Matching Computing Requirements of Stochastic Optimization Models and Cloud Computing Resources

Pedro H. Bernardino[1 2], **Daniel Sadoc Menasche**[2], **Mario Veiga Pereira**[1]

[1]PSR - Energy Consulting and Analytics, Rio de Janeiro – RJ

[2]Instituto de Computação – UFRJ, Rio de Janeiro – RJ – Brazil

Abstract. *Cloud computing offers scalable solutions for scientific computing, but efficiently allocating resources for stochastic optimization models remains challenging. This work uses real execution data from an energy sector company to develop machine learning models that predict execution time based on algorithm parameters and cloud infrastructure configurations. To optimize resource usage, we propose a utility-based framework that balances execution time and cloud costs. Our results highlight key factors affecting computational efficiency and provide insights for cost-effective resource provisioning, improving cloud utilization for stochastic optimization applications.*

1. Introduction

Cloud computing has become a *de facto* solution for addressing the demands of scientific computing. By enabling scalable resource allocation based on user requirements, it simplifies management and can reduce costs. However, aligning the diverse computing needs of various workloads with cloud resources remains a continuous challenge, especially as both demand and resource availability tremendously grow over the past few years.

In this work, we leverage real data from a production environment to address the following research question: *How should users of stochastic optimization models allocate cloud resources in order to achieve the best balance between performance (execution time) and cost (e.g., measured in the use of memory and CPU)?*

To address this question, we use data from PSR Energy Consulting and Analytics. PSR is one of the leading companies in the Brazilian and international energy market, and has been at the forefront of integrating cloud computing into scientific applications. In 2006, after a note about Amazon Web Services (AWS) was published at Nature [Butler 2006], PSR became the first enterprise in Latin America to adopt AWS, initiating their first run in November 2006. This early adoption enabled PSR to scale their computational resources efficiently.

Over the years, PSR's cloud utilization has grown significantly. Notably, they employed 30,000 cores to analyze the Pacific Northwest's energy system, making PSR the largest AWS user globally for two weeks.[1] Subsequently, in the field of cosmology, AWS usage peaked at 250,000 cores. This progression underscores the escalating demands for computational resources in scientific research and the critical need for efficient cloud resource allocation strategies.[2]

[1]`https://www.psr-inc.com/en/news/official-energy-planning-for-the-pacific-northwest-system-uses-psr-models/`

[2]The use of AWS by PSR is showcased by AWS itself as one of its leading case studies: `https://aws.amazon.com/solutions/case-studies/psr/`

Among the optimization algorithms for forecasting energy demand scenarios, the Stochastic Dual Dynamic Programming (SDDP) model stands out as it solves complex stochastic optimization problems by decomposing them into simpler subproblems and segmenting the problem into multiple stages, enabling the derivation of an efficient operational policy [Pereira and Pinto 1991]. The SDDP model takes as input a comprehensive set of parameters, including characteristics of the studied case, such as time horizon, number of systems and scenarios, level of simulation detail (in some cases, managing energy on an hourly scale), as well as data on thermal, renewable, and hydroelectric power plants, battery storage, energy demand scenarios, and operational constraints. The complexity of these inputs directly impacts both the model's results and its computational runtime.

Since the SDDP model is primarily executed on high-performance cloud computing machines, optimizing resource utilization is crucial to minimizing operational costs. In this context, this project aims to analyze the performance of multiple algorithm executions, considering both SDDP input parameters and the specifications of the AWS computing machines used. The objective is to develop an approach to assess execution efficiency, taking into account not only processing time but also the financial impact associated with computational resource usage.

Consulting companies such as PSR typically have external and internal users. External users are clients that use the software tools, such as SDDP optimizer, by themselves, whereas internal clients are employees of the consulting company that run the full pipeline and present to clients the final solution report. The costs of executions for internal and external users are disclosed to final users in both cases, indicating the relevance of understanding the tradeoff between cost and performance in both scenarios. In this work, we consider runs that were executed by internal users.

While answering our key question, we provide the following contributions:

Predicting workload execution time using production data. We use execution logs of a real production system to develop machine learning models capable of estimating execution time, helping users anticipate resource consumption and optimize cloud usage.

