Research Square

Preprints are preliminary reports that have not undergone peer review.
They should not be considered conclusive, used to inform clinical practice,
or referenced by the media as validated information.

Application Platform Abstraction Over Heterogeneous Cloud Infrastructure

Shaik Mohammed Sameer

shaiksameer3909sam@gmail.com

Koneru Lakshmaiah Education Foundation (KL University)

Pravallika Gubba

pravallikagubba5@gmail.com

Koneru Lakshmaiah Education Foundation (KL University)

Jeethu Kumar Ganchi

Jeethukumar573@gmail.com

Koneru Lakshmaiah Education Foundation (KL University)

Khammapati Ganesh

2300003359@kluniversity.in

Koneru Lakshmaiah Education Foundation (KL University)

Dr. N. Raghavendra Sai

nallagatlaraghavendra@kluniversity.in

Koneru Lakshmaiah Education Foundation (KL University)

Research Article

Keywords: cloud computing, multi-cloud, portability, interoperability, platform abstraction, Kubernetes, behavioral drift, observability, fault injection, reproducibility

Posted Date: November 26th, 2026

DOI: https://doi.org/10.21203/rs.3.rs-11864594/v1

License: This work is licensed under a Creative Commons Attribution 4.0 International License.
Read Full License

Additional Declarations: The authors declare no competing interests.

Application Platform Abstraction Over Heterogeneous Cloud Infrastructure

Shaik Mohammed Sameer, Pravallika Gubba, Jeethu Kumar Ganchi, Khammapati Ganesh
Department of Computer Science and Engineering
KL University, Andhra Pradesh, India
Guide: Dr. N. Raghavendra Sai

***Abstract*—Deploying the same cloud-native application across multiple cloud providers does not guarantee equivalent runtime behavior. This paper presents CLOUD-223, a platform abstraction and evaluation workflow for assessing portability across Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform (GCP). The workflow combines a provider-independent deployment path, a frozen reference implementation, shared telemetry, infrastructure-difference analysis, controlled fault injection, recovery checks, and SHA-256 evidence tracking. In the reported three-batch static-seed experiment, three provider-visible infrastructure leakage points were identified, five variance categories were tracked, and the aggregate behavioral-parity result was 33.33%. The project record separately reports a batch-level parity standard deviation of 47.14 percentage points and a static-seed repeatability standard deviation of 0.0. All five tested negative scenarios were reported as successfully recovered. These results apply only to the workload, configurations, and experimental conditions represented in the project record and should not be interpreted as general portability or resilience scores for AWS, Azure, or GCP. The study therefore treats deployment compatibility, runtime behavioral parity, and recovery behavior as separate properties of multi-cloud portability.**

***Index Terms*—cloud computing, multi-cloud, portability, interoperability, platform abstraction, Kubernetes, behavioral drift, observability, fault injection, reproducibility**

I. Introduction

Cloud applications are increasingly deployed across multiple providers for reasons that include resilience, regulatory requirements, workload placement, cost management, and reduced dependence on a single vendor. However, moving the same application between providers does not guarantee equivalent runtime behavior. Even when a deployment definition is accepted and the required resources are provisioned, differences in service semantics and infrastructure can remain visible during execution

You can fix this problem by using a provider abstraction layer. A provider abstraction layer gives a way for your apps to ask for cloud services and resources. This can get rid of the need for application programming interfaces (APIs) for every single provider. It also makes the work of setting things up for a platform much easier.. A provider abstraction layer does not make every environment exactly the same. Even if the services look the same they might act differently when it comes to scheduling, resource provisioning, networking, storage behavior startup behavior and runtime performance. Because of this you should only look at provisioning as a sign that an app can be deployed not as a sign that the app will behave the same way everywhere.

CLOUD-223 was made because of this difference. CLOUD-223 checks if a specific application can be deployed in the way, on AWS, Azure and GCP. CLOUD-223 also looks at whether the application can get results in its behavior by using a common yardstick and the evidence we find from experiments.

