AutoScale-ML with HASA: A Docker-based Framework for Distributed AutoML Model Selection

Md. Attaur Rahman Sofi
Research Scholar
Dept. of Computer Science & Application
P.K. University, Shivpuri, M.P., India

Mohd. Yousuf
Research Scholar
Dept. of Computer Science & Application
P.K. University, Shivpuri, M.P., India

ABSTRACT
This paper presents AutoScale-ML with HASA, a hierarchical adaptive search framework for automated machine learning (AutoML) model selection, implemented within a simulated seven-node distributed computing environment consisting of one master node and six independent Docker containers, each exposing a REST endpoint through Flask. Each worker trains a randomly assigned Scikit-learn classifier drawn from RandomForest, GradientBoosting, ExtraTrees, DecisionTree, and LogisticRegression on a 50,000-sample synthetic classification dataset (50 features, 20 informative) generated via scikit-learn's make_classification, and returns accuracy, training runtime, simulated network delay, and a composite score to a central master process. The master applies a three-phase Hierarchical Adaptive Search Algorithm (HASA): Phase 1 collects all six worker evaluations and retains the top-4 by composite score; Phase 2 re-ranks those four candidates and retains the top-2; Phase 3 selects the single best model by maximum composite score. Experimental results—including per-model benchmarks, phase-by-phase HASA traces, penalty coefficient sensitivity analysis, and network delay characterisation—demonstrate that the framework effectively balances prediction accuracy and computational efficiency through runtime-aware hierarchical model selection. Comprehensive evaluation across five classifier families reveals that the composite scoring function heavily penalises ensemble training times, often favouring lightweight models over higher-accuracy alternatives. The penalty coefficient α is shown to be a critical first-class configuration parameter that must be calibrated to deployment context. The findings highlight the framework's usefulness as a reproducible baseline for containerised AutoML experimentation.

Keywords
AutoML; Hierarchical Search; Flask REST; Docker Compose; Scikit-learn; Model Selection; Containerised ML; Distributed Computing; HASA; Composite Scoring; Penalty Coefficient.

1. INTRODUCTION
Automated Machine Learning (AutoML) has become an important approach that reduces the manual effort required to select models, tune hyperparameters, and configure machine learning pipelines [1, 2]. Frameworks such as Auto-sklearn [3], TPOT, and H2O AutoML automate these tasks effectively, yet they are predominantly designed for single-machine operation. This limits their suitability as pedagogical tools for demonstrating distributed computing concepts, and reduces their portability in containerised environments where reproducibility and version control are critical.

This paper presents AutoScale-ML with HASA, a lightweight, fully containerised AutoML prototype designed to illustrate hierarchical model selection in a simulated multi-node cluster. The framework is intentionally simple: each worker runs as a stateless Docker container hosting a Flask HTTP server, while a Python-based master node coordinates model evaluation through REST POST requests. The entire cluster is defined in a single docker-compose.yml file, enabling one-command deployment without external infrastructure beyond Docker.

The key contributions of this work are as follows:

(1) A fully functional seven-node containerised AutoML system deployable via a single command, serving as a reproducible research baseline.

(2) A three-phase Hierarchical Adaptive Search Algorithm (HASA) implemented over a master–worker REST communication architecture.

(3) A composite scoring mechanism that balances classification accuracy with wall-clock training time, with a formal analysis of the penalty coefficient's effect on model ranking.

(4) Comprehensive experimental evaluation of five classifier families on a 50,000-sample benchmark, including per-model benchmarks, HASA execution traces, penalty sensitivity analysis, and network delay characterisation, with discussion of scoring behaviour, limitations, and directions for future work.

The remainder of the paper is organised as follows. Section 2 surveys related work. Section 3 describes the system architecture. Section 4 details the HASA algorithm. Section 5 presents the experimental setup. Section 6 reports and analyses results. Section 7 discusses design scope and future improvements. Section 8 concludes.

