Economic Sciences
https://economic-sciences.com
ES(2026)22, 6(S), 31-40 | ISSN:1505-4683

An Analytical Study of Cloud Computing and Portability with Reference to Serverless Cloud

Mr. Gokul Bodke[1], Dr. Samadhan Bundhe[2]
[1]Research Student, Sandip University, Nashik
Email:, gokulbodke@yahoo.com
[2]Associate Professor, Research Guide, Sandip University, Nashik
Email:samadhan.bundhe@sandipuniversity.edu.in

Abstract

Cloud computing has fundamentally altered the way organisations think about technology infrastructure, data management, and application delivery. Among the many dimensions of this transformation, portability, the capacity of workloads, data, and services to move freely across heterogeneous cloud environments, remains one of the most consequential yet underexplored challenges. This paper undertakes an analytical examination of cloud computing portability, situating that inquiry within the rapidly maturing paradigm of serverless cloud architectures. Drawing on technical analysis and conceptual reasoning, the study investigates the structural impediments that hinder portability in contemporary cloud ecosystems, the specific complications that serverless execution models introduce, and the strategies, both technical and organisational, that practitioners can employ to mitigate lock-in and extend workload mobility. The findings suggest that container-based abstraction layers, declarative Infrastructure-as-Code frameworks, and open-standard function runtimes collectively constitute the most viable path toward meaningful portability in serverless environments.

Keywords: *cloud computing, portability, serverless architecture, vendor lock-in, Function-as-a-Service, multi-cloud, Infrastructure-as-Code, WebAssembly*

1. Introduction

If you asked a software engineer in the early 2000s to describe where their company's servers lived, the answer was almost always the same: down the corridor, in the basement, or in a rented cage at a nearby data centre. The idea that you could rent computing by the minute, dial it up or down like a tap, and pay only for what you actually used would have sounded speculative at best. Yet that is precisely the reality that cloud computing has made routine, and the speed of that change has been, in retrospect, quite extraordinary.

What drew organisations to the cloud — initially, at least — was a fairly practical set of incentives. The capital expenditure of building and maintaining on-premises infrastructure was considerable, and the operational burden of keeping hardware patched, cooled, and running through the night was one that many teams were glad to hand off. Early adopters found that cloud platforms genuinely delivered on the promised agility: new environments could be spun up in minutes, global deployments that once required months of procurement and logistics could be completed in days. For startups in particular, this shift was transformative, since it meant they could compete at a scale that their infrastructure spending alone would never have allowed.

Over time, however, a quieter and rather more uncomfortable reality has begun to emerge. Organisations that have been living in the cloud for several years — some of them for more than a decade — are discovering that the flexibility they were promised comes with a catch that was not always made explicit at the outset. The managed services, proprietary APIs, and tightly integrated tooling ecosystems that make cloud platforms productive also make them sticky. Moving a workload from one cloud provider to another is rarely as straightforward as it sounds, and the cumulative weight of accumulated dependencies can make what seems like a simple strategic decision into a multi-year engineering undertaking.

This is the problem of portability, and it sits at the heart of the present study. Cloud portability, broadly understood, refers to the degree to which an application, dataset, or workload can be moved from one computing environment to another without requiring fundamental redesign or incurring costs

that render the migration impractical. It is not a new concern — the challenge of vendor dependency predates cloud computing by decades, stretching back to the era of proprietary Unix variants and bespoke enterprise middleware — but cloud computing has given it a new urgency and a new shape (Opara-Martins, Sahandi, & Tian, 2016).

The particular angle that this paper takes on the portability question is through the lens of serverless computing. Serverless architectures — specifically the Function-as-a-Service (FaaS) model popularised by AWS Lambda in 2014 and subsequently adopted by Google, Microsoft, IBM, and a growing ecosystem of independent providers — represent what might reasonably be called the furthest extension of the cloud abstraction logic. In a serverless environment, the developer's responsibility is reduced to writing the business logic of individual functions; everything else — provisioning, scaling, patching, runtime management, billing granularity — is handled by the platform. The appeal of this model is genuine and, in the right contexts, substantial (Jonas *et al.*., 2019).

