Melding the Serverless Control Plane with the Conventional Cluster Manager for Speed and Compatibility

Leonid Kondrashov
NTU
Singapore

Lazar Cvetković
ETH Zurich
Switzerland

Hancheng Wang
Nanjing University
China

Boxi Zhou
NTU
Singapore

Dhairya Rungta
NTU
Singapore

Dmitrii Ustiugov
NTU
Singapore

Abstract

Modern serverless applications, often interactive with highly volatile traffic, challenge system scalability, demanding control planes that deliver low latency and cost efficiency. Analysis of production traces and existing systems reveals that current control plane designs (synchronous and asynchronous), particularly when built on conventional cluster managers like Kubernetes, struggle with this balance, often wasting significant CPU and memory resources on creating underutilized or idle instances. While clean-slate approaches like Dirigent offer performance gains, they sacrifice compatibility with established cluster management ecosystems.

We introduce *PulseNet*, a serverless system designed to achieve high performance and low cost while maintaining compatibility with conventional cluster managers. PulseNet employs a novel dual-track control plane. A standard asynchronous track manages long-lived, full-featured Regular Instances for handling predictable, sustainable traffic, preserving full compatibility and feature sets off the critical path. Concurrently, an expedited parallel track addresses excessive traffic bursts that trigger cold starts. This fast path utilizes node-local agents (Pulselet) to rapidly spawn short-lived Emergency Instances with a reduced feature set, critically bypassing the latency overhead of the main cluster manager.

Our experiments demonstrate that PulseNet, while remaining compatible with conventional managers for >98% invocation traffic, achieves 35% faster end-to-end performance at a comparable cost to the incompatible Dirigent system. PulseNet outperforms Kubernetes-compatible systems with synchronous control planes by 1.5-3.5× at 8-70% lower cost, and surpasses asynchronous counterparts by 1.7-3.5× at 3-65% lower cost.

1 Introduction

Serverless computing, often realized through Function-as-a-Service (FaaS), enables developers to deploy applications without managing the underlying cloud infrastructure. Cloud providers dynamically provision and scale resources in response to events, such as HTTP requests or database updates, offering elasticity and a pay-per-use model [7, 8, 11, 29–32], scaling function instances following the changes in their invocation traffic. Modern serverless applications, which are often interactive, exhibit highly volatile traffic patterns, stress the scalability of these systems, demanding designs that deliver both fast response times and resource efficiency.

Central to serverless platforms is the control plane, which manages the lifecycle of function instances. Systems with synchronous control planes (e.g., AWS Lambda [7], OpenWhisk [24]) respond immediately to invocations that require new instances. If no idle instance is available, the invocation path is blocked until a new instance is created, which minimizes latency for that specific invocation but may degrade overall performance and efficiency under a high instance creation load [28, 48]. Systems with asynchronous control planes (e.g., Knative [12, 14], Google Cloud Run [11]) handle scaling decisions off the critical path. Their autoscaler component monitors metrics and triggers instance creation when needed, making incoming function invocations wait if all instances are busy. Asynchronous scaling improves cluster resource usage efficiency but often introduces significant instance creation and control-plane decision-making and queuing delays, typically 1-3 seconds, before new instances can process incoming invocations, substantially degrading user experience [38, 44].

Our analysis of production traces and control plane overheads in Knative-based systems with synchronous and asynchronous control planes reveals that both approaches struggle when built atop conventional cluster managers, such as Kubernetes [15]. These managers, while feature-rich, introduce substantial instance creation overhead due to the active multi-round interaction with the conventional cluster manager, e.g., when allocating cluster resources and IP addresses, and setting up network routes. High function-instance churn exacerbates due to the volatility of serverless application traffic, which further degrades system performance. We find that systems with a synchronous control plane face performance bottlenecks during bursts of instance creations, while asynchronous systems suffer long queuing delays during cold starts. Mitigating cold starts via keepalive (pre-warmed idle instances) leads to significant resource waste, with idle instances consuming up to 70 and 87% of memory in systems with synchronous and asynchronous control planes. The state-of-the-art clean-slate approach, called Dirigent [28], offers performance gains but sacrifices compatibility with established cluster management ecosystems and lacks general-purpose features, e.g., hindering co-location with other services and adoption of standard tooling like service meshes.

We introduce *PulseNet*,[1] a serverless system designed to achieve high performance and low cost while maintaining compatibility with conventional cluster managers, such as Kubernetes. *PulseNet* employs a novel dual-track control plane. A standard asynchronous track comprises the conventional cluster manager that manages long-lived, full-featured *Regular Instances* for handling predictable, sustainable traffic. This operates off the critical path, preserving full compatibility and feature sets. Concurrently, an expedited parallel track addresses excessive traffic bursts that trigger instance creations. This fast path utilizes node-local agents (Pulselets) to rapidly spawn short-lived *Emergency Instances* with a reduced feature set – supporting only the features required by FaaS deployments – removing the overhead of the conventional cluster manager. Emergency Instances occupy only 10% of the cluster's CPU and memory resources, and are torn down after processing a single invocation. This dual approach minimizes instance-creation and other control-plane delays and reduces the cluster resource waste while integrating seamlessly with the existing infrastructure.

We prototype PulseNet in vHive [48] as a seamless extension of the Knative control plane running atop unchanged Kubernetes. Evaluation results show that *PulseNet* provides 35% performance improvement while maintaining cost parity with Dirigent. Additionally, when compared with other Kubernetes-compatible alternatives, *PulseNet* delivers 1.5-3.5× better performance than synchronous systems while reducing costs by 8-70%, and achieves 1.7-3.5× better performance than asynchronous systems while reducing costs by 3-65%. Furthermore, compared with Knative-based systems featuring linear-regression predictor and the state-of-the-art NHITS prediction model [26, 33], *PulseNet* delivers up to 4× better performance while reducing costs by 35-40%.

Our main contributions are:

- We identify two kinds of invocation traffic, sustainable and excessive, showing that the former utilizes >98% of the cluster resources while the latter stresses the control plane but consumes <2% of the cluster resources.
- This work is the first to comprehensively characterize the end-to-end and internal delays within systems that feature synchronous and asynchronous control planes, analyzing their performance and cluster resource usage (as a proxy for cost). We reveal that the existing systems waste 9-20% of the cluster CPU cycles and 70-87% of its memory capacity.
- We propose a novel *dual-track* control plane architecture as a seamless extension of the *production-grade* Knative control plane running on *unchanged* Kubernetes – and fully compatible with its features and optimizations for >98% invocation traffic – showing the feasibility and robustness of the approach. Our experiments show that PulseNet substantially outperforms not only Kubernetes-compatible systems but also Dirigent, their incompatible high-performance alternative, while reducing the cluster resource usage.

[1]We will release the code, traces, and toolchain after publication.

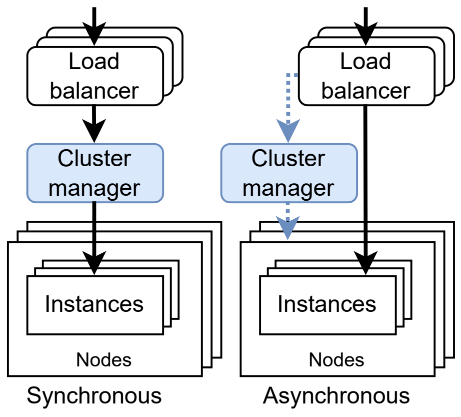


Figure 1. High-level overview of serverless architecture. The cluster manager is on the path only for cold invocations.

2 Background

Serverless systems architectures consist of load balancers, a cluster manager, and worker nodes that run function instances. Load balancers and function instances constitute the data plane. Serverless function invocations arrive from the application users to the scale-out load balancers, which route invocations to function instances to execute these invocations. In scenarios where a function has no non-busy instances to process the invocation, the cluster manager creates extra function instances on worker nodes.

Figure 1 shows two instance scaling approaches used by state-of-the-art commercial [7, 14] and open-source systems [12, 17, 24], the key difference between which is whether instance creations occur synchronously or asynchronously. For the synchronous case, the cluster manager issues an instance creation command on the critical path of an invocation. In this case, an invocation is early-bound to the sandbox that will be created. In the latter case, the cluster manager runs a separate periodic autoscaling process. This process gathers scaling metrics from load balancers, aggregates them, and issues instance creation commands outside of the critical path of an invocation, while the users' requests wait in the queue inside a load balancer.

