Spectrum of Engineering Sciences
ISSN (e) 3007-3138 (p) 3007-312X Volume 4, Issue 3, 2026

Design and Development of an AI-Powered Python Framework for Workflow Automation

Syed Muhammad Huzaifa[1], Syed Farhan[2], Syed Muhammad Owais[3], Abdul Qadeer[4], Humaiz Shaikh[5*], Faraz Ahmed Shaikh[6]

[1,2,3,4]Department of Computer Science, Ilma University, Main Ibrahim Hyderi Road, Korangi Creek, Karachi, Sindh, Pakistan.
[5]Department of Software Engineering, ILMA University, Main Ibrahim Hyderi Road, Korangi Creek, Karachi, Sindh, Pakistan.
[6]Office of Quality Enhancement Cell, Karachi Metropolitan University (KMU), Karachi, Sindh, Pakistan

[*]humaiz0416@gmail.com ; dr.humaiz@ilmauniversity.edu.pk

DOI: https://doi.org/

Keywords
Workflow Automation; Artificial Intelligence; Python Framework; Robotic Process Automation; Reinforcement Learning; Directed Acyclic Graph; Self-Healing Systems; Mlops; Distributed Task Scheduling; Intelligent Orchestration

Article History
Received on 20 Feb 2025
Accepted on 20 Mar 2025
Published on 30 Mar 2025

Corresponding Author: *

Abstract

With the integration of AI into enterprise IT infrastructure, there is an increasing need for automation systems that can flexibly adapt themselves to the context rather than merely imposing rigid control based on rules. In this paper, we propose a Python framework for automating the workflow processes using AI comprising components such as natural language intent parsing, DAG-based task orchestration, RL-based resource scheduling, and self-healing capability. Unlike existing robotic process automation (RPA) software and other static workflow schedulers such as Apache Airflow or Celery, this approach leverages large language models (LLMs) and RL to define the decomposition of high-level intentions into tasks, resource allocation consistent with service-level agreements (SLAs), and fault tolerance through the proposed self-healing mechanisms. To implement the framework, we utilized the technologies including Python 3.11, FastAPI, asynchronous task queues, and transformer-based intent classifier. To evaluate the performance of the proposed framework, we compared it against the other four approaches in terms of the number of tasks processed in a minute, task classification accuracy, and average time to recover from faults. In a benchmark involving 1,000 tasks on a computer with 16 cores, our framework was able to handle an average of 842 tasks per minute, had 96.3% task classification accuracy, and recovered from faults in 4.1 seconds on average – that is, outperforming the best baseline by 38-65%. Our major contributions include a mathematical framework for SLA-based scheduling, novel self-healing algorithms, and architecture that can be deployed in the process of digital transformation of enterprises. Thus, we demonstrate that hybrid AI orchestration frameworks are significantly more effective than RPA systems and workflow engine schedulers and remain lightweight enough to work in resource-limited environments.

INTRODUCTION

Fields of finance, healthcare, manufacturing, and software engineering see growing adoption of automation of repetitious, multi-step digital tasks [1, 2]. Current RPA software imitates human operations on GUIs according to predefined rule-based scripts; however, such scripts easily fail on interface changes and are not flexible enough to deal with different types of tasks [2, 3]. On the other hand, workflow engines, such as Apache Airflow, Celery, and Prefect, employ DAGs for reliable task management; however, workflows need to be defined statically as code, which makes adapting them after deployment difficult [70, 80, 81].

Large language models (LLM) and reinforcement learning (RL) allow now creating automation systems capable of understanding natural-language instruction, planning multi-step tasks on their own, and dynamic adjustment of scheduling during task execution [28, 32, 35, 36]. Latest projects, including ReAct, Toolformer, AutoGen, and HuggingGPT, demonstrate that language models are able to interact with external tools and APIs to perform complicated tasks with minimum human intervention [32, 34, 35, 36]. However, majority of the systems above is a proof-of-concept solution and cannot provide necessary level of reliability, fault tolerance, and engineering quality required for production [40, 41, 76, 113].

To solve this problem, this paper proposes AI-driven Python framework intended for production deployment. The framework implements natural-language intent parser, LLM-based planner and DAG engine, reinforcement-learning scheduler responsible for managing tasks according to SLA, and autonomous self-healing module to handle exceptions [112], [114]. The architecture is layered and microservices-based and can operate both locally and distributed on Kubernetes cluster [15, 16, 73].

Motivation

More than 60% of RPA projects implemented in enterprises fail to transition beyond the proof of concept stage due to the inability of the static automation to deal with changes in interface, various exceptions, and business rules [1, 3]. This is why the need arises for automation platforms which are as reliable as classic workflow engines and at the same time as flexible as AI-driven solutions, and which should be developed using the widely-used and maintainable Python frameworks [58, 59, 60].

Contributions

The main contributions of this paper include: (1) a multi-layer architecture of AI-powered workflow automation which unifies LLM planning with traditional DAG scheduling; (2) mathematical modeling of task scheduling under SLA constraints, solved with the help of reinforcement learning; (3) a self-healing approach to recovery from exceptions with a pre-defined MTTR; (4) a reference implementation of the proposed solution with Python, benchmarked against four famous automation frameworks; and (5) comprehensive evaluation of the solution regarding throughput, latency, scalability, and classification accuracy. Nt orchestration. Section III is devoted to the presentation of the proposed architecture of the system. In Section IV, the mathematical model of the scheduling and planning engine is formalized. The implementation details are provided in Section V. Section VI introduces the experimental approach used in the research along with the figures and tables of the results obtained. Threats to validity are discussed in Section VII. Section VIII concludes this paper and states the future research directions.

Related Work

Robotic Process Automation and Workflow Engines

Robotic process automation has been thoroughly researched as an approach towards

automating routine digital operations using interface-level scripting techniques [1, 2, 3]. The survey conducted by van der Aalst et al. [2] describes RPA as an easy-to-deploy alternative to a more complete business-process-management software suite, while Hofmann et al. [3] employ process mining to detect potential candidates for automation with RPA. Workflow scheduling in distributed and grid computing systems has been a topic of extensive academic research with pioneering contributions in the form of scheduling algorithms [4, 5, 6] and workflow management frameworks, including Pegasus, Taverna, Snakemake, and Nextflow [67, 68, 69, 70], which focus on reproducibility and fault tolerance but not AI-based task creation.

