Intelligent Optimization of Distributed Pipeline Execution in Serverless Platforms: A Predictive Model Approach

Group Members:
- **Chris Biju**
- **Sparsh Jha**

TCSS 562 Au 25 (Software Engineering for) Cloud Computing

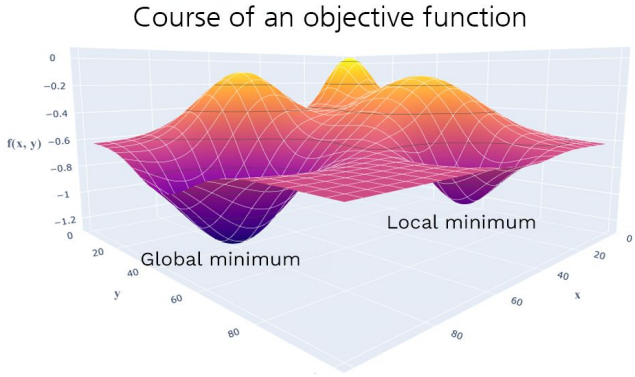
Motivation

- Serverless pipelines are powerful
- But performance varies wildly
- Hard to tune configurations
- Brute-force testing is expensive

Goal of the Paper

Goal:

Predict a pipeline's execution time without running hundreds of configurations, using XGBoost → pick optimal setup faster & cheaper.

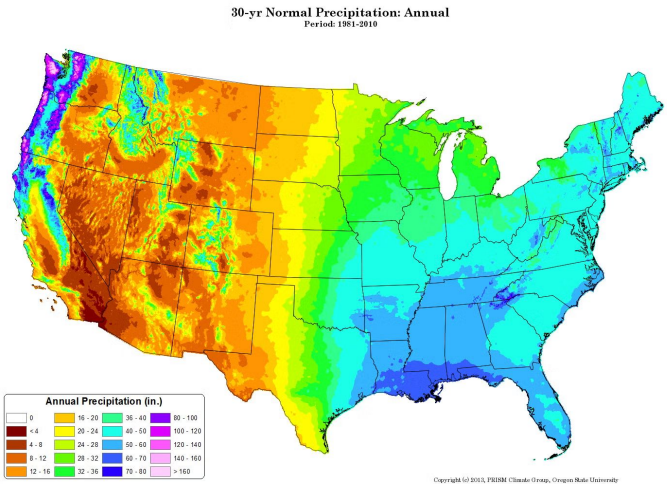


Background

- XGBoost:
  - Builds many small decision trees sequentially; each fixes prior errors.
  - Excellent at learning complex patterns in tabular data with many features.
- Hyperparameter Tuning:
  - The process of automatically searching for optimal model settings (e.g., depth, learning rate).
  - Strongly impacts model accuracy and performance.
- Feature Engineering:
  - Creating new input variables (e.g., memory-per-file) to expose hidden relationships.
  - Helps ML models learn patterns that raw data alone cannot reveal.

What Does This Pipeline Do?

Geospatial water consumption analysis pipeline.

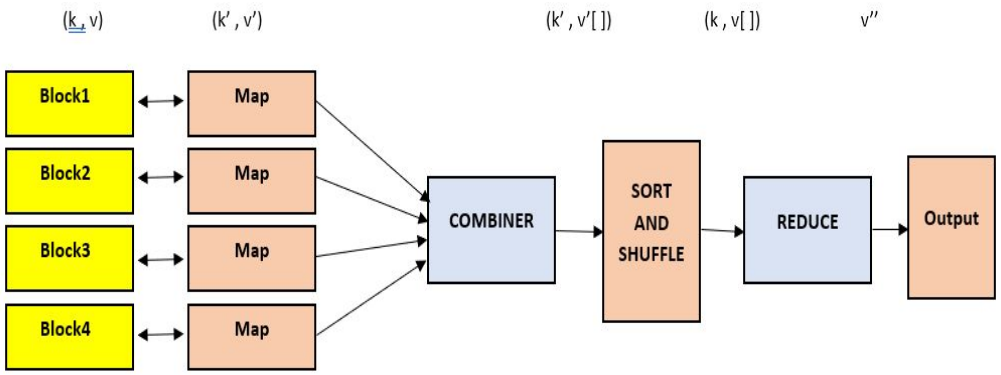


Serverless Architecture Overview

- **Lithops** orchestrates Python functions
- AWS Lambda executes them in parallel
- Highly scalable

How Parallel Execution Works

Data is split into chunks and each chunk is processed by Lambda.



