ISSN: 2583-9888
DOI: doi.org/10.51219/JAIMLD/Praveen-kumar/594

URF PUBLISHERS
connect with research world

Journal of Artificial Intelligence, Machine Learning and Data Science

https://urfpublishers.com/journal/artificial-intelligence

Vol: 3 & Iss: 3

Research Article

AI-Driven Hybrid Strategies for Cloud Migration and Modernization of Java Applications

Praveen Kumar Koppanati*

Citation: Praveen KK. AI-Driven Hybrid Strategies for Cloud Migration and Modernization of Java Applications. *J Artif Intell Mach Learn & Data Sci 2025* 3(3), 2848-2853. DOI: doi.org/10.51219/JAIMLD/Praveen-kumar/594

Received: 03 September, 2025; **Accepted:** 10 September, 2025; **Published:** 12 September, 2025

***Corresponding author:** Praveen Kumar Koppanati, USA, E-mail: praveen.koppanati@gmail.com

ABSTRACT

The modernization of enterprise Java applications is increasingly critical as organizations transition from monolithic architectures to cloud-native environments. Hybrid cloud strategies, supported by artificial intelligence (AI), enable a balanced approach that leverages the scalability of public cloud while preserving the control of on-premises infrastructure. This paper examines AI-driven methodologies that support each phase of migration, including discovery, decomposition, deployment planning, resource allocation, automation, and monitoring. Techniques such as microservices decomposition using clustering algorithms, reinforcement learning for dynamic autoscaling, and AI-assisted DevOps pipelines are analyzed in detail. Additionally, the role of Red Hat OpenShift Service on AWS (ROSA) is highlighted as a managed hybrid solution for migrating Java services to AWS without the need for native re-engineering. The study consolidates prior research and industry practices to provide a structured framework for AI-enhanced hybrid migration and modernization of Java applications, offering insights into benefits, challenges, and future research opportunities.

Keywords: Artificial intelligence, hybrid cloud, Java modernization, cloud migration, microservices decomposition, resource allocation, Open Shift, ROSA.

1. Introduction

Java continues to serve as a cornerstone of enterprise software development, particularly in domains such as finance, healthcare, government, insurance and logistics. Despite its ubiquity, many Java applications remain monolithic, tightly coupled, and dependent on legacy middleware. Such architectures restrict scalability, hinder integration with modern platforms, and increase the cost and complexity of maintenance.

Cloud migration has emerged as a viable solution for overcoming these limitations, enabling improved agility, cost efficiency, and access to elastic resources. However, traditional lift-and-shift migration approaches often fail to capture the full benefits of cloud adoption. Running a monolithic application in a virtualized cloud environment may provide limited scalability but does not inherently deliver cloud-native capabilities such as microservices-based agility, automated scaling, or container orchestration.

Hybrid cloud strategies address these challenges by allowing organizations to distribute workloads across public cloud and on-premises environments. This approach provides greater flexibility for latency-sensitive and compliance-critical components, while enabling scalable deployment of less sensitive services in the public cloud. Artificial intelligence (AI) further enhances hybrid migration by automating application discovery, accelerating decomposition into microservices, optimizing deployment placement, and enabling adaptive resource allocation across heterogeneous environments.

Recent advances, including AI-driven tools for monolith

decomposition, reinforcement learning for autoscaling, and AI-assisted DevOps automation, indicate a shift toward more intelligent and efficient modernization workflows. In addition, managed hybrid platforms such as Red Hat OpenShift Service on AWS (ROSA) extend these strategies by allowing organizations to migrate workloads into AWS clusters without building native container solutions from scratch, preserving operational consistency across environments.

This paper surveys AI-enabled methods for hybrid migration of Java applications, synthesizes prior academic and industrial research, and proposes a structured framework that incorporates assessment, decomposition, hybrid deployment planning, optimization, automation, and monitoring. The discussion highlights both the benefits and the challenges of AI-driven hybrid migration, providing a foundation for future exploration of modernization strategies in enterprise settings.