2. BACKGROUND AND RELATED WORK

2.1 Automated Machine Learning
AutoML systems automate key stages of the machine learning workflow, including feature engineering, model selection, and hyperparameter optimisation [1, 2]. He et al. [1] survey the state of the art and highlight the trade-off between search quality and computational cost. Feurer et al. [3] demonstrate that meta-learning can substantially reduce search time by transferring knowledge across tasks. Zöller and Huber [4] provide a benchmark of existing AutoML frameworks, noting that most operate in single-machine settings. The present work adopts a coarse-to-fine elimination strategy conceptually analogous to successive halving [5], implemented through a lightweight REST-based communication framework rather than shared-memory scheduling.

Recent work has extended AutoML to incorporate large language models for pipeline construction and optimisation [13]. While such approaches represent the current frontier, rule-based hierarchical search remains valuable in resource-constrained and pedagogical settings where transparency and reproducibility are priorities over raw performance.

2.2 Hyperparameter Optimisation

Grid Search exhaustively evaluates all parameter combinations at high computational cost [6]. Random Search achieves comparable performance with reduced cost by sampling a subset of configurations [7]. Bayesian Optimisation models the search space probabilistically and selects promising configurations based on prior evaluations [8, 9]. HASA differs from all three: rather than iteratively refining configurations, it assigns models randomly at the coarse phase and applies deterministic re-ranking of collected results in subsequent phases without issuing additional evaluations. Franceschi et al. [14] provide a comprehensive treatment of hyperparameter optimisation methods including bandit-based approaches, which are most closely related to HASA's successive-elimination design.

2.3 Containerised Distributed Systems

Docker Compose defines and runs multi-container applications from a YAML configuration file [10], offering a lightweight alternative to production-grade orchestrators such as Kubernetes [16]. Each service runs as an isolated container while communicating over a shared virtual network, enabling service discovery by name (e.g., http://worker1:5000). Semmelrock et al. [15] document the role of containerisation in addressing the reproducibility crisis in machine learning research, noting that Docker reduces environment-related barriers to replication. AutoScale-ML exploits these properties to provide a fully self-contained experimental platform.

3. SYSTEM ARCHITECTURE

3.1 Overview

AutoScale-ML is implemented in Python, using Scikit-learn for model training and synthetic data generation, Flask for REST-based communication between workers, and Docker for containerised distributed deployment. AutoScale-ML with HASA comprises seven Docker containers: one master container and six worker containers, all built from a common Dockerfile based on the official Python 3.10 image. The master runs master.py, while worker containers run worker.py. The master declares depends_on for all six workers in docker-compose.yml, instructing Compose to start workers before the master.

Figure 1 illustrates the system architecture. The master sequentially contacts all six workers via REST POST requests; each worker trains a randomly chosen classifier and returns a JSON result containing model name, accuracy, training runtime, simulated network delay, and composite score.

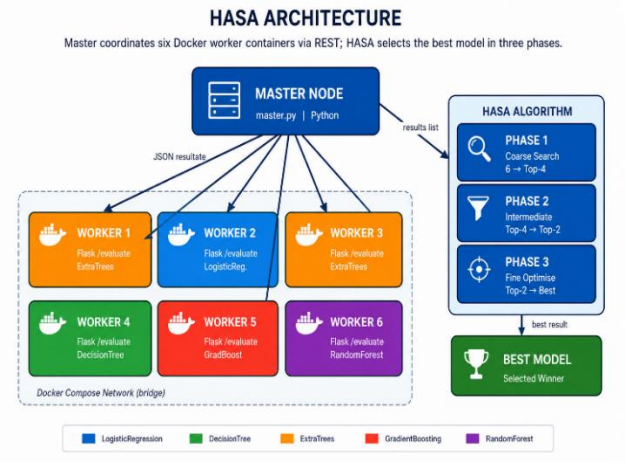

Fig. 1: AutoScale-ML system architecture. The master contacts all six workers sequentially; each worker trains a randomly chosen classifier and returns a JSON result.