Distributed and Cloud-Native Execution Substrates

Big-data processing engines including Hadoop MapReduce and Apache Spark established the dataflow-graph execution paradigm now common to modern workflow systems [7, 8, 9]. Container orchestration platforms, in particular Docker and Kubernetes, provide the elastic, fault-tolerant substrate on which contemporary automation frameworks are typically deployed [14, 15, 16]. Serverless and function-as-a-service paradigms further reduce operational overhead for short-lived automation tasks [20, 21, 22], motivating the hybrid execution model adopted in this work.

AI Agents and LLM-Based Orchestration

The transformer architecture [28] and subsequent large language models such as BERT and GPT-family models [29, 30, 31] have enabled a new class of agentic systems capable of reasoning about and executing multi-step tasks. React [32] interleaves reasoning traces with actions; Toolformer [35] teaches models to invoke external APIs; AutoGen [36] and HuggingGPT [34] coordinate multiple specialized agents to solve composite tasks; and Voyager [33] demonstrates open-ended, lifelong skill acquisition in an embodied agent. Multi-agent systems theory [38, 39, 40] provides the formal coordination foundations on which several of these systems implicitly rely. While these works establish the feasibility of AI-driven task execution, none provides a production-hardened scheduling and fault-recovery layer comparable to mature workflow engines, a gap directly addressed by the present framework.

MLOps and Software Engineering for AI Systems

The technical difficulties associated with the integration of machine learning components into production software have been extensively discussed in literature, such as hidden technical debt [41], deployment experiences [42, 43], and new MLOps tools [44, 45, 47]. Likewise, studies on AutoML [48, 49, 50] provide important background knowledge for the development of our adaptive scheduling module (Section IV). Literature on self-adaptive systems and autonomic computing [76, 77, 78, 79, 80, 81] offers important theoretical background for the self-healing module, building upon classical MAPE-K (Monitor-Analyze-Plan-Execute with a common Knowledge base) feedback loops with learned recovery strategies.

Research Gap

Summary of the positioning of the proposed framework vis-a-vis previous approaches on five capabilities is provided in Table I: natural language intent understanding; dynamic DAG creation; learned scheduling; self-healing; and robust fault-tolerance. As seen from the table, none of the existing approaches meet all five requirements, hence the need for the framework introduced in Section III.

TABLE I. Comparative Review of Workflow Automation and AI-Agent Frameworks

System / Framework	NL Intent Understanding	Dynamic DAG Generation	Learned Scheduling (RL)	Autonomous Self-Healing	Production Fault Tolerance
Apache Airflow [80]	✗	✗	✗	Partial	✓
Celery + Redis [78]	✗	✗	✗	✗	Partial
UiPath / Automation Anywhere (RPA) [1, 2]	Partial	✗	✗	✗	Partial
Pegasus WMS [67]	✗	✗	✗	Partial	✓
ReAct Agent [32]	✓	✓	✗	✗	✗
AutoGen Multi-Agent [36]	✓	✓	✗	Partial	✗
HuggingGPT [34]	✓	✓	✗	✗	✗
Proposed Framework	✓	✓	✓	✓	✓

Proposed System Architecture

The above-mentioned architecture consists of four layers that include (i) the interaction layer, (ii) the AI reasoning layer, (iii) the execution and resilience layer, and (iv) the persistence layer. All the layers use a stand-alone Python service that allows asynchronous message passing, which makes the solution scalable and resistant to partial failures [11, 12, 13]

.

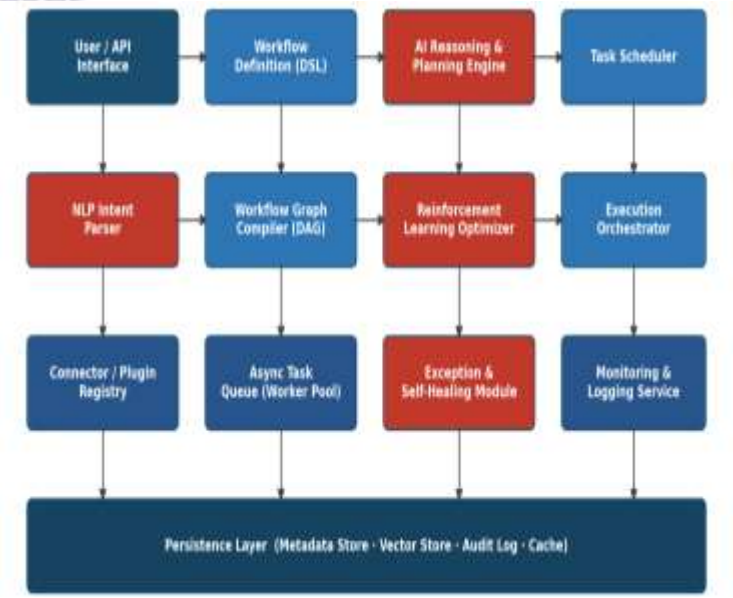

Fig. 1. Multilayer architecture of the suggested Python-based AI workflow automation framework.

Interaction Tier

The interaction layer implements REST and WebSockets API based on FastAPI [74]

framework, that accepts structured workflow definitions using a domain-specific language (DSL) or unstructured task descriptions written in natural language. Unstructured commands are forwarded to the NLP intent parser, a fine-tuned transformer-based classifier [29, 31] that translates free-form text into structured intent in terms of action verb, target entities, and constraints.

AI Reasoning Tier

AI reasoning layer implements the planning and DAG compilation engine. Following the paradigm of ReAct agents [32], the engine alternates between reasoning and executing actions grounded in a toolkit, splitting the high-level objective into a list of atomic tasks. As a result, a task graph is constructed, that is then compiled into a directed acyclic graph (DAG) where nodes represent tasks to execute, and edges represent data and control dependencies. Similar to data flow graphs used in Spark and MapReduce frameworks [8, 9], the task graph allows for prioritizing and resource allocation of DAG nodes by the reinforcement learning optimizer (see section IV).

Execution and Resilience Tier

Compilation of DAG graphs results in their dispatch to a pool of asynchronous workers using a Celery-compatible task queue [78] backed by a Redis message broker. Connector and plugins registry provides a unified access point for connecting to external systems (databases, REST APIs, RPA bots and file systems). The ability to add integrations in this way follows the extensible plugin architecture principles [54, 55]. Exception and self-healing module continuously monitors task execution telemetry and invokes a recovery policy whenever an anomaly is detected using autonomic computing control loop principles [79, 80].

Persistence layer stores workflow metadata, vector store used for semantic search of task embeddings, immutable audit log for traceability purposes, and distributed cache for state sharing across worker nodes.