Evaluating the impact of model (SDDP) and cloud (AWS) parameters. We conduct a detailed sensitivity analysis to determine the influence of algorithmic parameters and cloud infrastructure configurations on execution time.

Developing a utility-driven approach for cost-performance trade-offs in cloud computing. We introduce a structured methodology that allows users to systematically evaluate and optimize the allocation of cloud resources, balancing execution time and cost efficiency in stochastic optimization workflows.

Outline. Section 2 covers data preprocessing, including the matching of SDDP model parameters with AWS cloud attributes. Section 3 presents execution time prediction and Section 4 analyzes sensitivity to key factors. Section 5 introduces a utility-based resource allocation approach, Section 6 reviews related work, and Section 7 concludes.

2. Data Extraction, Organization, Processing, and Cleaning

The machine learning pipeline in this work consists of three steps: data extraction, algorithm parameterization, and result analysis. Initially, data is stored in two databases: an SQL database for SDDP algorithm parameters and an Oracle database for PSR cloud computing execution data at AWS. Once consolidated, the data undergoes cleaning and formatting before testing various machine learning models.

2.1. Data Extraction

Each execution initially contains hundreds of attributes, increasing complexity in processing time and memory.

Useful data. Execution data, including attributes from the SDDP algorithm and cloud processing system, is recorded only after a run completes. Consequently, certain attributes cannot be used for training, such as those tracking specific stage durations. The total execution time, in particular, serves as the target variable, and is used for testing the trained models.

Irrelevant and non-anonymized data. Attributes that do not impact execution performance, such as system messages or timestamps, are removed. Additionally, any data that could identify users or projects is excluded.

After filtering, the remaining attributes are categorized into two categories: **SDDP Model Attributes:** Number of generation agents, stopping criteria, and other SDDP modeling parameters. **AWS Machine Attributes:** Number of cores, RAM per process, and other cloud computing resource characteristics.

2.2. Matching Model and Machine Attributes

A key challenge is linking records between the SDDP model (SQL database) and AWS cloud infrastructure (Oracle database). This is done using execution timestamps and user IDs, which are consistent across both systems. To accommodate minor variations in timestamps, a 1-second tolerance is applied.[3]

Matching large sets of execution records introduces memory limitations. To address this, the data is partitioned into smaller batches. For n batches of algorithm executions and m batches of cloud configurations, $n \times m$ `join` operations are required. Although necessary, this increases processing time due to repeated access to overlapping data. To mitigate this, matched results are cached for reuse when inputs remain unchanged.

2.3. Data Processing and Cleaning

Preprocessing ensures the dataset meets the requirements for supervised learning, where input features are mapped to known outputs.

Cleaning process: Irrelevant columns, such as unique IDs and metadata, were removed using the *drop* function from *pandas*. The target variable *run_duration*, originally in "HH:MM:SS" format, was converted to seconds using *pd.to_timedelta* and *dt.total_seconds*.

Handling missing values and categorical variables: Columns with more than 75% missing values and records containing *NaN* were removed. Categorical features like *model_version* were transformed using one-hot encoding via *pd.get_dummies*, allowing numerical representation of categories.

[3]Increasing this tolerance raises the match count but risks false matches. Future work will adopt a unique execution identifier to simplify this process.

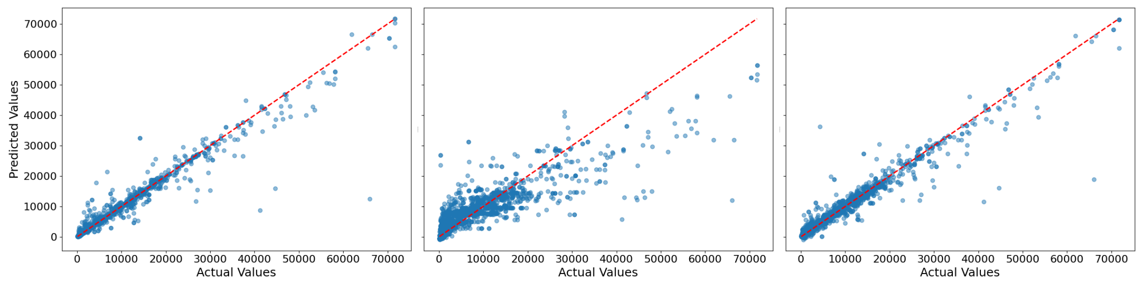


Figure 1. Scatter plots for Random Forest, Gradient Boosting and XGBoost models, respectively.

