Serverless Computing: An Empirical Evaluation of Application Performance and Power Consumption in Knative

Karim Djemame*
Computer Science
University of Leeds
Leeds, West Yorkshire, United Kingdom
k.djemame@leeds.ac.uk

Ben Norton
University of Leeds
Leeds, United Kingdom
sc21bphn@leeds.ac.uk

Abstract

This paper investigates Knative, an open source serverless framework, and its performance and power consumption behaviours when running different types of applications. As such, it addresses the following research question to gain an understanding of the power consumption and performance benefits Knative may bring to a cloud environment under varying workload conditions: How do different types of serverless applications impact power consumption and performance in Knative Serving, compared to equivalent applications deployed on containers? The experimental results reveal that Knative can eliminate the idle power consumption of deployed applications due to its scale-to-zero capabilities. More notably, for applications operating near CPU and memory limits, using Knative to scale additional instances of the application can reduce response times without proportionally increasing overall power consumption.

CCS Concepts

• **Computer systems** → **Cloud computing**; • **Computer systems organization** → *Edge computing*.

Keywords

Cloud computing, Serverless computing, Containerisation, Knative, Power efficiency

ACM Reference Format:
Karim Djemame and Ben Norton. 2025. Serverless Computing: An Empirical Evaluation of Application Performance and Power Consumption in Knative. In *2025 IEEE/ACM 18th International Conference on Utility and Cloud Computing (UCC '25), December 01–04, 2025, Nantes, France.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3773274.3774676

1 Introduction

Power consumption and energy efficiency are critical considerations in the design of large-scale computing infrastructures, such as cloud data centers. The cost of powering the servers within these facilities has steadily increased with the development of higher performance systems.

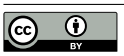
Thus, improving the energy efficiency of data centres is an important problem to tackle and requires optimisation across the cloud architecture. While enhancing cooling systems and hardware efficiency is essential, significant gains can also be achieved by reducing the energy consumption of the software running on this infrastructure, given that idle resources are estimated to account for 50% of data centre energy use [13]. One adopted approach to reducing this software-related power consumption is serverless computing which allows users to run applications as ephemeral, small runtime containers that can be invoked via event triggers and automatically scaled up and down from zero. When a serverless application is idle and receiving no requests, it is de-allocated after a certain time period, consuming no energy. Upon receiving new requests, it is automatically reallocated and scaled to meet demand. This dynamic scaling model makes serverless computing effective at reducing wasted energy usage from idle resources, as there is zero resource consumption when there is no demand [4]. It can also provide a cost-effective option for running applications in the public cloud, given Cloud Service Providers (CSPs) typically only charge for the number of code invocations and execution time. Other advantages include the elimination of infrastructure management for users, allowing them to focus solely on writing code, and built-in fault tolerance.

There are many serverless platforms available, broadly categorised as either commercial or open source. Commercial platforms, such as AWS Lambda and Azure Functions, manage the serverless infrastructure on behalf of the user, handling tasks like function execution, networking, and fault tolerance. However, using these platforms can subject developers to vendor lock in, where only functions corresponding with the platform's standards can be developed and executed.

On the other hand, open source serverless platforms like Knative [8] can help developers avoid vendor lock-in by bringing serverless to a private cloud and providing a pluggable framework that integrates with a wide range of third-party services. The paper investigates Knative as it is one of the most popular open source serverless frameworks and has been noted as a promising serverless platform suitable for further development and innovation [10];

While this paper shares similarities with prior academic work, e.g. [14], it differs in some key ways: 1) it focuses specifically on Knative's power consumption, a topic not previously explored, evaluated across five distinct application categories; 2) it explores power consumption at a fine-grained, pod level, rather than at a system-wide level, allowing a deeper insight into how pod deployments affect application performance and power usage, and 3) it evaluates not only the effects of workload intensity on performance

*Corresponding author.



UCC '25, Nantes, France

ACM ISBN 979-8-4007-2285-1/25/12
https://doi.org/10.1145/3773274.3774676

and power usage, but also concurrency and scaling for a more comprehensive analysis.

The paper's contributions are:

(1) An experimental architecture design for Knative that allows the performance and power consumption of running applications to be measured accurately in real time. A test suite of serverless workloads is developed to be run on the proposed test beds; this includes CPU, memory, disk I/O, and NIC I/O intensive serverless applications, as well as a multi-function workflow.
(2) an investigation of how different serverless application types impact Knative's power consumption and performance and identifies the underlying causes of these effects.
(3) a set of optimisation recommendations, based on new findings, to improve the power-performance trade-off in Knative, particularly in the context of specific workload characteristics.

