information

Article

Improving QoS Management Using Associative Memory and Event-Driven Transaction History

Antonella Di Stefano, Massimo Gollo and Giovanni Morana *

Deptement of Electrical, Electronics and Information Engineering, University of Catania, 95125 Catania, Italy; antonella.distefano@unict.it (A.D.S.); massimo.gollo@phd.unict.it (M.G.)
* Correspondence: giovanni.morana@unict.it

Abstract: Managing modern, web-based, distributed applications effectively is a complex task that requires coordinating several aspects, including understanding the relationships among their components, the way they interact, the available hardware, the quality of network connections, and the providers hosting them. A distributed application consists of multiple independent and autonomous components. Managing the application involves overseeing each individual component with a focus on global optimization rather than local optimization. Furthermore, each component may be hosted by different resource providers, each offering its own monitoring and control interfaces. This diversity adds complexity to the management process. Lastly, the implementation, load profile, and internal status of an application or any of its components can evolve over time. This evolution makes it challenging for a Quality of Service (QoS) manager to adapt to the dynamics of the application's performance. This aspect, in particular, can significantly affect the QoS manager's ability to manage the application, as the controlling strategies often rely on the analysis of historical behavior. In this paper, the authors propose an extension to a previously introduced QoS manager through the addition of two new modules: (i) an associative memory module and (ii) an event forecast module. Specifically, the associative memory module, functioning as a cache, is designed to accelerate inference times. The event forecast module, which relies on a Weibull Time-to-Event Recurrent Neural Network (WTTE-RNN), aims to provide a more comprehensive view of the system's current status and, more importantly, to mitigate the limitations posed by the finite number of decision classes in the classification algorithm.

Keywords: quality of service; microservices; decision making; knowledge management; edge–cloud continuum; information theory

check for updates

Citation: Di Stefano, A.; Gollo, M.; Morana, G. Improving QoS Management Using Associative Memory and Event-Driven Transaction History. *Information* **2024**, *15*, 569. https://doi.org/10.3390/info15090569

Academic Editor: Aneta Poniszewska-Maranda

Received: 15 July 2024
Revised: 19 August 2024
Accepted: 9 September 2024
Published: 18 September 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Today, many companies offer their services through large-scale, web-based, distributed applications, where the Quality of Service (QoS) aspects [1], related to non-functional requirements such as responsiveness, availability, and robustness, are as crucial as the type of service provided by the application itself. In this context, ensuring high QoS is paramount to maintaining user satisfaction and competitive advantage.

Managing QoS effectively for these types of distributed applications, composed of numerous components in the form of microservices [2], is a complex task that requires coordinating several aspects. These include understanding the relationships among components, their interactions, the available hardware, the quality of network connections, and the providers hosting them. Each component may be hosted by different resource providers, each offering unique monitoring and control interfaces. This diversity adds complexity to the management process, as it necessitates a comprehensive strategy that can accommodate various operational environments and performance metrics.

Furthermore, managing the application involves overseeing each component with a focus on global optimization rather than local optimization. This holistic approach ensures that the application performs optimally as a whole, rather than simply optimizing

individual components in isolation. This is crucial because the performance of one component can significantly impact the overall application performance, especially in tightly coupled systems.

Another critical challenge is that the implementation, load profile, and internal status of an application or any of its components can evolve over time. This evolution makes it challenging for a QoS manager to adapt to the dynamics of the application's performance. Controlling strategies often rely on the analysis of historical behavior; therefore, changes in the application can disrupt these strategies and reduce their effectiveness.

In previous work, the authors introduced SLADE, an SLA-driven [3], AI-based, Quality of Service manager, designed to monitor and control the behavior of distributed applications in highly dynamic and heterogeneous environments like the ones described above.

SLADE exploits a MAPE-like (Monitor, Analyze, Plan, and Execute) management loop [4–6] and relies on AI-based techniques to extract knowledge about the application's performance.

In this paper, the authors propose an extension of SLADE, aiming to improve the performance of the SLADE decision-making process by leveraging two new modules: (i) an associative memory [7] to speed up inference time and (ii) an event forecast module [8] to improve the analysis of the context in which SLADE operates, mitigating the problem of the finite number of decision classes. The associative memory module is implemented as a cache composed of multiple matrices for each Service Level Indicator (SLI) to which the SLA refers. The Event Forecasting Module exploits a WTTE-RNN [9] to build a model capable of providing the probability distribution of the occurrence of a given event. The sequence of events, obtained from the time series, is defined as an event-driven transaction history and considers the definition in [10].

The paper is structured as follows: Section 2 reviews the state of the art in recent distributed application monitoring and control techniques. Section 3 describes the background concepts and the reference scenario in which SLADE operates. Section 4 provides an overview of the phases within the MAPE-like management loop and how the knowledge base is used to coordinate the management operations. Section 5 provides an overview of the implementation of the new components in a use case scenario. Finally, Section 6 gives conclusions and indications about future works.

2. Related Works

SLADE is a QoS manager designed as an intelligent agent to satisfy the constraints expressed in SLAs using strategies based on Artificial Intelligence (AI) algorithms. The management of QoS for software applications as expressed in SLAs is a highly debated topic in the scientific literature, particularly in the cloud environment. In this context, the increased availability of resources and finer control over their allocation have made it possible to achieve more precise control over application performance compared to traditional distributed systems.

The operation of SLADE is based on defining the relationships within the triple <workload, performance, hardware resource> and the system's ability to learn from past observations to predict the optimal hardware configuration. This approach ensures that, given a specific workload, the hardware configuration that best meets the SLA constraints can be identified.

