12 Analytic PI - Constant: Distribution of the tracked error data for 4 random replications. 145
A.13 Manual PI - Constant: Distribution of the tracked error data for 4 random replications. 145

A.14 Analytic PI - Constant X Manual PI - Constant: Histogram distribution of the averaged tracked error data for each treatment. 146
A.15 Analytic PI - Constant X Manual PI - Constant: Boxplot distribution of the averaged tracked error data for each treatment. 147
A.16 Analytic PI - Constant: Distribution of the replica allocation data for 4 random replications. 148
A.17 Manual PI - Constant: Distribution of the replica allocation data for 4 random replications. 148
A.18 Analytic PI - Constant X Manual PI - Constant: Histogram distribution of the averaged replica allocation data for each treatment. 149
A.19 Analytic PI - Constant X Manual PI - Constant: Boxplot distribution of the averaged replica allocation data for each treatment. 149
A.20 Analytic PI - Constant X Manual PI - Constant: Distribution of the system response time data for 4 random replications.. 150
A.21 Analytic PI - Estimated: Distribution of the tracked error data for 4 random replications. 152
A.22 Analytic PI - Runtime X Analytic PI - Estimated: Histogram distribution of the averaged tracked error data for each treatment. 153
A.23 Analytic PI - Runtime X Analytic PI - Estimated: Boxplot distribution of the averaged tracked error data for each treatment. 153
A.24 Analytic PI - Estimated: Distribution of the replica allocation data for 4 random replications. 154
A.25 Analytic PI - Runtime X Analytic PI - Estimated: Histogram distribution of the averaged replica allocation data for each treatment. 155
A.26 Analytic PI - Runtime X Analytic PI - Estimated: Boxplot distribution of the averaged replica allocation data for each treatment. 156
A.27 Analytic PI - Runtime X Analytic PI - Estimated: Distribution of the system response time data for 4 random replications.. 157
A.28 SIMO PI x Independent: Distribution of the performance tracked error data for 1 random replication. 159

A.29 SIMO PI x Independent: Histogram distribution of the averaged performance tracked error data for each control preference. 160
A.30 SIMO PI x Independent: Boxplot distribution of the averaged performance tracked error data for each control preference. 160
A.31 SIMO PI x Independent: Distribution of the replica allocation data for 1 random replication. 161
A.32 SIMO PI x Independent: Histogram distribution of the averaged replica allocation data for each control preference. 162
A.33 SIMO PI x Independent: Boxplot distribution of the averaged replica allocation data for each control preference. 162
A.34 SIMO PI x Independent: Distribution of average system response time data for 5 random replications. 163

List of Tables

5.1 Model NRMSE. 52

6.1 PI Tuning Configuration 68
6.2 Factors for the experiments with PI and Fixed Action controllers. 74
6.3 Treatments for the experiments with PI and Fixed Action controllers. 75
6.4 Configuration for the PI-Fixed set of experiments. 76
6.5 Rate of items processed on time statistical observations for the PI x Fixed Action - 3 scenario. 77
6.6 SLA violation rate statistical observations for the PI x Fixed Action - 3 scenario. 77
6.7 Rate of items processed on time statistical observations for the PI x Fixed Action - 1 scenario. 78
6.8 SLA violation rate statistical observations for the PI x Fixed Action - 1 scenario. 78
6.9 Configuration for the Analytic-Manual PI set of experiments. 82
6.10 Replica allocation statistical observations for the Analytic PI x Manual PI scenario. 83
6.11 Configuration for the Runtime-Estimated PI set of experiments. 86
6.12 Tracked error statistical observations for the PI - Runtime x PI - Estimated scenario. 87

7.1 Business model for the cost controller. 94
7.2 Factors for the experiments with SIMO and independent controllers. 99
7.3 Treatments for the experiments with SIMO and Independent controllers. . . 101
7.4 Total execution cost statistical observations for the SIMO PI (0.2) x Independent scenario. 102

7.5 Total execution cost statistical observations for the SIMO PI (0.5) x Independent scenario. 102
7.6 Total execution cost statistical observations for the SIMO PI (0.8) x Independent scenario. 103
7.7 Averages of the metrics analyzed for each control preference for the SIMO PI x Independent scenario. 103
7.8 P-values of each control preference for the SIMO PI x Independent scenario. 104

A.1 Data variability considering replica allocation, tracked error and system response time for the PI x Fixed Action scenario. 144
A.2 Data variability considering replica allocation, tracked error and system response time for the Analytic PI x Manual PI scenario. 151
A.3 Data variability considering replica allocation, tracked error and system response time for the PI - Runtime x PI - Estimated scenario. 158
A.4 Data variability considering replica allocation, tracked error, total cost and system response time for the SIMO PI x Independent scenario. 164

B.1 Replica allocation statistical observations for the PI x Fixed Action - 3 scenario. 165
B.2 Replica allocation statistical observations for the PI x Fixed Action - 1 scenario. 166
B.3 Tracked error statistical observations for the PI x Fixed Action - 3 scenario. 167
B.4 Tracked error statistical observations for the PI x Fixed Action - 1 scenario. 168
B.5 Tracked error statistical observations for the Analytic PI x Manual PI scenario. 169
B.6 Rate of items processed on time statistical observations for the Analytic PI x Manual PI scenario. 169
B.7 SLA violation rate statistical observations for the Analytic PI x Manual PI scenario. 170
B.8 Replica allocation statistical observations for the PI - Runtime x PI - Estimated scenario. 171
B.9 Rate of items processed on time statistical observations for the PI - Runtime x PI - Estimated scenario. 171
B.10 SLA violation rate statistical observations for the PI - Runtime x PI - Estimated scenario. 172

B.11 Replica allocation statistical observations for the SIMO PI (0.2) x Independent scenario. 173
B.12 Replica allocation statistical observations for the SIMO PI (0.5) x Independent scenario. 173
B.13 Replica allocation statistical observations for the SIMO PI (0.8) x Independent scenario. 173
B.14 Performance controller: tracked error statistical observations for the SIMO PI (0.2) x Independent scenario. 174
B.15 Performance controller: tracked error statistical observations for the SIMO PI (0.5) x Independent scenario. 175
B.16 Performance controller: tracked error statistical observations for the SIMO PI (0.8) x Independent scenario. 175
B.17 Cost controller: tracked error statistical observations for the SIMO PI (0.2) x Independent scenario. 176
B.18 Cost controller: tracked error statistical observations for the SIMO PI (0.5) x Independent scenario. 176
B.19 Cost controller: tracked error statistical observations for the SIMO PI (0.8) x Independent scenario. 177
B.20 Rate of items processed on time statistical observations for the SIMO PI (0.2) x Independent scenario. 178
B.21 SLA violation rate statistical observations for the SIMO PI (0.2) x Independent scenario. 178
B.22 Rate of items processed on time statistical observations for the SIMO PI (0.5) x Independent scenario. 179
B.23 SLA violation rate statistical observations for the SIMO PI (0.5) x Independent scenario. 179
B.24 Rate of items processed on time statistical observations for the SIMO PI (0.8) x Independent scenario. 180
B.25 SLA violation rate statistical observations for the SIMO PI (0.8) x Independent scenario. 180

Chapter 1

Introduction

1.1 Context and motivation

The ability to efficiently process large amounts of data, such as financial transactions, requests to a web server or data coming from Internet of Things (IoT) sensors, is a persistent and highly desired goal for many distributed systems. One of the reasons for this is that, in some cases, the value associated with the data quickly drops after it was collected. For example, a log analysis software can automatically detect security attacks to a system, helping to contain or even prevent possible damage, but for that, such data needs to be analyzed as quickly as possible or any reaction could be delayed. Another example are social networks that want to quickly find what are the trending topics at the moment, and then deliver this content to their users before the information becomes obsolete. The processing demands of these workloads led to the development of the Data Stream Processing (DSP) paradigm, where data arrives continuously and needs to be processed in real time, according to a given arrival rate of new requests [171; 136; 127; 162].

Because data arrives in a continuous stream, DSP applications are commonly of long duration and subject to a variety of conditions in their execution environment, often in an unpredictable way. In this case, it is important to have the ability to adapt well to these scenarios. Furthermore, it is essential that during this process, Quality of Service (QoS) goals are maintained in accordance with the user's definitions. The solutions documented in the literature present a broad view of mechanisms, architectures and methodologies aimed at adapting DSP systems [10; 19; 85; 149; 143].

In this context, at the infrastructure level, adaptation consists mostly of the process known as resource automatic scaling [143; 161; 142; 115]. The rise of cloud elasticity solutions has increasingly attracted web application providers to move their workloads to the cloud [96; 165; 11]. By definition, the resource auto scaling problem for this type of application consists of provisioning or removing a set of resources autonomously and dynamically, lowering execution costs, and satisfying Service Level Agreements (SLA). If we consider an scenario where there is an increase in requests continually arriving at a web server, the available resources will become congested, forcing the automatic scheduler to add more resources to the infrastructure. The opposite also applies when the number of requests drops drastically, in which case the scheduler removes resources according to this new demand. This is a classic problem in the context of automatic scaling and system adaptation, commonly abstracted as a MAPE (Monitoring, Analysis, Planning and Execution) control loop [97].

Each of these phases presents its challenges, but for the understanding of this work, we highlight Monitoring and Planning. To determine whether any action should be taken on the system, it is initially necessary to monitor its behavior. In this case, the first challenge is to define good metrics with regards to the relevant QoS goals, whether they are performance or cost related, or even customized at the application level. Determining the monitoring interval is also important, as it can result in a costly process, for example, when the time grain between collections is very small, which can generate a high load on the system, or when the metrics of interest are highly customized, derived from other data that has yet to be collected and processed as well. The Planning phase estimates the amount of resources that must be provisioned. This is a difficult task because the scheduler needs to quickly determine the number of resources needed, without actually observing the behavior of the system under these conditions, based only on the model of the real application [138].

Although new paradigms and techniques for monitoring, planning and adapting DSP systems are continually evolving, many of the proposed solutions do not have the theoretical basis necessary to guarantee high levels of performance, accuracy and validity in their execution. Considering this, control theory can be a powerful tool to help systems that seek to maintain QoS goals in the context presented so far. Given their analytical approach and the variety of techniques used in their design, formal controllers can provide guarantees in terms of effectiveness when tracking the variables of interest. Control theory tries to solve the challenges in the field of computing systems in different scopes [12; 43; 56], having expanded its contributions to software adaptation in recent years [60; 61; 62; 134].

However, even with the observed advances, applying control techniques in computer systems still presents itself as a challenge [107; 79; 81]. In summary, we can highlight two main concerns: (i) the difficulty in abstracting the complex behavior of software into a suitable mathematical form for controller design, and (ii) the lack of methodologies and research in Software Engineering and Distributed Systems addressing solutions and control modeling in a robust and more generalized way [64]. Specifically, software applications can have complex, often non-linear, interactions with the hardware that support their execution, in addition to dynamic changes resulting from fluctuations in workload, for example, that can invalidate a model previously effective.

Considering this, in order to obtain mathematical models for computing systems, several methods of system identification can be used [109]. Generally speaking, these methods attempt to obtain an accurate model generated from input and output data collected from a running system. In this context, there are the First-Order Plus Dead Time systems, which are the object of study of the solution proposed by this work. First-order systems are those in which the relationship between input and output can be translated into a first-order differential equation. Thus, it can be said that such models are an approximation of the dynamic response of a process variable of a system to a given influence factor. FOPDT modeling has been widely used to capture process dynamics for the purpose of designing controllers for various systems, and has also been widely studied in the context of several works [27; 152; 40; 17; 168; 114].

Thus, among the benefits acquired when using analytical models to generate equations capable of describing a system, is the ability to extend the applicability of the controller design methodology to any system that can be formalized through its intermediate model. This means that, for example, defining and evaluating a control approach for systems aimed at DSP applications would allow the construction of controllers for a variety of systems, respecting their specificities, that closely resemble this model.

In this process, to better understand how to model such systems and build controllers, it is important to understand the concept of Feedback Control, commonly used in efforts to integrate control theory into computer systems [92]. Such an approach consists of using the system's output to determine adjustments to its input. This output is then compared to a given user-defined reference value, resulting in an error signal that is used by the controller to generate a corrective action on the system. By definition, the main purpose of a feedback system is to track a reference value while optimizing three aspects: (i) minimize the delay until the system reaches its expected state; (ii) minimize exaggerated corrective actions that can lead to over-provisioning of resources; and (iii) minimize the steady-state error, that is, the difference between the system output and a given reference value.

Considering this, any function that computes an output based on an input can be used as a controller in a feedback loop. Still, Proportional-Integral-Derivative (PID) is considered to be the most studied and used in practice given its level of robustness, clarity and simplicity [150; 92; 69; 154; 100; 70]. In general terms, this controller makes use of three components called proportional, integral and derivative. The first acts proportionally on the absolute error, while the second considers an accumulation of tracked errors, being especially useful to reduce steady-state errors. The derivative controller, on the other hand, makes use of the tendency to increase or reduce the error, being known as the control that tries to predict the future. It is important to note that tuning such gains is not an easy task, especially for less experienced users. Poor tuning can, for example, destabilize the system, increasing overshoot, delay, and consequently, the overall performance of the application being processed.

These controllers, however, are widely associated with systems that seek to control only one output acting on a single input, while several systems may be interested in controlling multiple outputs simultaneously. Control theory has evolved to define different approaches to deal with this use case. For example, MIMO (Multiple-Input, Multiple-Output) controllers act on multiple inputs controlling multiple outputs. Similarly, SIMO (Single-Input, Multiple-Output) controllers generate a single action to be applied over a given input, while looking at multiple outputs [135]. Another possible approach, for example, suggests the use of independent controllers acting on the system. However, the application of these types of controllers to DSP systems faces the same problems regarding their modeling, as mentioned above, adding the need to deal with multiple variables of interest simultaneously, which can also be conflicting in some manners.

Considering this, the objectives of this work are defined below.

1.2 Objectives

This work aims to apply and evaluate the use of control theory methodologies in micro-batch DSP systems. The scope here includes mainly the use of control strategies to model and orchestrate DSP systems with multiple objectives in order to provide solutions that maintain QoS goals based on performance and cost metrics. We also want to evaluate the applicability of system identification methods, such as the use of First-Order Plus Dead Time (FOPDT) models, regarding their effectiveness and accuracy in describing the behavior of micro-batch DSP systems.

In order to achieve the main objectives described above, specific goals of this work are as follows:

- Apply system identification methods to formalize the considered system;
- Evaluate how well FOPDT models describe the behavior of the considered system;
- Propose PID controllers aiming to track performance and cost metrics, both independently and simultaneously (SIMO approach);
- Tune the proposed PID controllers analytically, following strategies described in the literature;
- Integrate the proposed controllers to Asperathos by customizing control and monitoring plugins for the system;
- Compare the proposed controllers to different control approaches such as fixed action, manual tuning of PID gains, and independent controllers acting simultaneously on the system;
- Evaluate the proposed controllers regarding system throughput, resource allocation, execution cost and SLA violation when running an application that disaggregates energy data;

- Provide insights into the challenges related to the applicability of control theory concepts to micro-batch DSP systems;

1.2.1 Methodology

In terms of the methodology followed to achieve the desired goals, for the construction of the proposed FOPDT model, we collected data that reflected the behavior of the Asperathos system, in terms of its input and output metrics, when executing an application that disaggregates energy data using Non-Intrusive Load Monitoring (NILM) techniques.

Note that neural networks are popularly used to classify data, therefore, the application used to generate such model could consist of a variety of other classification algorithms. We also consider continuous monitoring environments, where the number of sensors does not change abruptly, and the tasks are considered to be homogeneous, which means they take approximately the same amount of time to finish.

Moreover, the model was generated using Matlab's *System Identification Toolbox* [120]. To evaluate that, we used the Normalized Root Mean Square Error (NRMSE) metric to determine how close the model is to represent the real system.

In order to analytically tune the performance PID controller, we used the generated FOPDT model together with Matlab's *Control System Toolbox* [118] and *PID Tuner* [119] component. For the SIMO approach, the methodology observed on the literature was used to implement the combination of the outputs for the considered intermediate controllers. Besides that, a utility function was defined to determine the level of user preference for each type of control, either focused on performance or on cost metrics.

To evaluate that, the defined experimental design presented a comparative analysis of scenarios composed of the different control approaches considered in the scope of this work. To have a general understanding of the experiment results, a descriptive analysis of the collected data was presented. Besides that, to analyze the significance of such results, a *t-test* analysis was performed. The control solutions were evaluated in terms of system throughput, resource allocation, execution cost and SLA violation.

1.3 Contributions

Contributions of this work include the use of system identification methods for modeling the system used as a use case, based on FOPDT models, followed by an evaluation of this modeling, showing its strengths and weaknesses, its effectiveness and accuracy in describing the behavior of the system and how we can benefit from using filters. The model was generated using a disaggreation algorithm based on NILM techniques. Since neural networks are commonly used to classify data, we could generalize the approach in terms of other classification algorithms with the same structure as the disaggregation one.

Furthermore, a PID controller is proposed to show how to apply this solution when trying to maintain QoS metrics and, from that, we present an analytical approach to tune the controller based on the initially proposed FOPDT model. It is important to highlight that, after the tuning techniques were applied, the derivative term did not pose as a significant factor to influence the given PID controller towards stability. In fact, the derivative term ended up making the system noisy, which negatively affected the output we were trying to obtain. This way, the final proposal is a Proportional-Integral (PI) controller, that was later evaluated.

In order to further evaluate the control methods applied here, we also propose a SIMO controller responsible for combining actions based on performance and cost metrics. For this, two different controllers were implemented, one for each type of metric of interest, being the SIMO controller responsible for combining the outputs of both, and applying a single corrective action on the system. The solution uses a utility function that allows users to prioritize the considered metrics according to their needs, and consequently, determine the weight of each control action to be combined. From the experiments, we saw that this helps to deal with conflicting interests and generate improvements in resource allocation.

To orchestrate the application we used Asperathos [6], a framework that automates the execution of data processing applications while maintaining custom QoS objectives. This framework uses Kubernetes [8] clusters to deploy containerized applications and process a set of tasks organized as jobs. In addition to that, its architecture allows the customization of plugins, which enabled the integration of the controllers and monitors proposed here, as well as the configuration necessary for its operation.

To validate the solution, we created a Kubernetes job that disaggregates a stream of micro-batches of energy data, collected from sensors and meters distributed in houses, buildings and industries [3]. From this, the controllers were evaluated in different scenarios. Results show that the PI controller performed better when compared to controllers that act on a fixed step, while the SIMO controller behaved better than scenarios in which the cost and performance controllers acted simultaneously. Besides that, we learned that the weight defined by users for each metric directly impacts how well each controller is able to follow its reference signal, and consequently, in breaking SLA agreements.

1.4 Summary

This work is organized as follows. In Chapter 2 important concepts for the understanding of this work are presented, such as Quality of Service, container orchestration and the Asperathos framework. Then, Chapter 3 presents an overview of the literature in which this research is inserted. In Chapter 4, we deepen the knowledge about control theory, and a formal definition of PID controllers, which is one of the objects of study of this work.

In Chapter 5 we present how the system was modeled using FOPDT-type models, in addition to an evaluation of their ability to describe the system addressed. Then, Chapter 6 presents the proposed PI controller with a focus on performance metrics, how its gains were tuned, as well as an evaluation that used Asperathos and an energy data processing application. In Chapter 7 we present a different controller with a focus on cost metrics, as well as the SIMO controller resulting from the combination of the control actions of this controller and the first one focused on performance metrics. We also present an statistical analysis of the data collected for the evaluation of the proposed approaches, better detailed on Appendix B, and a descriptive analysis on Appendix A.

Finally, Chapter 8 presents the conclusions of this work as well as some future activities that can be performed to extend this thesis.

Chapter 2

Background

2.1 Quality of Service

Quality of Service (QoS) can be defined as a characteristic related to the behavior of a given service, which determines the degree of satisfaction of a user when using it. For example, the study of QoS in networks, in its various subfields, comprises a set of service requirements to be met by the network while transporting a data stream [54]. In this context, possible metrics to consider are throughput, packet loss rate, delay and jitter, as well as reliability and availability measures and models that generally define the performance of a network.

On the other hand, as a concept, QoS can also be extended beyond its original network-related aspects to systems and operations. In this case, other QoS attributes are widely used in the evaluation of various systems, such as response time, throughput, failure probability, availability, among others [172]. Often, such objectives are closely related to requirements for the proper functioning of a given system. Therefore, when deploying applications, it is important that customers and service providers define QoS goals, in order to formally establish expectations and obligations, allowing actions that guarantee customer satisfaction. Furthermore, ensuring that QoS goals are met in distributed systems is fundamentally dependent on characteristics that vary from application to application. That is, each customer or service provider probably has their own interests in specific metrics to be used, as well as what is the acceptable level of satisfaction for each of them [23].

Considering this, this work proposes a control solution for data processing applications while maintaining QoS goals. For such applications, different metrics may be important, given, for example, the nature of its workflow. Here, we consider batch and stream processing. A batch can be defined as a predefined amount of data to be processed without any end-user interaction, within a specific time frame. On the other hand, stream processing is characterized by a workload of unknown total size, processed in real time [26].

Thus, it is clear that controlling such applications in order to maintain QoS goals can be a very difficult task and specific to the type of workflow considered, such as those described above. For batch processing, metrics such as execution progress can be interesting, while for stream processing, relevant metrics can be the rate of new work items entering the system or the size of the queue of tasks to be processed. Thus, it is important to understand how to deal with each of these approaches and their combination with QoS goals, in order to incorporate control algorithms in computing systems of this type, as we will see in the applications of interest presented in the rest of this work.

2.2 Container orchestration

By definition, container orchestrators are systems that manage clusters of machines, which in turn, serve applications deployed in a set of containers spread across the cluster [155]. When we think of orchestration services, one of their most attractive features is the automation provided for tasks such as initialization, provisioning and deployment of resources and applications, as well as monitoring the behavior of clusters, scheduling strategies and tolerance to failures. In general terms, the orchestration tool selects an appropriate host for the container being initialized based on specific rules defined by the user, and thus, operates them through the aforementioned functionalities.

In this context, containers offer a logical packaging mechanism that favors micro-services, smaller units of a complete application which will run entirely within clusters of containers, with all their requirements and dependencies. This model allows applications to be abstracted from the environment in which they are running, ensuring that developers are only concerned with the logic and dependencies of their systems, while operators are concerned with the management and deployment of the resources themselves. Considering this, users can create images of environments configured to run their applications, and such images can then be deployed in a container without much effort, in addition to being replicated as often as needed.

The commercial success of containers began with the arise of Docker, a popular containerization method supported by large platforms like Google Cloud Platform and container orchestrators like Kubernetes. Docker makes use of Linux isolation frameworks such as kernel *namespaces* to provide isolation from networks, filesystems, and similar resources, and *cgroups* to limit the use of resources such as memory, CPU, and bandwidth. Docker even provides access to a repository from which images can be retrieved and stored.

Considering this, the use of containers to orchestrate applications in production is continuously growing. Two recent surveys produced by OpenStack [1] and Kubernetes [44], one of the biggest container orchestrators on the market, confirm this information. OpenStack research states that in the year 2020, 66% of their users who need containers for orchestration use Kubernetes alongside OpenStack. The Kubernetes survey, in turn, shows that in that same year, 92% of participating users used the orchestrator for production systems, a significant increase of 300% from the 23% reported in the first survey conducted in 2016 [44]. For example, OpenStack and Kubernetes have been used in production processing large amounts of data by the European Organization for Nuclear Research (CERN), amounting to about 25 petabytes of analyzed data [121].

In more detail, Kubernetes, which today has one of the most active repositories on GitHub [68], has among its most relevant features the ability to run applications in cloud environments, given its integration with several providers. Kubernetes also allows users to customize the deployment of their services based on the resources available from the provider, such as SSD, network, etc. Many other functionalities are available such as restart automation, scaling and replication, including monitoring, load balancing and fault tolerance for federated distributed clouds.

Another example of an orchestration service known on the market is Docker Swarm [7]. It uses its cluster management capabilities to handle a set of Docker containers as a single service, through its containerization platform, the Docker Engine. One of the great advantages of this orchestrator is its compatibility with Docker, a widely used container provisioning service. This way, any tool that already uses Docker can use Docker Swarm to scale seamlessly across multiple hosts.

In this work, we use a configurable framework for automation and service orchestration called Asperathos, which supports the management of Kubernetes clusters, widely used in the evaluation of the proposed solution. More details on how Asperathos orchestrates applications on a Kubernetes cluster can be found in Section 2.3 below.

2.3 Asperathos

Desired goals when using an orchestration service may vary depending on the application being considered. For example, some applications aim to process their workloads in a predefined time, such as batch processing, while others seek to keep the flow of tasks performed at a certain level, such as stream processing. Regardless of the chosen metric, a smart allocation of resources is essential to avoid a potential waste of resources, which can cause considerable financial impacts for application managers [104].

To deal with this type of problem, we present Asperathos [6; 69; 145; 22], a framework that provides automation in the execution of Big Data applications in the cloud, while meeting predefined requirements for a certain quality of service. It also allows varied levels of configuration particular to the application of interest being executed. In general terms, Asperathos is composed of a set of components that communicate through REST APIs, and can be configured to act on Kubernetes, Spark and OpenStack clusters, for example, controlling their resources in order to maximize performance metrics of a given application.

2.3.1 Architecture

A diagram of the Asperathos architecture and how its components communicate is shown in Figure 2.1. The system consists of three main modules and an optional one: *(i)* the Manager, entry point for the user and responsible for initiating new submissions; *(ii)* the Monitor, in charge of grouping and publishing metrics collected from the application; *(iii)* the Controller, responsible for adjusting the amount of resources allocated to an application according to a certain control algorithm; and *(iv)* the Visualizer, an optional component that makes use of visualization tools to provide graphs about the progress of applications being executed by Asperathos.

For a better understanding of how the solution proposed in this work makes use of Asperathos, it is necessary to know some details about the components mentioned above:

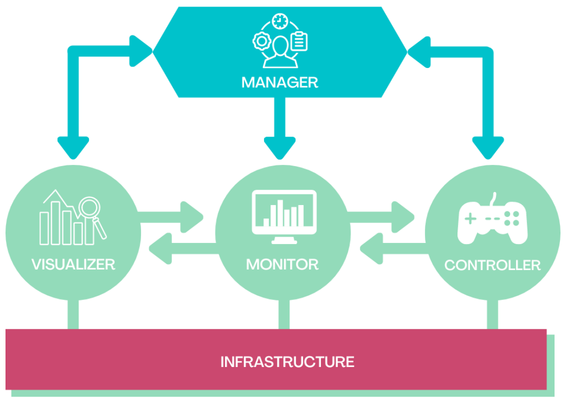


Figure 2.1: Asperathos architecture.

Manager. Responsible for initiating new submissions and providing the necessary environment for an application to run. For example, in a Kubernetes cluster, a submission comprises the application itself, packaged in a Docker image, and its respective parameters. The submission also contains details on the chosen control approach, e.g. controller settings, and additional scaling options such as the desired maximum number of replicas. In addition, the Manager is responsible for starting the other components of the system.

Monitor. Responsible for monitoring the infrastructure, platform or containers executing a submission and collecting metrics related to both the application, for example its progress over time, and the resources used, such as CPU and memory usage. As the metrics of interest can vary based on user needs, Asperathos allows such specifics to be implemented through the customization of plugins, which will be detailed later.

Controller. Responsible for triggering actions on the resources used, based on the metrics collected by the Monitor and the QoS goals defined by the users. In this context, metrics are retrieved from some application responsible for storing them, and for example, used to decide whether it is necessary to increase the amount of resources to be able to finish a task on time, while maintaining an acceptable level of QoS or, alternatively, decrease the amount of resources and minimize costs. Such decisions are made according to the results of a given control algorithm, which can also be customized by the user.

Visualizer. Provides a visualization platform where the user can follow the progress of a given application through graphics and images. In general terms, the service consumes the metrics collected by the Monitor, and from there, generates graphs that are incorporated into a Dashboard, later accessible to the appropriate users. Currently, Asperathos provides this Dashboard through Grafana, a well-known tool for monitoring and visualizing at the infrastructure and application level.

2.3.2 Custom plugins

Asperathos can be tailored to the specific needs of a given application by customizing plugins. As mentioned before, the plugable architecture of the components detailed in Section 2.3.1, allows various implementations of plugins to be integrated into the system, which is a feature used in this work for the addition of controllers and monitors that implement the control algorithms specific to the problem we seek to solve.

As a basis for the customization developed here, we use the set of plugins named *KubeJobs* already offered by Asperathos, which runs data processing applications in containers inside a Kubernetes cluster. As such, the *KubeJobs* package implements plugins to deploy, monitor and scale *Jobs* in a Kubernetes cluster.

A common workflow of an execution that used *KubeJobs* is detailed below:

- A given client sends a POST request to the Asperathos Manager containing a submission, and all the necessary configurations for the execution;
- The Manager creates a Redis service in the cluster, responsible for queuing the tasks to be processed;
- The containers responsible for running the application are initialized in the cluster;
- Tasks from the Redis queue start to be consumed by the application in the containers;
- The Monitor is triggered by the Manager and periodically retrieves the number of tasks processed from the Redis queue, or any other metrics of interest for the given application;

- Once the metrics are published, the Manager starts the Controller, which consumes the metrics and makes decisions about resource scaling based on some predefined control logic;
- The process repeats until all the tasks in the queue are completed, and if all goes well, the user-defined QoS parameters are respected.

As the workflow above suggests, Asperathos previously only supported batch processing applications by default. Thus, especially for the Monitor, its implementation was entirely based on the needs and demands of this type of application. For example, the performance metrics collected by the Monitor are focused on the progress of the submission and how far the remaining tasks are from being completed, at the same time that the QoS goals defined for that execution are respected.

However, in order for Asperathos to be compatible with different types of applications, especially those of stream processing, a new plugin for the Monitor was developed. Its focus is on collecting metrics related to a stream of data, such as the arrival and output rates of items in the system. From the point of view of QoS metrics, new reference values become of interest, which means that, in the new plugin, the way we calculate how far the system is from meeting the expectations for these metrics has also been modified. Details about this implementation will be presented later in this work, in Chapter 6.

2.4 Use case: Energy data processing in real time

Studies show that the energy consumption of commercial and residential buildings around the world is estimated at $30 - 40\%$ of what is currently generated [2] and, due to the increase in the use of new electronic devices, between other things, this number is expected to grow even further, increasing total energy consumption costs. Thus, it is clear that there is a growing demand for methods that help to reduce energy expenditure, especially from observations on how this energy is spent in practice. Based on information collected in real time on the consumption of various appliances in a building, for example, it is possible to generate recommendations to users in order to eliminate bad habits, such as leaving several lights on or appliances unnecessarily turned on at night, and reduce from 5 to 15% your current energy consumption [159; 74].

Considering this, recent energy monitoring techniques have used various algorithms with the purpose of transforming aggregate consumption measurements into information about the individual use of appliances. An example of such techniques is the Non-Intrusive Load Monitoring (NILM), also known as Non-Intrusive Appliance Load Monitoring (NIALM) [55], which has been increasing in popularity since companies responsible for distributing electricity began to collect more information about their customers' energy consumption.

Such collection takes place through the use of devices, such as sensors or energy meters, responsible for informing in real time about the consumption of equipment connected in a particular residence or building where they were installed. In this way, from the use of NIALM algorithms and consequent processing of disaggregation tasks, it is possible to know which devices have been on, for how long and even identify possible failures in the electrical installation. It is important to note that, if not managed properly, disaggregation tasks can accumulate, compromising the performance of the systems involved [76; 14; 55].

Thus, the application used as a use case in this work uses a NIALM technique to provide disaggregation of consumption measurements for a system called *LiteMe* [3], currently in its Beta version. LiteMe is based on Deep Neural Networks (DNNs), which, in general, receive energy measurements from sensors installed in a given building of interest, for example, and return the disaggregated consumption of each of the appliances in that building. Thus, real-time recommendations are generated for both suppliers and their users, making it possible to optimize the early detection of contracted energy usage patterns, in addition to reducing the cost of electricity bills. In this case, customers are interested in receiving real-time information about their consumption, such as peaks in demand that can lead to penalties. At the same time, the company providing such recommendations wants to satisfy these requirements as quickly as possible while minimizing computational costs.

Finally, in its Beta version, LiteMe's NIALM application runs on a single machine, supporting only a few users. However, for the version used in its release, the current workload is expected to increase at least a hundredfold, making the management and processing of such data much more cost-relevant, for example. Thus, a low-cost, distributed approach that still respects the user-defined QoS goals becomes quite useful. By using Asperathos together with the proposed control solution, it is possible to orchestrate a cluster of machines serving multiple containers running NIALM, making it possible to process a higher number of measurements through a *Job* that performs a stream of disaggregation tasks.

2.4.1 The workload

Considering this use case, in order to be able to run LiteMe NIALM instances on Asperathos, the framework used for container orchestration in this work, some adaptations were made. To provide such an orchestration, Asperathos requires that the executed application can be replicated, and that the work items to be processed are different from one another. However, for each disaggregator, LiteMe's NIALM application originally consumes data coming from a specific sensor, repeatedly checking for new measurements, which makes it difficult to replicate it in a distributed way by defining different work items.

Thus, our approach implements an adapted disaggregation model that processes measurements from various sensors in a predefined time interval. Such configuration becomes a work item of a *Job* in Asperathos, including the identifier of a sensor, a given time interval and, as required by the neural network used by LiteMe, a matrix containing the history of measurements previously disaggregated, which will help in the predictions made by the model. This approach was properly validated, comparing the results obtained in a standard run of NIALM with the results of the *Job* in Asperathos, which proved to be 100% compatible.

For the experiments and models generated here, we use a sample of 3600 real disaggregation tasks, regarding sensors used by LiteMe's solution. A snippet of this data containing the items sent to Asperathos can be seen below:

```
5c49bc284f0cccfec76febba;12/02/2020 14:17:00;12/02/2020 14:18:00;0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 42622.85572600632 8843.726676422843
    43390.676841217086 8664.516311959386292349306
5c49bc284f0cccfec76febba;12/02/2020 14:19:00;12/02/2020 14:20:00;0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 42622.85572600632 8843.726676422843 43390.676841217086
    8664.516311959396 42368.8547588586 8445.171428938385 41956.72369581707
     8025.18390685924413236673315
5c49bc284f0cccfec76febba;12/02/2020 14:19:00;12/02/2020 14:20:00;0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 42622.85572600632 8843.726676422843 43390.676841217086
    8664.516311959396 42368.8547588586 8445.171428938385 41956.72369581707
     8025.18390685929294139756O815
5c49bc284f0cccfec76febba;12/02/2020 14:17:00;12/02/2020 14:18:00;0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 42622.85572600632 8843.726676422843
    43390.676841217086 8664.51631195932866083678566
5c49bc284f0cccfec76febba;12/02/2020 14:16:00;12/02/2020 14:17:00;0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
    0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 42622.85572600632
    8843.72667642284361627188673З
```

Every line in the snippet corresponds to an item to be processed by Asperathos. The values are separated by a semicolon, being the first value the identifier of a sensor, the second a given time interval and the final value is the matrix containing the history of measurements previously disaggregated. For each replica running a Nialm instance, an item like this is processed. Such process consists of communicating with a given database that contains the aggregated data regarding the given sensor, for the given timestamp, and that later uses the given matrix to perform the disaggregation.

One consideration about this workload is that, from the perspective of the neural network used to process the data, if the values in the disaggregation matrix change, it does not make a difference in the processing itself. Such data are the power values that enter the network, and the magnitude of the data, in that sense, does not interfere in the processing.

If, by any means, you wish to change the number of layers or the model of the neural network in question, it also does not affect significantly the profile of the task. For instance, changing the number of the layers in the network can change a little the duration of the task, but such task would still be only CPU intensive, reading and returning the same amount of data.

Regarding the arrival rate of new work items, it does not vary in short time scales for an environment like the one considered here, nor in seconds or even in minutes. In the experiments later presented in Chapters 6 and 7, we consider variations that lasted a few minutes, considering, for example, load variations that were imposed not by the increase in the amount of data and sensors, but by data accumulation issues (e.g., a sensor that lost its connectivity starts to slowly send the data delayed as the individual bandwidth is low).

It is also important to highlight that, for this workload we consider tasks that are homogeneous in size, and therefore take the same amount of time to finish its processing, which is a characteristic of the tasks we focus on, like the disaggregation ones presented here.

Chapter 3

Related work

This chapter surveys the literature regarding the topics addressed in this work. Initially, studies on Quality of Service, focusing on web applications and DSP systems are presented, considering important aspects such as metrics and approaches most used in the literature for the use cases that fit the study proposed by this work. Then, we bridge the gap between modeling DSP systems and using control theory approaches applied to resource scaling scenarios. Control techniques that consider multiple variables of interest are also presented, along with how this can be applied in the context of computing and data processing systems discussed in this work, in addition to more specific approaches in this area such as fuzzy logic and machine learning.

3.1 Quality of service

3.1.1 Web applications

Web applications are each day more present in the daily lives of millions of people, being indispensable for both large companies and individual users. This translates into strict requirements regarding the performance, availability and reliability of such applications. Thus, non-functional aspects of web services started to need specific methods and tools for their management, the best known called Quality of Service (QoS) [122].

There is a certain consensus in determining the relevant QoS dimensions in a system as being: (i) those related to execution time, such as availability, transfer rate, reliability, among others; (ii) those related to configuration and cost management, which concerns the cost of a service and its ability to meet industry standards; and (iii) those related to security, such as authentication, authorization, confidentiality, tracking and data encryption, among others [45].

Within these dimensions, several works explore the challenge of maintaining QoS metrics in web services. Pioneers like Ranjan et al. [140] present QuID, a QoS-oriented framework for server migration in data centers. Results showed that the proposed algorithm generated up to 68% savings in resource allocation for certain workloads when compared to static allocation techniques. In general, such techniques would have to over-provision servers to be able to meet demand, as also noted by other works [21; 39]. Urgaonkar et al. [157], in turn, presents a solution for modeling multi-layer web applications that is based on the use of a network of queues to represent how these layers work together to process requests. The evaluation of the model demonstrated its usefulness in managing resources for web applications when executed under different workloads and different operating bottlenecks.

In this context, some works [38; 20; 37] show the benefits of dynamically reallocating resources based on workload variations rather than statically over-provisioning them. The objective is to maintain the QoS goals of on-demand applications, that is, to adapt to their needs as they arise. In this case, better resource utilization can be achieved, and the system may be able to react to an unexpected increase in the considered workload. Lopes et al. [111] addresses the problem of how a web application provider should plan long-term reservation contracts with an IaaS provider, so that its profitability is increased. For this, a model is proposed that can be used to guide this capacity planning activity.

In turn, Urgaonkar et al. [158] presents techniques for provisioning CPU and network resources on shared resource platforms, running potentially antagonistic applications. The nodes that make up the cluster are monitored at the kernel level, and this data is used to guide the allocation of applications to run on them. Then, techniques are proposed to over-reserve the resources of the cluster in a controlled manner, so that the platform can maximize its revenue, while providing QoS guarantees for the given web applications. However, the proposed system does not describe any approach to continuous monitoring of the environment, which is a limitation. In addition, provisioning decisions are based on the steady state of the application, rather than considering rapidly changing demands.

Yu and Lin et al. [170], in turn, propose a broker for web services with QoS restrictions. Each request arrives with a specific QoS goal associated with it. The broker then accepts requests only if the service's current workload allows QoS goals to continue to be maintained. Algorithms for resource allocation and reconfiguration of the already provisioned architecture are also proposed. However, the reference scenario only considers situations in which many customers are requesting the same type of service, limiting the evaluation of the approach.

Finally, Rahman et al. [139] presents challenges when describing QoS requirements for web applications. Based on a review of the state of the art in the area, the main QoS concerns and characteristics to be inserted in the proposed model are selected, whose objective is to predict QoS metrics during the development and implementation of web applications. Accordingly, Guitart et al. [72] presents a systematic review on the performance management of web services. It describes approaches to request scheduling, policy determination, and resource provisioning based on QoS objectives.

These studies are related to this work given that web requests are a continuous flow of small tasks to be processed, fitting the type of application of interest proposed here, which are micro-batch DSP systems. The basis for the provisioning and scaling of web applications, in addition to their monitoring, can be used together with control theory solutions to control QoS metrics, objective of this work. In this case, this area of study can also be considered the basis for the development of several approaches that make use of these metrics as part of their systems, as is the case of the DSP paradigm that we will address in the next section, and which is also part of the context of the solution proposed here.

3.1.2 Data stream processing

Data Stream Processing has emerged over the years as the reference paradigm for analyzing fast, continuous information streams, which often need to be processed with low latency to extract valuable information from raw data. When dealing with unlimited data streams, DSP applications are typically long-lived and therefore likely to experience varying workloads or working conditions over time. In order to maintain a good level of service given this variability, a lot of effort was spent on studying strategies for adapting DSP systems at runtime.

Dayarathna and Perera [49] review DSP systems in the broader field of event processing, discussing the architectural choices behind the most popular platforms and recent advances in applications, e.g. online learning, graph analysis, and more. In general lines, a systematic literature review of works dealing with the runtime adaptation of DSP systems is presented by Qin et al. [137]. Martin et al. [116; 117] present the applicability of a distributed tool for processing streams of events called *StreamMine3G* together with the analysis of data from social networks in real time. The parallelization and scaling of operators are extensively reviewed by Röger and Mayer [143], discussing issues associated with the implementation and control of application elasticity. Assunção et al. [53] also analyzes solutions for elasticity of DSP systems, with emphasis on systems deployed in highly distributed computing environments. As elasticity is considered a fundamental characteristic for modern DSP systems, a large number of works have investigated solutions for operator scheduling. It can be seen that the vast majority of them focus on the horizontal scaling of resources [33; 34; 66; 67; 73; 110].