1) A provider abstraction layer that presents a common deployment interface for AWS, Azure, and GCP.
2) A frozen-reference procedure for assessing behavioral portability rather than using deployment success as the final acceptance criterion.
3) A shared way to collect data and compare results to find differences that still show up after using the middleman.
4) A controlled fault-injection campaign that evaluates recovery separately from steady-state behavior.
5) An evidence model that associates experiment outcomes with deployment and test conditions through SHA-256 checksums.
6) An empirical evaluation that treats deployment portability and behavioral portability as distinct properties.

II. Background and Related Work

A. Cloud Portability and Interoperability

Cloud portability concerns the ability to move or operate an application, workload, or service in another cloud environment without unacceptable modification. Interoperability is broader and concerns the ability of heterogeneous systems or services to exchange information and work together. The two concepts overlap, but they are not equivalent. Cloud providers may expose services with similar names and capabilities while using different APIs, configuration models, scheduling policies, and runtime semantics.

B. Abstraction and Middleware

Middleware and platform-abstraction systems place a common interface between applications and provider-specific services. Prior work indicates that this approach can reduce provider dependence and simplify portability, although the abstraction layer may introduce additional overhead [2], [3]. For this study, the key limitation is that interface compatibility does not establish behavioral equivalence. A common API can simplify deployment while provider-specific execution characteristics remain visible at runtime.

C. Runtime and Empirical Portability

Runtime evaluation provides evidence that cannot be obtained from service descriptions alone. Studies of multi-cloud APIs have shown that provider-independent interfaces can exhibit different performance characteristics from native provider APIs under real workloads. Recent work on serverless-container platforms also indicates that nominal feature compatibility does not necessarily produce identical execution behavior [6]. Recent work also emphasizes that multi-cloud application design and runtime operation remain open research areas, reinforcing the value of measurements taken from actual deployments [4], [5].

D. Fault Injection and Resilience Evaluation

We need to look at recovery behavior. This is because systems that act the way during normal times might act very differently when a resource runs low or a part fails. Chaos engineering uses planned failures to see how these systems act. CLOUD-223 uses this idea for cross-provider portability. CLOUD-223 keeps fault tests separate from steady-state measurements, which matches studies on resilient multi-cloud service layers [3]. Doing this helps the study tell the difference between runtime behavior and how the system recovers after we pick certain failures.

E. Research Gap

Other studies look at portability, interoperability, abstraction, runtime comparison and resilience from angles. CLOUD-223 brings all these pieces together into one measurement workflow. The measurement workflow starts with a common deployment definition. It then uses a fixed reference to compare behavior gathers telemetry across providers looks at infrastructure differences runs controlled faults and saves all the evidence so it can be traced.

The gap we found is not just that we need another abstraction layer. The real gap is the need for a measurement workflow that can tell the difference between a deployment and what actually happens during runtime and recovery. This workflow must also link each result to the conditions that caused it. CLOUD-223 fills this gap by treating portability as a property we test in an experiment than just assuming it works because the deployment is compatible. Table I shows how CLOUD-223 compares to the types of previous work.

TABLE I
RESEARCH POSITIONING OF CLOUD-223

Approach	**Main emphasis**	**CLOUD-223 extension**
Portability surveys	Interoperability and vendor lock-in	Adds a measured experimental evidence path
Middleware / PaaS abstraction	Provider-independent service access	Adds direct runtime comparison
Runtime portability studies	Execution, latency, and elasticity	Adds a frozen reference and recovery evidence
Chaos engineering / fault injection	Resilience validation	Applies controlled recovery testing across providers
CLOUD-223	Abstraction, telemetry, fault injection, and evidence	Combines these elements into one behavioral acceptance workflow

III. RESEARCH QUESTIONS AND OBJECTIVES

The study is organized around five research questions:

RQ1: Can the provider-independent abstraction deploy the application definition on AWS, Azure and GCP without needing application code made just for one provider?

RQ2: Which infrastructure differences can the application still see after it deploys through the abstraction layer?