Standardization: Features were normalized using *StandardScaler* from *scikit-learn*, which centers data to zero mean and unit variance—ensuring all features contribute proportionally during model training.

3. Predicting the Execution Time

In this study, we assess model performance using standard regression metrics: the mean value of predictions, root mean squared error (RMSE), mean absolute error (MAE), mean absolute percentage error (MAPE), and the coefficient of determination (R^2). RMSE and MAE measure the magnitude of prediction errors, while MAPE expresses these errors as a percentage. The R^2 score indicates how well the model explains the variance in execution time, with values closer to 1 signifying better performance (see Table 1).

Table 1. Regression model evaluation metrics for predicting execution time (in seconds). The best values for each metric are highlighted in bold.

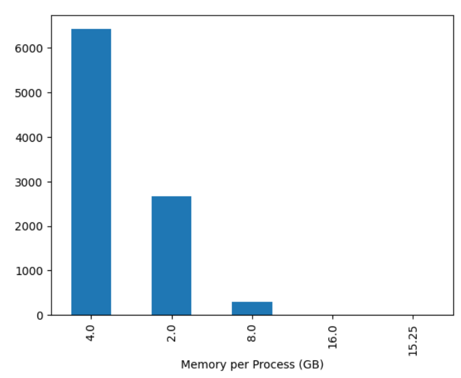
Metric	Random Forest	Gradient Boosting	XGBoost
Mean Predicted Time (s)	8337.94	8319.18	8368.35
RMSE (s)	**2361.22**	5343.56	2452.31
MAE (s)	**703.42**	3130.08	1027.31
MAPE (%)	**14.77%**	128.21%	29.46%
R^2 Score	**0.948**	0.733	0.944

To visually analyze model performance, we use scatter plots, which compare actual versus predicted execution times (Figure 1), and also leverage feature importance, which highlight the most influential attributes (Table 2).

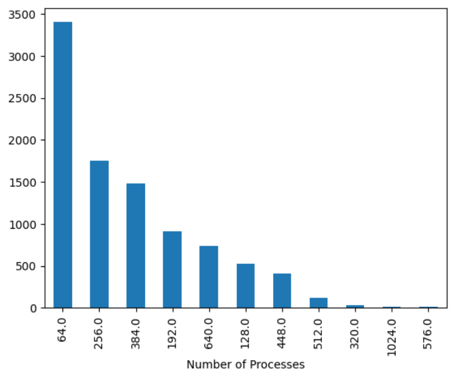
The models evaluated were RandomForestRegressor, GradientBoostingRegressor, and XGBRegressor. The dataset used for testing contains 217 features, with a training set of 7,516 samples and a test set of 1,879 samples. The training set has an average execution time of 8,250 seconds, while the test set averages 8,410 seconds.

Random Forest. Random Forest is an ensemble method that builds multiple decision trees and averages their predictions to improve accuracy and reduce overfitting. It showed the best overall performance, with strong alignment between predicted and actual execution times. Feature importance analysis highlights the relevance of SDDP model parameters in determining execution time.

Gradient Boosting. Gradient Boosting builds trees sequentially, where each new tree corrects errors from the previous ones. Despite this, it performed the worst, with more



(a) Memory per Process (GB)



(b) Number of Processes

Figure 2. Distribution of attributes: (a) memory/process; (b) # of processes.

variability in predictions and less emphasis on relevant features, indicating difficulty in capturing execution time patterns.

XGBoost. XGBoost is an optimized version of Gradient Boosting, introducing regularization and parallel processing for improved performance. It performed closely to Random Forest, showing reliable predictions, though with slightly more errors for longer executions.

