TU Dublin, Tallaght Campus

MSc DevOps
Research Project

Knatve Vs Azure Function

DRAFT VERSION 1 (261023)

Ainul Habib

X00159358

Department of Computing

Supervised by

GARY CLYNCH
Department of Computing

27 October 2023

Declaration

I hereby certify that the material, which I now submit for assessment on the programmes of study leading to the award of Master of Science, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution.

Ainul Habib
27 October 2023

Acknowledgements

.. TBD ..

List of Figures

List of Tables

Contents

Abstract

Serverless computing technology provided the technological infrastructure that made the evolution of cloud computing from VM(s) and containerized services (CaaS) to just a "Function as a Service" (FaaS). This paradigm shift brought simplification of Microservices design, and abstraction for infrastructure and server overheads.

Apart from all the goodies of Serverless, it provides a very real possibility for customers to get in a "locked-in" with a particular provider.

> *While the existing cloud solutions for public and private companies are vendor-locked-in by design, their existence is subject to the limited possibility of interoperating with other cloud systems.*
> Opara-Martins, Sahandi and Tian 2014

The Serverless market needs a standardization, and unification by way of API offering and runtime contracts, to free developers from the fear of vendor lock-in.

Knative is an Open-Source, Cloud-Native, Serverless offering, that extends Kubernetes (, a widly accepted PAAS,) with Serverless features.

Azure Functions is a cloud-based service that provides event-driven and compute-on-demand capabilities for various Azure or third-party services.

We evaluate both platforms based on several criteria, such as performance, cost, ease of use, portability, and ecosystem. We also present a case study of a real-world application that uses both platforms and discuss the challenges and benefits of each approach.

Our results show that Azure Functions offers a more mature and integrated solution for Serverless computing, while Knative provides more flexibility and control for developers who want to leverage the power of Kubernetes.

Introduction

In recent years, the landscape of cloud-native application development has witnessed a significant transformation, driven by the need for enhanced scalability, rapid deployment, and cost efficiency. As organizations strive to deliver seamless digital experiences to their users, the utilization of Serverless computing and container-based solutions has emerged as a pivotal paradigm. Technologies that have gained prominence in this context are AWS Lambda, Azure Functions and Knative.

Knative, an open-source project initiated by Google, offers a platform that abstracts away much of the complexities associated with deploying and managing containerized applications. It provides a set of building blocks for modern Serverless development, enabling developers to focus on code logic while benefiting from automatic scaling, event-driven architecture, and enhanced portability.

The concept of Serverless computing has garnered immense attention for its promise of facilitating a zero-administration approach to application deployment. This paradigm allows developers to write code in the form of functions that are executed in response to specific events, without the need to explicitly provision or manage underlying infrastructure. This streamlined approach not only accelerates development but also optimizes resource allocation and minimizes operational overhead.

As organizations evaluate these two technologies for their cloud-native initiatives, critical considerations arise regarding their architectural differences, performance characteristics, scalability models, and suitability for various use cases. While Knative provides a comprehensive Serverless platform, Serverless computing offers a more granular approach to event-driven execution. Deciding between these two options necessitates an in-depth analysis of their strengths and limitations.

This research thesis aims to explore and compare Knative and Serverless computing in the context of modern application development. By delving into their architectural foundations, deployment models, scalability mechanisms, integration capabilities, and real-world performance, this study seeks to provide a comprehensive understanding of when and how each technology should be employed. Through empirical evaluations and case studies, this research intends to guide practitioners and decision-makers in making informed choices that align with their specific application requirements and organizational goals.

In the following chapters, we will look into the core concepts of Knative and Serverless computing, examine their respective advantages and challenges, and present empirical insights derived from experimentation. By shedding light on the nuances of these technologies, this research endeavors to contribute to the ongoing discourse surrounding cloud-native development and aid in shaping the future of distributed application architectures.