In this context, Vijayakumar et al. [162] considered the problem of provisioning resources for DSP applications in virtualized or cloud environments. An algorithm capable of dealing with dynamic patterns in the arrival of data was developed. The algorithm promises to avoid any slowdowns in the processing of applications, while also conserving previously configured budgets to be spent on resources. The results show that the resource provisioning algorithm correctly converges to the optimal CPU allocation based on the data arrival rate and the respective computational needs. Furthermore, the algorithm proves to be effective even when significant changes occur in data arrival rates.

Ishii et al. [89], in turn, propose a system called *ElasticStream* that dynamically allocates cloud computating resources to a streaming data processing application. To minimize the amount spent when using a cloud environment, and at the same time meet the SLA, a linear programming problem was formulated to optimize costs as a trade-off between application latency and amount spent. A system was also implemented to dynamically add or remove computing resources on top of the middleware of the DSP system, named *System S*. It was then confirmed that the proposed approach could save 80% of costs while maintaining application latency compared to a less intelligent approach.

On another front, Das et al. [47] explores the effects of the batch size of data to be processed on the performance of streaming workloads. He points out that a larger batch size can increase the end-to-end latency between receiving data and getting the corresponding result and that, ideally, the system should operate using a batch size that minimizes latency, ensuring that data is processed as fast as it is received. Thus, a simple but robust control algorithm is proposed, which automatically adapts the batch size according to the application needs. Experiments have shown that it is possible to guarantee system stability and low latency for a wide range of workloads despite wide variations in data rates and operating conditions.

The definition of the metrics of interest to adapt DSP applications at runtime is an important step in the monitoring process of system elasticity. Application-oriented metrics capture aspects of the operation that can be directly perceived by users, for example, latency and processing accuracy. On the other hand, system-oriented metrics capture aspects of the system that can impact the application, such as resource utilization. Among application-oriented metrics, the most popular are latency and throughput. Latency plays an important role as DSP applications are often required to process events with near real-time requirements, and many adaptation solutions rely on latency as their primary performance metric [66; 94; 164; 163; 106]. In addition to latency, another popular performance metric is throughput [167; 93; 87; 36], which measures the number of work items processed per unit of time. This is the metric also chosen in the context of this work to be evaluated during the processing of the considered micro-batches of data.

In contrast, among the system-oriented metrics, the most used is resource utilization in general [33; 67; 73; 77; 28], which captures the utilization level of a computing resource, usually CPU. As in different application domains, usage is often used in conjunction with threshold-based adaptation policies, where actions are triggered whenever the usage level violates a predefined threshold value [67; 33; 82]. Such approaches tend to be more heuristic, and depend on a prior knowledge of the application's operation to accurately determine good limits that favor the system, and consequently, the data processing.

Some works explore control theory methods to design adaptation policies. In this case, three main entities are identified: perturbation, decision variables and system configuration. Disturbances represent dynamics that cannot be controlled, although their future value can be predicted (at least in the short term), while decision variables are mapped to adaptation actions. Control theory approaches are used in conjunction with a variety of adaptation mechanisms: operator scaling [28; 84; 50], load distribution [123; 124], backpressure [42], load shedding [95], among others. For example, Mencagli et al. [123] uses PID controllers and fuzzy logic in their solution for adapting DSP systems. PID controllers regulate load distribution between parallel operator instances, while fuzzy logic controls scheduling actions on longer timescales. Kalyvianaki et al. [95], instead, designs a discrete-time control algorithm for load shedding, which at each time step selects the number of tuples to be processed in order to maintain processing latency within a predefined value.

In the context of this work, the adaptation of DSP systems through the use of control theory methods was widely discussed. For example, the studies listed helped in choosing the QoS metrics of interest, considering the characteristics of this type of system. However, for cases of multi-objective control, rarely addressed in this context, there is a limitation in terms of defining the relationship between conflicting metrics, such as those evaluated here, performance and cost. In this case, the higher the performance, there is a tendency for higher costs with more resource utilization, when what is normally desired by users is high performance and low cost. Thus, determining a relationship between these variables, within the scope of regulatory control, is one of the challenges of this work.

3.2 Provisioning and scaling of resources

3.2.1 Overview

In the context of cloud computing, to efficiently use its elasticity capabilities, it is vital to automatically and intelligently provision resources, since over-provisioning leads to waste and unnecessary cost to users, while under-provisioning causes degradation of the performance and violation of SLA agreements. This mechanism of dynamically acquiring or releasing resources to meet varying QoS requirements is called *auto scaling*. This is extremely important in the context of real-time adaptation of DSP systems given the variations that may occur in the environment, in the workload, or possible disturbances, and which require quick actions to be able to maintain QoS metrics. Several works are dedicated to observing the state of the art in this area of research, defining sub-areas of interest, and pointing out the advances achieved so far [148; 85; 138; 112].

Among these areas, estimating the resources to be used is one of the most important points in the auto scaling process, as it determines the efficiency of provisioning. Its purpose is to identify the minimum amount of resources needed to process a workload, in order to determine if they are needed and how to perform scaling operations. An accurate estimate allows the autoscaler to quickly converge to optimal resource provisioning. On the other hand, estimation errors result in under-provisioning, which leads to a lengthy process and increased SLA violations, or over-provisioning, which incurs unnecessary costs, as already mentioned. There are several approaches to help estimate these resources, from more basic methods such as rule-based ones, to more sophisticated ones such as fuzzy inference, mathematical modeling, others that use machine learning, and combinations between them.

Rule-based approaches are widely adopted by industry-provided autoscalers such as Amazon's Auto Scaling Service [4]. Its kernel is a set of predefined rules consisting of trigger conditions and corresponding actions, e.g. *"If CPU utilization reaches* 70%*, add* 2 *instances"* and *"If CPU utilization drops below* 40%*, remove* 1 *instance"*. Users can use any metric, high or low level, to define trigger conditions, with the autoscaler's aim being to keep the parameters in question within the predefined upper and lower limits. Theoretically, simple rule-based approaches do not involve a precise resource estimate, which is just determined as a fixed action, such as adding or removing a certain amount or percentage of instances. This is considered to be the simplest version of auto scaling, typically being used as a benchmark for comparison in works that focus on other aspects of auto scaling, such as the work by Dawoud et al. [48], which aims to to compare vertical and horizontal scaling, or prototyping works, as done by Iqbal et al. [88]. In the same context, Morais et al. [125] presents an in-depth analysis of automatic and reactive provisioning services, identifying efficiencies and limitations of the approach.

While a simple rule-based autoscaler is easy to implement, it has two significant drawbacks. The first is that it requires detailed knowledge of application characteristics to determine limits and appropriate actions. Al-Haidari et al. [15] conducted a study to show that these parameters significantly affect the performance of the autoscaler. The second is that this approach is not able to adapt when dynamic changes occur in the workload or application. This means that fixed values for control actions, whether scaling horizontally or vertically, would become inappropriate when the workload changes dramatically. For example, if the application is provisioned on 4 instances at the start, adding 1 instance will increase the capacity by 25%. After a while, the cluster will have increased to 10 instances due to the increased workload and adding 1 instance in this case only adds 10% to the capacity. This can cause a delay in responding to sudden changes in workload, which we want to avoid with this work.

Considering this, RightScale et al. [9] proposed an interesting variation on the simple rule-based approach. Its central idea is to allow each instance to decide whether to shrink or expand the cluster according to predefined rules and then use a voting approach to make the final decision. Calcavecchia et al. [32] also proposed an approach based on decentralized rules. In its proposal, the instances are connected as a P2P network. Each instance contacts its neighbors to get its status, and from that, decides whether to remove itself or start a new instance, in a probability derived from the observed status.

The solution proposed here uses basic methods of provisioning and automatic scaling of resources, such as those based on rules and fixed actions, to evaluate the performance of the proposed controllers in relation to these simpler approaches. In this way, we seek to fill gaps in terms of dynamics in determining the size of the control action, and in defining the moments when actions must be taken according to the tracking error calculation described in detail in Chapters 4 and 6.

3.2.2 Control theory

Considering more robust approaches for resource estimation, in the context of provisioning and automatic scheduling are the mathematical models, abstracted in known concepts such as, for example, queuing theory and control theory, which is the object of study of great focus of this work. The interest in applying control theory concepts in the development of computer systems is not recent. Some works widely reference approaches already proposed in the literature that use control techniques to try to solve problems in the context of web applications, databases, traffic and data transfer, among others [98; 83; 18;

105]. Lui Sha et al. [146] proposed the use of control theory together with queue management to regulate resources of an Apache HTTP server. Parekh et al. [132], in turn, applied control theory concepts to propose a solution that sought to meet QoS objectives based on IBM Lotus Domino Server performance metrics, while Yixin et al. [51] presented a solution to simultaneously adjust memory and CPU resources of an Apache HTTP server, based on control techniques with multiple objectives.

In this context, Hellerstein et al. [78; 79; 81] presents a series of works that seek to expose the challenges related to the application of control theory in the development of computing systems, which are still relevant. Considering the provisioning of resources on demand, the delay observed when instantiating or deleting a virtual machine is a characteristic that can complicate the design of a given controller, depending, for example, on the frequency these operations happen. In addition, the modeling used to describe computing systems, as a data processing application, is a recurring challenge, including in this work. Hellerstein cites four possible approaches, among them the use of system identification techniques and stochastic processes such as queuing theory. Another factor observed is that control theory has been applied in the scope of computer systems in simpler controllers, such as PI control and even just Integral control, as presented by [132]. Furthermore, there is a dominance of solutions that use SISO-type (Single-Input, Single-Output) controllers, as opposed to controllers with multiple inputs and outputs such as MIMO and SIMO. Finally, Hellerstein points out that the scientific community tends to pay little attention to the use of filters, to the choice of observed outputs and the possible delays related to a system, which can lead to poor performance of the developed controllers, especially resulting in a long accommodation time or significant fluctuations.

Abdelzaher et al. [12] presents a sequence of use cases where control theory can be applied to different types of computer systems. Some examples include memory management of database systems, control of CPU utilization in embedded, distributed and real-time systems, automation of workload management in virtualized data centers, and in performance control and energy use in data centers. Furthermore, Maggio et al. [113] proposes a new approach to apply methods and tools based on feedback control during the actual design of computing systems components. This is done initially by obtaining an open-loop model of the system to be controlled, different from the closed-loop, or feedback systems, commonly discussed in the literature. After that, a controller model was designed and, according to the authors, the solution is considered simpler and less computationally bulky than the more classical approach. To support the proposed approach, a preemptive scheduler for a multi-tasking system was developed.

Papadopoulos et al. [131] suggests in their work that the use of feedback control applied to computing systems is still low given the complexity of defining good dynamic models of the considered systems. Thus, an approach is proposed to model computing systems trying to avoid the mentioned complexity, capturing only the dynamics that are really relevant to them. In the scope considered, the author seeks to base the modeling on what he calls the physical phenomena of a system. One of the exposed use cases is a framework to define resource allocation, such as memory and number of servers, in order to keep the application progress consistent with the desired throughput rate, for example. In this case, the author points out that the phenomenon to be considered is how the progress dynamically reacts to the variation of resources. A model is then built based on this metric alone, and is subsequently validated to show its effectiveness in terms of the proposed reference values. With this, the author concludes that there is validity in isolating the main physical phenomenon of the system to build models from them.

In turn, Kihl et al. [99] discusses in their work challenges for optimizing the use of resources in data centers in the cloud. For example, elasticity controllers must allocate enough resources to a running application to provide acceptable QoS and avoid unnecessary over-provisioning expenses. Possible solutions listed are based on machine learning algorithms, control theory and statistical analysis of workloads. This way, a new research area named Cloud Control is proposed, which addresses cloud management problems using control theory. The promise is that such an approach will transform current data centers into dynamic and self-managing infrastructures, guaranteeing quality of service.

Barna et al. [25] presents a control theory-based approach to automating the scaling of web applications instantiated in a cloud environment. For this, a PID controller was designed for a specific web application, and it was instantiated in two clouds, one public (Amazon EC2) and another private (SAVI). The experiments showed that the controller was able to effectively maintain given performance goals (CPU utilization) in the two clouds, which indicated a potential portability of such controller between different cloud environments, limited by the number of experiments conducted and metrics observed. It is important to highlight that the work considered the control of only one layer of the web application, and the tuning of the PID controller gains was done manually, although based on a coherent line of thought, valuing a faster response to possible values of error (high proportional), and a slower response to accumulated errors, only when they are considered large (low integral). The desired CPU utilization was between 70% and 40%, which is usually not the goal of regulatory control, which aims to track a single reference value rather than a range of values. This feature was based on the fact that the system input is the number of servers in a cluster, which limits the controller's action to integers, requiring these values to be rounded which makes it difficult to keep the output exactly at a reference value. Interesting future works include evaluating the use of PID controllers in cascade, trying to include the other layers of the web application, with each controller responsible for acting considering a different metric. This approach involves problems relevant to this thesis such as the combination and orchestration of such controllers, maintaining QoS goals when provisioning resources.

In the same topic as Barna et al. [25], Cerqueira and Solis [35] proposes the use of a PID controller to promote automatic scaling in a cloud environment, based on an efficient allocation of the amount of containers that serve requests in a web environment. Baresi et al. [24], in turn, presents an automatic scaling technique that allows containerized applications to scale their resources at the virtual machine and container level. This is possible thanks to a system component that behaves like a feedback controller, based on monitoring, analysis, planning and execution, commonly called MAPE. The presented solution extends a self-adapting framework previously developed by the authors called ECoWare. The new approach is decentralized, so that each layer of the considered application is provided with a local controller, responsible for maintaining a given variable of interest, such as response time, at acceptable levels even in the presence of disturbances. Based on control theory, this is done by calculating the resources needed to satisfy this condition, and applying equivalent actions on the system, considering its current state. The experiments showed that the developed controller performs better than Amazon's auto scaling functionality, also in containerized environments.

Multi-objective control

In addition to using a single auto-adaptive controller, some autoscalers employ more than one adaptive framework. In these schedulers, several non-adaptive or self-adaptive controllers are connected simultaneously, which start to actively switch the control of the system based on its performance. Self-adaptive controllers continuously adjust themselves in parallel, however, at any given time, only the highest performing controller selected can provision resources. Patikirikorala et al. [133] employed this approach, and Ali-Eldin et al. [16] proposed a self-adaptive switching approach based on classification of application workload characteristics. Chen et al. [41] proposed an approach that trains multiple resource estimation models and dynamically selects the one that performs best.

In this same context, Yixin et al. [52] proposes a framework to describe how we could scale control problems in computer systems. Such framework considers two aspects, one that scales based on the number of inputs and outputs of the system to be controlled, and another that scales based on different control objectives. In this study, it was observed that there is a tendency towards the development of multiple-input and multiple-output systems, motivated by scenarios that considered various control objectives. However, such systems can end up being complicated by potential conflicts between the objectives considered. It is also discussed how the framework can help in the decisions of decomposing a large-scale system into smaller and, consequently, more easily manageable systems. The proposed strategies range from centralized schemes that work well when the latency between the controller and the system is small, to distributed approaches that are effective when the considered objectives can be decomposed into independent objectives. That said, in practice there are still cases where the latency in the communication between the controller and the system is long, and cases where it is difficult to decompose big goals.

Thus, considering the works studied here, some challenges persist regarding the use of control theory techniques for the scheduling of micro-batch DSP systems. Few works have used the formal process of identifying systems to model their solutions, especially in the context of computing systems. The same applies for tuning the gains of PID controllers, which mostly used manual techniques to achieve the results obtained, in contrast to a more analytical approach such as the one used in this work. In addition, the adaptation of the approached systems has a greater focus on resources such as CPU and memory, and the horizontal scaling of virtual machines is still challenging given the characteristics of the control action for this type of resource, which must be an integer. Consequently, the orchestration of containers in the context of control theory was also little addressed in the studies considered, which is one of the objectives of this work. Finally, works that use SIMO-type controllers to control multiple-objective systems were more applied in the context of Engineering, as opposed to the results proposed in the following chapters.

3.2.3 Other approaches

The auto scaling of resources based on fuzzy logic is considered an advance over the rule-based approach, as they rely on the use of fuzzy inference, whose core is a set of predefined *if -else* rules, used to make provisioning decisions. The main advantage of fuzzy inference compared to simple rule-based reasoning is that it allows users to use linguistic terms like "high, medium, low", rather than precise numbers to define the conditions and actions to be taken, which makes it easier for humans to effectively represent their knowledge about the system of interest.

Fuzzy inference works as follows: inputs are first diffused using defined membership functions; then, such inputs are used to trigger actions related to all rules in parallel; the results of the rules are then combined and finally used as an output for control decisions. Representative approaches of this type include the one proposed by Frey et al. [65] and the work done by Lama et al. [101]. Since manually designing the set of rules is a heavy task, and these sets cannot timely handle changes in the environment and application, fuzzy logic-based schedulers are commonly coupled with machine learning techniques to automatically and dynamically learn such set of rules [102; 166; 91].

Tian et al. [154] presents a work that proposes an adaptive PI control system based on fuzzy logic. Through a basic PI controller, the system is able to dynamically adjust the proportion between incoming requests and their acceptance time, in order to reduce the acceptance of requests when the server is overloaded. Considering that in a real Internet environment the network and the server load continuously vary in an uncertain interval, a fuzzy control approach was used to adaptively adjust the PI controller parameters. The results demonstrate that in a stable load environment, the proposed system works as well as a basic PI controller, but acts much more stably and faster dealing with fluctuating loads.

Another solution that applies fuzzy logic is proposed by Wei et al. in [165]. They present an eQoS framework that monitors and controls the quality of service provided to clients on web servers. Two control approaches to manage server resources are proposed: one based on queuing theory and feedback control, and another based on fuzzy control within the framework itself. The authors argue that the former is desirable when web servers can be characterized as an M/G/1 queuing system, and if not, the fuzzy approach is more appropriate.

On another front are machine learning techniques, which are applied in resource estimation to dynamically build the resource consumption model under a specific workload, also called online learning. This way, different applications can use the schedulers without custom configurations. They are also more robust to runtime changes, as the learning algorithm can self-adaptively adjust the model in real time to any reasonable event. Machine learning algorithms are often implemented as feedback controllers. Despite their ease of use and flexibility, machine learning approaches have one major drawback. It takes time for them to converge to a stable model and therefore this can lead to the autoscaler performing poorly during the active learning period. Of course, the performance of the application is also affected in this process. Furthermore, it is difficult to predict the time it takes to converge, as it varies from case to case and from algorithm to algorithm. Online learning used by existing autoscalers can be divided into two types: reinforcement and regression learning [138].

To exemplify, here we cite examples of works that use reinforcement learning, which aims to allow the system to learn how to adaptively react in a specific environment to maximize its gain or reward. This approach is suitable for dealing with automated control problems, such as the auto scaling one that we are discussing here [173; 169; 153; 103]. For the auto scaling problem, the goal of the learning algorithm is to generate a table specifying the best provisioning action in each state. The learning process is similar to a trial and error approach. The learning algorithm chooses an individual operation and then observes the result. If the result is positive, the scheduler is more likely to take the same action the next time it faces a similar situation. The most used reinforcement learning algorithm in the literature is Q-learning.

For the context of this work, this type of approach combined with control theory techniques has not been addressed so far given the specificities involved in machine learning models.

Chapter 4

Control systems

4.1 Feedback control

Considering a given system, in general lines, the problem to be solved when using control techniques is to find the best input configuration that will produce an output compatible with a given reference value. When using feedback control, this problem is solved by continuously comparing the current output of the system with the reference to be followed, applying actions on its input in order to correct any differences that may exist between the observed output and the previously defined reference. [92].

For example, considering a system that has: (i) as input, a set of virtual machines responsible for processing requests; (ii) as an output, the number of requests processed per second; and (iii) as a reference to be maintained, the value of 1000 requests per second; a feedback control system would work as follows. If the output observed at a given moment is equal to 2000 requests per second, it means that the system is accelerated, as this value is far above the reference. The controller then triggers an action on the input seeking to reduce the number of requests processed, that is, it is necessary to shut down a certain number of virtual machines. Similarly, if the output is much below the reference, for example, 500 requests per second, the input would be adjusted upwards, i.e., more virtual machines would be turned on to increase the computational power, and consequently, increase the output value.

It is important to note that the process of approximating the output value to the defined reference value happens gradually, continuously comparing the two values and applying the necessary actions on the input at each iteration. Thus, because of the use of the output value

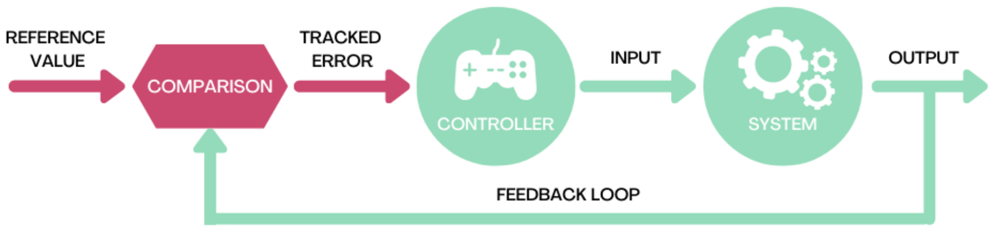


Figure 4.1: Generic architecture of a feedback control loop.

to determine which control action to take, the feedback control is said to close the loop, as we can see in Figure 4.1.

Considering this, the above definition of feedback control introduces important concepts for the understanding of this work:

System input. Also known as control input, it is the directly manipulated variable capable of influencing the behavior of the system being controlled.

System output. Also known as process output, it is the variable that the system must be able to influence. As this cannot happen directly, the controller influences the output through the input.

Reference value. Value that must be replicated as the system's output. There is no feedback control without defining a reference value. It is also important to note that the system will try to reproduce *exactly* this value, that is, it is not possible to use feedback control to maintain a metric between one value and another, or in the *best* possible value.

Tracked error. Distance between the current observed system output value and the desired reference value. In a simple formula, the tracked error can be defined as in (4.1):

$$e = reference - output(t) \tag{4.1}$$

Corrective action. Also called a control action, this variable defines an action to be taken over the input, calculated based on the tracked error. The controller can calculate this action without having detailed knowledge about the behavior of the system, but it has to know which direction stimulates the output up or down, that is, if the input needs to be increased or decreased to influence the output upwards, for example. The magnitude of the action will depend on which control strategy is being used for the calculation.

With these well-defined concepts, the understanding of feedback control is facilitated, but it is still necessary to know if each iteration converges to a certain expected value and how fast such convergence happens. However, this process should not result in a destabilization of the system, which can happen, for example, when applying corrective actions of very high magnitude. Such actions can generate oscillations, which are a result of the variation between different configurations in a fast and aggressive way. As opposed to that, if the corrective action is of very low magnitude, the system will respond slowly to disturbances, and the tracked error will persist for several iterations, also compromising the system's ability to maintain the previously defined QoS goals. Thus, it is said that in order to achieve a satisfactory system response, the corrective action must have a sufficiently high magnitude so as not to make the system unstable.

In addition, achieving good performance is essential when controlling systems, that is, it is necessary to ensure that the response time to changes is fast enough to eliminate the tracked error in an agile way. The quality of the system is measured by the accuracy with which it is able to follow the reference value. Considering this, we can say that the behavior of a feedback control system is usually evaluated in terms of stability, performance and accuracy levels.

It turns out that, often, not all of these goals can be achieved simultaneously. In particular, the design of a feedback system usually involves a trade-off between stability and performance, because a system that responds too quickly, that is, that in theory performs well, also tends to oscillate more, given sudden changes in corrective actions. In this case, generally speaking, it may be better to make several small adjustments quickly, rather than a few major adjustments occasionally. In the first case, the corrective action will be taken quickly, before the tracked error increases substantially. In the second case, the magnitude of the error will probably be greater, which means a greater chance of exaggerated compensations, with an associated risk of instability.

Nevertheless, it is important to analyze the peculiarities of each system before deciding how often and how intensely the controller should act. For instance, using the same example of the system input being virtual machines, turning this infrastructure on and off too often can be costly. Or even when we have more than one controller acting on the same system, it is common for one to act more frequently than the other, and one to apply more intense corrective actions than the other. Therefore, trade-offs should always be carefully analyzed on a case-by-case basis.

4.1.1 Choosing control variables

There will often be more than one variable candidate for input and output of controller systems. However, it is important to carefully choose which metrics would be best used to achieve a given control objective, especially given their nature and, consequently, from the point of view of a solution implementation, how such variables fit into the system.

In this section we present criteria that can be used to evaluate different possible control variables [92]. First, for system input variables, it is important to consider:

Availability. Only values that can be influenced directly and immediately are indicated to be input variables.

Responsiveness. The system must respond quickly to a change in its input to achieve good performance and high accuracy when following a given reference value. Therefore, it is important to avoid inputs that may suffer from latency or delays.

Granularity. It is desirable to be able to adjust the system input in small increments to achieve high accuracy in tracking a reference value.

Directionality. It is necessary to know whether increasing the input results in a increased or decreased output. If an increase in the input results in a decrease in the output, then it is necessary to use an inverted loop when building the controller.

In contrast, for system output variables, one must consider:

Availability. The variable must be quickly and accurately observable, in a reliable way, preferably without delay.

Relevance. The chosen output should be a good measure of the behavior you want to control. In cases where the interest is in measuring the quality of service of the system in general, then a variety of metrics can be used to represent this idea, and the process of choosing which variable is the most informative for the considered goal should be very careful.

Responsiveness. The output metric should quickly integrate changes in the system state or behavior. Once again, this means avoiding variables that suffer from delays, such as when the output metric is the result of calculating an average of values, or when the value needs to propagate through the system in order to be properly observed.

Smoothness. Disturbances in the output variable will result in abrupt control actions, which should normally be avoided. Therefore, it is desirable that the output does not need to be filtered, for example, or suffer from noise.

4.2 Controller types

Controllers can be designed based on a variety of objectives, the most common being disturbance rejection, optimization and regulatory control [12]. The first aims to ensure that disturbances acting on the system do not significantly affect the observed output. Optimization seeks to obtain the best possible value for the output, such as configuring the maximum number of clients on an Apache Server in order to minimize the system's response time. In turn, regulatory control seeks to ensure that the observed output is equal to a user-defined reference value, which is the case of the feedback systems that we used for the solution of this work.

Considering regulatory control, any function that computes an output based on an input can be used as a controller in a feedback loop. A standard approach to this type of computation in orchestrators is for controllers to act with a predefined response each time the tracked error indicates the need for a control action. In other words, this controller horizontally scales the system by a certain fixed number of replicas.

Opposite to this solution, PID controllers, widely studied in the universe of control theory, offer a more sophisticated approach. To compose the corrective action to be applied on the system, such controllers add three modules called *proportional, integral* and *derivative* control. Thus, as in any feedback system, the controller output is computed based on the input, but modulated by three different gains, defined here as: proportional k_p, integral k_i, and derivative k_d [92; 86].

4.2.1 Proportional control

Letting the magnitude of the corrective action depend on the magnitude of the tracked error causes a low magnitude error to generate small adjustments, while a high magnitude error results in a larger corrective action. The simplest way to achieve this goal is to let the controller output be proportional to the tracked error, as we see in (4.2):

$$u_{proportional}(t) = k_p e(t) \tag{4.2}$$

Where $u_{proportional}$ represents the proportional control action, the gain k_p is a positive constant, and e is the tracked error value.

However, in general, this approach is insufficient to eliminate tracked errors when the system is in what we call a steady-state, that is, when all transient responses disappear. In proportional control, the system output will always be less than the desired setpoint. This is because, by definition, such controllers produce a non-zero output only if it receives a non-zero input. So, if the tracked error disappears, then the proportional controller will not be able to produce an output. The consequence is that some residual error will continue to persist when the only type of control used is proportional.

One way to try to reduce the impact of this situation is to increase the gain k_p, but if it is too large, other problems can be introduced as a result of an exaggerated corrective action, generating system instability. Less elegant than this option is to intentionally set the reference value to be greater than what is actually desired, so that even when the system output is less than this value, the result will still be satisfactory. Looking for a better solution for such cases, there is a controller that can automatically eliminate steady-state errors, as we will see next in the definition of integral control.

4.2.2 Integral control

In general terms, integral control is based on using the tracked error accumulated over time. While proportional control, which is based on momentary error, and therefore has its effect reduced by generating a low magnitude corrective action for very small errors, integral control amplifies these small errors by accumulating them and, over time, this will result in more significant corrective actions. Thus, this feature is what makes the integral control a good approach for reducing steady-state errors.

The output of an integral controller is proportional to the integral of the tracked error over time, as seen in (4.3):

$$u_{integral}(t) = k_i \int_0^t e(\tau) d\tau \tag{4.3}$$

Where $u_{integral}$ represents the integral control action, the gain k_i is a positive constant, and the described integral is simply a generalization of the sum of the tracked errors over time.

This dependence on past values implies that an integral controller has non-trivial dynamics reflected in the behavior of the system. For example, if a positive tracked error persists for a long time, then the sum of errors calculated by the integral controller will increase. The result of this is a positive corrective action over the system input even when the tracked error has been eliminated at a given point in time. In this situation, the system output will be greater than the reference value, generating a negative error that will decrease the value of the sum of accumulated errors until eventual stabilization of the system.

Considering this, depending on the chosen values for the gain k_i, these oscillations can decrease more or less quickly. Therefore, it is important to find the best parameters for the gains of an integral controller, as well as for the other PID components, in order to obtain the most acceptable behavior of the system, which we call here controller tuning.

4.2.3 Derivative control

While the integral controller monitors values in the past, the derivative control proposes to anticipate the future. By definition, a derivative is characterized as the rate of change over a certain value, so considering the tracked error in a system, if its derivative is positive, it can be said that such an error is currently growing, and vice-versa for when the derivative is negative. Thus, a corrective action can be immediately applied, even if the tracked error value is still small, in order to act on the system before the observed error becomes too large.

Thus, the output of the derivative controller is proportional to the derivative of the tracked error, as we see in (4.4):

$$u_{derivative}(t) = k_d \frac{de(t)}{dt} \tag{4.4}$$

Where $u_{derivative}$ represents the derivative control action, the controller gain k_d is a positive constant, and the derivative of e can be approximated as the amount of change in e since its last observation.

One problem with derivative control is the potential presence of high frequency noise in the system. While, by nature, an integral controller smoothes the effect of possible noise, considering the calculation of the derivative of a polluted signal, the derivative controller will potentialize the effect of such noise. For this reason, there is a need to smooth these signals apart from the controller's natural process. However, in addition to adding complexity, this also creates a risk of compromising the motivation of the derivative control itself, because if the signal is excessively smoothed, the variations important for the control would also be eliminated.

Thus, while proportional control is central to feedback systems, and integral control is necessary to eliminate steady-state errors, derivative control is less used in practice. In fact, the controllers known as PI are the most frequently used variant in applications [92; 12].

4.2.4 PID control: proportional, integral, derivative

A controller that unites the three previously mentioned terms (proportional, integral and derivative) is called a PID controller. We have (i) the proportional control that simply multiplies the tracked error by a gain k_p, reacting to the absolute value of the error; (ii) the integral control that multiplies the accumulated of previous errors by a gain k_i, which makes this controller especially useful for reducing steady-state errors; and finally, (iii) the derivative control that tries to anticipate the future considering the rate of change in the error value, multiplying its derivative by a gain k_d. Thus, in the time domain, the output of a PID controller is defined by (4.5):

$$u_{pid}(t) = k_p e(t) + k_i \int_0^t e(\tau)d\tau + k_d \frac{de(t)}{dt} \tag{4.5}$$

Where $u_{pid}(t)$ is the output of the PID controller and e is the given tracked error. This expression can be transformed to the frequency domain, resulting in the transfer function at (4.6) below:

$$K_{pid}(s) = k_p + \frac{k_i}{s} + k_d s \tag{4.6}$$

Note that the process of tuning such gains for each term in a PID controller is an extremely important task, but it may not be simple. There are several known techniques to improve the effectiveness of these controllers in a system, increasing the speed of convergence and eliminating exaggerated control actions as well as steady-state errors. Some examples of these techniques are those of Ziegler-Nichols (ZN), Frequency Domain Method (FDM), and Damped Oscillation Method (DOM) [150]. In Chapter 7 of this document, we will further discuss these techniques and the importance of an analytically based tuning to obtain a stable and efficient control of the system.

4.3 Considerations when implementing a controller

Implementing a feedback loop in a PID controller involves some decisions in addition to the general stability and performance concerns already presented in this work. Among them are the following:

Actuator saturation. In principle, there is no limit to the magnitude of the control action. When the considered gain is sufficiently large, the result of the action can also be arbitrarily large. The problem can arise if the system plant cannot faithfully follow this signal, for example, due to lack of power to respond to a very large action. In the case of a set of servers, their maximum number is limited, so once they are all connected, demands beyond that number from the controller cannot be met. At the other extreme, the number of servers, fundamentally, can never be less than zero, and so on.

The component that translates the value of the control action into an action itself is called an actuator. Thus, the problem of the controlled system not being able to follow a certain control action is called actuator saturation. This situation can greatly limit the performance of the system as a whole. It is also important to note that such limitations will not be detected by the transfer function of the system. Instead, the highest magnitude of the control action of a given system must be estimated separately, and then the ability of that system to follow that signal must be evaluated.

Windup of the integral term. Actuator saturation can have a peculiar effect when it occurs in a controller with an integral term. When this happens, as the actuator is not able to pass proper values to the plant, tracked errors will not be corrected, and therefore will persist. The integral term, then, will add them indefinitely, reaching very high values. This can be a problem in the future when the system is no longer saturated and the error changes sign, taking a long time for the integrator to decrease what was accumulated, and go back to being efficient regarding the current tracked error. To prevent this kind of situation, one can simply stop adding values to the integral term when saturation is identified.

Previous definition of integral term. The opposite problem occurs when the system is initialized for the first time or when large changes are made to the reference value. Such sudden changes can easily saturate actuators. In such cases, it is possible to pre-set a value for the controller integral term to an appropriate value, in order to make the system respond more smoothly to changes in the reference value.

Choosing an actuation interval. The frequency in the application of control actions can follow two principles. In general, it is better to take many small corrective actions quickly than a few actions of greater magnitude. In particular, it is beneficial to respond to any deviation from the desired behavior before it has a chance to become too big. Doing so not only makes it easier to keep the process under control, it also prevents the effect of large deviations on the system. However, there is not much benefit in manipulating a process much faster than it can respond. Generally speaking, for example, if the dynamics of the controlled system changes on a time scale of minutes, then control actions should be applied every few seconds. If the process only changes once or twice a day, then applying actions every few minutes should suffice.

4.4 Challenges for computer systems

The growing interest in the use of control theory for solutions aimed at computing systems such as managing web applications, scheduling cloud resources, and various network operations, raises a discussion on how to better incorporate such concepts and strategies in the computational universe. While techniques involving control theory are already widely used in the design of industrial processes and equipment, for example, their use in the design of computer systems is relatively new [81].

Such systems typically operate by sharing resources between applications, for example, a Kubernetes cluster with multiple nodes running different containers. Therefore, in a real scenario, it is desirable that the allocation of these resources is carefully defined, in order to dynamically distribute them so that the applications in question can maintain their goals in terms of quality of service [149]. Considering this, QoS metrics are constantly used to monitor services and evaluate their efficiency according to the preferences of a given customer. However, as mentioned earlier, providing such services without violating SLAs (Service Level Agreements) is a major computing challenge [148].

Control strategies can be used to achieve these goals, allied to the scheduling of services and provisioning of resources in the context of computing systems. In general, we can list 5 steps needed to apply control techniques in such systems: (i) define the control objective; (ii) describe the variables of interest to the system in formal terms; (iii) model the relationship between the input of the system and the respective observed output; (iv) define the controller design; and (v) evaluate the resulting control system.

However, these steps present certain challenges specific to the nature of computing systems. For example, increasing the complexity of the controller typically increases the complexity of its implementation, which can result in the development of a less robust solution, sensitive to unpredictable disturbances, such as an abrupt variation in the input rate of a given workload. Furthermore, depending on the type of input and output calculated by the system, the more components there are in its design, such as actuators and sensors, there may be some impact on the system's performance that must be considered.

Another challenge is to incorporate the principles and techniques of control engineering into a kind of framework that can be applied to different applications in computer systems, as is the case of widely used design patterns, such as MVC (Model-View-Controller), and Publish-Subscriber, for example. In this context, several problems could benefit from a control framework, such as regulating the provisioning of resources in order to achieve certain QoS objectives, or managing the number of processes triggered to maximize data throughput of a system.

Computing systems may still have other characteristics that present themselves as challenges for integration with control theory. It is common for such systems to seek to maintain different QoS metrics at a certain reference value, for example, low CPU and memory usage and, consequently, low execution cost in terms of used resources. In a scenario like this, it would be necessary to observe different metrics, and to use more than one controller to maintain different reference values. Managing independent controllers can be a problem, even if their actions are executed at different points in time, as we will see in more detail in Chapter 7, as they can compete with each other, destabilizing the system. Such a situation happens especially in cases where the metrics observed have a conflicting nature, as is the case of low cost and high performance, for example. Solutions in control theory propose to solve similar cases through the use of controllers with multiple objectives, as we will also see in Chapter 7, however, as they are more advanced approaches, the integration with computer systems is also made difficult.

Chapter 5

Applying First-Order Plus Dead Time models to DSP systems

5.1 System identification methods

System identification is an area of mathematical modeling that uses input and output data, often experimentally collected, to identify the dynamic characteristics of a system. Often, the models obtained from this process are then used to build controllers of different types, such as the Proportional-Integral-Derivative. According to the literature, different approaches can be adopted to generate such models [30]:

Analytical Method (Phenomenological). Equations and parameters are determined based on the principles of Physics, Chemistry and Biology, using mass and energy balance equations.

Empirical or Heuristic Method (Experimental). The system is considered a "black box" with certain inputs and outputs. In this case, a set of experiments are carried out to obtain such parameters during the evolution of the system to its steady-state, from which a model of the system would be determined. In general, such models are less complex than analytically obtained models.

In practice, it is common to combine the two approaches, acting in two stages. The first takes into account the physical laws and the particular working conditions to establish hypotheses about the structure and properties of the model to be identified. In the second, more experimental stage, the hypotheses previously established are adopted, and experimental measurements are made to build the model. Thus, in general, the identification process can be divided into the following steps [30; 92]:

- **Dynamic tests and data collection.** The collected data has the same role as constitutive equations in theoretical modeling, as they provide the specific basis for the development of a given model. As the model obtained by the identification method is entirely based on experimental data, it is important to note that information that is not contained in the data cannot freely appear in the model, just as it is unreasonable to expect an unspecified constitutive equation to contribute to the quality of the final theoretical model;
- **Correct choice of the structure of the models.** Consists of determining the terms that should compose the models by recognizing the importance of these different terms, using the so-called identification data and avoiding the over-parametrization that occurs when more terms than necessary are used;
- **Estimation of parameters using suitable numerical methods.**
- **Verification of the models' ability to represent the studied process.**

Thus, we can define that a system identification method is an experimental approach used to derive mathematical models of dynamic systems, using data collected from their behavior. Here, we consider that the main objective of this method is to generate models that are later used to design controllers for regulatory processes. It is important to note that the modeling process invariably involves approximations since many real systems are, to some extent, nonlinear, time-varying, and distributed. Thus, it is unlikely that any set of models will contain the structure of the system in all its details. In this case, a more realistic objective is to identify a model that provides an acceptable approximation in the context of the application in which it is used.

Considering this, in the design of a controller, the first step often involves determining the model using step response data, where the objective is to determine a transfer function for a First-Order Plus Dead Time (FOPDT) system or Second-Order Plus Dead

Time (SOPDT) [46]. When the controller is of the Proportional-Integrative or Proportional-Integral-Derivative type, we assume that the model will often have a continuous-time transfer function, based on the FOPDT or SOPDT structure. This is because the result obtained through these types of models is a good approximation for the monotonic step response, and without the overshoot of many processes found in the industrial control area, and in our case, also computational. Methods for estimating the parameters of these transfer functions using the step response are popular, as for instance, Ziegler-Nichols and Oldenbourg-Satorious.

In this context, a transfer function is used to encapsulate, in the frequency domain, the effect that a system has on its input. For example, if the input is given by $u(s)$, then the output $y(s)$ is simply given by (5.1):

$$y(s) = G(s)u(s) \tag{5.1}$$

Where $G(s)$ is the system transfer function in the frequency domain. In general terms, the output $y(s)$ can then be transformed to the time domain and, from that, the behavior of the system is obtained.

This way, if we have a good theoretical model to represent a system, it is possible to derive its transfer function from the given model by calculating a *Laplace* transformation of the differential equation that describes the dynamics of the system. However, it is often not possible to obtain a good theoretical model that represents a system, as we saw in Section 4.4. In this case, it is necessary to calculate the transfer function from the system identification process described above, considering two aspects in particular:

Static relationship of input and output. If a change of a certain magnitude is applied to the input, what is the size and direction of the change reflected in the output?

Process dynamic response. If an input change is applied suddenly, how long does it take for the system to respond?

Such questions are answered through observations. All measurements are made in an open-loop run, and without a controller calculating the next control action. Thus, it is possible to adjust the system input arbitrarily, in order to observe only the actual response of the system to a given step size, for a pre-defined period of time, independent of any control action.

Briefly, to obtain the static characteristics of the process it is necessary to apply a change in the input, wait for the system to stabilize, and then save the output result. It is important to note that there is a minimum change to be applied to the input to be able to see any reflection in the system output. For large magnitude changes, the system may begin to saturate and no longer reflect the expected changes in the output. In this context, the magnitude of the process gain provides information about the size of the control action needed to observe significant changes in the system output. In addition, the process gain signal provides information about the direction of the input-output relationship.

To measure dynamic responses, it is necessary to observe the behavior of the system during its execution. For this, the system must be initially at rest, that is, without any change in the input. Then, a sudden change is applied to the input, and we save all the operations that occurred until the output of the system is obtained. It is important to repeat the process a few times to accommodate different amplitudes in the input values. It is also important to note, that some things might influence the system behavior and can be reflected in the measurements. For instance, if we run the same experiment multiple times, and detect a considerable difference in the observed outputs, this is possibly an indication of the amount of noise in the system.