Discussion. Overall, Random Forest and XGBoost provided strong predictive performance (see scatter plots in Figure 1 and evaluation metrics in Table 1), allowing us to derive some key observations. First, all models exhibit increased errors for longer iterations. This is expected, as absolute error grows with larger iteration sizes, and the limited availability of samples for very large cases further exacerbates this effect. Second, Random Forest and XGBoost achieved significantly higher accuracy compared to Gradient Boosting, as reflected in their lower RMSE and higher R^2 scores. Third, the XGBoost model did not seem to account for the number of processes, while Gradient Boosting placed little emphasis on memory per process, potentially contributing to their respective inaccuracies.

4. Sensitivity Analysis

We observe that attributes related to the SDDP algorithm exhibit significantly greater importance compared to cloud-related attributes, such as the number of cores and memory (see Table 2). In this section, we aim to answer the following question: why do the algorithm parameters hold greater relevance than the cloud parameters?

Based on this observation, we can formulate some hypotheses. First, the sample of different values for these fields may have been insufficient for the model to detect their influence on iteration runtime. Second, these factors may indeed have had less impact on iteration performance than the model parameters. A third hypothesis is that the machine parameters were optimally balanced with the complexity of the case, making the machine parameters appear as a dependent variable of the SDDP algorithm's complexity from the perspective of the machine learning model. We will now evaluate these possibilities.

Hypothesis 1: Low variability of cloud parameters in the training set. Figure 2 reports the distribution of machine-related parameters. By analyzing the distribu-

Table 2. Feature importance (normalized) for execution time prediction, grouped by category. SDDP-related features consistently dominate across all models.

Category	Feature Name	XGBoost	Gradient Boosting	Random Forest
SDDP	`nffr`	0.075	0.160	0.140
	`model version 17.3.11rc5`	0.070	0.110	0.120
	`nsim`	0.040	0.070	0.075
	`nrag`	0.035	0.060	0.060
	`model version 17.3.5`	0.029	0.030	0.030
	`ncpl percentage total`	0.025	0.026	0.020
AWS	Memory per Process (GB)	0.020	0.001	0.015
	Number of Processes	0.001	0.020	0.010

tion of these parameters, it appears that their variability is not so limited that the model would completely disregard the machine configurations. This may have contributed to the lower importance of machine-related attributes, but not to the extent of being the sole determining factor.

Hypothesis 2: Cloud parameters are unconditionally irrelevant. This hypothesis is not strongly supported, as the relevance of cloud machine attributes varied across all tested machine learning models. Notably, the most accurate model, Random Forest, was the one that considered both parameters (memory per process and number of processes), whereas the other two models disregarded one parameter each.

Another approach to testing this hypothesis would be to analyze a subset of data where SDDP algorithm parameters have minimal variation, in a complex case with a long execution time, and examine how execution time fluctuates with changes in machine attributes. We leave such analysis as subject for future work.

Hypothesis 3: Cloud parameters were optimized by the algorithm execution team, leading to correlation with model parameters. This hypothesis suggests that the users responsible for these executions generally possess sufficient expertise to select the most efficient parameters for each case. As a result, cloud parameters may act as a dependent variable of the algorithm's attributes. In other words, whenever the algorithm demands more complexity and execution time, a proportionally more powerful machine is likely chosen, thereby maintaining a stable execution time relative to simpler cases.

Under this hypothesis, the cloud parameters are conditionally irrelevant given the SDDP-related parameters. Indeed, in the extreme case wherein users set AWS-related parameters, and those parameters are functionally determined by the SDDP-related parameters, the latter is sufficient to predict execution time.

To validate this hypothesis, we will need to leverage our execution time prediction model for simulations and analyze the impact of perturbations in cloud parameters. Additionally, we will conduct a new efficiency analysis of cloud resource utilization. These aspects will be further explored in the next section.

5. Resource Allocation Using Utility Functions

In this section we consider the problem of resource allocation using utility functions. First, we parametrize a measure of cost in USD/minute, representing the cost per minute of execution (Section 5.1). Then, we introduce our utility function (Section 5.2). This is followed by a methodology to categorize executions (Section 5.3), results on the categorization of real production data (Section 5.4) and counterfactual analysis (Section 5.5).