1 Path to Serverless

> *The promise of "increased agility, resilience, scalability, and developer productivity" and a desire for a "clear separation of concerns," has fuelled interest in and adoption of Microservices, and even helped to popularize important practices, such as DevOps.* (Killalea 2016)

Microservices architecture facilitates the application's capability to scale horizontally, better than any tightly coupled monolith, with the least amount of overhead. But there are other complicated issues related to distributed systems, which are more complicated requiring a complex and costly solution. They include issues like complex error handling, remote process calls, maintaining application logs, etc.

Running infrastructure is always expensive and has its own management overheads. Adoption of Microservices in the application architecture will involve expansion of the existing organization's IT infrastructures. The hidden cost related to distributed computing like networking, security, logs, etc. adds to the overall platform bill.

Many traditional applications were written as a monolith, with an N-tire system, which includes a Rational Database system and complex remote procedure calls. Design challenges for transitioning to Microservices have often been a painful and costly journey. This raises difficult questions for proposers of Microservices to answer to stakeholders.

Transition and adoption of Microservices are also driven by business agility, requiring faster deployment iteration and support for high scalability, but with reduced cost. Cloud computing does offer some relief, by taking away the pain of managing the infrastructure on-premises. Most infrastructure can be configured and provisioned on requirement basis, using a "pay-as-you-go" model.

Even the best Microservices solution cannot fulfill all business agility. Faster and quick-rolling out of business changes to market on time, is still a challenge for the architects and managers.

This is where Serverless offers a distinct advantage. Serverless technology streamlines the complexity of traditional systems by eliminating the need to run application servers and manage infrastructure. Developers leverage from Serverless framework, focusing on the core business problem, rather than worrying about infrastructure issues and bottlenecks.

Serverless computing's single-purpose APIs and web services make Microservices loosely coupled, scalable, and capable of delivering highly agile business solutions, easily and quickly.

Using a Serverless framework developer's energy and focus are more inclined towards business logic, and application flows and least towards any infrastructure or framework-related issues.

> *Market share for Serverless architecture is slated to increase by roughly 25.70 % by the end of 2035, adding expected revenue of approx 193 billion.* (*Global Market*

Insights Inc. Report ID: GMI3796 2022)

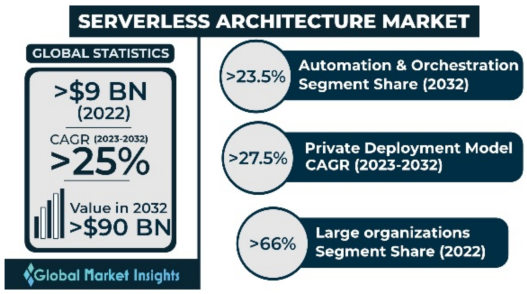


Figure 1: Serverless Global Market Share
(Ref: *Global Market Insights Inc. Report ID: GMI3796* 2022)

2 Serverless Framework

2.1 FAAS: Function As A Service

Function as a Service or FaaS is the basic building block of Serverless framework, (including AWS Lambda or Azure Function).

> *Function as a service (FaaS) is a category of cloud computing services that provides a platform allowing customers to develop, run, and manage application functionalities without the complexity of building and maintaining the infrastructure typically associated with developing and launching an app.*
> (*Wikipedia : Function as a service* 2023)

In FaaS, the cloud vendors are responsible for providing and managing the complexity of related application server, infrastructure, frameworks, security, and everything that the application needs to execute a specific function in an application.

> *The "compute containers" executing your functions are ephemeral, with the FaaS provider creating and destroying them purely driven by runtime need. Most importantly, with FaaS the vendor handles all underlying resource provisioning and allocation—no cluster or VM management is required by the user at all.*
> (*Serverless Architectures by Mike Roberts* 2018)

Following FaaS approach, Serverless offers on-demand functionality, which might live just for a single invocation. The cloud infrastructure will scale it to zero in preserve infrastructure and running costs. As such FaaS functionalities are opinionated stateless applications. This is a key difference between other architectural cloud-based offerings like containers and PaaS. AWS Lambda is the first publicly cloud-based offering of FaaS or Serverless.