To set up the transfer function itself, three main parameters are considered:

Process gain K. Ratio of the applied input value and the final system output value after all transient effects disappear.

Time constant T. The time required for the system to settle into a new steady-state after experiencing some disturbance. The time constant is normally defined as the time required for the process to reach $2/3$ of its final value.

Delay τ. Delay time until a change in input starts to affect the output of the system.

Some models and tuning methods use these same parameters to identify systems. Here, we consider self-regulating processes, those that in response to a given input reach a steady-state, possibly after some delay, but without overshoot or oscillations. The frequency domain model in (5.2) is often used to describe the one-step response for this type of process, where K is the process gain, T is time constant and τ is the delay.

$$H(s) = \frac{K}{1 + sT} e^{-s\tau} \tag{5.2}$$

In this context, there are the First-Order Plus Dead Time systems, which are the object of study of the solution proposed by this work. FOPDT modeling has been widely used to capture process dynamics for the purpose of designing controllers for various systems, and has also been widely studied in the context of several works [27; 152; 40; 17; 168; 114].

For example, in the design of a feedback control loop it is important to consider its performance when there is a change in the tracked reference value, or a disturbance in the workload in question. In addition, it is also important to be aware of the system's level of robustness to changes in process characteristics, and its fragility to the variation of its own parameters. Therefore, approximations such as FOPDT are useful to allow such considerations to be modeled, which justifies their increasing use for this type of solution [126].

5.2 Application use case: Asperathos

Considering self-regulatory processes, we apply FOPDT modeling to model Asperathos, a system previously described in Section 2.3. In line with the concepts presented in the previous section, we have seen that FOPDT systems are commonly used to empirically describe various dynamic processes, and therefore, it is a good starting approach. Thus, we first seek to observe how well the model to be generated is able to represent Asperathos, evaluating the results using known metrics detailed later. Next, we discuss possible alternative approaches to consider when generating the model, which will then be used to tune a PI controller developed for Asperathos (more details in Chapter 6).

Thus, for the construction of the model, we collected data that reflected the behavior of the Asperathos system when executing an application that disaggregates energy data, as described in Section 2.4. A few considerations were followed, as described bellow:

- The workload is a sample of real energy disaggregation data, provided by the LiteMe solution. This is representative of the environment where the controller will actuate. A snippet of the items sent to Asperathos for processing can be found in Section 2.4;

- The control input for this system is the number of Kubernetes replicas ready to process new items. For this modeling, we used 7 replicas. This number was chosen because, considering the computational power available, it is an input of a size enough to see a reflection in the system output, and not too large so the system may begin to saturate;
- The observed control output for this system is the throughput of items processed by Asperathos. This is what we call *step response*, which is the observed behavior of the system when there is a step change in the input;
- The arrival rate of new items was 8 items per second. Considering the control input, this number of items is enough to keep the replicas busy, since each replica can process 1 item at a time;
- The system should be initially at rest, i.e., no other operations should be executing prior to the experiment initialization;
- No control action is taken. In this case, we implemented a customized plugin on Asperathos that determines the number of replicas according to a list of values previously passed to the system. This changes only happen when a pre-defined amount of time has passed, in our case, 5 minutes, which is enough to see the system stable for the application we are processing;
- At this point, since there is no controller in action, we also do not defined any reference value to be followed.

It is important to highlight that, for the curve fitting algorithm to be successful in identifying the system, it is necessary that the observations are made for a sufficient time for the observed output to reach the steady-state, when there are no more relevant variations in the variable of interest. The system response under these conditions was then observed, and the respective models and transfer functions were generated using linear regression techniques and Matlab's *System Identification Toolbox* [120].

Table 5.1: Model NRMSE.

Model	**Average NRMSE**	**Model**	**Average NRMSE**
Model 1	0.637	Model 6	0.613
Model 2	0.639	**Model 7**	**0.605**
Model 3	0.613	Model 8	0.606
Model 4	0.609	Model 9	0.613
Model 5	0.611	Model 10	0.654

In total, 10 repetitions were performed following these considerations, each one generating a model from the collected data. This is necessary since we need to be sure no disturbances or other noise is affecting the system when collecting the measurements, and therefore, affecting the generated models. Next, we compare such models with a real execution in the system by computing the average square error (NRMSE), which is the average of the squares of the differences between observed and predicted values. This is a commonly used measure to define how close a linear regression model is to the real system it wants to represent.

Thus, in order to calculate the average NRMSE for each model, we compared the data collected for each real execution with a given model, and then averaged the results. This means that, for Model 1 in Table 5.1, 0.637 is the mean NRMSE obtained from the squared errors calculated in comparing that model with the data collected from repetition number 1 to repetition number 10, and so on for each one of the others. All the results are listed in Table 5.1.

Note that, for a linear regression model, there is an error that is introduced when the model does not actually fit the data. Therefore, the goal of a good model is to minimize this error. Considering this, from the results in Table 5.1, we see that the best performance indicators, that is, the lowest mean NRMSE values, are those of the 7, 8 and 4 models, respectively. In Figure 5.1, we see the representation of the Model 7, compared with the data sets obtained in the execution of number 7, the one that generated the model itself, and of numbers 8 and 4, which generated the next best models in terms of NRMSE. From the results, we see that the behavior of the real executions does not vary substantially from one

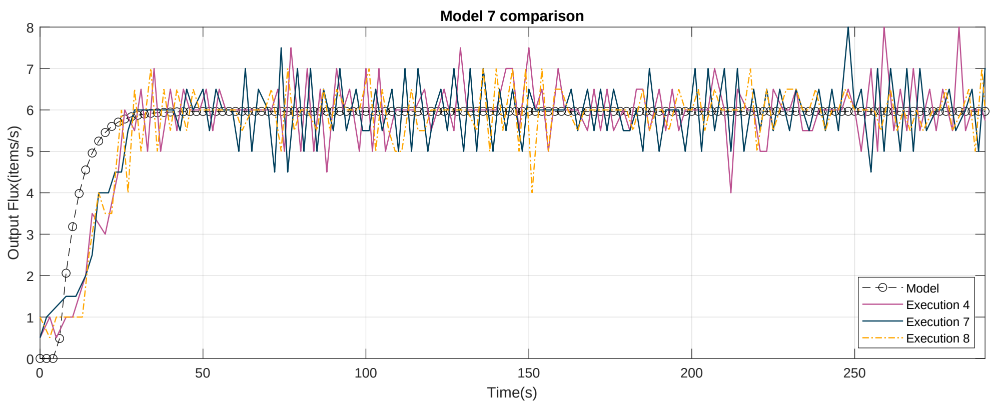


Figure 5.1: Model 7 analysis compared with real executions on Asperathos.

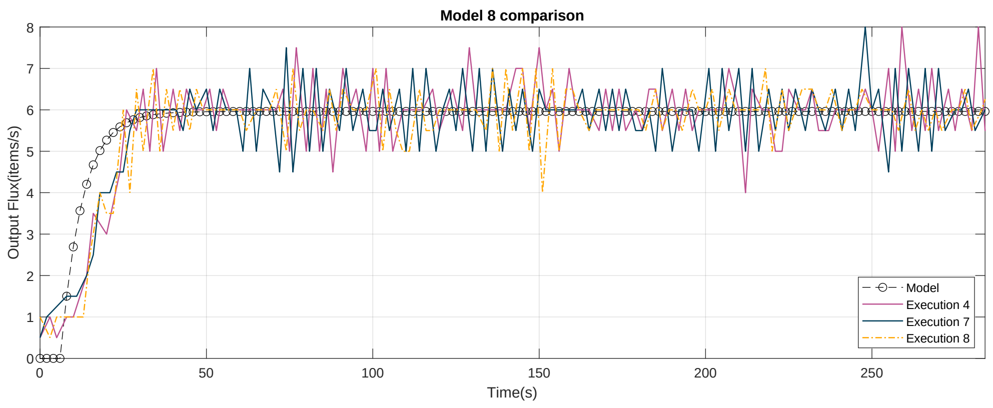


Figure 5.2: Model 8 analysis compared with real executions on Asperathos.

to another, and that they all follow the curve of the model in question similarly.

Next, we can say that there is not so much difference between the models generated by observing the behavior of Model 8, for example, the second smallest NRMSE. Figure 5.2 shows the generated model, comparing it with datasets from the same executions as in Figure 5.1. As expected due to the very close NRMSE values, little difference is observed between the models considered, which are similar both in system delay and in rise time. This reaffirms that step response measurements are faithful to the behavior of the system at different iterations. An important observation is that the figures do not seek to differentiate in detail each of the lines representing the executions, but to show the similarity of behavior between the three. The colors and lines used try to differentiate them minimally, but without any special emphasis so far.

Thus, we can say that, although all models have managed, in a steady-state, to reach the expected output for the given input, they do not accurately reflect the behavior of the real

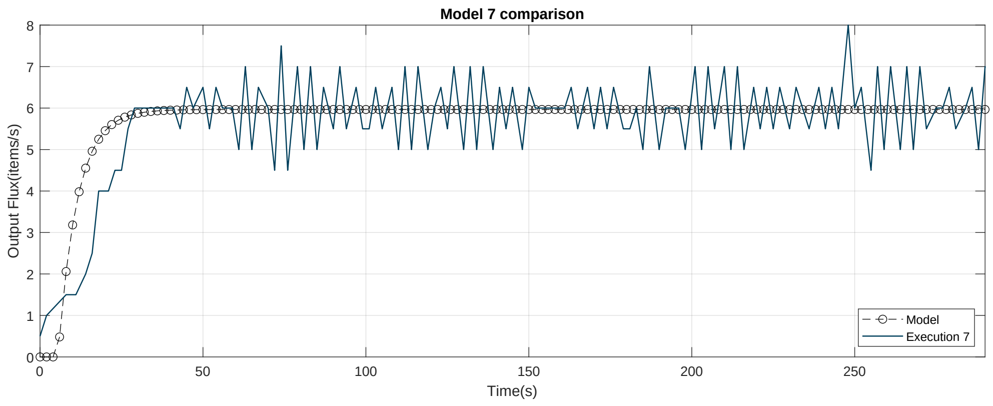


Figure 5.3: Model 7 highlight: analysis compared with the execution number 7 on Asperathos.

system, neither in amplitude nor in waveform. What the wave format of the executions show us, deviates from what is expected in FOPDT systems, where the result must be achieved exponentially. In the case of Asperathos, it is noticed that changes in the step generate almost immediate responses in the system, generating this up and down effect in the curve, even though it is already close to the behavior expected by the model.

Figure 5.3 highlights the Model 7, the lowest NRMSE, when compared to the real execution used for its generation. In this image, the nature of the waveform and the constant variations around the curve of the model becomes even clearer.

After these considerations, although we understand that the model does not have a high accuracy when compared to real executions, we realize that it is still possible to describe the system's behavior at some level through this representation. Therefore, we go ahead with Model 7, mostly because it has the lowest NRMSE, and from it, we analyze how the obtained transfer function reflects what was graphically observed. We generate the transfer function in (5.3), where 0.8523 is the process gain K, 5.901 is the time constant T, and 5.5 is the delay τ:

$$G(s) = \frac{0.8523}{5.901s + 1} e^{-5.5s} \tag{5.3}$$

With this transfer function, we can, for example, tune a PID controller in a more informed way, starting from a known reference point of the system, which will help us to provide guarantees that the real system response to a controller with certain gains will be acceptable according to the response observed in the model. More details on this analysis are described in Chapter 6.

Another important consideration is that, since Asperathos is designed to execute a variety of data processing workloads, we need to define the scope of the solution proposed here. For the Asperathos modeling just presented, we used an application that classifies and disaggregates energy data, which in our case was an implementation of the NIALM algorithm. Thus, we can say that other classification algorithms using neural networks, a popular approach to categorize data from such workloads, could also be used as the application use case to model Asperathos following the process described in this chapter. This is possible because such algorithms follow a similar processing approach even considering different workloads.

In this topic, the workload used here consisted of an homogeneous pool of tasks as described in Section 2.4. This means that, for a workload with different processing tasks, as long as such tasks are also homogeneous, which means that each of them takes about the same time to process, the modeling could be equally performed as described in this chapter. In this case, only if the time to process such tasks is different than the one for the workload used here, a change would be observed mainly in the system process gain.

Moreover, the process and considerations presented here to generate an FOPDT model, can be followed for an extensive assortment of systems that fit the criteria for this type of solution.

5.3 Using filters

Finally, considering the results obtained from the model presented here, a persistent oscillation in the output variable observed in each of the real executions of the system is noteworthy. Considering the use case application, these fluctuations can be the result of several factors, such as the processing time of an item, the time required to place and remove items from a queue, the time it takes to load the disaggregation libraries, among others. If we think that these factors can still be affected by disturbances, and that the observed oscillations are the result of noise, we can attenuate them by applying filters commonly used in control theory for this purpose.

In this context, a filter is a kind of algorithm used mainly to reduce noise in a given signal, such as the output variable of the control process. A more general way of defining a filter is as a compensator that corrects or deals in some way with some element related to the behavior of the system while seeking to avoid distortions in the signal. Another definition is given by Hellerstein et. al [80], where filters are described as system elements that precondition a signal before it is used, without significantly affecting its nature in the context of the controlled system.

The most common type of filter is the first-order, where the output tries to achieve the reference value exponentially over time. In this context, there are low-pass filters, which attenuate high frequencies (sudden changes), and pass on low frequencies (slow changes). This makes this type of filter ideal for reducing noise in a signal because such noise tends to be of higher frequency than expected changes in a process. Traditional variables used to measure the performance of a filter are, for example, the amount of distortion introduced and how quickly the filter transitions between passing a signal or blocking it. In this case, in the face of poor performance, the main shortcoming of low-pass filters is the instability they induce, especially motivated by possible delays that may occur in the system's response to variations in its input [57].

It is for this reason that, in general, metrics that respond more slowly to changes in the system are not normally indicated to represent output variables, for example. Therefore, it is desired that such a variable is already reasonably smoothed, and does not need to be filtered. Some authors [92] even say that it is often better to use a noisy signal than to smooth it with filters, as the benefits obtained from reducing noise do not compensate for a possible delay introduced.

For example, considering a PID controller, while the integral term has a tendency to smooth out noise, the derivative term amplifies it considerably by responding sharply to changes in the input. However, we do not want to base control actions on random noise, but on the general trend observed in the tracked error. So, if we want to use the term derivative in a noisy system, we need to smooth it out in some way. A widely used option is the use of filters. However, once again, it is important to note that a more aggressive smoothing will allow a higher derivative gain, and therefore a more aggressive control action, but will also introduce a longer delay in the system response, which can go against the whole initial purpose of using the derivative term.

Generally speaking, the consensus is that filters should be carefully evaluated before being properly inserted as part of the system, but if used responsibly, they bring real benefits when dealing with noise. Some examples of practical application are given by Janert et al. [92], one of the use cases presented being the waiting queue. To control this system, two controllers are used in cascade: *Controller 1*, which acts on a set of servers based on the rate of change in the size of the queue, and *Controller 2*, which outputs the desired rate of change, later used as a reference value by *Controller 1*. At a given moment of the experiment, when adding the derivative term to *Controller 1*, it was observed that the number of active servers was varying with a frequency considered high, introducing noise to the system. To reduce the number of control actions on the servers, the author suggests the insertion of a filter between the two proposed controllers, which would have the objective of smoothing the output of *Controller 1* to try to reduce the oscillations caused in *Controller 2*, which in turn uses this output as its reference value. After the modifications, it was observed that such a filter was actually able to stabilize the output of *Controller 1*, causing the number of active servers to fluctuate less, without affecting the queue size. In this way, a good result was obtained for the objectives in question.

Considering the concepts presented so far, we will present below a sample of what the expected behavior of Asperathos would be like when we insert a filter to reduce possible noise in the system's output variable.

5.3.1 Inserting a low-pass filter on Asperathos

When inserting a low-pass filter in Asperathos, in general, what we expect is that the resulting curve is possibly smoothed, being more similar to the generated model both in shape and in wave amplitude. In a simplistic way, this would indicate that such a model would better describe the system's behavior, and consequently, a controller tuning obtained from it would probably present more accurate results. Initially, to visualize the impact of this on the system, we use the `lowpass` function from Matlab itself to filter the output signal from Asperathos. Returning to the modeling described by (5.3), we will use Model 7 to compare it to the result of the executions after using the low-pass filter. Figure 5.4 shows such a model compared to the same runs of subsequent smaller NRMSEs detailed in Figure 5.1, but now with the signal filtered.

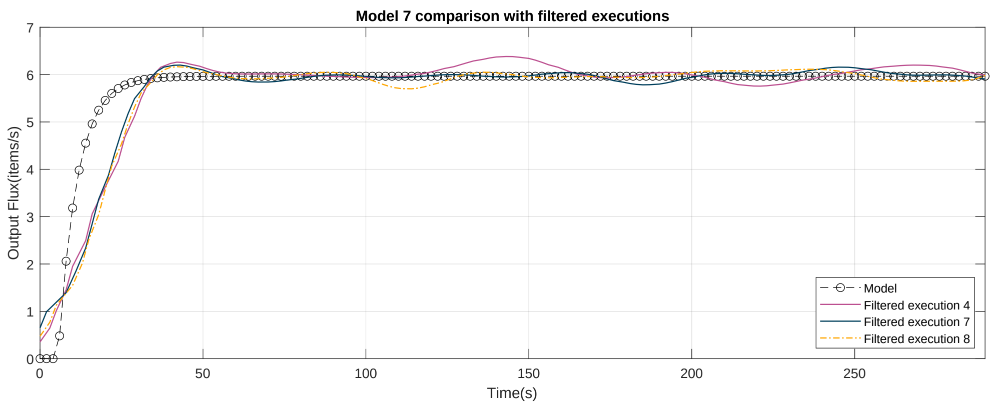


Figure 5.4: Model 7 analysis compared with filtered executions on Asperathos.

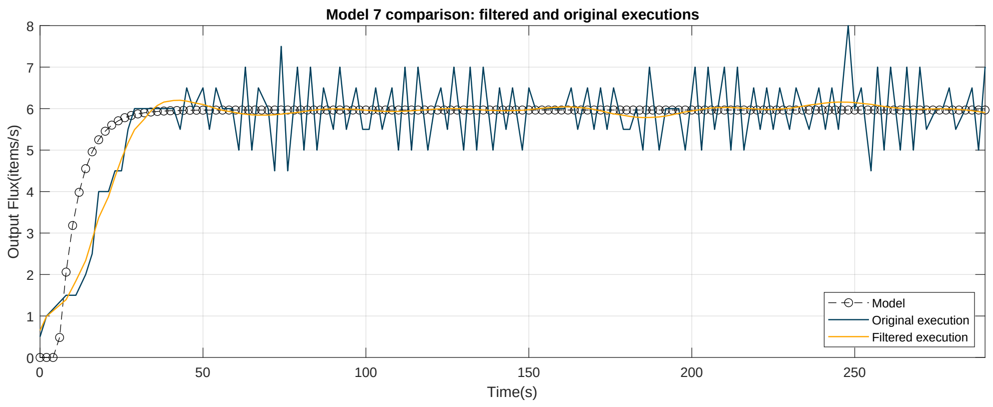


Figure 5.5: Chosen model highlighted: Model 7 analysis compared with its original and filtered execution on Asperathos.

In this case, we clearly see that the curves of the executions more significantly resemble the model represented by the circled black line. From the definition of low-pass filters, since the high frequencies are attenuated, and the low frequencies are passed on, we can infer that the Asperathos output values that were out of tune were disregarded, reducing the oscillations that were frequent before and with very accentuated characteristics.

Complementarily, in Figure 5.5, we isolate the execution number 7, both in its original form, represented by the blue line, and with the filtered signal, represented by the yellow line. While the original execution exhibits more oscillatory characteristics, the filtered execution is well-behaved, presenting itself close to the curve that describes the generated model. With this result, we have clear indications that this type of filter can be a good option to reduce noise in a signal like this, given that they tend to have a higher frequency than the expected changes according to the model.

Chapter 6

Adaptive control of DSP systems

6.1 Context and motivation

Software applications are subject to different operating conditions, such as variation in the availability rate of a service, changes in system objectives, among others. Consequently, new techniques to handle possible runtime changes, that do not result in downtime, are being developed. One of the most common approaches in the literature is software adaptation, which consists of adapting the software application itself at runtime and also throughout its development cycle, from requirements definition to design, construction, testing, deployment, maintenance and evolution.

Shevtsov et al. [147] presents a systematic review of studies that apply control theory in software adaptation, excluding physical resources or other infrastructure-level components. Among the main characteristics of the state of the art identified by the given work are: (i) the main motivation for the use of control theory in this context is the formal guarantees that can be obtained with its use; (ii) the main types of applications that use this type of solution are e-commerce and data processing; (iii) the most used models are linear, non-time-variant; (iv) PID and MPC (Model Predictive Control) are the most commonly used controller types in software adaptation; (v) PID is more used for regulatory control purposes and disturbance rejection in SISO-type systems, while MPC is more used to achieve optimal results in multipurpose systems; finally, (vi) robustness and cost are the most analyzed system properties, together with control characteristics such as stability and settling time.

In this context, control theory is one of the approaches considered to meet the demands in the design of software adaptation mechanisms [29; 80; 63; 174]. Some works that use control theory in adapting computing systems focus on controlling resources such as CPU and memory, among others at the infrastructure level [13; 51]. However, as already discussed in the motivation of this work, applying control theory in systems that seek to maintain QoS metrics through the allocation of resources such as servers, virtual machines and containers, can present particularities in the monitoring phase, in addition to the difficulty in modeling the system and apply control actions accurately.

Considering this, unlike software adaptation, we want to use control theory to adapt resources at the infrastructure level. When we add a cloud environment to this scenario, for example, we have the concept of elasticity, which allows the adjustment of computational resources at runtime to meet the demands of a given application. This helps prevent system performance from degrading, while reducing operating costs by reducing potential waste of resources.

However, providing efficient policies for resource allocation is a challenging task. Among the various techniques proposed in the literature, we range from simple rules that define a given action upon an event (*if-then*), to the use of complex machine learning algorithms. Here, we apply control theory when building feedback controllers, to implement a level of elasticity. Such controllers are designed to be stable, avoiding oscillations, quickly accommodating and responding appropriately to disturbances, while maintaining QoS metrics such as response time or throughput.

Ullah et al. [156] presents a detailed systematic review of works that apply control theory to provide elasticity in cloud environments. Among the topics covered, some conclusions regarding the types of metrics used as a reference to maintain QoS, the input and output variables considered, and the types of controllers most frequently used are interesting for our scope. For example, the most frequent control objectives were regulatory or optimal, both focused on improving the utilization of computing resources, maintaining an acceptable level of system performance, while reducing operational costs. From the point of view of the controllers developed for this purpose, the modeling approaches ranged from black box, queuing theory, among others, while the controllers used ranged from PID to MPC.

In this same context, the scope of the controllers developed here mainly includes stream processing applications. Roger et al. [143] lists relevant works in the area, approaching the union of elasticity, control theory, and DSP systems. One of the most important features to consider is how to parallelize the processing of a large stream of data. In this context, the use case presented here considers Asperathos as a container orchestrator, responsible for coordinating the parallelization of executions through replicas in a Kubernetes cluster, which, in turn, are responsible for processing the proposed tasks. In our case, such a task is the energy data disaggregation described in Section 2.4. More details about Asperathos can be found in Section 2.3.

Thus, in general lines, we initially propose a controller that acts on the resources orchestrated by Asperathos, seeking to maintain a certain quality of service directly related to performance metrics of an application that disaggregates a stream of energy data. The remainder of the chapter presents how the variables of interest for the proposed controller were selected, which control approaches were considered within the defined scope, some details on how such controller was implemented in Asperathos, in addition to an assessment of how the considered control approaches performed in relation to actual executions in the system.

6.2 Selecting control variables

In Section 4.1, important characteristics for the selection of metrics and variables of interest when implementing a controller are described. Complementary to this, Ullah et al. [156] lists the most used metrics and variables in the context of control theory applied to elasticity solutions and data processing. In the controller proposed here, we prioritize performance metrics. In this case, for the reference value to be tracked by this controller type, common metrics are response time and throughput [129; 59; 58; 108; 91; 141; 130]. As for the selection of the system input variable, given that the type of scaling considered here is horizontal, the most common input is the size of the cluster.

Based on this, we want to select the control variables to be used in the definition of the proposed performance controller. Here we consider the system as the Asperathos framework described in Section 2.3, processing a stream of energy disaggregation data, as described in Section 2.4. Thus, regarding the components of a feedback control loop: (i) the reference value to be maintained is the input or arrival rate of tasks continuously submitted to a Job in Asperathos; (ii) the system output is the throughput (number of completed tasks per unit of time); and (iii) the system input is the number of replicas in the Kubernetes cluster running the application.

Note that, for the reference value, we use the arrival rate of new tasks, which can change during a given execution. This means that, every time the arrival rate changes, the reference value also changes based on that. For this reason, considering the self-regulating type of control we try to achieve, our scope is limited to applications that do not constantly vary its arrival rate, since this would impose a high variation in the reference value, and therefore, increase the chances of overshoot every time it changes.

Moreover, to better understand the choice of the output variable in the definition of this controller, it is first necessary to understand the four main metrics related to the behavior of the type of system we want to control, represented here by Asperathos, which has a behavior similar to other DSP systems. They are: (i) queue size, (ii) duration of a task, (iii) number of Kubernetes replicas and (iv) system throughput.

Initially, it is important to think about what you want to control, that is, the possible metrics to be observed as the output of the system. Considering this, the queue size and duration of a task share a negative point: the case of the empty queue. A populated queue means that items added to it are accumulating, and the system is likely under-provisioned, meaning less computing power than necessary is being used to process such items. However, an empty queue indicates one of two possibilites: either the system is stable and no action is currently required, or it is over-provisioned, i.e. more computing power than necessary is being used. Consequently, in this over-provisioning situation, the tracked error would not be able to register a change in the workload when the input rate decreases, for example, leading to unnecessary increase in the processing cost. Similarly, the duration of a task is also not affected in this case, as even if the workload decreases, there will still be enough computing power to keep such tasks processing in the expected time.

On the other hand, using the number of Kubernetes replicas solves the problem of over-provisioning cases, since it would be directly observing the provisioned resources, and, knowing the average processing time of a task, it is possible to determine if it is necessary to add or remove resources from the system by calculating the tracked error. However, deriving the system throughput from the number of Kubernetes replicas being used creates a fixed variance of that rate depending basically just on the current number of replicas in the system. This would represent an impact on a possible analytic approach to the proposed controller design, more specifically on the system identification process described in Chapter 5, where we seek to determine its dynamics, that is, how the output behaves given a certain input, until its stabilization. In this case, the identification would be compromised as the output would present an instantaneous reaction to changes in the input. A consequence of this is that the dynamics of the process are not realistic enough, and the system would lose the benefits of using a controller. A possible solution to the problem of this instant response would be to introduce delays in the developed monitoring plugin. Even so, instead of having a system identification approach that results in a robust model, we would have to add a new parameter to the implementation, which would depend on specific infrastructures, platforms and applications.

This way, prioritizing the ideal conditions for system identification, its input rate and throughput must be observed at runtime. However, since the throughput calculation is limited by the input rate, in some situations, our controller cannot use only the real-time throughput to identify and act in over-provisioning situations, as exemplified above. Thus, it is necessary to combine two metrics: first, we use the input rate and throughput calculated at runtime to enable an accurate identification of the system dynamics; then we determine the cases where compensatory actions are needed to resolve observability issues that can lead to over-provisioning of the system, and then the throughput is calculated as an estimate based on the number of Kubernetes replicas being used.

Considering this, the tracked error of the proposed performance controller is calculated as in (6.1):

$$e_{perf} = throughput(t) - input_{rate}(t) \tag{6.1}$$

The system throughput, represented in the equation by $throughput$, is calculated based on (i) how many work items were completed in a time interval t, which we call the *runtime throughput*, or (ii) as an estimate based on the number of replicas in the system in a time interval t and the average processing time of a work item, which we call *estimated throughput*. For this, we assume the user is familiar with the mean service time of the considered application, and that this value is representative of the service times distribution. This calculation will be useful to mitigate over-provisioning situations. Finally, the system input rate, represented in the equation by $input_{rate}$, is based on the number of items added to the to-be processed queue in a time interval t.

6.3 Control approaches

In general, we can classify control approaches used in the literature as:

Classic. This family of controllers is the most common and relatively simple to implement. Fixed gain controllers are an example of a classic control. As the name suggests, these are characterized by gain parameters estimated offline, which remain fixed at runtime. This estimation can be done using trial and error methods, or through some kind of modeling like Ziegler-Nichols. An example of such a controller is the PID [25] one, widely used in this solution. Another example of classic control is the one that allows such gain parameters to be adapted at runtime, adjusting to changes in the environment. This is also the case for a special class of PID controllers, called self-tuning [71].

Advanced. Controllers of this type usually include solutions that combine different control methods into one. In this case, multiple controllers can be active at the same time, or they can be activated under predefined conditions. The first case comprises a range of cascaded controllers, usually with objectives that align in the same direction, while an example of the second type are the gain scheduling controllers.

Considering this, in the context of the solution proposed here, we initially want to evaluate the behavior of two types of classical controllers when applied to DSP systems: Fixed Action and PID controllers. Considering the control variables described above, we will now define how the proposed performance controllers were implemented.

6.3.1 Fixed Action control

This type of controller, often used in orchestrators, acts with a predefined response whenever the tracked error indicates the need for a control action. So, in our context, a Fixed Action controller scales the system up or down only in fixed steps of n replicas. Note that, in this case, we are not talking about gain parameters, but about the size of the corrective action itself. An application of this type of control can be commonly found in cloud resource auto scaling techniques, where some conditions are predefined, and a fixed action is applied. For example, a scaling rule for a service like Amazon EC2 Auto Scaling [4] might consist of monitoring the average CPU utilization of a cluster of virtual machines. If this average exceeds 50%, the service adds a new machine to the cluster. Note that, in this case, the corrective action is fixed at 1 extra virtual machine.

In contrast, PID controllers use a more sophisticated approach, applying proportional, integral, and derivative gains along with current, past, and estimated future tracked errors. To illustrate an example of how a Fixed Action controller may not be suitable in some scenarios, consider the case where the system has a sudden but temporary increase in its input rate. The controller would react for a while, but as the input rate normalizes again, the tracked error would no longer be able to identify that the queue has grown. A possible solution would be to monitor the queue, which again would introduce a customization that would make the system more complex, and dependent on specific information of the application and infrastructure considered. As for the PID controller, these momentary spikes would generate errors that would be accumulated by its integral term and, thus, naturally mitigated.

Furthermore, one of the clearest disadvantages of using a Fixed Action controller is that a fixed step that is too low can result in slow reactions to situations where a greater and faster corrective action is needed, or similarly, a step that is too high may imply a sudden reaction when a minor control action would suffice. Such sudden variations could also be responsible for unwanted oscillations in the system, as discussed earlier. In the case of the PID controller, the gains, when well tuned, offer an adequate reaction to the size of the tracked error.

Considering this, a Fixed Action controller was implemented and added to Asperathos as a plugin. Such implementation follows the definition described here, the size of the control action being previously configured by the user, and applied to the system each time the reference value is not being followed, that is, whenever any tracked error different than zero is registered.

6.3.2 Proportional-Integral control

For the Proportional-Integral control, finding appropriate values for its gains, what we call tuning here, can be a frustrating task: with two (for a PI controller) or even three (for a

PID controller) parameters, the number of possible combinations to be tested is quite large. Furthermore, it is often difficult to intuitively predict what effect the performance of a feedback loop will have after an increase or decrease in any of the parameters of that controller. Therefore, some sort of guide in this direction is highly desirable [92].

The tuning of a PID controller can be based on a good analytic model of the system, or from measurements of its dynamics obtained through experimental observations, as mentioned in Chapter 5. Among the tuning methods available in the literature, the Ziegler–Nichols rules are a classic set of heuristics that require little information about the system process. Going further, one can adjust a transfer function of a known model, such as the FOPDT, basing this model on the experimental results obtained from the dynamics of the system. From there, suitable values for controller gains can be set more precisely, consuming less time.

Considering this, one of the main goals of a successful tuning is to arrive at a stable system. Furthermore, it is important to note that control systems can be optimized considering different behaviors depending on specific situations. For example, systems that require faster responses are more susceptible to noise and oscillations, while systems that are slower can provide better accuracy and robustness when in steady-state. In this case, we can define important aspects to be considered regarding the performance of feedback control systems:

1. Is a persistent steady-state error acceptable? For systems in general, a persistent steady-state error is usually not acceptable, suggesting the need to use integral control. However, sometimes the system may require faster responses which are more important than eliminating such errors, as the use of integral control tends to slow down the system response.

2. How acceptable is the occurrence of oscillations? How quickly does the system return to normality? Again, oscillatory behavior is usually not acceptable, especially because of the overshoots that tend to happen and can, for example, violate QoS goals previously defined by users. However, for faster response, systems with oscillations can be useful to a certain extent.

3. How fast does the system have to respond to changes in input? The response time is generally determined by the time it takes for the system to reach two-thirds of its new steady state, assuming non-oscillating systems.

4. Should the system be robust to noise? Noise is a high-frequency disturbance. To lessen its influence, the system needs to be relatively slower. This normally precludes the use of derivative control.

All the standard tuning rules, like Ziegler–Nichols and other methods, work based on the choice of certain settings based on these questions, especially the accuracy and response time desired for the system. These choices tend to lead the system to acceptable performance for most applications, but it is important to note that some may require specific settings defined on a case-by-case basis.

That said, there are some general statements about the effect of changes in the gains of a PID controller that can be made. This can be useful for small manual adjustments of these parameters after results are systematically obtained through the formal methods outlined here. In general, increasing controller gains leads to faster response but also tends to make the system less stable. For the derivative term, however, the system tends to remain stable, given that the input signal is sufficiently noise-free, which is less common than it seems.

We can summarize the general rules for adjusting the gains of a PID controller as follows:

Increase in proportional gain k_p:

- Faster response;
- Lower system stability;
- Risk of overshoot;
- Noise increase;

Increase in integral gain k_i:

- Slower response;
- Lower system stability;
- Reduces noise;
- Eliminates steady-state errors faster

Table 6.1: PI Tuning Configuration

ID	r_t	s_t	$overshoot$	$k_{p(perf)}$	$k_{i(perf)}$
1	6.10	93.10	0.00	1.1776	0.0682
2	4.57	50.10	16.60	1.2833	0.1171
3	4.73	43.00	27.00	1.0922	0.1589
4	7.09	36.60	2.46	0.8355	0.1142
5	**7.42**	**60.00**	**0.00**	**0.9817**	**0.0871**

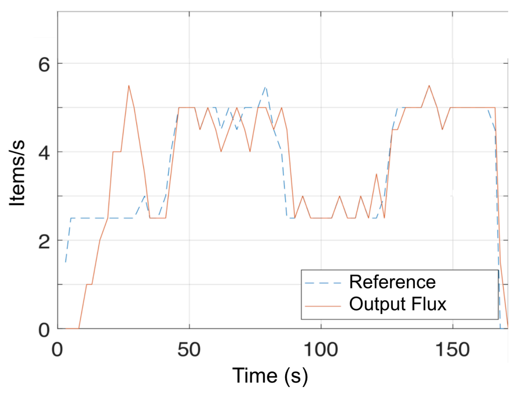
Increase in derivative gain k_d:

- Faster response;
- Improved system stability;
- Excessive noise increase;

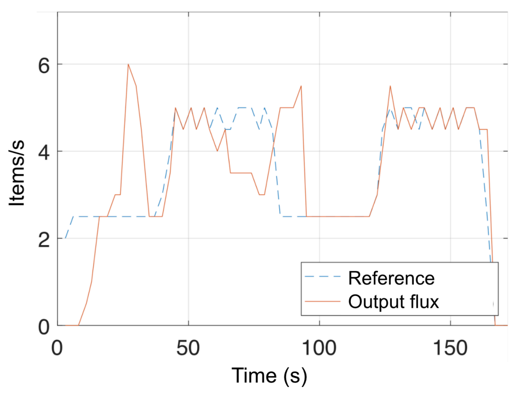
Considering this, we used the FOPDT model generated earlier, even with its limitations, to perform a more grounded tuning of the gains of the proposed PID controller. Using Matlab's own tuning methods, *Control System Toolbox* [118] and *PID Tuner* [119], we evaluated how each dynamic parameter of the system influenced different executions. Table 6.1 presents the achieved tuning settings, where r_t is the system rise time, s_t is the settling time, and $k_{p(perf)}$ and $k_{i(perf)}$ are the respective proportional and integral gains of the proposed performance controller. Our executions did not generate significant values for the derivative gain, so we simplified the approach, effectively designing a PI controller.

Finally, these gains were applied to real executions on our system. The combination of the overshoot, rise and settling time parameters is used to generate the respective gains $k_{p(perf)}$ and $k_{i(perf)}$. Considering this, Figure 6.1 shows the configurations of numbers 5 (6.1a) and 4 (6.1b) in Table 6.1, the first being the one that presented the best results with respect to overshoot, rise time and system accommodation.

We can see that, for the configuration 4, in Figure 6.1b, a higher value of integral gain ($k_{i(perf)} = 0.1142$) probably influenced the system to react more aggressively to accumulated errors, possibly caused by overshoots, not behaving as expected when tracking the reference value. Thus, according to Figure 6.1a, when using the configuration of number 5, with $k_{p(perf)} = 0.9817$ and $k_{i(perf)} = 0.0871$, the controller tracks the reference value with



(a) Tuning 5 on Table 6.1.



(b) Tuning 4 on Table 6.1.

.

Figure 6.1: Tracking of reference value with given PI tuning configurations.

reasonable rise and settling time, and virtually no overshoot. For this reason, this was the configuration chosen for the gains of the proposed PI controller.

Note that, despite the limitations of the model, we still got a reasonably adequate tuning as a reference. Thus, in the evaluation made below, we will use the gains described here as parameters of the proposed PI performance controller.

6.4 Evaluation

In this section we present the experiments performed to evaluate how a DSP system behaves when using different types of control strategies, including Proportional-Integral and Fixed Action control. In addition, we apply the proposed PI controller gain tuning described in Section 6.3.2 above, and compare this approach with a purely manual tuning. Finally, system throughput estimates were used to deal with possible over-provisioning conditions. In the use cases explored here, the effectiveness of the performance controller is evaluated in terms of user-defined QoS metrics, such as replica allocation, system response time and throughput.

6.4.1 Experimental design

In Chapters 5 and 6, several control approaches were presented that can be combined to form different compositions. Each of these approaches is associated with a set of factors, which are independent variables of the configuration of the component in question. This section initially presents basic concepts of an experimental design and the consequent results obtained by applying it to the evaluation of the solutions presented here.

An experimental design aims to define experiments in such a way that the most information is obtained with the least amount of experiments possible. Experiments, in this context, can be simulations or measurements in real (or close to real) environments. Besides that, experiments can be designed for different purposes, including: deciding between alternatives (comparative experiments), identifying which factors influence more than one response variable (selective experiments), adjusting/optimizing the experimental process, etc. In the context of this work the experimental design was carried out to mainly decide between alternatives.

To understand the experimental design carried out, it is necessary to introduce some important terms. The following are generally used in the design and analysis of experiments [90]:

Response variable. The response variable (or dependent variable) is the result of the experiment, that is, what you want to measure.

Factors. A factor is an independent variable (that can be controlled) that can take on different values and that affects the response variable. Through an analysis of experiments it is possible to quantify the effect of different factors on the response variable.

Replication. It is possible to repeat the same experiment n times. This repetition is called replication.

Interaction. Two factors interact if the effect verified for one factor on the response variable depends on the level of the other factor.

An adequate analysis of experiments makes it possible to quantify the effect of factors on the response variable, and when associated with a significance study, it makes it possible to identify which factors are statistically significant for the response variable.

Considering this, the three most commonly used types of experiment designs are [90]:

Simple design. It starts with a certain configuration for the parameters and varies one factor at a time to identify the effect that the factors have on the response variable. This method does not take into account the interactions between factors, which can lead to erroneous results when there are interactions.

Complete factorial design. Conduct experiments for each of the levels of all factors. This type of design of experiments is the most complete, however, when the number of factors and/or levels is very large, it becomes too expensive.

Fractional factorial design. This type of design is indicated when the number of experiments to be carried out with a complete factorial design is very large. Only a fraction of all possibilities are used here. The number of experiments is smaller, but it is not possible to study all possible interactions between the factors.

Considering the number of factors described later is this chapter, and the goals for the proposed experiments, we opted for a simple design. Besides that, there are some considerations that helped in deciding the technique to be used to evaluate the system. The key consideration is the life-cycle stage in which the system is. Measurements are only indicated if something similar to the proposed system already exists. Simulation and analytic modeling are mostly used for situations where measurement is not possible, but in general it is a better practice when the analytic modeling or simulation is based on previous measurement. Sometimes it is also helpful to use two or more techniques simultaneously or sequentially. For our case, measurements of a real system were used to generate an analytic model, which in turn, was used to suggest the appropriate configuration parameters for the proposed control system.

Besides that, to analyze the significance of the experiment results, a *t-test* analysis was performed. This analysis is a statistical test used to formally compare the means of two groups. This approach is often performed in hypothesis testing to determine whether a treatment actually has an effect on the population of interest, or whether two groups are different from one another. Note that a *t-test* can only be used when comparing the means of two groups. If you want to compare more than two groups, or if you want to do multiple pairwise comparisons, use an *ANOVA* test.

The *t-test* is a parametric test of difference, meaning that it makes the same assumptions about your data as other parametric tests. We checked the *t-test* assumptions to be true for our data, as follows:

- Are independent;
- Are *approximately* normally distributed;
- Have a similar amount of variance within each group being compared (homogeneity of variance).

It is important to note that, throughout the statistical analysis later presented in this chapter, we need to consider the effects of practical and statistical significance, since the presence of the latter does not necessarily mean that the results are practically significant in a real-world sense of importance.