2. Background and Related Work

2.1. Microservices decomposition for java

Mono2Micro is an IBM-developed tool for decomposing monolithic Java applications into functionally cohesive microservices. It uses runtime call graph and business use-case data to recommend partitions; evaluations show it significantly outperforms prior methods in partition quality and receives positive practitioner feedback.

CARGO (Context-sensitive Label Propagation) refines existing partitioning techniques by building a context- and flow-sensitive system dependency graph for Java EE monoliths. In experiments, CARGO improved partition quality, reduced distributed transactions, lowered latency by ~11 %, and increased throughput by ~120 %.

2.2. Hybrid cloud migration advisors

Atlas, a hybrid cloud migration advisor, assists in selecting which microservices to push to the public cloud and which to retain on-premise. Using data-driven learning, it balances three metrics—latency, availability, and cost—and offers migration plans that yield ~21 % better API latency and ~11 % cost reduction with fewer disruptions compared to conventional approaches.

2.3. AI-driven resource allocation

A reinforcementlearning (RL) based framework addresses dynamic resource allocation in hybrid cloud microservices. It continuously adapts provisioning to anticipated demands, improving cost efficiency by 30-40 %, improving resource utilization by 20-30 %, and reducing latency by 15-20 % during peak loads.

2.4. Enterprise AI tools for modernization

IBM offers AIpowered refactoring tools (Transformation Advisor, Mono2Micro, Migration Toolkit) integrated into Cloud Pak for Applications on OpenShift (hybrid Kubernetes platform). These tools aid in assessing and refactoring applications and support both VM and container environments.

Generative AI assistants (e.g., IBM's "Migration Factory" with AI assistants) provide rolebased, conversationbased interfaces that accelerate migration by enabling consultants to generate code, prompts, and user personas faster. These "gen AI assets" help automate manual processes in modernization.

Industry discussions highlight hybrid cloud's role in balancing performance, compliance, and cost especially for AI workloads. Hybrid strategies enable onprem inference, cloudbased training, and data locality, improving flexibility and addressing regulatory constraints.

AI integration into cloud migration enhances automation, governance, predictive analytics, resource optimization, and decision support though challenges like security, complexity, cost, and skills gaps remain.

3. Integrated AI-Driven Hybrid Java-Migration Framework

Figure 1 outlines a proposed framework combining the tools and concepts surveyed:

3.1. Discovery & assessment:

- Use IBM's Transformation Advisor and Migration Toolkit to evaluate Java monolith architecture, containerization readiness, and refactoring opportunities.
- Augment with AI assistants to support exploration, stakeholder interviews, and documentation generation.

3.2. Decomposition & partitioning:

- Apply Mono2Micro to obtain an initial microservice partitioning.
- Refine with CARGO's contextsensitive dependency analysis to reduce transaction coupling and improve performance.

3.3. Hybrid deployment planning:

- Use Atlas to determine which microservices should remain onpremise and which can move to public cloud, optimizing latency, cost, and availability.

3.4. Resource allocation optimization:

- Deploy an RLbased resource manager that dynamically adjusts allocations across hybrid environments, achieving cost, utilization, and latency benefits.

3.5. Automation & assistantdriven execution:

- Leverage AI assistants and generative AI assets (as in IBM's Migration Factory concept) for automating coding, deployment scripts, CI/CD configuration, and consultative decision workflows.

3.6. Monitoring & continuous optimization:

- Employ AI monitoring (AIOps) to detect performance anomalies, suggest tuning adjustments, and manage operations across hybrid environments (though specific Javamigration use is extrapolated from AIOps general practice)

4. Detailed Framework Discussion

The proposed AI-driven hybrid framework for Java application modernization can be structured into six major phases: 1) discovery and assessment; 2) decomposition and partitioning; 3) hybrid deployment planning; 4) resource allocation optimization; 5) automation and assistant-driven execution; and 6) monitoring and continuous optimization. Each phase addresses a specific challenge in transitioning legacy monolithic Java workloads toward scalable, hybrid, cloud-native architectures. The following subsections elaborate on these phases and provide representative technical examples.