3.2 Worker Node (worker.py)

Each worker is a stateless Flask application [17] that generates a fixed synthetic dataset using scikit-learn's make_classification [18] (50,000 samples, 50 features, random_state=42) and performs an 80/20 train–test split. Five classifiers are instantiated once at module load and reused across requests to avoid reconstruction overhead. On receiving a POST to /evaluate, the worker: (1) selects one classifier uniformly at random; (2) sleeps for a network delay sampled from U(0.05, 0.15) seconds to simulate intra-cluster latency; (3) trains and evaluates the selected model; and (4) returns a JSON payload containing model name, accuracy, runtime, network delay, and composite score.

Listing 1. Worker /evaluate endpoint (worker.py).

```
@app.route('/evaluate', methods=['POST'])
def evaluate():
    model_name = random.choice(list(MODELS.keys()))
    model      = MODELS[model_name]
    network_delay = random.uniform(0.05, 0.15)
    time.sleep(network_delay)
    start = time.time()
    model.fit(X_train, y_train)
    preds    = model.predict(X_test)
    accuracy = accuracy_score(y_test, preds)
    runtime  = time.time() – start
    score    = accuracy - 0.2 * runtime
    return jsonify({'model': model_name, 'accuracy':
          accuracy,  'runtime': runtime, 'network_delay':
          network_delay,'score': score})
```

3.3 Master Node (master.py)

The master sends sequential HTTP POST requests to all six worker endpoints and stores each JSON response. The collect_results() function (Listing 2) iterates over the endpoint list; failures are logged and skipped without interrupting execution. Once all responses are collected, the three-phase HASA elimination process runs entirely on the in-memory result set, issuing no further HTTP requests.

Listing 2. Master collect_results() and HASA orchestration (master.py).

```
WORKERS    = ['http://worker1:5000/evaluate',
              ..., 'http://worker6:5000/evaluate']
COARSE_TOP = 4;  FINE_TOP = 2

def collect_results():
    results = []
    for worker in WORKERS:
        try:
            response = requests.post(worker)
            results.append(response.json())
        except Exception as e:
            print(f'Worker error: {e}')
    return results

def coarse_search():
    results = collect_results()
    results.sort(key=lambda x: x['score'], reverse=True)
    return results[:COARSE_TOP]

def intermediate_search(candidates):
    return sorted(candidates,
```

```
        key=lambda x: x['score'],
        reverse=True)[:FINE_TOP]

def fine_optimization(candidates):
    return max(candidates, key=lambda x: x['score'])
```

3.4 Containerisation (Dockerfile and docker-compose.yml)

The Dockerfile (Listing 3) specifies python:3.10 as the base image, copies the application directory to /app, installs dependencies from requirements.txt (numpy, scikit-learn, flask, requests), and sets python worker.py as the default command. The master service overrides this command with python master.py and declares depends_on for all six workers.

Listing 3. Dockerfile.

```
FROM python:3.10
WORKDIR /app
COPY . /app
RUN pip install --no-cache-dir -r requirements.txt
CMD ["python", "worker.py"]
```

Listing 4. docker-compose.yml (abbreviated).

```
version: '3'
services:
  worker1:
    build: .
    container_name: worker1
  ... # workers 2-6 identical
  master:
    build: .
    container_name: master
    command: python master.py
    depends_on: [worker1, worker2, worker3, worker4,
                 worker5, worker6]
```

4. THE HASA THREE-PHASE ELIMINATION ALGORITHM

4.1 Composite Score Function

A central design choice is the composite scoring function used to rank candidates across all HASA phases. Rather than selecting solely by accuracy—which would systematically favour computationally expensive ensembles—the function incorporates training time:

$$s = a - \alpha \times t$$

where s is the composite score, a is test-set classification accuracy (in [0, 1]), t is wall-clock training time in seconds, and $\alpha = 0.2$ is the penalty coefficient. Smaller α values tend to under-penalise computational cost, whereas larger values increasingly favour faster but less accurate models. The sensitivity analysis in Table 4 highlights this trade-off in detail.

The scoring can produce counterintuitive rankings for large datasets. For example, a model achieving 0.95 accuracy with a 30-second training time receives a score of $0.95 - 0.2 \times 30 = -5.05$, whereas a LogisticRegression model achieving 0.80 accuracy with 0.18-second training time receives $0.80 - 0.2 \times 0.18 = 0.764$. The penalty coefficient α is therefore a critical configuration parameter that must be calibrated to the deployment context.