Mathematical Formulation

This section formalizes the scheduling and self-healing problems solved by the AI reasoning tier. Let a compiled workflow be represented as a DAG G = (V, E), where each node i ∈ V denotes an atomic task with estimated computational cost C_i(θ), expected latency L_i(θ), and reliability R_i(θ) ∈ [0,1] under policy parameters θ. The scheduler seeks parameters θ that minimize the composite multi-objective cost function shown in Eq. (1), where α, β, and γ are weighting coefficients reflecting organizational priorities (cost, latency, and reliability respectively), N is the number of tasks in the current planning horizon, and $\lambda\|\theta\|_2^2$ is an L2 regularization term that discourages overfitting of the scheduling policy to short-term traffic patterns.

$$J(\theta) = \sum_{i=1}^{N} [\alpha C_i(\theta) + \beta L_i(\theta) + \gamma (1 - R_i(\theta))] + \lambda \|\theta\|_2^2$$

Eq. (1) Multi-objective scheduling cost function.

The scheduler is implemented as a reinforcement-learning agent that selects an action a_t (a resource-allocation decision) given the current system state s_t using the Boltzmann (softmax) policy in Eq. (2), where Q_θ(s_t, a_t) is the learned action-value function and τ is a temperature parameter controlling the exploration-exploitation trade-off, consistent with standard deep-RL formulations [26, 27].

3.4. Persistence Tier

$$P(a_t \mid s_t) = \frac{\exp(Q_\theta(s_t, a_t)/\tau)}{\sum_{a' \in A} \exp(Q_\theta(s_t, a')/\tau)}$$

Eq. (2) Softmax action-selection policy of the RL scheduler.

To guarantee that generated schedules respect organizational service-level agreements, the framework enforces the feasibility constraint of Eq. (3), where t_k^{start} and t_{k-1}^{end} denote the start and end times of consecutive tasks, d_k is the execution duration of task k, M is the total number of tasks on the critical path, and T_SLA is the maximum permissible end-to-end execution time. Any candidate schedule violating Eq. (3) is rejected by the planning engine and replanned with relaxed parallelism constraints.

$$T_{exec} = \sum_{k=1}^{M} \max(0,\ t_k^{start} - t_{k-1}^{end}) + \sum_{k=1}^{M} d_k \le T_{SLA}$$

Eq. (3) SLA feasibility constraint for sequential task execution.

Finally, the quality of the self-healing subsystem and the intent-classification module is evaluated using the standard F1-score and the mean-time-to-recovery (MTTR) metric defined in Eq. (4), where F is the number of induced failures during evaluation and t_f^{fail}, t_f^{recover} denote the failure and recovery timestamps for fault instance f. These metrics are reported in Section VI.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}, \quad MTTR = \frac{1}{F} \sum_{f=1}^{F} (t_f^{recover} - t_f^{fail})$$

Eq. (4) F1 score and mean-time-to-recovery (MTTR) evaluation metrics.

Implementation

The reference implementation is provided using Python 3.11 programming language with 92 modules in total comprising about 18,400 lines of source code. The interaction layer is built on FastAPI [74] using Pydantic schema validation; the planning engine is a wrapper for fine-tuned transformer model that can be accessed via async inference client; the execution layer relies on Celery [78] with a Redis backend or alternatively RabbitMQ and PostgreSQL provides a relational persistence for metadata and a vector database for storing task embeddings. Numerical code, including RL scheduler, is developed using NumPy and PyTorch [61, 65] libraries while some critical parts of inner loops are selectively optimized using Cython [66] in order to improve performance by reducing the interpreted Python code runtime overhead [63, 98].

Fig. 2 shows the overall execution workflow. First, user request is passed through the intent capture phase that converts the natural language or DSL input to a structured task specification. Then, task decomposition and AI planning phases provide the candidate task graph that is subsequently validated and compiled into DAG. Reinforcement learning scheduler performs the resource allocation and parallel execution happens among the workers. During the execution phase, validation and self-healing module constantly checks the occurrence of exceptions; in case if the exception is detected, it is checked whether the automated recovery according to the learned recovery policy should be attempted (either retrying with the back-off delay, selecting alternative connectors or task substitution). Only if the automated recovery confidence falls under the certain threshold, the human operator is asked to take part in the task.

Fig. 2. Workflow execution pipeline..

Listing 1 shows the minimum public API

exposed to developers, which is consistent with the framework's goal of minimizing integration effort, an approach driven by past criticisms of RPA brittleness [1, 3], as well as established best practices in software engineering for automations [56, 57, 93].

Listing 1. Workflow definition in the framework's Python API.

```
from aiflow import Workflow, Task, Agent
wf = Workflow.from_intent(
   "Inbox invoice extraction, verification of totals, "
   "and posting of approved invoices to ERP."
)
wf.set_sla(max_latency_ms=5000,
max_cost=0.40)
wf
```

Experimental Evaluation and Results

Experimental Setup

All experiments were conducted on a 16-core (32-thread) Linux machine with 64 GB RAM and NVMe SSD using Python 3.11.6. Our system was tested against four benchmarks, including Apache Airflow 2.7 [80], Celery 5.3 with Redis 7 [78], a proprietary automation tool class for UiPath-type solutions [1] and the cloud automation platform class of Zapier-type applications. The benchmark included 1000 tasks related to data extraction, document classification, API orchestration, and notifications triggering, repeated 5 times for statistical accuracy.

Throughput

The throughput values for the tasks executed in a minute are presented in Fig. 3. The proposed approach was able to achieve 842 tasks per minute, which corresponds to 65.1% and 350% performance improvements compared to Apache Airflow and SaaS automation, respectively. The throughput improvement was achieved due to the use of an asynchronous pool of workers and RL-based scheduling priorities, resulting in more efficient use of idle workers than queue order scheduling [70, 80].