2.1 Control Plane Architecture

The synchronous approach, employed by state-of-the-art systems such as AWS Lambda [7] and some open-source systems, e.g., OpenWhisk [24], allows for an immediate scaling reaction as soon as an invocation arrives. Other systems, including Knative [12], Google Cloud Run [11], and FunctionGraph [38], use the asynchronous approach associated with a slower reaction time: these systems typically aggregate function invocation statistics over a period of time and only start scaling when they can confirm the change in the invocation traffic. Hence, in the worst case, an invocation can wait for the entire autoscaling period (Knative's default is 2s [13]) before the cluster manager requests an instance to be created. Moreover, the asynchronous approach exhibits higher queuing delays in the control plane [28], often resulting in a high tail latency [38]. For example, in the scenario with a single instance serving function invocations, all invocations are queued into the single existing instance until the control plane creates more instances.

2.2 Today's FaaS Systems

Most of today's open-source serverless systems such as Knative [12], OpenFaaS [17], OpenWhisk [24], Fission [10] are built on top of Kubernetes [15], a conventional general-purpose cluster manager widely used for managing microservices. These systems utilize the abstractions and features Kubernetes exposes (e.g., declarative scaling abstractions, placement service, cluster state storage). However, as the prior work [28] shows, Kubernetes features come with a high cost, especially for instance creation delays (sometimes referred to a cold-start delays). Concretely, the control plane overhead surges on concurrent function instance creations. This problem is especially relevant for workloads with high instance churn, as is the case for serverless workload [28, 44].

To ameliorate the performance limitations of today's cluster managers, real systems overprovision the number of instances and keep them alive for a prolonged (keepalive) period to avoid instance creation (cold-start) delays for future invocations of the same functions [44, 46]. Instance overprovisioning leads to increased memory capacity occupied by idle instances, which is a key factor in operational costs in serverless systems, e.g., in AWS Lambda, memory accounts for up to 40% of serverless deployment operational costs [21, 37]. Despite extensive prior research efforts in intelligent instance provisioning using predictor models [33, 43–45], these predictive approaches have limited applicability: they are only compatible with the asynchronous control plane-based designs. Systems with synchronous control planes (usage of which is more widespread in real systems, e.g., AWS Lambda), spawn new instances upon detecting any change in the number of in-flight invocations, leaving no time for predictor inference. Moreover, predictive approaches may impose high inference overheads, consuming a substantial fraction of cluster resources (§6.3.2), which prior works often overlook.

System	React. time	CM Perf.	Predict. Comp.	Conv. CM Comp.	Resour. Waste
AWS Lambda (sync)	✓	✗	✗	✓*	High
OpenWhisk (sync)	✓	✗	✗	✓	High
Knative, GCR (async)	✗	✗	✓	✓	Moder.
Dirigent	✓	✓	✓	✗	Low
PulseNet	✓	✓	✓	✓	Low

Table 1. Comparison of the existing approaches and the proposed PulseNet in reaction time (React. time), cluster manager performance (CM Perf.), compatibility with predicting models (Predict. Comp.), compatibility with conventional cluster managers (Conv. CM Comp.), and Resource Waste.
**AWS Controllers for Kubernetes integrates with Lambda [23].*

2.3 Dirigent, a Clean-Slate Cluster Manager for Serverless Clouds

An alternative approach to improving cluster manager performance is to design the cluster manager from scratch, which Dirigent [28] employs. However, such an approach, while yielding multiple orders of magnitude performance gains, breaks compatibility with the existing infrastructure, lacks the generality Kubernetes has, and affects the set of workloads the system can run. For example, Dirigent lacks support for persistent volumes, hence it is unable to run stateful workloads in the same logical cluster. The latter impedes performance optimizations with co-locating serverless workloads and conventional services (*e.g.,* databases, cloud storage like AWS S3 [4], AWS ElastiCache [6], and Aurora DB [5]) to enable near-storage processing, which is possible if both services are controlled by the same cluster manager [22, 36]. Furthermore, Dirigent lacks compatibility with service mesh deployments and West-East traffic, and other fine-grained network configurations and policies, which can limit networking speed in the cluster [27].

We summarize approaches for building serverless systems in Table 1 and show the tradeoffs associated with each of them. Our system PulseNet, aims to get the best from the synchronous and asynchronous approaches, providing high-performance with minimal resource overprovisioning *and* remaining compatible with conventional cluster managers.

3 Characterization of the Traffic Patterns & Existing Control Planes' Performance

In this section, we study the function invocation traffic patterns occurring in production deployments (§3.1), break down the scaling delays in the existing systems with synchronous and asynchronous control planes (§3.2), instance-creation throughput of these control planes (§3.3), and resource efficiency of modern serverless deployments (§3.4). We use vHive [48] research framework to configure two setups based on Knative [12] and Kubernetes [15] for our experiments, which are widely used in commercial serverless deployments [1, 14]. The first configuration uses vanilla Knative that features an asynchronous control plane. The second configuration is a modified Knative version that features a synchronous control plane, with an autoscaling policy similar to AWS Lambda [20, 21, 46]. §5 provides more details about these configurations and the evaluation methodology that uses sampled production traces [41, 47].

3.1 Invocation Traffic Patterns

We start by identifying the requirements for effectively handling serverless traffic and explore how different traffic patterns impose different requirements for the system. We simulate a serverless system with a synchronous control plane with a keepalive period of 10 minutes when replaying an hour from the entire production function invocation trace released by Azure Functions [41, 44], which contains around 25 thousand functions with their duration and inter-arrival time (IAT) distributions. The simulator models the number of active instances as the number of function invocations in-flight and the number of idle instances at each moment in time. The modeled system can instantly spawn required function instances. We assume that each instance occupies one CPU core during invocation processing.

In our experiments, we observe that only 0.1% of invocations trigger instance creations, which can nevertheless overload the conventional control plane, as shown by prior work [28]. We reveal that instances that serve this traffic – which we refer to as *excessive* – occupy <2% of the cluster's CPU resources. In contrast, the warm invocation traffic – which we refer to as *sustainable* – utilizes >98% of the CPU.

Sustainable and excessive traffic types exhibit very different behaviors. Sustainable traffic creates no load on the conventional control plane but occupies the vast majority of the CPU resources. In contrast, excessive traffic consumes little CPU resources but the response time to such invocations depends heavily on the reaction time and speed of the control plane. Thus, an ideal system should identify both traffic types and optimize for them differently. The control plane should carefully place the instances that serve sustainable traffic and balance the load while adjusting the instance number in the background. As for the excessive invocations, the control plane should optimize for the scaling speed.

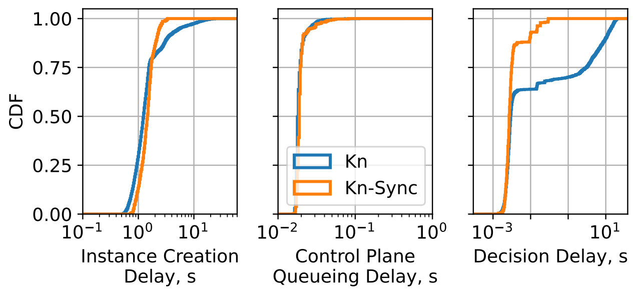


Figure 2. Cumulative distribution functions (CDFs) for the three types of delays occurring in the systems with synchronous (Kn-Sync) and asynchronous (Kn) control planes.

3.2 Analysis of the Scaling Delays Occurring in Control Planes

Next, we analyze the delays in the control plane of real-world synchronous and asynchronous Knative-based systems, which arise in these systems under realistic load because they do not tailor to the traffic characteristics we identify in the previous section. In these experiments, we use the In-Vitro [47] load generator that replays a sampled production trace containing 400 functions.

We identify and study three sources of queuing in serverless control planes: the instance creation delays, the internal control plane queuing delays, and the decision delays. We plot cumulative distribution functions (CDFs) for each source in Figure 2 for vanilla Knative (Kn) and its modified synchronous analogue (Kn-Sync). The figure shows that the sources can cause queuing delays in the range from milliseconds to 10s of seconds, which is comparable to the execution duration time of the invocations [44]. Hence, if any of those delays happen on the critical path of invocation handling, they would be seen as noticeable delays to end-to-end request latency. We discuss each delay source in detail below.

3.2.1 Instance Creation Delay. We consider the instance creation delay as the time the cluster manager takes to create the instance after receiving the command to increase the number of replicas. We find that the node-side delays are responsible for 1-3s of the delay. The instance creation delays arise on the target worker host primarily due to the comprehensive set of features that the existing conventional cluster managers support for long-running services. Specifically, it includes the allocation of an IP address, integration of this IP into a cluster-level network overlay, and creation of several system entities: pod sandbox, user container, and sidecar container. All these actions also require multi-trip interactions with Cluster Manager that saves the corresponding bookkeeping in its persistent database.