In the cloud environment, numerous solutions exist that allow for QoS and SLA management through appropriate resource allocation. The use of SLAs is comprehensively explained in [3]. The authors propose utilizing SLAs to define QoS guarantees, manage resources, and establish formal contracts between providers and consumers. These contracts specify the resources, QoS standards, obligations, and guarantees for delivering a particular service or resource. In [11], the author presents a solution for real-time SLA management. This solution facilitates proactive SLA management during runtime through the provisioning of cloud services and resources. Within this framework, SLAs are negotiated between cloud users and providers at runtime using an SLA manager. A Resource Manager proactively allocates resources to minimize SLA violations and misdetection

costs. The focus of this solution is mainly on SLA management and, unlike SLADE, it does not address handling multiple cloud providers or selecting resources based on specific application needs. Other works, such as [12], are centred solely on resource provisioning, which involves identifying adequate resources for a given workload based on the desired QoS requirements. SLADE, however, also addresses resource scheduling, which involves mapping and executing application workloads on designated resources through resource provisioning. Several solutions, including [13], offer this capability. An example is the framework presented in [14]. Ref. [14] focuses on the idea that provisioning adequate resources to cloud workloads is closely related to the QoS requirements of these workloads and frames the allocation of optimal workload–resource pairs as an optimization problem. The proposed resource provisioning framework includes the autonomic scheduling of resources and monitoring service behavior to adjust dynamically in case of QoS violations. Many solutions for QoS management aim to optimize hardware resource management by analyzing and forecasting workloads [15–17]. SLADE, leveraging control theory techniques as in [18], focuses on controlling application performance by managing the relationship between workload and hardware configuration, making corrective actions less dependent on the workload.

In [19], the authors propose an SLA-driven architecture for the automatic provisioning, scheduling, allocation, and dynamic management of cloud resources to avoid QoS violations. This architecture is based on the Web Services Agreement (WS-Agreement) specification and provides a multi-provider and multi-level framework for building and deploying cloud services across different cloud levels. However, it is limited by the set of providers compatible with the WS-Agreement specification.

SLADE was designed to address the complexities of managing resources across multiple independent providers and diverse programmable interfaces. These challenges are common in environments such as the Edge–Cloud Continuum (ECC) [20], where, despite no universally accepted and rigorous definition existing [21], one has to deal with heterogeneous resource clusters and interact with interfaces including REST-APIs such as those used by Kubernetes (K8s) [22], a de facto standard container orchestrator engine, or Virtual Machine Monitor (VMM) Command Line Interfaces (CLIs).

In [23], the authors explore Collaborative Cloud Computing as an emerging computational model for sharing on-demand cloud resources in a multi-cloud environment where resource providers share unused resources to meet their SLA commitments to users. This approach, however, emphasizes cloud–provider collaboration to minimize the number of SLAs. In contrast, SLADE, as in [24], focuses on a user-centric approach, optimizing the monitored application.

In this paper, the capabilities of SLADE are extended using associative memory and an event prediction system based on historical data from past observations. This solution is inspired by the approaches proposed in [7,10], which introduce examples of a new class of management systems based on the General Theory of Information [25,26].

3. Background and Reference Scenario

3.1. Distributed Software Applications

The last decade has witnessed a fundamental paradigm shift in modern, large-scale software systems design, development, and deployment processes, transitioning from a monolithic architectural style to microservice architectures [27].

From a straightforward perspective, the most significant difference lies in the number of software units built and deployed to make the application operational. Traditionally, the frameworks for web applications included at least three logical tiers: a frontend component (a set of HTML pages and JavaScript code downloaded and executed in the client's browser), a backend server-side component that receives requests from the frontend, and a database with which the backend interacts to enable data persistence. Despite achieving modularity through proper classes, packages, and modules separation, all business logic for handling incoming requests typically resides within a single running process.

As internet availability increased, so did the demand for new application functional requirements. This led to a continuous cycle of extending, developing, validating, deploying, and managing the backend, which serves as the application's core. Consequently, even small, marginal changes in the codebase required executing an extensive pipeline of activities to maintain application operation and currency.

Considering the Pareto principle, where, for example, 80% of requests typically involve only 20% of the functionalities, the monolith approach can lead to significant resource waste. Each instance of the monolith contains all the features and logic of the application, regardless of which specific functionality needs scaling, resulting in the inefficient use of resources and increased operational costs.

Unlike traditional monolithic applications, distributed applications consist of multiple independent and autonomous components which work together to provide a cohesive set of services [28]. These application components are also referred to as microservices. The suffix *micro* does not necessarily mean that each service is small in size; rather, it emphasizes the modularity and independence of each component

Each microservice of a distributed application encapsulates a specific aspect of high-level business functionality, focusing on a minimal set of related business domain entities. This architecture allows development, deployment, and management teams to operate independently of one another. Teams can concentrate on a single component without needing to understand the internal workings of others, promoting greater efficiency and specialization.

Despite the modularity of this approach and the ability to treat components as black boxes, each component must precisely understand the input and output parameters of every service exposed by other components. Efficient inter-component communication is crucial to ensuring seamless and reliable interactions. Components exchange data through well-defined interfaces and protocols that abstract their business logic. This clear definition of communication pathways allows components to interact consistently and predictably, even as individual components are updated.

From an application user's perspective, each requested functionality is executed transparently, without any awareness of the underlying architectural segmentation. Each request initiates an inter-component service path, where multiple components communicate and collaborate to fulfill the user's request.

3.2. Distributed Environment

Microservice architecture has transformed the application design process and significantly impacted software deployment strategies. By leveraging application decomposition, applications can achieve greater scalability and fault tolerance. This approach provides an efficient way to allocate computing, storage, and network resources in a fine-grained manner.

The ability to independently deploy and scale each service in a microservices architecture enables the more efficient use of resources if compared to the monolithic approach.

Teams can optimize the application globally by fine-tuning individual components. This involves focusing on less critical services and making incremental improvements.

In a microservices architecture, each service can be managed using two primary techniques: horizontal scaling and vertical scaling. Horizontal scaling, also known as scaling out, involves adding new instances of identical components to distribute the load across the replicas. This method enhances performance, fault tolerance, and availability since the failure of one instance does not affect the overall service. Load balancers are typically used to evenly distribute incoming requests among the available instances, ensuring that no single instance becomes a bottleneck.

Vertical scaling, or scaling up, involves increasing the amount of resources within an existing component instance. This approach increases the computational power, memory, or storage capacity of a single component deployment to handle a larger load. While vertical scaling can be simpler to implement since it does not require changes to the architecture or additional components such as a load balancer, it has some intrinsic limitations. There is a

physical limit to how much a single instance can be scaled up, and it does not provide the same level of fault tolerance as horizontal scaling, as the service still relies on a single node.

In practice, the choice between horizontal and vertical scaling often depends on specific requirements and contextual factors, such as monetary budget and the remaining error budget. For instance, if an organization has a limited monetary budget, vertical scaling might be preferred initially, as it involves simply upgrading existing hardware rather than deploying and maintaining additional instances. In a pay-per-use context, dealing with software entities such as application components involves increasing the resource limits of the software instances. However, vertical scaling has its limits, and beyond a certain point, it may become more cost effective to add more instances horizontally.