5.1. Assessing Cost per Minute

The cost of running an AWS instance is inherently complex, influenced by multiple factors such as the selected usage model, regional demand for computing resources, and even variations between machines with identical specifications but different component manufacturers. However, since this complexity falls outside the primary scope of this study, we simplify the AWS cost estimation by modeling it as a linear function of the number of processes and the total memory usage (in GB) during execution. Specifically, we approximate the cost as:

$$C = k_1 N + k_2 M, \tag{1}$$

where N denotes the number of processes utilized by the machine, and M represents the total memory (in GB) allocated during execution.

This cost-per-minute estimate enables us to define a function that evaluates the efficiency of computational resource utilization, ensuring a more streamlined analysis, as further discussed in the sequel.

5.2. Utility Function

The purpose of the utility function is to quantitatively assess the efficiency of cloud resource utilization for a given execution. More specifically, it evaluates whether the computational power of the selected machine is appropriately matched to the complexity of the algorithm.

The first step is to analyze the dependence of this function on the relevant variables. Execution efficiency should be inversely proportional to both execution time and cost per unit of time. An optimal execution is one that minimizes both factors, achieving a balanced trade-off between speed and resource consumption.

Let C denote the cost per unit of time and T the execution time. We initially define utility as:

$$U = 1/(T \cdot C). \tag{2}$$

However, in certain scenarios, the most cost-efficient machine may not be the most suitable choice. For instance, in time-sensitive applications, obtaining results as quickly as possible may take precedence over cost considerations. To account for this, we introduce priority weights α and β to balance the trade-off between execution time and cost.

To avoid numerical instability caused by extremely small values of T and C, we replace the division with a summation of logarithmic terms. Since both variables are inversely proportional to utility, we express their logarithms as negative terms in the utility function.

Additionally, normalization is applied to T and C to ensure that absolute magnitudes do not dominate the comparison, focusing instead on relative differences. We employ min-max scaling, transforming each variable into the range $[0, 1]$ as follows:

$$\tilde{X} = \frac{X - X_{\min}}{X_{\max} - X_{\min}}, \tag{3}$$

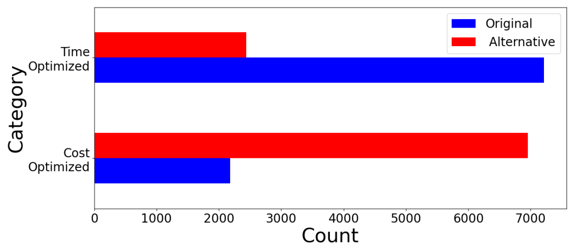


Figure 3. Category distribution for the original setup, in blue (Section 5.4) and alternative setup, in red (Section 5.5)

where X represents the original variable, and $X_{\min}$ and $X_{\max}$ denote its minimum and maximum values within the dataset, respectively.

Incorporating these considerations, the final utility function is given by:

$$\hat{U} = \log \tilde{U} = -\alpha \log \tilde{T} - \beta \log \tilde{C}. \tag{4}$$

A key insight from this formulation is that we can leverage a neural network to extrapolate T values for configurations that have not been explicitly tested. This enables us to estimate the T curve across different C values, evaluate the utility of each configuration, and identify optimal choices for specific values of α and β. [4]

5.3. Categorizing Executions

For this study, we classified execution priorities into the following categories:

- **Cost-optimized:** $\alpha_C = 0.2, \beta_C = 0.8$. Cost-optimized runs prioritize cost-effectiveness, focusing on completing the execution with a lower-cost machine rather than minimizing execution time.
- **Time-optimized:** $\alpha_T = 0.8, \beta_T = 0.2$. Time-optimized runs, on the other hand, are more urgent runs that over-provision computing resources to ensure timely job completions.