And yet, as this paper will argue, serverless computing also represents the model that is, in structural terms, most resistant to portability. The reason is not simply that serverless functions are hard to move — the function code itself is often quite portable — but rather that the event-driven integrations, trigger mechanisms, managed service bindings, and configuration artefacts that give serverless applications their power are deeply and deliberately specific to their host platform. Porting a Lambda function to Azure Functions is not merely a matter of copying code; it involves rethinking the entire integration architecture from the ground up.

The research question motivating this paper can therefore be stated plainly: given the structural tensions between the serverless execution model and cloud portability, what can organisations and developers do to preserve meaningful workload mobility? The paper approaches this question through an analytical framework that draws on existing literature, comparative platform evaluation, and architectural pattern analysis. It does not claim to offer a definitive solution to the portability problem — that would require a degree of standardisation across the industry that does not yet exist — but it aims to provide a structured account of the challenge and a practically grounded set of strategies for managing it.

The global context for this inquiry is significant. According to Gartner (2024), worldwide end-user spending on public cloud services reached approximately USD 679 billion in 2024, and the figure is forecast to exceed USD 1 trillion by 2027. The International Data Corporation (IDC) has similarly reported that more than 90 percent of enterprises now operate in multi-cloud or hybrid-cloud configurations, with the average large enterprise consuming services from three or more distinct cloud providers (IDC, 2023). In this environment, the portability question is not a theoretical edge case; it is a practical concern that affects the strategic choices of the overwhelming majority of organisations that have committed to cloud infrastructure (Bodke & Deshpande, 2023).

The paper proceeds as follows. Section 2 establishes the theoretical and definitional foundations. Section 3 presents a structured literature review of prior research on cloud portability and serverless computing. Section 4 analyses the structural sources of serverless lock-in. Section 5 surveys available mitigation strategies. Section 6 offers a comparative evaluation of major serverless platforms. Section 7 presents illustrative analytical scenarios. Section 8 discusses future directions, and Section 9 concludes.

2. Theoretical Framework and Definitions

2.1 Defining Cloud Computing

The most widely cited authoritative definition originates with the National Institute of Standards and Technology (NIST), which characterises cloud computing as a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources — including networks, servers, storage, applications, and services — that can be rapidly provisioned and released with minimal management effort (Mell & Grance, 2011). This definition is notable for its operational focus: it foregrounds the provisioning dynamic and the relationship between consumer and provider without prescribing the underlying physical topology.

Cloud deployments are conventionally characterised along two dimensions. The first concerns service model: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). The second concerns deployment model: public, private, hybrid, and multi-cloud configurations. Each combination presents a distinct portability profile, since the level of abstraction determines how much provider-specific convention is embedded in the application architecture (Armbrust *et al.*., 2010).

2.2 Understanding Cloud Portability

Cloud portability refers to the ease with which an application, dataset, or workload can be transferred from one cloud environment to another without requiring fundamental redesign or incurring prohibitive migration costs. It subsumes several distinct concerns: application portability (can the application run on a different platform without code changes?), data portability (can datasets be extracted and reloaded across storage systems with divergent APIs?), and service portability (does the target environment offer equivalent managed services?). Impediments arise at the infrastructure, platform, application, and organisational levels.

2.3 The Serverless Execution Model

Serverless computing, and specifically the FaaS pattern, removes server management from the developer's concern entirely. Functions are invoked in response to discrete events and billed per invocation-millisecond. AWS Lambda (2014), Google Cloud Functions, Microsoft Azure Functions, and open-source alternatives such as Apache OpenWhisk and Knative all implement variations of this model. The defining commercial and technical characteristics of these platforms have been surveyed comprehensively by Castro *et al.*. (2019) and by Van Eyk *et al.*. (2017), whose findings establish the baseline against which subsequent portability analysis in this paper proceeds.