On the other hand, if maintaining a low error rate is critical and the error budget (the allowable amount of errors or downtime) is tight, horizontal scaling might be more appropriate. Horizontal scaling improves fault tolerance and reliability because the load is distributed across multiple instances. If one instance fails, the others can continue to handle requests, minimizing downtime and maintaining service quality.

Initially, deploying components on bare-metal servers was the standard approach in the evolution of distributed systems. This method involves dedicating entire physical servers to specific applications or services, leading to the underutilization of resources and increased management effort due to extensive manual configuration and maintenance.

The introduction of virtualization marked a significant shift in deployment strategies, allowing multiple virtual instances, or Virtual Machines (VMs), to run on a single physical server. Each VM operates with its operating system and isolated environment. Virtualization technology improves resource utilization and simplifies management through hypervisors, which could dynamically allocate resources to VMs as needed.

However, VMs also have their drawbacks, such as the overhead associated with running multiple operating systems on a single server, which could lead to inefficiencies in resource usage. Additionally, while provisioning and managing VMs is more efficient than bare-metal servers, it still involves considerable effort and complexity.

Container technology introduces another revolutionary change in deployment strategies by allowing the sharing of the host operating system's kernel and isolating applications at the process level. This approach significantly reduces the overhead associated with virtualization, enabling the more efficient use of resources [29]. Containers are lightweight, start quickly, and consume fewer resources than VMs, making them ideal for microservices architectures where applications are composed of many small, independent services.

The adoption of containerization, supported by the use of Container Orchestrator Engines (COEs) like Kubernetes [22], not only improves the management of applications through the automation of activities such as the deployment, scaling, and configuration of their components but also increases efficiency in resource usage and overall performance [30], showing reductions in CPU and memory usage up to, respectively, 25% and 30% compared to a monolithic architecture.

Additional features such as self-healing, load balancing, and rolling updates are inherently integrated.

Leveraging COEs does not reduce the responsibility of properly allocating resources and designing the placement and horizontal or vertical scaling policies of components within the deployment infrastructure. The deployment process must still involve capacity planning and ongoing refinement, considering several critical factors, including communication overhead between microservices and potential faults in the machines or networks they rely on.

Over time, as the application scales and usage patterns change, component resource allocations need to be reviewed and adjusted to prevent resource wastage or shortages that could impact performance and reliability.

The placement and resource refinement of components also involve decisions about their location to minimize latency and maximize data throughput. Factors such as cross-

cluster, cross-technology, and cross-administrative domain considerations must be taken into account in the software application performance management process.

3.2.1. Configurations

The main challenge with resource allocation, load balancing, and component placement lies in their continuous nature. Kubernetes, for example, allows resource allocation in fractions of CPU and memory units, such as 500 millicores or 512 MB. Additionally, horizontal scaling can theoretically support an infinite number of replicas per component, though this is impractical in reality.

Beyond CPU, memory, and replica counts, additional software-specific parameters such as the number of cores to use, thread pool size, and cache dimensions can significantly enhance component performance. These parameters add another layer of complexity to resource tuning but also provide opportunities for optimizing performance.

By discretizing the continuous space of resource allocations and software parameters per component, *local configurations* can be defined. Each configuration is a labeled set of resource and parameter values representing a potential setup for a component. For example, a component configuration might specify 2 CPU cores, 2048 MB of memory, a thread pool size of 10, and a cache size of 512 MB, while being deployed on a Google Cloud Platform (GCP) cluster rather than an Amazon Web Service (AWS) one.

Depending on the provider, the number of resources, and other business-specific factors, a cost—monetary only, in the simplest case shown in Table 1—can be associated with each configuration.

Table 1. Local configuration parameters for different environments.

Configuration	CPU	Memory	Replicas	Thread Pool Size	Cache Size	Monetary Cost	Alt Name
cfg_small_aws_eu	2	2048 MB	5	10	512 MB	$50	0
cfg_small_gcp_eu	2	2048 MB	5	10	512 MB	$40	1
cfg_small_gcp_us	2	2048 MB	5	10	512 MB	$40	2

The component's local configurations simplify performance optimization by providing a finite set of options to evaluate. Each configuration can be tested and compared to identify the best-performing setups. Moreover, configurations can be standardized across different environments, ensuring consistent performance regardless of the underlying infrastructure.

Configurations also facilitate performance consistency across different providers or cluster types. By defining configurations that yield similar performance metrics, organizations can ensure that their applications perform predictably, regardless of where they are deployed. This abstraction allows for easier migration between cloud providers or scaling across heterogeneous environments.

Once local configurations are defined, they can be dynamically tied to each application software component and aggregated into a *global configuration*, which represents the comprehensive setup of all components, as shown in Table 2. The global configuration provides a holistic view of resource allocations, performance parameters, and associated costs, facilitating better management and optimization of the application as a whole.

Table 2. Possible application global configurations.

Global Config ID	Global Alt Name	Component 1 Alt Name	Component 2 Alt Name	Component 3 Alt Name	Component 4 Alt Name	Total Cost
1	0000	0	0	0	0	$160
2	0001	0	0	0	1	$180
…	…	…	…	…	…	…
80	2221	2	2	2	1	$300
81	2222	2	2	2	2	$350

3.3. Quality of Service Manager

SLADE, introduced in [31], is an SLA-driven, AI-based Application Performance Manager (APM) designed as a self-contained, intelligent, and autonomous software agent with the primary goal of monitoring and controlling the behavior of a given application. As a goal-oriented agent, it uses QoS constraints provided as a Service Level Agreement (SLA) to set its objectives (SLA-driven) and employs machine learning and deep learning algorithms (AI-based) to plan actions and adapt to environmental changes and fluctuations to ensure QoS fulfillment.

SLADE exploits a closed loop largely inspired by the MAPE-K (Monitor, Analyze, Plan, Execute, and Knowledge) management loop and relies heavily on a centralized, consistent knowledge base (KB), which is essential for coordinating the management process and ensuring compliance with the QoS (Quality of Service) constraints.