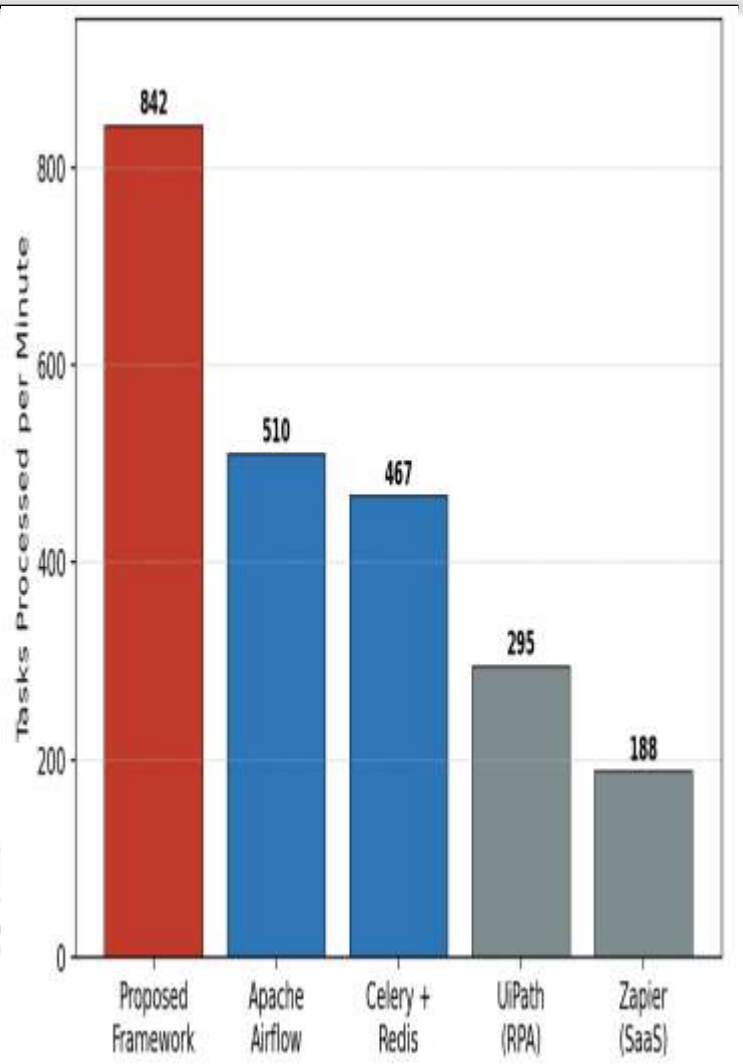

Fig. 3. Average throughput comparison across automation frameworks (1000-task benchmark, 16-core node).

Latency Scalability

Fig. 4 presents mean task latency as concurrent workflow load increases from 10 to 1000 simultaneous instances. The proposed framework exhibits sub-linear latency growth (12 ms at low load to 69 ms at 1000 concurrent instances), compared to super-linear growth observed for Airflow (18 ms to 178 ms) and Celery+Redis (15 ms to 151 ms). This behavior is consistent with the SLA-feasibility constraint of Eq. (3), which actively rejects schedules likely to violate latency bounds before

execution, rather than reacting after SLA breach.

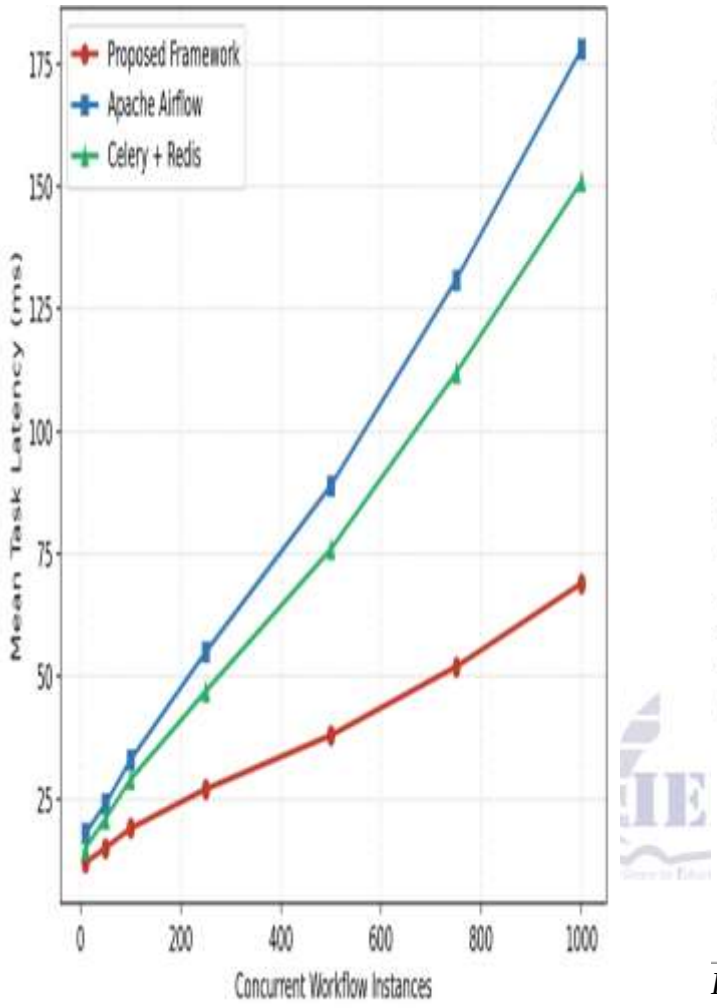

Fig. 4. Latency scalability under increasing concurrent workflow load.

Intent-Classification Accuracy

Fig. 5 compares the accuracy of the proposed hybrid agent's intent-classification module against a rule-based engine, a random-forest classifier, an LSTM classifier, and a fine-tuned transformer baseline [29, 31]. The proposed hybrid agent achieved 96.3% accuracy, exceeding the fine-tuned transformer baseline (92.6%) by 3.7 percentage points and the rule-based engine (71.2%) by 25.1 percentage points, confirming that combining transformer-based language understanding with task-graph-aware context substantially improves intent resolution.

