A Comprehensive Study of Knative-Based Serverless Architecture for Cloud-Native Application Delivery

Ziye Zang

Nanjing University of Information Science and Technology
2543650304@nuist.edu.cn

Abstract. With the continuous evolution of cloud computing, application deployment models have gradually shifted from traditional virtual machines and microservice architectures toward more abstract serverless computing paradigms. Serverless computing aims to shield developers from low-level infrastructure management by enabling on-demand execution and elastic resource scaling, thereby significantly reducing operational complexity. Although commercial Function-as-a-Service (FaaS) platforms have accelerated the adoption of serverless technologies, they often suffer from limitations such as vendor lock-in, limited portability, and reduced system transparency.

Knative is an open-source project hosted by the Cloud Native Computing Foundation (CNCF) that provides standardized serverless capabilities on top of Kubernetes. This paper presents a comprehensive study of Knative-based serverless architecture from architectural, experimental, and practical perspectives. By analyzing the core design principles of Knative Serving and Knative Eventing and conducting experimental evaluations on autoscaling behavior, this study assesses the effectiveness of Knative in cloud-native application delivery. The advantages, limitations, and future research directions of Knative-based serverless platforms are also discussed.

Keywords: Cloud Computing · Serverless Computing · Knative · Kubernetes · Cloud-Native Architecture

1 Introduction

Cloud computing infrastructures and application architectures have experienced significant transformations over the past decade. The evolution from Infrastructure-as-a-Service (IaaS) to container-based and microservice-oriented architectures has greatly improved resource utilization and deployment flexibility. Nevertheless, even in containerized environments, developers are still required to manage service deployment, capacity planning, autoscaling policies, and runtime operations, which introduces considerable complexity.

Serverless computing has emerged as a promising paradigm to further abstract infrastructure management. By delegating server provisioning, scaling decisions, and resource scheduling to the platform, serverless architectures allow developers to focus primarily on application logic. This paradigm has proven particularly effective in scenarios such as web APIs, event-driven data processing, and automation tasks.

Despite these advantages, most existing serverless platforms are tightly coupled with specific cloud providers. Such designs often lead to vendor lock-in, limited portability across environments, and reduced visibility into system internals. These issues become especially problematic in private cloud, hybrid cloud, and edge computing scenarios.

Knative addresses these challenges by introducing a Kubernetes-native serverless framework based on open standards. Rather than replacing Kubernetes, Knative extends it with higher-level abstractions to support request-driven and event-driven workloads. This paper investigates the design, implementation, and experimental behavior of Knative-based serverless architectures, with a particular focus on autoscaling and resource efficiency.

The main contributions of this paper are as follows:

- A systematic analysis of Knative architecture and its core components;
- An experimental evaluation of Knative Serving autoscaling behavior;
- A discussion of practical use cases, limitations, and future research directions.

2 Related Work and Background

2.1 Cloud-Native Computing

Cloud-native computing emphasizes containerization, microservices, and automated operations to build scalable and resilient distributed systems. Kubernetes has become the de facto standard for container orchestration, providing essential features such as service discovery, load balancing, and resource scheduling.

However, Kubernetes primarily operates at the infrastructure level and requires developers to understand a wide range of low-level concepts, including Pods, Deployments, Services, and autoscaling mechanisms. This complexity often increases the operational burden on development teams.

2.2 Serverless Computing Models

Serverless computing enables applications to execute in response to events or requests, with resources allocated dynamically based on actual demand. Function-as-a-Service (FaaS) platforms exemplify this model by eliminating the need for explicit server management and enabling fine-grained billing.

Nevertheless, existing FaaS platforms typically rely on proprietary runtimes and APIs, which hinders application portability and flexibility. Applications developed for one platform often require significant modifications to migrate to another environment.

Table 1. Comparison between traditional microservice architecture and Knative-based serverless architecture

Aspect	**Traditional Microservices**	**Knative Serverless**
Deployment Model	Always-on services	On-demand services
Scaling Mechanism	Manual or HPA-based	Automatic, scale-to-zero
Resource Utilization	Fixed resource allocation	Dynamic, request-driven
Operational Complexity	Medium	Higher (additional abstractions)
Portability	Platform-dependent	High (Kubernetes-native)
Use Case Focus	Long-running services	Event-driven and bursty workloads

2.3 Knative in the CNCF Ecosystem

Knative is a key serverless project within the CNCF ecosystem. Its primary goal is to provide a consistent and extensible serverless abstraction on Kubernetes, allowing applications to run across diverse infrastructures. By integrating with existing cloud-native tools such as Istio and Prometheus, Knative leverages the strengths of the Kubernetes ecosystem while avoiding closed platform designs.