The KB acts as a centralized repository of shared information, ensuring consistency and reliability in the manager's decision-making process. These decisions are directly influenced by specific information in the KB. This information includes the following:

(i) Metrics (SLIs) defined in the SLA and their related constraints, such as SLOs, represent the QoS constraints, which represent the ultimate goal of the manager.
(ii) A Time Series Database (TSDB), where the metrics' runtime values are collected, stored, and indexed by time.
(iii) A Multi-Dimensional Vector (MDV) that summarizes the expected behavior of each component in terms of QoS metrics values, given specific workloads and hardware configurations as described in Section 3.2.1. This data structure significantly influences the SLADE management capabilities because each entry represents a suitable target for its control actions.
(iv) A set of Resource Provider Configurations (RPCs) that relates the resource provider, the hardware configuration, and the set of actions that can be executed on a specific provider to deploy a component with that hardware configuration.

It is worth mentioning that in the SLADE concept of the MAPE-K management loop, the phases may not operate sequentially but can instead function in parallel and concurrently. The primary objective of each phase is to continuously gather new information or elaborate existing data, transforming them into actionable updated KB content.

The monitoring phase (M-phase) of SLADE involves collecting both static and dynamic information, such as application and infrastructure topology, the state of the network and, most importantly, the desired performance state of the application. During this phase, led by SLA declaration, the metrics of interest are collected and stored in the TSDB. Existing solutions might be well suited for this purpose, such as Prometheus [32]. Additional metadata, such as the *observability window size* and the *resolution* (step) of each time series, may also be useful.

The M-phase is not limited to time series data; it can also operate with other pillars of the observation process, such as traces or logs, which may be semi-structured or text-based forms of information. Traces and logs are particularly suited to directly describe information as events and support time series analysis and forecasting for both the en-

vironment (e.g., COE or VMM) and the managed application. While dealing with time series data requires quantizing and structuring them appropriately, traces and logs provide complementary insights that can enhance the overall observability and analysis processes.

Whenever the M-phase modifies the KB—whether by detecting changes in metadata, declared SLA, or previously mentioned static information—the analysis phase (A-phase) must be triggered to start the reasoning process. The main goal of the A-phase is to evaluate the application's performance state, either current or future, by comparing the current QoS indicators against the predefined and desired ones. This evaluation determines whether the application is currently underperforming or is likely to underperform in the future. The activities in this phase can vary based on the level of prevention expected from the manager. If no forecasting activity is performed, for either SLI, the manager will operate in a *conservative mode*. If time series forecasting is exploited, one or more risk indexes have to be computed to highlight potential performance issues; the manager will operate in *proactive mode*, and preventive measures will be taken to mitigate risks before they impact the system. The analysis phase (A-phase) may uncover new patterns or information that can immediately enhance the KB, making this updated information available for other phases, such as planning and execution, to use in real-time.

The planning phase (P-phase) is the brain of the manager. It represents the decision-making core of the manager, where at least the corrective strategy is developed. The P-phase selects which actions should be taken—including no-operation actions—and evaluates whether additional monitoring or further analysis is required. Based on the insights from the A-phase, the P-phase formulates a plan to address any detected or predicted issues. The decision-making activity is a two-step process.

Based on the current workload level, the first step identifies a suitable hardware configuration for each component that can guarantee the fulfillment of the SLA-specified performance. In the second step, SLADE selects the best hardware configurations among the identified options: the final selection depends on factors such as minimizing overall configuration cost, reducing the impact on the error budget, and minimizing the number of deployments. SLADE uses an AI-based solution trained on the MDV to identify the hardware configurations that solve (or prevent) the QoS violation.

One of the strengths of SLADE is its ability to abstract the configuration required to ensure the application meets its expected QoS level, regardless of the deployment environment. This is achieved by leveraging the Resource Provider Configurations (RPCs) data structure, which maps vendor-agnostic configurations to vendor-specific sequences of actions that must be performed.

The control of SLADE can be defined as *absolute*, meaning that each configuration is uniformly applied, regardless of the application's current deployment state or its components. For example, SLADE decisions are not simply "scale-out", where one resource is added each time a decision is made. Instead, the control is *declarative*, bringing the application precisely to the desired state by explicitly specifying the exact number of replicas or the number of resources allocated for each component. This approach ensures consistency and predictability without relying on incremental adjustments.

When a global configuration is chosen, the execution phase (E-Phase) is responsible for applying it. As detailed in Section 3.2.1, global configurations are represented as sequences of individual component local configurations, with labels expressed as integers for simplicity. The positional order of these labels in the global configuration sequence matters, as it determines which local configuration must be applied to each component.

By referencing the RPC data structure, the appropriate sequence of actions for each local configuration is identified and executed, ensuring compatibility with the specific deployment environment and with the corresponding resource provider.

Due to RP heterogeneity, the sequence of actions to be applied can vary depending on the RP itself. For instance, a Container Orchestrator Engine like Kubernetes exposes an HTTP API server, which serves as a single entry point for managing the cluster. This allows SLADE to execute the necessary actions by invoking HTTP requests to the API

server and changing the state of the object that Kubernetes manages in a declarative manner. A relative scale-out operation—corresponding to an increase in the number of application component replicas—can be easily translated into an absolute configuration by sending an HTTP UPDATE request to the Kubernetes API server along with the name of the component—deployment name, in the Kubernetes jargon—and the number of desired replicas. The operational aspect of executing the action is delegated to Kubernetes, which is responsible for ensuring the number of running replicas accordingly. When dealing directly with resource providers such as Amazon Web Services (AWS) or Google Cloud Platform (GCP), applying configurations might involve interacting with their respective APIs to provision resources. For instance, scaling out in AWS might require API calls to modify Auto Scaling groups or adjust resource allocation within an Elastic Kubernetes Service (EKS) cluster. In a more generic situation, where custom user-provisioned infrastructure is used, Infrastructure as Code (IaaS) tools like Ansible [33] or Terraform [34] come into play. These tools allow for declarative configurations to be applied across a variety of environments. For example, Terraform allows to define and manage the infrastructure using configuration files, enabling to provision and adjust resources on multiple providers at once. Similarly, Ansible uses playbooks to automate the deployment and scaling of the infrastructure, ensuring that the desired state is achieved. In these cases, the role of the RPC is to map the proper IaaS procedure and execute the necessary commands to apply the desired configurations. In a Terraform-based environment, the RPC might generate the necessary configuration files and initiate Terraform commands to apply changes. In an Ansible setup, the RPC could generate and execute playbooks to manage the infrastructure state.