3.2.2 Control Plane Queuing Delay. We measure the latency of communication between the front service of the Knative control plane, called Knative Autoscaler, and the

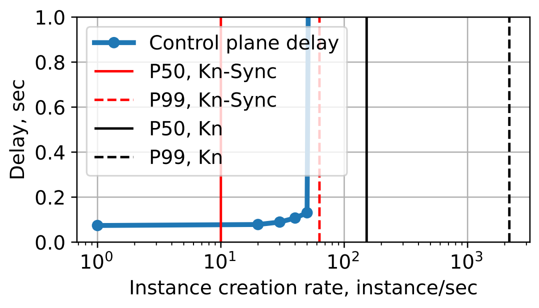


Figure 3. The delays occurring in the Knative control plane under various instance-creation rates, measured with a microbenchmark. The red and black lines show the instance creation 50-th and 99-th percentile rates in systems with a simulated synchronous (Kn-Sync) and asynchronous (Kn) control planes, respectively, when replaying invocations from a sampled production trace.

persistent Cluster Manager's database behind the Kubernetes API Server. Although these delays' median is below 20ms, we observe that sporadic bursts of instance-creation requests from Load Balancer can cause delays up to 140ms.

3.2.3 Decision-Making Delay. We measure the decision-making latency as a delay between the moment Knative Autoscaler can observe an increase in concurrency (i.e., invocations in-flight) and the moment it decides to increase the number of instances. We find that the asynchronous and synchronous control plane make 65% and 85% of decisions, respectively, under 10ms. The control plane can make decisions so quickly when they create the first active instance of a function after its period of inactivity. However, for the rest of the decisions, the control planes can take up to 20 seconds, forming a long tail, particularly profound for the asynchronous control plane. We find that adjusting the number of instances, i.e., scaling not from zero, often leads to such delayed decisions, as the control plane takes time to confirm the changing trend in the traffic by averaging the arrival rate over a time window (1 minute by default).

To summarize, the instance creation delays are the biggest contributor to overall control plane delays, even in the median latency, prompting a redesign of the control plane components on the critical path of instance creations. The other two components exhibit median time under 20ms but require careful tail-latency optimizations. Overall, an ideal system can achieve end-to-end scaling time under 200ms if it combines fast instance creation, quick decision making as in the synchronous control planes, and can avoid congestion inside the control plane components.

3.3 Instance Creation Throughput of a Conventional Control Plane

Next, we evaluate if the conventional control plane's throughput is sufficient to serve the instance creation requests, which Load Balancer initiates in systems with a synchronous and asynchronous control planes.

We evaluate the performance of the control plane under high instance-creation load to see whether conventional control planes are suitable for the high instance creation rate common in FaaS deployments [44]. In this experiment, we use a microbenchmark to trigger instance creations inside the Knative-Kubernetes control plane, which we have carefully tuned as described in §5, to identify the throughput limits of a conventional control plane. We run this benchmark atop a KWOK [16] cluster, which features the real-world system's control plane but emulates Worker Nodes.

Figure 3 shows that the control plane can sustain a stable control plane throughput of 50 cold starts per second, improving from two cold starts per second as reported by prior work [28].[2] We compare the control plane throughput to the instance creation rates estimated using the methodology from §3.1 while replaying the same production trace. We find that Knative control plane's throughput is sufficient to serve the median instance creation rate occurring in a system with a synchronous control plane (red solid line) but is 3× lower than the median instance creation rate occurring in a system with an asynchronous control plane (black solid line).[3] This result indicates that in a large-scale system, internal control plane queuing can be more significant than our results measured with a sampled trace (§3.2.2). Moreover, rare bursts of instance creations can easily overwhelm any control plane, as indicated by the 99-th percentiles (dashed lines), requiring 1.2× and 40× higher throughput from synchronous and asynchronous control planes, respectively.

3.4 Cluster Resource Efficiency Analysis

Here, we analyze the usage of memory and CPU by busy and idle instances, as well as the control plane components, which are major components of the operational cost of a serverless deployment [21]. We use the same two Knative-based systems with synchronous and asynchronous control planes, and the same setup as in the previous section §3.2. We observe that idle instances occupy 70% and 87% of the overall memory footprint attributed to function instances in systems with synchronous and asynchronous control planes, respectively. Using a synchronous control plane leads to less memory efficiency and higher cost. We measure substantial CPU overheads induced by the control planes: control plane components use 20% and 9% of CPU cycles in the systems

[2]This is however not a contradiction: careful configuration and tuning (§5) has substantially increased Knative control plane's throughput.
[3]This observation corroborates the need for a high-performance control plane in the systems that feature an asynchronous control plane [28].

with asynchronous and synchronous control planes, respectively. These numbers are attributed to the control planes' activity for creating, managing, and tearing down instances.

The above high CPU and memory overheads motivate an efficient control-plane design that should not only provision instances on time but also minimize the instance creation rate to avoid wasting cluster resources on instances with a low reuse probability, bloating the cluster's operational cost.

3.5 Takeaways

Two types of traffic. We categorize function invocation traffic into two categories: sustainable and excessive. Serving sustainable traffic takes the majority (>98%) of the cluster resource but has no load on the control plane (§3.1). In contrast, excessive traffic can sporadically strain the control plane by generating a high volume of instance creation requests but requires little (<2%) of the cluster resources for processing. A well-designed system needs to tailor to both types of traffic.

Scaling delays in control planes primarily stem from the process of instance creation, which is often complex due to the extensive feature sets supported by the conventional cluster managers, such as Kubernetes. Supporting a reduced feature set when spawning instances to serve sudden spikes of (excessive) traffic can reduce the instance creation time.

Cluster resource efficiency needs improvement. Existing systems tend to frequently create and tear down instances, which leads to severe CPU waste. They also tend to keep idle instances alive for prolonged periods of time, taking up substantial cluster memory capacity. Instead, an efficient control plane should spawn fewer instances and reduce the keepalive time to improve resource efficiency and, thus, the overall operational cost.

4 Design

4.1 Key Ideas

Based on the insights from §3, we devise three design requirements for an efficient control plane for serverless clouds.

First, the system needs to efficiently serve both the *sustainable* and *excessive* fractions of the function invocation traffic (§3.1). This suggests a control plane with two tracks, each of which handles instance creation for different traffic classes. Hence, to create instances for sustainable traffic, the system can employ the asynchronous vanilla track that monitors the changes in the function-invocation request concurrency and adds and removes function instances accordingly. At the same time, the system can use the second, expedited track to quickly launch Emergency Instances while bypassing the conventional cluster manager. These instances serve the excessive traffic caused by sudden peaks in the invocation traffic, but do not result in long-lasting instance overprovisioning.

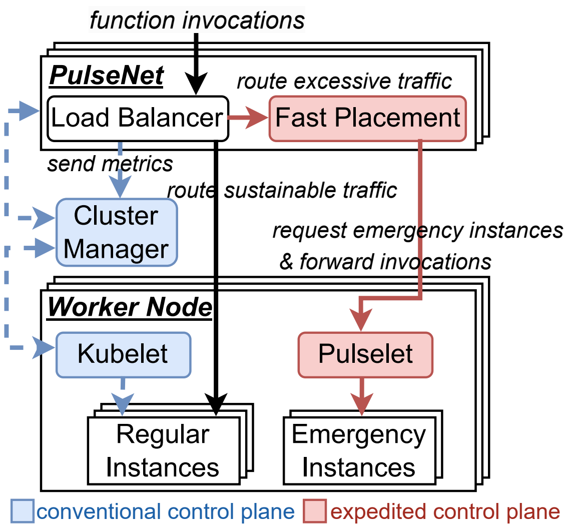


Figure 4. PulseNet architecture overview. The design combines the added expedited control plane with the conventional control plane.

Second, an efficient control plane must start instance scaling immediately – without waiting for the conventional cluster manager to comprehensively evaluate the cluster status, trading off placement quality over reaction time. We borrow this idea from speculative execution, e.g., in an out-of-order CPU where it can execute instructions before their dependencies are resolved. We prioritize scaling decision speed over careful load balancing since excessive traffic is responsible for a small fraction of the worker node utilization.