For each execution in our dataset (training + testing), the utility function was assessed using the weights of both categories, and the highest resulting value, along with the corresponding category, was recorded.

$$\hat{U}_{i,C} = -\alpha_C \log \tilde{T}_i - \beta_C \log \tilde{C}_i, \quad \hat{U}_{i,T} = -\alpha_T \log \tilde{T}_i - \beta_T \log \tilde{C}_i \tag{5}$$

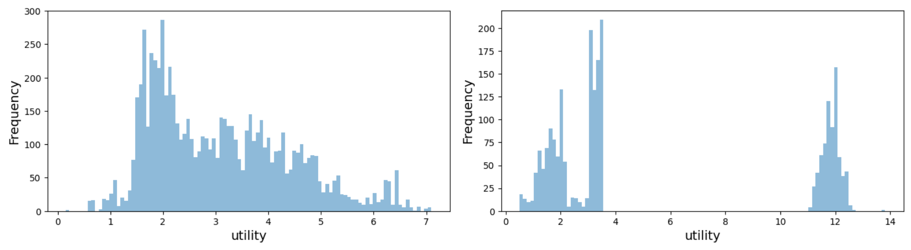
We denote by $\hat{U}_i$ the maximum achievable utility at run i,

$$\hat{U}_i = \max(\hat{U}_{i,C}, \hat{U}_{i,T}). \tag{6}$$

Let σ_i be the class of run i, $\sigma_i \in \{C, T\}$,

$$\sigma_i = \begin{cases} C, & \text{if } \hat{U}_{i,C} \geq \hat{U}_{i,T} \text{ (run } i \text{ classified as a cost-optimized)} \\ T, & \text{if } \hat{U}_{i,C} < \hat{U}_{i,T} \text{ (run } i \text{ classified as a time-optimized)} \end{cases} \tag{7}$$

[4] In this work, we use a neural network to estimate T as a function of C for (T, C) pairs not present in our dataset. In future research, we plan to derive analytical approximations for $T(C)$, allowing us to directly solve the utility maximization problem as a single-variable optimization of $U(C)$.



(a) Time-Optimized Runs **(b)** Cost-Optimized Runs

Figure 4. Utility for (a) time-optimized and (b) cost-optimized runs.

Note that our classification assumes that cloud configurations are selected to maximize, either explicitly or implicitly, the considered utility function. In what follows, we assess how actual users choices are reflected by this model, given historical data of available runs (Section 5.4), followed by a counterfactual what-if analysis considering alternative runs and cloud parameters beyond those present in our dataset (Section 5.5).

5.4. Reverse-Engineering Utility: Do Users Tend to Prioritize Cost or Time?

Next, we investigate which utility function best aligns with the data observed in our traces.

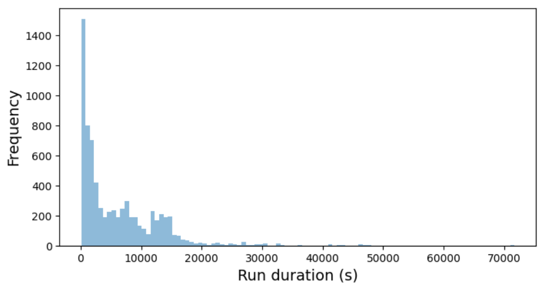
We have detailed information on the configuration of each execution, including the selected SDDP model parameters and cloud infrastructure choices. Assuming that system users had a rough expectation of execution time and an implicit understanding of the trade-off between time and cost, we can infer whether their decisions prioritized one factor over the other.

To simplify the analysis, we focus on the two utility functions defined in the previous section: cost-optimized and time-optimized. We then determine which of these functions better fits the observed data by evaluating which one yields a utility maximum value, to identify if this execution focuses on saving cloud resources or getting the results fast not prioritizing the run cost.

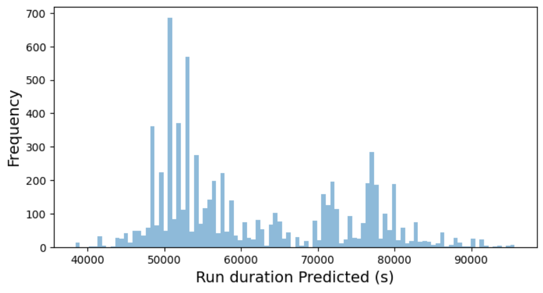
Time-optimized versus cost-optimized. In our analysis, we observed a shift in execution priorities based on the available alternatives. Initially, in the first setup, the majority of executions were time-optimized (see blue bars in Figure 3). However, given the nature of these executions (test iterations and runs with planned time scope) it does not seem reasonable that the vast majority falls under the time-optimized category.