SLADE is designed to prevent or correct QoS violations by providing the optimal hardware configuration for each component of the managed application. It optimizes the key parameters: inference time, for quickly obtaining the target configuration; training time, to enable rapid reconfiguration of the controlling system; and accuracy, to ensure the correct configuration is used to achieve and maintain the desired QoS.

Additionally, SLADE operates independently of the QoS constraints defined by the SLOs but focuses only on SLIs. This implies that it does not require retraining when QoS constraints change over time.

4. Proposed Approach

Despite using the MDV data structure to label configurations and tag them with actions as introduced in Section 3.2.1, which reduces the dimension of the action space for QoS fine-tuning, SLADE still faces some limitations.

There is a significant constraint due to the finite number of possible hardware configurations that SLADE can suggest for maintaining QoS. This poses two main challenges. Firstly, the type of knowledge representation makes it difficult to determine when and which new class(es) will need to be added. Secondly, SLADE employs a multi-class Random Forest classification [35] that must be retrained from scratch whenever a new class is introduced.

To overcome these limitations and improve the overall SLADE performance, the authors extended SLADE by introducing two new modules: (i) Associative Memory [7], used to speed up the inference time and improve the accuracy of the decision-making process, and (ii) an Event Forecast Module (EFM), built on top of an event-driven transaction history [10], used to improve the A-phase and mitigate the problem related to the finite number of hardware configuration classes. Restructuring the Multi-Dimensional Vector (MDV) data can effectively integrate both modules.

4.1. Associative Memory

The associative memory, shown in Figure 1, contains all the actual observations of the system, representing its "field experience". It is implemented as a cache, composed of multiple matrices (one for each SLI metric), with the workload and SLI as inputs and the hardware configuration as the output. If the system has previously encountered the same

set of values, the associative memory will return the corresponding hardware configuration without invoking the trained ML algorithm. Statistically, associative memory improves both inference time and accuracy. Inference time is improved because the cache has constant query complexity, whereas the Random Forest complexity can range from quasi-linear to cubic time, depending on the variables considered. Accuracy is enhanced because, in a model consistent with the modeled reality, the associative memory's accuracy will be equal to that of the trained ML algorithm.

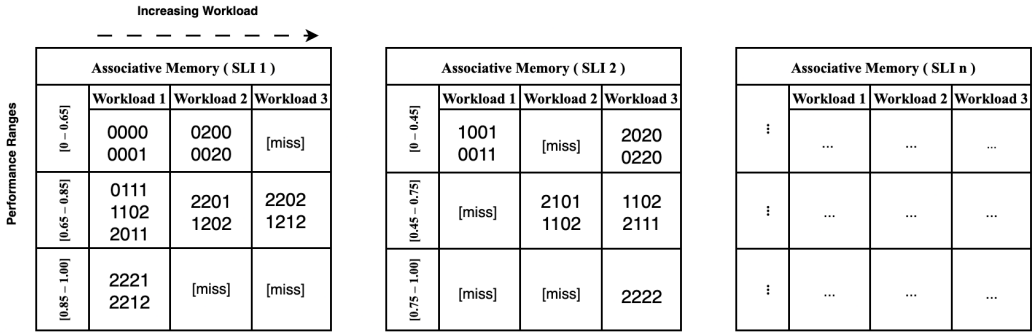

Figure 1. Associative memory.

4.2. Event Forecasting Module

As previously mentioned, the Event Forecast Module (EFM) has been introduced to overcome the limitation represented by the finite number of hardware configurations in the classification algorithm.

The EFM aims to solve an incremental learning task, which is the ability of a classifier to integrate new data and new classes during runtime. In the case of SLADE, the problem arises because a hardware configuration, or class, is considered only after at least one component has utilized it. However, for a class to be used, it must already be known, which by design, it is not.

To resolve this issue and enable the system to learn the new class and collect related data, the system must be capable of recognizing when a new hardware configuration is necessary. This requires a modification in how the controller interprets the data.

In particular, the controller must shift from the absolute configuration selection provided by SLADE to a "relative" approach, where it learns the indications of moving to the "next class", regardless of its presence in the set of known configurations inside the KB. The "next class" concept can be encoded into a *cfg_up* event.

Leveraging hardware configuration ordering by resource quantity, it is reasonable to assume that a class that can enhance the application's performance exists beyond the last one considered by the classifier.

To generate the *cfg_up* event, the idea is to transit from numerical metric values analysis to events triggering, identify all "move to the next class" events, and create an algorithm capable of anticipating such requests. When the "move to the next class" event is foreseen while at least one component is already in the highest-performing class, the system interprets this condition as an indication that a new class needs to be introduced, thereby triggering an incremental learning process.

To enable this process, two additional supporting components need to be introduced, cooperating to build the EFM as shown in Figure 2.

The Event Emitter (EE) monitors time series data and generates an event whenever specific conditions are met. These conditions define the event *firing police* and consist of rules and thresholds stored in the knowledge base as metadata.

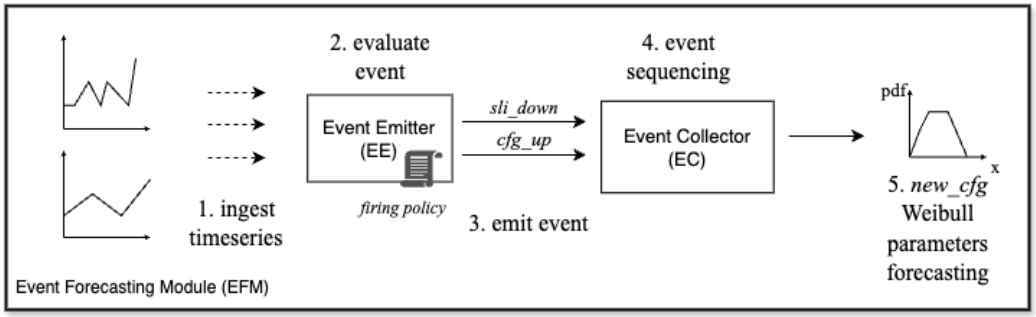

Figure 2. Event Forecasting Module architecture.

Within the context of SLADE, the EE is responsible for generating events related to changes in workload, configuration, or SLI deviations based on predefined thresholds. At least six types of events can be identified by the EE, each related to one of three factors: (i) *workload_up* and *workload_down*, triggered by a workload increase or decrease of at least 5% within a given time range; (ii) *SLI_up* and *SLI_down*, triggered by a similar change in one or more SLIs; and (iii) *cfg_up*, i.e., the "move to the next class" event and *cfg_down*, indicating a change in the hardware configuration of at least one application component.