The hypothesis testing procedure determines whether the considered sample results are likely to be representative if you assume the null hypothesis is correct for the population. If such results are sufficiently improbable under that assumption, then you can reject the null hypothesis and conclude that an effect exists, meaning your results are statistically significant. On the other hand, practical significance relates to the magnitude of the effect. That said, no statistical test can tell you whether the effect is large enough to have an importance in the context of the given study. To achieve that, you need to apply concepts and state of the art research on the subject of interest to determine whether the effect is big enough to be meaningful in the real world.

Now for the experiments described in this section, in summary, four main comparative scenarios were proposed:

I. Analytic PI - Runtime x Fixed Action - 3. For this use case, the *Analytic PI - Runtime* strategy considers the use of the PI controller, with its gain parameters tuned according to the analytic model generated in Chapter 5. The *Fixed Action - 3* treatment considers the use of what we call a Fixed Action controller, that increases or decreases the control action in 3 steps each time. For both cases, the throughput response variable is calculated at runtime.

II. Analytic PI - Runtime x Fixed Action - 1. In this scenario we use the same PI control described above. For the *Fixed Action - 1* treatment, we consider the use of the Fixed Action controller, this time increasing or decreasing the control action in 1 step each time. For both cases, the throughput response variable is calculated at runtime.

III. Analytic PI - Constant x Manual PI - Constant. In this case, the same PI controller is used as before, compared with a *Manual PI - Constant* treatment, in which the gain parameters of the given controller are manually tuned. For this configuration we consider the workload input rate to be a constant of 3 items per second. For both cases, the throughput response variable is calculated at runtime.

IV. Analytic PI - Runtime x Analytic PI - Estimated. Finally, for the last scenario we compare the Analytic PI controller with an instance of the same controller now calculating the throughput response variable as an estimate based on the average time to process an work item and the amount of resources used.

Considering these comparative scenarios, we define the response variables as: (i) the system throughput; (ii) resource utilization; and (iii) response time. For (i), the throughput is the rate at which requests can be serviced by the system, which in our case is the rate of processed items per second. The second metric, (ii), refers to the number of replicas instantiated in a Kubernetes cluster to process such work items. And (iii) is the interval between the start of a request submission and the end of the corresponding response from the system, which in our case is the time it takes for an item to be processed once it enters the system.

Next, we need to define the factors that may affect such variables. Table 6.2 showcase the factors that varied between different treatments. Some considerations about the input rate and the tracked error are as follows:

Input rate. This refers to the arrival rate of new work items. The input rate follows two type of workloads, one that sends 3 items per second over the entire duration of the stream, and other that varies between 2 and 4 work items, in a controlled uniformed manner. This means that, if we consider the duration of a stream in an experiment to be 20 minutes, every

Table 6.2: Factors for the experiments with PI and Fixed Action controllers.

Factor	**Description**	**Levels**	
Control Strategy	Type of controller used	Proportional-Integral	Fixed Action
Control tuning	Approach to configure control gains	Manual	Analytic
Control configuration	Specified control gains and configuration	P: 1.0; I: 1.0	P: 0.9817; I: 0.0871
		Actuation size: 3	Actuation size: 1
Input rate	Arrival rate of new processing items	3 items/s	2-4 items/s
Tracked error	Calculation of the tracked error	Runtime	Estimated

5 minutes the input rate is going to change from 2 to 4 or vice-versa. This is a limitation with regards to the types of workloads our control system can handle.

Tracked error. Refers to the difference between the input rate and the system throughput at a given moment. Two types of tracked errors are considered here. The first, called *runtime*, uses the throughput calculated based on how many work items were completed in a time interval t, measured at runtime. The other type is what we call *estimated*, which uses the throughput calculated as an estimate based on the number of replicas in the system in a time interval t and the average processing time of a work item. More details on the assumptions about this variable can be found on Section 6.2.

Other factors remained the same throughout the experiments, such as:

- *Maximum of concurrent replicas:* 8. This limitation is due to the size of the Kubernetes cluster used to perform the experiments, and the resources required by each execution of the application begin processed.
- *Estimated system response time:* 1.5 seconds. All work items sent to the system are of the same type, and considered to take the same time to process. This means our scope is limited to an homogeneous workload.
- *Stream duration:* 20 minutes. Long enough so we can variate the input rate every 5 minutes.

Table 6.3: Treatments for the experiments with PI and Fixed Action controllers.

Control Strategy	**Control tuning**	**Control configuration**	**Input rate**	**Tracked error**
Proportional-Integral	Manual	Proportional: 1; Integral: 1	3 items/s	Runtime
	Analytic	Proportional: 0.9817; Integral: 0.0871	3 items/s	Runtime
			2-4 items/s	Runtime
				Estimated
Fixed Action	Fixed	Actuation size: 3	2-4 items/s	Runtime
		Actuation size: 1	2-4 items/s	Runtime

- *Workload size:* 3600 items, considering the duration of the stream and the time to process an item.
- *Data collection:* Intervals of 2 seconds. Since each item takes approximately 1.5 seconds to process, collecting data every 2 seconds allows the system to react soon enough if any changes are necessary.

Instances of each proposed scenario were created and evaluated. Table 6.3 showcases each treatment and its given configuration values. Also, each treatment was replicated 15 times over the course of the experiments.

Finally, the execution environment to perform the experiments consisted of a Kubernetes cluster managed by the Asperathos framework. The workload used is a sample of real energy disaggregation data, provided by the LiteMe solution (a snippet of these items can be found in Section 2.4). Custom control and monitoring plugins were implemented to encapsulate the proposed controllers and monitors, allowing the execution of DSP applications, with a focus on performance metrics. Among the plugins used is *KubeJobs* for integration with Kubernetes, *StreamKubejobs* for monitoring stream processing applications, in addition to the *FixedAction* and *PI* controllers for a Fixed Action and Proportional-Integral control strategies, respectively. All were implemented in `python` and are available in the official Asperathos [6] repository.

Table 6.4: Configuration for the PI-Fixed set of experiments.

Control strategy	**Control tuning**	**Control configuration**	**Input rate**	**Tracked error**
Proportional-Integral	Analytic	Proportional: 0.9817 Integral: 0.0871	2-4 items/s	Runtime
Fixed Action	Fixed	Actuation size: 3		
		Actuation size: 1		

6.4.2 Scenarios I and II: Analytic PI x Fixed Action

The objective of this experiment is to evaluate if there is any difference when using a PI control over a more simplistic approach such as the Fixed Action one. Such evaluation takes into consideration the system throughput, resource utilization and response time. Table 6.4 highlights the combination of factors used for the experiments in these scenarios.

We can see that the Proportional-Integral controller was configured with the gains previously defined, $k_{i(perf)} = 0.0871$ and $k_{p(perf)} = 0.9817$. As for the Fixed Action controller, we first run a set of experiments with actuation size of 3, which represents a more incisive approach to the presence of a tracking error. This means that whenever there is any deviation detected by the tracked error, an action of size 3 is applied to the system, adding or removing 3 replicas from the Kubernetes cluster, depending on the error sign.

Next, we configure the Fixed Action controller with a less aggressive actuation size of 1. This is a default value, often used in controllers of this type precisely because of its more moderate nature. In this case, whenever there is any deviation detected by the tracked error, 1 replica is added or removed from the Kubernetes cluster.

In the next sections, we firstly present a descriptive analysis of the collected data and then an statistical analysis to support the discussion of the results.

A. Descriptive analysis

A descriptive analysis of the data for this scenario can be found on Appendix A, Section A.1.

B. Statistical analysis

In this section, we highlight the statistical analysis for the system response time variable. The complementary statistical analysis of the data for this scenario, including the tracked error and replica allocation, can be found on Appendix B, Section B.1.

Previously on this chapter, we defined that the response time is given by the time it takes for an item to be processed once it enters the system, which is considered to be approximately 1.5 seconds for each item. Here, we look at this metric from two perspectives, first we calculate the rate of items processed on time, i.e. items that took a maximum of 1.5 seconds to finish, and considering this same prerogative, the rate of requests that violate a given SLA of 1.5 seconds response time per item.

Table 6.5 showcases the observations and hypothesis about the rate of items processed on time for the PI x Fixed Action - 3 scenario. Complementary to that, Table 6.6 presents what was observed from the data regarding the SLA violation rate for this same use case.

Table 6.5: Rate of items processed on time statistical observations for the PI x Fixed Action - 3 scenario.

Observation from the data:	The average rate of items processed on time for the PI controller is greater than that for the Fixed Action controller of size 3.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average rate of items processed on time in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Fixed Action - 3 is not equal to 0.

Table 6.6: SLA violation rate statistical observations for the PI x Fixed Action - 3 scenario.

Observation from the data:	The average SLA violation rate for the PI controller is lesser than that for the Fixed Action controller of size 3.
Null hypothesis:	The average SLA violation rate in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. In general lines, considering a confidence interval of 95%, if the *p-value* is less than 0.05 then you can reject the null hypothesis and conclude that the difference between means in the two categories is statistically significant. Note that this same principle will be used for all the tests performed using the Student's distribution.

Considering the hypothesis in Table 6.5, the results show a *p-value* of $2.168e-05$, with a 95% confidence interval of $[10.75499, 22.67834]$. The sample estimate for the mean in group Analytic PI is 94.22222, while the mean for the Fixed Action - 3 group is 77.50556. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[-22.67834, -10.75499]$. The sample estimate for the mean in group Analytic PI is 5.777778, while the mean for the Fixed Action - 3 group is 22.494444.

From these results, we can then reject the null hypothesis for both cases, and say that the differences in means between the two groups are in fact statistically significant.

Now considering the configuration of an actuation size of 1 for the Fixed Action controller, Table 6.7 showcases the observations and hypothesis about the rate of items processed on time for the PI x Fixed Action - 1 scenario. Complementary to that, Table 6.8 presents what was observed from the data regarding the SLA violation rate for this same use case.

Table 6.7: Rate of items processed on time statistical observations for the PI x Fixed Action - 1 scenario.

Observation from the data:	The average rate of items processed on time for the PI controller is greater than that for the Fixed Action controller of size 1.
Null hypothesis:	The average rate of items processed on time in each group is the same.

Table 6.8: SLA violation rate statistical observations for the PI x Fixed Action - 1 scenario.

Observation from the data:	The average SLA violation rate for the PI controller is lesser than that for the Fixed Action controller of size 1.
Null hypothesis:	The average SLA violation rate in each group is the same.

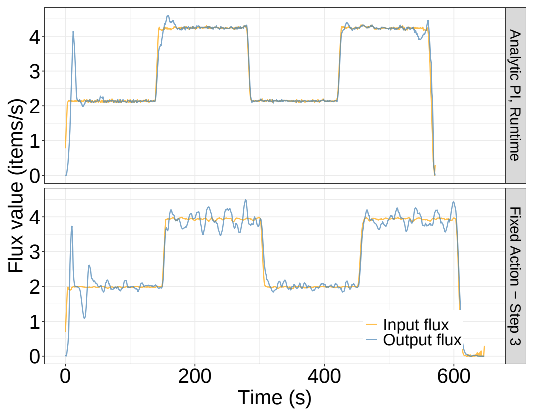
A Student's *t-test* was then performed to analyze if the differences observed from data are statistically significant for both of these cases. Considering the observations in Table 6.7, the results show a *p-value* of 0.6211, with a 95% confidence interval of $[-5.671753, 3.745827]$. The sample estimate for the mean in group Analytic PI is 94.22222, while the mean for the Fixed Action - 1 group is 95.18519. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[-3.745827, 5.671753]$. The sample estimate for the mean in group Analytic PI is 5.777778, while the mean for the Fixed Action - 1 group is 4.814815.

Considering these results, we can not reject the null hypothesis that the average rate of items processed on time and the average SLA violation rate in each group are the same. We conclude that because the *p-value* is high considering the expected standards of < 0.05, and the confidence interval includes a 0 difference, which means that there is still a possibility that the average means in the two observed groups are actually the same.

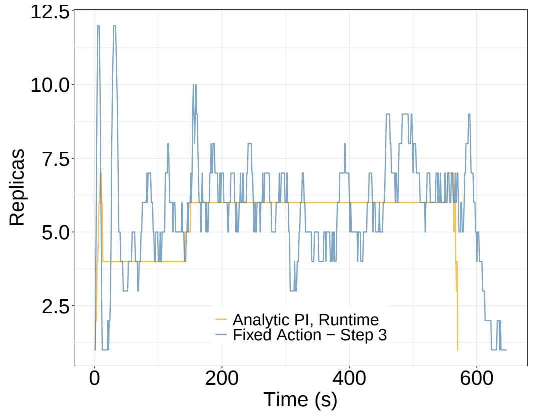
C. Discussion

Figure 6.2 shows the results of the Analytic PI x Fixed Action - 3 configuration. We can see that the tracking of the reference value in Figure 6.2a is apparently less favorable for the configuration that uses the Fixed Action controller with a step size of 3. However, the results of the analysis of the tracked error for this use case in B.1.2 are not statistically significant. This might be because the error values collected are so close together that the difference in means is very short and the amount of data was not big enough to provide statistically significant results. Another option is that the use of the mean is misleading when incorporating the values that standout. On the other hand, using the median to summarize the data from different replications would also not solve the possible issue since we are not interested in the value lying at the midpoint of this distribution, but on the values that are somewhat different than the expected and that might not be represented by the median value.

We can then look at the replica allocation to explain this difficulty in following the reference value showcased in Figure 6.2a. The large variation in the number of replicas being added and removed from the cluster, as expected by the size of the chosen step, is clearly demonstrated in Figure 6.2b. The statistical analysis in B.1.1 confirms that the differences for the replica allocation are in fact statistically significant, although not that significant from a practical perspective, with means of 5.41 for the Analytic PI, while the mean for the Fixed Action - 3 group is 5.68. However, in that case, we understand that low variation in the

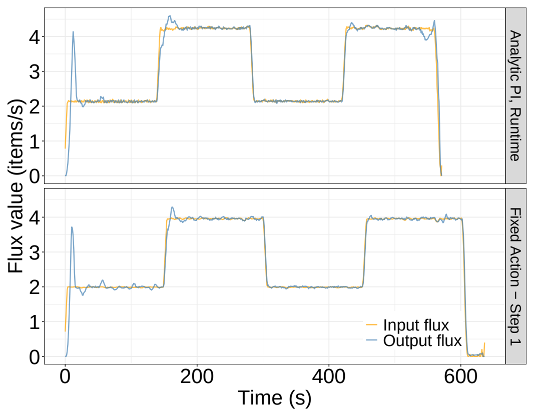


(a) Tracking of reference value.

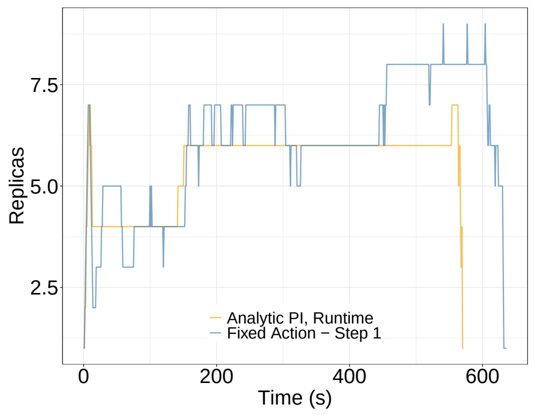


(b) Allocated replicas.

Figure 6.2: Analytic PI x Fixed Action - 3.



(a) Tracking of reference value.



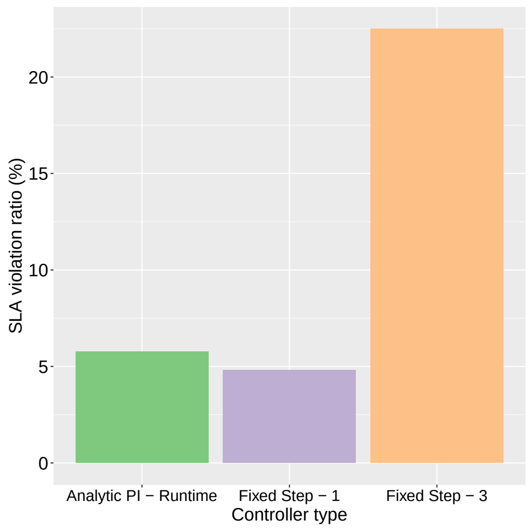
(b) Allocated replicas.

Figure 6.3: PI Control x Fixed Action - 1.

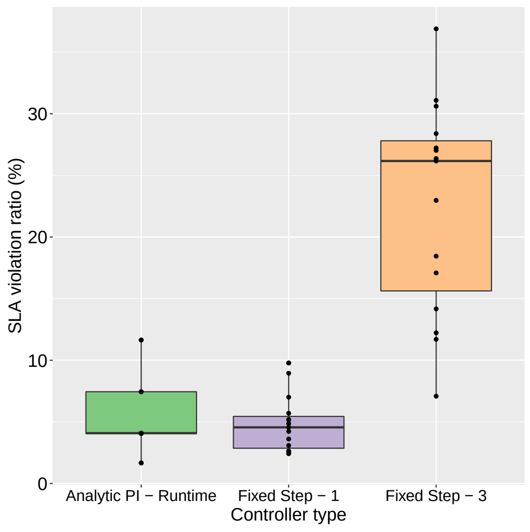
allocation is also of importance here, since availability can be compromised in the case of acquiring and removing replicas in such high frequency, as we see in Figure 6.2b for the Fixed Action - 3 approach. Besides that, the price of these operations can also escalate quickly [1].

Figure 6.3 shows the results of the Analytic PI x Fixed Action - 1 configuration. We can see in Figure 6.3a that the tracking of the reference value is closer to what is expected, given that the control action of size 1 is not as aggressive as the size 3. However, the above-desired

[1]For example, a sudden spike in the number of pods could trigger the scaling of node pools in managed clusters in cloud providers, such as AWS and Azure. Then, even if this would be quickly scaled down, there would still be cost and API availability impacts.



(a) Mean SLA violation ratio.



(b) Distribution of the SLA violation ratio.

Figure 6.4: Analytic PI x Fixed Action: SLA violation ratio.

variation in replica allocation is still visible in Figure 6.3b. One of the objectives that we seek to achieve is a small variation in the allocation of replicas according to the considered load, which for this experiment did not justify the results for the Fixed Action - 1 configuration.

Statistical analysis confirm that the differences in means between the Analytic PI x Fixed Action - 1 replica allocation are statistically significant, although not from a practical perspective, being observed from the data an average of 5.41 replicas for the Analytic PI approach, and 6.09 for the Fixed Action - 1. In this case, the moderately high variance in Table A.1 for the Fixed Action - 1 configuration also contributes to the conclusion that replica allocation is varying more than what is usually desired from an availability point of view, as for the Fixed Action - 3 scenario as well.

Finally, analyzing the response time, we take a look at the SLA violation ratio, that is derived from the response time values that are higher than 1.5 seconds. Figure 6.4 showcases the distribution of the violation ratio for the treatments considered here.

From Figure 6.4a we can see that the Fixed Action - 3 configuration is the one that has the highest average SLA violation ratio, while the Fixed Step - 1, is the lowest very close together with the Analytic PI one. Complementary to that, Figure 6.4b showcases the distribution of this data. The statistical analysis in 6.4.2 confirms that the differences between the Fixed Step - 3 and Analytic PI SLA violation ratios are in fact statistically significant. However,

Table 6.9: Configuration for the Analytic-Manual PI set of experiments.

Control strategy	Control tuning	Control configuration	Input rate	Tracked error
Proportional-Integral	Analytic	Proportional: 0.9817 Integral: 0.0871	3 items/s	Runtime
	Manual	Proportional: 1.0 Integral: 1.0		

for the Fixed Step - 1 treatment, we can not determine the same, possibly because the means are too similar and/or the data is not sufficient to validate that.

Considering all the discussion about system response time, throughput and replica allocation, we can say that the user will likely be paying more than necessary when using the Fixed Action - 3 approach, exceeding possible cost limits, for example. For the Fixed Action - 1 approach, replica allocation varies above the desired for the given input rate, although we can not confirm nor deny that the consequent violation ratio reflects that on the final performance of this controller. Also, in terms of total cost, when adding up the cost for the replicas and the violation fees that might occur, this analysis could be further extended to better indicate which approach performs better in that sense.

Thus, considering the comparative scenario described here, we conclude that the approach that uses the proposed Analytic PI controller showcases better results overall, as described by the analysis above, than the Fixed Action controller that acts on the system in fixed steps.

6.4.3 Scenario III: Analytic PI x Manual PI

The goal of this experiment is evaluate if there is any difference when using a PI control manually tuned over control gains obtained from an analytic model. Here, we named the first approach *Manual PI*, and the latter *Analytic PI*. Such evaluation takes into consideration the system throughput, resource utilization and response time. Table 6.9 showcases the combination of factors used for the experiments proposed for this scenario.

According to that, for both executions we used the proposed Proportional-Integral performance controller, using two tuning configurations, one generated based on the FOPDT model of the system and the other manually defined. Thus, for the Analytic PI, we consider the values for the proportional and integral gains to be $k_{i(perf)} = 0.0871$ and $k_{p(perf)} = 0.9817$. Then, for the Manual PI, considering the system dynamics to perform a disaggregation task, we choose the default values of $k_{i(perf)} = 1.0$ and $k_{p(perf)} = 1.0$.

This experiment is expected to show how a precise tuning of a PI controller can benefit system performance, as well as decrease the number of iterations needed to achieve good manual tuning, especially for less experienced users. In the next sections, we firstly present a descriptive analysis of the collected data and then an statistical analysis to support the discussion of the results.

A. Descriptive analysis

A descriptive analysis of the data for this scenario can be found on Appendix A, Section A.2.

B. Statistical analysis

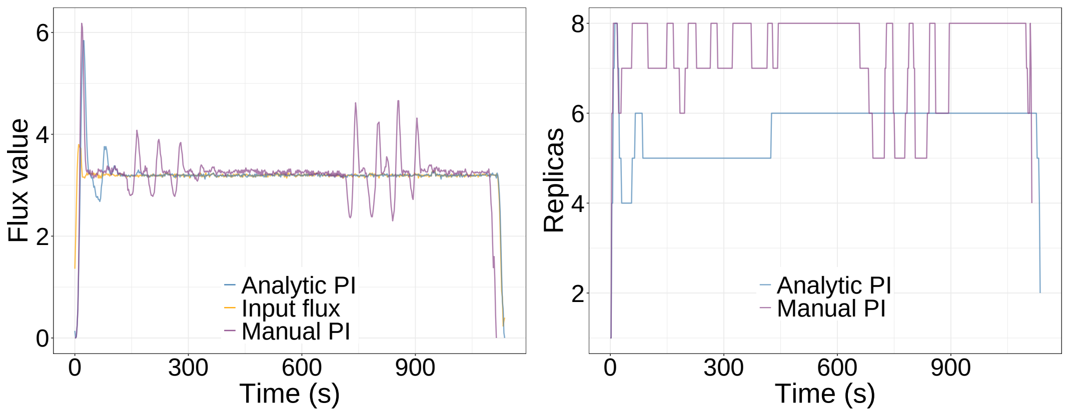
In this section, we highlight the statistical analysis for the replica allocation variable. The complementary statistical analysis of the data for this scenario, including the tracked error and system response time, can be found on Appendix B, Section B.2.

Table B.1 showcases the observations and hypothesis about the allocation of replicas for the Analytic PI x Manual PI scenario.

Table 6.10: Replica allocation statistical observations for the Analytic PI x Manual PI scenario.

Observation from the data:	The average replica allocation for the Analytic PI tuning is lesser than that for the Manual PI.
Null hypothesis:	The average replica allocation in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant. The results show a *p-value* $< 2.2e - 16$, with a 95% confidence interval of $[-1.829002, -1.644918]$. The sample estimate for the mean in group Analytic PI is 5.520328, while the mean for the Manual PI group is 7.257288, which is higher. Considering this, we can then reject the null hypothesis and say that the differences in means



(a) Tracking of reference value. (b) Allocated replicas.

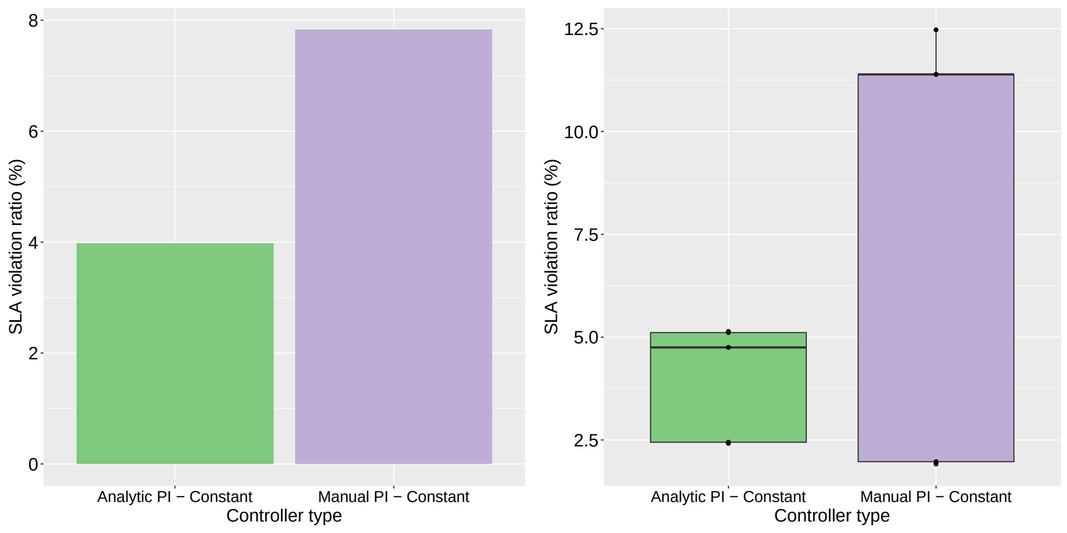
Figure 6.5: Analytic PI x Manual PI.

between the two groups are statistically significant. As for practical significance, we can say that the effect of this difference, especially when extrapolated for a higher workload, and consequentially a higher replica allocation, can be of importance when calculating the cost of the resources for the Manual PI approach, which has the higher mean.

C. Discussion

Figure 6.5 shows the results of the Analytic PI x Manual PI tuning configuration. As we can see in Figure 6.5a, the Manual PI approach, represented by the purple line, apparently had more difficulty when tracking the system reference value. This can happen because, due to the high variation in the observed replica allocation, more items are accumulating at the end of the queue, and when there is a sudden change and more replicas are instantiated, consequently more items are able to leave the queue and be processed. However, we do not want items to accumulate that much in the queue, possibly causing SLA issues if an item takes too long to finish. Even so, the results of the analysis of the tracked error for this use case in B.2.1 are not statistically significant. This can be possibly explained by the nature of the variable, as presented in Section B.1.2.

Moreover, this situation is reflected in Figure 6.5b, where we see that the number of allocated replicas varied more for the Manual PI than for the Analytic PI approach, causing some unwanted level of destabilization in the system. Statistical analysis confirm that the differences in means between the Analytic PI x Manual PI replica allocation are statistically



(a) Mean SLA violation ratio. (b) Distribution of the SLA violation ratio.

Figure 6.6: Analytic PI x Manual PI: SLA violation ratio.

significant, being observed from the data an average of 5.52 for the Analytic PI approach, and 7.25 for the Manual PI. In this case, the moderately high variance in Table A.2 for the Manual - PI configuration also contributes to the conclusion that replica allocation is varying more than what is usually desired, specially from an availability and cost perspective.

Finally, analyzing the response time, we take a look at the SLA violation ratio, that is derived from the response time values that are higher than 1.5 seconds. Figure 6.6 showcases the distribution of the violation ratio for the treatments considered here.

From Figure 6.6a we can see that the Manual PI configuration has an average SLA violation ratio higher than the Analytic PI one. Complementary to that, Figure 6.6b showcases the distribution of this data. Although the boxes overlap, the median line of the Manual PI approach lies outside of the Analytic PI box. This means that there is likely to be a statistically significant difference between these two values. This is confirmed by the statistical analysis in B.2.2.

Considering all the discussion about the response variables in this scenario, we can say that the user will likely be paying more than necessary when using the Manual PI approach, exceeding possible cost limits. Thus, we conclude that the approach that uses the proposed Analytic PI controller performs better, as described by the analysis above, than the Manual PI approach.

Table 6.11: Configuration for the Runtime-Estimated PI set of experiments.

Control strategy	Control tuning	Control configuration	Input rate	Tracked error
Proportional-Integral	Analytic	Proportional: 0.9817	2-4 items/s	Runtime
		Integral: 0.0871		Estimated

Nevertheless, it is important to emphasize that, although a gradual manual tuning can eventually achieve results as good as those obtained through analytic tuning, this task can be exhausting, and require several iterations. Also, less experienced users, who are likely to try more extreme values early on, may experience other problems caused by a faulty tuning. This is a problem because the overshoot observed in Figure 6.5, and caused by a more aggressive proportional gain, can completely destabilize the system. In this case, by overshoot we mean the spikes in replica allocation and the system throughput.

6.4.4 Scenario IV: PI - Runtime x PI - Estimated

The goal of this experiment is to evaluate if there is any difference when calculating the tracked error using the throughput measured at runtime, or as an estimate based on the number of replicas in use and the time it takes to process a given item. As previously described, the use of an estimated throughput tries to mitigate over-provisioning conditions that might happen otherwise. Here, we named one approach *PI - Runtime* and the other *PI - Estimated*. Their evaluation considers the system throughput, resource utilization and response time. Table 6.11 highlights the combination of factors used for the experiments proposed for this scenario.

To that end, we used the Proportional-Integral performance controller, using the tuning configuration provided by the FOPDT model of the system described in Chapter 5. We run a workload ranging from 2 to 4 items per second, this rate being the reference value that the system should track. As for the tracked error, two approaches are considered, one that uses the effective number of completed items in a given time interval, named *Runtime* in Table 6.11, and another that uses an estimate of completed items, called *Estimated*.

In the next sections, we firstly present a descriptive analusis of the collected data and then an statistical analysis to support the discussion of the results.

A. Descriptive analysis

A descriptive analysis of the data for this scenario can be found on Appendix A, Section A.3.

B. Statistical analysis

In this section, we highlight the statistical analysis for the tracked error variable. The complementary statistical analysis of the data for this scenario, including the replica allocation and system response time, can be found on Appendix B, Section B.3.

Table 6.12 showcases the observations and hypothesis about the tracking of the reference value for the PI - Runtime x PI - Estimated scenario. Remember that, for this system, the reference value is determined by the input rate of new work items, which for the treatments defined in Table 6.11 varies from 2 to 4 items per second. Thus, in order to quantify how well the system is following the reference value, we take into consideration the tracked error based on this input rate, which is equal to 0 when the system processes the same amount of incoming items, i.e. follows the reference value, and different than 0 otherwise.

Table 6.12: Tracked error statistical observations for the PI - Runtime x PI - Estimated scenario.

Observation from the data:	The average tracked error for the PI - Runtime approach is lesser than that for the PI - Estimated.
Null hypothesis:	The average tracked error in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for this case. The results show a *p-value* of 0.5112, with a 95% confidence interval of $[-0.03014138, 0.06050473]$. The sample estimate for the mean in group PI - Estimated is 0.016959064, while the mean for the PI - Runtime group is 0.001777389.

Considering these results, we can not reject the null hypothesis that the average tracked error in each group is the same. We mostly conclude that because the *p-value* is high considering the expected standards of < 0.05, and the confidence interval implies that there is still a possibility that the average means in the two observed groups are actually the same.

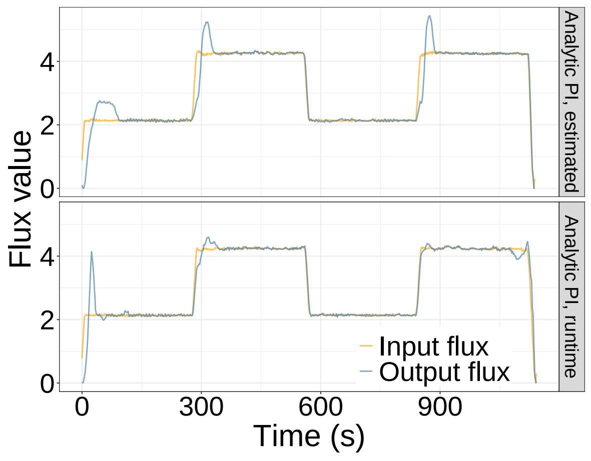


Figure 6.7: PI - Estimated x PI - Runtime - Tracking of reference value.

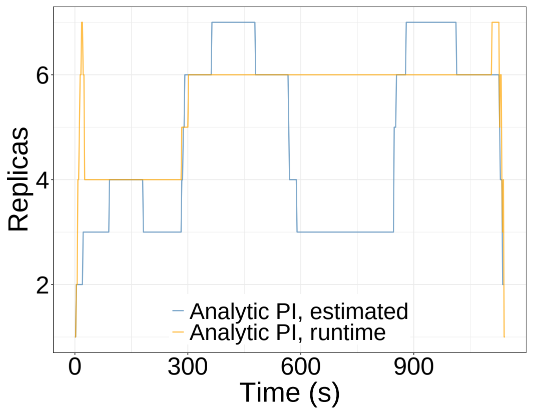
C. Discussion

Figure 6.7 shows the results of the Runtime x Estimated configuration. As we can see, the performance controller is able to track the reference values without any further difficulties, both for the Runtime and the Estimated approaches, that calculate throughput at runtime and as an estimate, respectively. However, we can see that for the Estimated configuration, small disturbances are detected when the input rate changes from 2 to 4 items per second.

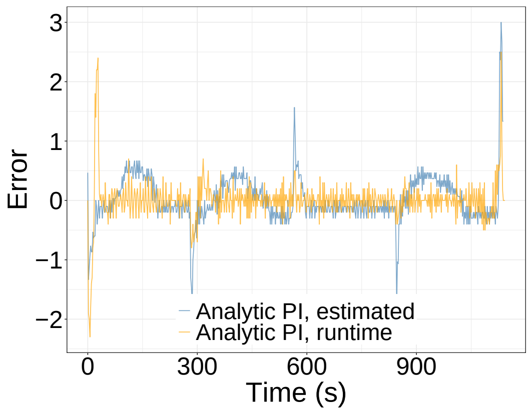
Additionally, Figure 6.8 depicts the behavior of the system in terms of resource usage of the considered Kubernetes cluster and the calculated tracked error for this scenario. In Figure 6.8a, we can see possible over-provisioning situations happening by looking at the yellow line on this graph, which represents the system configuration using the Runtime approach. When the input rate drops to 2 tasks per second, this approach keeps the number of replicas at 6, when the ideal value would be approximately 3, considering each item takes 1.5 seconds to finish. Therefore, we can define this as over-provisioning, since the system is operating at more than necessary utilization to maintain its QoS goals.

Statistical analysis confirm that the differences in means between the Runtime x Estimated approaches regarding the replica allocation are statistically significant, being observed from the data an average of 5.41 for the Runtime, and 4.76 for the Estimated configuration. In this case, replica allocation for the Estimated approach is, in average, lower than that of the Runtime approach.

Next, Figure 6.8b shows the calculated tracked error for both configurations. We see that



(a) Allocated replicas.



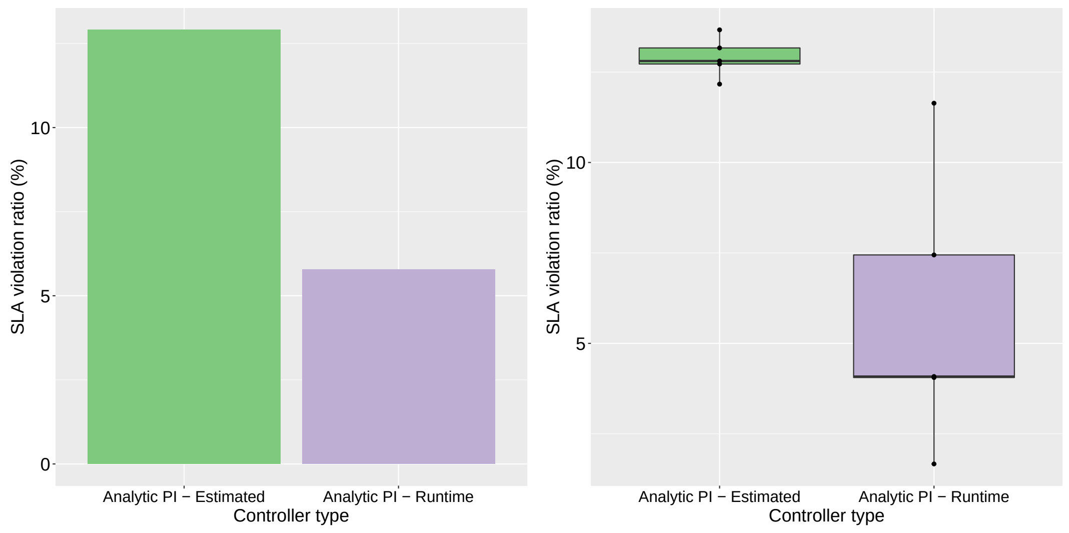
(b) Tracked error.

Figure 6.8: PI - Estimated x PI - Runtime.

the error for the Runtime approach, represented by the yellow line, does not register the need for a reduction in the number of replicas when the input rate decreases, that is, the error does not grow sufficiently to indicate a control action that would prevent over-provisioning of the resources. This happens because, as the system seeks to guarantee that as many tasks are executed as those arriving in a given time interval, there is a limitation on the number of tasks completed at runtime. In this case, even if the input rate decreases, for the system, the throughput remains in accordance with the tracked reference value, which, in theory, means that no control action should be performed even if the computational power used is larger than necessary.

In contrast, as shown by the blue lines in Figures 6.8a and 6.8b, the tracked error for the estimated approach is able to detect the over-provisioning, allowing the controller to generate actions to reduce the number of allocated replicas. This happens because knowing how long on average a task takes to complete, it is possible to estimate the processing capacity of the system given the number of replicas being used. In an over-provisioning situation, this means that the estimated throughput will be greater than the tracked reference value, indicating the need for a control action that decreases the number of Kubernetes replicas. Consequently, such behavior prevents the resource utilization costs from growing unnecessarily, and is therefore more suitable in these cases.

Finally, analyzing the response time, we take a look at the SLA violation ratio, that is derived from the response time values that are higher than 1.5 seconds. Figure 6.9 showcases



(a) Mean SLA violation ratio. (b) Distribution of the SLA violation ratio.

Figure 6.9: PI - Runtime x PI - Estimated: SLA violation ratio.

the distribution of the violation ratio for the treatments considered here.

From Figure 6.9a we can see that the PI - Estimated configuration has an average SLA violation ratio higher than the PI - Runtime one. Complementary to that, Figure 6.9b showcases the distribution of this data. The statistical analysis in B.3.2 confirms that the differences between the PI - Estimated and the PI - Runtime SLA violation ratios are in fact statistically significant. However, since we identified that there is a possible over-provisioning of resources for the PI - Runtime approach, this can explain why the violation ratio ends up smaller than the one that tries to mitigate this condition.

A further analysis on the impact on the total cost of using substantially more replicas but violating less SLA goals are needed to in fact determine which one performs better with regards to this metric.

Chapter 7

A multiple-objective control approach

7.1 Context and motivation

There is a demand for solutions suitable for computing systems that aim to meet a variety of requirements simultaneously, usually related to QoS objectives. Usually, developers resort to implementing custom control algorithms to ensure that the constraints imposed by the considered metrics are met. Such algorithms use user-defined heuristics and rules, and are often organized into multiple loops, that is, an inner loop that optimizes one metric, an outer loop that optimizes another metric, and so on.

However, this heuristic-based approach is generally not robust enough. First, defining the order in which each variable of interest is modified in a cycle, and the degree of change required, are essential tasks, but not necessarily easy to configure in an algorithm. Consequently, this approach requires the development of complex algorithms, which can end up introducing bugs in the system. Furthermore, while an execution takes place, there is a risk that situations not anticipated by the algorithm may occur, which can cause considerable deviations from the reference values that are sought to be reached [135; 160].

Alternatively, the use of control theory has proven successful in these situations [151]. In this case, it is possible to systematically quantify how important each of the multiple objectives considered is, and what is the effect of acting on each of the metrics related to them. However, most standard control techniques consider systems with only one input and one output, leaving out cases in which one wants to control multiple outputs in a coordinated manner. Simplifying this process with the use of several individual controllers, each responsible for a single output, can lead to situations where these controllers end up competing against each other, especially in the case of conflicting metrics of interest.

Nevertheless, control theory has evolved to define types of controllers that can handle multiple inputs and outputs. For example, MIMO controllers (Multiple-Input, Multiple-Output) are capable of acting on multiple inputs controlling multiple outputs, while SIMO controllers (Single-Input, Multiple-Output) act on a single input, with an effect observed on multiple outputs. The literature also describes solutions that use MISO controllers (Multiple-Input, Single-Output), capable of acting on more than one input, controlling a single output.

In this chapter we describe how to move forward with the application of control theory techniques in micro-batch DSP systems by considering multiple objectives of interest. Initially, we present a controller with a different objective from the performance PI controller already well defined in the context of this work, but which also works on top of a Kubernetes cluster. Next, we describe what kind of control approach was used to support the regulation of the set of relevant metrics established by the user. Finally, we perform an evaluation of how these controllers act in a coordinated manner on the system, extending the same use case of energy data disaggregation detailed in Section 2.4.

7.2 Definition of a controller for a cost variable

In the context of the systems of interest in this work, a latent concern is the cost associated with running the related applications. Processing large data sets requires greater computational power, which tends to be more expensive. The intelligent control and provisioning of these resources is, therefore, one of the most significant factors to avoid the natural negative impact that unwanted high expenses can bring to the users. It is important to ensure that sufficient resources are provisioned to meet the demand of running applications, but also to be aware of situations where over-provisioning may occur, unnecessarily increasing the execution cost.

Considering this demand, the cost controller proposed here was designed to keep the running cost per minute of a micro-batch DSP system at a certain user-defined level. The following sections describe the business model followed to determine the value of a Kubernetes replica being used, as well as the definition of the controller itself in its variables of interest and implementation details.