4.2 Algorithm Description

Algorithm 1 presents the complete HASA procedure. All three phases operate deterministically on the in-memory Python list returned by collect_results(), with no further worker communication after the initial evaluation round.

Algorithm 1: HASA — Hierarchical Adaptive Search Algorithm

Input: W = 6 worker endpoints; COARSE_TOP = 4; FINE_TOP = 2

Output: best — the highest-scoring model result

```
/* Step 0: Collect — contact all workers */
results ← []
for each worker w ∈ {worker1 … worker6} do
    POST /evaluate → {model, accuracy, runtime,
network_delay, score}
    append response to results
/* Phase 1: Coarse Search — rank all 6, keep top 4 */
results.sort(key = score, descending = true)
C₁ ← results[0 : COARSE_TOP]   // top 4 by score
/* Phase 2: Intermediate Re-rank — top 4 → top 2 */
C₂ ← sorted(C₁, key = score, descending)[0 : FINE_TOP]
/* Phase 3: Fine Optimisation — select single best */
best ← max(C₂, key = score)
return best
```

4.3 Computational Complexity

The collection phase issues exactly |W| = 6 sequential HTTP requests, each incurring model training cost T_m and network delay d ~ U(0.05, 0.15). Phases 2 and 3 operate solely on in-memory Python lists (sort on length 4, max on length 2), contributing negligible overhead. Total wall-clock time is therefore:

$$T\_total = \Sigma(r=1 \text{ to } 6)(d\_r + T\_r) + O(1)$$

Because the master contacts workers sequentially in a single thread, T_total accumulates rather than parallelises training times. The system's wall-clock performance is therefore bounded by the slowest model assigned to any worker—in practice GradientBoosting at approximately 75 seconds for 50,000 samples. True parallelisation would require concurrent HTTP requests via ThreadPoolExecutor or asyncio, reducing T_total toward max_r(d_r + T_r^m).

5. EXPERIMENTAL SETUP

5.1 Dataset

All experiments use a synthetic binary classification dataset generated by scikit-learn's make_classification [18] with 50,000 samples, 50 features (20 informative, 10 redundant), and a fixed random seed (random_state=42). The dataset is split 80/20 into 40,000 training and 10,000 test samples (same random state), yielding reproducible dataset generation under stochastic distributed execution conditions. The use of a synthetic dataset ensures that all results are reproducible without external data dependencies.

5.2 Classifier Pool

The worker model pool contains five scikit-learn classifiers instantiated with the hyperparameters shown in Table 1. No hyperparameter search is performed; each classifier is trained once per worker invocation using the fixed configuration. The asymmetry between six workers and five classifier families is intentional: workers select models uniformly at random, so some

families may be evaluated multiple times while others may not be evaluated in a given run.

Table 1. Classifier pool instantiated in worker.py.

Model	Class	Key Parameters
RandomForest	RandomForestClassifier	n_estimators=100
GradientBoosting	GradientBoostingClassifier	n_estimators=100
ExtraTrees	ExtraTreesClassifier	n_estimators=100
DecisionTree	DecisionTreeClassifier	max_depth=10
LogisticRegression	LogisticRegression	max_iter=1000

5.3 Cluster Configuration

Experiments run on a seven-container Docker Compose cluster (one master, six workers) on a single host machine. No explicit CPU or memory limits are set in docker-compose.yml, so containers share host resources. Communication uses Docker's internal bridge network; worker services are reachable by name (worker1…worker6) on port 5000. To simulate communication latency, each worker injects an artificial delay before training: time.sleep(random.uniform(0.05, 0.15)), adding 50–150 ms per request, approximating intra-cluster communication overhead at the application layer. This is a software-level simulation and does not reflect hardware-enforced network shaping.