RQ3: How much does the observed application behavior actually match the reference during our tests?

RQ4: Can the platform recover, from the specific infrastructure and workload faults we choose under the test conditions?

RQ5: Can decisions about portability be kept as evidence connected to the deployment and test conditions that led to them?

The goals are simple: create a version-locked reference find infrastructure leakage check for similarity gather evidence from deployment and telemetry and test recovery through controlled negative tests. These goals are intentionally limited to the setup described in the project record.

A. Research Hypotheses

H1: A way to abstract away the provider can deploy the logical application definition on AWS, Azure and GCP under the tested configuration

H2: Some differences, between providers can still be seen when the application is deployed using a common abstraction.

H3: Being able to deploy an application does not always mean it will behave the way under the same workload.

H4: Intentional faults can be added to test how well recovery works independently of how the application behaves in conditions

IV. METHODOLOGY

A. Research Design

The evaluation follows a controlled design. The same clear application definition and measurement steps are planned to be used for each target provider while a fixed reference implementation acts as the baseline, for comparing behavior. Runtime measurements are collected through a common telemetry path. Fault scenarios are executed separately from the steady-state experiment so that recovery can be assessed as an independent portability property.

The design separates three levels of portability. Deployment portability asks whether the application can be provisioned through the common path. Behavioral portability asks whether observed runtime measurements fall within the defined acceptance envelope relative to the reference. Recovery portability

TABLE II
EXPERIMENTAL SETUP AND REPRODUCIBILITY RECORD

Parameter	Recorded value / evidence status
Target providers	AWS, Azure, GCP
Application workload	`orders-api` container image
AWS runtime	EKS; `t3.medium` node pool
Azure runtime	AKS; `Standard_D2s_v5`
GCP runtime	GKE Autopilot
Kubernetes clusters	One target cluster per provider
Application/runtime stack	React 19, Node.js 20, Docker, Kubernetes
Workload seed	Static seed
Experimental batches	3 comparison batches
Repeatability trials	3 trials; SD = 0.0 for static-seed data
Negative-test campaign	5 scenarios; 5/5 recovered
Variance categories	IOPS, latency, startup, memory, CPU
Leakage points	Storage, network, cold-start
Node / instance configuration	AWS `t3.medium`; Azure `Standard_D2s_v5`; GKE Autopilot
Network configuration	Cross-provider network behavior evaluated
Storage configuration	Cross-provider storage behavior evaluated
Workload execution	Static-seed, three-batch comparison
Telemetry	OpenTelemetry + Prometheus
Fault health polling	500 ms polling for the chaos/recovery workflow
Acceptance criteria	Parity evaluated across three comparison batches; recovery target $\geq$80%
Statistical procedure	3-batch parity SD = 47.14 pp; 3-trial static-seed repeatability SD = 0.0
Evidence integrity	SHA-256 manifest; 218 evidence files reported

asks whether the application returns to its expected operational state after selected faults.

B. Experimental Setup and Reproducibility

In this project evaluation I looked at three providers. I set up one Kubernetes cluster for each provider. Used the `orders-api` workload. I also used a seed, three comparison batches and five negative-test scenarios. The runtime targets I recorded were AWS EKS with a `t3.medium` pool, Azure AKS with `Standard_D2s_v5` and GKE Autopilot.

I also tracked five variance categories. The results showed a 33.33% aggregate parity score and a 47.14-percentage-point batch-level parity standard deviation. For the seed the repeatability standard deviation was 0.0. I also saw 100% recovery across the five tests. You can find the rest of the reproducibility fields, in Table II. Table II summarizes the configuration, the workload, the provider targets, the evaluation batches, the fault scenarios and the evidence controls used in this study.

C. Evaluation Dimensions

The evaluation dimensions are summarized in Table III.