To verify if executions used oversized machines, we introduced an alternative option that allowed users to leverage cheaper machines (further discussed in Section 5.5), providing an opportunity to reassess their optimization choices. The results showed a significant shift: cost optimization became the dominant choice (see Figure 3). This shift in execution patterns highlights the impact of available alternatives on user decisions. With limited options, users prioritized time efficiency, likely due to defaults or perceived constraints. However, access to cost-efficient choices led to adaptive decision-making, suggesting that initial time-focused executions were over-provisioned and could have been optimized with better resource allocation.

Histogram of utility. Fig. 4 shows utility histograms for (a) time-optimized and (b) cost-optimized runs. Most time-optimized runs have high utility, but some used ex-



(a) Execution Time (Section 5.4)



(b) Execution Time (Section 5.5)

Figure 5. Comparison of execution time histograms for time-optimized runs before (a) and after (b) exploring alternative cloud configurations.

cessive resources without much gain in speed (left side of Fig. 4(a)). Cost-optimized runs fall into two groups: well-provisioned runs with high utility (right side of Fig. 4(b)), and under-provisioned ones corresponding to long execution times (left side of Fig. 4(b)).

5.5. Exploring Alternative Cloud Configurations

Thus far, our analysis has been based on configurations actually selected by users, i.e., configurations that are present in our dataset. However, what if future users had access to a broader range of cloud resource options? To explore this, we simulate an alternative scenario by modifying resource allocations and assessing its impact on execution time and utility.

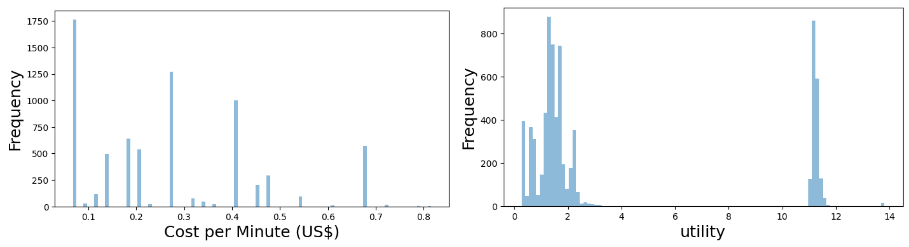
We focus on executions classified as time-optimized and reduce their allocated memory and number of processes by half. Using our trained model, we estimate the corresponding execution times for these new configurations and recompute their utilities based on cost and execution time. By doing so, we produce new alternatives for future users, which are analyzed below.

Simulating Execution Time for Runs. We use the pre-trained Random Forest Regressor to estimate the execution time of the new alternatives. The execution time across all runs before considering the additional alternative option is reported in Figure 5(a). After reducing the number of processes by half, and adding it as an alternative, we obtained the results reported in Figure 5(b).

As expected, execution time increased significantly. A key point to highlight is that most initial executions used 64 processes, and when halved, they were run with 32 processes. The machine learning model lacked prior execution data for 32-process configurations, which may affect its ability to accurately simulate runs on these unrecorded setups. Nonetheless, the model correctly captured the inverse relationship between execution time and both the number of processes and available memory.

Since the simulated cost had both parameters halved, the cost distribution remains roughly unchanged in shape, with all values approximately half of their original amounts. The resulting distribution is reported in Figure 6(a). Due to space limitations, the original distribution is not shown here but it roughly corresponds to Figure 6(a) with the values in the x-axis doubled.

Assessing Utilities Given Alternative Cloud Configurations. Analyzing the



(a) Estimated Cost (Section 5.5) **(b)** Utility Histogram

Figure 6. (a) Estimated cost histogram for time-optimized runs after exploring alternative cloud configurations; (b) Utility histogram: Time-optimized runs in scenario of Section 5.5.

new distribution of the utility function in Figure 6(b), we observe that time-optimized runs have split into two distinct subgroups. In theory, this division explicitly distinguishes runs that benefited from the machine reduction from those that were negatively impacted. Some runs experienced a slight increase in execution time, which was compensated by the lower cost, while others saw a drastic execution time increase, reducing overall efficiency despite the decreased cost per minute.

Another notable observation is the reassignment of execution categories. Recall that before the machine capacity reduction, most executions were classified as time-optimized. Now, some of those runs prioritize resource savings over execution time, lowering operational costs for executions that do not require absolute time prioritization (see discussion on time-optimized versus cost-optimized runs in the beginning of Section 5.4).