From a broader perspective, Knative reflects an emerging trend toward standardized and Kubernetes-native serverless platforms. By relying on open interfaces and cloud-native primitives, Knative enables serverless workloads to run consistently across public clouds, private data centers, and hybrid cloud environments. This focus on portability and standardization distinguishes Knative from many proprietary serverless offerings and aligns with the CNCF vision of building interoperable cloud-native ecosystems.

3 System Architecture of Knative

3.1 Overall Architecture

Knative is a Kubernetes-native serverless framework composed of two major subsystems: Knative Serving and Knative Eventing. Both subsystems are implemented using Kubernetes Custom Resource Definitions (CRDs) and corresponding controllers, which allows Knative to extend Kubernetes without modifying its core components. This design enables seamless integration with existing Kubernetes clusters and ensures compatibility with the broader cloud-native ecosystem.

In the overall layered architecture, Kubernetes is responsible for low-level resource management tasks, including pod scheduling, container lifecycle management, and node-level resource allocation. Knative operates as an abstraction layer above Kubernetes, providing higher-level serverless capabilities such as request routing, automatic scaling, and event delivery. By building on native Kubernetes primitives, Knative inherits Kubernetes' reliability and scalability while offering a more developer-friendly programming and deployment model.

From a design perspective, Knative introduces a trade-off between abstraction and system complexity. By providing higher-level serverless abstractions,

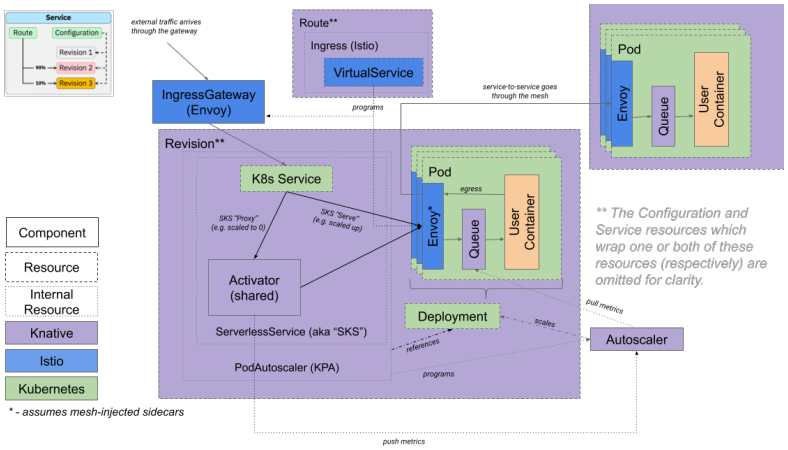

Fig. 1. Overall architecture of Knative-based serverless platform

Knative significantly simplifies application deployment, scaling, and traffic management. At the same time, this abstraction layer increases the number of control-plane components involved in the system, such as autoscalers, activators, and eventing infrastructure. As a result, operating a Knative-based platform may require additional operational effort compared to plain Kubernetes deployments. Nevertheless, this trade-off is often acceptable in scenarios where development efficiency and elastic resource utilization are prioritized.

3.2 Knative Serving

Knative Serving is designed to support request-driven serverless workloads. It introduces a set of high-level abstractions, including *Service*, *Revision*, *Route*, and *Configuration*, which collectively manage the lifecycle of a serverless application. Developers define the container image and runtime configuration for a service, while Knative automatically handles deployment, versioning, and traffic routing.

Each *Revision* in Knative Serving represents an immutable snapshot of the application code and configuration. This immutability enables reliable traffic management strategies, such as canary releases and blue-green deployments, by allowing traffic to be split across multiple revisions. The *Route* component dynamically directs incoming requests to the appropriate revisions according to user-defined policies.

A key feature of Knative Serving is its autoscaling mechanism, which adjusts the number of active service replicas based on request concurrency or request rate. When incoming traffic increases, Knative rapidly scales up the number of pods to meet demand. Conversely, when traffic decreases, replicas are gradually scaled down. Furthermore, Knative supports scale-to-zero, allowing services to

release all computing resources during idle periods, thereby improving overall resource efficiency.

3.3 Knative Eventing

Knative Eventing is designed to support event-driven architectures by decoupling event producers from event consumers. It introduces a set of components, including *Sources*, *Brokers*, *Triggers*, and *Channels*, which together form a flexible and extensible event processing pipeline.