4.1. Discovery and assessment

The discovery stage establishes a foundation for migration by evaluating the structure, dependencies, and cloud readiness of legacy Java systems. Traditional approaches rely heavily on manual code reviews and static analyzers, which often overlook runtime behaviors and hidden interdependencies. AI-enabled tools such as IBM Transformation Advisor and the Migration Toolkit extend this process by applying heuristic rules and machine learning models trained on prior migration projects. These systems identify non-portable APIs, assign risk scores to components, and recommend container-friendly runtimes.

A simplified illustration of an AI-driven analyzer is shown below:

```java
// Example integration with AI-based assessment
import com.ibm.ai.migration.AnalysisClient;

public class MigrationAssessment {
    public static void main(String[] args) {
        AnalysisClient client = new AnalysisClient("apikey-12345");
        String report = client.analyzeProject("/src/legacy-java-app");
        System.out.println("AI-driven assessment report:");
        System.out.println(report);
    }
}
```

The output of such tools typically provides actionable insights, such as identifying classes that can be containerized without modification, highlighting APIs requiring refactoring, and recommending suitable Java runtimes for cloud deployment.

4.2. Decomposition and Partitioning

Migrating monolithic Java applications to a hybrid cloud often requires decomposition into microservices. AI-assisted frameworks such as **Mono2Micro** and **CARGO** have advanced this process by combining call graph analysis with context- and flow-sensitive dependency modeling. These methods enable the grouping of classes into service candidates that minimize inter-service coupling while preserving functional cohesion.

An example of graph-based clustering is shown below:

```python
from sklearn.cluster import KMeans
import networkx as nx

# Load Java call graph
G = nx.read_gml("java_call_graph.gml")
X = nx.adjacency_matrix(G).todense()

# AI-driven clustering of classes into microservices
kmeans = KMeans(n_clusters=5, random_state=42).fit(X)

for node, cluster_id in zip(G.nodes(), kmeans.labels_):
    print(f"Class {node} -> Microservice {cluster_id}")
```

Once decomposition is determined, service clusters can be re-implemented as Spring Boot services.

```java
@RestController
@RequestMapping("/inventory")
public class InventoryService {
    @GetMapping("/{id}")
    public Inventory getItem(@PathVariable String id) {
        return InventoryRepository.findById(id);
    }
}
```

By coupling AI-generated decomposition with developer refactoring, enterprises can accelerate the transition from monoliths to modular, hybrid-ready architectures.

4.3. Hybrid deployment planning

Determining the optimal placement of services across public cloud and on-premises infrastructure is critical in hybrid environments. AI-assisted planners such as **Atlas** evaluate cost, latency, and availability metrics to recommend deployment strategies. For example, services processing sensitive payment data may remain on-premises, while stateless order-processing services may be deployed to the cloud.

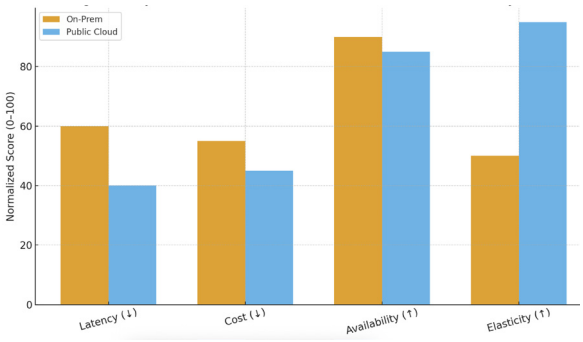

Figure 1: Hybrid Trade-Offs.