Third, to enable fast Emergency Instance creation, we limit the feature set Emergency Instances support. Namely, we support OCI (Docker) image deployment, outbound connections, logging, CPU and memory quotas, and system call filtering, while we sacrifice features required for long-running and stateful services, including Emergency Instance resources registrations with the cluster manager. Importantly, Cluster Manager offers the complete feature set to Regular Instances, which serve *all* warm invocations (99.9% of the traffic). Hence, Cluster Manager can still intelligently allocate >95% of the cluster resources among the Regular Instances, which also benefit from the rich feature set discussed in §2.3.

4.2 PulseNet Architecture Overview

Function invocations arrive at *PulseNet* that comprises two components: *Load Balancer* and *Fast Placement*. Similarly to the existing designs (§2), Load Balancer identifies the target function and finds a Node with an existing non-busy instance of that function to route the invocation for processing. As in the systems with a vanilla asynchronous control plane, Cluster Manager continuously monitors and predicts the trends in the function invocation traffic, adjusting the number of instances for the functions that experience up or down trends in the invocation traffic. We refer to these instances as *Regular Instances*. Importantly, Regular Instances creations are off the function invocations critical path.

When routing a function invocation, if the Load Balancer finds no idle Regular Instances available, it marks the invocation as excessive traffic and routes it to the *Fast Placement* component. Fast Placement requests an *Emergency Instance* creation on a Node, where PulseNet's agent, *Pulselet*, spawns an Emergency Instance to process the excessive traffic. Pulselet is a fast per-node alternative of *Kubelet*, which is a Kubernetes per-node agent for spawning function instances.

4.3 PulseNet Load Balancer and Fast Placement

PulseNet's Load Balancer monitors concurrency (i.e., the number of requests in-flight or in the per-instance queues) and routes function invocations to the available Regular Instances[4], similarly to Knative's Activator [3]. In PulseNet, Regular Instance creation is *never* on the critical path of an invocation, allowing the provider to deliver a rich feature set to the deployed functions – without concerns about the additional delays it brings. In contrast, existing systems with synchronous or asynchronous control planes suffer from the lengthy instance creation delays [7, 11, 12, 28, 38].

While keeping Regular Instances busy, Load Balancer requests Fast Placement to create Emergency Instances for the excessive traffic. Fast Placement forwards the Emergency Instance creation requests to Pulselets on the Worker Nodes in a Round-Robin fashion. Since Emergency Instances only handle the excessive traffic, the system does not need to keep them alive for future invocations and tears each instance down once it completes processing its assigned invocation. If Emergency Instance or Pulselet fails, Fast Placement receives an error or a time-out notification and can retry running the failed invocations or expose the error to the application.

Load Balancer collects the Emergency Instance usage metrics from Fast Placement to monitor invocation traffic changes and passes these metrics down to Cluster Manager. However, to avoid unnecessary fluctuations in the Regular Instance count, for excessive traffic, Load Balancer uses a heuristic to distinguish the long-term traffic trend changes from sporadic traffic changes. Hence, Emergency Instances handle infrequent and sporadic invocations, which do not require spawning long-living Regular Instances. This mechanism allows the system to defer or even skip the Regular Instances instantiation. However, if sporadic invocations hint at a change in the traffic trends, Load Balancer informs Cluster Manager, which adjusts the number of Regular Instances according to the historical data collected over a time window, as in Knative [2], or predicted concurrency [33].

[4]We consider an instance available if its per-instance queue is not full.

4.4 Emergency Instances and Compatibility with the Conventional Cluster Managers

To minimize the startup delay, PulseNet adopts three techniques. First, Pulselet maintains a pool of pre-created TUP/TAP network devices with pre-initialized IP addresses. Second, Pulselet uses VM snapshotting to quickly start instances, while caching and leveraging information about which snapshot exists on which nodes. Finally, Pulselet supports the subset of the regular Kubelet functionalities, which are required for FaaS deployment operation. Namely, Cluster Manager is unaware of Emergency Instances and does not maintain any state associated with them (e.g., IP address, CPU and memory resource quotas, readiness probes). This reduces the interaction between Cluster Manager, its internal cluster state database (e.g., `etcd` in Kubernetes), and Worker Nodes. Instead, Pulselet independently creates new instances, assigns them an IP address, CPU and memory quotas, and notifies Load Balancer of instance existence.

Emergency Instances can open outbound connections to communicate with other services inside the cluster (e.g., cloud storage like Amazon S3), and on the Internet. Serverless functions in the existing commercial systems are also typically deployed behind a NAT [49, 50], similarly to PulseNet Emergency Instances. Pulselet also drops support for network-attached storage (e.g., K8s volumes), since serverless functions are stateless and communicate through cloud storage and caching services [6, 35, 40]. In contrast to Emergency Instances, Regular Instances can use the full set of features offered by the conventional cluster manager, which can enable better performance for processing in Regular Instances (e.g., service mesh support for West-East traffic when accessing conventional services within the same cluster [27]) but might require slower bootstrapping. For example, providers can decide to avoid using snapshotting for Regular Instances to reduce the security attack service, e.g., random generator state replication across instances [25].

4.5 Implementation

4.5.1 PulseNet Load Balancer and Fast Placement. We implement PulseNet prototype in vHive [48], an open-source framework widely used for serverless systems research in academia and industry. vHive deploys Knative [12] production FaaS framework, used in commercial serverless offerings [14], atop the Kubernetes [15] cluster manager. We modify several Knative components, namely Activator and Autoscaler, leaving Kubernetes completely unchanged. We extend Knative Activator to divide the traffic into sustainable and excessive, sending the latter immediately to the Fast Placement component. In contrast, vanilla Activator buffers these invocations and asks Knative Autoscaler to create more instances. Instead, Fast Placement requests Emergency Instances to be created through its expedited path, without requesting Regular Instances from the Cluster Manager. We augment Activator to request new Regular Instances from Cluster Manager only if they satisfy the filtering criteria.

4.5.2 Metrics Filtering Heuristic. We implement a simple heuristic to reduce the control plane's CPU overhead of creating Regular Instances and the memory overhead of keeping idle instances alive when the probability of repeated invocations is low. When the function invocation arrives and the Load Balancer routes it to an Emergency Instance for processing, the Load Balancer then assesses the probability of future invocations arriving within the keepalive period before including this invocation in the metrics it sends to Cluster Manager. Cluster Manager can then take these metrics into account in addition to the warm invocation traffic when adjusting the number of Regular Instances in its regular cycle. Load Balancer compares PulseNet's keepalive period to the filtering threshold chosen from the function's inter-arrival time (IAT) distribution (e.g., median IAT), which it collects over the preceding hour. The PulseNet Load Balancer only includes the invocation in the metrics for the Cluster Manager if the PulseNet keepalive period is larger than the function's filtering threshold. Effectively, the creation of only some Emergency Instances leads to the creation of Regular Instances, thus minimizing the cluster resource usage. PulseNet's keepalive and filtering threshold are configurable parameters, which we empirically evaluate in §6.1.

4.5.3 Pulselet and Emergency Instances. We implement Pulselet in Golang, deploying it alongside the regular kubelet. The vanilla Kubernetes cluster manager deploys function instances as containers, which we have decided to keep to realistically evaluate the overheads imposed by the conventional control plane. Although in commercial systems, providers tend to use MicroVMs for function deployment, we find the type of deployment irrelevant for our prototype because the focus of this work is on the conventional control plane overheads, which need to be realistic. Implementing support for a complete set of features supported by a system like AWS Lambda or Kubernetes for such evaluation is impractical in a research setting and might not be representative of a real system.

We isolate Emergency Instances in Firecracker MicroVMs used in AWS Lambda [21], adopting their snapshotting technology for rapid bootstrapping. Hence, the speed of Emergency Instance spawning is representative of the state-of-the-art production systems.

5 Methodology

Hardware setup. We run PulseNet and other serverless systems on an 8-node c220g5 Cloudlab cluster [9]. Each node has two Intel Xeon Silver 4114 CPUs @ 2.20GHz, each with 10 physical cores, 192 GB DRAM, and an Intel SSD.

Real Control-Plane Deployment with an Emulated Large-Scale Cluster. For some experiments in §3 and §6, we use KWOK [16] v0.6.1 to simulate large numbers of worker nodes while retaining a real Knative-Kubernetes control plane, enabling high-scale evaluation of our serverless system without the prohibitive cost of provisioning physical clusters. KWOK supports modeling node and pod behavior to stress-test control plane performance and experiment with scenarios, such as high-rate sandbox creation by the control plane (§3.3) or configurable instance creation time (§6.2.3), which is difficult to observe with real hardware outside at a non-production scale. This approach uses real control plane components with all their interactions, as in a large-scale production deployment, while modeling the load on worker nodes. This makes large-scale and configurable experiments (§6.4.2) feasible in a small-scale research setting.