6. Related Work

In this section, we discuss key research efforts related to execution time estimation in cloud environments and the application of machine learning for resource allocation.

Execution Time Estimation. Accurately predicting execution time is crucial for optimizing resource utilization and cost efficiency in cloud computing. Zhang et al. [Zhang et al. 2024] proposed a quality-driven approach for cloud service selection, emphasizing the importance of performance prediction in service-oriented environments. Similarly, [Delimitrou and Kozyrakis 2014] introduced Quasar, a system that leverages workload characterization to improve resource management decisions, thereby enhancing execution time estimations.

Autonomic Computing. Tesauro [Tesauro 2007] presented a manifesto highlighting the potential of machine learning, and reinforcement learning in particular, in autonomic computing, supported by case studies demonstrating its effectiveness in real-time self-management scenarios. This work underscores the adaptability of RL techniques in dynamic cloud environments, enabling systems to learn optimal policies for resource allocation. In this work, we consider a utility-driven approach for resource allocation, leaving dynamic aspects as subject for future work.

Resource Allocation in Cloud Environments. Efficient resource allocation is vital for cloud service providers to meet service level agreements (SLAs) while maximizing resource utilization. [Menache et al. 2014, Perez-Salazar et al. 2022] addressed this challenge by developing a dynamic resource allocation framework that achieves near-optimal efficiency. Their approach balances the trade-off between resource utilization and SLA compliance, providing a practical solution for cloud resource management.

Additionally, Babaioff et al. [Babaioff et al. 2017] introduced the Economic Resource Allocation (ERA) framework, which integrates economic principles into cloud resource scheduling and pricing. ERA aims to enhance resource usage efficiency by aligning pricing strategies with demand predictions, offering a flexible layer that can operate atop various cloud infrastructures.

These studies collectively contribute to the advancement of execution time estimation and resource allocation strategies in cloud computing, highlighting the role of machine learning and economic models in enhancing cloud service efficiency. In this work, we use real production datasets to further indicate the feasibility of execution time predictions and we use those predictions to parametrize a utility-driven approach for resource allocation.

7. Conclusion and Future Work

This work proposed a machine learning approach to estimate execution time for stochastic optimization models in cloud environments, using both algorithm parameters and cloud infrastructure configurations. Additionally, we introduced a utility-based method to evaluate the trade-off between cost and execution time, facilitating efficient resource allocation.

Our results highlight the strong influence of SDDP model parameters on AWS execution time and show that optimizing the interplay between these parameters and cloud configurations can improve resource allocation. The utility analysis further revealed that some executions could lower costs by considering cheaper machines. As future work, we aim to develop an automated recommendation system to help users select optimal execution configurations, improving cost-performance trade-offs and making cloud resource allocation more intuitive and efficient.

References

Babaioff, M. et al. (2017). ERA: A framework for economic resource allocation for the cloud. In *International Conference on World Wide Web Companion*, pages 635–642.

Butler, D. (2006). Amazon puts network power online. *Nature*, 444:528.

Delimitrou, C. and Kozyrakis, C. (2014). Quasar: Resource-efficient and qos-aware cluster management. In *Conf. Architectural Support for Programming Lang. and Operating Systems (ASPLOS)*, pages 127–144.

Menache, I., Shamir, O., and Jain, N. (2014). On-demand, spot, or both: Dynamic resource allocation for executing batch jobs in the cloud. In *ICAC*, pages 177–187.

Pereira, M. V. and Pinto, L. M. (1991). Multi-stage stochastic optimization applied to energy planning. *Mathematical programming*, 52:359–375.

Perez-Salazar, S., Menache, I., Singh, M., and Toriello, A. (2022). Dynamic resource allocation in the cloud with near-optimal efficiency. *Operations Research*, 70(4):2517–2537.

Tesauro, G. (2007). Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing*, 11(1):22–30.

Zhang, W., Li, M., and Chen, X. (2024). Quality-aware resource allocation in cloud computing: A machine learning approach. *IEEE Transactions on Cloud Computing*, 12(1):34–47.