3. Literature Review

3.1 Cloud Computing: Foundations and Adoption Trajectory

The foundational scholarly account of cloud computing's economic and architectural logic was provided by Armbrust *et al.*. (2010) in their influential Berkeley view paper. The authors identified ten obstacles and opportunities for cloud computing, among which data confidentiality, data transfer bottlenecks, and vendor lock-in featured prominently. Their observation that 'the construction of very large datacenters at low-cost locations revolutionizes economics' captured the essential promise of the cloud model, while their candid treatment of lock-in anticipated concerns that have since become central to enterprise cloud strategy. The NIST definitional framework formalised by Mell and Grance (2011) provided the taxonomic vocabulary — service models, deployment models, essential characteristics — that has organised cloud discourse ever since.

Taibi, Lenarduzzi, and Pahl (2017) examined the motivations and challenges driving cloud-native architectural transitions, finding that agility and reduced operational overhead consistently ranked as primary adoption drivers, while concerns about vendor dependency and skills scarcity were the leading sources of hesitance. Their mixed-method study, drawing on interviews with industry practitioners, revealed that many organisations underestimated migration complexity at the point of initial adoption — a finding directly relevant to the portability challenge this paper addresses. Bermbach, Wittern, and Tai (2017) contributed a benchmarking methodology for evaluating cloud service quality from a client perspective, proposing metrics that include availability, latency consistency, and what they term 'service interface stability' — a construct closely related to API portability as discussed in Section 4.

The global quantitative context for cloud adoption has been documented extensively by market research organisations. Gartner (2024) reported worldwide public cloud spending of approximately USD 679 billion in 2024, representing a compound annual growth rate of approximately 18 percent over the preceding five years. IDC (2023) found that more than 90 percent of enterprises operated across multiple cloud providers, with multi-cloud configurations driven principally by considerations of cost optimisation, regulatory compliance, and risk

distribution. Bodke and Deshpande (2023) extended this analysis in the Indian enterprise context, finding that while cloud adoption rates in the subcontinent broadly tracked global patterns, the regulatory and data-localisation concerns specific to Indian markets created additional portability imperatives not present in Western-market studies.

3.2 Vendor Lock-In: Theory and Evidence

The theoretical treatment of vendor lock-in predates cloud computing, with roots in the economics of switching costs and network effects. Opara-Martins, Sahandi, and Tian (2016) provided a systematic review and classification of cloud lock-in types, distinguishing between technical lock-in (arising from API and data format dependencies), legal and contractual lock-in (arising from service agreements and jurisdiction), and human capital lock-in (arising from accumulated expertise in provider-specific tooling). Their taxonomy remains the most comprehensive framework for classifying cloud dependency and is adopted, with modifications appropriate to the serverless context, as the analytical structure for Section 4 of this paper.

Pahl, Brogi, Soldani, and Jamshidi (2019) surveyed container technologies as a mechanism for mitigating cloud lock-in, identifying the OCI container image standard as the most significant standardisation advance in the preceding decade. Their analysis established that containerisation substantially reduces runtime and dependency lock-in while leaving data and service integration dependencies unaddressed — a distinction that is particularly important in the serverless context, where container images can address function portability without resolving the deeper integration-layer dependencies.

3.3 Serverless Computing: Research Landscape

The academic literature on serverless computing has grown rapidly since the commercial introduction of FaaS platforms. Baldini *et al.*. (2017) provided an early survey of serverless trends and open problems, identifying cold start latency, limited execution duration, and statelessness constraints as primary technical challenges. Their treatment of portability was brief but prescient, noting that 'the diversity of trigger and binding mechanisms across platforms creates significant integration friction for cross-platform deployment.' Jonas *et al.*. (2019), in the widely cited Berkeley view on serverless computing, updated and extended this analysis, arguing that serverless would progressively displace server-ful computing for a broad class of workloads while acknowledging that the 'current lack of standard interfaces between cloud services' remained a significant barrier to adoption for latency-sensitive and integration-heavy applications.

Leitner, Wittern, Spillner, and Hummer (2019) conducted a mixed-method empirical study of FaaS development practice in industrial settings, finding that practitioners were highly aware of lock-in risks but frequently proceeded with provider-native implementations due to time-to-market pressure and the superior developer experience offered by tightly integrated platforms. This tension — between the strategic value of portability and the short-term productivity advantage of native integration — is a recurring theme in the practitioner literature and informs the strategic framing of Section 5.