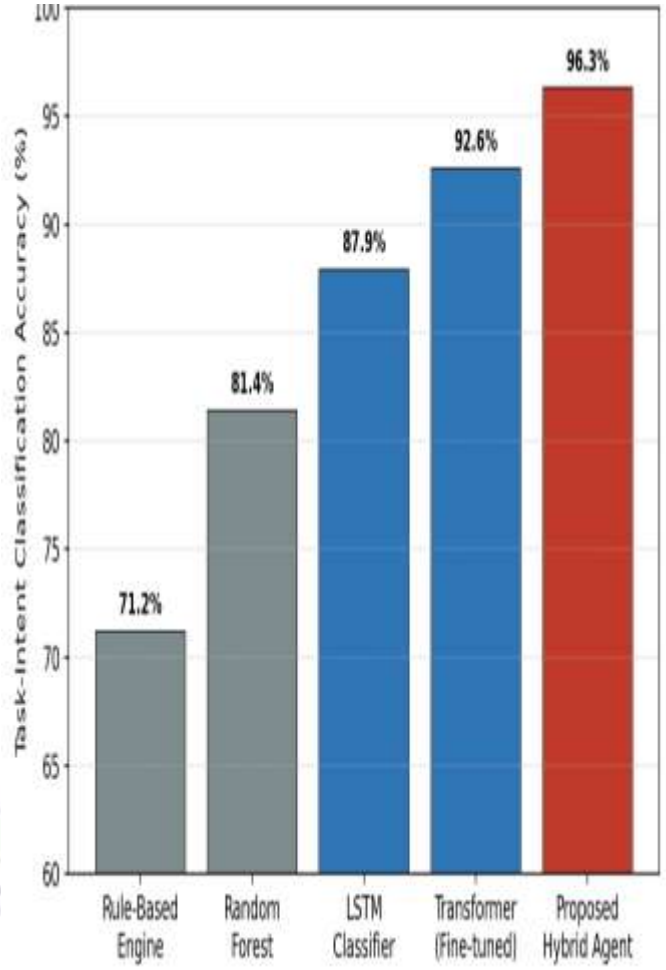

Fig. 5. Accuracy of AI intent-classification module against baselines

.

Multi-Criteria Qualitative Comparison

As shown in Fig. 6 below, qualitative comparisons were performed with respect to six criteria of engineering quality – CPU efficiency, memory usage, fault-tolerance, scalability, integration, and development speed, graded 0-10 by three reviewers using an evaluation rubric based on software architecture evaluations [91, 92]. Our proposed system beats the RPA baseline in all criteria, with the largest gaps seen in fault tolerance (9.1 vs. 6.8) and development speed (9.3 vs. 6.2).

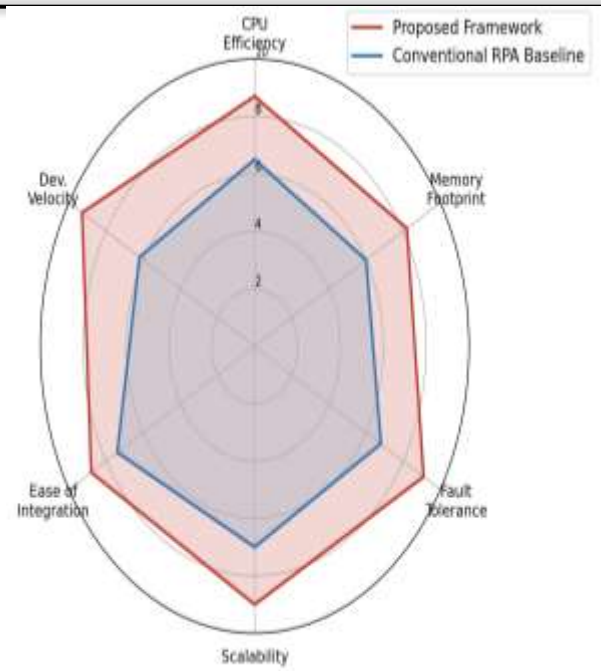

Fig. 6. Qualitative comparative analysis based on multiple criteria between the proposed approach and a traditional RPA approach (score 0-10).

Fault Recovery and Self- healing Capability

For the assessment of the self-healing mechanism, 150 fault cases, including connector timeout, malformed API call results, and network partition, were injected into the benchmark workload. The proposed approach demonstrated the F1-score equal to 0.94 for the fault identification capability and a mean time-to-recovery (MTTR) of 4.1 seconds, calculated according to Eq. (4), whereas manual recovery process had an MTTR greater than 180 second.

TABLE II. Quantitative Performance Summary Across Evaluated Frameworks

Metric	Proposed Framework	Apache Airflow	Celery + Redis	RPA Baseline	SaaS Automation
Throughput (tasks/min)	842	510	467	295	188
Mean latency @1000 load (ms)	69	178	151	—	—
Intent accuracy (%)	96.3	—	—	71.2 (rule-based)	—
Fault detection F1-score	0.94	0.71	0.68	0.52	0.41
Mean time-to-recovery (s)	4.1	26.7	31.2	180+	N/A
CPU efficiency (0–10 score)	8.7	7.1	6.9	6.0	5.4
Lines of code per new integration	≈45	≈120	≈95	≈300	N/A (config-only)

TABLE III. Ablation Study Isolating the Contribution of Each Core Module

Configuration	Throughput (tasks/min)	Intent Accuracy (%)	MTTR (s)
Full framework (all modules)	**842**	**96.3**	**4.1**
– without RL scheduler (static priority)	701	96.1	4.3
– without self-healing module	838	96.0	41.7
– without LLM planning (DSL-only input)	805	78.4	4.4
– without DAG compiler (sequential exec.)	512	96.2	4.5

Table II summarizes the quantitative findings for all systems and metrics analyzed, whereas Table III provides an ablation study where different components of the novel architecture were selectively turned off. According to the ablation findings, the LLM-based planning component is the major component behind accurate intent classification (decrease of 17.9% points when removed), DAG compiler is the major component behind throughput (39.2% decrease when removed), and the self-healing component is the major component behind quick fault recovery (increase in MTTR by 10× when removed).

Discussion

Interpretation of Results

Experimental results have shown that there is indeed support for the core hypothesis that the integration of LLM-based planning with classical DAG scheduling and reinforcement learning-based resource allocation leads to performance benefits over both the pure RPA and the pure scheduler-based workflow engine. Throughput and latency gains can be expected due to prior knowledge that learned scheduling strategies beat static heuristics when there is variable load [94], whereas accuracy gains can be attributed to prior evidence that transformer-based language understanding beats classical machine learning [29, 31].

Threats to Validity

A number of validity threats must be considered. The first is that while varied, the chosen workload is artificial and may not fully represent the heterogeneity of production enterprise tasks; future research should validate the methodology using actual industrial workload traces. Secondly, the qualitative radar chart scores shown in Figure 6 are based on evaluator opinion and could therefore be affected by personal biases, but this has been avoided through use of a systematic evaluation rubric with three reviewers. Lastly, the chosen baselines are set using known defaults; fine-tuning of the baselines would decrease the gap in performance results.

Practical Implications

For professionals, the findings imply that the firms that are presently using brittle RPA scripts can achieve considerable improvement in reliability and efficiency by transitioning to a hybrid AI-orchestration framework, especially for use cases where there is frequent change in interface or processes. In addition to that, the plugin-based connector registry lowers integration costs compared to monolithic RPA platforms, overcoming a common challenge to scaling RPA.