Software setup. We use vHive [48], i.e., Knative v1.13 running on top of Kubernetes v1.29. Regular Instances run atop containerd v1.6.18, while Emergency Instances execute in AWS Firecracker VMs v1.10.1. We limit each Knative instance's concurrency to 1 (i.e., maximum per-instance queue depth), similarly to AWS Lambda [20]. We assume all container images and VM snapshots are cached in memory on each node, similarly to prior work [28]. In all Knative-based configurations to ensure measurement stability, we disable SMT and fix the CPU frequency to the base frequency, disable the Knative panic mode, and set the Load Balancer's (called Activator in Knative) replica count to 1 but ensure that it never becomes a bottleneck.[5]

Workload. We use In-Vitro [47] methodology for representative trace sampling and load generation. In all experiments, we use a 400-function trace sample with per-function IAT and duration distributions, chosen to apply the maximum possible load to the cluster without reaching 100% CPU utilization at any point throughout the experiment. Only in §6.4.2, we use a bigger trace with 2000 functions to evaluate the performance of a larger, 50-node cluster. We run experiments for an hour, discarding the first 20 minutes as a warm-up. Similarly to the prior work [28, 47], we use a synthetic spin-loop function with a programmed duration taken from the trace.

Baselines. We compare PulseNet to five state-of-the-art serverless systems. We use four baselines based on Knative [12], which is a K8s-based serverless system widely used in commercial offerings [14]. First is vanilla Knative (**Kn**) with an asynchronous control plane, which uses its default concurrency-based autoscaling policy with the default 60-second autoscaling window. We carefully configure the Knative deployment to maximize the baseline control plane's performance: we increase concurrency and request

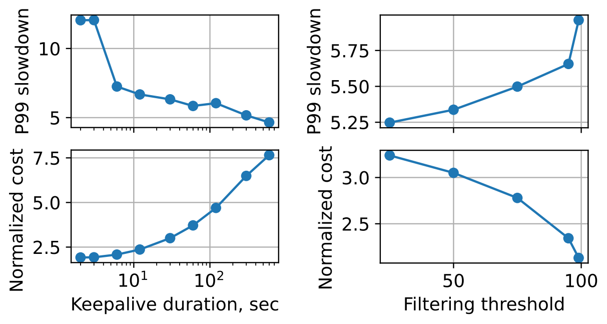
[5]During the peak utilization in our experiments, we observe that Knative Activator uses less than one CPU core with a 99-th percentile <10ms routing delay.

rate limits to the Kubernetes API Server in Knative components and Kubernetes Controller Manager, and also increase the CPU and memory quotas for Knative core components. We deploy five Kubernetes API Server replicas and use the default 60-second autoscaling window, but disable panic mode to stabilize the autoscaling. The second one is Knative-Synchronous (**Kn-Sync**), the Knative version we implement with synchronous instance creation similar to AWS Lambda. Specifically, we have modified Autoscaler to trigger new instance creations when it cannot find an available instance for a new request and retain the instances for a fixed period of inactivity. We use a 10-minute keep-alive period, which is estimated to be the keep-alive duration for AWS Lambda [46]. We also use Dirigent [28], a clean-slate serverless cluster manager with a high-performance asynchronous control plane, which is the state-of-the-art academic system. Finally, (**Kn-NHITS**) and (**Kn-LR**) that use NHITS [26] and lightweight linear regression prediction models (demonstrated as the most accurate predictors for FaaS by the prior work [33]), respectively, replacing the default Knative autoscaling policy. We train the models on the one-hour-long part of the trace that precedes the part used in the evaluation.

Performance and Cost Metrics. We are using several metrics to evaluate the systems. Our main characteristics of the systems are function invocation performance and the costs that the serverless system incurs to provide that level of performance. To evaluate performance, we use the geometric mean of the tail (99-th percentiles) of per-function slowdown. Lower slowdown is better, with the slowdown of 1 meaning that the system's end-to-end response time is as low as measured in an unloaded system.[6] To estimate the cost-efficiency, we use the total memory footprint of all instances in the cluster, normalized to the total memory footprint of non-idle instances. We call this metric *normalized cost*. We also evaluate other cost sources, such as instance creation rates and the CPU cycles used by control plane components. We derive these metrics by collecting cluster-wide metrics with Prometheus [18] and Kubernetes Metrics Server [19].

6 Evaluation

We evaluate PulseNet to answer the following questions: (1) How to set PulseNet' parameters for the best performance and lowest overhead? (§6.1) (2) How does PulseNet's performance compare with existing systems? (§6.2) (3) How does PulseNet's cluster resource utilization compare with existing systems? (§6.3) (4) How does PulseNet balance performance and cost trade-offs? (§6.4)



(a) Keepalive duration. (b) Filtering threshold.

Figure 5. Effects of keepalive duration (a) and filtering threshold (b) on PulseNet performance and cost. Measurements are collected for realistic workloads sampled from Azure Functions trace.

6.1 PulseNet Sensitivity Studies

PulseNet's performance is primarily affected by two parameters: (1) keepalive duration and (2) metric filtering threshold. This section evaluates the impact of these parameters on system performance and cost. The experimental results in this section are obtained by running realistic workloads sampled from the Azure Functions trace on a physical testbed. Furthermore, we analyze how to set these parameters.

6.1.1 Keepalive Duration. The keepalive duration determines how long a Regular Instance can remain idle before it is terminated. A smaller keepalive increases the cold start probability, degrading performance. A higher keepalive causes resource wastage, increasing cost. We sweep the keepalive duration from 2s to 600s to measure its impact on system performance and cost. As Figure 5(a) shows, when the keepalive duration reaches 60s, further increasing the keepalive duration significantly increases cost with minimal performance improvement. Thus, we set the keepalive duration to 60s.

6.1.2 Metric filtering threshold. The filtering threshold determines the confidence threshold to create new function instances. Specifically, a lower filtering threshold makes the system more likely to create new instances, potentially leading to resource over-allocation and higher costs. A higher filtering threshold makes the system more conservative in creating new instances, potentially increasing the number of cold starts and degrading performance. We increase the threshold from 25% to 99%. Figure 5(b) shows that PulseNet achieves the best balance between performance and cost when the filtering threshold is 50%, which we use below.

[6]We first calculate per-invocation slowdown by dividing the end-to-end response time by expected execution duration. Then we compute the per-function 99-th percentile of slowdown. Finally, we aggregate per-function slowdown with the geometric mean.

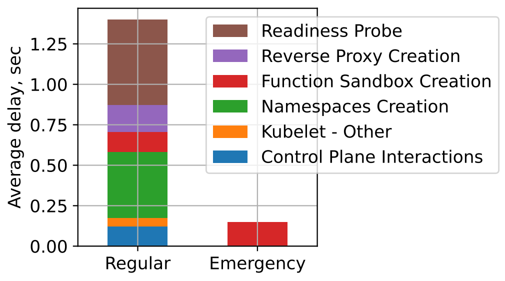


Figure 6. Breakdown of creation delays for both types of instances.

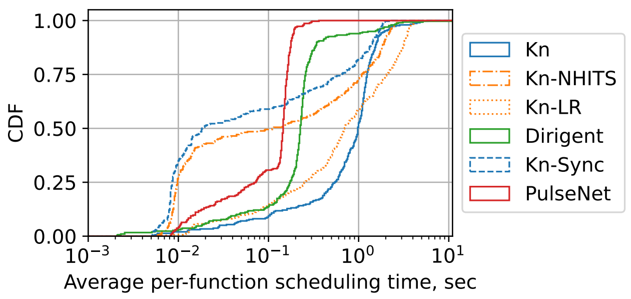


Figure 7. CDFs of average per-function scheduling delay in evaluated systems under sampled production workload.

Given the obtained results, we choose the 60-second keepalive period and the 50-th percentile as metric filtering threshold in PulseNet as our parameters for evaluation.

6.2 Control Plane Performance Analysis

In this section, we evaluate PulseNet's control plane performance from three perspectives: instance creation delays, scheduling delays, and the system's sensitivity to instance creation delays.

6.2.1 Instance Creation Delay Breakdown. We first measure and analyze the time cluster managers take to create an instance, as shown in Figure 6.