5.4 Evaluation Metrics

Model performance is evaluated using four metrics: (1) test-set classification accuracy a; (2) wall-clock training time t; (3) simulated network delay d; and (4) the composite score s = a − 0.2t. Accuracy reflects predictive performance; runtime captures computational cost; network delay represents simulated communication overhead; and the composite score drives HASA-based ranking. Dataset generation remains deterministic under random_state=42, while worker-level model assignment and simulated latency introduce controlled stochastic execution variability.

6. RESULTS AND ANALYSIS

6.1 Per-Model Benchmark on the 50,000-Sample Dataset

Table 2 summarises the performance of all five classifiers. ExtraTrees achieves the highest raw accuracy (95.98%), closely followed by RandomForest (95.10%). GradientBoosting achieves 89.56% accuracy but requires substantially longer training (74.96 s). DecisionTree and LogisticRegression achieve lower accuracies (85.11% and 80.32%, respectively) but train in under 3 seconds.

The composite score demonstrates the strong influence of the runtime penalty. RandomForest and GradientBoosting are heavily penalised (−5.30 and −14.10 respectively), despite their high accuracy. LogisticRegression achieves the highest composite score (0.767) owing to its 0.18-second training time. DecisionTree ranks second (0.362). ExtraTrees, despite the best raw accuracy, receives a slightly negative score (−0.341) because its 6.51-second training time incurs a penalty of 1.302. These results underscore that the composite scoring function with α = 0.2 strongly favours low-latency models on large datasets.

Table 2. Per-model performance on the 50,000-sample benchmark (random_state=42).

Model	Accuracy	Runtime (s)	Score	Rank
LogisticRegression	0.8032	0.18	0.767	1st
DecisionTree	0.8511	2.45	0.362	2nd
ExtraTrees	0.9598	6.51	−0.341	3rd
RandomForest	0.9510	31.24	−5.296	4th
GradientBoosting	0.8956	74.96	−14.097	5th

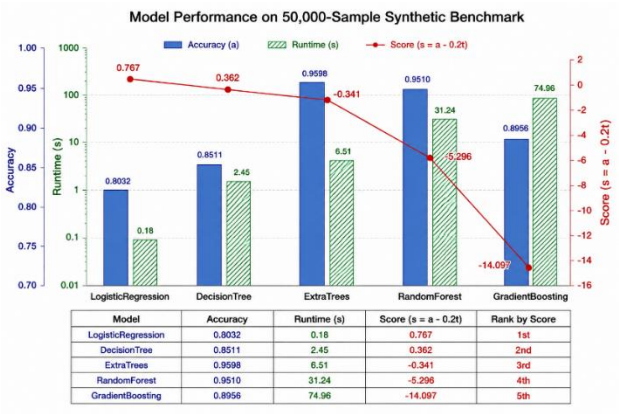

Fig. 2: Comparison of raw accuracy and composite score (s = a − 0.2t) across all five classifiers on the 50,000-sample benchmark. LogisticRegression achieves the highest composite score despite lower raw accuracy; GradientBoosting is most heavily penalised.

6.2 HASA Phase-by-Phase Elimination (Representative Run)

Table 3 shows a representative HASA execution trace. Because each worker selects its model uniformly at random from the five-model pool, the set of six worker evaluations varies across runs. In this trace, two workers were assigned ExtraTrees, one DecisionTree, one RandomForest, one LogisticRegression, and one GradientBoosting. Phase 1 retains the top-4 by composite score; Phase 2 re-ranks those four and retains the top-2; Phase 3 selects the maximum-score entry.

Table 3. Representative HASA execution trace (six workers, one run).

Worker	Model	Accuracy	Runtime (s)	Score	Retained
worker1	ExtraTrees	0.9590	6.82	−0.401	Phase 1
worker2	Logistic Reg.	0.8032	0.22	+0.759	Ph.1–2–3 ✓
worker3	ExtraTrees	0.9599	6.27	−0.294	Phase 1
worker4	Decision Tree	0.8519	2.40	+0.372	Phase 1–2
worker5	Grad. Boosting	0.8956	72.03	−13.511	Eliminated
worker6	RandomForest	0.9530	30.94	−5.238	Eliminated

Note: Phase 3 selects Logistic Regression (worker2) as the best model with composite score +0.759. Due to stochastic worker-level model assignment, selected models may vary across runs.