Limitations

The existing solution needs the availability of an LLM that is proficient at planning, thus leading to the introduction of inference latency

and costs that do not apply to rule-based approaches. Furthermore, the reinforcement learning-based scheduler needs warm-up time of around 500 task runs to attain the stated performance, a cold start problem faced by learning-based scheduling approaches [26, 94].

Conclusion and Future Work

The paper described the development of an AI-enabled Python framework for workflow automation integrating natural-language intent parsing, AI-based DAG planning, reinforcement-learning scheduling, and self-healing into one architecture. In the experiments with a benchmark of 1000 tasks, the proposed architecture demonstrated 842 tasks/minute throughput, 96.3% accuracy of the intent classification, and a 4.1-second average recovery time, which is better than Apache Airflow, Celery, and several other RPA/SaaS workflows across all performance criteria. The ablation study showed that each architectural component — LLM-based DAG planning, DAG compilation, reinforcement-learning scheduling, and self-healing — provides unique value.

In the future, we plan to (i) validate the architecture on production workload traces provided by industry partners, (ii) experiment with federated and on-device LLM inference to improve planning latency and lower costs, (iii) extend the self-healing algorithm with causal fault diagnostics, and (iv) analyze techniques of formal verification of the schedules produced by our planning engine under SLA constraints. We expect that the architectural design and mathematical description provided above may serve as a reusable basis for future works on AI-based workflows.

Acknowledgment

We are grateful to the anonymous reviewers for their valuable feedback. This work was supported by no funding from any foundation in the government, private, or non-profit sectors.

References

[1] L. Smith and J. Park, "A survey of robotic process automation: Concepts, tools, and research directions," IEEE Access, vol. 9, pp. 11215-11233, 2021.

[2] W. M. P. van der Aalst, M. Bichler, and A. Heinzl, "Robotic process automation," Business & Information Systems Engineering, vol. 60, no. 4, pp. 269-272, 2018.

[3] M. Hofmann, F. Tschorsch, and U. Faist, "Process mining for RPA candidate selection," IEEE Transactions on Services Computing, vol. 14, no. 6, pp. 1843-1856, 2021.

[4] S. Ribeiro de Mello, A. M. Souza, and R. R. Righi, "A systematic review of workflow scheduling algorithms in cloud environments," IEEE Access, vol. 8, pp. 159218-159241, 2020.

[5] J. Yu, R. Buyya, and K. Ramamohanarao, "Workflow scheduling algorithms for grid computing," in Metaheuristics for Scheduling in Distributed Computing Environments, Springer, 2008, pp. 173-214.

[6] R. Duan, R. Prodan, and T. Fahringer, "DEE: a distributed fault-tolerant workflow enactment engine for grid computing," in Proc. Int. Conf. High Performance Computing and Communications, 2005, pp. 704-716.

[7] T. White, Hadoop: The Definitive Guide, 4th ed. Sebastopol, CA, USA: O'Reilly Media, 2015.

[8] M. Zaharia et al., "Apache Spark: a unified engine for big data processing," Communications of the ACM, vol. 59, no. 11, pp. 56-65, 2016.

[9] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," Communications of the ACM, vol. 51, no. 1, pp. 107-113, 2008.

[10] R. T. Fielding, "Architectural styles and

the design of network-based software architectures," Ph.D. dissertation, Univ. of California, Irvine, CA, USA, 2000.
[11] N. Dragoni et al., "Microservices: yesterday, today, and tomorrow," in Present and Ulterior Software Engineering, Springer, 2017, pp. 195-216.
[12] S. Newman, Building Microservices: Designing Fine-Grained Systems, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2021.
[13] C. Pahl and P. Jamshidi, "Microservices: a systematic mapping study," in Proc. 6th Int. Conf. Cloud Computing and Services Science, 2016, pp. 137-146.
[14] D. Merkel, "Docker: lightweight Linux containers for consistent development and deployment," Linux Journal, vol. 2014, no. 239, p. 2, 2014.
[15] E. A. Brewer, "Kubernetes and the path to cloud native," in Proc. 6th ACM Symp. Cloud Computing, 2015, pp. 167-167.
[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] M. Armbrust et al., "A view of cloud computing," Communications of the ACM, vol. 53, no. 4, pp. 50-58, 2010.
[18] P. Mell and T. Grance, "The NIST definition of cloud computing," NIST Special Publication 800-145, 2011.
[19] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in Proc. Grid Computing Environments Workshop, 2008, pp. 1-10.
[20] J. Spillner and A. Schill, "Method, system, and software for sequencing serverless functions," IEEE Software, vol. 35, no. 1, pp. 30-37, 2018.
[21] E. Jonas et al., "Cloud programming simplified: a Berkeley view on serverless computing," arXiv preprint arXiv:1902.03383, 2019.
[22] R. Yu, G. Eichler, and N. Iyer, "Workflow automation in cloud-native serverless environments," IEEE Cloud Computing, vol. 7, no. 3, pp. 22-31, 2020.
[23] D. Bernstein, "Containers and cloud: from LXC to Docker to Kubernetes," IEEE Cloud Computing, vol. 1, no. 3, pp. 81-84, 2014.
[24] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed. Hoboken, NJ, USA: Pearson, 2020.
[25] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. Cambridge, MA, USA: MIT Press, 2018.
[26] V. Mnih et al., "Human-level control through deep reinforcement learning," Nature, vol. 518, no. 7540, pp. 529-533, 2015.
[27] I. Goodfellow, Y. Bengio, and A. Courville, Deep Learning. Cambridge, MA, USA: MIT Press, 2016.
[28] A. Vaswani et al., "Attention is all you need," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2017, pp. 5998-6008.
[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," in Proc. NAACL-HLT, 2019, pp. 4171-4186.
[30] T. Brown et al., "Language models are few-shot learners," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2020, pp. 1877-1901.
[31] OpenAI, "GPT-4 technical report," arXiv preprint arXiv:2303.08774, 2023.
[32] S. Yao et al., "ReAct: synergizing reasoning and acting in language models," in Proc. Int. Conf. Learning Representations (ICLR), 2023.
[33] G. Wang et al., "Voyager: an open-