Design Space Analysis

- The authors ran 148 pipeline executions, systematically varying memory, splits, storage, input sizes, and various execution parameters.
- This created a broad configuration landscape to understand how each parameter affects performance.
- DSA provides real ground truth but is expensive, slow, and scales poorly for large pipelines.

Design Space Analysis

Parameters Adjusted During the DSA.

- Splits: Range of 2 to 6, to balance parallelism. More than 6 adds unnecessary overhead, while fewer than 2 limits efficiency.
- Allocated Memory: 1,024 MB to 3,008 MB (maximum in Lithops). Less than 1,024 MB was insufficient for the data used, leading to suboptimal performance.
- Ephemeral Storage: 512 MB to 8,192 MB, varied to support different temporary storage needs.
- vCPUs: Indirectly set by allocated memory in AWS Lambda (0.85 to 1.61 vCPUs). Not directly controlled, but adjusted automatically based on memory.
- Input Files and Size: Configurations with 5 or 15 files, ranging from 0.25 to 1 GB, based on a real-world use case analyzing water consumption in Murcia.

Why DSA Alone Is Not Enough

- Although DSA reveals true optimal configurations, it requires many full pipeline runs, making it impractical for frequent tuning.
- Parameter interactions are non-linear and complex; manual tuning cannot capture these dependencies.
- A predictive model is needed to reduce expensive experimentation while keeping accuracy.

Dataset: Inputs + Engineered Features

- The model uses both raw parameters (memory, splits, data size) and derived features like memory-per-file, storage-per-GB, threads-per-worker.
- Feature engineering helps expose hidden relationships in parallel performance.
- All 148 runs form a dataset capturing diverse real-world scenarios.

Dataset: Inputs + Engineered Features

Table 2: Derived Parameters from Feature Engineering

Derived Parameter	Description
memory_per_file	Memory allocated per file processed (MB)
storage_per_file	Temporary storage per file (MB)
vcpus_per_file	vCPUs allocated per file
files_per_vcpu	Number of files processed per vCPU
size_per_file	Size of each file (GB)
memory_per_gb	Memory allocated per GB of input size
vcpus_per_gb	vCPUs allocated per GB of input size
storage_per_gb	Temporary storage per GB of input size (MB)
threads_per_worker	Threads running per worker process
memory_per_thread	Memory allocated per thread (MB)
vcpus_per_thread	vCPUs allocated per thread
memory_per_thread_vcpus_ratio	Ratio of memory to vCPUs per thread

Preprocessing & Transformations

- Data was normalized, outliers kept (to help model learn inefficient configs), and Gaussian noise was injected for generalization.
- Execution time was log-transformed to reduce variance and stabilize predictions.
- A 70/30 train–test split ensures the model learns patterns without overfitting.

Model: XGBoost + Optuna Optimization

- XGBoost was chosen for its ability to capture nonlinear interactions across many numerical features.
- Optuna performed Bayesian hyperparameter tuning to find optimal depth, learning rate, regularization, and subsampling.
- This combination yields a robust, high-accuracy predictor even with a relatively small dataset.

Model: XGBoost + Optuna Optimization

Table 3: Best Hyperparameters Found Using Optuna

Hyperparameter	Value
Max Depth	4
Learning Rate	0.005193
Number of Estimators	2268
Subsample	0.7467
Colsample by Tree	0.9654
Gamma	0.0101
L1 Regularization (Reg_Alpha)	0.0914
L2 Regularization (Reg_Lambda)	0.1549

Model Performance: Accuracy Gains

- XGBoost significantly outperformed baseline models:
  - 75.34% lower MAE than the average baseline
  - ~69% improvement over linear & PCA-based regression
- The model captures complex non-linear interactions that linear and PCA-based regression fail to capture.

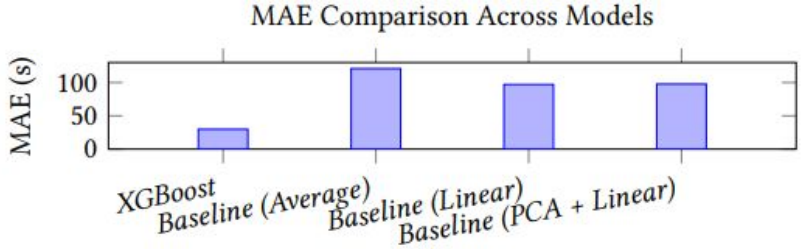


Figure 3: Mean Absolute Error (MAE) comparison across models.