A deployment plan may be expressed in Kubernetes YAML as follows:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
spec:
  replicas: 2
  template:
    spec:
      nodeSelector:
        cloud/hybrid: "onprem"
      containers:
      - name: payment-service
        image: onprem-registry/java/payment-service:latest
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: order-service
spec:
  replicas: 3
  template:
    spec:
      containers:
      - name: order-service
        image: registry.aws/java/order-service:latest
```

This ensures compliance requirements are met while leveraging cloud scalability for non-sensitive workloads.

4.4. Resource allocation optimization

Even with optimal placement, workloads may experience fluctuating demands. Reinforcement learning (RL) enables adaptive resource management by dynamically adjusting compute and memory allocations in response to observed conditions.

In production, this framework would be extended into Kubernetes operators that autonomously scale pods across hybrid nodes based on RL-trained policies.

4.5. Automation and assistant-driven execution

AI assistants streamline repetitive migration tasks such as

creating Dockerfiles, generating deployment manifests, and configuring CI/CD pipelines. By leveraging generative AI, engineers can describe deployment requirements in natural language and receive production-ready artifacts.

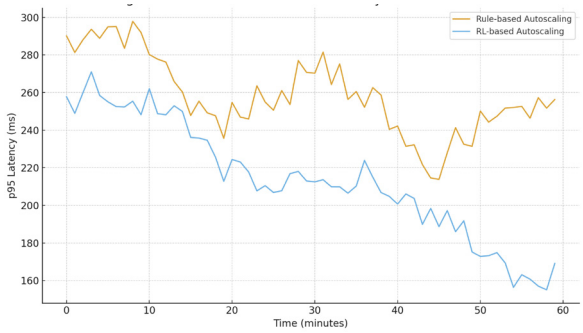

Figure 2: RL Autoscaler Reduces Latency Faster Under Load.

A simplified RL loop for autoscaling can be demonstrated as:

```python
import random

class HybridEnv:
    def __init__(self):
        self.state = {"cpu": 50, "latency": 120}

    def step(self, action):
        if action["scale_up"]:
            self.state["cpu"] -= 10
            self.state["latency"] -= 20
        else:
            self.state["cpu"] += 10
            self.state["latency"] += 15
        reward = -self.state["latency"]
        return self.state, reward

env = HybridEnv()
for episode in range(10):
    action = {"scale_up": random.choice([True, False])}
    state, reward = env.step(action)
    print(f"Episode {episode}, State={state}, Reward={reward}")
```

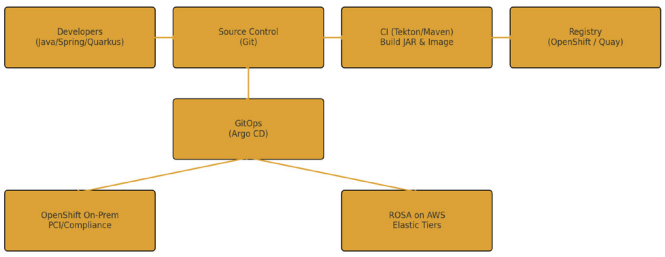

Figure 3: Automation Pipeline: CI/CD + GitOps Orchestrating Hybrid Deployments.

For instance, generating a Dockerfile for a Spring Boot service may result in:

```dockerfile
FROM eclipse-temurin:17-jdk-alpine
WORKDIR /app
COPY target/app.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java","-jar","app.jar"]
```

Similarly, a CI/CD pipeline can be automatically generated in YAML:

This level of automation reduces development overhead while ensuring consistency across environments.

```yaml
stages:
  - build
  - deploy

build:
  script:
    - mvn clean package
    - docker build -t registry/java-app:$CI_COMMIT_SHA .
    - docker push registry/java-app:$CI_COMMIT_SHA

deploy:
  script:
    - kubectl apply -f k8s/deployment.yaml
```

4.6. Monitoring and continuous optimization

Post-migration, hybrid environments require robust monitoring and anomaly detection. **AIOps** solutions combine machine learning with observability data to predict and remediate incidents proactively.

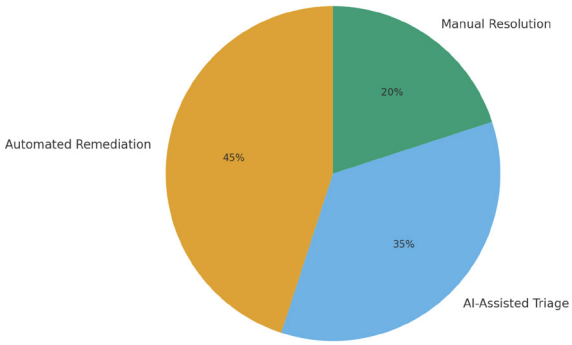

Figure 4: Incident Handling Breakdown After AIOps Adoption.

For example, latency anomalies can be detected using Isolation Forest:

```python
from sklearn.ensemble import IsolationForest
import numpy as np

latency = np.array([[120], [125], [118], [500], [520]])  # anomalies
model = IsolationForest(contamination=0.1).fit(latency)
predictions = model.predict(latency)

for val, pred in zip(latency, predictions):
    status = "Anomaly" if pred == -1 else "Normal"
    print(f"Latency {val[0]} ms -> {status}")
```

Such detection mechanisms can be integrated with monitoring tools like Prometheus and Grafana to generate real-time alerts and trigger automated remediation workflows.

4.7. Leveraging red hat openshift service on AWS (ROSA)

A significant challenge in hybrid migration is maintaining operational consistency across on-premises and cloud environments. Organizations often operate Red Hat OpenShift clusters within their data centers, but extending workloads into the public cloud can require complex engineering efforts, particularly when configuring native Kubernetes services such as Amazon Elastic Kubernetes Service (EKS). Red Hat OpenShift Service on AWS (ROSA) addresses this challenge by offering a fully managed OpenShift environment directly on AWS infrastructure.

ROSA enables enterprises to adopt a hybrid strategy

without re-architecting applications for cloud-native services. The platform provides uniform Kubernetes APIs, integrated security policies, and managed cluster operations, ensuring that applications deployed on ROSA behave consistently with their on-premises counterparts. This uniformity reduces the operational overhead associated with managing multiple orchestration environments and facilitates workload portability.

For Java modernization specifically, ROSA offers built-in support for enterprise runtimes such as JBoss EAP, Quarkus, and Spring Boot, making it well suited for hosting containerized Java microservices. Compliance-sensitive services can be retained within on-premises OpenShift clusters, while stateless or scalable components can be migrated into ROSA clusters on AWS. This distribution allows organizations to meet governance requirements while still benefiting from the elasticity of the cloud.

.7.1. Cluster provisioning and access: Provision a multi-AZ ROSA cluster with STS and opinionated CIDRs:

```
rosa login --env=production
rosa create cluster --cluster-name=java-modernization --region=us-east-1 --version=4.15.17
  --multi-az --sts --machine-cidr=10.0.0.0/16 --service-cidr=172.30.0.0/16 \
  --pod-cidr=172.31.0.0/16 --host-prefix=23 --compute-nodes=6

# Bootstrap local admin for initial access
rosa create admin --cluster=java-modernization

# Log in to the ROSA API and create a target project/namespace
oc login https://api.java-modernization.xxxxx.p1.openshiftapps.com:6443 -u kubeadmin -p <s
oc new-project retail-payments
```

4.7.2. Identity and secrets: *IAM Roles for Service Accounts (IRSA):* Use OIDC-backed STS so pods obtain AWS IAM credentials without static access keys.

IAM trust policy (assume role via service account)

```
{
  "Version": "2012-10-17",
  "Statement": [{
    "Effect": "Allow",
    "Principal": {
      "Federated": "arn:aws:iam::<ACCOUNT_ID>:oidc-provider/oidc.eks.<region>.amazonaws.co
    },
    "Action": "sts:AssumeRoleWithWebIdentity",
    "Condition": {
      "StringEquals": {
        "oidc.eks.<region>.amazonaws.com/id/<OIDC_ID>:sub": "system:serviceaccount:retail-
      }
    }
  }]
}
```

Annotated service account in OpenShift

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: payment-sa
  namespace: retail-payments
  annotations:
    eks.amazonaws.com/role-arn: arn:aws:iam::<ACCOUNT_ID>:role/rosa-retail-payments-payment
```

Database credentials as a Secret

```
apiVersion: v1
kind: Secret
metadata:
  name: rds-creds
  namespace: retail-payments
type: Opaque
data:
  username: bXlfcmRzX3VzZXI=   # base64("my_rds_user")
  password: c3VwZXJfc2VjdXJl   # base64("super_secure")
```