The paper is structured as follows: the related work is reviewed in section 2. The experimental environment setup and test functions for various serverless use case applications are described in Section 3. Section 4 evaluates the results of the experiments and uses our findings to support or challenge the hypotheses. Section 5 concludes the paper and describes future work.

2 Related Work

A meta-analysis of academic studies on serverless computing is found in [15] to highlight the lack of awareness of performance variance. Mohanty et al. [12] compare the performance of open-source serverless platforms (Kubeless, OpenFaaS, Fission, and Apache OpenWhisk), finding Kubeless to exhibit the most consistent performance. Similarly, Li et al. [10] evaluate the performance of Nuclio, OpenFaaS, Knative, and Kubeless by generating HTTP workloads for empty functions. Contrary to the earlier study, Nuclio is identified as the most performant, followed by Knative. Knative is also found to scale the fastest, as it can launch multiple container instances at a time.

The EneA-FL scheme to promote smart energy management in serverless computing scenarios where Federated Learning clients are characterised by highly heterogeneous computing capabilities and energy budgets is presented in [1]. Workload awareness to optimise energy consumption and tackle the power control problem in data centres from the perspective of serverless computing is leveraged in [14].

Further research into serverless computing platforms highlights their energy efficiency compared to traditional container-based solutions. For instance, [3] demonstrate that OpenFaaS consumes significantly less power than Docker containers under memory-intensive workloads, achieving a 58% reduction in energy consumption when compared to Kubernetes-based [9] implementations. The approach in [6] seeks to reduce the power consumption of OpenFaaS while maintaining function Service Level Agreements (SLA), striking a balance between performance and power usage.

Some academic studies aim to improve the energy efficiency of serverless computing frameworks through scheduling and resource allocation strategies. [5] present an energy-efficient pluggable solution to scheduling in Kubernetes in a serverless computing environment through the integration of a Machine Learning-based model into the scheduler. An investigation of the key factors influencing serverless energy consumption in Software Defined Networks by examining the correlations between the energy usage of serverless applications and traffic flow patterns is found in [2].

3 Methodology

To guide the methodology and experiment design process, we adapt a version of the Cloud Evaluation Experiment Methodology (CEEM) framework [11], which advocates for clear, reproducible test specifications, ensuring reliable results.

3.1 Applications

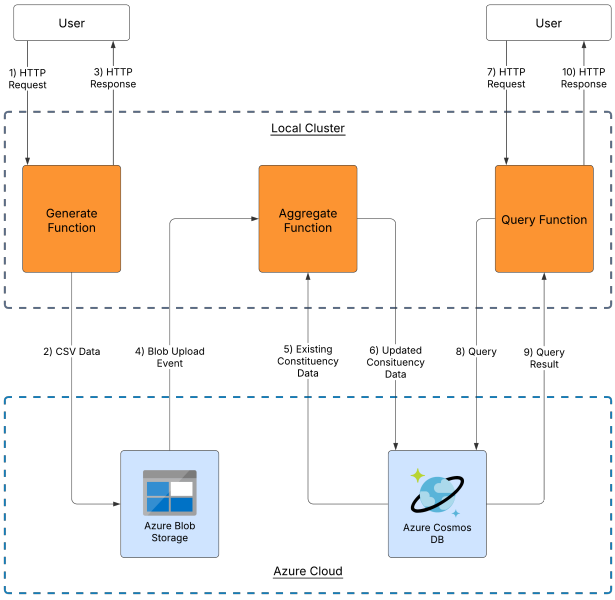


Figure 1: Test Workflow Architecture

In this section, we provide a design overview of the applications used to target each previously identified system resource or feature. Each application is designed to target a single resource or feature.

CPU: To stress the CPU we follow prior literature and create a simple serverless function that computes Pi to any specified number of decimal places. This is also the method used by benchmarking software to stress a system's CPU.

Memory: To stress RAM, a serverless function is created that multiplies two N x N matrices, where N, typically a large value, is provided by the user. Although this is the most common method to stress the memory of a system, it can also place a significant load on the CPU, which could be a factor to consider during evaluation.

Disk I/O: The system disk is stressed through a serverless function that iteratively reads and writes random data to a local lightweight, self-contained SQLite database. Each iteration consists of a single read and a single write to the database, with the number of iterations specified by the user.

NIC I/O: To stress the NIC, a similar serverless function to the Disk I/O application is used, except it reads and writes to a remote database hosted on Azure cloud instead (Microsoft Azure Table Storage). Through executing read and write operations on a remote source, the network connection becomes the primary bottleneck, thereby stressing the NIC.