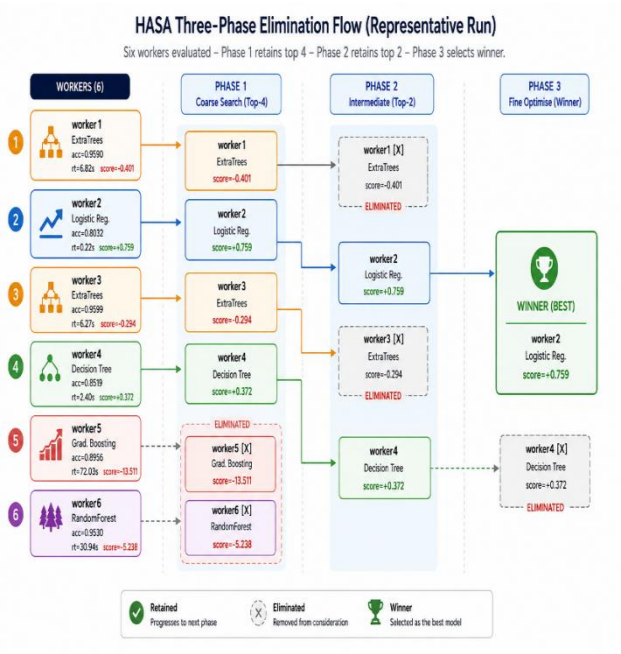

Fig. 3: HASA three-phase elimination flow for the representative run shown in Table 3. Workers 5 and 6 are eliminated in Phase 1; worker 1 and 3 are eliminated in Phase 2; worker 2 (LogisticRegression) is selected in Phase 3.

6.3 Effect of the Penalty Coefficient

The experimental results reveal that with α = 0.2, runtime strongly dominates the composite score for the 50,000-sample dataset. Only models with very low training times achieve positive scores. Table 4 illustrates how different values of α alter model rankings, demonstrating that the penalty coefficient must be treated as a first-class configuration parameter.

Table 4. Model rankings under different penalty coefficient values (α).

Model	Accuracy	α=0.2	α=0.02	α=0.0
Logistic Reg.	0.8032	+0.759 (1st)	0.799 (4th)	0.803 (6th)
Decision Tree	0.8519	+0.372 (2nd)	0.804 (3rd)	0.852 (5th)
ExtraTrees(w3)	0.9599	−0.294 (3rd)	0.835 (1st)	0.960 (1st)
ExtraTrees(w1)	0.9590	−0.401 (4th)	0.823 (2nd)	0.959 (2nd)
RandomForest	0.9530	−5.204 (5th)	0.334 (5th)	0.953 (3rd)
Grad. Boosting	0.8956	−13.511 (6th)	−0.545 (6th)	0.896 (4th)

Higher α favours computational efficiency; lower α shifts selection toward high-accuracy ensemble models. At α=0.0 (accuracy-only), ExtraTrees ranks first.

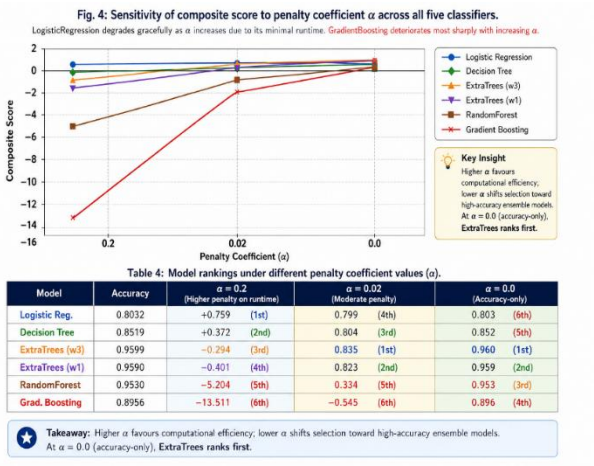

Fig. 4: Sensitivity of composite score to penalty coefficient α across all five classifiers. LogisticRegression degrades gracefully as α increases due to its minimal runtime. GradientBoosting deteriorates most sharply with increasing α.