Event sources generate events from various systems, such as message queues, object storage services, or custom applications. These events are delivered to Brokers, which act as centralized event routers. Triggers define filtering rules that determine how events are dispatched to specific consumers. Knative Eventing adopts the CloudEvents specification as a standardized event format, ensuring consistent event representation across heterogeneous systems.

By abstracting event routing and delivery, Knative Eventing enables developers to build loosely coupled and highly scalable event-driven applications. This design promotes interoperability, simplifies integration of diverse event sources, and supports dynamic scaling of event consumers based on workload characteristics.

4 Experimental Setup and Performance Evaluation

4.1 Experimental Environment

The experimental environment is built on a local Kubernetes cluster deployed using Minikube, which provides a lightweight and reproducible platform for evaluating cloud-native applications. This setup simulates a small-scale cloud environment and allows controlled experimentation without relying on external cloud resources.

Knative Serving and Knative Eventing are installed on top of the Kubernetes cluster to provide serverless capabilities for request-driven and event-driven workloads, respectively. In addition, Istio is integrated as the service mesh to manage ingress traffic, service-to-service communication, and basic observability features. This configuration reflects a typical cloud-native stack in which serverless frameworks coexist with service mesh technologies.

Additional tools are deployed for request generation and system observation in order to simulate varying workload conditions. These tools enable the generation of bursty traffic patterns and support real-time monitoring of system behavior, which is essential for analyzing the dynamic characteristics of serverless platforms.

4.2 Experimental Procedure

To evaluate the behavior of Knative Serving under different workload conditions, a sample web service implemented in Go is deployed as a Knative service. The

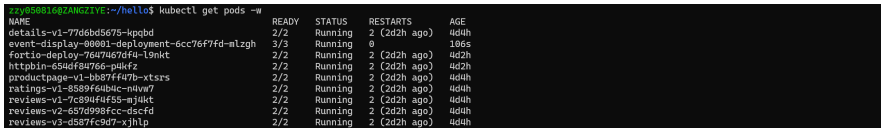

Fig. 2. Real-time monitoring of pod status using `kubectl get pods -w`

Table 2. Summary of experimental observations in Knative Serving

Observation Aspect	Experimental Findings
Scaling trigger	Service replicas scale based on incoming request load
Scale-up behavior	New pods are created rapidly under increased traffic
Scale-down behavior	Replicas are gradually reduced as traffic decreases
Scale-to-zero	Service scales down to zero during idle periods
Cold-start effect	Initial request experiences additional latency
Resource utilization	Efficient usage during low or idle workload

application exposes a simple HTTP endpoint and serves as a representative request-driven workload.

During the experiment, request traffic is generated to emulate different load patterns, including periods of increasing traffic, sustained load, and idle intervals. These patterns are designed to trigger autoscaling events and to observe how Knative responds to workload fluctuations.

System behavior is monitored in real time using native Kubernetes commands. In particular, the `kubectl get pods -w` command is used to continuously observe pod status changes during the experiment. This approach allows direct observation of pod creation and termination events as the request load changes over time, providing concrete evidence of the autoscaling process.

4.3 Results and Analysis

This experiment focuses on qualitative observation of autoscaling behavior instead of precise latency or throughput benchmarking. The experimental results demonstrate that Knative dynamically adjusts the number of service replicas according to incoming request load. When traffic increases, new pods are rapidly created to handle concurrent requests, ensuring that the service remains responsive under higher demand.

As the workload decreases, the system gradually scales down the number of replicas. When the service becomes idle, Knative further reduces the number of active pods and eventually scales the service down to zero. This scale-to-zero behavior highlights one of the key advantages of serverless architectures, namely efficient resource utilization during idle periods.

Although cold-start latency is observed during the initial handling of requests after a period of inactivity, the overall system behavior aligns well with the design objectives of serverless computing. In particular, the trade- off between

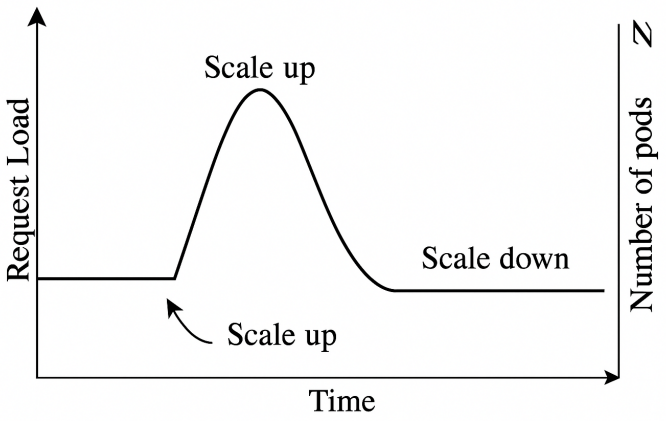

Fig. 3. Illustration of autoscaling behavior in Knative Serving

resource efficiency and response latency becomes evident. While the scale-to-zero mechanism significantly reduces resource consumption, it introduces additional latency for the first request when a service is reactivated.