Workflow: To stress platform workflow capabilities, we implement a workflow consisting of three functions (Figure 1):

Generate Function – When invoked, the generate function produces a random dataset and uploads it to Microsoft Azure Blob Storage. This dataset simulates voting data in a democratic election process and is stored as a CSV file. It includes vote counts for each party, along with metadata such as constituency and polling station details. The function plays a critical role in the workflow, providing input data required by subsequent functions. Its design attempts to mirror how real-world serverless workflows are triggered, where periodic data uploads to cloud storage trigger a sequence of functions to process the data step by step.

Aggregate Function – This function is triggered when the Generate Function uploads a new dataset to remote blob storage. Upon invocation, it determines the constituency of the new voting data and aggregates it with existing constituency data in Azure Cosmos DB. The updated totals are then written back to the database. This function represents the transformation step common in real-world serverless workflows, where raw input data is processed for later use.

Query Function – This function is triggered by an HTTP GET request specifying a constituency and queries the Cosmos DB for the relevant voting data and returns the result as an HTTP response. It is the final step of the workflow, where processed data is exposed through a stateless API, which is an approach widely used in production serverless systems to enable on-demand data access by users or other services.

3.2 Metrics

To measure pod power consumption in Knative, we track *Pod CPU Power, Pod RAM Power,* and *Pod OTHER power* metrics. CPU and RAM power are measured independently since these components are the primary contributors to dynamic power consumption. For simplicity, the power consumption of other system components are placed under the umbrella of *OTHER*. We measure pod power directly because it likely yields more accurate power measurements than estimating the percentage of total system power consumed by a pod.

Measuring pod performance and resource utilisation on Knative is done via *Pod Response Time, Active pods, Pod CPU Usage,* and *Pod Memory Usage*. *Pod Response Time* is selected because this is a widely adopted metric in serverless computing literature for measuring performance [15]. *Active pods* helps quantify if Knative is auto-scaling an application and to what extent. *Pod CPU/memory utilisation* are measured to understand their impact om power consumption.

To measure power consumption on Docker, we track *Container CPU Power*. For performance and resource utilisation, we use *Container Response Time*, *Container CPU Usage*, and *Container Memory Usage*. Only CPU power consumption is measured here because Docker does not provide a direct way to monitor container power usage. Estimating CPU power draw on Docker is done by using the CPU's utilisation and Thermal Design Power (TDP) in an adapted version of the power model proposed in [3].

3.3 Experimental Design

Each test is structured around the independent variables whose effects on performance and power usage we aim to measure. The set of independent variables are *deployment platform*, *application type*, *workload intensity, concurrency level, test duration,* and *ramp-up period*. Two identical tests are created for each application type, with one run on Knative and the other on Docker. Combined, these tests are designed to address all the experiment's requirements.

Each test is divided into a series of test cases (Table 1). Test cases TC1-1 to TC1-5 apply varying load intensities while keeping the request rate consistent, but non-concurrent. Test cases TC2-1 to TC2-3 are designed to evaluate how request concurrency and auto-scaling affect pod performance and power draw. All three cases maintain a consistent medium-high workload intensity but progressively increase the number of concurrent request threads. TC2-1 targets high concurrency without triggering Knative's auto-scaler. TC2-2 raises concurrency to prompt the provisioning of at least one additional pod, while TC2-3 aims to trigger at least two additional pods. Each test runs for ten minutes, with the first five minutes serving as a ramp-up period to the target concurrency level. This ramp-up phase allows us to observe how performance, power consumption, and resource utilisation change as additional pods are deployed. The final five minutes, during which the application operates at target concurrency, are used to assess each platform's ability to handle sustained high loads. It is important to note that during the scaling tests, newly provisioned pods will likely experience colds starts, which may temporarily increase the response time of the application. Lastly, a separate, one-off test is designed to assess the idle power consumption of applications running on Docker. The results can then be compared to Knative, whose idle application power usage is zero due to its scale-to-zero capability. This also enables insights into whether certain applications consume more power than others when idle. For this test, all five designed applications are run simultaneously on Docker in an idle state for 30 minutes, with the average power consumption of each application over the entire test duration then measured.

Table 1: Design Approach to Application Stress Testing

Test Case	Workload Intensity	Request Concurrency	Duration (mins)	Ramp-up (mins)
TC1-1	Low	None	5	0
TC1-2	Medium	None	5	0
TC1-3	Medium-high	None	5	0
TC1-4	High	None	5	0
TC1-5	Very high	None	5	0
TC2-1	Medium-high	Medium-high	10	5
TC2-2	Medium-high	High	10	5
TC2-3	Medium-high	Very high	10	5

3.4 Hypotheses

We posit the following hypotheses regarding the expected outcomes, which address all the experiment's requirements:

- **HYP-01** - *The performance and power consumption of applications will vary depending on application type, but patterns of variation will remain similar across platforms.* CPU and memory-intensive applications are likely to perform well as both Docker containers and Knative pods, but will draw more power due to their higher computational demands. In contrast, I/O-bound applications may exhibit lower performance because of read/write bottlenecks and additional latency introduced by network communication, but they will consume less power due to reduced reliance on CPU and memory. The workflow application is expected to consume the most power overall, as it invokes multiple functions during execution.
- **HYP-02** - *Across all applications, Docker containers will have lower response times and power consumption than Knative pods under increasing load with no request concurrency.* This is due to the extra routing layers and infrastructure in Knative, which likely introduces an overhead [10].
- **HYP-03** - *Power consumption of both Knative pods and Docker containers will increase with CPU and memory utilisation.* This is due to CPU and memory being the primary drivers of dynamic power.
- **HYP-04** - *Across all applications, Knative pods will have lower response times but higher power consumption than Docker containers under increasing request concurrency.* This is due to Knative's ability to scale up pods in response to demand, whereas Docker lacks auto-scaling.

3.5 Testbed

The Knative testbed architecture Design is shown in Figure 2. To run Knative, a base Kubernetes (K8s) Cluster is required, which Knative extends. Kepler [7], built for obtaining accurate power measurements on K8s clusters, is used to scrape power metrics about each cluster component at both process and pod levels. The Docker test bed is more basic than the Knative setup. Each application simply runs on Docker Engine, without an orchestration layer like K8s. The test beds configuration is shown in Table 2.

Table 2: Configuration of Experimental Test Beds

Property	Value	Property	Value
OS	Linux Ubuntu 24.04.1 LTS	Docker Engine	v28.0.4
CPU	Intel Core i7-6700 CPU 3.40GHz	Minikube	v1.35.0
vCPUs	8	Minikube Knative Serving	v1.17
CPU TDP	65 W	Apache JMeter	v5.6.3
RAM	16GB	Kourier	v1.17
SSD	256GB	Sslip.io	v3.2.6
Kepler	v0.8.0	Kube-Prometheus	v0.14.0

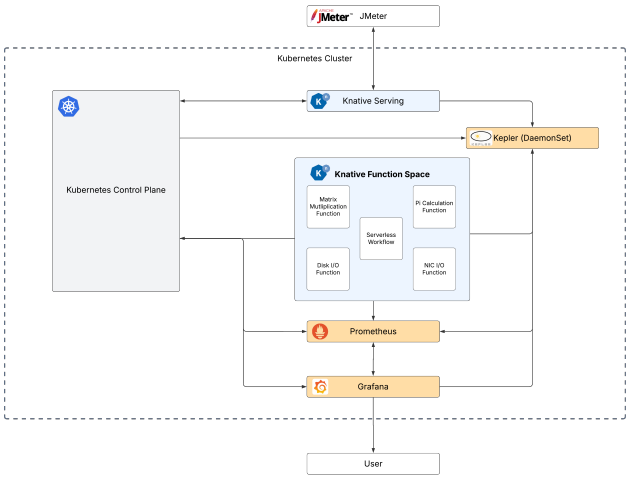


Figure 2: Knative Test Bed Architecture

4 Results and Discussion

Given the volume of results collected and the limited space of the paper, not all application test data can be presented and analysed within the main body. Therefore, this evaluation primarily focuses on selected results from the workflow application. Comprehensive results for the CPU-Stress, Memory-Stress, as well as the idle power of containers on Docker are found in the Appendix.

4.1 Workflow Stress

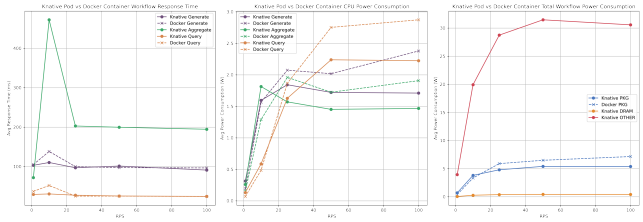


Figure 3: Results of Non-concurrent Workflow Tests comparing Knative and Docker in terms of response time and power consumption

Figure 3 illustrates how the *Workflow* application exhibited slightly different performance and power consumption patterns under increasing non-concurrent workload intensity compared to the other applications. Notably, Docker's response time for both the Generate and Query functions was generally equal to Knative's, but in some cases, it was even higher. However, the response times for all functions began to plateau as workload intensity increased, indicating that the workflow was likely experiencing throttling effects.

A second key difference observed in these tests is that, from 25 RPS and above, each function's CPU power consumption was higher when run as a Docker container than as a Knative pod. As a result, the CPU power usage of the entire workflow was approximately 1-3 W higher on Docker under medium to very high workload intensities. The overall CPU power consumption of the workflow was lower than that of both the *CPU-Stress* and *Memory-Stress*

applications, whereas its Memory and OTHER power usage was comparable.