Regular Instances incur significant creation delays, primarily due to the comprehensive feature set provided by conventional cluster managers, such as Kubernetes. The largest contributors to this delay are:

- **Readiness probes** that introduce a 500ms delay, on average, largely due to the minimum 1-second polling interval set by Kubernetes.
- **Namespace and networking setup**, which requires several round-trips with the cluster manager components, accounts for over 400ms delay.
- **Reverse (Queue) Proxy** container and the function sandbox creation contribute over 250ms delay.

In our evaluation, the function handler is a Golang binary with a negligible initialization overhead. However, in other runtimes, e.g., Java, runtime initialization times can take seconds, further compounding the delay.

Emergency Instances, in contrast, can rapidly bootstrap, bypassing many of the above steps. PulseNet's node-local agent, Pulselet, supports only a minimal set of the required features, foregoing non-essential one such as readiness probes and complex networking setup. Instead, it focuses solely on restoring a MicroVM using a function snapshot, which reduces the average creation delay to approximately 150 milliseconds-about 10 times faster than Regular Instances. Pulselet also creates and maintains a pool of virtual network devices and local IP addresses, which can be quickly assigned to newly created VMs.

It is important to note that Regular Instance creation delays always occur off the critical path for request handling in PulseNet. As a result, the observed cold start latency for user requests is dominated by the much shorter Emergency Instance creation time. This design reduces cold start delays by over 8× compared to systems like Kn-Sync, where Regular Instance creation occurs on the critical path.

6.2.2 Scheduling Delays Analysis. We continue the evaluation with the scheduling delays of different serverless systems. Scheduling delay includes cold start time, data plane queuing, request routing, and load balancing. To measure the scheduling delay, we subtract the function's execution time from the inside function instance from the end-to-end invocation latency. Figure 7 shows the distribution of average scheduling delay for each system.

As shown in Figure 7, Knative and Dirigent have median delays of approximately 1s and 200ms, respectively, matching their instance creation times, and both exhibit tail latencies up to 4s for certain functions. This is because their autoscalers cannot quickly react to traffic fluctuations. Kn-NHITS and Kn-Sync, due to the high instance retention, have 40% to 50% of functions experiencing warm starts below 10ms. However, the remaining functions are delayed up to 2s due to slow instance creation. Kn-LR shows high latency, indicating poor prediction for unpredictable real-world traffic scenarios. Compared with the above baselines, PulseNet reduces critical path cold-start latency by 10× and eliminates worst-case scheduling delays. For example, PulseNet outperforms prediction-based systems (Kn-NHITS and Kn-LR) when handling unpredictable workloads. Unlike Kn-Sync, PulseNet delivers excellent responsiveness without excessive instance provisioning, especially for functions with frequent cold starts. PulseNet's improvement of scheduling delay stems from the proposed dual-path control plane design. The design's expedited path bypasses cluster manager bottlenecks by quickly creating Emergency Instances for traffic bursts.

6.2.3 Sensitivity to Various Instance Creation Delays. In this section, we evaluate the impact of instance creation

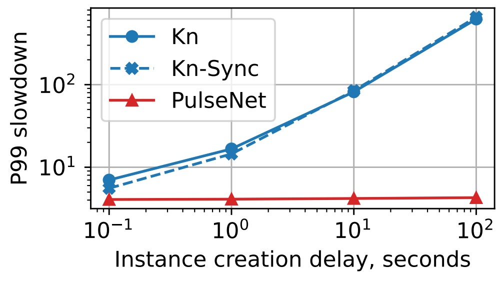
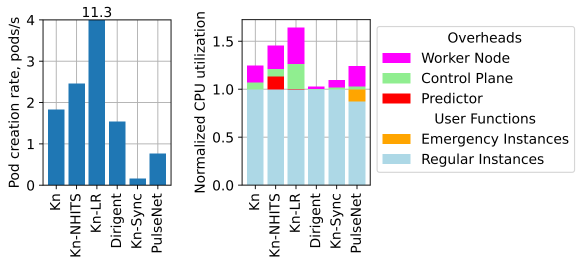


Figure 8. Slowdowns of the systems in cluster managers with different simulated instance creation delays. Measured for sampled production workload. Lower is better.



(a) Pod creation rate. (b) CPU utilization breakdown.

Figure 9. Instance creation rates observed (a) and CPU utilization breakdown (b) in the systems under sampled production workload.

delays on the performance of different serverless systems. Specifically, because instance creation delays can vary significantly across different sandbox technologies or resource conditions in serverless systems, it is important to study how instance creation delays affect serverless system performance. In this experiment, we use KWOK [16] to construct a simulation environment that can precisely control instance creation delays. We use KWOK to vary instance creation delays from 100 milliseconds (comparable to container creation) to 100 seconds (comparable to full VM boot) and measure the performance of different serverless systems under the same Azure Functions production trace.

As shown in Figure 8, as instance creation delay increases, the performance of Knative and Kn-Sync degrades significantly. The geometric mean of the 99-th percentile slowdown shows steady degradation with instance creation delay increases from 100ms to 100s. This indicates that these systems' performance heavily depends on the underlying instance creation speed. In contrast, PulseNet maintains relatively stable performance. This is because PulseNet's fast-path control plane can quickly create Emergency Instances to handle burst traffic, thus significantly alleviating the impact of high instance creation delays.

In summary, by creating Emergency Instances for excessive traffic, PulseNet effectively eliminates worst-case scheduling delays. Furthermore, based on this design, even in environments with high instance creation delays, PulseNet delivers more stable performance

6.3 Control Plane Resource Efficiency Analysis

In this section, we analyze PulseNet's control plane resource efficiency from three perspectives: instance creation rate, cluster-wide CPU utilization, and memory utilization across the system.

6.3.1 Instance Creation Rate. In this experiment, we measure the instance creation rate over time while executing the Azure Functions trace and show the results in Figure 9(a). A lower instance creation rate reduces the control plane overhead and improves system stability.

As shown in Figure 9(a), Kn-Sync has the lowest instance creation rate as it uses a long 10-minute keepalive duration, allowing most instances to be reused. Knative and Dirigent show comparable instance creation rates as they employ similar autoscaling policies to respond to current traffic. The prediction model-based Kn-NHITS and Kn-LR have the highest instance creation rates. This is because they need to adjust the number of instances based on prediction results frequently. Compared with Knative, PulseNet reduces the instance creation rate by 60%. This is because PulseNet's filtering mechanism intelligently avoids creating instances for unnecessary requests, instead handling these requests with Emergency Instances. This design can reduce the computational overhead within the control plane of the serverless system and improve system stability. We study the additional computational resources necessary to handle the instance creations in the following section.

6.3.2 Cluster-wide CPU Cycles Utilization Breakdown. We define CPU overhead as additional CPU consumption beyond the resources needed to serve the user functions, which incur costs for the service providers. CPU overhead includes all control plane components (sandbox management, traffic predictions, health and metric gathering), data plane components (load balancer, ingress), and any computation overhead produced by prediction mechanisms. We collect these CPU utilization components using Kubernetes Metrics Server [19].

As shown in Figure 9(b), PulseNet has similar overhead as Knative since it only adds a negligible extra computational overhead for Emergency Instance management. Emergency Instances account for only 10% of total CPU usage by the instances. Prediction-based systems, Kn-NHITS and Kn-LR, have the highest CPU overhead due to additional compute resources required for predictions and handling more instance creations (we exclude their training time as an overhead).

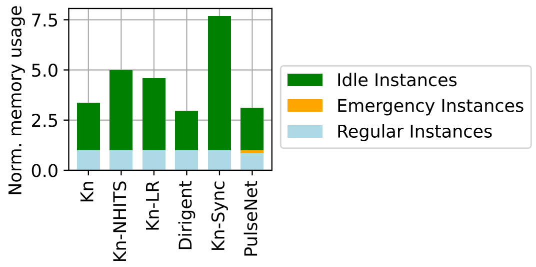


Figure 10. Normalized memory usage for different serverless systems under sampled production workload. Lower is better.

Dirigent can handle the same instance creation rate with lower overhead due to its lighter cluster management implementation. Kn-Sync, due to its longer keepalive time, reduces the frequency of instance creation, thereby decreasing CPU overhead. Although PulseNet CPU overhead is higher than that of Kn-Sync, PulseNet achieves a better balance of resource usage by trading slightly increased CPU overhead for significantly reduced memory usage. We will analyze this in depth in Section 6.3.3.