7.2.1 Business model

The on-demand instance business model is well known for its pay-per-use approach. Many cloud providers offer this instance type, which is commonly used to host various types of applications such as short-lived or irregular workloads. Considering this scaling model, the cost controller proposed here implements an adapted version of it. Each Kubernetes replica is charged per minute of usage, and its price varies according to the total usage of the cluster resources. We also assume that each replica is the same size, that is, it has the same maximum amount of allocated resources. Thus, the total usage of the cluster is calculated as the maximum number of replicas that can be instantiated in it.

Regarding the pricing chosen, there is a difference in value from certain usage limits, here defined by 30% and 90% of the maximum number of replicas allocated. The values are described in Table 7.1. To exemplify a feasible scenario, we consider that the maximum number of replicas of a cluster is 12. Thus, the limits are reached on 4 replicas (30%) and 11 replicas (90%).

The prices in Table 7.1 are based on the *Amazon AWS* [5] instance billing model, which charges a fixed price for one minute of usage. For reference, we have selected the price for a *c5.large* instance (2 vCPUs and $4GB$ RAM), described by AWS as ideal for compute-intensive applications, with a value equivalent to $\$0.13$ per hour. Assuming that each of the replicas allocates 1 vCPU and $2GB$ of memory, the base price of a replica is $\$0.13/2 = \$0.065$ per hour, and $\$0.00108$ per minute. From this, alternative replica price values were defined, used when cluster utilization exceeded the limits defined in Table 7.1.

7.2.2 Gain scheduling control

In certain systems, it may happen that a control cycle needs to be operated under a variety of conditions. Meeting different conditions may require that the behavior of the system vary at different points in time. This can be achieved by setting different values for the gains of the controller in question, and at runtime, selecting the most appropriate value according to the

Table 7.1: Business model for the cost controller.

Price of one replica per minute[*]		
Number of replicas	**Cluster utilization**	**Price per replica ($k_{p(cost)}$)**
0-3	Base	0.00108
4-10	30%	0.001404
11-12	90%	0.002052

[*]Each replica has 1 vCPU and 2GB of memory.

current system conditions. Such a process is known as gain scheduling.

There are several possible signs that indicate the set of values to be used as gains. Commonly, the system's own input or output is used, but the signal can also be something completely external. Some possibilities include using the system input to select different execution modes when the workload is high or low, using the current time of day to prepare the system for the famous "rush hour", or even using the magnitude of the tracked error to make the controller actions more aggressive when the error grows exaggerated, for example, when controlling a queue of tasks to be executed. For all these cases, instead of a single set of gains, there are different sets triggered by certain conditions. It is important to note that, at any given time, exactly one of these sets is active providing gains to the controller.

Considering this, the cost controller implementation makes use of a gain scheduling approach to provide a control based on different gains depending on the state of the system. In this case, considering the business model described here, the definition of the gains is based on the usage limits established for the cluster. This knowledge is incorporated in the tuning of the proportional term of this controller, the gains being equal to the prices in Table 7.1, varying as each usage limit is reached by the system. Thus, the proportional gain $k_{p(cost)}$ is initially equal to 0.00108, then when the cluster reaches 30% of utilization, the gain is equal to 0.001404, and at 90% it is equal to 0.002052. It is important to note that the gain scheduling algorithm implemented here is a method already used for this type of solution [75; 175].

In this way, the corrective action for that controller can be defined as in (7.1):

$$u_{cost} = k_{p(cost)} \times e_{cost} \tag{7.1}$$

In the above equation, $k_{p(cost)}$ is the proportional gain according to the cluster utilization values defined in Table 7.1, and e_{cost} is the tracked error for the cost controller.

To calculate e_{cost}, it is first necessary to calculate the processing $cost$ at a given point in time. In our deployment, a third-party application is responsible for monitoring cluster usage, and then assigning a certain price to each replica based on that usage, according to Table 7.1. This value is then sent to the plugin specific to that controller added to the Asperathos Monitor module, which encapsulates the remaining cost calculation logic, defined by (7.2):

$$cost = cost_{replica}(t) \times replicas_{up}(t) \tag{7.2}$$

The price of each replica at minute t is represented by $cost_{replica}(t)$, and $replicas_{up}(t)$ represents the number of replicas being used in the same minute t.

Finally, the proposed controller was designed to keep the execution cost per minute of a micro-batch DSP system at a certain user-defined level. Considering this, the rest of the system variables involved are: (i) the system input, which is the number of Kubernetes replicas managed by Asperathos; (ii) the system output, which is the current cost of the replicas processing the application; and (iii) the reference value to be followed, which is the user-defined cost per minute. Thus, the previously calculated $cost$ in (7.2) is used to calculate the tracked error, which, in turn, defines how far the system is from the reference. Equation (7.3) describes this variable:

$$e_{cost} = cost(t) - cost_{ref} \tag{7.3}$$

Where $cost(t)$ represents the compute cost at minute t and $cost_{ref}$ is the user-defined reference value that the controller seeks to maintain.

7.3 Definition of a multiple objective controller

After understanding how the cost controller was implemented, in order to simultaneously regulate cost and QoS metrics in a micro-batch DSP system, we propose the use of a SIMO PI

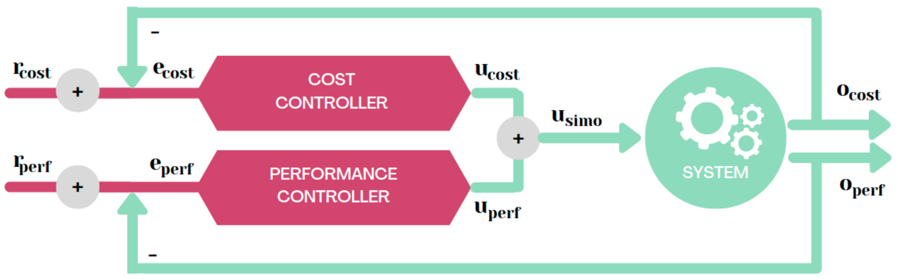


Figure 7.1: Architectural model of the proposed SIMO PI controller.

controller that acts on a single input and influences multiple outputs. Such controller makes use of specific control theory approaches for the cases where there are multiple objectives of interest, and, inserted in the context of orchestration of containerized applications, they help to define actions related to the automated provisioning of these resources.

Considering the problem described, initially it is necessary to define two independent controllers, one focused on tracking performance metrics, and the other focused on controlling the cost metrics of the resources used. For experimentation purposes, the first is represented by the performance controller described earlier, while the second is the cost controller just presented. Then, the modeling associated with regular SIMO controllers is applied on the considered system in order to combine the resulting actions of each one of them in a single corrective action on the input [128; 31].

To better illustrate the solution, Figure 7.1 highlights the main components of this approach, which are the Cost Controller, the Performance Controller, and the System itself. This architectural model aims to make the System able to track two reference values, the cost r_{cost}, and the performance r_{perf}. These two values are used to calculate, respectively, the tracked error for the cost metric e_{cost} and the tracked error for the performance metric e_{perf}, which by definition are calculated as the difference between the considered references and the observed outputs.

Furthermore, we can see that a single control action u is applied to the System, which is composed of the combination of the corrective actions u_{cost}, resulting from the Cost Controller, and u_{perf}, resulting from the Performance Controller, as defined by (7.4):

$$u = u_{cost} + u_{perf} \tag{7.4}$$

Where the control actions u_{cost} and u_{perf} are defined by the control strategies considered for each of them, in this case, a PID approach, as we see next in (7.5):

$$\begin{cases} u_{cost} = k_{p(cost)}e_{(cost)} + k_{d(cost)}\dot{e_{(cost)}} + k_{i(cost)}\int e_{(cost)}dt \\ u_{perf} = k_{p(perf)}e_{(perf)} + k_{d(perf)}\dot{e_{(perf)}} + k_{i(perf)}\int e_{(perf)}dt \end{cases} \tag{7.5}$$

It is important to notice that both the considered control actions, u_{cost} and u_{perf}, have their own proportional, integral and derivative gains. As discussed earlier, after the tuning based on the model generated for the system, our controllers use only proportional and integral gains, characterizing a PI controller. Besides that, compensation operations are introduced to smooth out possible side effects arising from the combination of the control actions, defined here as u_{cost} for the cost controller, and u_{perf} for the performance controller. Next, a utility function is defined aiming to prioritize the impact of such corrective actions, as described in (7.6):

$$u_{simo} = \alpha u_{perf} + (1 - \alpha)u_{cost} \tag{7.6}$$

Here, α represents the user's preference for cost and performance. For example, $\alpha = 0.8$ represents a high preference for performance and quality of service metrics, $\alpha = 0.5$ does not have a specific preference for any of the options, and $\alpha = 0.2$ represents a high preference for cost metrics.

Finally, considering the same energy data disaggregation use case described in Section 2.4, the logic defined in (7.6) has been encapsulated in a new SIMO controller which was implemented as a plugin in Asperathos. Internally, what actually happens is that, in this case, the Asperathos controller is based on both controllers defined here as Cost Controller and Performance Controller, and acts on the system according to the preferences defined by the user.

7.4 Evaluation

In this section we present the experiments performed to evaluate how a DSP system behaves when using different types of control strategies to handle scenarios where multiple-objectives are defined. Considering this, a SIMO controller is evaluated with a series of configurations that seek to show how different cost and QoS priorities can influence the behavior of the system. We also evaluated alternative approaches such as the use of independent controllers acting on the same system. Finally, details regarding the financial impacts of these control techniques are represented in terms of SLA violation. In the use cases explored here, the effectiveness of the controllers is evaluated in terms of user-defined QoS metrics, such as replica allocation, system response time and throughput.

7.4.1 Experimental design

In this chapter, two control approaches were presented that can be combined to form different compositions. Each of these approaches is associated with a set of factors, which are independent variables of the configuration of the component in question. Considering this, this section defines the experimental design proposed for this set of experiments and the consequent results obtained by applying it to the evaluation of the solutions presented here.

Considering the number of factors described later is this chapter, and the goals for the proposed experiments, we opted for a simple design. For our case, measurements of a real system were used to evaluate the control approaches. Besides that, to analyze the significance of the experiment results, a *t-test* analysis was performed. More details on the concepts for the experimental design used here can be found on Section 6.4.1.

Now for the experiments described in this section, as a continuation of the comparative scenarios previously described and evaluated in this work, one main scenario is proposed:

V. SIMO PI x Independent controllers. For this use case, the *SIMO PI* controller considers the use of the PI performance controller defined in Chapter 6, combined with the Cost controller defined in this chapter. This happens by implementing a Single-Input Multiple-Output (SIMO) control approach that encapsulates the logic of combining control actions and applying a single one over a given system. In our case, we also define different levels of preference for each controller in the combination of actions. On the other hand, the *Independent* approach considers the performance and cost controllers, but they act independently over the same system, with the performance one being the most frequent.

Considering this scenario, we define the response variables as: (i) the system throughput;

Table 7.2: Factors for the experiments with SIMO and independent controllers.

Factor	**Description**	**Levels**		
Control Strategy	Type of controller used	SIMO	Independent	
Control tuning	Approach to configure control gains	Analytic	Gain scheduling	
Control configuration	Specified control gains and configuration	P: 0.9817; I: 0.0871	Price per replica	
Controller preference	Preference for each controller when acting over the system	0.2	0.5	0.8

(ii) resource utilization; (iii) response time and (iv) the cost of an execution. For (i), the throughput is the rate of processed items per second. The second metric, (ii), refers to the number of replicas instantiated in a Kubernetes cluster to process such work items. Besides that (iii) is the time it takes for an item to be processed once it enters the system. And (iv) refers to the cost of an execution considering how much a Kubernetes replica would cost on an Amazon AWS infrastructure.

Next, we need to define the factors that may affect such variables. Table 7.2 showcase the factors that varied between different treatments. Some considerations about the control configuration and the controller preference are as follows:

Control configuration. This refers to the configuration gains of each controller. For the PI performance controller, the same gains defined in Section 6.3.2 and shown on Table 7.2 are used. For the Cost controller, the gains are defined by the price per replica defined on Table 7.1.

Controller preference. Our evaluation consider user preferences regarding control actions, which are defined as $\alpha = 0.2$ when there is a preference for cost metrics, $\alpha = 0.5$ for no preference in particular, and $\alpha = 0.8$ favoring performance metrics.

Other factors remained the same throughout the experiments, such as:

- *Maximum of concurrent replicas:* 12. This limitation is due to the size of the Kubernetes cluster used to perform the experiments, and the resources required by each execution of the application begin processed.

- *Estimated system response time:* 1.5 seconds. All work items sent to the system are of the same type, and considered to take the same time to process. This means our scope is limited to an homogeneous workload.
- *Workload size:* 3600 items.
- *Stream duration:* 20 minutes.
- *Data collection:* Intervals of 2 seconds.
- *Input rate:* Ranging from 2 to 9 tasks per second.
- *Reference value:* For the Performance PI controller, the reference value is the considered input rate of new items; for the Cost controller, it is a given desired cost defined by the user.

Instances of each proposed scenario were created and evaluated. Table 7.3 showcases each treatment and its given configuration values. The table was reduced with focus on the controller preferences for each control strategy. Also, each treatment was replicated 15 times over the course of the experiments.

For the experiments with the SIMO control approach and combined actions, we assume that the application in question is runtime sensitive and respects cost constraints. We also assume that the execution environment starts with the optimal number of replicas to satisfy both performance and cost constraints, and that replicas processing tasks always use the same amount of resources. To show how controllers react to various changes in workload, different input rate levels are modeled at runtime. Considering this, the peak in the input rate must be large enough to force scaling in, i.e. more replicas to be added, but not so large that it exceeds the cluster limits in such a way as to cause actuator saturation. Furthermore, we want to test executions using all price limits defined in Table 7.1.

Finally, the execution environment to perform the experiments consisted of a Kubernetes cluster managed by the Asperathos framework. The workload used is a sample of real energy disaggregation data, provided by the LiteMe solution (a snippet of these items can be found in Section 2.4). Custom control and monitoring plugins were implemented to encapsulate the proposed controllers and monitors, allowing the execution of DSP applications, with a

Table 7.3: Treatments for the experiments with SIMO and Independent controllers.

Control Strategy	**Controller preference**
SIMO	0.2
	0.5
	0.8
Independent	No preference

focus on performance metrics. Among the plugins used is *KubeJobs* for integration with Kubernetes, *StreamKubejobs* for monitoring stream processing applications, in addition to the *PI*, *Cost* and *SIMO* controllers for the proposed control strategies. All were implemented in `python` and are available in the official Asperathos [6] repository. Also, the same application used in the evaluation described in Section 6.4 is used here.

7.4.2 Scenario V: SIMO PI x Independent control

The objective of this set of experiments is to evaluate if there is any difference when using a SIMO PI control over a more simplistic approach such as the Independent one. The first one combines the individual actions of the cost and performance controllers into a general control action, while the second one presents independent controllers acting in different moments, with the performance one being the most frequent. Such evaluation takes into consideration the system throughput, resource utilization, response time and total execution cost.

In the next sections, we firstly present a descriptive analysis of the collected data and then an statistical analysis to support the discussion of the results.

A. Descriptive analysis

A descriptive analysis of the data for this scenario can be found on Appendix A, Section A.4.

B. Statistical analysis

In this section, we highlight the statistical analysis for the total execution cost variable. The complementary statistical analysis of the data for this scenario, including the tracked error, replica allocation and system response time, can be found on Appendix B, Section B.4.

Considering this, the total execution cost is given by a calculation of the amount of time a given replica was used, and how much it costs per minute. Replica prices follow the defined on Table 7.1. Table 7.4 showcases the observations and hypothesis about the total execution cost for the SIMO PI with $\alpha = 0.2$ X Independent scenario.

Table 7.4: Total execution cost statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average total execution cost for the SIMO PI controller with $\alpha = 0.2$ is lesser than that for the Independent configuration.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average total execution cost in each group is the same.
Alternative hypothesis:	The true difference in means between group SIMO PI (0.2) and group Independent is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. For the SIMO PI (0.2) configuration, the results show a *p-value* of $6.79e-07$, with a 95% confidence interval of $[-2.090660, -1.304838]$. The sample estimate for the mean in group SIMO PI (0.2) is 14.22201, while the mean for the Independent group is 15.91976. Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

Similarly to the previous analysis, Tables 7.5 and 7.6 showcase the observations and hypothesis about the tracked error for the SIMO PI with $\alpha = 0.5$ and $\alpha = 0.8$.

Table 7.5: Total execution cost statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average total execution cost for the SIMO PI controller with $\alpha = 0.5$ is lesser than that for the Independent configuration.
Null hypothesis:	The average total execution cost in each group is the same.

Table 7.6: Total execution cost statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average total execution cost for the SIMO PI controller with $\alpha = 0.8$ is lesser than that for the Independent configuration.
Null hypothesis:	The average total execution cost in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. For the SIMO PI (0.5) configuration, the results show a *p-value* of 0.00403, with a 95% confidence interval of $[-1.0146574, -0.2594618]$. The sample estimate for the mean in group SIMO PI (0.5) is 15.28270, while the mean for the Independent group is 15.91976. Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

For the SIMO PI (0.8) configuration, the results show a *p-value* of 0.06767, with a 95% confidence interval of $[-0.75022767, 0.03071007]$. The sample estimate for the mean in group SIMO PI (0.8) is 15.56000, while the mean for the Independent group is 15.91976. Considering these results, we can not reject the null hypothesis that the average total execution cost in each group is the same.

Finally, in summary, Table 7.7 showcases the averages of the metrics observed for each treatment of the SIMO PI x Independent configuration.

Table 7.7: Averages of the metrics analyzed for each control preference for the SIMO PI x Independent scenario.

	$\alpha = 0.2$	$\alpha = 0.5$	$\alpha = 0.8$	Independent
Replica allocation	7.244643	7.436266	7.460573	7.397482
Tracked error (Performance)	-0.04404762	-0.008976661	0.03957587	0.05590528
Tracked error (Cost)	0.003079901	0.003951514	0.004196507	0.004281179
Response time	40.33860	71.88246	82.92333	50.86568
SLA violation	59.66140	28.11754	17.07667	49.13432
Total execution cost	14.22201	15.28270	15.56000	15.91976

Complementary to that, Table 7.8 showcases the *p-values* resulting from the statistical tests performed on the data of each treatment for the SIMO PI x Independent configuration.

Table 7.8: P-values of each control preference for the SIMO PI x Independent scenario.

	$\alpha = 0.2$	$\alpha = 0.5$	$\alpha = 0.8$
Replica allocation	0.3613	0.8363	0.7454
Tracked error (Performance)	0.2885	0.3471	0.8002
Tracked error (Cost)	0.001019	0.4452	0.8484
Response time	7.303e-08	1.006e-12	1.39e-14
SLA violation	7.303e-08	1.006e-12	1.39e-14
Total execution cost	6.79e-07	0.00403	0.06767

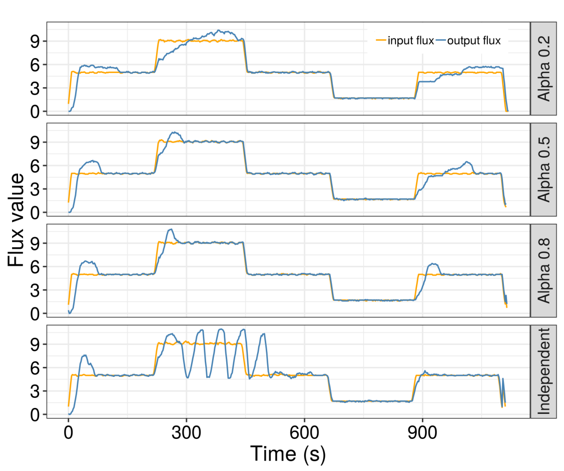
C. Discussion

The initial results of the experiments regarding the tracking of the reference value can be seen in Figure 7.2, and complemented by Figure 7.3 that showcases the tracked error, replica allocation and queue size for this scenario.

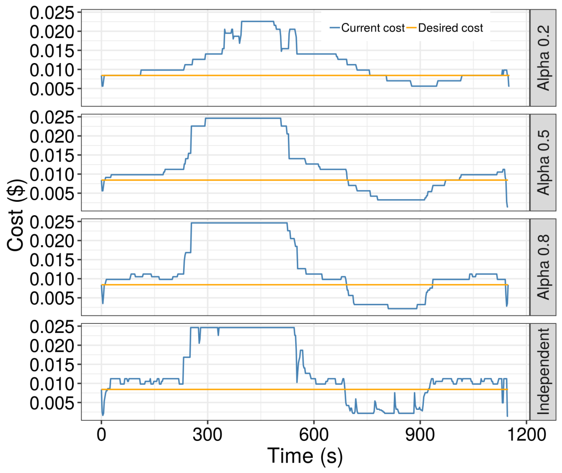
Figure 7.2a shows that for $\alpha = 0.8$, which favors performance constraints, the reference value is apparently followed without major problems despite small disturbances when there is a spike in the input rate. On the other hand, for $\alpha = 0.2$, which favors cost constraints, we can see that the system takes longer to be able to track the reference, especially when the workload changes. It is possible that this happens because the cost controller is delaying the scale up operations, as the utilization price of cluster resources increases in proportion to the increase in the number of replicas, as well as the total processing cost.

If we look at $\alpha = 0.5$, a certain balance is achieved. As for the case of independent controllers, we observed a less satisfactory behavior. The system does not appear to be able to track the reference value for almost half of the execution, causing an apparent destabilization of the system. This possibly happens because the controllers end up competing against each other, as the cost controller triggers actions to reduce the number of replicas, while the performance controller seeks to increase available resources to maintain its QoS goals. In general, such conditions are not desired because they can, for example, affect application availability when considering computing systems.

For the performance controller, as previously stated in Section B.4.2, the statistical anal-



(a) Performance controller.



(b) Cost controller.

Figure 7.2: Tracking of reference value for the SIMO and independent controllers approach.

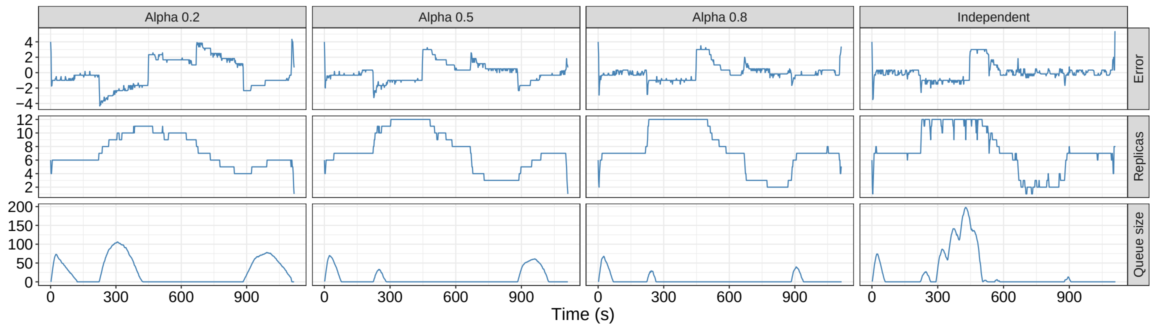


Figure 7.3: Tracked error, number of replicas and queue size metrics for the SIMO and independent controllers approach.

ysis for the tracking of the reference value uses the tracked error as a metric to evaluate the former. However, when comparing each control preference to the Independent approach, we can not reject the null hypothesis that there is no difference between the average error for each case. This might be because the error values collected are so close together that the difference in means is very short and the amount of data was not big enough to provide statistically significant results. Another option is that averaging this particular variable gives a general vision of the errors that occurred in a time range, whilst we are interested in the differences between the reference value and the actual throughput in each point of the execution. Therefore, although we visually see differences in Figure 7.2, the error metric does not reflect them well enough for an statistical analysis.

For the cost controller, as seen in Figure 7.2b, neither of the approaches manage to closely

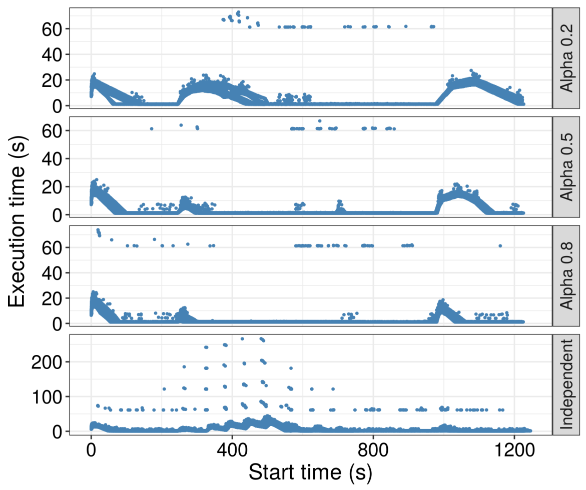


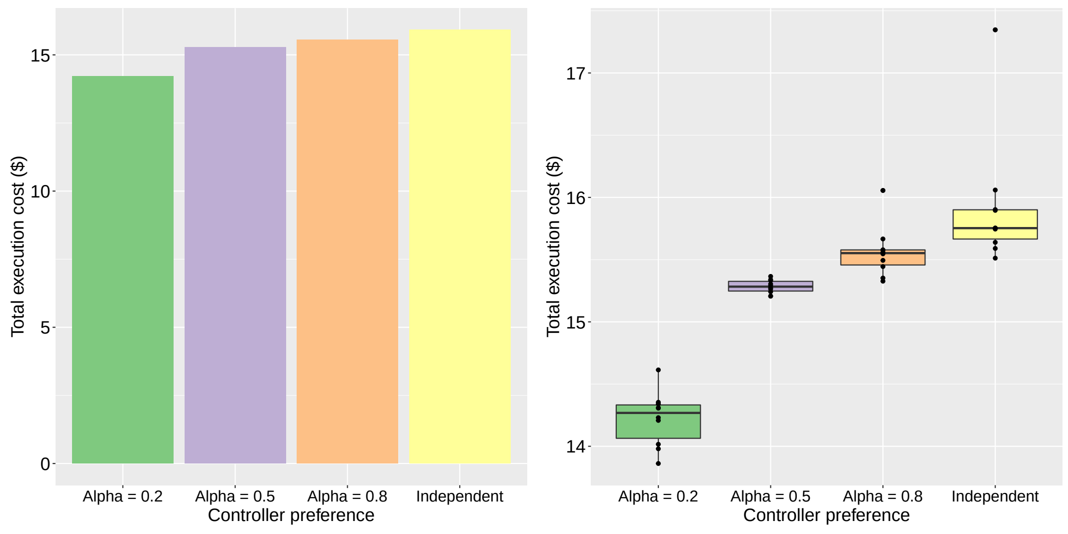
Figure 7.4: Execution time of tasks processed using the SIMO control approach and independent controllers.

follow the reference value. For $\alpha = 0.2$, when there is a preference for cost, is when the controller is the most able to do that. This is also stated by the statistical analysis in B.4.2, in which only this case presents a statistically significant difference between the tracked errors for the SIMO PI (0.2) and Independent configuration.

We then tried to evaluate the performance of each approach from a different perspective. Figure 7.4 shows the direct effect of different control approaches on the system response time, which is in practice the execution time of a task. For independent controllers, this scales quickly, and some tasks may remain on the system unfinished for more than 100 seconds. These results are related to the increase in queue size that normally happens when there is a spike in the input rate, as we can see in Figure 7.3.

In case of performance preference, as seen for $\alpha = 0.8$, it mainly takes from 1.5 to 2 seconds for a task to complete. For the SIMO approach, as expected, since time to complete a task is a performance metric in nature, the system tends to behave better when there is a greater preference for performance. In this case, the statistical analysis for the response time metric also states that there is in fact a statistically significant difference between each control preference for the SIMO PI and the Independent configuration, as seen in Tables 7.7 and 7.8.

From a financial point of view, Figure 7.5 presents the total cost of execution considering the previously defined prices for cluster resources. From Figure 7.5a we can see that the differences in average are actually small from each approach, but even so, we can see that



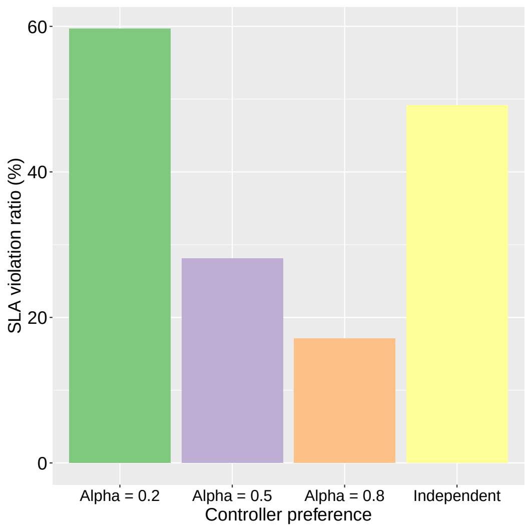
(a) Average execution cost. (b) Distribution of execution cost.

Figure 7.5: Financial impact for the SIMO PI and Independent configuration.

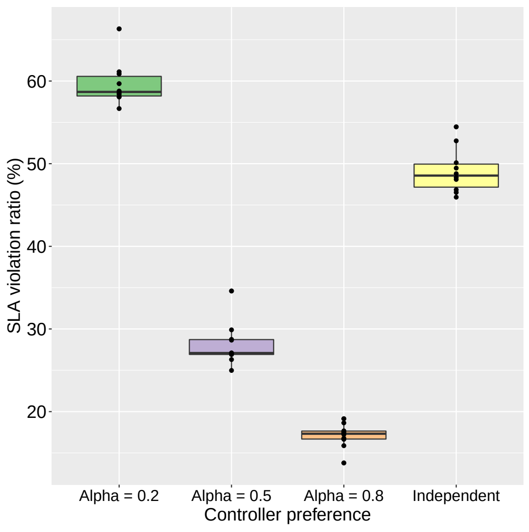
the higher the preference for cost metrics, the less is spent processing a workload, which is an expected result in this case. Figure 7.5b shows the distribution of this data, which showcases the differences a little more given the spacing between the boxes, although data does not seem to vary a lot among different approaches. Statistical analysis on Section B.4.3 confirms a statistically significant difference between the total execution cost for each control preference and the Independent approach.

On the other hand, Figure 7.6 shows the impact of performance on SLA violations, a very important metric when considering QoS goals. Here we assume that the SLA is based on the time to complete a task, with 1.5 seconds being considered acceptable. This is the average time it takes for a standard disaggregation operation to run, so it is a good value to define the SLA goals. Thus, from Figures 7.6a and 7.6b we see that for $\alpha = 0.8$, which represents a higher preference for performance, SLA violations happen around 15% of tasks and increase with higher preferences for cost metrics. Statistical analysis on Section B.4.3 and Tables 7.7 and 7.8 confirms a statistically significant difference between the SLA violation ratio for each control preference and the Independent approach.

Considering this, we can say that a user needs to carefully choose their preferences regarding performance and cost metrics. There are clear gains and losses involved in this choice, because if performance-related aspects are to be favored, cost metrics will be af-



(a) Average SLA violation ratio.



(b) Distribution of SLA violation ratio.

Figure 7.6: SLA impact for the SIMO PI and Independent configuration.

fected, and vice versa. As we can see, these metrics tend to be conflicting, meaning that high levels of performance typically require high resource utilization, consequently increasing the computational cost, which can end up going against user-defined cost preferences. In computer systems, for example, high availability is desired in most cases, which points to a tendency of these systems to favor performance metrics. However, if a workload can be processed at a slower pace without causing major losses in terms of quality of service, cost metrics can be favored by keeping resource utilization at lower levels.

Thus, deciding which preference values for cost and performance metrics are best suited for a given application is an important part of the process of choosing the control and orchestration strategies you want to adopt.

Chapter 8

Conclusions

This work proposed the application of control theory to orchestrate applications in micro-batch data stream processing systems. Given the nature of this type of system, work items must be processed in real time, thus, traditional adaptation approaches aimed at provisioning and scaling resources are evaluated in this work. In this context, it is clear that a fundamental part of this process is the monitoring of applications, which may not be an easy task as it depends on various characteristics such as the collect interval of the data of interest, or the level of customization and processing that may still be involved in this step.

Considering this difficulty, the use of control theory techniques to model these systems can help provide more accurate information about their behavior when executing the applications of interest, which in its turn, can be very helpful when adapting and scaling the resources needed to process a given stream of data. Thus, this work used system identification methods to provide a modeling of a DSP system based on the FOPDT approach. The system considered here used a neural network to classify and disaggregate energy data, which in our case was an implementation of the NIALM algorithm.

The evaluation of the generated models showed that, despite all of them having managed to reach the expected output for the given input, the transients of the executions did not accurately reflect the behavior of the real system. For FOPDT systems, the expectation is that the result is reached exponentially, which was not observed according to the amplitude and waveform of the results. There were indications that this was due to the system used as a use case responding almost immediately to changes in the input, and not exponentially, as expected, making it difficult to obtain more accurate results.

We also conclude that, from a generalization perspective, other classification algorithms using neural networks, a popular approach to categorize data from such workloads, could also be used as the application use case for this modeling. This is possible because such algorithms follow a similar processing approach even considering different workloads. Note that, the workload used here consisted of an homogeneous pool of tasks, which means that each of them takes about the same time to process. In this case, for a workload with different processing tasks, as long as such tasks are also homogeneous, the modeling could be equally performed as described in Chapter 5. For instance, changing the number of the layers in the network can change a little the duration of the task, but such task would still be only CPU intensive, reading and returning the same amount of data.

Next, considering the generated models, we presented a compensatory approach widely used in control theory, which makes use of filtering techniques to smooth out possible noise present in a given execution. Thus, a low-pass filter was incorporated into the system through modeling done in Matlab. We observed that, when removing the signals considered as disturbances, the resulting curve resembled more closely the model generated both in terms of waveform and amplitude. However, there are advantages and disadvantages in using filters, because when smoothing the signal, information about the execution is inevitably lost, which can generate a slower system response to unpredictable changes that may happen.

Then, a PI controller focusing on performance metrics was proposed, aiming to keep the system's throughput at the same level as its arrival rate, as expected for DSP systems. This controller was tuned using the FOPDT model as a basis, considering that it was able to represent the system up to a certain level and, therefore, it was used as a good starting point for tuning the gains of the proposed controller. From this tuning, we initially evaluate the controller comparing its performance against a classical control approach, which acts on the system in fixed-step actions, regardless of the magnitude of the error. Then, the controller was evaluated in relation to less grounded manual tuning approaches, considering the point of view of less experienced users. And finally, we evaluate the effects on using two different throughput calculation methods, one that computed the amount of processed items in real time, and other that is an estimate based on some known information about the system.

The results of these experiments were evaluated using statistical tests to provide an acceptable level of significance for the experimentation. The considered treatments for each comparative scenario were presented and evaluated regarding some metrics such as the system throughput, its ability to track a given reference value, the allocation of resources and the amount of SLA violations. We could see that most of the results were statistically significant, and that the controller tuned using the previously generated FOPDT model performed better overall. Besides that, a descriptive analysis of these results is presented in Appendix A.

A comment on the significance of the results is that, even when such results are statistically significant, but the means between two groups being compared are very similar and, in that sense, not that significant from a practical perspective, we can still see its importance from a different point of view. For instance, for the replica allocation variable in the experiments that compared the PI and the Fixed Action approach, the means are too close between both treatments and results are still statistically significant. In this case, we understand that low variation in the allocation is also of importance here, since availability can be compromised in the case of acquiring and removing replicas in such high frequency, as observed from the results for this scenario. For example, a sudden spike in the number of pods could trigger the scaling of node pools in managed clusters in cloud providers, such as AWS and Azure. Then, even if this would be quickly scaled down, there would still be cost and API availability impacts.

Furthermore, in order to extend the applicability of the methods evaluated here, a SIMO controller was also proposed, which aims to generate control actions based on performance and cost metrics. For this, two different controllers were implemented, one for each type of metric of interest, with the SIMO controller being responsible for combining the output of them both, and applying a single corrective action over the system. The solution uses a utility function that allows users to prioritize the metrics considered according to their needs. This controller was evaluated considering scenarios with different preferences for each metric of interest, in addition to a scenario where both the performance and cost controller acted independently on the system.

The results showed that using a combined control instead of an independent one presents an overall better performance than having independent controllers acting at the same time, however, deciding the preference values for the metrics of interest is an important part in the process of choosing the control strategy to be adopted. For this set of experiments, results were also evaluated using statistical tests to provide an acceptable level of significance. The metrics used for this evaluation included the system throughput, its ability to track a given reference value (for both cost and performance-focused controllers), the allocation of resources, the amount of SLA violations and the total execution cost for different types of controller preferences. A descriptive analysis of these results is also presented in Appendix A.

In terms of practical aspects of the solution, we used Asperathos to execute and orchestrate containerized applications in Kubernetes clusters, whose plugin customization architecture allowed the integration of the controllers proposed here, as well as the monitoring settings necessary for its functioning. For experimentation purposes, an application for processing micro-batches of stream data was executed, consisting in tasks to disaggregate energy data, part of the real use case proposed by LiteMe [3], that benefits from good results obtained in this work. Note that this workload used for system modeling and experimentation analysis is common for IoT sensor data. These facts contributes as an impact factor of this solution, by using a real use case and usual tools like Kubernetes itself.

The results of this work were incorporated into the paper *"Single-Input Multiple-Output Control for Multi-Goal Orchestration"* [144], published in the *2020 Utility and Cloud Computing* (UCC) conference. In addition, this study was part of the *ATMOSPHERE* project, that aimed to architect and implement a platform for the orchestration of secure cloud applications. This was a Brazil-Europe partnership, with contributions from universities such as the Universidade Federal de Campina Grande (UFCG), Technische Universität Dresden (TUD), University of Brasília (UnB), among others. Currently, this work is inserted in the context of the *LiteCampus* project, aimed at intelligent solutions for energy data processing, in partnership with the *Rede Nacional de Ensino e Pesquisa* (RNP), the company *Smartiks Ltda.*, and the *Agência Brasileira de Inovação Industrial* (EMBRAPII).

Finally, considering the results observed so far, future work could focus on trying to extend the solution for heterogeneous workloads, with tasks of different sizes, and a more varied arrival rate. Our scope does not consider these cases, focusing only on behaved workloads with overall same sized tasks, having similar service times, and arriving at a somewhat behaved rate. Since the arrival rate is used as the reference value of the control system, having a workload with an arrival rate that varies a lot makes it difficult to adapt the system without causing unwanted overshoot and system destabilization, as we saw on the experiments presented in this work. Furthermore, some techniques using machine learning to help adapt the control gains at runtime could be used to make the solution more robust to workload changes.

Bibliography

[1] Openstack user survey 2020. https://www.openstack.org/analytics. [Online; Last access: August 19th, 2022].

[2] Energy outlook 2019. Energy Information Administration. http://www.eia.doe.gov/oiaf/ieo/index.htm, 2019. [Online; Last access: August 19th, 2022].

[3] LiteMe Inteligência Energética. https://liteme.com.br/, 2019. [Online; Last access: August 19th, 2022].

[4] Amazon EC2 Auto Scaling. https://aws.amazon.com/pt/ec2/autoscaling/, 2021. [Online; Last access: August 19th, 2022].

[5] Amazon EC2 Instance types and billing. https://aws.amazon.com/ec2/instance-types/?nc1=h_ls, 2021. [Online; Last access: August 19th, 2022].

[6] Asperathos. https://github.com/ufcg-lsd/asperathos, 2021. [Online; Last access: August 19th, 2022].

[7] Docker Swarm. https://www.docker.com/products/docker-swarm/, 2021. [Online; Last access: August 19th, 2022].

[8] Kubernetes. https://www.kubernetes.io/, 2021. [Online; Last access: August 19th, 2022].

[9] Rightscale - Understanding the Voting Process. https://docs.rightscale.com/cm/rs101/understanding_the_voting_process.html#overview, 2021. [Online; Last access: August 19th, 2022].

[10] Ahmed S. Abdelhamid, Ahmed R. Mahmood, Anas Daghistani, and Walid G. Aref. Prompt: Dynamic data-partitioning for distributed micro-batch stream processing systems. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2455–2469, New York, NY, USA, 2020. Association for Computing Machinery.

[11] T. Abdelzaher, K.G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: a control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[12] Tarek Abdelzaher, Yixin Diao, Joseph L. Hellerstein, Chenyang Lu, and Xiaoyun Zhu. *Introduction to Control Theory And Its Application to Computing Systems*, pages 185–215. Springer US, Boston, MA, 2008.

[13] Tarek Abdelzaher, J.A. Stankovic, Chenyang Lu, Ronghua Zhang, and Ying Lu. Feedback performance control in software services. *IEEE Control Systems Magazine*, 23(3):74–90, 2003.

[14] S. N. Akshay Uttama Nambi, Thanasis G. Papaioannou, Dipanjan Chakraborty, and Karl Aberer. Sustainable energy consumption monitoring in residential settings. In *2013 Proceedings IEEE INFOCOM*, pages 3177–3182, 2013.

[15] F. Al-Haidari, M. Sqalli, and K. Salah. Impact of cpu utilization thresholds and scaling size on autoscaling cloud resources. In *Proceedings of the 2013 IEEE International Conference on Cloud Computing Technology and Science - Volume 02*, CLOUDCOM '13, page 256–261, USA, 2013. IEEE Computer Society.

[16] Ahmed Ali-Eldin, Johan Tordsson, Erik Elmroth, and Maria Kihl. Workload classification for efficient auto-scaling of cloud resources. 2013.

[17] Elham Almodaresi and Mohammad Bozorg. Computing stability domains in the space of time delay and controller coefficients for fopdt and sopdt systems. *Journal of Process Control*, 24(12):55–61, 2014.

[18] Eitan Altman, Tamer Başar, and R. Srikant. Congestion control as a stochastic control problem with action delays. *Automatica*, 35(12):1937–1950, December 1999.

[19] L. Amini, N. Jain, A. Sehgal, J. Silber, and O. Verscheure. Adaptive control of extreme-scale stream processing systems. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 71–71, 2006.