Predicting Optimal Configurations

- Using the trained model, the authors predicted execution time for unseen configurations.
- Results showed optimal performance at 5 splits and 2048 MB memory, aligning with intuition yet discovered automatically.
- This allows fast identification of ideal parameters without rerunning the full pipeline.

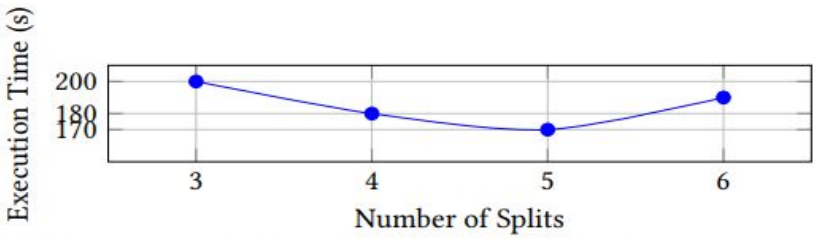


Figure 1: Predicted execution times for different splits.

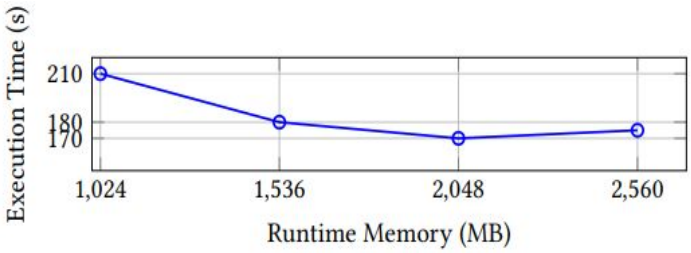


Figure 2: Predicted execution times for different memory allocations.

Cost Analysis: Real Savings

- The best configuration reduces execution time by up to 79.9% and runtime cost by ~20%.
- Savings accumulate across repeated runs, making predictive tuning worthwhile.
- With a training cost of $38.75, break-even occurs after ~562 executions (~2 months at 10 runs/day).

Table 7: Configuration Comparison: Minimum vs. Maximum Duration and Costs

Parameter	Minimum Duration	Maximum Duration
Number of Files	5	5
Splits	5	2
Input Size (GB)	0.25	0.25
Runtime Memory (MB)	2000	1024
Ephemeral Storage (MB)	1024	1024
vCPUs	1.13	0.58
Duration (s)	184.08	915.89
Cost per Execution (USD)	0.281	0.350
Cost Difference (USD)	0.069	

Long-Term Cost Accumulation

- Because many pipelines run daily or hourly, small per-run savings translate into large long-term gains.
- The model effectively reduces experimentation overhead by ~30%.
- This makes it viable for enterprise-scale workloads and recurring analysis pipelines.

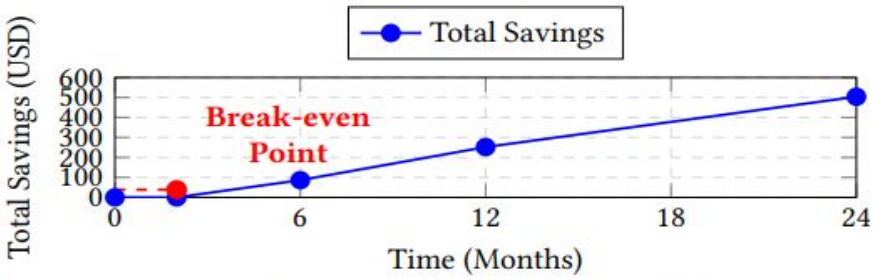


Figure 4: Projected cost savings over time, assuming 10 executions per day. The break-even point occurs at approximately 2 months.

Real vs Predicted Fit

- Predictions closely match real execution times, demonstrating strong generalization.
- The model successfully identifies near-optimal configurations it has never directly observed.
- Residual patterns indicate low bias and effective learning of non-linear patterns.