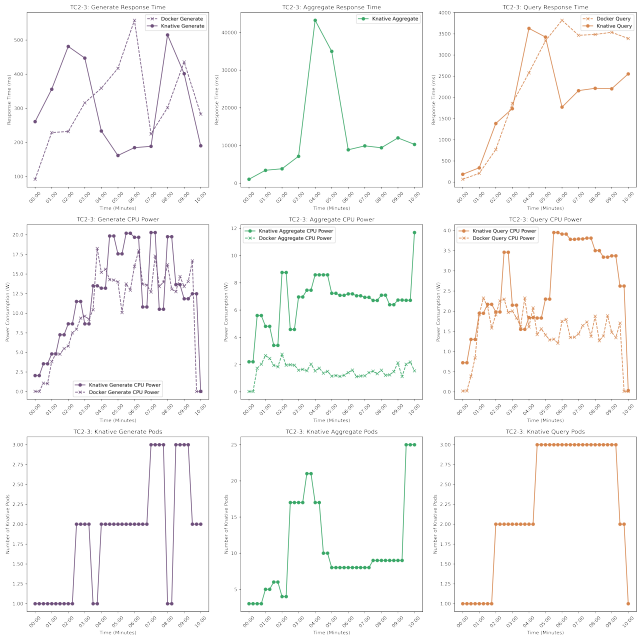


Figure 4: Results of the Workflow-Stress TC2-3 concurrency test, showing Knative vs Docker performance and power consumption while scaling up to 145 RPS

Figure 4 shows how each function in the *Workflow* application scales up to 145 concurrent requests. The *Aggregate* function performs the worst, with response times peaking at 40 seconds midway through the test before dropping to 10 seconds, and then gradually rising again. This occurs despite deploying up to 25 pods, which is significantly more than any other application deployed at this concurrency level.

This high number of deployments is likely due to the *Aggregate* function's reliance on two remote storage devices, thus introducing a network latency which causes requests to accumulate more quickly and trigger aggressive scaling. Consequently, Knative's CPU power consumption for this function remains higher than Docker's throughout the test, averaging 7-9 W compared to 2 W. Although, notably, Knative's CPU power usage at 17 pods is similar to that at 8-9 pods, suggesting the additional pods contribute little computational load and may be bottlenecked by I/O operations rather than CPU.

The *Generate* function performed well on both Knative and Docker, with Knative generally achieving lower response times due to its auto-scaling capability. Exceptions occurred at *03:30* and *08:00*, when the auto scaler temporarily scaled down to a single pod before recovering. In terms of power consumption, the function's behaviour is not dissimilar to that of the *Memory-Stress* application. Although Docker consumed less CPU power overall, it frequently peaked at levels close to Knative's, even when Knative was running more pods. Additionally, Knative's CPU power consumption remained similar between 2 and 3 pods, suggesting that the *Generate* function is not CPU-intensive, and therefore the extra pods contributed less significant increases to power usage.

Lastly, the *Query* function performed moderately well, with response times peaks at 3.5 to 4 seconds on both platforms. Interestingly, Knative's response time and power consumption remained comparable to Docker's when running two pods. Only after the third pod was deployed was Knative's response time dropped 1-1.5 seconds below Docker's, accompanied by a corresponding increase in CPU power consumption of around 2 W.

4.2 Hypotheses Revisited

We address each of the hypotheses stated in section 3.4, using the results of the experiment, and determine whether each hypothesis is proven correct.

HYP-01 – This hypothesis was *confirmed*. The *Memory-Stress* application consistently performed the worst across all tests and consumed the most power, likely due to its high CPU and memory demands. In contrast, the *Workflow* application consumed the least power under non-concurrent workloads while maintaining low response times. However, under concurrent workloads, it exhibited increased power consumption and longer response times beyond that of the *CPU-Stress* application. For each application under non-concurrent workloads, the Docker container and Knative pod versions demonstrated the same performance and power consumption trends as workload intensity increased. Under concurrent workloads, this trend similarity largely persisted. The only notable deviations emerged when Knative scaled horizontally by deploying multiple pods, while Docker remained constrained to a single container, resulting in a divergence in both performance and power consumption. However, since Docker does not support automatic scaling in this setup, it remains inconclusive whether these trends would have continued with multiple Docker containers.