[20] Artur Andrzejak, Martin Arlitt, and Jerry Rolia. Bounding the resource savings of utility computing models. 01 2003.

[21] Karen Appleby, Sameh A. Fakhouri, Liana L. Fong, Germán S. Goldszmidt, Michael H. Kalantar, Srirama M. Krishnakumar, Donald P. Pazel, John A. Pershing, and Benny Rochwerger. Oceano-sla based management of a computing utility. *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, pages 855–868, 2001.

[22] Igor Ataide, Gabriel Vinha, Clenimar Souza, and Andrey Brito. Implementing quality of service and confidentiality for batch processing applications. pages 258–265, 12 2018.

[23] C. Aurrecoechea, A. Campbell, and Linda Hauw. A survey of qos architectures. *Multimedia Systems*, 6:138–151, 1998.

[24] Luciano Baresi, Sam Guinea, Alberto Leva, and Giovanni Quattrocchi. A discrete-time feedback controller for containerized cloud applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, page 217–228, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Cornel Barna, Marios Fokaefs, Marin Litoiu, Mark Shtern, and Joe Wigglesworth. Cloud adaptation with control theory in industrial clouds. In *2016 IEEE International Conference on Cloud Engineering Workshop (IC2EW)*, pages 231–238, 2016.

[26] M. Ben-Ari. *Principles of Concurrent and Distributed Programming (2nd Edition) (Prentice-Hall International Series in Computer Science).* Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[27] Qiang Bi, Wen-Jian Cai, Eng-Lock Lee, Qing-Guo Wang, Chang-Chieh Hang, and Yong Zhang. Robust identification of first-order plus dead-time model from step response. *Control Engineering Practice*, 7(1):71–77, 1999.

[28] Michael Borkowski, Christoph Hochreiner, and Stefan Schulte. Minimizing cost by reducing scaling operations in distributed stream processing. *Proc. VLDB Endow.*, 12(7):724–737, March 2019.

[29] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. *Engineering Self-Adaptive Systems through Feedback Loops*, pages 48–70. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[30] Kevin Burn and Chris Cox. A hands-on approach to teaching system identification using first-order plus dead time modelling of step response data. *The International Journal of Electrical Engineering & Education*, 57(1):24–40, 2020.

[31] Yao Cai, Qiang Zhan, and Xi Xi. Neural network control for the linear motion of a spherical mobile robot. *International Journal of Advanced Robotic Systems*, 8(4):32, 2011.

[32] Nicolo M. Calcavecchia, Bogdan Alexandru Caprarescu, Elisabetta Di Nitto, Daniel J. Dubois, and Dana Petcu. Depas: A decentralized probabilistic algorithm for auto-scaling, 2012.

[33] Valeria Cardellini, Matteo Nardelli, and Dario Luzi. Elastic stateful stream processing in storm. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 583–590, 2016.

[34] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter Pietzuch. Integrating scale out and fault tolerance in stream processing using operator state management. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, page 725–736, New York, NY, USA, 2013. Association for Computing Machinery.

[35] Marcelo Cerqueira de Abranches and Priscila Solis. An algorithm based on response time and traffic demands to scale containers on a cloud computing system. In *2016 IEEE 15th International Symposium on Network Computing and Applications (NCA)*, pages 343–350, 2016.

[36] Javier Cerviño Arriba, Evangelia Kalyvianaki, Joaquin Salvachua, and Peter Pietzuch. Adaptive provisioning of stream processing systems in the cloud. 05 2012.

[37] Abhishek Ch and Prashant Shenoy. Effectiveness of dynamic resource allocation for handling internet flash crowds. 12 2003.

[38] Abhishek Chandra, Pawan Goyal, and Prashant Shenoy. Quantifying the benefits of resource multiplexing in on-demand data centers. 01 2003.

[39] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001.

[40] Mayank Chaturvedi, Prateeksha Chauhaan, and Pradeep Juneja. *Design of Time-Delay Compensator for a FOPDT Process Model*, volume 396, pages 205–211. 01 2016.

[41] Tao Chen and Rami Bahsoon. Self-adaptive and online qos modeling for cloud-based software services. *IEEE Trans. Softw. Eng.*, 43(5):453–475, May 2017.

[42] Xin Chen, Ymir Vigfusson, Douglas Blough, Fang Zheng, Kun-Lung Wu, and Liting Hu. Governor: Smoother stream processing through smarter backpressure. pages 145–154, 07 2017.

[43] Betty H. Cheng, Rogério Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. *Software Engineering for Self-Adaptive Systems: A Research Roadmap*, page 1–26. Springer-Verlag, Berlin, Heidelberg, 2009.

[44] Cloud Native Computing Foundation. Cncf survey 2020. https://www.cncf.io/wp-content/uploads/2020/11/CNCF$_{S}urvey_{R}eport_{2}020.pdf$, 2020. $[Online; Last access : August 19th, 2022]$.

[45] Marco Comuzzi and Barbara Pernici. A framework for qos-based web service contracting. *ACM Trans. Web*, 3(3), July 2009.

[46] Chris Cox, John Tindle, and Kevin Burn. A comparison of software-based approaches to identifying fopdt and sopdt model parameters from process step response data. *Applied Mathematical Modelling*, 40(1):100–114, 2016.

[47] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive stream processing using dynamic batch sizing. In *Proceedings of the ACM Symposium on Cloud Computing*, SOCC '14, page 1–13, New York, NY, USA, 2014. Association for Computing Machinery.

[48] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In P. Venkata Krishna, M. Rajasekhara Babu, and Ezendu Ariwa, editors, *Global Trends in Computing and Communication Systems*, pages 11–25, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[49] Miyuru Dayarathna and Srinath Perera. Recent advancements in event processing. *ACM Comput. Surv.*, 51(2), February 2018.

[50] Tiziano De Matteis and Gabriele Mencagli. Proactive elasticity and energy awareness in data stream processing. *J. Syst. Softw.*, 127(C):302–319, May 2017.

[51] Yixin Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *NOMS 2002. IEEE/IFIP Network Operations and Management Symposium. ' Management Solutions for the New Communications World'(Cat. No.02CH37327)*, pages 219–234, 2002.

[52] Yixin Diao, Joseph L. Hellerstein, and Sujay Parekh. Control of large scale computing systems. *SIGBED Rev.*, 3(2):17–22, April 2006.

[53] Marcos Dias de Assunção, Alexandre da Silva Veith, and Rajkumar Buyya. Distributed data stream processing and edge computing: A survey on resource elasticity and future directions. *Journal of Network and Computer Applications*, 103:1–17, 2018.

[54] Yezid Donoso and Ramón Fabregat. Network optimization using evolutionary algorithms in multicast transmission. In Mario Freire and Manuela Pereira, editors, *Encyclopedia of Internet Technologies and Applications*, pages 339–345. Hershey, PA: IGI Global, 2008.

[55] Corentin Dupont, Mehdi Sheikhalishahi, Federico M. Facca, and Fabien Hermenier. An energy aware application controller for optimizing renewable energy consumption in data centres. In *Proceedings of the 8th International Conference on Utility and Cloud Computing*, UCC '15, page 195–204. IEEE Press, 2015.

[56] Xavier Dutreilh, Aurélien Moreau, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. From data center resource allocation to control theory and back. In *Proceedings of the 2010 IEEE 3rd International Conference on Cloud Computing*, CLOUD '10, page 410–417, USA, 2010. IEEE Computer Society.

[57] George Ellis. Chapter 9 - filters in control systems. In George Ellis, editor, *Control System Design Guide (Fourth Edition)*, pages 165–183. Butterworth-Heinemann, Boston, fourth edition edition, 2012.

[58] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Lakew, Ivona Brandic, and Erik Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65, 05 2016.

[59] Soodeh Farokhi, Ewnetu Bayuh Lakew, Cristian Klein, Ivona Brandic, and Erik Elmroth. Coordinating cpu and memory elasticity controllers to meet service response time constraints. In *2015 International Conference on Cloud and Autonomic Computing*, pages 69–80, 2015.

[60] Antonio Filieri, Carlo Ghezzi, Alberto Leva, and Martina Maggio. Self-adaptive software meets control theory: A preliminary approach supporting reliability requirements. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 283–292, USA, 2011. IEEE Computer Society.

[61] Antonio Filieri, Carlo Ghezzi, and Giordano Tamburrelli. A formal approach to adaptive software: Continuous assurance of non-functional requirements. *Form. Asp. Comput.*, 24(2):163–186, March 2012.

[62] Antonio Filieri, Henry Hoffmann, and Martina Maggio. Automated design of self-adaptive software with control-theoretical formal guarantees. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 299–310, New York, NY, USA, 2014. Association for Computing Machinery.

[63] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolas D'Ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro Vittorio Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Software engineering meets control theory. In *2015 IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 71–82, 2015.

[64] Antonio Filieri, Martina Maggio, Konstantinos Angelopoulos, Nicolás D'ippolito, Ilias Gerostathopoulos, Andreas Berndt Hempel, Henry Hoffmann, Pooyan Jamshidi, Evangelia Kalyvianaki, Cristian Klein, Filip Krikava, Sasa Misailovic, Alessandro V. Papadopoulos, Suprio Ray, Amir M. Sharifloo, Stepan Shevtsov, Mateusz Ujma, and Thomas Vogel. Control strategies for self-adaptive software systems. *ACM Trans. Auton. Adapt. Syst.*, 11(4), February 2017.

[65] Stefan Frey, Claudia Lüthje, Christoph Reich, and Nathan Clarke. Cloud qos scaling by fuzzy logic. In *Proceedings of the 2014 IEEE International Conference on Cloud Engineering*, IC2E '14, page 343–348, USA, 2014. IEEE Computer Society.

[66] Tom Z. J. Fu, Jianbing Ding, Richard T. B. Ma, Marianne Winslett, Yin Yang, and Zhenjie Zhang. Drs: Auto-scaling for real-time stream analytics. 25(6):3338–3352, December 2017.

[67] Bugra Gedik, Scott Schneider, Martin Hirzel, and Kun-Lung Wu. Elastic scaling for data stream processing. *Parallel and Distributed Systems, IEEE Transactions on*, 25:1447–1463, 06 2014.

[68] GitHub Inc. The 2020 state of the octoverse. https://octoverse.github.com/. [Online; Last access: August 19th, 2022].

[69] Armstrong Goes, Fabio Morais, Eduardo Falcão, and Andrey Brito. Assuring cloud qos through loop feedback controller assisted vertical provisioning. 01 2019.

[70] Siqian Gong, Beibei Yin, and Kai-yuan Cai. An adaptive pid control for qos management in cloud computing system. In *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 142–143, 2018.

[71] Siqian Gong, Beibei Yin, Wenlong Zhu, and Kaiyuan Cai. An adaptive control strategy for resource allocation of service-based systems in cloud environment. In *2015 IEEE International Conference on Software Quality, Reliability and Security - Companion*, pages 32–39, 2015.

[72] Jordi Guitart, Jordi Torres, and Eduard Ayguadé. A survey on performance management for internet applications. *Concurrency and Computation: Practice and Experience*, 22:68–106, 01 2010.

[73] Vincenzo Gulisano, Ricardo Jiménez-Peris, Marta Patiño-Martínez, Claudio Soriente, and Patrick Valduriez. Streamcloud: An elastic and scalable data streaming system. *IEEE Transactions on Parallel and Distributed Systems*, 23:2351–2365, 2012.

[74] Yukang Guo, Matt Jones, Benjamin Cowan, and Russell Beale. Take it personally: Personal accountability and energy consumption in domestic households. In *CHI '13 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '13, page 1467–1472, New York, NY, USA, 2013. Association for Computing Machinery.

[75] C.C. Hang, K.J. Astrom, and Q.G. Wang. Relay feedback auto-tuning of process controllers — a tutorial review. *Journal of Process Control*, 12(1):143–162, 2002.

[76] G.W. Hart. Nonintrusive appliance load monitoring. *Proceedings of the IEEE*, 80(12):1870–1891, 1992.

[77] Thomas Heinze, Valerio Pappalardo, Zbigniew Jerzak, and Christof Fetzer. Auto-scaling techniques for elastic data stream processing. In *Proceedings of the 8th ACM International*

Conference on Distributed Event-Based Systems, DEBS ’14, page 318–321, New York, NY, USA, 2014. Association for Computing Machinery.

[78] Joseph L. Hellerstein. Challenges in control engineering of computing systems. In *Proceedings of the 2004 American Control Conference*, volume 3, pages 1970–1979 vol.3, 2004.

[79] Joseph L. Hellerstein, Yixin Diao, S. Parekh, and D.M. Tilbury. Control engineering for computing systems - industry experience and research challenges. *IEEE Control Systems Magazine*, 25(6):56–68, 2005.

[80] Joseph L. Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M. Tilbury. *Feedback Control of Computing Systems*. John Wiley Sons, Inc., Hoboken, NJ, USA, 2004.

[81] Joseph L. Hellerstein, Sharad Singhal, and Qian Wang. Research challenges in control engineering of computing systems. *IEEE Transactions on Network and Service Management*, 6(4):206–211, 2009.

[82] Christoph Hochreiner, Michael Vögler, Stefan Schulte, and Schahram Dustdar. Elastic stream processing for the internet of things. 06 2016.

[83] C.V. Hollot, V. Misra, D. Towsley, and Wei-Bo Gong. A control theoretic analysis of red. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No.01CH37213)*, volume 3, pages 1510–1519 vol.3, 2001.

[84] Mohammadreza Hoseinyfarahabady, Albert Zomaya, and Zahir Tari. Qos- and contention-aware resource provisioning in a stream processing engine. pages 137–146, 09 2017.

[85] Hameed Hussain, Saif Ur Rehman Malik, Abdul Hameed, Samee Ullah Khan, Gage Bickler, Nasro Min-Allah, Muhammad Bilal Qureshi, Limin Zhang, Wang Yongji, Nasir Ghani, Joanna Kolodziej, Albert Y. Zomaya, Cheng-Zhong Xu, Pavan Balaji, Abhinav Vishnu, Fredric Pinel, Johnatan E. Pecero, Dzmitry Kliazovich, Pascal Bouvry, Hongxiang Li, Lizhe Wang, Dan Chen, and Ammar Rayes. A survey on resource allocation in high performance distributed computing systems. *Parallel Computing*, 39(11):709–736, 2013.

[86] Oladimeji Ibrahim, Zaihar Yahaya, and Nordin Saad. Pid controller response to set-point change in dc-dc converter control. 7:294–302, 06 2016.

[87] Shigeru Imai, Stacy Patterson, and Carlos A. Varela. Uncertainty-aware elastic virtual machine scheduling for stream processing systems. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 62–71, 2018.

[88] Waheed Iqbal, Matthew Dailey, and David Carrera. Sla-driven adaptive resource management for web applications on a heterogeneous compute cloud. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom '09, page 243–253, Berlin, Heidelberg, 2009. Springer-Verlag.

[89] Atsushi Ishii and Toyotaro Suzumura. Elastic stream computing with clouds. In *2011 IEEE 4th International Conference on Cloud Computing*, pages 195–202, 2011.

[90] R. Jain. *The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling*. Wiley New York, 1991.

[91] Pooyan Jamshidi, Amir Sharifloo, Claus Pahl, Hamid Arabnejad, Andreas Metzger, and Giovani Estrada. Fuzzy self-learning controllers for elasticity management in dynamic cloud architectures. In *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pages 70–79, 2016.

[92] Philipp K. Janert. *Feedback Control for Computer Systems*. O'Reilly Media, Inc., 2013.

[93] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: Fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 783–798, USA, 2018. USENIX Association.

[94] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 249–262, New York, NY, USA, 2018. Association for Computing Machinery.

[95] Evangelia Kalyvianaki, Themistoklis Charalambous, Marco Fiscato, and Peter Pietzuch. Overload management in data stream processing systems with latency guarantees. In *7th*

IEEE International Workshop on Feedback Computing, United States, 2012. IEEE. IEEE International Workshop on Feedback Computing ; Conference date: 01-01-2012.

[96] Pankaj Deep Kaur and Inderveer Chana. A resource elasticity framework for qos-aware execution of cloud applications. *Future Generation Computer Systems*, 37:14–25, 2014. Special Section: Innovative Methods and Algorithms for Advanced Data-Intensive Computing Special Section: Semantics, Intelligent processing and services for big data Special Section: Advances in Data-Intensive Modelling and Simulation Special Section: Hybrid Intelligence for Growing Internet and its Applications.

[97] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[98] Srinivasan Keshav. A control-theoretic approach to flow control. *SIGCOMM Comput. Commun. Rev.*, 21(4):3–15, August 1991.

[99] Maria Kihl, Erik Elmroth, Johan Tordsson, Karl Erik Årzén, and Anders Robertsson. The challenge of cloud control. In *8th International Workshop on Feedback Computing (Feedback Computing 13)*, San Jose, CA, June 2013. USENIX Association.

[100] Diwakar T. Korsane, Vivek Yadav, and Kiran H. Raut. Pid tuning rules for first order plus time delay system. 2014.

[101] Palden Lama and X. Zhou. Efficient server provisioning with end-to-end delay guarantee on multi-tier clusters. *17th Int. Workshop on Quality of Service*, pages 1–9, 01 2009.

[102] Palden Lama and Xiaobo Zhou. Autonomic provisioning with self-adaptive neural fuzzy control for end-to-end delay guarantee. In *Proceedings of the 2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '10, page 151–160, USA, 2010. IEEE Computer Society.

[103] Han Li and Srikumar Venugopal. Using reinforcement learning for controlling an elastic web application hosting platform. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC '11, page 205–208, New York, NY, USA, 2011. Association for Computing Machinery.

[104] Hao li, Jianhui Liu, and Guo Tang. A pricing algorithm for cloud computing resources. volume 1, pages 69 – 73, 06 2011.

[105] Kang Li, M.H. Shor, J. Walpole, C. Pu, and D.C. Steere. Modeling the effect of short-term rate variations on tcp-friendly congestion control behavior. In *Proceedings of the 2001 American Control Conference. (Cat. No.01CH37148)*, volume 4, pages 3006–3012 vol.4, 2001.

[106] Teng Li, Zhiyuan Xu, Jian Tang, and Yanzhi Wang. Model-free control for distributed stream data processing using deep reinforcement learning. *Proc. VLDB Endow.*, 11(6):705–718, February 2018.

[107] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated control in cloud computing: Challenges and opportunities. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds*, ACDC '09, page 13–18, New York, NY, USA, 2009. Association for Computing Machinery.

[108] X. Liu, X. Zhu, S. Singhal, and M. Arlitt. Adaptive entitlement control of resource containers on shared servers. In *2005 9th IFIP/IEEE International Symposium on Integrated Network Management, 2005. IM 2005.*, pages 163–176, 2005.

[109] Lennart Ljung. *System Identification (2nd Ed.): Theory for the User*. Prentice Hall PTR, USA, 1999.

[110] Bjorn Lohrmann, Peter Janacik, and Odej Kao. Elastic stream processing with latency guarantees. *Proceedings - International Conference on Distributed Computing Systems*, 2015:399–410, 07 2015.

[111] Raquel Lopes, Francisco Brasileiro, and Paulo Ditarso Maciel. Business-driven capacity planning of a cloud-based it infrastructure for the execution of web applications. In *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1–8, 2010.

[112] Raquel V. Lopes and Daniel Menascé. A taxonomy of job scheduling on distributed computing systems. *IEEE Transactions on Parallel and Distributed Systems*, 27(12):3412–3428, 2016.

[113] Martina Maggio and Alberto Leva. Toward a deeper use of feedback control in the design of critical computing system components. In *49th IEEE Conference on Decision and Control (CDC)*, pages 5985–5990, 2010.

[114] S. Majhi and D.P. Atherton. Online tuning of controllers for an unstable fopdt process. *Control Theory and Applications, IEE Proceedings -*, 147:421 – 427, 08 2000.

[115] Vania Marangozova-Martin, Noël de Palma, and Ahmed El Rheddane. Multi-level elasticity for data stream processing. *IEEE Transactions on Parallel and Distributed Systems*, 30(10):2326–2337, 2019.

[116] André Martin, Andrey Brito, and Christof Fetzer. Real time data analysis of taxi rides using streammine3g. In *Proceedings of the 9th ACM International Conference on Distributed Event-Based Systems*, DEBS '15, page 269–276, New York, NY, USA, 2015. Association for Computing Machinery.

[117] André Martin, Andrey Brito, and Christof Fetzer. Real-time social network graph analysis using streammine3g. In *Proceedings of the 10th ACM International Conference on Distributed and Event-Based Systems*, DEBS '16, page 322–329, New York, NY, USA, 2016. Association for Computing Machinery.

[118] Matlab. Matlab control system tootlbox. https://www.mathworks.com/products/control.html. [Online; Last access: August 19th, 2022].

[119] Matlab. Matlab pid tuner. https://www.mathworks.com/help/control/ref/pidtuner-app.html. [Online; Last access: August 19th, 2022].

[120] Matlab. Matlab system identification tootlbox. https://www.mathworks.com/products/sysid.html. [Online; Last access: August 19th, 2022].

[121] Lucas Mearian. The state of the octoverse 2017. Computerworld - https://www.computerworld.com/article/2960642/cloud-storage/cerns-data-stores-soar-to-530m-gigabytes.html. [Online; Last access: August 19th, 2022].

[122] Daniel A. Menascé. Qos issues in web services. *IEEE Internet Computing*, 6(6):72–75, November 2002.

[123] Gabriele Mencagli, Massimo Torquati, and Marco Danelutto. Elastic-ppq: A two-level autonomic system for spatial preference query processing over dynamic data streams. *Future Gener. Comput. Syst.*, 79:862–877, 2018.

[124] Gabriele Mencagli, Massimo Torquati, Marco Danelutto, and Tiziano De Matteis. Parallel continuous preference queries over out-of-order and bursty data streams. *IEEE Transactions on Parallel and Distributed Systems*, PP, 03 2017.

[125] Fábio Morais, Raquel Lopes, and Francisco Brasileiro. Provisionamento automático de recursos em nuvem iaas: eficiência e limitações de abordagens reativas. In *Anais do XXXV Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Porto Alegre, RS, Brasil, 2017. SBC.

[126] Cristina I. Muresan and Clara M. Ionescu. Generalization of the fopdt model for identification and control purposes. *Processes*, 8(6), 2020.

[127] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martín Abadi. Naiad: A timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 439–455, New York, NY, USA, 2013. Association for Computing Machinery.

[128] Jin Seok Noh, Geun Hyeong Lee, Ho Jin Choi, and Seul Jung. Robust control of a mobile inverted pendulum robot using a rbf neural network controller. In *2008 IEEE International Conference on Robotics and Biomimetics*, pages 1932–1937, 2009.

[129] Pradeep Padala, Kai-Yuan Hou, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, and Arif Merchant. Automated control of multiple virtualized resources. In *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, page 13–26, New York, NY, USA, 2009. Association for Computing Machinery.

[130] Pradeep Padala, Kang G. Shin, Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, and Kenneth Salem. Adaptive control of virtualized resources in utility computing environments. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, EuroSys '07, page 289–302, New York, NY, USA, 2007. Association for Computing Machinery.

[131] Alessandro Vittorio Papadopoulos, Martina Maggio, and Alberto Leva. Control and design of computing systems: What to model and how. *IFAC Proceedings Volumes*, 45(2):102–107, 2012. 7th Vienna International Conference on Mathematical Modelling.

[132] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus. Using control theory to achieve service level objectives in performance management. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No.01EX470)*, pages 841–854, 2001.

[133] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A multi-model framework to implement self-managing control systems for qos management. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '11, page 218–227, New York, NY, USA, 2011. Association for Computing Machinery.

[134] Tharindu Patikirikorala, Alan Colman, Jun Han, and Liuping Wang. A systematic survey on the design of self-adaptive software systems using control engineering approaches. In *Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS '12, page 33–42. IEEE Press, 2012.

[135] Raghavendra Pradyumna Pothukuchi, Amin Ansari, Petros Voulgaris, and Josep Torrellas. Using multiple input, multiple output formal control to maximize resource efficiency in architectures. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 658–670, 2016.

[136] Zhengping Qian, Yong He, Chunzhi Su, Zhuojie Wu, Hongyu Zhu, Taizhi Zhang, Lidong Zhou, Yuan Yu, and Zheng Zhang. Timestream: Reliable stream computation in the cloud. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, page 1–14, New York, NY, USA, 2013. Association for Computing Machinery.

[137] Cui Qin, Holger Eichelberger, and Klaus Schmid. Enactment of adaptation in data stream processing with latency implications—a systematic literature review. *Information and Software Technology*, 03 2019.

[138] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. *ACM Comput. Surv.*, 51(4), July 2018.

[139] Wan Nurhayati Wan Ab. Rahman and F. Meziane. Challenges to describe qos requirements for web services quality prediction to support web services interoperability in electronic commerce. 2008.

[140] S. Ranjan, Jerry Rolia, H. Fu, and E. Knightly. Qos-driven server migration for internet data centers. pages 3 – 12, 02 2002.

[141] Jia Rao, Yudi Wei, Jiayu Gong, and Cheng-Zhong Xu. Qos guarantees and service differentiation for dynamic cloud applications. *IEEE Transactions on Network and Service Management*, 10(1):43–55, 2013.

[142] Sajith Ravindra, Miyuru Dayarathna, and Sanath Jayasena. Latency aware elastic switching-based stream processing over compressed data streams. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 91–102, New York, NY, USA, 2017. Association for Computing Machinery.

[143] Henriette Röger and Ruben Mayer. A comprehensive survey on parallelization and elasticity in stream processing. *ACM Comput. Surv.*, 52(2), April 2019.

[144] Lilia Sampaio, Armstrong Goes, Maxwell Albuquerque, Diego Gama, Jose Schmid, and Andrey Brito. Single-input multiple-output control for multi-goal orchestration. pages 206–215, 12 2020.

[145] Lilia Sampaio, Clenimar Souza, Gabriel Vinha, and Andrey Brito. Asperathos: Running qos-aware sensitive batch applications with intel sgx. pages 89–96, 09 2019.

[146] Lui Sha, Xue Liu, Ying Lu, and Tarek Abdelzaher. Queueing model based network server performance control. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 81–90, 2002.

[147] Stepan Shevtsov, Mihaly Berekmeri, Danny Weyns, and Martina Maggio. Control-theoretical software adaptation: A systematic literature review. *IEEE Transactions on Software Engineering*, 44(8):784–810, 2018.

[148] Sukhpal Singh and Inderveer Chana. Qos-aware autonomic resource management in cloud computing: A systematic review. *ACM Comput. Surv.*, 48(3), December 2015.

[149] Sukhpal Singh and Inderveer Chana. A survey on resource scheduling in cloud computing: Issues and challenges. *J. Grid Comput.*, 14(2):217–264, June 2016.

[150] Sigurd Skogestad. Simple analytic rules for model reduction and pid controller tuning. *Journal of Process Control*, 13(4):291–309, 2003.

[151] Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley Sons, Inc., Hoboken, NJ, USA, 2005.

[152] Zhen Sun and Zhenyu Yang. System identification for nonlinear fopdt model with input-dependent dead-time. In *15th International Conference on System Theory, Control and Computing*, pages 1–6, 2011.

[153] Gerald Tesauro. Online resource allocation using decompositional reinforcement learning. In *Proceedings of the 20th National Conference on Artificial Intelligence - Volume 2*, AAAI'05, page 886–891. AAAI Press, 2005.

[154] Fuquan Tian, Wenbo Xu, and Liu Juan. Web qos control using fuzzy adaptive pi controller. *International Symposium on Distributed Computing and Applications to Business, Engineering and Science*, 0:72–75, 08 2010.

[155] A. Tosatto, P. Ruiu, and A. Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems*, pages 70–75, July 2015.

[156] Amjad Ullah, Jingpeng Li, Yindong Shen, and A. Hussain. A control theoretical view of cloud elasticity: taxonomy, survey and challenges. *Cluster Computing*, 21:1735–1764, 2018.

[157] Bhuvan Urgaonkar, Giovanni Pacifici, Prashant Shenoy, Mike Spreitzer, and Asser Tantawi. An analytical model for multi-tier internet services and its applications. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 291–302, New York, NY, USA, 2005. Association for Computing Machinery.

[158] Bhuvan Urgaonkar, Prashant Shenoy, and Timothy Roscoe. Resource overbooking and application profiling in shared hosting platforms. *SIGOPS Oper. Syst. Rev.*, 36(SI):239–254, December 2003.

[159] Akshay S.N. Uttama Nambi, Antonio Reyes Lua, and Venkatesha R. Prasad. Loced: Location-aware energy disaggregation framework. In *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, BuildSys '15, page 45–54, New York, NY, USA, 2015. Association for Computing Machinery.

[160] Augusto Vega, Alper Buyuktosunoglu, Heather Hanson, Pradip Bose, and Srinivasan Ramani. Crank it up or dial it down: Coordinated multiprocessor frequency and folding control. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, page 210–221, New York, NY, USA, 2013. Association for Computing Machinery.

[161] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. pages 374–389, 10 2017.

[162] Smita Vijayakumar, Qian Zhu, and Gagan Agrawal. Dynamic resource provisioning for data streaming applications in a cloud environment. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science*, pages 441–448, 2010.

[163] Ke Wang, Avrilia Floratou, Ashvin Agrawal, and Daniel Musgrave. Spur: Mitigating slow instances in large-scale streaming pipelines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2271–2285, New York, NY, USA, 2020. Association for Computing Machinery.

[164] Yidan Wang, Zahir Tari, Mohammadreza Hoseinyfarahabady, and Albert Zomaya. Model-based scheduling for stream processing systems. pages 215–222, 12 2017.

[165] Jianbin Wei and Cheng-Zhong Xu. Feedback control approaches for quality of service guarantees in web servers. In *NAFIPS 2005 - 2005 Annual Meeting of the North American Fuzzy Information Processing Society*, pages 700–705, 2005.

[166] Jing Xu, Ming Zhao, Jose Fortes, Robert Carpenter, and Mazin Yousif. On the use of fuzzy modeling in virtualized data center management. In *Proceedings of the Fourth International Conference on Autonomic Computing*, ICAC '07, page 25, USA, 2007. IEEE Computer Society.

[167] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling stream processing systems to scale-in and scale-out on-demand. pages 22–31, 04 2016.

[168] Zhenyu Yang and Glen T. Seested. Time-delay system identification using genetic algorithm – part one: Precise fopdt model estimation. *IFAC Proceedings Volumes*, 46(20):561–567, 2013. 3rd IFAC Conference on Intelligent Control and Automation Science ICONS 2013.

[169] Lenar Yazdanov and Christof Fetzer. Vscaler: Autonomic virtual machine scaling. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, page 212–219, USA, 2013. IEEE Computer Society.

[170] Tao Yu and Kwei-Jay Lin. The design of qos broker algorithms for qos-capable web services. *International Journal of Web Services Research*, 1(4), 2004.

[171] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 423–438, New York, NY, USA, 2013. Association for Computing Machinery.

[172] Liangzhao Zeng, Boualem Benatallah, Anne H.H. Ngu, Marlon Dumas, Jayant Kalagnanam, and Henry Chang. Qos-aware middleware for web services composition. *IEEE Trans. Softw. Eng.*, 30(5):311–327, May 2004.

[173] Qian Zhu and Gagan Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. *IEEE Trans. Serv. Comput.*, 5(4):497–511, January 2012.

[174] Xiaoyun Zhu, Mustafa Uysal, Zhikui Wang, Sharad Singhal, Arif Merchant, Pradeep Padala, and Kang Shin. What does control theory bring to systems research? *SIGOPS Oper. Syst. Rev.*, 43(1):62–69, January 2009.

[175] K.J. Åström and T. Hägglund. Automatic tuning of simple regulators with specifications on phase and amplitude margins. *Automatica*, 20(5):645–651, 1984.

Appendix A

Descriptive analysis of experimentation data

A.1 Scenarios I and II: Proportional-Integral x Fixed Action

As described by Table 6.4, two control strategies are defined for this scenario, one called Fixed Action and the other Proportional-Integral. For the the Fixed Action approach, we consider two control configuration values, 1 and 3, which are named in the graphs as *Fixed Step - 1* and *Fixed Step - 3*, respectively. The Proportional-Integral scenario is named *Analytic PI - Runtime*, since the throughput is calculated at runtime. The remainder levels of the described factors do not change between treatments.

Considering this, in this section we present a summary and distribution of the collected data with regards to the system throughput, represented by the tracked error in this analysis, replica allocation and response time.

A.1.1 Tracked error

First, we present the data for the tracked error variable for the three treatments in question: Analytic PI, Fixed Action - 1 and Fixed Action - 3. Initially we are interested in 4 random replications of the experiment, for each treatment, to showcase how the data spreads before summarizing it by a single number.

Figure A.1 showcases the distribution of the tracked error for the Analytic PI control strategy.

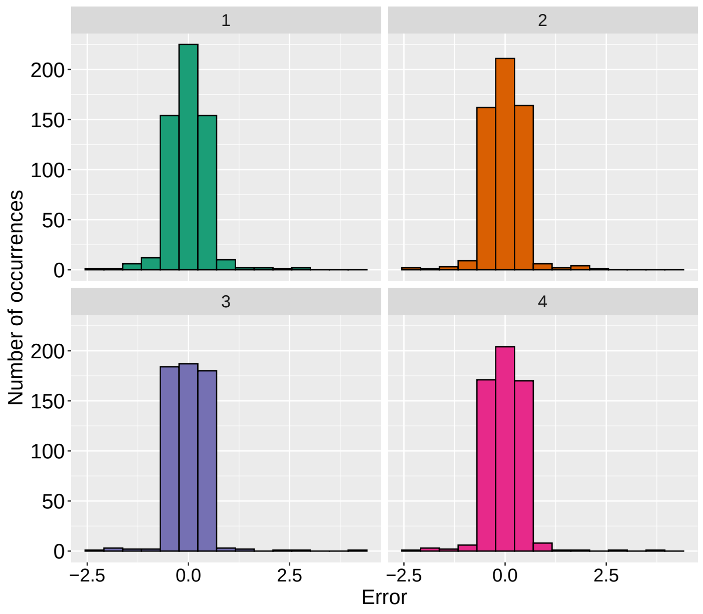


Figure A.1: Analytic PI: Distribution of the tracked error data for 4 random replications.

Then, Figure A.2 showcases the distribution of the tracked error for the Fixed Action - 3 control strategy.

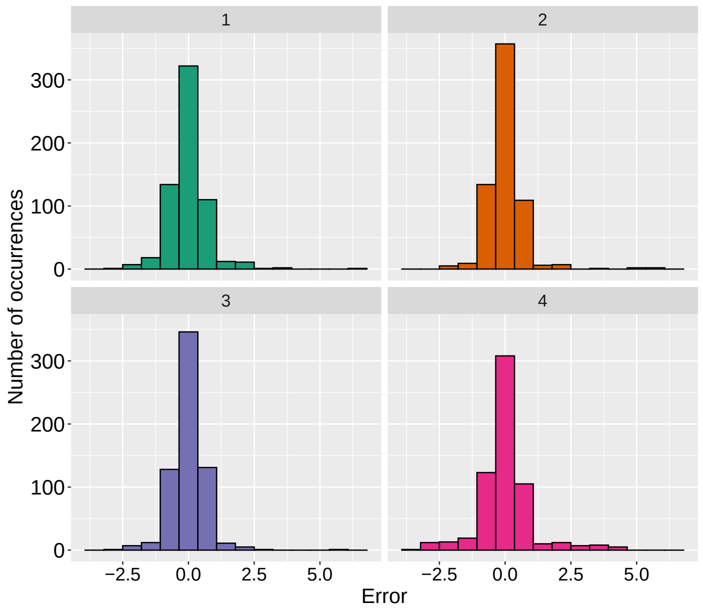


Figure A.2: Fixed Action 3: Distribution of the tracked error data for 4 random replications.

Finally, Figure A.3 showcases the distribution of the tracked error for the Fixed Action -

1 control strategy.

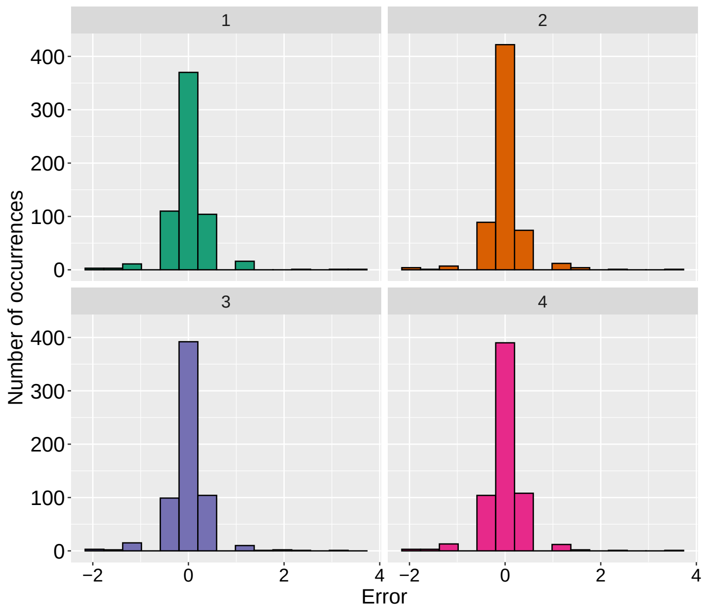


Figure A.3: Fixed Action 1: Distribution of the tracked error data for 4 random replications.

We can see that for the three distributions just presented, the values remain mainly in the bins around 0.0. An initial analysis indicates that the errors are mainly small, and therefore the changes in the system due to the tracked error are also possibly small, but still happening quite frequently, specially for the Fixed Action - 3 approach.

Considering this, next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. This single number is usually called an average of the data. Three popular alternatives to summarize a sample are to specify its mean, median, or mode. These measures are what statisticians call indices of central tendencies.

We are mostly interested here in the mean and median of the observations, since our variables are numerical and they are usually the ones chosen to summarize this type of data. In general lines, the sample mean is obtained by taking the sum of all observations and dividing this sum by the number of observations in the sample. The median is obtained by sorting the observations in an increasing order and taking the observation that is in the middle of the series.

When choosing between this two indexes some things must be taken into consideration. We should define whether the total of all observations is of any interest. If yes, then the mean is a proper index of central tendency. If the total is of no interest, and the histogram that describes the raw data is skewed, the median is more representative of a typical observation than the mean.

It is important to notice that both indexes present some downsides. For instance, the mean is affected more by outliers than the median. A single outlier can make a considerable change in the mean, this being particularly true for small samples. Meanwhile, the median is resistant to several outlying observations. A good aspect of the mean is that it gives equal weight to each observation and in this sense makes full use of the sample. On the other hand, the median ignore a lot of the information presented by the data.

Considering this, we decided to summarize the data using the mean index for the initial analysis. Figure A.4 showcases a histogram of the mean values for the tracked error data for the Analytic PI, Fixed Action 1 and Fixed Action 3 configuration.

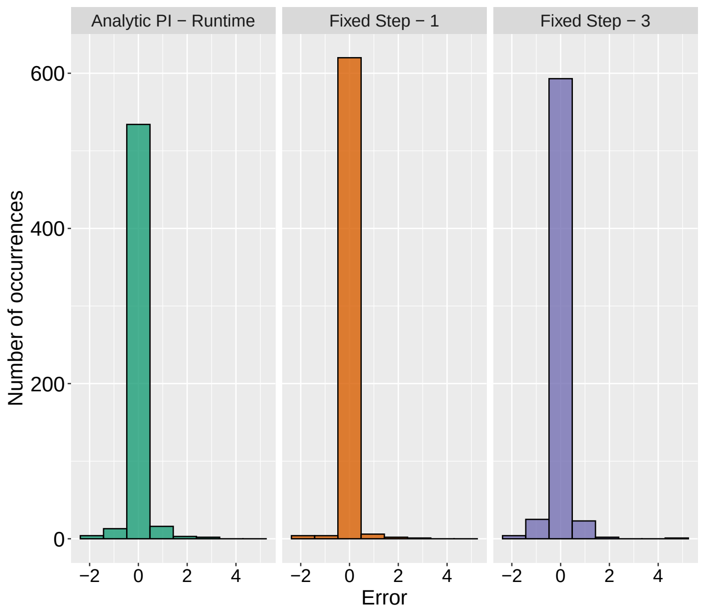


Figure A.4: Analytic PI X Fixed Action 1 x Fixed Action 3: Histogram distribution of the averaged tracked error data for each treatment.

Another way to observe the distribution of the summarized data is shown in Figure A.5, which presents a boxplot graph to highlight the interquartile range of this data. Quartiles divide the data into four parts at 25, 50, and 75%. Thus, 25% of the observations are less than or equal to the first quartile Q1, 50% of the observations are less than or equal to the second quartile Q2, and 75% are less than or equal to the third quartile Q3. Notice that the second quartile Q2 is also the median.

For the tracked error variable in this scenario we visually see almost no difference between the boxes, which tends to mean that the data has a small variability. On the other hand, a lot of points are placed outside of the boxes, which possibly represent a few outliers and/or some extent of significant differences between the data values.

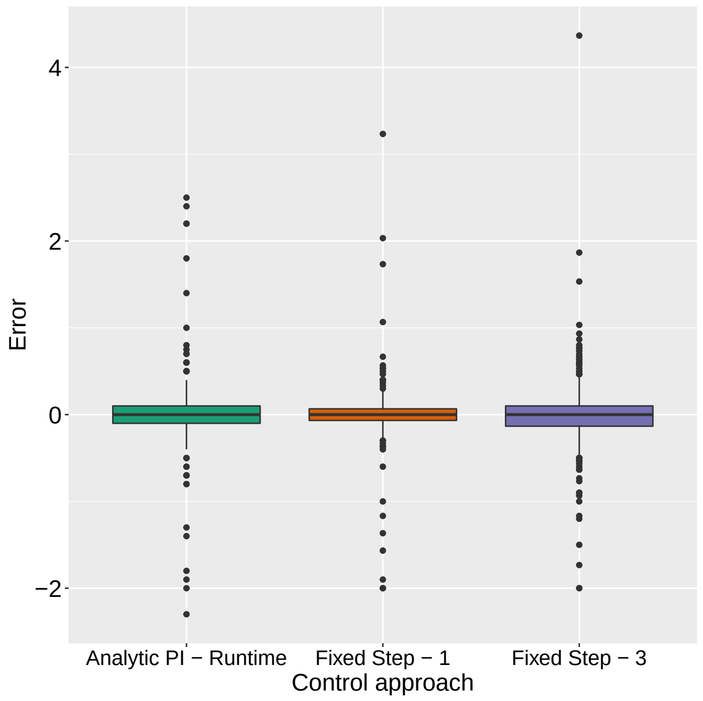


Figure A.5: Analytic PI X Fixed Action 1 x Fixed Action 3: Boxplot distribution of the averaged tracked error data for each treatment.