6.3.3 Memory Usage.

In this section, we evaluate the memory usage of various serverless systems by replaying the Azure Functions trace. We show our results in Figure 10. Kn-Sync exhibits the highest memory usage because it employs a long keepalive period of 10 minutes to handle requests. The prediction model-based Kn-NHITS and Kn-LR tend to provision more instances than Knative, leading to higher memory usage. We attribute this result to the prediction quality on a highly volatile part of the traffic. Dirigent, due to its fast instance creation speed, can create instances more quickly, reducing queueing and preventing subsequent overreaction by the autoscaler. PulseNet uses our proposed metric filtering technique that allows some of the requests to be handled by an Emergency Instance without creating the long-living Regular Instance. These Emergency Instances contribute 10% of total non-idle instance memory usage. Compared with Knative and Kn-Sync, PulseNet improves memory utilization by 8% and 60%, respectively. Compared with Dirigent, although PulseNet needs to create additional Emergency Instances, these instances are only created when needed to handle burst traffic and are destroyed immediately after use.

In summary, through our proposed dual-path control plane architecture and intelligent filtering mechanism, PulseNet achieves the best balance between memory and CPU resources. For example, PulseNet reduces the memory utilization of Knative and Kn-Sync by 8% and 60%, respectively, with negligible CPU overhead. This balance provides cloud providers with a cost-effective solution for serverless deployments.

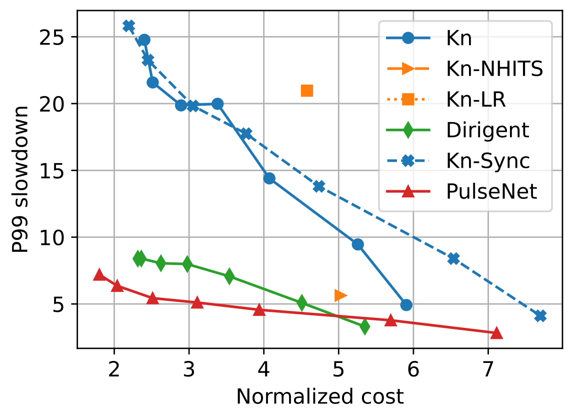


Figure 11. Performance-cost trade-off comparison for different serverless systems under sampled production workload. Lower is better for both axes.

6.4 Trade-off Space Analysis of Performance & Cost

In this section, we compare different serverless systems' performance and resource usage. We also validate these results in a large-scale cluster with emulated worker nodes.

6.4.1 Experiments with a Real System.

In this section, we evaluate the trade-off between performance and cost for different systems. We obtain the performance-cost tradeoff for different systems by varying instance retention-related parameters (*e.g.*, keepalive period, autoscaling window) from 6 seconds to 10 minutes. As shown in Figure 11, PulseNet achieves the best tradeoff between performance and cost. Compared with Knative and its variants (such as Kn-NHITS and Kn-LR), PulseNet achieves higher performance at lower resource costs. Specifically, PulseNet outperforms Kubernetes-compatible systems with synchronous control planes by 1.5-3.5× at 8-70% lower cost, and surpasses asynchronous counterparts by 1.7-3.5× at 3-65% lower cost. Also, PulseNet delivers up to 4× better performance while reducing costs by 35-40% PulseNet achieves 35% faster end-to-end performance at a comparable cost to the Dirigent system. The reason behind PulseNet's optimal tradeoff is its dual-path control plane design. This design effectively manages sustainable traffic with Regular Instances and excessive traffic with Emergency Instances, thereby achieving higher performance with fewer resource costs.

6.4.2 Large-Scale Experiments.

Then, we use the same methodology and metrics as in Section 6.4.1 to evaluate the performance-cost tradeoffs in the large-scale cluster by simulating 50 worker nodes with KWOK and running real control plane components. The experimental results show that in this large-scale cluster, PulseNet still demonstrates significant improvements compared with state-of-the-art systems. Specifically, PulseNet outperforms Kubernetes-compatible systems with both synchronous and asynchronous control planes by up to 3× at up to 3× lower cost.

This is because, as the cluster size increases, traditional systems encounter severe control plane congestion issues. Additionally, system configurations that performed well in smaller clusters can overload the control plane under larger workloads. PulseNet effectively addresses these problems through our proposed dual-path control plane architecture.

6.5 Discussion: Snapshot Caching on the Cluster Nodes

In our implementation and evaluation, we consider the VM snapshots cached on each node in the cluster. However, in real systems, such storage overheads can be prohibitive, so that the provider can restrict caching of the snapshots to a subset of nodes. To understand the snapshot caching requirements, we devise an analytical model that describes request execution on Emergency Instances when playing the full Azure Functions trace.

Our analysis shows that most functions (>95%) have only 0.1 Emergency Instances in the whole cluster, on average. For these functions, caching function snapshots on a handful of nodes is sufficient. The biggest contributor function to Emergency Instance utilization uses around 50 Emergency Instances. Given that full Azure Functions trace requires approximately 1000-node cluster to run, the Emergency Instances of that function would be distributed across 5% of all nodes. Therefore, for accurate load balancing and placement decisions, the system needs to cache its snapshots on around 10% of the cluster nodes.

In summary, we estimate that PulseNet should perform as well in systems with limited snapshot availability on the node, given that the replication and distribution can accommodate the biggest contributors to Emergency Instance usage.

7 Related Work

Control plane design. The related work has shown today's control planes suffer scheduling decision propagation delays due to internal congestion, which negatively affects cold start performance. Liu *et al.* [38] find that Kubernetes-based serverless systems (such as Knative [12], OpenWhisk [24]) experience delays from hundreds of milliseconds to seconds due to issues like asynchronous startup, state synchronization, and declarative abstraction tax. Dirigent [28] proposes a novel control plane design which significantly improves control plane throughput and latency. However, Dirigent sacrifices Kubernetes compatibility and lacks enterprise-level features Kubernetes provides by default. For example, Dirigent does not implement quality-of-service, naming and discovery services, and persistent volumes. Furthermore, Dirigent relaxes cluster state reconstruction guarantees on recovery from failures, which makes it unsuitable for workloads that depend on exact cluster state reconstruction. Our work optimizes control plane latency while maintaining full compatibility with the Kubernetes ecosystem.

Sandboxing technology. This class of work aims to reduce the sandbox creation latency by proposing new underlying execution environments. Firecracker [21] reduces sandbox startup time through lightweight virtualization technology, while not sacrificing strong isolation guarantees. gVisor [51] reduces sandbox creation time by using a lightweight user space kernel. SOCK [42] bases its solution on a lean container system. However, all these works primarily focus at the level of a single worker node. Our work addresses the control plane problems, and is orthogonal to these works since they can be integrated as an function execution environments.

Scheduling policies. While performance can be improved through system redesign, scheduling policies also matter, especially for cold starts. IceBreaker [43] uses the Fourier transforms to predict the invocation concurrency of a function. Based on prediction results, it decides whether to pre-warm the function. Joosen *et al.* [33] analyze periodic patterns in function invocations. These patterns have significant implications for optimizing resource reservation schemes in serverless platforms. Jiagu [39] reduces latency caused by prediction through decoupling prediction and scheduling. Most invocations can be quickly scheduled through a fast path without complex prediction. Hermod [34] proposes a hybrid load balancing technique that concentrates function invocations on fewer servers during low load while selecting the least-loaded servers during high load. This approach balances resource efficiency and low latency. Our work can serve as the underlying platform where any of these policies can be integrated.

8 Conclusion

PulseNet addresses the conflict between rapid scaling and resource efficiency in serverless systems on conventional cluster managers, such as Kubernetes, where prior approaches compromise latency, efficiency, or compatibility. Distinguishing between sustainable and excessive traffic, PulseNet uses a dual-track control plane. A standard asynchronous path manages compatible, long-lived, full-featured Regular Instances for sustainable traffic. An expedited path uses node-local Pulselet to rapidly launch lightweight Emergency Instances for excessive traffic, bypassing manager overhead for low latency. PulseNet provides superior speed and resource efficiency while maintaining full compatibility with conventional managers for over 98% of invocation traffic.

References

[1] 2020. CNCF Survey. Available at https://www.cncf.io/wp-content/uploads/2020/11/CNCF_Survey_Report_2020.pdf.

[2] 2025. About autoscaling - Knative. Available at https://knative.dev/docs/serving/autoscaling/.

[3] 2025. About load balancing - Knative. Available at https://knative.dev/docs/serving/load-balancing/.

[4] 2025. Amazon S3. Available at https://aws.amazon.com/s3/.