Mohanty, Premsankar, and Di Francesco (2018) evaluated open-source serverless frameworks as portability alternatives to commercial offerings, finding that while platforms such as Apache OpenWhisk and OpenFaaS provided meaningfully greater deployment flexibility, they lagged commercial offerings in managed operations, global reach, and integration breadth. Hendrickson *et al.*. (2016) presented OpenLambda, a research platform designed to study FaaS execution semantics, and highlighted the challenge of designing function execution environments that are simultaneously efficient and portable across heterogeneous infrastructure.

3.4 Portability Standards and Emerging Solutions

The standardisation dimension of serverless portability has attracted attention from both the academic community and industry consortia. The Cloud Native Computing Foundation (CNCF) Serverless Working Group published the CloudEvents specification (CNCF, 2019), which defines a vendor-neutral schema for event data. The specification's adoption by Google Cloud (through

Eventarc), Microsoft Azure (through Event Grid), and an expanding ecosystem of open-source tools represents a notable advance in event-layer portability, though its coverage of the full integration surface remains partial.

Bodke (2022) examined the practical implications of Infrastructure-as-Code adoption for cloud governance and portability, finding that organisations with mature IaC practices demonstrated significantly lower migration friction than those relying on console-driven configuration management. The study's finding that IaC adoption correlated with reduced lock-in depth — measured through a composite index of API dependency, tooling specificity, and documented migration effort — provides empirical support for the IaC-based portability strategies discussed in Section 5.3.

WebAssembly as a portable serverless runtime has attracted growing research interest. The WASI specification, which provides a standardised system interface for WebAssembly modules, has been proposed as a basis for cross-provider function portability by multiple research groups, and Cloudflare's commercial deployment of Wasm-based Workers demonstrates that the model is viable at production scale. Zimmermann (2017) identified service interface granularity as a key determinant of microservice and function portability, arguing that the discipline of designing small, well-bounded service interfaces is itself a portability strategy, independent of the tooling choices made at deployment time.

3.5 Gap Analysis

The existing literature, taken together, establishes the problem of cloud portability clearly and provides a solid foundation of conceptual frameworks, empirical findings, and standardisation proposals. What it has not done — and what this paper aims to contribute — is a unified analytical treatment that situates serverless computing specifically within the portability problem space, maps the structural sources of serverless lock-in comprehensively across event-source, runtime, binding, and performance dimensions, and evaluates the available mitigation strategies against a common framework. The paper addresses this gap through the analysis presented in Sections 4 through 8.

4. Cloud Portability: A Structural Analysis

4.1 Dimensions of Vendor Lock-In

Following the taxonomy proposed by Opara-Martins *et al.*. (2016), cloud vendor lock-in can be understood across three principal dimensions: technical, contractual, and human capital. In the serverless context, technical lock-in predominates and manifests through API coupling, event-source dependency, proprietary runtime behaviour, and service binding configurations. Each of these is examined in turn below.

API lock-in arises when application code interacts with cloud services through provider-specific SDKs and endpoint semantics. An AWS Lambda function that writes to S3, reads from DynamoDB, and publishes to SNS is encoded with assumptions about those APIs' semantics, error conventions, and data formats that have no direct portable equivalent. Replacing the underlying services on a different provider requires refactoring that can substantially exceed the complexity of the function logic itself.

Data gravity, as documented by Bermbach *et al.*. (2017), creates a second dimension of lock-in through the economics of egress fees and the operational risk of migration windows. Once a substantial volume of data resides within a provider's storage infrastructure, migration to a competing environment involves financial costs that accumulate with dataset scale, temporal costs associated with the migration window, and risk costs arising from the possibility of data loss or corruption during transit.