These results further suggest that workload patterns play a crucial role in determining the suitability of serverless platforms. For bursty or event-driven workloads, Knative's autoscaling mechanism provides substantial benefits. However, for latency-sensitive applications with continuous traffic, careful configuration or hybrid deployment strategies may be required.

5 Use Cases

Knative-based serverless architecture is applicable to a wide range of cloud-native scenarios. In event-driven data processing pipelines, Knative Eventing enables loose coupling between components. For API backends, Knative Serving simplifies deployment and scaling while supporting traffic management strategies such as canary releases. In hybrid and edge computing environments, Knative provides consistent serverless capabilities across heterogeneous infrastructures. For example, in an event-driven data processing scenario, Knative Eventing can be used to trigger processing functions upon file uploads or message arrivals. Such a design enables elastic scaling based on event frequency while avoiding the need to maintain always-on processing services.

6 Limitations and Challenges

The limitations discussed in this paper do not necessarily indicate inherent weaknesses of Knative itself. Instead, they largely reflect the fundamental characteristics of serverless computing and the intrinsic complexity of cloud-native systems built on Kubernetes.

One of the most notable challenges is cold-start latency, which occurs when a service scales from zero replicas to handle incoming requests. While the scale-to-zero mechanism significantly improves resource efficiency during idle periods, it may introduce additional response latency for the first request. This trade-off is common across serverless platforms and requires careful consideration in latency-sensitive applications.

Another challenge lies in the operational complexity of Knative deployments. Although Knative simplifies application-level concerns such as deployment and scaling, it introduces additional control plane components, including autoscalers, activators, and eventing infrastructure. As a result, operating a Knative-based platform still demands a solid understanding of Kubernetes internals, networking, and observability tools.

Furthermore, debugging and monitoring event-driven and highly dynamic serverless applications can be more difficult compared to traditional long-running services. The ephemeral nature of serverless workloads, combined with frequent scaling events, makes it challenging to trace execution paths and diagnose performance issues without comprehensive logging and monitoring support.

Despite these challenges, they should be viewed as design trade-offs rather than critical drawbacks. With appropriate workload selection, architectural design, and supporting toolchains, Knative can effectively serve as a powerful serverless layer for cloud-native application delivery.

7 Conclusion

This paper presented a comprehensive study of Knative-based serverless architecture for cloud-native application delivery. By examining the design principles and core components of Knative Serving and Knative Eventing, this study analyzed how serverless abstractions can be effectively built on top of Kubernetes without sacrificing openness or portability. Through experimental evaluation, the autoscaling behavior of Knative Serving was observed and analyzed, demonstrating its ability to dynamically adjust resource usage in response to workload fluctuations and to achieve efficient scale-to-zero behavior during idle periods.

Based on the architectural analysis and experimental results, this study shows that Knative provides a flexible and extensible serverless solution suitable for a wide range of cloud-native scenarios. In particular, Knative is well suited for request-driven microservices, event-driven data processing pipelines, and hybrid or edge computing environments where portability and consistency across infrastructures are critical requirements.

For practical applications, Knative can be adopted as a serverless layer for Kubernetes-based platforms to simplify application deployment and scaling while retaining compatibility with existing cloud-native ecosystems. However, practitioners should carefully consider operational complexity and cold-start latency when deploying Knative in production environments, and complement it with appropriate observability and monitoring tools.

Future research may focus on optimizing cold-start performance, improving developer experience through enhanced tooling and abstractions, and exploring deeper integration with emerging cloud-native technologies such as service meshes, event streaming platforms, and edge computing frameworks. Further experimental studies under large-scale and real-world workloads would also provide valuable insights into the performance and reliability of Knative-based serverless systems.

Acknowledgments

The author would like to thank the instructors and classmates for their valuable feedback and discussions.

References

1. Cloud Native Computing Foundation: CNCF Landscape. `https://landscape.cncf.io`
2. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, Omega, and Kubernetes. Commun. ACM (2016)
3. Knative Documentation. `https://knative.dev`
4. Baldini, I. et al.: Serverless Computing: Current Trends and Open Problems. Research Advances in Cloud Computing (2017)
5. CloudEvents Specification. `https://cloudevents.io`