D. Behavioral Parity

For each measured dimension i, let x_i be the observed value, r_i the corresponding frozen-reference value, and τ_i the acceptance tolerance for that dimension. A measurement passes when its deviation from the reference lies within that tolerance:

$$b_i = \begin{cases} 1, & |x_i - r_i| \leq \tau_i, \\ 0, & |x_i - r_i| > \tau_i. \end{cases} \tag{1}$$

The aggregate behavioral-parity score is

TABLE III
EVALUATION DIMENSIONS

Dimension	Purpose of the measurement
Deployment portability	Determines whether the common deployment path succeeds on each provider.
Behavioral parity	Measures how closely observed application behavior matches the frozen reference.
Infrastructure leakage	Identifies provider-specific differences that remain visible to the application.
Recovery portability	Determines whether the system returns to the expected state after controlled faults.
Evidence integrity	Links results to the corresponding deployment and test conditions through checksums.

TABLE IV
BEHAVIORAL AND INFRASTRUCTURE DIMENSIONS ASSESSED

Dimension	Comparison basis	Role
IOPS	Cross-provider / reference	Infrastructure variance
Request latency	Frozen reference	Behavioral comparison
Startup time	Frozen reference	Behavioral comparison
Memory footprint	Frozen reference	Behavioral comparison
CPU utilization	Cross-provider / reference	Infrastructure variance

$$\text{Behavioral Parity}(\%) = \frac{\sum_{i=1}^{M} b_i}{M} \times 100, \tag{2}$$

where M is the number of evaluated measurements. The project evaluation artifact reports 33.33% for the three-batch static-seed experiment. The reported score is retained as an aggregate outcome of the evaluated configuration.

The behavioral dimensions used in the steady-state comparison are summarized in Table IV.

E. Fault-Injection Campaign

The recovery campaign contains five negative tests: pod failure (NT-1), network-latency injection (NT-2), memory pressure (NT-3), storage throttling (NT-4), and forced cold-start (NT-5). Recovery is judged from the observed return to the expected operational state. All five scenarios are reported as recovered, giving a campaign-level recovery rate of 100%.

V. SYSTEM ARCHITECTURE

The CLOUD-223 is built as a platform with six layers. The topmost layer, called the presentation layer uses React 19 TypeScript and Vite. The next layer, the application and API layer uses Node.js 20 Express and Socker.IO to provide the API surface. The abstraction layer sets up the interface creates the ProviderFactory and builds a capability model that stores what each provider can do. The provider layer follows the abstraction and pulls in the AWS SDK v3, the Azure Resource Manager SDK with MSAL support and Google Cloud client libraries [7]–[9]. The infrastructure and runtime layer

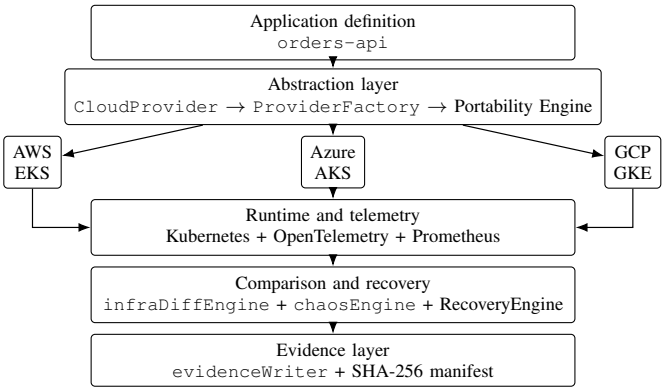

Fig. 1. CLOUD-223 architecture from application definition through provider abstraction, runtime observation, comparison, recovery, and evidence generation.