The events emitted by the Event Emitter (EE) must be collected and ordered chronologically by the Event Collector (EC), which is responsible for identifying meaningful episodes (or patterns) within the event history timeline.

The EC plays a crucial role in organizing these events into a coherent sequence that reflects the temporal order and relationships between different events.

For instance, an episode (or pattern) might include a series of events where a workload increase (workload_up) is followed by a decline in SLI (SLI_down) and a subsequent configuration adjustment (cfg_up), ideally ending with an SLI recovery (SLI_up), i.e., with the resolution or the avoidance of an SLA breach.

In the context of SLADE, the EC is designed to predict the cfg_up event. As described in the next section, the EC utilizes the Weibull Time-To-Event Recurrent Neural Network (WTTE-RNN) to obtain the probability that the cfg_up event will happen shortly and using this information to enable SLADE to operate in a proactive mode and to trigger—automatically—the incremental learning process. This last represents a significant improvement in SLADE.

5. Implementation Overview

5.1. Associative Memory

The associative memory is a set of matrices (one for each SLI) built similarly to the Multi-Dimensional Vector. The idea is that, like the classification algorithm executed in SLADE, the expected workload level and the desired performance range are presented as inputs, and as output, it provides the hardware configuration associated with each monitored application component.

Starting from an initial base configuration (row) and a minimum load (column), the range of values for each metric belonging to the SLIs (matrix entry) is recorded. Each time a hardware configuration is modified, a new row is generated. A new column is generated each time the load increases by a given percentage value (usually 10%) compared to the previously recorded minimum value. In all cases, the values of the SLI metrics are recorded as matrix entries.

Based on this information, each time a column or row is added, SLADE updates the associative memory, or the set of memories, as follows:

- Each load level becomes a row of a matrix.
- Each range of metric values becomes a column of the matrix.
- The observed hardware configuration for that load and that range of associated metric values becomes the matrix entry.

Whenever a new workload value and metric values are presented as input, the system queries both the associative memory and the standard SLADE classifier in parallel. The associative memory selects the rows corresponding to the load and all the columns that contain the desired metric values within the defined range. The set of obtained hardware configurations, if existing—i.e., seen in the past—is filtered by cost and returned to the execution phase. If the set is empty after the last filtering, the classifier's result is taken. Using the first available result, whatever it may be, ensures that, in the worst case, the performance is at least equal to that of the classifier. Figure 3 shows the results of a series of tests performed to evaluate the impact of the cache on the performance of the QoS manager. The evaluation considers three different scenarios: the first scenario includes five different workload ranges and eight different hardware configurations (5/8); the second scenario includes eight workload ranges and ten different hardware configurations (8/10); and lastly, the third scenario includes ten workload ranges and twelve different hardware configurations (10/12). Each of these scenarios is evaluated for different percentages of "known conditions", i.e., conditions previously encountered by the controlling system. When these conditions are encountered as inputs, the result can be obtained from the cache without relying on the classification algorithm, reducing the overall time needed to retrieve it. The value of each bar in the graph actually represents the ratio between the time needed to retrieve the hardware configuration using only the classification algorithm and the same time when the cache (with a related percentage of known conditions) is used.

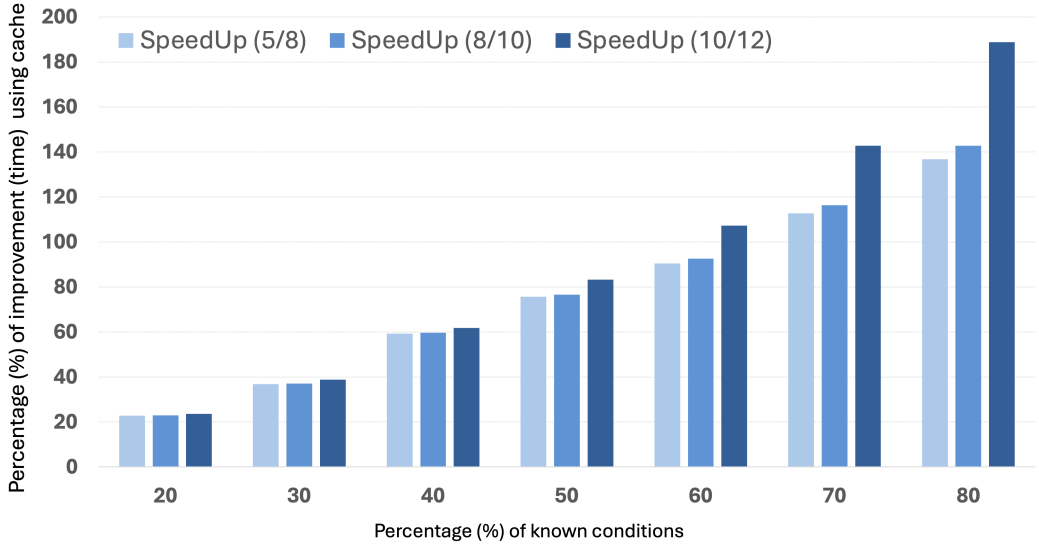

Figure 3. SpeedUp using cache.

From Figure 3, it is possible to see the following:

- The higher the percentage of "known conditions", the greater the speedup obtained using the cache. This ranges from a 23% improvement for the cache on configuration 5/8—20% (5 workloads, 8 hardware configurations, and 20% known conditions) to 188% (almost 2×) for the cache on configuration 10/12—80%.
- The higher the number of workload ranges and hardware configurations, the greater the impact of the cache.

5.2. Event Prediction

Event forecasting is a powerful tool for decision-making, aiding in the optimization of various processes, from efficient resource allocation to risk mitigation. The scientific literature offers several approaches [8,36] to predicting future events by integrating current trends with historical data. In this paper, the Weibull Time To Event Recurrent Neural Network (WTTE-RNN, introduced in [9]) is used to predict the *cfg_up* event. The idea

behind this approach is to build a model capable of providing the probability distribution of the occurrence of a given event.

In the WTTE-RNN approach, a recurrent neural network is used to predict the alpha (α) and beta (β) parameters of the Weibull distribution, which respectively represent "how soon" and "how sure" the event will manifest. The model used is the standard model taken from the aforementioned thesis.