4.7.3. Build and deploy: OpenShift S2I, Deployment, Route:

S2I BuildConfig (Maven/Java 17)

```
apiVersion: build.openshift.io/v1
kind: BuildConfig
metadata:
  name: payment-service
  namespace: retail-payments
spec:
  source:
    type: Git
    git:
      uri: https://github.com/example-org/payment-service.git
  strategy:
    type: Source
    sourceStrategy:
      from:
        kind: ImageStreamTag
        name: java:openjdk-17-ubi8
        namespace: openshift
  output:
    to:
      kind: ImageStreamTag
      name: payment-service:latest
```

App deployment + service + TLS route

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
  namespace: retail-payments
spec:
  replicas: 3
  selector:
    matchLabels: {app: payment-service}
  template:
    metadata:
      labels: {app: payment-service}
    spec:
      serviceAccountName: payment-sa
      containers:
      - name: payment
        image: image-registry.openshift-image-registry.svc:5000/retail-payments/payment-
        env:
        - name: SPRING_DATASOURCE_URL
          value: jdbc:postgresql://my-rds.abc123.us-east-1.rds.amazonaws.com:5432/payment
        - name: SPRING_DATASOURCE_USERNAME
          valueFrom: {secretKeyRef: {name: rds-creds, key: username}}
        - name: SPRING_DATASOURCE_PASSWORD
          valueFrom: {secretKeyRef: {name: rds-creds, key: password}}
        ports:
        - containerPort: 8080
        resources:
          requests: {cpu: "250m", memory: "256Mi"}
          limits:   {cpu: "750m", memory: "768Mi"}
---
apiVersion: v1
kind: Service
metadata:
  name: payment-service
  namespace: retail-payments
spec:
  selector: {app: payment-service}
  ports:
  - name: http
    port: 80
    targetPort: 8080
---
apiVersion: route.openshift.io/v1
kind: Route
metadata:
  name: payment
  namespace: retail-payments
spec:
  to: {kind: Service, name: payment-service}
  port: {targetPort: http}
  tls: {termination: edge}
```

4.7.4. Gitops with openshift gitops (Argo CD):

```
apiVersion: argoproj.io/v1alpha1
kind: Application
metadata:
  name: retail-payments-app
  namespace: openshift-gitops
spec:
  project: default
  source:
    repoURL: https://github.com/example-org/retail-payments-gitops.git
    targetRevision: main
    path: overlays/rosa-us-east-1
  destination:
    server: https://kubernetes.default.svc
    namespace: retail-payments
  syncPolicy:
    automated: {prune: true, selfHeal: true}
    syncOptions: ["CreateNamespace=true"]
```

4.7.5. CI/CD with tekton pipelines: Build JAR, build/push image (Buildah), then update the deployment and wait for rollout.

```yaml
apiVersion: tekton.dev/v1beta1
kind: Pipeline
metadata:
  name: payment-build-deploy
  namespace: retail-payments
spec:
  tasks:
  - name: build
    taskRef: {name: maven}
    params:
    - name: GOALS
      value: ["-DskipTests", "clean", "package"]

  - name: build-image
    runAfter: ["build"]
    taskRef: {name: buildah}
    params:
    - name: IMAGE
      value: "quay.io/example/payment-service:$(tasks.build.results.commit)"