A.1.2 Replica allocation

For the replica allocation variable we followed the same premises defined for the tracked error descriptive analysis for this scenario. First, we present the data of 4 random replications of the experiment, for each treatment, to showcase how the data spreads before summarizing it by a single number.

Figure A.6 showcases the distribution of the replica allocation for the Analytic PI control strategy. We can see that the distribution for this scenario varies between replications, but mainly stay in the 5 to 8 area.

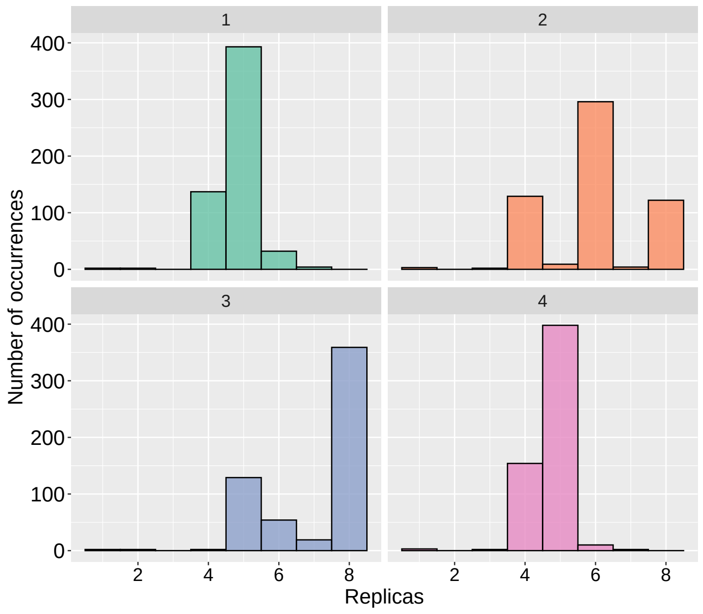


Figure A.6: Analytic PI: Distribution of the replica allocation data for 4 random replications.

Then, Figure A.7 showcases the distribution of the replica allocation for the Fixed Action - 3 control strategy. For this variable, the distribution is much more sparse, varying from 0 to 12, which possibly indicates a high variability of the data.

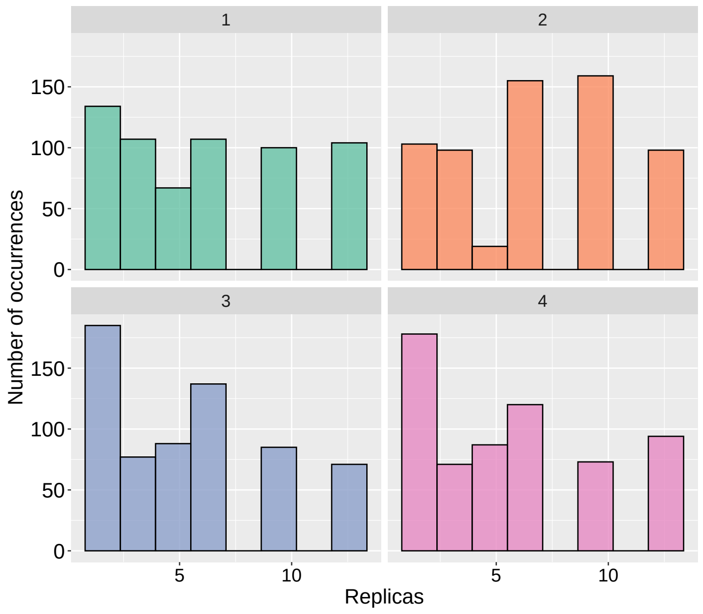


Figure A.7: Fixed Action 3: Distribution of the replica allocation data for 4 random replications.

Finally, Figure A.7 showcases the distribution of the replica allocation for the Fixed

Action - 1 control strategy. In this case, data is still more sparse than the Analytic PI one, but less than the observed for the Fixed Action - 3 configuration.

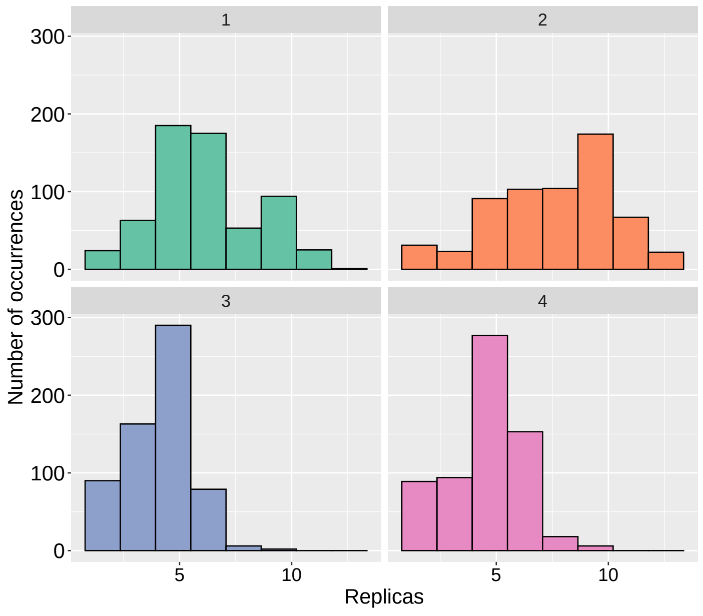


Figure A.8: Fixed Action 1: Distribution of the replica allocation data for 4 random replications.

Considering this, next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. We decided to summarize the data using the mean index for the initial analysis. Figure A.9 showcases a histogram of the mean values for the replica allocation data for the Analytic PI, Fixed Action 1 and Fixed Action 3 configuration.

Comparatively, the Analytic PI approach remains varying less among the three scenarios. For the Fixed Action options, they both present a somewhat sparse histogram, being the Fixed Action - 1 more centered around the $6 - 7$ bin.

Another way to observe the distribution of the summarized data is shown in Figure A.10, which presents a boxplot graph to highlight the interquartile range of this data. For the replica allocation variable in this scenario we visually see the Analytic PI box is shorter than the other two, confirming a less variant data. For the Fixed Action - 3 configuration, which appears to be the larger box, we see more points are placed outside of the boxes, which possibly represent a few outliers. We also notice that the median lines for the Fixed Action - 1 and 3 cross each others boxes, which could mean it would be difficult to determine a statistically significant difference between them.

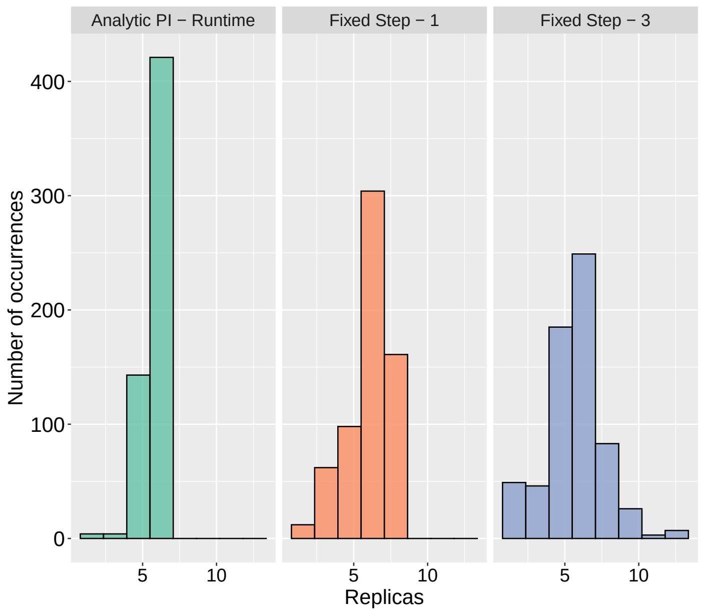


Figure A.9: Analytic PI X Fixed Action 1 x Fixed Action 3: Histogram distribution of the averaged replica allocation data for each treatment.

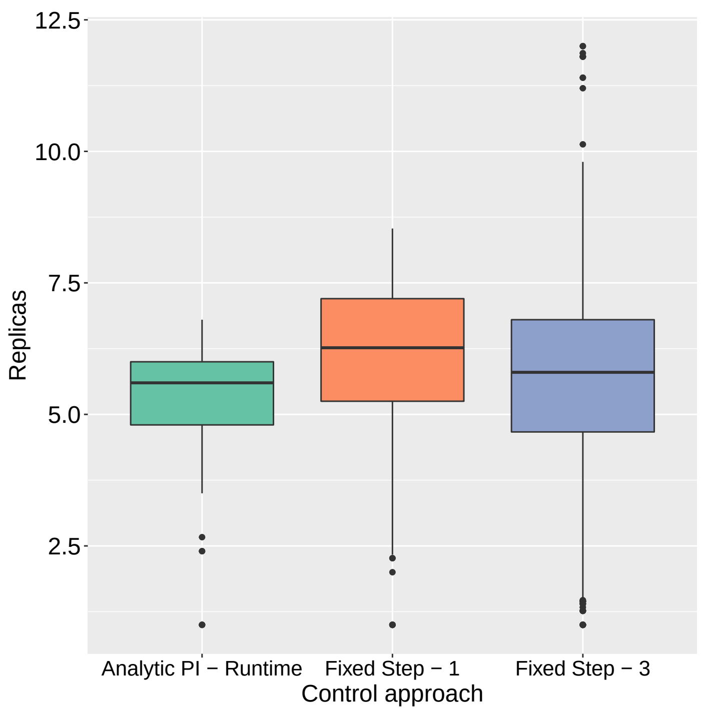


Figure A.10: Analytic PI X Fixed Action 1 x Fixed Action 3: Boxplot distribution of the averaged replica allocation data for each treatment.

A.1.3 Response time

For the response time variable we monitor the time an item stays in the system until it is finally processed. The descriptive analysis for this scenario considers the average of what we call *item duration* for 4 random replications of the experiments for each treatment, to showcase how the data spreads.

Figure A.11 showcases the distribution of the response time in average for the Analytic PI, Fixed Action - 1 and Fixed Action - 3 control strategies.

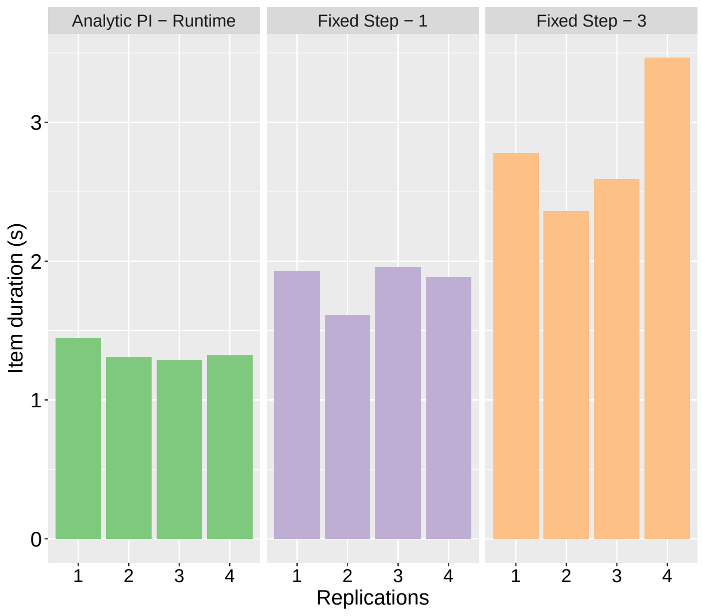


Figure A.11: Analytic PI X Fixed Action 1 x Fixed Action 3: Distribution of the system response time data for 4 random replications..

We can see that the average time an item stays in the system is possibly less for the Analytic PI scenario, and higher for the Fixed Step - 3 scenario. Further statistical analysis of this data is presented on Section 6.4.2.

A.1.4 Data variability

For further knowledge of data variability, that also supports the observed from the visual analysis on the sections above, Table A.1 presents the variability for the summarized data for each scenario. The analysis considers the tracked error, replica allocation, and response time which is the time an item stays in the system until it is fully processed.

Table A.1: Data variability considering replica allocation, tracked error and system response time for the PI x Fixed Action scenario.

	Replica allocation		**Tracked error**		**Response time**	
	Variance	*S. deviation*	*Variance*	*S. deviation*	*Variance*	*S. deviation*
Analytic PI	0.761	0.872	0.130	0.361	4.08	2.02
Fixed Step - 3	3.90	1.97	0.128	0.358	66.8	8.17
Fixed Step - 1	2.53	1.59	0.0709	0.266	25.7	5.07

A.2 Scenario III: Analytic PI x Manual PI

As described by Table 6.9, two tuning configuration methods are defined for this scenario, one called Analytic PI and the other Manual PI. For both executions we used the proposed Proportional-Integral performance controller, using two tuning configurations, one based on the FOPDT model of the system and the other manually defined. Thus, for the Analytic PI, we consider the values for the proportional and integral gains to be $k_{ip} = 0.0871$ and $k_{pp} = 0.9817$. Then, for the Manual PI, we chose the default values of $k_{ip} = 1.0$ and $k_{pp} = 1.0$. In the graphs, the two treatments are named *Analytic PI - Constant* and *Manual PI - Constant* as a reference to its workload input rate which does not vary over time.

Considering this, in this section we present a summary and distribution of the collected data with regards to the system throughput, represented by the tracked error in this analysis, replica allocation and response time.

A.2.1 Tracked error

First, we present the data for the tracked error variable for the two treatments in question: Analytic PI and Manual PI. Initially we are interested in 4 random replications of the experiment, for each treatment, to showcase how the data spreads before summarizing it by a single number.

Figure A.12 showcases the distribution of the tracked error for the Analytic PI tuning configuration.

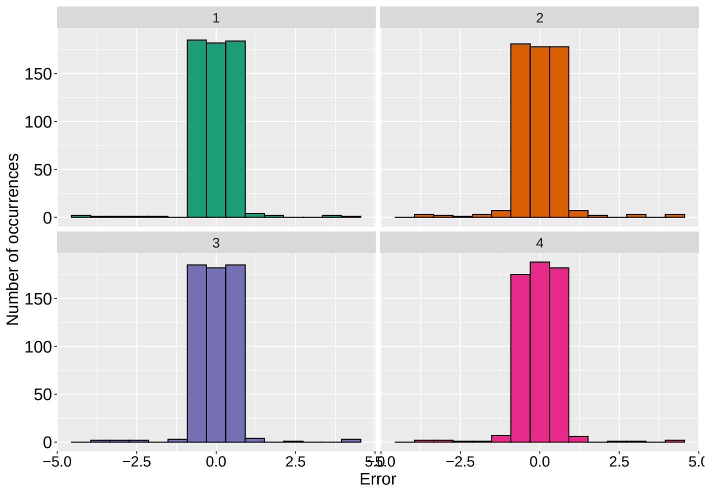


Figure A.12: Analytic PI - Constant: Distribution of the tracked error data for 4 random replications.

Then, Figure A.13 showcases the distribution of the tracked error for the Manual PI tuning configuration.

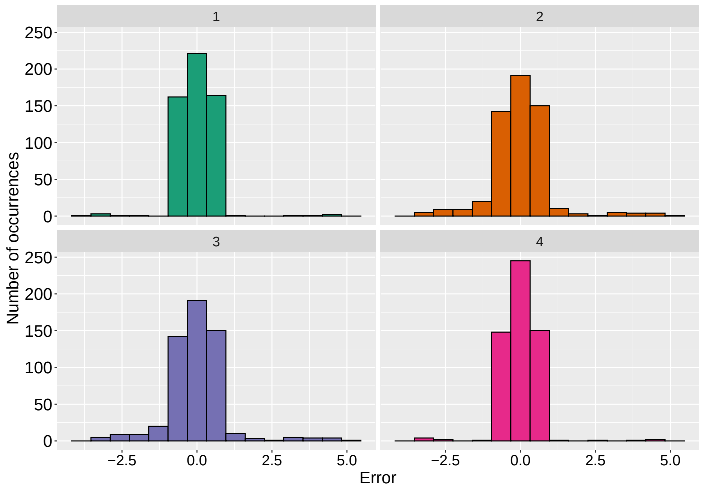


Figure A.13: Manual PI - Constant: Distribution of the tracked error data for 4 random replications.

We can see that for the two distributions just presented, the values remain mainly in the bins around 0.0. For the Analytic PI scenario, the distribution seems almost equally centered at the three bins in the middle, while for the Manual PI one they differ in height, and therefore, in frequency. An initial analysis indicates that, although the errors are mainly small, changes in the system due to the tracked error are still happening quite frequently.

Next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. This single number is usually called an average of the data. We are mostly interested here in the mean and median of the observations, since our variables are numerical and they are usually the ones chosen to summarize this type of data.

Considering this, we decided to summarize the data using the mean index for the initial analysis. Figure A.14 showcases a histogram of the mean values for the tracked error data for the Analytic PI and Manual PI configuration.

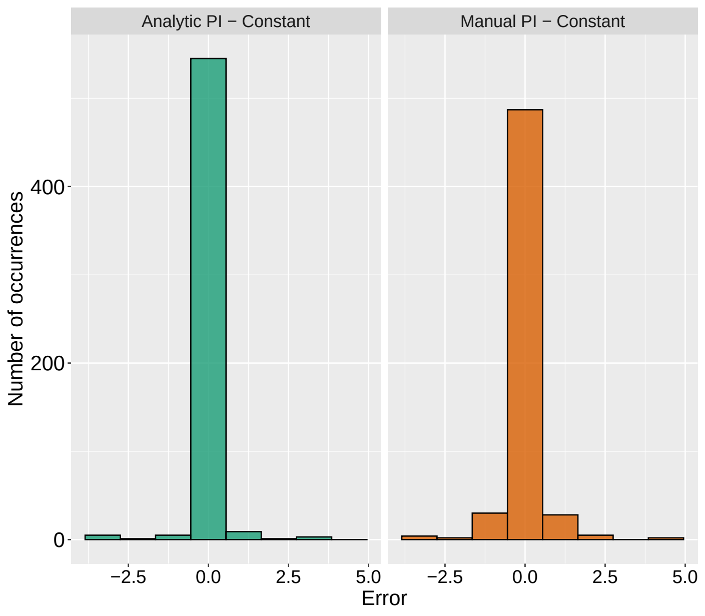


Figure A.14: Analytic PI - Constant X Manual PI - Constant: Histogram distribution of the averaged tracked error data for each treatment.

Another way to observe the distribution of the summarized data is shown in Figure A.15, which presents a boxplot graph to highlight the interquartile range of this data. Notice that the second quartile Q2 in the plot is also the median value.

For the tracked error variable in this scenario we visually see almost no difference between the boxes, which tends to mean that the data has a small variability. On the other hand, a lot of points are placed outside of the boxes, which possibly represent a few outliers and/or some extent of significant differences between the data values. Although the box for the Manual PI approach is larger than the one for the Analytic PI, which could indicate a difference between the two scenarios, the median lines cross each other boxes, which, in its turn, could indicate it would be difficult to determine a statistically significant difference.

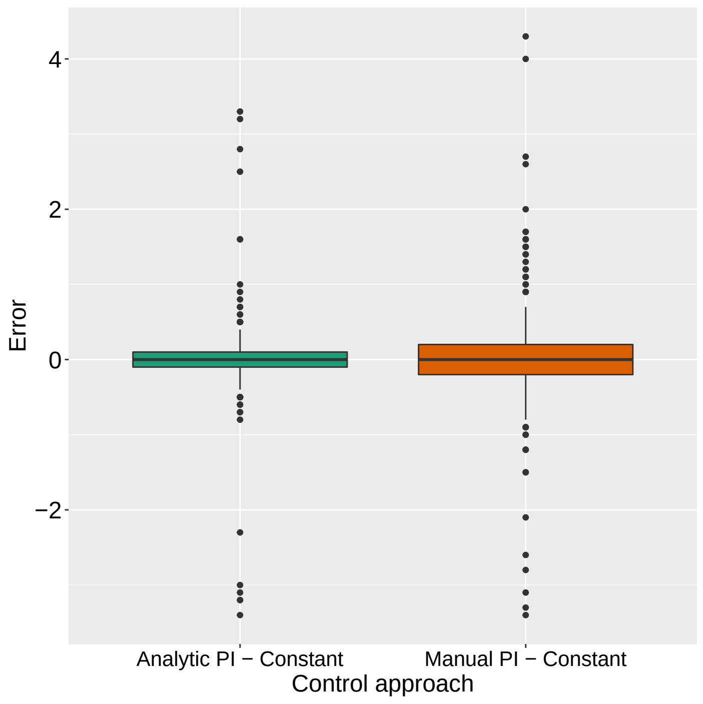


Figure A.15: Analytic PI - Constant X Manual PI - Constant: Boxplot distribution of the averaged tracked error data for each treatment.

A.2.2 Replica allocation

For the replica allocation variable we followed the same premises defined for the tracked error descriptive analysis for this scenario. First, we present the data of 4 random replications of the experiment, for each treatment, to showcase how the data spreads before summarizing it by a single number.

Figure A.16 showcases the distribution of the replica allocation for the Analytic PI configuration tuning. We can see that the distribution for this scenario varies between replications, with replication 3 and 4 centered around the bin of value 4 which is expected for an input rate of 3 items per second.

Then, Figure A.17 showcases the distribution of the replica allocation for the Manual PI configuration tuning. For this variable, the distribution is more centered on the bin of value 8, with some small variance on the other bins for replications 2 and 3.

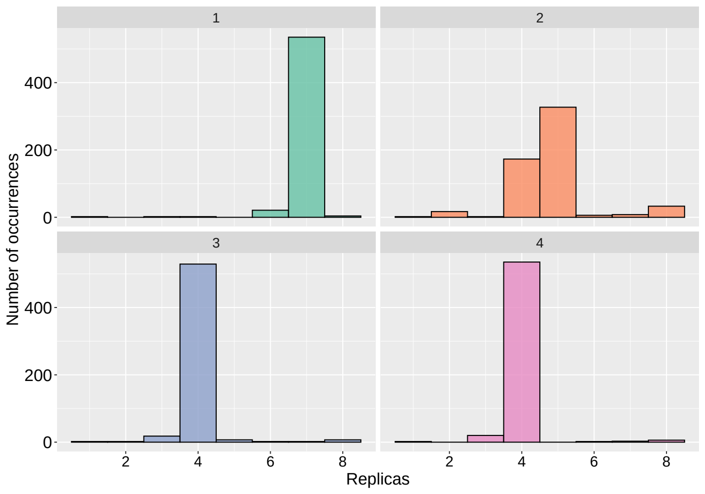


Figure A.16: Analytic PI - Constant: Distribution of the replica allocation data for 4 random replications.

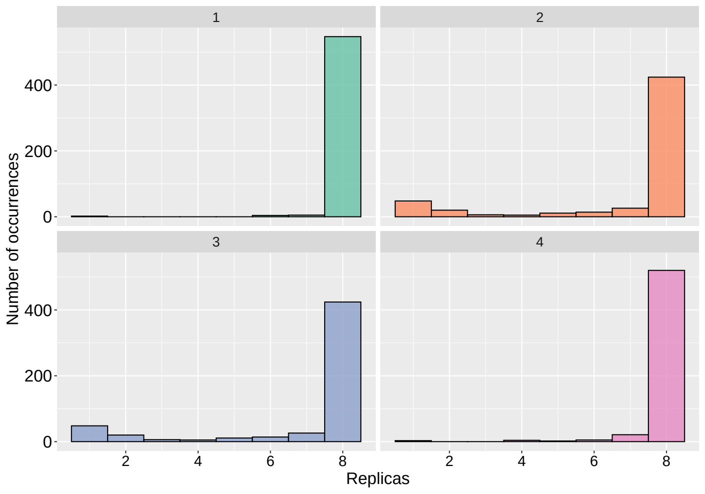


Figure A.17: Manual PI - Constant: Distribution of the replica allocation data for 4 random replications.

Considering this, next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. We decided to summarize the data using the mean index for the initial analysis. Figure A.18 showcases a histogram of the mean values for the replica allocation data for the Analytic PI and Manual PI configuration.

Comparatively, the Analytic PI approach remains varying less among the two scenarios, centering its data around only two bins. For the Manual PI approach, it presents a somewhat sparse histogram, a little skewed to the right.

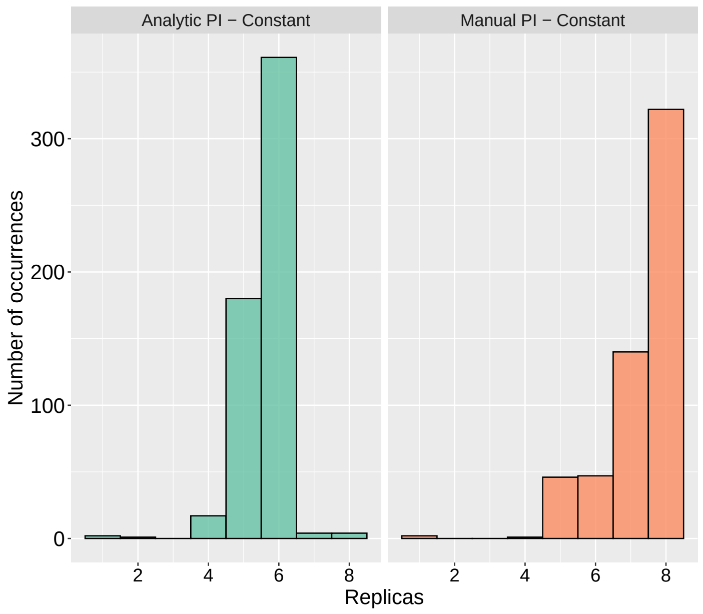


Figure A.18: Analytic PI - Constant X Manual PI - Constant: Histogram distribution of the averaged replica allocation data for each treatment.

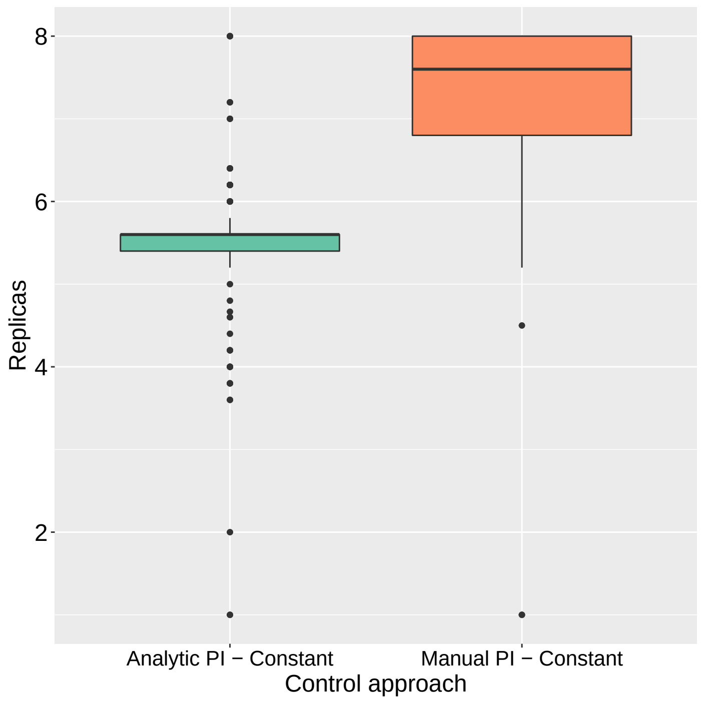


Figure A.19: Analytic PI - Constant X Manual PI - Constant: Boxplot distribution of the averaged replica allocation data for each treatment.

Another way to observe the distribution of the summarized data is shown in Figure A.19, which presents a boxplot graph to highlight the interquartile range of this data. For the replica allocation variable in this scenario we visually see the Analytic PI box is shorter than the Manual PI, possibly confirming a less variant data. We also notice that neither the boxes nor the median lines for the Manual PI and Analytic PI configuration cross each others boxes, which indicates there is a possible statistically significant difference between them.

A.2.3 Response time

For the response time variable we monitor the time an item stays in the system until it is finally processed. The descriptive analysis for this scenario considers the average of what we call *item duration* for 4 random replications of the experiments for each treatment, to showcase how the data spreads.

Figure A.20 showcases the distribution of the response time in average for the Analytic PI and Manual PI tuning configuration options.

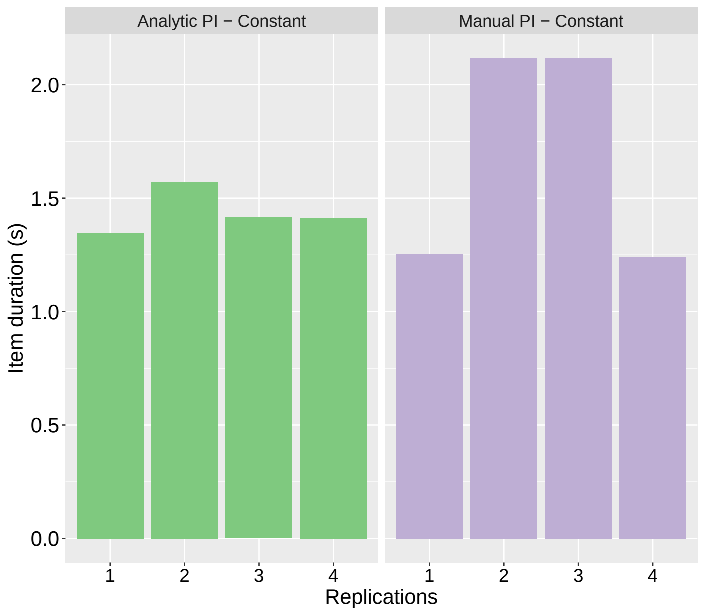


Figure A.20: Analytic PI - Constant X Manual PI - Constant: Distribution of the system response time data for 4 random replications..

We can see that the average time an item stays in the system is possibly less for the Analytic PI scenario, and higher for the Manual PI scenario. Further statistical analysis of this data is presented on Section B.2.2.

A.2.4 Data variability

For further knowledge of data variability, that also supports the observed from the visual analysis on the sections above, Table A.2 presents the variability for the summarized data for each scenario. The analysis considers the tracked error, replica allocation, and response time which is the time an item stays in the system until it is fully processed.

Table A.2: Data variability considering replica allocation, tracked error and system response time for the Analytic PI x Manual PI scenario.

	Replica allocation		**Tracked error**		**Response time**	
	Variance	*S. deviation*	*Variance*	*S. deviation*	*Variance*	*S. deviation*
Analytic PI	0.267	0.517	0.218	0.467	5.87	2.42
Manual PI	0.965	0.982	0.354	0.595	19.4	4.41

A.3 Scenario IV: PI - Runtime x PI - Estimated

As described by Table 6.11, two throughput and, as a consequence, tracked error calculation methods are defined for this scenario, one called PI - Runtime and the other PI - Estimated. For both executions we used the proposed Proportional-Integral performance controller, using two different methods to calculate the throughput, one measured at runtime, and the other as an estimate based on the number of replicas in use and the time it takes to process a given item. In the graphs, the two treatments are named *Analytic PI - Runtime* and *Analytic PI - Estimated.*

Considering this, in this section we present a summary and distribution of the collected data with regards to the system throughput, represented by the tracked error in this analysis, replica allocation and response time. Note that, for this scenario, we only present the individual data regarding the PI - Estimated configuration. This is because the PI - Runtime configuration is the same sample data as described in A.1 for all the variables of interest.

A.3.1 Tracked error

First, we present the data for the tracked error variable for the PI - Estimated configuration. Initially we are interested in 4 random replications of the experiment to showcase how the data spreads before summarizing it by a single number.

Figure A.21 showcases the distribution of the tracked error for the PI - Estimated configuration.

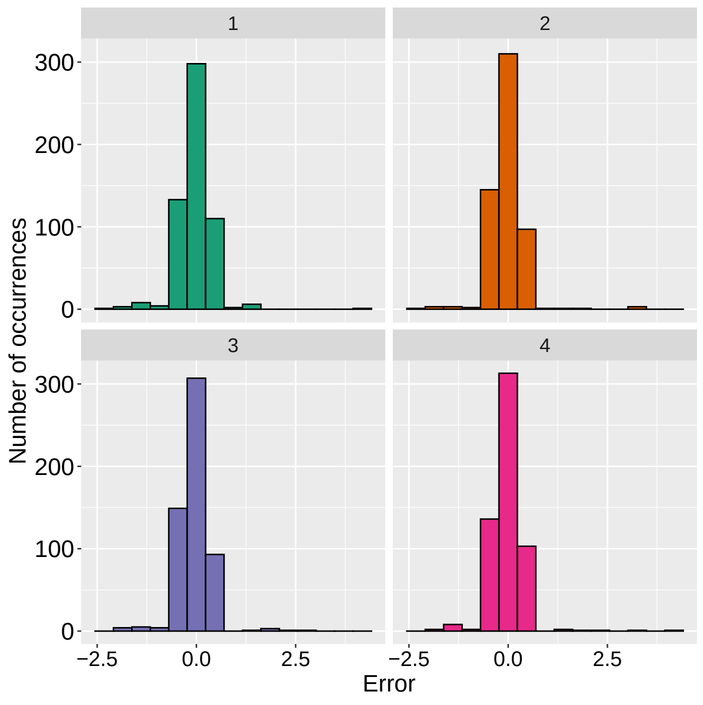


Figure A.21: Analytic PI - Estimated: Distribution of the tracked error data for 4 random replications.

We can see that for the distribution just presented, the values remain mainly in the bins around 0.0. An initial analysis indicates that, although the errors are mainly small, changes in the system due to the tracked error are still happening quite frequently.

Next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. This single number is usually called an average of the data. We are mostly interested here in the mean and median of the observations.

Considering this, we decided to summarize the data using the mean index for the initial analysis. Figure A.22 showcases a histogram of the mean values for the tracked error data for the PI - Runtime and PI - Estimated configuration.

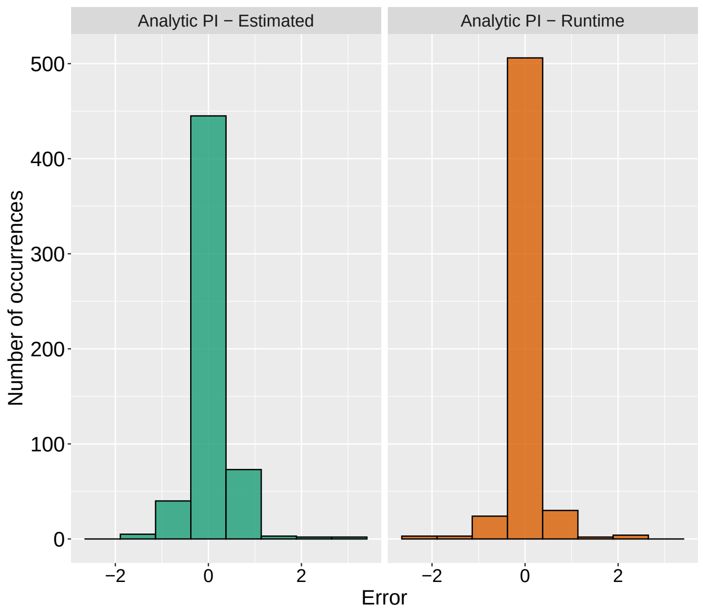


Figure A.22: Analytic PI - Runtime X Analytic PI - Estimated: Histogram distribution of the averaged tracked error data for each treatment.

Another way to observe the distribution of the summarized data is shown in Figure A.23.

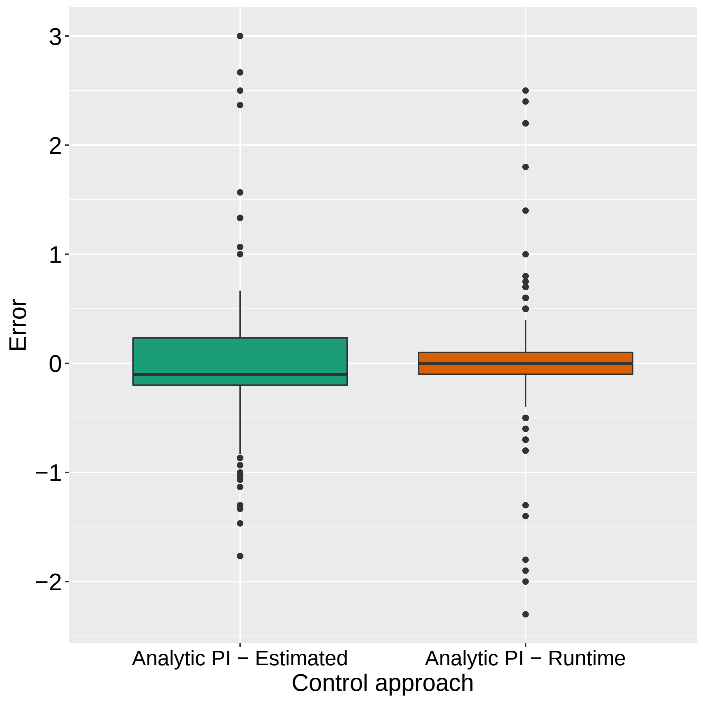


Figure A.23: Analytic PI - Runtime X Analytic PI - Estimated: Boxplot distribution of the averaged tracked error data for each treatment.

For the tracked error variable in this scenario we see that the PI - Estimated box is larger than the PI - Runtime one, which could indicate the first one varies more over time. Besides that, a lot of points are placed outside of the boxes, which possibly represent a few outliers and/or some extent of significant differences between the data values. Although the box for the PI - Estimated approach is larger than the one for the PI - Runtime, which could indicate a difference between the two scenarios, the median lines cross each other boxes, which, in its turn, could indicate it would be difficult to determine a statistically significant difference.

A.3.2 Replica allocation

For the replica allocation variable we followed the same premises defined for the tracked error descriptive analysis for this scenario. First, we present the data of 4 random replications of the experiment to showcase how the data spreads before summarizing it by a single number.

Figure A.24 showcases the distribution of the replica allocation for the PI - Estimated configuration.

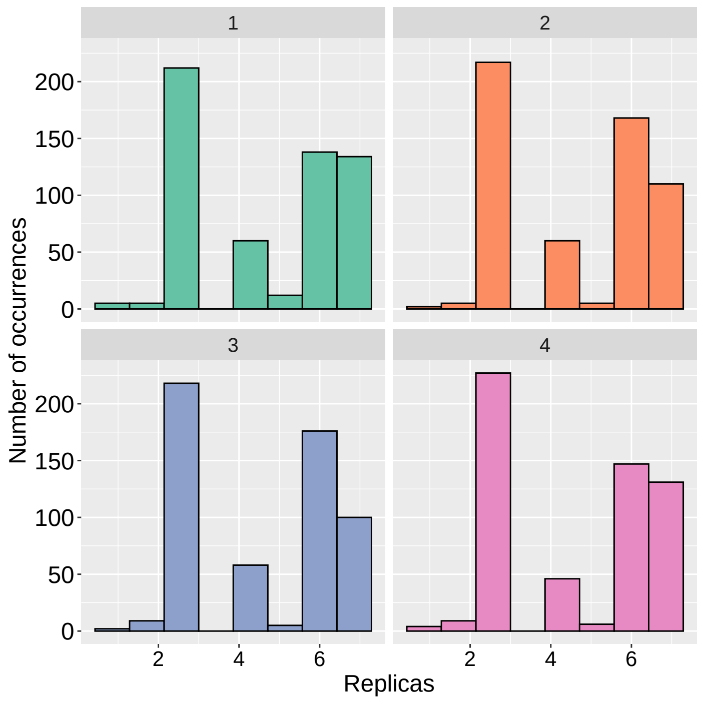


Figure A.24: Analytic PI - Estimated: Distribution of the replica allocation data for 4 random replications.

We can see that the distribution for this scenario varies between replications, more concentrated in the bins around the 3 and 6 bins, which possibly reflects the workload arrival rate of new items which varies between 2 and 4 items per second.

Considering this, next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. We decided to summarize the data using the mean index for the initial analysis.

Figure A.25 showcases a histogram of the mean values for the replica allocation data for the PI - Runtime and PI - Estimated configuration. Comparatively, the PI - Runtime approach remains varying less among the two scenarios, centering its data around only two bins, mainly the number 6. For the PI - Estimated approach, data is more sparse meaning the replica allocation varied more. Since the use of this approach intended to mitigate or at least decrease resource over-provisioning, this could be an indication that replica allocation is better adapting to the varied workload in this scenario.

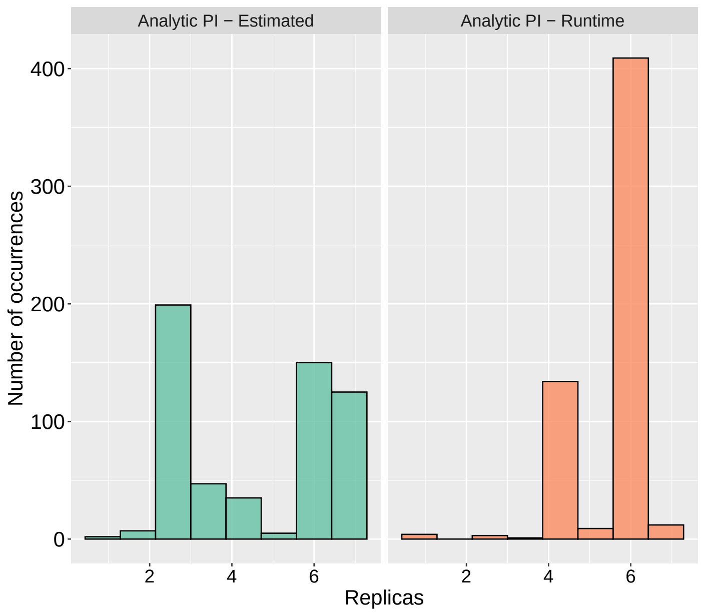


Figure A.25: Analytic PI - Runtime X Analytic PI - Estimated: Histogram distribution of the averaged replica allocation data for each treatment.