4.2 Portability Metrics

Assessing portability requires a multi-dimensional rubric. A composite portability index might evaluate: migration complexity (the number and nature of modifications required to move a workload); migration time (the calendar time needed, including testing and validation); migration cost (engineering labour, data transfer, and parallel operation expenditure); and operational parity (the degree to which the migrated application's performance matches that of the original deployment). Organisations that evaluate their portability only when migration is needed are, by definition, assessing it under adverse conditions.

5. Serverless Cloud and the Portability Challenge

5.1 Sources of Serverless Lock-In

Serverless architectures introduce a distinctive and, in important respects, more acute form of portability challenge than traditional cloud deployments. Baldini *et al.*. (2017) noted early that 'the diversity of trigger and binding mechanisms across platforms creates significant integration friction,' and subsequent platform proliferation has deepened rather than resolved this fragmentation.

The most fundamental source of lock-in is event-source integration. In practice, the power of a FaaS platform derives not from its function execution capability in isolation but from its deep integration with the provider's event ecosystem. An AWS Lambda function might be triggered by an S3 object upload, an SNS notification, an API Gateway request, a DynamoDB stream, or an EventBridge rule. These integrations are declared through provider-specific configuration and have no portable equivalent. Moving such a function to Google Cloud Functions requires redesigning the entire event-routing architecture from scratch.

A second source of lock-in is the function runtime and execution environment. While major providers support popular runtimes — Node.js, Python, Java, Go — the precise versions supported, cold-start behaviour, environment variables available, file-system access model, and concurrency handling semantics differ meaningfully across platforms. Code that depends on undocumented or provider-specific runtime behaviour may fail after migration even if the function logic is superficially portable.

5.2 The Stateless Advantage and Its Limits

Serverless functions' stateless execution model offers a theoretical portability advantage: since function instances do not retain in-memory state between invocations, there is no session state to migrate. However, stateless functions typically interact with stateful backend services — databases, caches, object stores — that are usually managed offerings from the same provider. The function's portability is therefore only as complete as the portability of its entire dependency graph.

5.3 Cold Start Behaviour and Performance Portability

Performance characteristics vary considerably across serverless providers and represent an underappreciated dimension of portability. Wen *et al.*. (2021) demonstrated empirically that cold start penalties — the latency incurred when initialising a new function execution environment — differ substantially between providers in terms of magnitude, frequency, and the factors that influence it. Mitigation strategies, including provisioned concurrency (AWS Lambda), minimum instance settings (Google Cloud Run), and always-warm options on various platforms, are themselves provider-specific, creating a situation in which addressing the cold start problem on one platform generates configuration artefacts that do not translate elsewhere.

6. Strategies for Achieving Serverless Portability

6.1 Abstraction Layers and Frameworks

The most widely adopted strategy for improving serverless portability is the use of abstraction frameworks that interpose a provider-neutral layer between application logic and platform-specific APIs. The Serverless Framework enables developers to define functions, events, and resources in a YAML configuration file that the framework translates into provider-specific deployment artefacts, supporting AWS, Azure, Google Cloud, and several additional providers. Terraform by HashiCorp allows infrastructure to be declared in a unified configuration language and provisioned across multiple cloud providers through provider-specific plugins, providing a consistent operational model that reduces the tooling-dependency dimension of lock-in.

6.2 Container-Based Serverless and Open Standards

The containerisation of function workloads represents perhaps the most structurally compelling approach to serverless portability. By packaging functions and their dependencies as OCI-compliant container images, organisations can decouple function execution from provider-specific runtimes. Pahl *et al.*. (2019) established that containerisation substantially reduces runtime and dependency lock-

in, and major providers have converged on supporting container images as a deployment artefact: AWS Lambda supports container images up to 10 GB, Google Cloud Run executes containerised workloads with a serverless operational model, and Azure Container Apps offers a similar abstraction. An organisation that packages serverless workloads as container images retains the option to run those workloads on self-hosted Kubernetes with Knative, on any major provider's container serverless offering, or on edge compute platforms.

The CNCF CloudEvents specification (CNCF, 2019) addresses event-source integration by defining a vendor-neutral schema for event data. CloudEvents adoption reduces, though does not eliminate, coupling between serverless applications and provider-specific event sources, and its uptake by Google Eventarc and Azure Event Grid represents meaningful progress toward event-layer interoperability.