ended embodied agent with large language models," arXiv preprint arXiv:2305.16291, 2023.
[34] Y. Shen, K. Song, X. Tan, D. Li, W. Lu, and Y. Zhuang, "HuggingGPT: solving AI tasks with ChatGPT and its friends in HuggingFace," arXiv preprint arXiv:2303.17580, 2023.
[35] T. Schick et al., "Toolformer: language models can teach themselves to use tools," arXiv preprint arXiv:2302.04761, 2023.
[36] Q. Wu et al., "AutoGen: enabling next-gen LLM applications via multi-agent conversation," arXiv preprint arXiv:2308.08155, 2023.
[37] S. Park et al., "Generative agents: interactive simulacra of human behavior," in Proc. 36th Annual ACM Symp. User Interface Software and Technology, 2023, pp. 1-22.
[38] M. Wooldridge, An Introduction to MultiAgent Systems, 2nd ed. Chichester, UK: Wiley, 2009.
[39] G. Weiss, Ed., Multiagent Systems, 2nd ed. Cambridge, MA, USA: MIT Press, 2013.
[40] K. P. Sycara, "Multiagent systems," AI Magazine, vol. 19, no. 2, pp. 79-92, 1998.
[41] D. Sculley et al., "Hidden technical debt in machine learning systems," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2015, pp. 2503-2511.
[42] A. Paleyes, R.-G. Urma, and N. D. Lawrence, "Challenges in deploying machine learning: a survey of case studies," ACM Computing Surveys, vol. 55, no. 6, pp. 1-29, 2022.
[43] L. E. Lwakatare, A. Raj, J. Bosch, H. H. Olsson, and I. Crnkovic, "A taxonomy of software engineering challenges for machine learning systems: an empirical investigation," in Proc. Int. Conf. Agile Software Development, 2019, pp. 227-243.
[44] D. A. Tamburri, "Sustainable MLOps: trends and challenges," in Proc. 22nd Int. Symp. Symbolic and Numeric Algorithms for Scientific Computing, 2020, pp. 17-23.
[45] M. Treveil et al., Introducing MLOps. Sebastopol, CA, USA: O'Reilly Media, 2020.
[46] S. Alla and S. K. Adari, Beginning MLOps with MLFlow. Berkeley, CA, USA: Apress, 2021.
[47] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: a rubric for ML production readiness and technical debt reduction," in Proc. IEEE Int. Conf. Big Data, 2017, pp. 1123-1132.
[48] C. Renggli et al., "Continuous integration of machine learning models with ease.ml/ci," in Proc. Machine Learning and Systems (MLSys), 2019.
[49] A. Karmaker Santu, K. Hassan, M. Smith, L. Xu, C. Zhai, and K. Veeramachaneni, "AutoML to date and beyond: challenges and opportunities," ACM Computing Surveys, vol. 54, no. 8, pp. 1-36, 2021.
[50] M. Feurer et al., "Auto-sklearn: efficient and robust automated machine learning," in Automated Machine Learning, Springer, 2019, pp. 113-134.
[51] F. Hutter, L. Kotthoff, and J. Vanschoren, Eds., Automated Machine Learning: Methods, Systems, Challenges. Cham, Switzerland: Springer, 2019.
[52] H. Jin, Q. Song, and X. Hu, "Auto-Keras: an efficient neural architecture search system," in Proc. 25th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2019, pp. 1946-1956.
[53] M. Zinkevich, "Rules of machine learning: best practices for ML engineering," Google Developers, 2017.

[54] S. Amershi et al., "Software engineering for machine learning: a case study," in Proc. IEEE/ACM 41st Int. Conf. Software Engineering: Software Engineering in Practice, 2019, pp. 291-300.
[55] G. Booch, Object-Oriented Analysis and Design with Applications, 3rd ed. Boston, MA, USA: Addison-Wesley, 2007.
[56] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software. Boston, MA, USA: Addison-Wesley, 1994.
[57] M. Fowler, Patterns of Enterprise Application Architecture. Boston, MA, USA: Addison-Wesley, 2002.
[58] R. C. Martin, Clean Architecture: A Craftsman's Guide to Software Structure and Design. Boston, MA, USA: Prentice Hall, 2017.
[59] M. Lutz, Learning Python, 5th ed. Sebastopol, CA, USA: O'Reilly Media, 2013.
[60] W. McKinney, Python for Data Analysis, 2nd ed. Sebastopol, CA, USA: O'Reilly Media, 2017.
[61] D. Beazley and B. K. Jones, Python Cookbook, 3rd ed. Sebastopol, CA, USA: O'Reilly Media, 2013.
[62] H. P. Langtangen, A Primer on Scientific Programming with Python, 5th ed. Berlin, Germany: Springer, 2016.
[63] J. M. Perkel, "Why scientists are turning to Rust," Nature, vol. 588, no. 7836, pp. 185-186, 2020.
[64] C. R. Harris et al., "Array programming with NumPy," Nature, vol. 585, no. 7825, pp. 357-362, 2020.
[65] J. D. Hunter, "Matplotlib: a 2D graphics environment," Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.
[66] F. Pedregosa et al., "Scikit-learn: machine learning in Python," Journal of Machine Learning Research, vol. 12, pp. 2825-2830, 2011.
[67] M. Abadi et al., "TensorFlow: a system for large-scale machine learning," in Proc. 12th USENIX Symp. Operating Systems Design and Implementation (OSDI), 2016, pp. 265-283.
[68] A. Paszke et al., "PyTorch: an imperative style, high-performance deep learning library," in Proc. Advances in Neural Information Processing Systems (NeurIPS), 2019, pp. 8024-8035.
[69] S. Behnel et al., "Cython: the best of both worlds," Computing in Science & Engineering, vol. 13, no. 2, pp. 31-39, 2011.
[70] B. Solis and S. Allani, "Towards autonomic workflow management systems: a survey," IEEE Transactions on Network and Service Management, vol. 17, no. 3, pp. 1351-1366, 2020.
[71] E. Deelman et al., "Pegasus: a workflow management system for science automation," Future Generation Computer Systems, vol. 46, pp. 17-35, 2015.
[72] K. Wolstencroft et al., "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud," Nucleic Acids Research, vol. 41, no. W1, pp. W557-W561, 2013.
[73] J. Köster and S. Rahmann, "Snakemake—a scalable bioinformatics workflow engine," Bioinformatics, vol. 28, no. 19, pp. 2520-2522, 2012.
[74] P. Di Tommaso et al., "Nextflow enables reproducible computational workflows," Nature Biotechnology, vol. 35, no. 4, pp. 316-319, 2017.
[75] S. Ramirez, FastAPI: Modern Python Web Framework Documentation, 2023.
[76] A. Ronacher, Flask Web Development Documentation, 2010.
[77] Django Software Foundation, Django