HYP-02 - This hypothesis was *disproven*. Under non-concurrent workloads, the *CPU-Stress* application exhibited marginally lower response times and power consumption when deployed as Docker containers compared to Knative pods. Similarly, the *Matrix-Stress* application followed the same pattern in response times, though it showed a more pronounced difference in power consumption between Docker and Knative. However, this trend did not hold for the *Workflow* application, where the workflow's functions each consumed less power as Knative pods at medium to high workloads than their Docker container counterparts. Additionally, there were instances where the *Generate* and *Query* functions responded quicker when deployed as Knative pods. Overall, performance was largely consistent between platforms across all applications, while power consumption exhibited greater variability depending on the application type.

HYP-03 - This hypothesis is *partially proven*. Our results only provide data on the relationship between power consumption and resource utilisation for Knative pods, so the hypothesis cannot be fully validated for both platforms. It is only assumed that Docker containers exhibit similar behaviour, based on their comparable performance and power consumption to Knative in the non-concurrent workload tests. Focusing on Knative, the *CPU-Stress* results clearly show that CPU power consumption increases proportionally with

CPU utilisation, supporting the hypothesis. However, the *Memory-Stress* results reveal that higher memory utilisation does not necessarily result in increased memory power usage, thus presenting a deviation from the hypothesis.

HYP-04 - This hypothesis was *disproven*. Under concurrent workloads, all applications on Knative demonstrated reduced response times compared to Docker, due to Knative's ability to deploy additional pods. However, the assumption that more pods would always lead to higher power usage did not hold. For instance, the *Matrix-Stress* application maintained comparable power usage to Docker while operating with two active pods, thereby contradicting the hypothesis.

4.3 Recommendations

Based on our findings, several recommendations can be made:

1) Adopting a serverless architecture such as Knative can help eliminate idle power consumption of deployed applications. However, a potential drawback, though not explored in this project, is the occurrence of cold starts, which introduce additional latency and may increase power usage when applications are first invoked.

2) Scaling memory-intensive applications on Knative can enhance response times without significantly increasing power consumption, as higher memory utilisation does not necessarily correlate with increased memory energy usage.

3) For applications already operating near CPU and memory limits, using Knative to scale additional pods can reduce response times without proportionally increasing overall power consumption, unlike at lower resource utilisation levels.

4.4 Limitations

While this project contributes new findings to research on serverless computing, it is important to consider the limitations of the work conducted.

1) Knative's test results are benchmarked against standalone Docker containers, which by default do not auto-scale. While this choice of benchmark is consistent with prior work, it does mean our evaluation might not fully capture Knative's performance and power consumption advantages or disadvantages under scaling conditions, as it was not benchmarked against frameworks capable of auto-scaling.

2) Knative's power consumption was captured using Kepler, providing power usage measurements across a range of components. For the Docker benchmark, only CPU power consumption could be measured. This could potentially introduce small inaccuracies into the comparative analysis.

3) The experiment was executed solely on a single controlled testbed. Typically, the cloud infrastructures that Knative and Docker are run on consist of heterogeneous, multi node clusters that can exhibit a wider range of performance and power consumption behaviours. As a result, our findings may not fully generalise to large-scale, production-level cloud environments.

5 Conclusion

In summary, this paper addresses the research question: *"How do different types of applications impact the power consumption and performance of pods in Knative Serving, compared to containers running equivalent applications on Docker?"* To explore this, CEEM was used to define a clear set of requirements and guide the experiment design. A set of targeted recommendations was developed to guide users on how Knative can be used to increase application performance without significantly impacting power usage.

To improve the reliability of the findings, future work should include further experiments to account for performance variability and to solidify findings. A natural extension of this project would be to examine how the power consumption of Docker containers running on a Kubernetes cluster compares to Knative pods. This would help identify differences in scaling behaviour between the two platforms and determine whether one is more power-efficient or performant when scaling to multiple containers or pods. Additionally, it would be valuable to explore the impact of Knative Eventing on performance and power consumption, as this paper focused solely on Knative Serving, and these components are typically used together in real-world deployments.

References

[1] A. Agiollo, P. Bellavista, M. Mendula, and A. Omicini. 2024. EneA-FL: Energy-aware orchestration for serverless federated learning. *Future Generation Computer Systems* 154, C (June 2024), 219–234.

[2] A. Alhindi and K. Djemame. 2024. SDN Traffic Flows in a Serverless Environment: a Categorization of Energy Consumption. In *Proc. of the 17th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2024)*. Sharjah, UAE.

[3] A. Alhindi, K. Djemame, and F. Banaie. 2022. On the power consumption of serverless functions: an evaluation of openFaaS. In *Proceedings of the 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC 2022)*. IEEE, Vancouver, 366–371.

[4] M. Chadha, T. Subramanian, E. Arima, M. Gerndt, M. Schulz, , and O. Abboud. 2023. GreenCourier: Carbon-Aware Scheduling for Serverless Functions. In *Proceedings of the 9th International Workshop on Serverless Computing* (Bologna, Italy) *(WoSC '23)*. ACM, New York, NY, USA, 18–23.