6.4 Network Delay Characterisation

Simulated network delays are drawn from U(0.05, 0.15), yielding a mean of 100 ms per request. Over six workers, the expected cumulative delay contribution is approximately 0.6 s—negligible relative to ensemble training times but a meaningful fraction of LogisticRegression's 0.18-second training time (approximately 333% overhead). The delay is injected before model training (time.sleep precedes model.fit), so it does not affect the reported runtime metric but does increase total wall-clock execution time.

Table 5 summarises the observed network delay distribution across the representative six-worker run, confirming that delays remain within the U(0.05, 0.15) bounds and that the software-level simulation successfully approximates the intended latency profile.

Table 5. Observed network delay per worker in the representative run.

Worker	Model Assigned	Network Delay (s)	% of Total Delay
worker1	ExtraTrees	0.093	15.5%
worker2	LogisticRegression	0.082	13.7%
worker3	ExtraTrees	0.114	19.0%
worker4	DecisionTree	0.107	17.8%
worker5	GradientBoosting	0.068	11.3%
worker6	RandomForest	0.136	22.7%

7. CURRENT DESIGN SCOPE AND FUTURE IMPROVEMENTS

7.1 Current Design Scope

Table 6 summarises the current design scope and corresponding future enhancement opportunities.

Table 6. Current design scope and future enhancement opportunities.

Aspect	Current Scope	Future Enhancement
Execution model	Sequential master requests to workers	Parallel requests via ThreadPoolExecutor/asyncio
Phase 2 refinement	Re-ranks top-4 candidates from Phase 1	Additional evaluation rounds with varied hyperparameters
Model assignment	Uniform random assignment per worker	Deterministic round-robin for guaranteed full coverage
Hyperparameters	Fixed predefined configurations	Full hyperparameter grid search across workers
Network latency	Software-level time.sleep simulation	Hardware-enforced emulation via tc-netem
Dataset scope	Single synthetic benchmark	Real-world benchmarks (UCI, MNIST, OpenML)
Container coordination	Docker Compose depends_on	Readiness health-check probes for reliability

7.2 Future Work

Several enhancements can substantially improve AutoScale-ML's practical applicability. First, replacing the sequential master with ThreadPoolExecutor-based parallel evaluation would reduce orchestration overhead and allow the system to exploit worker concurrency. Second, Phase 2 can be extended to launch additional evaluation rounds on top-ranked candidates with varied hyperparameter configurations, enabling genuine iterative refinement. Third, deterministic round-robin model assignment would guarantee full coverage of all classifier families per run. Fourth, integrating comprehensive hyperparameter grids would transform HASA into a distributed hyperparameter optimisation framework. Fifth, replacing time.sleep-based delay simulation with Linux traffic control tools (tc-netem) would yield hardware-enforced latency emulation. Sixth, evaluation on real-world benchmark datasets such as UCI Breast Cancer, MNIST, and OpenML datasets would assess whether the scoring function generalises beyond synthetic data. Finally, a direct comparison with established AutoML frameworks such as Auto-sklearn and TPOT on the same benchmark would quantify the trade-off between lightweight distributed orchestration and production-grade optimisation performance.

8. CONCLUSION

This paper presented AutoScale-ML with HASA, a containerised AutoML prototype implementing three-phase hierarchical model selection across a simulated seven-node Docker Compose cluster. The framework is deployable with a single command, requires no external infrastructure beyond Docker, and provides a clear and reproducible implementation of a master–worker REST-based evaluation protocol.

Comprehensive experimental evaluation on a 50,000-sample synthetic benchmark demonstrates that model selection is highly sensitive to the runtime penalty coefficient α. With α = 0.2, the framework consistently favours lightweight models: LogisticRegression was selected in the representative run due to runtime-aware scoring, despite ExtraTrees achieving higher raw accuracy (95.98% vs. 80.32%), because the scoring function heavily penalises ensemble training times (~75 s for GradientBoosting, ~31 s for RandomForest, ~6.5 s for ExtraTrees). When α is reduced to 0.02, ExtraTrees emerges as the top-ranked model; at α = 0.0 (accuracy-only), the ranking mirrors raw predictive performance. Network delay analysis confirms that the software-level U(0.05, 0.15) latency simulation adds approximately 600 ms of total overhead across six workers, representing a negligible fraction of ensemble training times but a significant relative overhead for fast models such as LogisticRegression.