6.3 Infrastructure-as-Code and Declarative Configuration

Infrastructure-as-Code practices contribute to portability by making provider dependencies explicit, version-controlled, and reproducible. Bodke (2022) found empirically that organisations with mature IaC practices demonstrated significantly lower migration friction than those relying on console-driven configuration management. Declarative configuration also facilitates incremental portability strategies, in which individual functions or service boundaries are migrated piece by piece rather than through a wholesale platform migration, substantially reducing migration risk.

6.4 Hexagonal Architecture and Domain Isolation

The hexagonal architecture pattern — ports and adapters — isolates application core domain logic from external dependencies by interposing adapter interfaces. Applied to serverless computing, this means that a function handler should be a thin adapter that translates provider-specific event structures into domain-neutral data structures and delegates processing to a core service layer ignorant of its cloud context. Zimmermann (2017) identified interface granularity as a key determinant of service portability, lending theoretical support to this design discipline.

7. Comparative Analysis of Major Serverless Platforms

7.1 AWS Lambda

AWS Lambda remains the dominant commercial FaaS platform by market adoption and feature breadth. Its portability profile is characterised by deep integration with the AWS service ecosystem. Lambda supports container image deployments, partially mitigating the runtime portability challenge, and the Serverless Application Model (SAM) provides a structured deployment framework. Lambda-native applications typically have the deepest event-source dependencies and therefore face the most comprehensive re-architecture effort upon migration.

7.2 Google Cloud Functions and Cloud Run

Google's serverless portfolio spans Cloud Functions and Cloud Run. Cloud Run's container-native approach makes it inherently more portable than Cloud Functions, and its compatibility with the Knative serving specification means workloads can, in principle, be moved to self-hosted Knative environments. Google's Eventarc platform provides a CloudEvents-compatible event routing layer, representing a notable commitment to open-standard event formats.

7.3 Azure Functions

Azure Functions offers a binding model that abstracts I/O integration through declarative configuration. While this model improves developer productivity, the bindings are Azure-specific and do not translate to other providers. Azure's Durable Functions extension addresses stateful workflow orchestration, providing constructs for orchestrations, activities, and entities that are not available in equivalent form elsewhere — representing a significant portability liability for applications that rely on it heavily.

7.4 Open-Source and Self-Hosted Alternatives

The open-source serverless ecosystem offers alternatives that prioritise portability over managed-service convenience. Apache OpenWhisk, which underpins IBM Cloud Functions, is deployable on any Kubernetes cluster. Knative provides serverless serving and eventing primitives on top of Kubernetes. Mohanty *et al.*. (2018) evaluated these platforms and found that while they provide meaningfully greater deployment flexibility, they lag commercial offerings in managed operations, global scalability, and integration breadth — a trade-off that organisations must weigh against their specific portability requirements.

8. Illustrative Scenarios and Analytical Discussion

To ground the foregoing analysis in practical terms, two representative scenarios illuminate the portability challenge from contrasting angles.

Consider first a digital media company that built its content-processing pipeline as a collection of AWS Lambda functions triggered by S3 object uploads, with results persisted to DynamoDB and notifications dispatched through SNS. The application was developed rapidly and makes extensive use of native AWS SDK calls within function bodies. When the company's leadership considers migrating to Google Cloud, the assessment reveals that while the function code runs standard Python, the event-source architecture requires complete reimplementation on Google's event infrastructure, and the DynamoDB data model has no direct equivalent in Google's managed offerings. The migration is technically feasible but constitutes a near-ground-up re-architecture of the integration layer.

A contrasting scenario involves a financial technology startup that adopted a containerised serverless architecture from the outset. All workloads are packaged as OCI container images deployed on Google Cloud Run, with event routing handled through CloudEvents-compliant messages. IaC configurations are written in Terraform, and business logic is isolated behind port interfaces following the hexagonal architecture pattern. When a potential acquisition requires migrating to an Azure infrastructure, the engineering team redeploys container images to Azure Container Apps with configuration changes to the IaC layer and adapter re-implementations for the messaging integration — a manageable effort within a planned sprint cycle.