[5] R. Chiorescu and K. Djemame. 2024. Scheduling Energy-Aware Multi-Function Serverless Workloads in OpenFaaS. In *Proc. 20th Conference on the Economics of Grids, Clouds, Software, and Services (GECON'2024)*. Springer, Rome, Italy. LNCS 15358.

[6] X. Jia and L. Zhao. 2021. RAEF: Energy-efficient Resource Allocation through Energy Fungibility in Serverless. In *Proceedings of the 27th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE, Beijing, China, 434–441.

[7] Kepler Project. 2025. Kubernetes Efficient Power Level Exporter. https://sustainable-computing.io/.

[8] Knative Project. 2025. Knative. Providing the building blocks for creating modern, cloud-based applications. https://knative.dev/docs/.

[9] Kubernetes Authors. 2024. Kubernetes Documentation. https://kubernetes.io/docs/home/

[10] J. Li. 2021. Analyzing Open-Source Serverless Platforms: Characteristics and Performance (S). In *Proceedings of the 33rd International Conference on Software Engineering and Knowledge Engineering (SEKE2021, Vol. 2021)*. KSI Research Inc., 15–20.

[11] Z. Li, L. OBrien, and H. Zhang. 2013. CEEM: A Practical Methodology for Cloud Services Evaluation. In *2013 IEEE Ninth World Congress on Services*. 44–51.

[12] S.K. Mohanty, G. Premsankar, and M. Di Francesco. 2018. An Evaluation of Open Source Serverless Computing Frameworks. In *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 115–120.

[13] P. Patros, J. Spillner, A. Papadopoulos, B. Varghese, O. Rana, and S. Schahram. 2021. Toward Sustainable Serverless Computing. *IEEE Internet Computing* 25, 6 (2021), 42–50.

[14] J. Stojkovic, N. Iliakopoulou, T. Xu, H. Franke, and J. Torrellas. 2024. EcoFaaS: Rethinking the Design of Serverless Environments for Energy Efficiency. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 471–486.

[15] J. Wen, Z. Chen, F. Sarro, and S. Wang. 2025. Unveiling overlooked performance variance in serverless computing. *Empirical Softw. Engg.* 30, 2 (Jan. 2025), 26 pages.

A Appendix

A.1 Idle Test

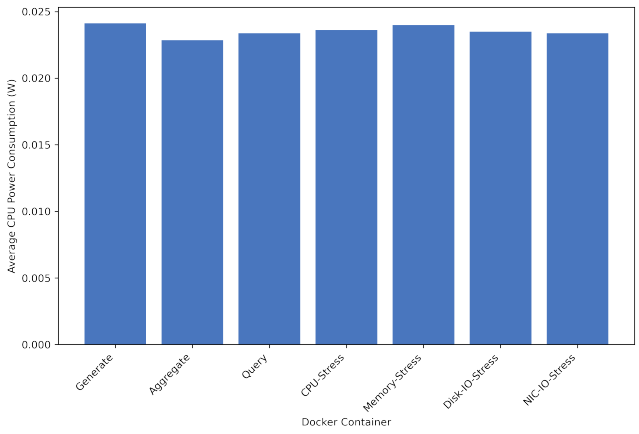


Figure 5: Idle Power of Containers on Docker

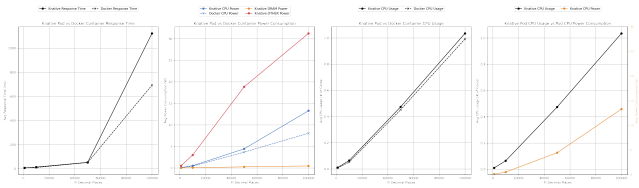


Figure 6: Results of Non-concurrent CPU-Stress Tests comparing Knative and Docker in terms of response time, power consumption, and CPU usage

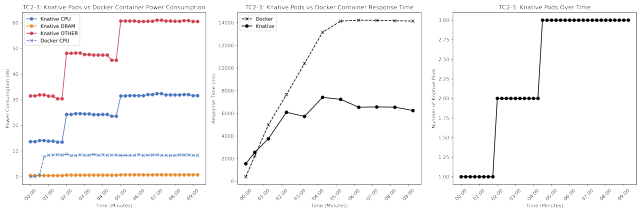


Figure 7: Results of the CPU-Stress TC2-3 concurrency test, showing Knative vs Docker performance and power consumption while scaling up to 145 RPS

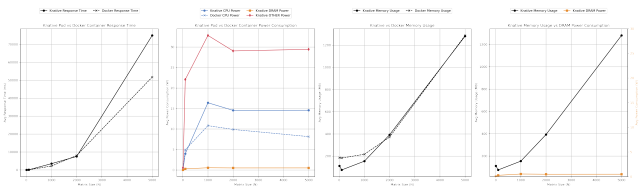


Figure 8: Results of Non-concurrent Memory-Stress Tests comparing Knative and Docker in terms of response time, power consumption, and memory usage

Figure 5 shows the idle CPU power consumption of all test applications on the Docker test bed. There is no equivalent plot for the Knative test bed, as it scaled applications down to zero when idle, resulting in no power consumed. Overall, idle CPU power usage on Docker showed minimal variation between applications, averaging around 0.024 Watts (W) per application, which is 0.024 W more than on Knative.