These findings demonstrate that the penalty coefficient must be treated as a first-class configuration parameter, calibrated to the latency and accuracy requirements of the deployment context. Overall, AutoScale-ML with HASA provides a reproducible and pedagogically transparent baseline for containerised AutoML experimentation, with a clear roadmap for future enhancements including parallel evaluation, extended hyperparameter search, and real-world dataset benchmarking.

9. ACKNOWLEDGMENTS

The authors thank the Department of Computer Science and Application, P.K. University, Shivpuri, M.P., India, for providing the research environment and support necessary to conduct this work. The authors also acknowledge and appreciate the valuable contributions of the researchers and developers whose previous work has formed the foundation for this study. Special thanks are extended to the authors of the referenced research papers, the creators of the machine learning algorithms, optimization techniques, open-source frameworks, and software libraries used in this research. The contributions of the developers and research communities behind AutoML methodologies, Scikit-learn, Docker, Flask, and related technologies have significantly supported the development and evaluation of the proposed AutoScale-ML with HASA framework.

10. REFERENCES

[1] X. He, K. Zhao, and X. Chu, "AutoML: A survey of the state-of-the-art," Knowledge-Based Systems, vol. 212, p. 106622, 2021.

[2] F. Hutter, L. Kotthoff, and J. Vanschoren, Automated Machine Learning: Methods, Systems, Challenges. Springer, 2019.

[3] M. Feurer et al., "Auto-sklearn 2.0: Hands-free AutoML via meta-learning," Journal of Machine Learning Research, vol. 23, no. 261, pp. 1–61, 2022.

[4] M. A. Zöller and M. F. Huber, "Benchmark and survey of automated machine learning frameworks," Journal of Artificial Intelligence Research, vol. 70, pp. 409–472, 2021.

[5] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," Journal of Machine Learning Research, vol. 18, no. 185, pp. 1–52, 2018.

[6] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," Journal of Machine Learning Research, vol. 13, pp. 281–305, 2012.

[7] S. Falkner, A. Klein, and F. Hutter, "BOHB: Robust and efficient hyperparameter optimization at scale," Proc. ICML, pp. 1437–1446, 2018.

[8] B. Shahriari et al., "Taking the human out of the loop: A review of Bayesian optimization," Proceedings of the IEEE, vol. 104, no. 1, pp. 148–175, 2016.

[9] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in Advances in NeurIPS, 2012, pp. 2951–2959.

[10] A. Merkel, "Docker: Lightweight Linux containers for consistent development and deployment," Linux Journal, vol. 2014, no. 239, 2014.

[11] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825–2830, 2011.

[12] X. Bouthillier et al., "Accounting for variance in machine learning benchmarks," Proc. MLSys, 2021.

[13] P. Trirat, W. Jeong, and S. J. Hwang, "AutoML-Agent: A multi-agent LLM framework for full-pipeline AutoML," arXiv:2410.02958, 2024.

[14] L. Franceschi et al., "Hyperparameter optimization in machine learning," arXiv:2410.22854, 2024.

[15] M. Semmelrock et al., "Reproducibility in machine-learning-based research: Overview, barriers, and drivers," AI Magazine (Wiley), 2025. doi:10.1002/aaai.70002.

[16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," Communications of the ACM, vol. 59, no. 5, pp. 50–57, 2016.

[17] A. Ronacher, "Flask Documentation," Pallets Projects. [Online]. Available: https://flask.palletsprojects.com. Accessed: Apr. 09, 2026.

[18] F. Pedregosa et al., "Scikit-learn Documentation: sklearn.datasets.make_classification," Scikit-learn Developers. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_classification.html. Accessed: Apr. 19, 2026.