These scenarios illustrate a fundamental insight: portability is not primarily a migration problem; it is an architectural property that must be designed in. The cost of building in portability through modest abstraction effort during initial development is almost always less than the cost of remediation when migration is required under operational or commercial pressure.

9. Future Directions in Serverless Portability

9.1 WebAssembly as a Portable Runtime

WebAssembly (Wasm), originally a compilation target for web browsers, has emerged as a compelling runtime for serverless workloads owing to its near-native performance, language agnosticism, and strong security isolation properties. The WebAssembly System Interface (WASI) extends the Wasm execution model beyond the browser, providing a standardised interface between Wasm modules and host environments. Platforms such as Cloudflare Workers and Fastly Compute have adopted Wasm as a first-class deployment target. If Wasm runtimes achieve broad adoption across major cloud providers, they could constitute the most significant advance in serverless portability yet, providing a portable artefact alternative to the container image for function workloads.

9.2 Multi-Cloud Orchestration Platforms

A second trajectory involves the maturation of multi-cloud orchestration platforms. Independent management layers including Crossplane and Terraform CDK represent an emerging category that abstracts cloud resource management toward genuine provider neutrality. The challenge for multi-cloud orchestration in the serverless context is that abstraction must operate at a sufficiently high semantic level to span meaningful differences between provider offerings, not merely at the configuration-syntax level.

9.3 Standardisation Efforts

The CNCF Serverless Working Group and associated projects represent the most organised community effort to address serverless standardisation. In addition to CloudEvents (CNCF, 2019), the working group has contributed to open function runtimes and stateful serverless specifications. Historical precedents — the OCI container image standard being a particularly instructive example — suggest that standardisation is possible when it serves the collective interest of the ecosystem. The extent to which open standards achieve adoption by major commercial providers will substantially determine the pace at which the portability landscape improves.

10. Conclusion

This paper has examined cloud computing portability through the analytical lens of serverless architectures, arguing that the FaaS execution model, while offering substantial operational and economic advantages, introduces a distinct and in some respects more acute form of vendor dependency than traditional cloud deployments. The analysis traced the sources of serverless lock-in to event-source integration, provider-specific runtimes, and proprietary binding and trigger mechanisms, and demonstrated that these dependencies cannot be resolved by attention to function code alone but require comprehensive architectural consideration.

The strategies reviewed — abstraction frameworks, container-based deployment, open-standard event formats, Infrastructure-as-Code practices, and hexagonal application design — collectively constitute a toolkit for managing the portability dimension of serverless architecture decisions. No single strategy is sufficient in isolation; portability is best treated as an architectural quality attribute that requires deliberate investment across multiple layers of the application stack.

What this analysis ultimately underscores is a broader truth about the relationship between technology choice and organisational agency. Cloud computing was adopted, in large part, to give organisations greater flexibility and responsiveness. Portability is the mechanism by which that flexibility is preserved over time, against the natural centripetal forces of accumulated dependency. Organisations that treat portability as a first-class architectural concern — not a migration plan of last resort — are those that retain the strategic options that motivated their cloud investments in the first place.

References

1. Bodke G & Samadhan Bundhe “Vendor Lock-In and Cloud Portability in Serverless Environments: An Empirical Investigation with Special Reference to the Indian IT Industry”. Journal of Informatics Education and Research Vol 6Issue 1(2026)
2. Bodke, G. (2022). Infrastructure-as-Code adoption and cloud governance: Empirical evidence from enterprise deployments. *International Journal of Cloud Computing and Services Science, 11*(3), 142–158.
3. Bodke, G., & Deshpande, R. (2023). Multi-cloud strategy and portability imperatives in Indian enterprise IT: A survey study**.** *Journal of Information Technology and Management, 15*(2), 87–105.
4. Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., & Zaharia, M. (2010). A view of cloud computing. Communications of the ACM, 53(4), 50–58. https://doi.org/10.1145/1721654.1721672
5. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., & Suter, P. (2017). Serverless computing: Current trends and open problems. In Research Advances in Cloud Computing (pp. 1–20). Springer.
6. Bermbach, D., Wittern, E., & Tai, S. (2017). Cloud service benchmarking: Measuring quality of cloud services from a client perspective. Springer.
7. Castro, P., Ishakian, V., Muthusamy, V., & Slominski, A. (2019). The rise of serverless computing. Communications of the ACM, 62(12), 44–54. https://doi.org/10.1145/3368454
8. Cloud Native Computing Foundation (CNCF). (2019). CloudEvents specification version 1.0. https://cloudevents.io
9. Gartner. (2024). Worldwide end-user spending on public cloud services forecast, 2022–2027. Gartner Research.
10. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-Dusseau, A. C., &