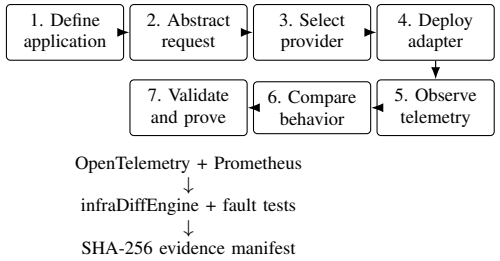

Fig. 2. Evaluation workflow used to deploy, observe, compare, test, and record the CLOUD-223 experiment.

handles orchestration for Kubernetes [10]. Finally the data and observability layer keeps data in PostgreSQL or SQLite based on where the system's deployed. It also uses OpenTelemetry, Prometheus and the evidence writer; the evidence writer saves the evidence manifests, from experiments. Evaluation starts by defining the application. The ProviderFactory picks the target cloud. Sends the needed operations to the correct provider adapters. Runtime telemetry is then. Processed by the infraDiffEngine to compare it with reference measurements. The evidenceWriter then saves the evidence. Adds a checksum. The overall architecture of the platform is shown in Fig. 1. This approach aims to reduce the differences that arise from how each provider deploys the system.

VI. Implementation

A. Platform Components

Besides the provider abstraction there are four parts that make up the evidence path. These parts are infraDiffEngine, chaosEngine, RecoveryEngine and evidenceWriter. These four parts work together to link the provider abstraction to runtime observation, infrastructure-difference analysis, controlled failures, recovery checks and evidence generation. You can see how the evaluation sequence works in Fig. 2. If you want to see what each part does please look at Table V.

B. Infrastructure Difference Engine

The `infraDiffEngine` compares infrastructure and application measurements collected from the evaluated providers.

TABLE V
Main platform components

Component	Role in the research workflow
Provider abstraction / ProviderFactory	Provides a common deployment path while keeping provider-specific SDK operations behind the abstraction.
infraDiffEngine	Compares infrastructure and application measurements and identifies leakage.
chaosEngine	Executes the controlled fault scenarios.
RecoveryEngine	Checks whether the application returns to the expected operational state.
evidenceWriter	Stores experiment outcomes and links them to SHA-256 evidence records.

The frozen KPI record tracks five variance categories—IOPS, latency, startup, memory, and CPU—while the reported leakage findings identify storage, network, and cold-start differences that remained visible after abstraction.

C. Observability

OpenTelemetry and Prometheus are used to collect runtime and deployment measurements. OpenTelemetry provides the telemetry instrumentation and collection framework used in the evidence path [11]. Observability is central to the evaluation because the principal portability claims depend on what the application actually does during execution. A provider reporting that a resource has been created is not, by itself, evidence that the resulting application runtime is equivalent to the reference.

D. Evidence Model

The evidence writer stores key performance indicator (KPI) results and negative-test outcomes together with SHA-256 cryptographic hashes. The project presentation reports 218 tracked evidence files in the current validation artifact. The *Application*, *Deployment*, *Provider*, *TelemetryRecord*, *NegativeTest*, and *EvidenceRecord* entities provide a traceable path from an experimental result back to its deployment and test condition. The hash is used for evidence integrity; it does not replace the need to preserve the raw measurements and analysis procedure.

E. Reproducibility Protocol

The experiment is intended to be traceable to a version-controlled environment. The evidence model is designed to record the application or platform commit, container image information, Kubernetes and runtime versions, provider region, resource allocation, cluster configuration, workload and seed, experiment duration and repetitions, telemetry interval, dependency versions, configuration profile, and the procedure used to generate the evidence-manifest checksum.

The project evaluation artifact provides the implementation stack, provider targets, workload identity, seed condition,

TABLE VI
FROZEN EXPERIMENTAL OUTCOMES AND EVIDENCE STATUS

Metric	Result	Evidence status
Infrastructure leakage points	3	Storage, network, and cold-start
Variance categories	5	IOPS, latency, startup, memory, CPU
Behavioral parity	33.33%	Aggregate score across three batches
Batch parity standard deviation	47.14 pp	Reported across the three-batch parity sample
Static-seed repeatability SD	0.0	Separate repeated measurement under static-seed conditions
Recovery rate	100%	5/5 negative tests recovered

batch/trial counts, KPI outcomes, and evidence mechanism. Table II summarizes the configuration used for the reported evaluation.

VII. EXPERIMENTAL RESULTS

The frozen evaluation record reports six principal quantitative outcomes: three infrastructure leakage points, five variance categories, an aggregate behavioral-parity score of 33.33% across three batches, a batch-level parity standard deviation of 47.14 percentage points, a separate static-seed repeatability standard deviation of 0.0, and 100% recovery across five negative tests.