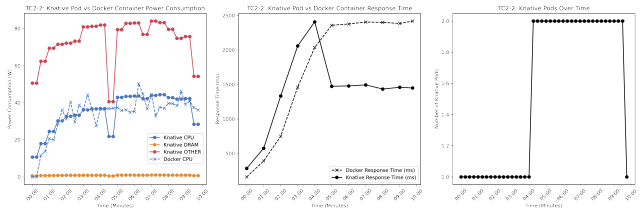


Figure 9: Results of the Memory-Stress TC2-2 concurrency test, showing Knative vs Docker performance and power consumption while scaling up to 75 RPS

A.2 CPU Stress

Figure 6 shows that, with the exception of the *100,000* decimal places test, Knative and Docker exhibited similar response times under increasing CPU load without concurrency. While CPU utilisation remained comparable across both platforms, Docker consistently consumed less CPU power than Knative at all workload levels. As expected, Knative's CPU power consumption scaled proportionally with its CPU usage as the workload intensity increased. This is a trend that is assumed to also apply to Docker. Notably, despite the application primarily targeting the CPU, Knative's OTHER component power consumption was significantly higher than that of the CPU. In contrast, Knative's memory power consumption remained nearly constant across all workloads, as anticipated, since the workload was not designed to stress memory.

Figure 7 shows that under increasing concurrent requests, Knative scaled more effectively than Docker, maintaining roughly half the response time at a peak load of 145 concurrent RPS. Until the first additional pod was deployed at 02:00, both platforms showed similarly rising response times, although Docker consumed less CPU power than Knative. After 02:00, Knative kept response times stable by deploying additional pods, while Docker, lacking dynamic scaling, continued to degrade. However, Knative's improved performance came at a cost, with each new Knative pod increasing CPU power consumption by 8–10 W and OTHER power usage by nearly 20 W. In contrast, Docker's CPU usage remained steady at just under 10 W throughout. This trend in rising power consumption on Knative is unsurprising because, as seen in previous test cases, CPU power consumption is roughly proportional to CPU utilisation. Therefore, doubling the number of CPU-intensive pods leads to a significant increase in CPU usage and, consequently, power consumption.

A.3 Memory Stress

The *Memory-Stress* application showed significantly higher response times than the *CPU-Stress* application under increasing non-concurrent load. As shown in Figure 8, Docker and Knative exhibited similarly rising response times, except at the maximum matrix size, where their performance began to diverge. Overall, CPU power consumption for both platforms rose sharply with matrix size up

to 1000, with Docker generally consuming the same as Knative. However, beyond *N = 1000,* Docker began to consume noticeably less power than Knative, with a CPU usage difference of 5-7 W. The higher workload intensities saw Knative's CPU power usage drop slightly and then plateau, while Docker's steadily but modestly declined. This possibly indicates that both the Knative pod and Docker container were throttling under higher workloads due to resource contention affecting the Uvicorn process serving the application. The most valuable insight from these results is that Knative's memory power consumption remains nearly constant, even as memory usage increases significantly with workload intensity. In fact, memory power consumption is around 1 W at just under 200 MB of usage and remains roughly the same for 1.3 GB of usage. This suggests that, unlike CPU scaling, memory scaling is highly power-efficient. It is assumed that Docker exhibits the same trend, given its comparable performance and memory utilisation.

For the Memory-Stress application's concurrency tests, another noteworthy insight emerges. As shown in Figure 9, Docker's CPU power consumption remained mostly comparable to Knative's throughout the TC2-2 test, even after Knative deployed a second pod at 05:00. In fact, there are several points where Docker's CPU power briefly exceeds Knative's. Once an additional pod was scaled, Knative achieved a response time nearly half that of Docker, while consuming on average only 2 W more CPU power than the single Docker container. By contrast, the *CPU-Stress* concurrency tests saw Knative consume 15 W more CPU power at two pods than Docker at one container, also achieving half the response time. A similar pattern appears in the Memory-Stress TC2-3 test, though with a third pod, Knative's CPU power usage increases slightly further beyond Docker's.