Arpaci-Dusseau, R. H. (2016). Serverless computation with OpenLambda. In Proceedings of the 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '16). USENIX Association.
11. International Data Corporation (IDC). (2023). IDC futurescape: Worldwide cloud 2023 predictions. IDC.
12. Jonas, E., Schleier-Smith, J., Sreekanti, V., Tsai, C. C., Khandelwal, A., Pu, Q., Shankar, S., Carreira, J., Krauth, K., Yadwadkar, N., Gonzalez, J. E., Popa, R. A., Stoica, I., & Patterson, D. A. (2019). Cloud programming simplified: A Berkeley view on serverless computing. arXiv preprint arXiv:1902.03383.
13. Kritikos, K., Skrzypek, P., & Papazoglou, M. (2019). A survey on serverless frameworks and platforms. In Proceedings of the 12th IEEE International Conference on Cloud Computing (CLOUD 2019). IEEE.
14. Leitner, P., Wittern, E., Spillner, J., & Hummer, W. (2019). A mixed-method empirical study of function-as-a-service software development in industrial practice. Journal of Systems and Software, 149, 340–359. https://doi.org/10.1016/j.jss.2018.12.013
15. Mell, P., & Grance, T. (2011). The NIST definition of cloud computing (NIST Special Publication 800-145). National Institute of Standards and Technology. https://doi.org/10.6028/NIST.SP.800-145
16. Mohanty, S. K., Premsankar, G., & Di Francesco, M. (2018). An evaluation of open source serverless computing frameworks. In Proceedings of IEEE CloudCom 2018 (pp. 115–120). IEEE.
17. Opara-Martins, J., Sahandi, R., & Tian, F. (2016). Critical analysis of vendor lock-in and its impact on cloud computing migration: A business perspective. Journal of Cloud Computing, 5(1), Article 4. https://doi.org/10.1186/s13677-016-0054-z
18. Pahl, C., Brogi, A., Soldani, J., & Jamshidi, P. (2019). Cloud container technologies: A state-of-the-art review. IEEE Transactions on Cloud Computing, 7(3), 677–692. https://doi.org/10.1109/TCC.2017.2702586
19. Sbarski, P., & Kroonenburg, S. (2017). Serverless architectures on AWS. Manning Publications.
20. Spillner, J. (2020). Practical reference architectures for serverless applications. In Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2020). IEEE.
21. Taibi, D., Lenarduzzi, V., & Pahl, C. (2017). Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. IEEE Cloud Computing, 4(5), 22–32. https://doi.org/10.1109/MCC.2017.4250931
22. Van Eyk, E., Toader, L., Talluri, S., Versluis, L., Uta, A., & Iosup, A. (2017). Serverless is more: From PaaS to present cloud computing. IEEE Internet Computing, 22(5), 8–17. https://doi.org/10.1109/MIC.2018.053681358
23. Wen, J., Liu, Y., Chen, Z., Ma, Y., & Hu, H. (2021). Understanding the performance of serverless computing functions on cloud platforms. IEEE Transactions on Services Computing, 14(6), 1815–1829. https://doi.org/10.1109/TSC.2021.3055892
24. Zimmermann, O. (2017). Microservices tenets: Agile approach to service development and deployment. Computer Science — Research and Development, 32(3–4), 301–310. https://doi.org/10.1007/s00450-016-0337-0