After transforming the time series into events, the value of *workload_up*, *workload_down*, *SLI_up*, and *SLI_down* are used as inputs for the RNN, while the Weibull distribution is used to approximate the TTE distribution for the *cfg_up* event.

The model is still in the experimental phase, but the initial results seem to demonstrate the validity of the proposed approach.

In particular, considering only certain values of β, the value of the α parameter is effectively used in both the analysis phase and the planning phase, in the following ways:

- In the analysis phase, the values are used to enable SLADE to operate in a proactive mode. Since a *cfg_up* event occurs as a result of either a rapid workload increase or a degradation of the SLI, a rising probability of a *cfg_up* event can be interpreted as a signal of one of these preliminary conditions, allowing the system to take action to avoid an SLI violation. This is particularly useful when the error budget is low.
- In the planning phase, the values are used in conjunction with the current hardware configuration. If the probability of a *cfg_up* event is high and at least one component is already in the highest-performing class, SLADE will interpret this as an indication that a new, better hardware configuration is needed and will trigger the incremental learning process.

Event Forecasting Module Implementation Design

The Event Forecasting Module (EFM) implementation design comprises two main, independent components that fulfill the functional requirements described in Section 5.2. These components, the Event Emitter (EE) and Event Collector (EC), are event-driven systems that use a third-party message broker, such as Apache Kafka or RabbitMQ, to facilitate asynchronous communication.

The EE, functioning as a proactive Message Publisher, collects data on workload, application configurations, and Service Level Indicators (SLIs) from Prometheus [32]. Under specific conditions, it translates these data into corresponding events and publishes them on the message broker. These events are published on dedicated message broker topics and include (I) the event generation timestamp, (II) the event type, and (III) the current value of the corresponding metric. As illustrated in Figure 2, the firing policy configuration is crucial in determining the conditions and thresholds for event generation.

The EC, acting as a passive component, subscribes to the topics where the EE emits events. Its primary task is to use the data from the EE to predict α and β Weibull distribution parameters using the WTTE-RNN model.

These parameters are used to assess the likelihood and timing of *cfg_up* events.

The current EFM implementation manages the detection of new class events initially through alerts, employing tools such as Prometheus AlertManager to notify operators. This allows for manual configuration discovery and subsequent updates to the knowledge base.

6. Conclusions and Future Direction

This article introduces two mechanisms to enhance the performance of a QoS manager. In particular, the focus is on improving system performance and scalability by referencing associative memory and utilizing an event prediction system based on event sequencing.

The proposed approach has been applied in SLADE, an SLA-driven, AI-based QoS manager introduced for ECC. Extending SLADE with the new components significantly improves its effectiveness. From a performance perspective, the use of associative memory allows the system to enhance both the average inference time and the accuracy of the results. In terms of scalability, the system has shown the ability to automatically determine

when the introduction of a new class is necessary, overcoming the limitation imposed by using a finite number of classes.

For the future, the following extensions are planned:

- Implementing the system using the "Structural Machine" model introduced in [37]. In particular, it is planned to map the functionalities offered by the autopoietic manager with those of the component that interfaces SLADE directly with resource providers and to use the classification algorithm to trigger the resource allocation procedure.
- Exploring different solutions to exploit better the information obtained from the event-based transaction system. In this context, the aim is to improve time-to-event prediction using more events and, at the same time, extend the set of predicted events to increase the capabilities of the Analysis phase of MAPE in anticipating possible unfavorable scenarios.
- Adopt, in the planning phase, the use of N single-class classifiers [38] instead of a single classifier for N classes to better manage the issue of the finite number of classes.

Author Contributions: Conceptualization, A.D.S. and G.M.; Writing—original draft, A.D.S., M.G. and G.M.; Writing—review & editing, A.D.S. and M.G.; Project administration, G.M. All authors have read and agreed to the published version of the manuscript.

Funding: This work was partially supported by the European Union under the Italian National Recovery and Resilience Plan (NRRP) of NextGenerationEU, partnership on "Telecommunications of the Future" (PE00000001—program "RESTART").

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The data presented in this study are available on request from the corresponding author.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Di Stefano, A.; Di Stefano, A.; Morana, G. Improving QoS through network isolation in PaaS. *Future Gener. Comput. Syst.* **2022**, *131*, 91–105. [CrossRef]
2. Fowler, M.; Lewis, J. Microservices. 2014. Available online: https://martinfowler.com/articles/microservices.html (accessed on 14 July 2024).
3. Buyya, R.; Yeo, C.S.; Venugopal, S.; Broberg, J.; Brandic, I. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Gener. Comput. Syst.* **2009**, *25*, 599–616. [CrossRef]
4. Kephart, J.; Chess, D. The vision of autonomic computing. *Computer* **2003**, *36*, 41–50. [CrossRef]
5. Iglesia, D.G.D.L.; Weyns, D. Mape-k formal templates to rigorously design behaviors for self-adaptive systems. *ACM Trans. Auton. Adapt. Syst.* **2015**, *10*, 1–31. [CrossRef]
6. Halima, R.B.; Hachicha, M.; Jemal, A.; Kacem, A.H. Mape-k patterns for self-adaptation in cyber-physical systems. *J. Supercomput.* **2022**, *79*, 4917–4943. [CrossRef]
7. Mikkilineni, R.; Kelly, W.P.; Crawley, G. Digital Genome and Self-Regulating Distributed Software Applications with Associative Memory and Event-Driven History. *Computers* **2024**, *13*, 220. [CrossRef]
8. Zhao, L. Event Prediction in the Big Data Era: A Systematic Survey. *ACM Comput. Surv.* **2021**, *54*, 1–37. [CrossRef]
9. Martinsson, E. WTTE-RNN: Weibull Time to Event Recurrent Neural Network. Master's Thesis, Chalmers University of Technology, Gothenburg, Sweden, 2016.
10. Mikkilineni, R.; Kelly, W.P. Machine Intelligence with Associative Memory and Event-Driven Transaction History. *Preprints* 2024, *in press*. [CrossRef]
11. Nadeem, S.; Amin, N.u.; Zaman, S.K.u.; Khan, M.A.; Ahmad, Z.; Iqbal, J.; Khan, A.; Algarni, A.D.; Elmannai, H. Runtime Management of Service Level Agreements through Proactive Resource Provisioning for a Cloud Environment. *Electronics* **2023**, *12*, 296. [CrossRef]
12. Singh, S.; Chana, I. Cloud resource provisioning: Survey, status and future research directions. *Knowl. Inf. Syst.* **2016**, *49*, 1005–1069. [CrossRef]
13. Gill, S.S.; Chana, I. A Survey on Resource Scheduling in Cloud Computing: Issues and Challenges. *J. Grid Comput.* **2016**, *14*, 217–264. [CrossRef]