The reported outcomes describe the tested workload and configuration. They are evaluation results for this experiment, not provider-wide benchmarks. Table VI records the principal outcomes and their experimental interpretation.

A. Infrastructure Leakage

Three infrastructure leakage points were identified in the frozen project record: storage, network, and cold-start. These leakage points are analytically distinct from the five tracked variance categories: IOPS, latency, startup, memory, and CPU. The distinction is retained because a variance category describes a measured dimension, whereas a leakage point describes a provider-visible difference identified by the analysis.

The important observation is not that a fixed number of leakage points should occur in every cloud deployment. The finding is more limited and more useful: provider-specific behavior can remain observable after a common abstraction has been applied. The exact leakage pattern is expected to depend on the workload, configuration, resource allocation, and measurement conditions. Table VII summarizes the named and unnamed variance categories.

B. Behavioral Parity

The reported behavioral-parity result is 33.33% for the three-batch static-seed experiment. The two standard-deviation values describe different aspects of the experiment: 47.14 percentage points refers to variation across the three-batch parity sample, whereas 0.0 refers to repeatability under the fixed static-seed condition. They are therefore not interchangeable statistics.

The project presentation also gives a representative cold-start readout for the `orders-api` walkthrough: 480 ms on AWS EKS with a `t3.medium` node pool, 810 ms on Azure AKS with `Standard_D2s_v5`, and 390 ms on GKE Autopilot. These three values are presented as a representative readout, not as the complete raw measurement matrix used to calculate the aggregate parity score.

The 33.33% value is an aggregate parity score; it is not a percentage difference between providers. Its interpretation depends on the project's workload and acceptance definition and is therefore reported specifically for the evaluated configuration.

TABLE VII
VARIANCE CATEGORIES AND REPORTED LEAKAGE POINTS

Item	Recorded status
IOPS	Tracked variance category
Latency	Tracked variance category
Startup	Tracked variance category
Memory	Tracked variance category
CPU	Tracked variance category
Storage	Reported leakage point
Network	Reported leakage point
Cold-start	Reported leakage point

TABLE VIII
REPRESENTATIVE COLD-START READOUT FROM THE PROJECT WALKTHROUGH

Provider	Runtime target	Cold-start
AWS	EKS; `t3.medium`	480 ms
Azure	AKS; `Standard_D2s_v5`	810 ms
GCP	GKE Autopilot	390 ms

C. Recovery

All five negative tests were reported as recovered. These tests were failure, network-latency injection, memory pressure, storage throttling and forced cold-start. Recovery was assessed by looking at the return to the expected state not just by the disappearance of an error message.

The 100% recovery value applies to these five tested scenarios. Recovery does not establish a general resilience guarantee. A broader claim would require fault classes, more repetitions and measurements of recovery variability, across different environments.

VIII. DISCUSSION

The main discovery is that deployment compatibility and runtime equivalence are properties. A common abstraction can turn an application definition into provider- operations. Differences in storage, networking, compute behavior, startup characteristics or failure handling stay visible.

The results support a staged approach to portability assessment. Configuration validation checks if the planned deployment can be described. Deployment validation checks if the needed resources can be set up. Behavioral validation compares what happens at runtime, with a reference and an acceptance envelope. Fault validation looks at whether chosen failures bring the system to the expected state. By keeping these checks separate we can see exactly which stage of portability is impacted.

The evidence-manifest mechanism gives traceability by linking the application, deployment, provider, telemetry and negative-test records. The SHA-256 hash helps check that the recorded evidence is intact. It does not prove validity on its own. Therefore a reproducible artifact should keep the measurements, configurations analysis code fault definitions and the evidence-generation procedure.

The reported 33.33% behavioral-parity value and three leakage points come from the configuration. They are shown as experiment-level evidence not as provider rankings or universal portability scores. Thus the main contribution is a measurement workflow and experimental demonstration, not a portability benchmark.