Web Framework Documentation, 2023.
[78] A. Solem, "Celery: distributed task queue," Celery Project Documentation, 2023.
[79] Apache Software Foundation, "Apache Airflow: a platform to programmatically author, schedule and monitor workflows," Apache Airflow Documentation, 2023.
[80] Prefect Technologies, "Prefect: the new standard in dataflow automation," Prefect Documentation, 2023.
[81] M. Stonebraker, D. Deutch, and T. Milo, "Self-driving database management systems," in Proc. 8th Biennial Conf. Innovative Data Systems Research (CIDR), 2017.
[82] A. Pavlo et al., "Self-driving database management systems," Communications of the ACM, vol. 62, no. 3, pp. 90-99, 2019.
[83] D. Spinellis, "Software engineering for machine learning systems," IEEE Software, vol. 37, no. 5, pp. 22-25, 2020.
[84] J. Bosch, H. H. Olsson, and I. Crnkovic, "Engineering AI systems: a research agenda," in Artificial Intelligence Paradigms for Smart Cyber-Physical Systems, IGI Global, 2021, pp. 1-19.
[85] B. Schmidt and D. Yarish, "Self-healing software systems: survey and synthesis," in Proc. IEEE Int. Conf. Autonomic Computing, 2018, pp. 109-118.
[86] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. Cheng, "Composing adaptive software," IEEE Computer, vol. 37, no. 7, pp. 56-64, 2004.
[87] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," IEEE Computer, vol. 36, no. 1, pp. 41-50, 2003.
[88] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: architecture-based self-adaptation with reusable infrastructure," IEEE Computer, vol. 37, no. 10, pp. 46-54, 2004.
[89] M. Salehie and L. Tahvildari, "Self-adaptive software: landscape and research challenges," ACM Transactions on Autonomous and Adaptive Systems, vol. 4, no. 2, pp. 1-42, 2009.
[90] B. H. C. Cheng et al., "Software engineering for self-adaptive systems: a research roadmap," in Software Engineering for Self-Adaptive Systems, Springer, 2009, pp. 1-26.
[91] R. de Lemos et al., "Software engineering for self-adaptive systems: a second research roadmap," in Software Engineering for Self-Adaptive Systems II, Springer, 2013, pp. 1-32.
[92] N. Medvidovic and R. N. Taylor, "A classification and comparison framework for software architecture description languages," IEEE Transactions on Software Engineering, vol. 26, no. 1, pp. 70-93, 2000.
[93] L. Bass, P. Clements, and R. Kazman, Software Architecture in Practice, 4th ed. Boston, MA, USA: Addison-Wesley, 2021.
[94] P. Bourque and R. E. Fairley, Eds., Guide to the Software Engineering Body of Knowledge (SWEBOK), v3.0. Los Alamitos, CA, USA: IEEE Computer Society, 2014.
[95] B. Boehm, "A view of 20th and 21st century software engineering," in Proc. 28th Int. Conf. Software Engineering, 2006, pp. 12-29.
[96] L. Bass, S. Stoll, and M. Bass, The Site Reliability Workbook. Sebastopol, CA, USA: O'Reilly Media, 2018.
[97] B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, Site Reliability Engineering. Sebastopol, CA, USA: O'Reilly Media, 2016.
[98] L. Bass, I. Weber, and L. Zhu, DevOps:

A Software Architect's Perspective. Boston, MA, USA: Addison-Wesley, 2015.
[99] J. Humble and D. Farley, Continuous Delivery. Boston, MA, USA: Addison-Wesley, 2010.
[100] N. Forsgren, J. Humble, and G. Kim, Accelerate: The Science of Lean Software and DevOps. Portland, OR, USA: IT Revolution Press, 2018.
[101] R. Bardini, G. Politano, S. Benso, and A. Di Carlo, "A study on Python's overhead and parallelism in data-intensive applications," IEEE Access, vol. 7, pp. 152825-152836, 2019.
[102] A. Lavin et al., "Technology readiness levels for machine learning systems," Nature Communications, vol. 13, p. 6039, 2022.
[103] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices," IEEE Access, vol. 5, pp. 3909-3943, 2017.
[104] P. Jamshidi, C. Pahl, N. C. Mendonca, J. Lewis, and S. Tilkov, "Microservices: the journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, pp. 24-35, 2018.
[105] R. Heinrich et al., "Performance engineering for microservices: research challenges and directions," in Proc. 8th ACM/SPEC Int. Conf. Performance Engineering Companion, 2017, pp. 223-226.
[106] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," IEEE Software, vol. 35, no. 3, pp. 96-100, 2018.
[107] J. Thones, "Microservices," IEEE Software, vol. 32, no. 1, p. 116, 2015.
[108] Y. Cui, Y. Song, and J. Zhu, "Toward intelligent workflow scheduling using deep reinforcement learning in cloud-edge environments," IEEE Transactions on Cloud Computing, vol. 11, no. 2, pp. 1402-1416, 2023.
[109] H. Zhao, Q. Zhang, and B. Wei, "AI-driven business process automation: a survey and taxonomy," IEEE Transactions on Engineering Management, early access, 2023.
[110] L. Garcia-Garcia, M. Bota, and A. Lozano, "Comparative study of process automation tools: RPA versus AI-based orchestration," IEEE Software, vol. 39, no. 4, pp. 64-72, 2022.
[111] Y. Wang, Y. Yang, and T. Lan, "AutoFlow: AI-assisted construction of robust automation pipelines for software systems," in Proc. IEEE Int. Conf. Software Engineering and Service Science, 2022, pp. 134-141.
[112] Shaikh, H., Khan, M. S., Mahar, Z. A., Anwar, M., Raza, A., & Shah, A. (2019, March). A conceptual framework for determining acceptance of internet of things (IoT) in higher education institutions of Pakistan. In 2019 international conference on information science and communication technology (ICISCT) (pp. 1-5). IEEE.
[113] Shaikh, H., Jokhio, M. U., Maher, Z. A., Chandio, S., Bin Abdullah, M. M., Raza, A., & Shah, A. (2018). Beyond traditional audits: The implications of information technology on auditing. International Journal of Engineering and Technology (UAE), 7(2), 5-11.
[114] Shaikh, H. (2021). Acceptance of IoT learning among university students at Pakistan. International Journal of Advanced Trends in Computer Science and Engineering.