Another way to observe the distribution of the summarized data is shown in Figure A.26, which presents a boxplot graph to highlight the interquartile range of this data. For the replica allocation variable in this scenario we visually see the PI - Runtime box is shorter than the Manual PI, possibly confirming a less variant data. We also notice that the median line of the PI - Runtime configuration cross the PI - Estimated box, which could indicate a possible difficulty in defining if the two are statistically significant different.

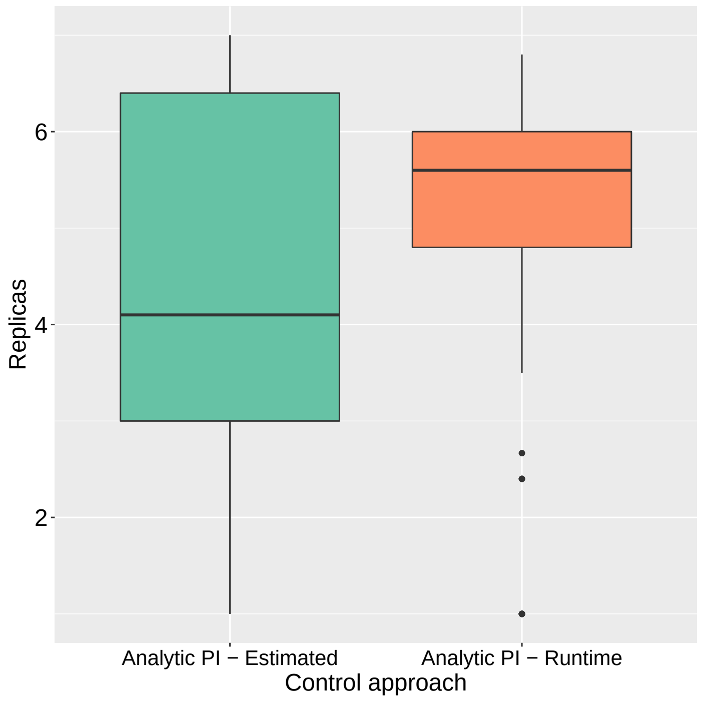


Figure A.26: Analytic PI - Runtime X Analytic PI - Estimated: Boxplot distribution of the averaged replica allocation data for each treatment.

A.3.3 Response time

For the response time variable we monitor the time an item stays in the system until it is finally processed. The descriptive analysis for this scenario considers the average of what we call *item duration* for 4 random replications of the experiments for each treatment, to showcase how the data spreads.

Figure A.27 showcases the distribution of the response time in average for the PI - Runtime and PI - Estimated configuration options. We can see that the average time an item stays in the system is possibly less for the PI - Runtime scenario, and higher for the PI - Estimated scenario.

Although we see that initial data indicates the PI - Estimated approach performs better when allocating a rightful amount of replicas according to the workload, instantiating new resources more frequently, especially when the workload arrival rate increases, could cause a little delay on the processing of new requests. This could explain why this configuration apparently performs worse when it comes to response time. On the other hand, the PI - Runtime configuration could be biased by a larger amount of replicas than the necessary, which could also explain why it performs better in this case.

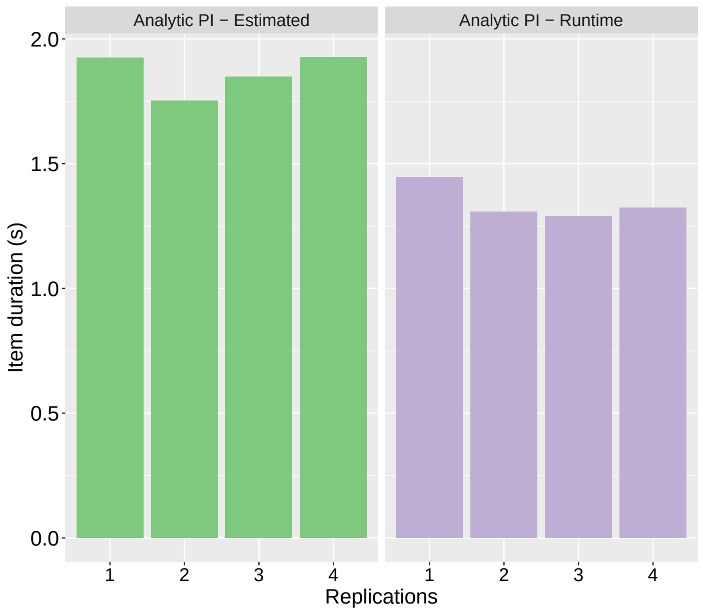


Figure A.27: Analytic PI - Runtime X Analytic PI - Estimated: Distribution of the system response time data for 4 random replications..

Further statistical analysis of this data is presented on Section B.3.2.

A.3.4 Data variability

For further knowledge of data variability, that also supports the observed from the visual analysis on the sections above, Table A.3 presents the variability for the summarized data for each scenario. The analysis considers the tracked error, replica allocation, and response time which is the time an item stays in the system until it is fully processed.

Table A.3: Data variability considering replica allocation, tracked error and system response time for the PI - Runtime x PI - Estimated scenario.

	Replica allocation		**Tracked error**		**Response time**	
	Variance	*S. deviation*	*Variance*	*S. deviation*	*Variance*	*S. deviation*
PI - Runtime	0.761	0.872	0.130	0.361	4.08	2.02
PI - Estimated	2.85	1.69	0.175	0.418	7.84	2.80

A.4 Scenario V: SIMO x Independent control

This set of experiments collects data regarding two different controllers, one called *SIMO PI* that combines the individual actions of the cost and performance controllers into a general control action, and the other a more simplistic approach called *Independent*, that uses independent controllers acting in different moments, with the performance one being the most frequent. Such evaluation takes into consideration the system throughput, resource utilization, response time and total execution cost. For the SIMO PI approach, different preferences for each controller are evaluated, which are defined as $\alpha = 0.2$ when there is a preference for cost metrics, $\alpha = 0.5$ for no preference in particular, and $\alpha = 0.8$ favoring performance metrics.

Considering this, in this section we present a summary and distribution of the collected data with regards to the system throughput, represented by the tracked error in this analysis, replica allocation, response time and execution cost.

A.4.1 Tracked error

First, we present the data for the tracked error variable for the performance controller that acts within the both approaches considered here. We selected 1 random replication of the experiment, for each control preference, to showcase how the data spreads before summarizing it by a single number.

Figure A.28 showcases the distribution of the performance tracked error for the SIMO PI and Independent tuning configuration.

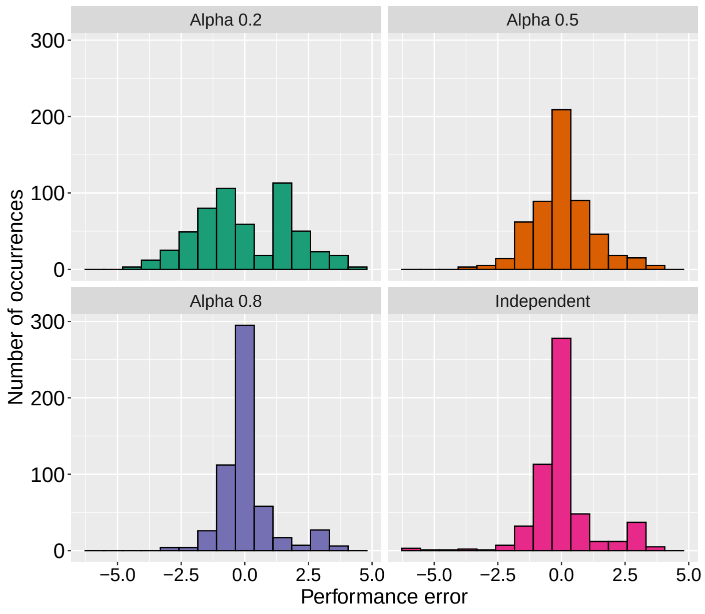


Figure A.28: SIMO PI x Independent: Distribution of the performance tracked error data for 1 random replication.

We can see that for the four distributions just presented, the values are pretty spread around the bins, specially for the Alpha 0.2 configuration. For the Alpha 0.5 scenario, the distribution seems almost equally distributed around the bins, while for the Alpha 0.8 preference, the values are more concentrated around the 0.0 bin. An initial analysis indicates that, although the errors are mainly small, changes in the system due to the performance tracked error are still happening quite frequently.

Next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. This single number is usually called an average of the data. We are mostly interested here in the mean and median of the observations, since our variables are numerical and they are usually the ones chosen to summarize this type of data.

Considering this, we decided to summarize the data using the mean index for the initial analysis. Figure A.29 showcases a histogram of the mean values for the performance tracked error data for the proposed scenarios. Another way to observe the distribution of the summarized data is shown in Figure A.30, which presents a boxplot graph to highlight the interquartile range of this data. Notice that the second quartile Q2 in the plot is also the median value.

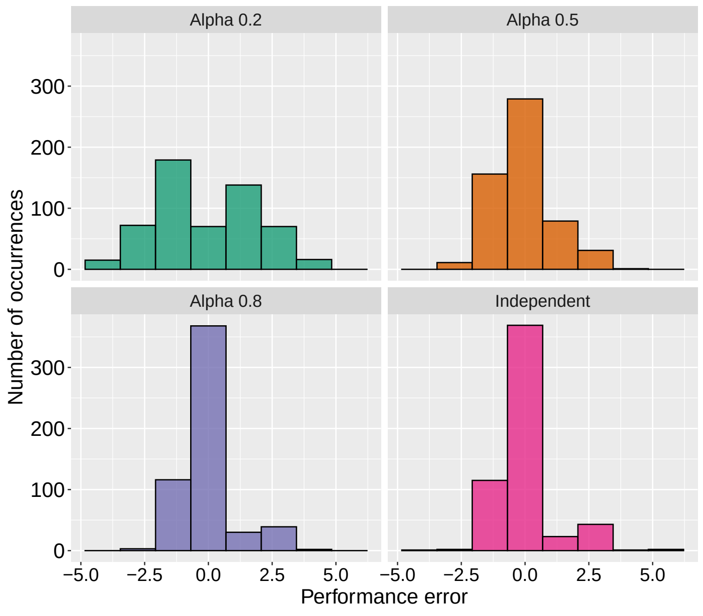


Figure A.29: SIMO PI x Independent: Histogram distribution of the averaged performance tracked error data for each control preference.

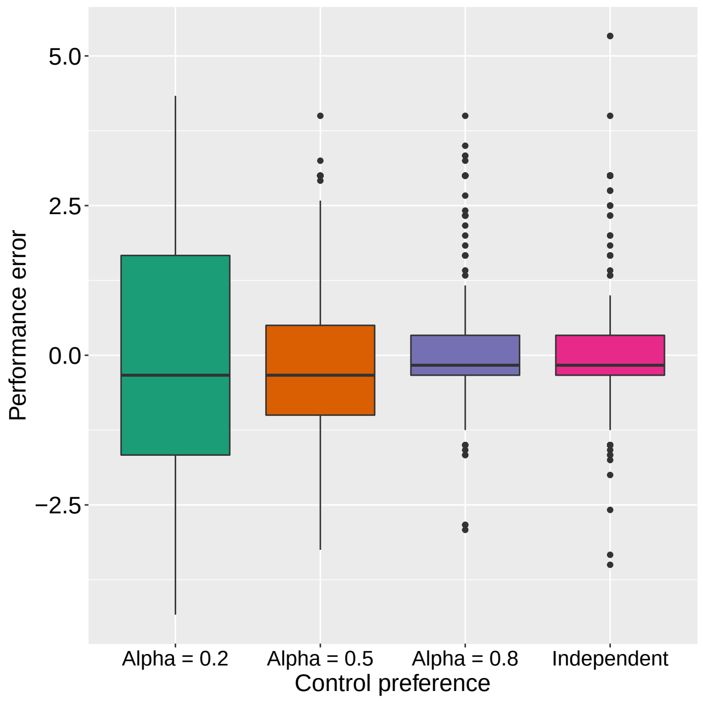


Figure A.30: SIMO PI x Independent: Boxplot distribution of the averaged performance tracked error data for each control preference.

For the performance tracked error variable in this scenario we visually see some differences between the boxes, which tends to mean that the data has a considerate variability. On the other hand, a lot of points are placed outside of the Alpha 0.8 and Independent boxes, which possibly represent a few outliers and/or some extent of significant differences between the data values. Although the box for the Alpha 0.2 approach is larger than the other ones, which could indicate a difference between the scenarios, the median lines cross the other boxes, which, in its turn, could indicate it would be difficult to determine a statistically significant difference.

A.4.2 Replica allocation

For the replica allocation variable we followed the same premises defined for the tracked error descriptive analysis for this scenario. First, we present the data of 1 random replication of the experiment, for each control preference, to showcase how the data spreads before summarizing it by a single number.

Figure A.31 showcases the distribution of the performance tracked error for the SIMO PI and Independent tuning configuration.

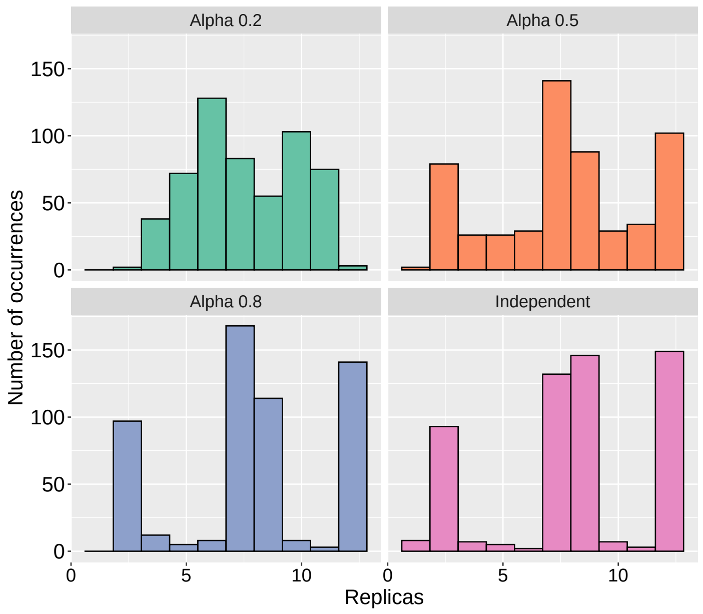


Figure A.31: SIMO PI x Independent: Distribution of the replica allocation data for 1 random replication.

Next, we want to summarize the data from different replications by a single number, for each piece of data collected, in our case, every 2 seconds. We decided to summarize the data using the mean index for the initial analysis.

Figure A.32 showcases a histogram of the mean values for the replica allocation data for the proposed scenarios.

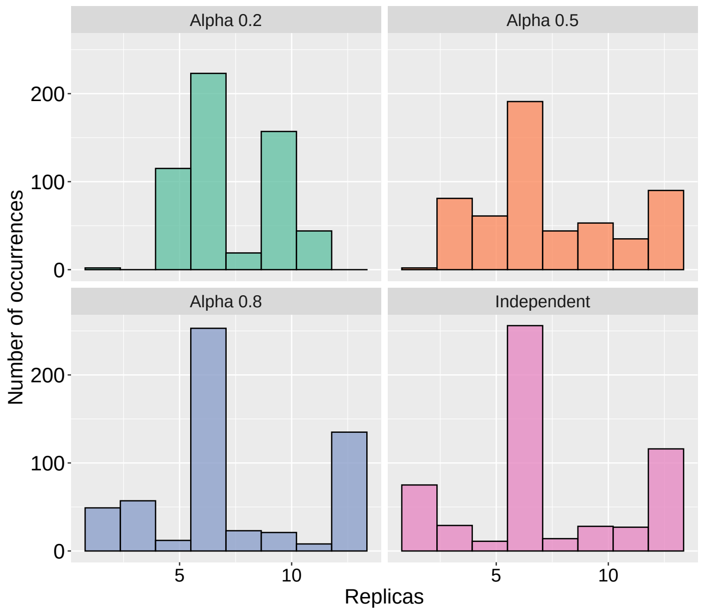


Figure A.32: SIMO PI x Independent: Histogram distribution of the averaged replica allocation data for each control preference.

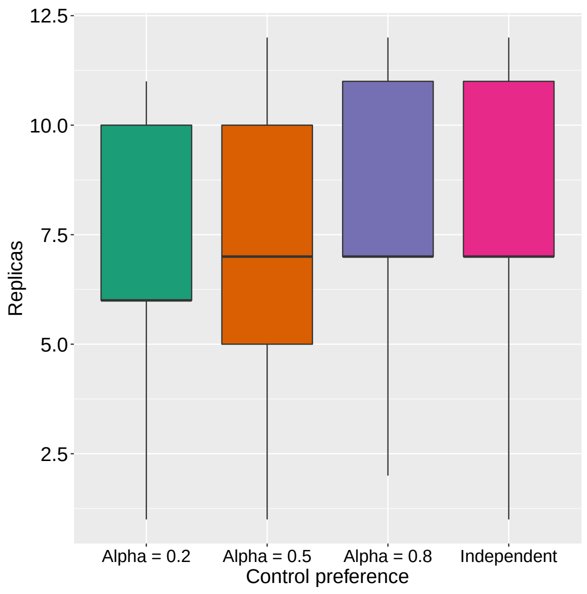


Figure A.33: SIMO PI x Independent: Boxplot distribution of the averaged replica allocation data for each control preference.

Another way to observe the distribution of the summarized data is shown in Figure A.33, which presents a boxplot graph to highlight the interquartile range of this data. We can see that the boxes are large for all the scenarios, which possibly indicates a high data variability. From the intersection of the boxes, we also anticipate that it might be difficult to determine a statistically significant differences between the scenarios.

A.4.3 Response time

For the response time variable we monitor the time an item stays in the system until it is finally processed. The descriptive analysis for this scenario considers the average of what we call *item duration* for 5 random replications of the experiments for each control preference, to showcase how the data spreads.

Figure A.34 showcases the distribution of the response time in average for the SIMO PI and Independent configuration.

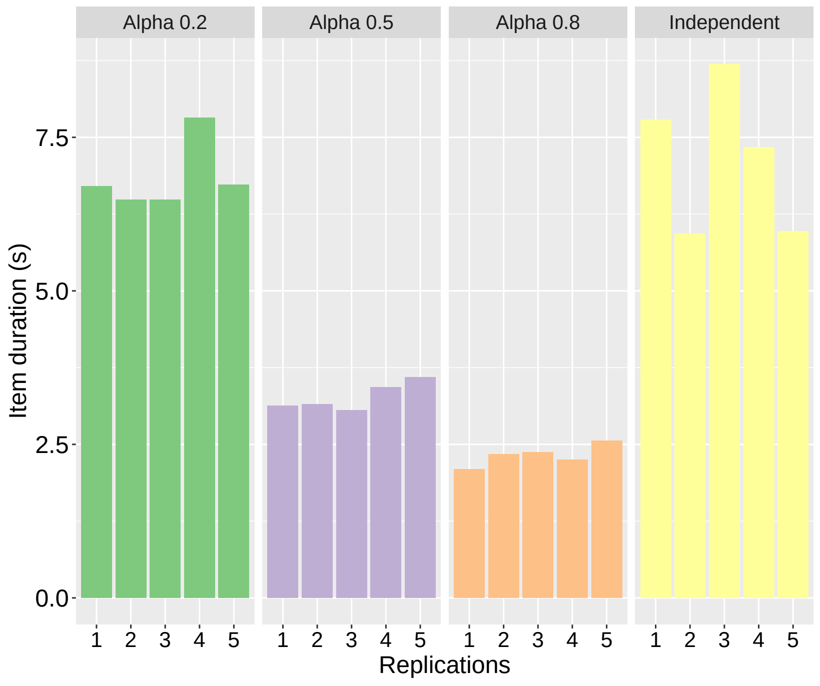


Figure A.34: SIMO PI x Independent: Distribution of average system response time data for 5 random replications.

We can see that the average time an item stays in the system is possibly less for the Alpha 0.8 scenario, and higher for the Alpha 0.2 and Independent scenarios. Further statistical analysis of this data is presented on Section B.4.3.

A.4.4 Data variability

For further knowledge of data variability, that also supports the observed from the visual analysis on the sections above, Table A.2 presents the variability for the summarized data for each scenario. The analysis considers the tracked error, replica allocation, and response time which is the time an item stays in the system until it is fully processed.

Table A.4: Data variability considering replica allocation, tracked error, total cost and system response time for the SIMO PI x Independent scenario.

	Replicas		**Error (Perf.)**		**Error (Cost)**		**Total cost**		**Resp. time**	
	Var	*Sd*	*Var*	*Sd*	*Var*	*Sd*	*Var*	*Sd*	*Var*	*Sd*
$\alpha = 0.2$	4.89	2.21	3.70	1.92	0.0000207	0.00455	0.0478	0.219	40.2	6.34
$\alpha = 0.5$	8.87	2.98	1.40	1.18	0.0000514	0.00717	0.00296	0.0544	20.7	4.55
$\alpha = 0.8$	10.3	3.21	10.7	10.3	0.0000570	0.00755	0.0415	0.204	13.0	3.61
Indep.	10.7	3.27	1.25	1.12	0.0000557	0.00747	0.277	0.527	125	11.2

Appendix B

Statistical analysis of experimentation data

B.1 Scenarios I and II: Proportional-Integral x Fixed Action

Considering the same scenario presented in Appendix A, Section A.1, the statistical analysis of the experimentation data showcases the following results.

B.1.1 Replica allocation

Table B.1 showcases the observations and hypothesis about the allocation of replicas for the PI x Fixed Action - 3 scenario.

Table B.1: Replica allocation statistical observations for the PI x Fixed Action - 3 scenario.

Observation from the data:	The average replica allocation for the PI controller is lesser than that for the Fixed Action controller of size 3.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average replica allocation in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Fixed Action - 3 is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant. In general lines, considering a confidence interval of 95%, if the *p-value* is less than 0.05 then you can reject the null hypothesis and conclude that the difference between means in the two categories is statistically significant. Note that this same principle will be used for all the tests performed using the Student's distribution.

The results show a *p-value* of 0.002128, with a 95% confidence interval of $[-0.43220276, -0.09581768]$. The sample estimate for the mean in group Analytic PI is 5.41690, while the mean for the Fixed Action - 3 group is 5.68091, which is higher. Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

Now considering the configuration of an actuation size of 1 for the Fixed Action controller, Table B.2 showcases the observations and hypothesis about the allocation of replicas for the PI x Fixed Action - 1 scenario.

Table B.2: Replica allocation statistical observations for the PI x Fixed Action - 1 scenario.

Observation from the data:	The average replica allocation for the PI controller is lesser than that for the Fixed Action controller of size 1.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average replica allocation in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Fixed Action - 1 is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for this case as well as for the Fixed Step - 3 treatment. The results show a *p-value* $< 2.2e-16$, with a 95% confidence interval of $[-0.8160899, -0.5303130]$. The sample estimate for the mean in group Analytic PI is 5.41690, while the mean for the Fixed Action - 1 group is 6.090101, which is higher. Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are also statistically significant.

B.1.2 Tracking of the reference value

Table B.3 showcases the observations and hypothesis about the tracking of the reference value for the PI x Fixed Action - 3 scenario. Note that, for this system, the reference value is determined by the input rate of new work items, which vary between 2 and 4 items per second. Thus, in order to quantify how well the system is following the reference value, we take into consideration the tracked error, which is equal to 0 when the system processes the same amount of incoming items, i.e. follows the reference value, and different than 0 otherwise.

Table B.3: Tracked error statistical observations for the PI x Fixed Action - 3 scenario.

Observation from the data:	The average tracked error for the PI controller is greater than that for the Fixed Action controller of size 3.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average tracked error in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Fixed Action - 3 is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for this case. The results show a *p-value* of 0.9557, with a 95% confidence interval of $[-0.03930561, 0.04159426]$. The sample estimate for the mean in group Analytic PI is 0.0017773893, while the mean for the Fixed Action - 3 group is 0.0006330656.

Considering these results, we can not reject the null hypothesis that the average tracked error in each group is the same. We mostly conclude that because the *p-value* is very high considering the expected standards of < 0.05, and the confidence interval includes a 0 difference, which means that there is still a possibility that the average means in the two observed groups are actually the same.

Now considering the configuration of an actuation size of 1 for the Fixed Action controller, Table B.4 showcases the observations and hypothesis about the tracking of the reference value for the PI x Fixed Action - 1 scenario.

Moreover, a Student's *t-test* was also performed to analyze if the difference observed from data is statistically significant. The results show a *p-value* of 0.9743, with a 95% con-

Table B.4: Tracked error statistical observations for the PI x Fixed Action - 1 scenario.

Observation from the data:	The average tracked error for the PI controller is greater than that for the Fixed Action controller of size 1.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average tracked error in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Fixed Action - 1 is not equal to 0.

fidence interval of $[-0.03551775, 0.03670456]$. The sample estimate for the mean in group Analytic PI is 0.001777389, while the mean for the Fixed Action - 1 group is 0.001183984. Considering these results, for the same reasons as described for the Fixed Action - 3 treatment, we also can not reject the null hypothesis, which means the differences in means between the two groups are not statistically significant considering this sample.

B.2 Scenario III: Analytic PI x Manual PI

Considering the same scenario presented in Appendix A, Section A.2, the statistical analysis of the experimentation data showcases the following results.

B.2.1 Tracking of the reference value

Table B.5 showcases the observations and hypothesis about the tracking of the reference value for the Analytic PI x Manual PI scenario. Remember that, for this system, the reference value is determined by the input rate of new work items, which for this case is a constant of 3 items per second. Thus, in order to quantify how well the system is following the reference value, we take into consideration the tracked error based on this input rate.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for this case. The results show a *p-value* of 0.9865, with a 95% confidence interval of $[-0.06312582, 0.06204362]$. The sample estimate for the mean in group Analytic PI is 0.0001757469, while the mean for the Manual PI group is 0.0007168459.

Considering these results, we can not reject the null hypothesis the average tracked error

Table B.5: Tracked error statistical observations for the Analytic PI x Manual PI scenario.

Observation from the data:	The average tracked error for the Analytic PI tuning is lesser than that for the Manual PI.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average tracked error in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Manual PI is not equal to 0.

in each group is the same. We mostly conclude that because the *p-value* is very high considering the expected standards of < 0.05, and the confidence interval includes a 0 difference, which means that there is still a possibility that the average means in the two observed groups are actually the same.

B.2.2 System response time

Table B.6 showcases the observations and hypothesis about the rate of items processed on time for the Analytic PI x Manual PI scenario. Complementary to that, Table B.7 presents what was observed from the data regarding the SLA violation rate for this same use case.

Table B.6: Rate of items processed on time statistical observations for the Analytic PI x Manual PI scenario.

Observation from the data:	The average rate of items processed on time for the Analytic PI tuning is greater than that for the Manual PI.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average rate of items processed on time in each group is the same.
Alternative hypothesis:	The true difference in means between group Analytic PI and group Manual PI is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. Considering the hypothesis in Table B.6, the results show

Table B.7: SLA violation rate statistical observations for the Analytic PI x Manual PI scenario.

Observation from the data:	The average SLA violation rate for the Analytic PI tuning is lesser than that for the Manual PI.
Null hypothesis:	The average SLA violation rate in each group is the same.

a *p-value* of 0.0106, with a 95% confidence interval of $[1.030918, 6.680193]$. The sample estimate for the mean in group Analytic PI is 96.02778, while the mean for the Manual PI group is 92.17222. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[-6.680193, -1.030918]$. The sample estimate for the mean in group Analytic PI is 3.972222, while the mean for the Manual PI group is 7.827778.

Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

B.3 Scenario IV: PI - Runtime x PI - Estimated

Considering the same scenario presented in Appendix A, Section A.3, the statistical analysis of the experimentation data showcases the following results.

B.3.1 Replica allocation

Table B.8 showcases the observations and hypothesis about the allocation of replicas for the PI - Runtime x PI - Estimated scenario.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant. The results show a *p-value* of $6.83e - 16$, with a 95% confidence interval of $[-0.8106836 - 0.4985545]$. The sample estimate for the mean in group PI - Runtime is 5.416900, while the mean for the PI - Estimated group is 4.762281, which is in fact less. Considering this, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

Table B.8: Replica allocation statistical observations for the PI - Runtime x PI - Estimated scenario.

Observation from the data:	The average replica allocation for the PI - Runtime approach is greater than that for the PI - Estimated.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average replica allocation in each group is the same.
Alternative hypothesis:	The true difference in means between group PI - Runtime and group PI - Estimated is not equal to 0.

B.3.2 System response time

Table B.9 showcases the observations and hypothesis about the rate of items processed on time for the PI - Runtime x PI - Estimated scenario. Complementary to that, Table B.10 presents what was observed from the data regarding the SLA violation rate for this same use case.

Table B.9: Rate of items processed on time statistical observations for the PI - Runtime x PI - Estimated scenario.

Observation from the data:	The average rate of items processed on time for the PI - Runtime approach is greater than that for the PI - Estimated.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average rate of items processed on time in each group is the same.
Alternative hypothesis:	The true difference in means between group PI - Runtime and group PI - Estimated is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. Considering the hypothesis in Table B.9, the results show a *p-value* of 0.01393, with a 95% confidence interval of $[-11.905847, -2.349708]$. The sample estimate for the mean in group PI - Runtime is 94.22222, while the mean for the PI - Estimated group is 87.09444. Complementary to that, for the SLA violation rate, the same

Table B.10: SLA violation rate statistical observations for the PI - Runtime x PI - Estimated scenario.

Observation from the data:	The average SLA violation rate for the PI - Runtime approach is lesser than that for the PI - Estimated.
Null hypothesis:	The average SLA violation rate in each group is the same.

p-value is obtained, with a 95% confidence interval of $[2.349708, 11.905847]$. The sample estimate for the mean in group PI - Runtime is 5.777778, while the mean for the PI - Estimated group is 12.905556.

Considering these results, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant, for both of the cases presented.

B.4 Scenario V: SIMO x Independent control

Considering the same scenario presented in Appendix A, Section A.4, the statistical analysis of the experimentation data showcases the following results.

B.4.1 Replica allocation

Table B.11 showcases the observations and hypothesis about the allocation of replicas for the SIMO PI with $\alpha = 0.2$ X Independent scenario.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant. In general lines, considering a confidence interval of 95%, if the *p-value* is less than 0.05 then you can reject the null hypothesis and conclude that the difference between means in the two categories is statistically significant. Note that this same principle will be used for all the tests performed using the Student's distribution in this section.

The results show a *p-value* of 0.3613, with a 95% confidence interval of $[-0.4812391, 0.1755608]$. The sample estimate for the mean in group SIMO PI (0.2) is 7.244643, while the mean for the Independent group is 7.397482. Considering these results, we can not reject the null hypothesis that the average replica allocation in each group is

Table B.11: Replica allocation statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average replica allocation for the SIMO PI controller with $\alpha = 0.2$ is lesser than that for the Independent configuration.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average replica allocation in each group is the same.
Alternative hypothesis:	The true difference in means between group SIMO PI (0.2) and group Independent is not equal to 0.

the same. We mostly conclude that because the *p-value* is above the expected standards of < 0.05, and the confidence interval includes a 0 difference, which means that there is still a possibility that the average means in the two observed groups are actually the same.

Note that we want to compare each controller preference configuration from the SIMO control strategy against the Independent strategy. Thus, similarly to the previous analysis, Tables B.12 and B.13 showcase the observations and hypothesis about the allocation of replicas for the SIMO PI with $\alpha = 0.5$ and $\alpha = 0.8$.

Table B.12: Replica allocation statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average replica allocation for the SIMO PI controller with $\alpha = 0.5$ is greater than that for the Independent configuration.
Null hypothesis:	The average replica allocation in each group is the same.

Table B.13: Replica allocation statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average replica allocation for the SIMO PI controller with $\alpha = 0.8$ is greater than that for the Independent configuration.
Null hypothesis:	The average replica allocation in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. For the SIMO PI (0.5) configuration, the results show a *p-value* of 0.8363, with a 95% confidence interval of $[-0.3293160, 0.4068834]$. The sample estimate for the mean in group SIMO PI (0.5) is 7.436266, while the mean for the Independent group is 7.397482. Considering these results, we can not reject the null hypothesis that the average replica allocation in each group is the same.

Likewise, for the SIMO PI (0.8) configuration, the results show a *p-value* of 0.7454, with a 95% confidence interval of $[-0.3180956, 0.4442785]$. The sample estimate for the mean in group SIMO PI (0.8) is 7.460573, while the mean for the Independent group is 7.397482. Considering these results, we can not reject the null hypothesis that the average replica allocation in each group is the same.

B.4.2 Tracking of the reference value

For this metric, the reference value changes according to the internal controller being used for both the SIMO and the Independent approach. This is because for the performance controller, the reference value is the input rate of new work items, and for the cost controller, it is a user-defined desired cost. Thus, in order to quantify how well the system is following the reference value, we take into consideration the tracked error, which is equal to 0 when the system follows the reference value, and different than 0 otherwise.

Considering this, we will first evaluate the tracked error for the internal performance controller. Table B.14 showcases the observations and hypothesis about the tracking of the reference value for the SIMO PI with $\alpha = 0.2$ X Independent scenario.

Table B.14: Performance controller: tracked error statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.2$ is lesser than that for the Independent configuration.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average tracked error in each group is the same.
Alternative hypothesis:	The true difference in means between group SIMO PI (0.2) and group Independent is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. The results show a *p-value* of 0.2885, with a 95% confidence interval of $[-0.2846588, 0.0847530]$. The sample estimate for the mean in group SIMO PI (0.2) is -0.04404762, while the mean for the Independent group is 0.05590528. Considering these results, we can not reject the null hypothesis that the average tracked error in each group is the same.

Similarly to the previous analysis, Tables B.15 and B.16 showcase the observations and hypothesis about the tracked error for the SIMO PI with $\alpha = 0.5$ and $\alpha = 0.8$.

Table B.15: Performance controller: tracked error statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.5$ is lesser than that for the Independent configuration.
Null hypothesis:	The average tracked error in each group is the same.

Table B.16: Performance controller: tracked error statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.8$ is lesser than that for the Independent configuration.
Null hypothesis:	The average tracked error in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. For the SIMO PI (0.5) configuration, the results show a *p-value* of 0.3471, with a 95% confidence interval of $[-0.20021370, 0.07044982]$. The sample estimate for the mean in group SIMO PI (0.5) is -0.008976661, while the mean for the Independent group is 0.055905276. Considering these results, we can not reject the null hypothesis that the average tracked error in each group is the same.

Likewise, for the SIMO PI (0.8) configuration, the results show a *p-value* of 0.8002, with a 95% confidence interval of $[0.1428702, 0.1102114]$. The sample estimate for the mean in group SIMO PI (0.8) is 0.03957587, while the mean for the Independent group is 0.055905276. Considering these results, we also can not reject the null hypothesis that the average tracked error in each group is the same.

Next, we then evaluate the tracked error for the internal cost controller. Table B.17 showcases the observations and hypothesis about the tracking of the reference value for the SIMO PI with $\alpha = 0.2$ X Independent scenario.

Table B.17: Cost controller: tracked error statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.2$ is lesser than that for the Independent configuration.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average tracked error in each group is the same.
Alternative hypothesis:	The true difference in means between group SIMO PI (0.2) and group Independent is not equal to 0.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. The results show a *p-value* of 0.001019, with a 95% confidence interval of $[-0.0019166155, -0.0004859401]$. The sample estimate for the mean in group SIMO PI (0.2) is 0.003079901, while the mean for the Independent group is 0.004281179. Considering these results, for this particular case so far, we can then reject the null hypothesis and say that the differences in means between the two groups are statistically significant.

Similarly to the previous analysis, Tables B.18 and B.19 showcase the observations and hypothesis about the tracked error for the SIMO PI with $\alpha = 0.5$ and $\alpha = 0.8$.

Table B.18: Cost controller: tracked error statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.5$ is lesser than that for the Independent configuration.
Null hypothesis:	The average tracked error in each group is the same.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for these cases. For the SIMO PI (0.5) configuration, the results show a

Table B.19: Cost controller: tracked error statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average tracked error for the SIMO PI controller with $\alpha = 0.8$ is lesser than that for the Independent configuration.
Null hypothesis:	The average tracked error in each group is the same.

p-value of 0.4452, with a 95% confidence interval of $[-0.0011766546, 0.0005173237]$. The sample estimate for the mean in group SIMO PI (0.5) is 0.003951514, while the mean for the Independent group is 0.004281179. Considering these results, we can not reject the null hypothesis that the average tracked error in each group is the same.

Likewise, for the SIMO PI (0.8) configuration, the results show a *p-value* of 0.8484, with a 95% confidence interval of $[-0.0009534998, 0.0007841557]$. The sample estimate for the mean in group SIMO PI (0.8) is 0.004196507, while the mean for the Independent group is 0.004281179. Considering these results, we also can not reject the null hypothesis that the average tracked error in each group is the same.

B.4.3 System response time

The response time is given by the time it takes for an item to be processed once it enters the system, which is considered to be approximately 1.5 seconds for each item. Here, we look at this metric from two perspectives, first we calculate the rate of items processed on time, i.e. items that took a maximum of 1.5 seconds to finish, and considering this same prerogative, the rate of requests that violate a given SLA of 1.5 seconds response time per item.

Table B.20 showcases the observations and hypothesis about the rate of items processed on time for the SIMO PI with $\alpha = 0.2$ X Independent scenario. Complementary to that, Table B.21 presents what was observed from the data regarding the SLA violation rate for this same use case.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. Considering the hypothesis in Table B.20, the results show a *p-value* of $7.303e-08$, with a 95% confidence interval of $[-13.069785, -7.984383]$. The sample estimate for the mean in group SIMO PI (0.2) is 40.33860, while the mean for

Table B.20: Rate of items processed on time statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average rate of items processed on time for the SIMO PI controller with $\alpha = 0.2$ is lesser than that for the Independent configuration.
Research question:	Is this difference due to chance or is it a statistically significant difference?
Null hypothesis:	The average rate of items processed on time in each group is the same.
Alternative hypothesis:	The true difference in means between group SIMO PI (0.2) and group Independent is not equal to 0.

Table B.21: SLA violation rate statistical observations for the SIMO PI (0.2) x Independent scenario.

Observation from the data:	The average SLA violation rate for the SIMO PI controller with $\alpha = 0.2$ is greater than that for the Independent configuration.
Null hypothesis:	The average SLA violation rate in each group is the same.

the Independent group is 50.86568. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[7.984383, 13.069785]$. The sample estimate for the mean in group SIMO PI (0.2) is 59.66140, while the mean for the Independent group is 49.13432.

Considering these results, we can then reject the null hypothesis for both cases, and say that the differences in means between the two groups are in fact statistically significant.

Now for the 0.5 control preference, Table B.22 showcases the observations and hypothesis about the rate of items processed on time for the SIMO PI with $\alpha = 0.5$ X Independent scenario. Complementary to that, Table B.23 presents what was observed from the data regarding the SLA violation rate for this same use case.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. Considering the hypothesis in Table B.22, the results show a *p-value* of $1.006e - 12$, with a 95% confidence interval of $[18.48668, 23.54687]$.

Table B.22: Rate of items processed on time statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average rate of items processed on time for the SIMO PI controller with $\alpha = 0.5$ is greater than that for the Independent configuration.
Null hypothesis:	The average rate of items processed on time in each group is the same.

Table B.23: SLA violation rate statistical observations for the SIMO PI (0.5) x Independent scenario.

Observation from the data:	The average SLA violation rate for the SIMO PI controller with $\alpha = 0.5$ is lesser than that for the Independent configuration.
Null hypothesis:	The average SLA violation rate in each group is the same.

The sample estimate for the mean in group SIMO PI (0.5) is 71.88246, while the mean for the Independent group is 50.86568. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[-23.54687, -18.48668]$. The sample estimate for the mean in group SIMO PI (0.5) is 28.11754, while the mean for the Independent group is 49.13432.

Considering these results, we can then reject the null hypothesis for both cases, and say that the differences in means between the two groups are in fact statistically significant.

Finally, for the 0.8 control preference, Table B.24 showcases the observations and hypothesis about the rate of items processed on time for the SIMO PI with $\alpha = 0.8$ X Independent scenario. Complementary to that, Table B.25 presents what was observed from the data regarding the SLA violation rate for this same use case.

A Student's *t-test* was performed to analyze if the difference observed from data is statistically significant for both cases. Considering the hypothesis in Table B.24, the results show a *p-value* of $1.39e-14$, with a 95% confidence interval of $[29.95371, 34.16158]$. The sample estimate for the mean in group SIMO PI (0.8) is 82.92333, while the mean for the Independent group is 50.86568. Complementary to that, for the SLA violation rate, the same *p-value* is obtained, with a 95% confidence interval of $[-34.16158, -29.95371]$. The sample esti-

Table B.24: Rate of items processed on time statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average rate of items processed on time for the SIMO PI controller with $\alpha = 0.8$ is greater than that for the Independent configuration.
Null hypothesis:	The average rate of items processed on time in each group is the same.

Table B.25: SLA violation rate statistical observations for the SIMO PI (0.8) x Independent scenario.

Observation from the data:	The average SLA violation rate for the SIMO PI controller with $\alpha = 0.8$ is lesser than that for the Independent configuration.
Null hypothesis:	The average SLA violation rate in each group is the same.

mate for the mean in group SIMO PI (0.8) is 17.07667, while the mean for the Independent group is 49.13432.

Considering these results, we can then reject the null hypothesis for both cases, and say that the differences in means between the two groups are in fact statistically significant.