IX. Scope and Limitations

The numbers in this paper come from the frozen CLOUD-223 project evaluation artifact. This study is limited to the provider set, workload identity, runtime targets, batch/trial structure, named variance categories, leakage findings, aggregate parity, repeatability statistic and the five-test recovery outcome that we set up in Table II.

Because of this the study is an evaluation of one specific setup. It is not a benchmark for AWS, Azure or GCP, as platforms. The 33.33% parity score only describes what we recorded in this comparison. It does not measure the quality of a provider. Show a universal portability percentage. Also the 100% recovery result only covers the five fault scenarios we tested. This result does not describe every way cloud-native systems can fail.

If you want to repeat this study on your own you should use the experiment package provided as supplementary material. That package includes the observations, tolerance configuration, environment metadata, experiment code fault definitions and the SHA-256 evidence manifest. These materials help people verify the work. They also let people recompute provider-level statistics confidence intervals and effect sizes if they need to.

X. Threats to Validity

A. Internal Validity

Public cloud measurements can be influenced by things happening in the background decisions made by the provider short-term network issues and the cost of collecting data. The evaluation record provided does not separate these factors on its own. Because of this a difference that is seen cannot always be blamed on how the provider acts. A better design, in the future should take measurements during set time periods. Say what was happening each time a measurement was taken.

B. External Validity

The assessment includes 3 providers, each with a single Kubernetes cluster. It doesn't include cross-region deployments, multiple clusters, stateful or storage heavy workloads or long running production style experiments. Thus, the results should be viewed as evidence for the configuration and workload tested, not as a general characterisation of multi-cloud portability.

C. Construct Validity

The 33.33% parity result depends on the definition of behavioural agreement, the choice of the threshold values and the dimensions in the comparison. Changing the acceptance envelope may change the parity value reported. The individual observations and the pass/fail counts should thus be preserved with the experimental artefact, i.e. the precise thresholds.

D. Reproducibility

The experiment uses a fixed seed and three batches. The project record shows 47.14 percentage points as the deviation across the three-batch parity sample and 0.0 as a separate repeatability standard deviation under the static-seed condition. The second number shows repeatability for the fixed condition. Neither number shows variability across other workloads other times, other regions or other provider configurations.

The five fault scenarios give negative-path coverage but they do not cover all possible failures in cloud-native systems. More experiments should include cases, like node loss, dependency failure, credential expiration, control-plane interruption, service throttling and cascading failures. These cases need to be tested times to estimate recovery variability.

XI. Conclusion

CLOUD-223 gives a step- by- step process to test how behavior works across many different cloud environments. The platform mixes provider abstraction with a fixed reference, shared telemetry looks at infrastructure differences and uses controlled fault injection, recovery checks and checksummed evidence. The method does not simply see a deployment as proof that runtime behavior's the same. In the tested conditions three infrastructure leakage points were found. Five variance categories were. Behavioral parity was 33.33%. The batch-level parity standard deviation was 47.14 percentage points. Static. Seed repeatability SD was 0.0 and all five negative tests recovered successfully. These are experiment-level outcomes. CLOUD-223's main contribution is separating deployment, recovery portability into stages along with an evidence path that links results, to deployment and test conditions. The reproducibility materials listed in the Scope and Limitations section define the artifact needed for verification.

XII. Future Work

I think the next evaluation stage needs to move using the same old static-seed batches. We should use dynamic. Randomized workloads instead. This will give us a better view of how behavioral stability holds up when things start changing. It would also help to increase the number of repeated trials. Doing trials will help us get more reliable estimates of variability. I also suggest running -cluster and cross-region experiments. We need to see if the observations stay the same when the deployment topology changes.

We can also make the fault-injection library bigger. We should add things like node loss, dependency failures, credential expiration, control-plane interruptions, service throttling and cascading faults. If the experimental setup allows it we should

not just say if a system recovered or not. We should actually record the detection time recovery time and recovery variability.