14. Gill, S.S.; Buyya, R. Resource Provisioning Based Scheduling Framework for Execution of Heterogeneous and Clustered Workloads in Clouds: From Fundamental to Autonomic Offering. *J. Grid Comput.* **2019**, *17*, 385–417. [CrossRef]
15. Gmach, D.; Rolia, J.; Cherkasova, L.; Kemper, A. Workload Analysis and Demand Prediction of Enterprise Data Center Applications. In Proceedings of the 2007 IEEE 10th International Symposium on Workload Characterization, Boston, MA, USA, 27–29 September 2007; pp. 171–180. [CrossRef]
16. Di, S.; Kondo, D.; Cirne, W. Host load prediction in a Google compute cloud with a Bayesian model. In Proceedings of the SC'12: International Conference on High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 24–29 June 2012; pp. 1–11. [CrossRef]
17. Khan, A.; Yan, X.; Tao, S.; Anerousis, N. Workload characterization and prediction in the cloud: A multiple time series approach. In Proceedings of the 2012 IEEE Network Operations and Management Symposium, Maui, HI, USA, 16–20 April 2012; pp. 1287–1294. [CrossRef]
18. Padala, P.; Zhu, X.; Uysal, M.; Wang, Z.; Singhal, S.; Merchant, A.; Salem, K. Adaptive control of virtualized resources in utility computing environments. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Portugal, 21–23 March 2007; Volume 41, pp. 289–302. [CrossRef]
19. Garcis, A.; Blanquer, I.; García, V. SLA-driven dynamic cloud resource management. *Future Gener. Comput. Syst.* **2014**, *31*, 1–11. [CrossRef]
20. Bulej, L.; Bureš, T.; Filandr, A.; Hnětynka, P.; Hnětynková, I.; Pacovský, J.; Sandor, G.; Gerostathopoulos, I. Managing latency in edge–cloud environment. *J. Syst. Softw.* **2021**, *172*, 110872. [CrossRef]
21. Khalyeyev, D.; Bureš, T.; Hnětynka, P. Towards a Reference Component Model of Edge-Cloud Continuum. In Proceedings of the 2023 IEEE 20th International Conference on Software Architecture Companion (ICSA-C), L'Aquila, Italy, 13–17 March 2023; pp. 91–95. [CrossRef]
22. Kubernetes, Production-Grade Container Orchestration. Available online: https://kubernetes.io/ (accessed on 1 July 2024).
23. ks, S.; Jaisankar, N. An automated resource management framework for minimizing SLA violations and negotiation in collaborative cloud. *Int. J. Cogn. Comput. Eng.* **2020**, *1*, 27–35. [CrossRef]
24. Di Modica, G.; Di Stefano, A.; Morana, G.; Tomarchio, O. On the Cost of the Management of user Applications in a Multicloud Environment. In Proceedings of the 2019 7th International Conference on Future Internet of Things and Cloud (FiCloud), Istanbul, Turkey, 26–28 August 2019; pp. 175–181. [CrossRef]
25. Mikkilineni, R. Mark Burgin's Legacy: The General Theory of Information, the Digital Genome, and the Future of Machine Intelligence. *Philosophies* **2023**, *8*, 107. [CrossRef]
26. Mikkilineni, R. A New Class of Autopoietic and Cognitive Machines. *Information* **2022**, *13*, 24. [CrossRef]
27. Auer, F.; Lenarduzzi, V.; Felderer, M.; Taibi, D. From monolithic systems to Microservices: An assessment framework. *Inf. Softw. Technol.* **2021**, *137*, 106600. [CrossRef]
28. Villamizar, M.; Garcés, O.; Castro, H.; Verano, M.; Salamanca, L.; Casallas, R.; Gil, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In Proceedings of the 2015 10th Computing Colombian Conference (10CCC), Bogotá, Colombia, 21–25 September 2015; pp. 583–590. [CrossRef]
29. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and Linux containers. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA , 29–31 March 2015; pp. 171–172. [CrossRef]
30. Tapia, F.; Mora, M.A.; Fuertes, W.; Aules, H.; Flores, E.; Toulkeridis, T. From Monolithic Systems to Microservices: A Comparative Study of Performance. *Appl. Sci.* **2020**, *10*, 5797. [CrossRef]
31. Di Stefano, A.; Gollo, M.; Morana, G. *Forthcoming*. An SLA-driven, AI-based QoS Manager for Controlling Application Performance on Edge Cloud Continuum. In Proceedings of the 2024 IEEE International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), Reggio Emilia, Italy, 26–29 June 2024.
32. Rabenstein, B.; Volz, J. *Prometheus: A Next-Generation Monitoring System (Talk)*; SoundCloud Ltd.: Dublin, Ireland, 2015.
33. Red Hat, Inc. Ansible. Available online: https://ansible.com (accessed on 1 July 2024).
34. HashiCorp, Inc. Terraform. Available online: https://terraform.io (accessed on 1 July 2024).
35. Chaudhary, A.; Kolhe, S.; Kamal, R. An improved random forest classifier for multi-class classification. *Inf. Process. Agric.* **2016**, *3*, 215–222. [CrossRef]
36. Shyalika, C.; Wickramarachchi, R.; Sheth, A. A Comprehensive Survey on Rare Event Prediction. *arXiv* **2023**, arXiv:2309.11356.
37. Mikkilineni, R. The Science of Information Processing Structures and the Design of a New Class of Distributed Computing Structures. *Proceedings* **2022**, *81*, 53. [CrossRef]
38. Désir, C.; Bernard, S.; Petitjean, C.; Heutte, L. A New Random Forest Method for One-Class Classification. In *Structural, Syntactic, and Statistical Pattern Recognition*; Gimel'farb, G., Hancock, E., Imiya, A., Kuijper, A., Kudo, M., Omachi, S., Windeatt, T., Yamada, K., Eds.; Elsevier: Berlin/Heidelberg, Germany, 2012; pp. 282–290.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.