  - name: deploy
    runAfter: ["build-image"]
    taskRef: {name: openshift-client}
    params:
    - name: SCRIPT
      value: |
        oc -n retail-payments set image deploy/payment-service \
           payment=quay.io/example/payment-service:$(tasks.build.results.commit)
        oc -n retail-payments rollout status deploy/payment-service
```

5. Challenges and Best Practices

5.1. Data security and compliance

Hybrid cloud may entail sensitive data crossing boundaries. AI deployment must conform to compliance and governance policies, especially in sectors like finance and healthcare. Hybrid deployment planning (via Atlas) mitigates risk by enabling strategic placement of sensitive components.

5.2. Skills and organizational readiness

Adopting this framework requires skills in AI, ML, Kubernetes, Java refactoring, and cloud operations. Enterprises must invest in training or partner with experienced consultancies (e.g., via IBM Migration Factory models).

5.3. Cost and ROI

While AIdriven automation accelerates migration and yields resource savings, the cost of tools, cloud infrastructure, and skilled labor must be weighed. A phased, pilot-based adoption (start small and scale) can validate value before large investments.

5.4. Tool integration and interoperability

Combining Mono2Micro, CARGO, Atlas, RL resource managers, and AI assistants demands well-designed integration. Standard APIs, containerized deployment of tooling, and unified dashboards are recommended.

5.5. Trust, explainability, and transparency

AI-generated decomposition, migration plans, or resource actions must be explainable to architects and stakeholders. Tools like Mono2Micro and CARGO should surface reasoning (e.g., dependency graphs, business use-case links) to build trust.

6. Conclusion and Future Work

An AI-driven hybrid framework for modernizing enterprise Java applications has been outlined, integrating automated discovery and assessment, algorithmic decomposition, hybrid deployment planning, reinforcement-learning-based resource optimization, assistant-driven automation, and AIOps-enabled operations. The synthesis indicates that tools such as Mono2Micro and CARGO can improve service boundaries and cohesion, planners similar to Atlas can rationalize cross-environment placement against cost-latency-availability objectives and learning-based controllers can adapt resource allocations to workload dynamics. Managed OpenShift on AWS (ROSA) further supports consistency across on-premises and public cloud by preserving the OpenShift developer and operations model while exposing AWS elasticity and services.

The principal implication is that modernization outcomes depend less on a single migration pattern and more on a disciplined, AI-assisted workflow spanning assessment to operations. When applied cohesively, the approach reduces manual refactoring effort, narrows architectural risk, and promotes continuous optimization in heterogeneous environments. The framework is technology-agnostic at the pattern level but benefits from curated tooling and platform choices that minimize operational variance across sites.

Future research directions include explainable AI for decomposition and placement to increase stakeholder trust and auditability, domain-specific AIOps for Java runtime telemetry, linking bytecode-level signals to SLO enforcement, rigorous economic analyses of hybrid AI orchestration at scale, including carbon/energy considerations and reference pipelines and benchmarks that standardize evaluation across assessment, decomposition, placement, and adaptive control.

7. References

1. A. Kalia. Mono2Micro: A Practical and Effective Tool for Decomposing Monolithic Java Applications to Microservices. *arXiv*, Jul. 2021.
2. V. Nitin. CARGO: AIGuided Dependency Analysis for Migrating Monolithic Applications to Microservices Architecture. *arXiv*, 2022.
3. KH. Chow. Atlas: Hybrid Cloud Migration Advisor for Interactive Microservices. *arXiv*, 2023.
4. B. Barua, M. S. Kaiser. AIDriven Resource Allocation Framework for Microservices in Hybrid Cloud Platforms. *arXiv*, 2024.
5. https://www.researchgate.net/publication/387121578_The_Intersection_of_AI_and_Cloud_Modernization_Challenges_and_Opportunities.
6. Omoike, Oreoluwa. (2024). Leveraging AI TO Improve Cloud Modernization. 04. 688-691.
7. Siddharth Choudhary Rajesh, Dr. Vishwadeepak Singh Baghela. Enhancing Cloud Migration Efficiency with Automated Data Pipelines and AI-Driven Insights. International Journal of Innovative Science and Research Technology (IJISRT), 2025; 9: 3670-3690.
8. M. A. Khan, R. Walia. Intelligent Data Management in Cloud Using AI. *2024 3rd International Conference for Innovation in Technology (INOCON)*, Bangalore, India, 2024; 1-6.
9. Perugu, Prasanna Kumar. AI and Machine Learning in Cloud Migration: Enhancing Efficiency and Performance. August 15, 2024.
10. Kim, Y., Park, J., Kwon, H. AI-driven decision-making in cloud resource management: A case study ofpredictive analytics. International Journal of Cloud Computing, 2020;7: 305-322.
11. McGrath, G, Short, J. The evolution of serverless computing: AI and its impact on modern cloudarchitecture. IEEE Cloud Computing, 2019; 6: 6-14.