In analyses I believe we should report confidence intervals and effect sizes next to the parity measurements. This will make it easier to tell if a provider difference is actually important or if it is normal variation, in public-cloud environments. Once we do that we can add providers and workload classes without having to change the core evidence model.

Looking ahead the telemetry we collect could help with predictive portability analysis. We could use measurements to guess the likelihood of behavioral drift before a migration happens. We could even extend the evidence model to include compliance and security records. That way we can look at portability, reliability and governance inside the same deployment evidence framework.

XIII. Artifact and Data Availability

I suggest you include the experiment package as an extra item. This frozen experiment package should hold the observations, the threshold configuration, the environment metadata, the analysis code, the fault definitions and the SHA-256 evidence manifest. The project presentation says there are 218 tracked evidence files. You must keep those 218 tracked evidence files along, with the submission artifact. Please make sure the two reported standard deviations stay as statistics.

References

[1] C. Ramalingam and P. Mohan, "Addressing Semantics Standards for Cloud Portability and Interoperability in Multi Cloud Environment," *Symmetry*, vol. 13, no. 2, Art. no. 317, 2021, doi: 10.3390/sym13020317.

[2] S. Bharany *et al.*, "Efficient Middleware for the Portability of PaaS Services Consuming Applications among Heterogeneous Clouds," *Sensors*, vol. 22, no. 13, Art. no. 5013, 2022, doi: 10.3390/s22135013.

[3] E. Bainomugisha and A. Mwotil, "Crane Cloud: A Resilient Multi-Cloud Service Abstraction Layer for Resource-Constrained Settings," *Development Engineering*, vol. 7, Art. no. 100102, 2022, doi: 10.1016/j.deveng.2022.100102.

[4] J. Alonso, L. Orue-Echevarria, V. Casola, A. I. Torre, M. Huarte, E. Osaba, and J. L. Lobo, "Understanding the Challenges and Novel Architectural Models of Multi-Cloud Native Applications — A Systematic Literature Review," *Journal of Cloud Computing*, vol. 12, Art. no. 6, 2023, doi: 10.1186/s13677-022-00367-6.

[5] M. Moravcik, P. Segec, M. Kontsek, and L. Zidekova, "Model-Driven Approach to Cloud-Portability Issue," *Applied Sciences*, vol. 14, no. 20, Art. no. 9298, 2024, doi: 10.3390/app14209298.

[6] D. T. T. Nguyen and T. de Souza-Daw, "Evaluating Multi-Cloud Portability and Elastic Scaling Across Serverless-Container Platforms," *International Journal of Cloud Applications and Computing*, vol. 16, no. 1, pp. 1–22, 2026, doi: 10.4018/IJCAC.406105.

[7] Amazon Web Services, "AWS SDK for JavaScript v3 — Developer Guide," AWS Documentation, 2026. [Online]. Available: https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/, accessed Aug. 2026.

[8] Microsoft, "Azure SDK and Microsoft Authentication Library Documentation," Microsoft Learn, 2026. [Online]. Available: https://learn.microsoft.com/en-us/entra/msal/, accessed Aug. 2026.

[9] Google Cloud, "Cloud Client Libraries Documentation," Google Cloud, 2026. [Online]. Available: https://docs.cloud.google.com/sdk/docs/libraries-reference, accessed Aug. 2026.

[10] Cloud Native Computing Foundation, "Kubernetes Documentation," Kubernetes, 2026. [Online]. Available: https://kubernetes.io/docs/home/, accessed Aug. 2026.

[11] OpenTelemetry Authors, "OpenTelemetry Documentation," OpenTelemetry, 2026. [Online]. Available: https://opentelemetry.io/docs/, accessed Aug. 2026.

Supplementary Files

This is a list of supplementary files associated with this preprint. Click to download.

- S5generatemanifest.py
- S4BatchResultsTEMPLATE.csv
- S3FaultScenarios.csv
- S2EnvironmentMetadata.json
- S1ReportedOutcomes.csv
- S0README.md