[5] 2025. Aurora DB. Available at https://aws.amazon.com/rds/aurora/.

[6] 2025. AWS ElastiCache. Available at https://aws.amazon.com/elasticache/.

[7] 2025. AWS Lambda. Available at https://aws.amazon.com/.

[8] 2025. Azure Functions. Available at https://azure.microsoft.com/en-us/products/functions.

[9] 2025. CloudLab. Available at https://www.cloudlab.us/.

[10] 2025. Fission: Open Source, Kubernetes-Native Serverless Framework. Available at https://fission.io.

[11] 2025. Google Cloud Run. Available at https://cloud.google.com/run.

[12] 2025. Knative. Available at https://knative.dev/.

[13] 2025. Knative Autoscaling Configuration. Available at https://github.com/knative/serving/blob/main/config/core/configmaps/autoscaler.yaml#L126.

[14] 2025. Knative Offerings. Available at https://knative.dev/docs/install/knative-offerings/.

[15] 2025. Kubernetes. Available at https://kubernetes.io.

[16] 2025. KWOK. Available at https://kwok.sigs.k8s.io/.

[17] 2025. OpenFaaS. Available at https://www.openfaas.com/.

[18] 2025. Prometheus. Available at https://prometheus.io/.

[19] 2025. Resource metrics pipeline - Kubernetes. Available at https://kubernetes.io/docs/tasks/debug/debug-cluster/resource-metrics-pipeline/.

[20] 2025. Understanding Lambda function scaling - AWS Documentation. Available at https://docs.aws.amazon.com/lambda/latest/dg/lambda-concurrency.html.

[21] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Symposium on Networked Systems Design and Implementation (NSDI)*. 419–434.

[22] Amazon Web Services. 2025. Amazon S3 Object Lambda. Available at https://aws.amazon.com/s3/features/object-lambda/.

[23] Amazon Web Services. 2025. Introducing the AWS Controllers for Kubernetes (ACK). Available at https://programming.am/lambda-with-kubernetes-aws-controllers-and-integration-use-cases-8d4bc77d0ce8.

[24] Apache. 2025. OpenWhisk. Available at https://openwhisk.apache.org/.

[25] Marc Brooker, Adrian Costin Catangiu, Mike Danilov, Alexander Graf, Colm MacCárthaigh, and Andrei Sandu. 2021. Restoring Uniqueness in MicroVM Snapshots. *CoRR* abs/2102.12892 (2021).

[26] Cristian Challu, Kin G. Olivares, Boris N. Oreshkin, Federico Garza Ramírez, Max Mergenthaler Canseco, and Artur Dubrawski. 2023. NHITS: Neural Hierarchical Interpolation for Time Series Forecasting. In *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI*. 6989–6997.

[27] Christian Posta. 2025. Application Network Functions With ESBs, API Management, and Now.. Service Mesh? Available at https://blog.christianposta.com/microservices/application-network-functions-with-esbs-api-management-and-now-service-mesh/.

[28] Lazar Cvetković, François Costa, Mihajlo Djokic, Michal Friedman, and Ana Klimovic. 2024. Dirigent: Lightweight Serverless Orchestration. In *Proceedings of the 30th ACM Symposium on Operating Systems Principles (SOSP)*. 369–384.

[29] Simon Eismann, Joel Scheuner, Erwin Van Eyk, Maximilian Schwinger, Johannes Grohmann, Nikolas Herbst, Cristina L. Abad, and Alexandru Iosup. 2022. The State of Serverless Applications: Collection, Characterization, and Community Consensus. *IEEE Trans. Software Eng.* 48, 10 (2022), 4152–4166.

[30] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 475–488.

[31] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *SIGMOD Conference*. 857–871.

[32] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019).

[33] Artjom Joosen, Ahmed Hassan, Martin Asenov, Rajkarn Singh, Luke Nicholas Darlow, Jianfeng Wang, and Adam Barker. 2023. How Does It Function?: Characterizing Long-term Trends in Production Serverless Workloads. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SOCC)*. 443–458.

[34] Kostis Kaffes, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2022. Hermod: principled and practical scheduling for serverless functions. In *Proceedings of the 2022 ACM Symposium on Cloud Computing (SOCC)*. 289–305.

[35] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. 2018. Pocket: Elastic Ephemeral Storage for Serverless Analytics. In *Proceedings of the 13th Symposium on Operating System Design and Implementation (OSDI)*. 427–444.

[36] Peter Kraft, Qian Li, Kostis Kaffes, Athinagoras Skiadopoulos, Deeptaanshu Kumar, Danny Cho, Jason Li, Robert Redmond, Nathan W. Weckwerth, Brian S. Xia, Peter Bailis, Michael J. Cafarella, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Michael Stonebraker, Lalith Suresh, Xiangyao Yu, and Matei Zaharia. 2022. Apiary: A DBMS-Backed Transactional Function-as-a-Service Framework. *CoRR* abs/2208.13068 (2022). https://doi.org/10.48550/ARXIV.2208.13068 arXiv:2208.13068

[37] Zijun Li, Jiagan Cheng, Quan Chen, Eryu Guan, Zizheng Bian, Yi Tao, Bin Zha, Qiang Wang, Weidong Han, and Minyi Guo. 2022. RunD: A Lightweight Secure Container Runtime for High-density Deployment and High-concurrency Startup in Serverless Computing. In *Proceedings of the 2022 USENIX Annual Technical Conference (ATC)*. 53–68.

[38] Qingyuan Liu, Dong Du, Yubin Xia, Ping Zhang, and Haibo Chen. 2023. The Gap Between Serverless Research and Real-world Systems. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SOCC)*. 475–485.

[39] Qingyuan Liu, Yanning Yang, Dong Du, Yubin Xia, Ping Zhang, Jia Feng, James R. Larus, and Haibo Chen. 2024. Harmonizing Efficiency and Practicability: Optimizing Resource Utilization in Serverless Computing with Jiagu. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*. 1–17.

[40] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware Data Passing for Chained Serverless Applications. In *Proceedings of the 2021 USENIX Annual Technical Conference (ATC)*. 285–301.

[41] Microsoft Azure. 2023. Azure Public Dataset: Azure LLM Inference Trace. Available at https://github.com/Azure/AzurePublicDataset/blob/master/AzureLLMInferenceDataset2023.md.

[42] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2018. SOCK: Rapid Task Provisioning with Serverless-Optimized Containers. In *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*. 57–70.

[43] Rohan Basu Roy, Tirthak Patel, and Devesh Tiwari. 2022. IceBreaker: warming serverless functions better with heterogeneity. In *Proceedings of the 27th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVII)*. 753–767.

[44] Mohammad Shahrad, Rodrigo Fonseca, Iñigo Goiri, Gohar Irfan Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*. 205–218.

[45] Arjun Singhvi, Arjun Balasubramanian, Kevin Houck, Mohammed Danish Shaikh, Shivaram Venkataraman, and Aditya Akella. 2021. Atoll: A Scalable Low-Latency Serverless Platform. In *Proceedings of the 2021 ACM Symposium on Cloud Computing (SOCC)*. 138–152.

[46] Dmitrii Ustiugov, Theodor Amariucai, and Boris Grot. 2021. Analyzing Tail Latency in Serverless Clouds with STeLLAR. In *Proceedings of the 2021 IEEE International Symposium on Workload Characterization (IISWC)*. 51–62.

[47] Dmitrii Ustiugov, Dohyun Park, Lazar Cvetkovic, Mihajlo Djokic, Hongyu Hè, Boris Grot, and Ana Klimovic. 2023. Enabling In-Vitro Serverless Systems Research. In *Proceedings of the 4th Workshop on Resource Disaggregation and Serverless (WORDS)*. 1–7.

[48] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XXVI)*. 559–572.

[49] Michal Wawrzoniak, Gianluca Moro, Rodrigo Bruno, Ana Klimovic, and Gustavo Alonso. 2024. Off-the-shelf Data Analytics on Serverless. In *Proceedings of the 14th Conference on Innovative Data Systems Research (CIDR)*.

[50] Michal Wawrzoniak, Ingo Müller, Gustavo Alonso, and Rodrigo Bruno. 2021. Boxer: Data Analytics on Network-enabled Serverless Platforms. In *Proceedings of the 11th Conference on Innovative Data Systems Research (CIDR)*.

[51] Ethan G. Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2019. The True Cost of Containing: A gVisor Case Study. In *Proceedings of the 11th workshop on Hot topics in Cloud Computing (HotCloud)*.