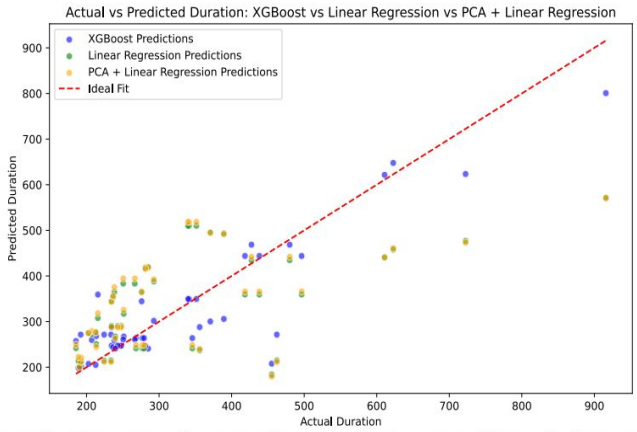


Figure 8: Comparison of actual vs. predicted duration for various models, highlighting the performance of *XGBoost* against simpler regression methods.

Residual Analysis

- Residuals from XGBoost are centered near zero with significantly lower spread than linear models.
- This indicates the model captures important interactions missing from simpler methods.
- The residual distribution validates the reliability of predictions for decision-making.

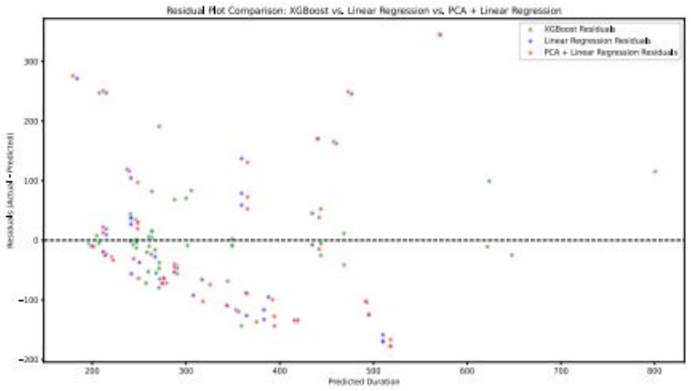


Figure 6: Residual comparison across models: *XGBoost* displays symmetric residuals around zero, suggesting higher accuracy and lower bias.

Highlights

- Predicts optimal configurations without full DSA
- Cuts execution time by up to 79.9% (compared to Full DSA), cost by ~30%
- Reduces number of required experiments by 30%
- Provides a multi-parameter optimization solution applicable across AWS, Azure, & GCP via Lithops

Critique: Strengths

- Strong Practical Relevance
  - Based on a real geospatial water-consumption pipeline.
  - Uses actual distributed serverless execution (Lithops + AWS Lambda).
- Thoughtful Feature Engineering
  - Attempts to capture complex interactions between memory, storage, vCPUs, and data size.
- Transparent and Comprehensive Evaluation
  - Includes model comparisons (baselines vs XGBoost) and multiple diagnostic plots.
  - Honest reporting of negative results (CTGAN, ensembles, feature selection).

Critique: Weaknesses

- Performance Variability Limitations
  - Only 1 observation performed per execution; performance varies in serverless for the same config.
- Narrow Design Space
  - Does not address what percentage of DSA their 148 executions cover.
  - Not representative of the full Lambda configuration spectrum.
- Evaluation Weaknesses
  - Test configurations similar to training ones
  - No variance analysis, or confidence intervals
- Overstated Claims & Cost Analysis Issues
  - “79.9% improvement” compares best vs worst config; not realistic.
  - Cost savings do not account for cold starts, S3 costs, or repeated runs.
  - Claims of “universal applicability” unsupported by cross-pipeline tests.

Future Work

- Strengthen Statistical Rigor
  - Incorporate multiple runs per configuration to capture serverless variance.
  - Add confidence intervals, variance estimates, and significance tests to improve reliability.
  - Model noisy behavior explicitly (e.g., probabilistic regression).
- Expand Dataset & Design Space
  - Run broader DSA across larger memory ranges, more split values, and multiple pipelines. (Costly, but necessary on pipelines with large variability in parameters)
  - Validate generalization across different workloads and cloud providers (Azure, Google Cloud).
- Advanced Optimization Approaches
  - Use ensembles (stacking, blending) to reduce model bias and variance.
  - Apply active learning to selectively explore high-impact configurations with minimal cost.

Q&A