2.2 Functional Triggers

FaaS functionalities are typically triggered by events. Event Types are defined by the vendors. The vendors for Serverless are also responsible in managing connections from various message sources. For Example: Azure Functions can react to Messaging Sources like Event-Grid, Event-Hub, Blob Storage or simple HTTP triggers. This approach fits perfectly with Microservices patterns of keeping the application event-driven and asynchronous.

One important feature of Functional Trigger, offered by Serverless, is that the developers do not need to write any infrastructure or platform-related code in this application. The Serverless framework also manages the consumption of data and its conversion to the "Cloud Event" format.

FaaS, will not require any application server to up in running, it can be "Scaled to Zero" if there is no event triggered. This difference, as stated earlier my Mark Roberts, becomes a key

factor in making FaaS solutions cost-effective. FaaS applications are booted and scaled up when an event is triggered from a configured event source.

However, the initial request in Faas has higher latency, (could be up to seconds,) than any other application platform. This “Cold restart” is an issue for FaaS and for Serverless offerings. The restart will depend on many factors, predominantly the application runtime and its runtime libraries. The effect of “cold restart” on the Quality of Service depends on type of traffic and applications.

2.3 Vendor Locking

In cloud computing, the *"vendor lock-in"* is a major problem. When an organization adopts a specific cloud vendor, the migration of application and data to alternative providers becomes extremely difficult and highly expensive. The cloud vendor’s framework and their adoption in application makes it incompatible with other vendors. The *Vendor Lock-in* occurs in various ways:

- by designing a system incompatible with software developed by other vendors
- by using proprietary standards or closed architectures that lack interoperability with other applications
- by licensing the software under exclusive conditions

Vendor lock-in discourages organizations in adopting new and emerging cloud technologies like Serverless. There is no standardization of Serverless design, data structure and code that could favour an easy transition from one cloud vendor to other, without an expensive redesign of the entire application.

> *Existing cloud computing solutions for enterprises have not been built with interoperability and portability in mind, hence applications are usually restricted to a particular enterprise’s cloud or a cloud service provider.* (Opara-Martins, Sahandi and Tian 2014)

In reality, this claim is somewhat correct because existing cloud solutions are tightly coupled to the proprietary technology they were designed for, consequently, locking customers into a single cloud infrastructure, platform, or service; preventing interoperability and portability of data or software created by them.

3 Cloud Native approach for Serverless

Cloud Native Computing Foundation (CNCF) introduced Kubernetes to counter vendor lock-in issues created by cloud vendors offering PaaS. Kubernetes is a new platform allowing containerized applications to run in their own sandbox (POD), isolated in their namespace.

Kubernetes is now accepted as the standard for PaaS. All major cloud vendors support the deployment of Kubernetes clusters in their cloud environments

4 Knative

The knative project was started by Google in 2018, with the aim to facilitate the deployment of a Serverless application using Kubernetes. Knative offers Serverless features like Scale to Zero, auto scalability, abstraction from PaaS and Infrastructure, and use of the eventing framework. Knative, leveraging from the underlying Kubernetes platform, offers a portable runtime and open API, which makes Serverless applications fully Cloud Native. This increases interoperability and eases the transfer of applications from existing cloud providers to others.

> *Knative is an open-source initiative aiming to provide a platform for developing container applications on top of the Kubernetes container orchestration platform. It offers ready-to-use components built with Kubernetes Custom Resource Definitions (CRDs) for developers to build, deploy, and scale their functions end-to-end from source code.* (Lin and Glikson 2019)

4.1 Knative Function

Knative provides the framework for creating a Cloud Native Serverless function that is deployed to a Kubernetes cluster. There are three components to Knative (a) Build, (b) Serving, (c) Eventing.

Knative Build component is responsible for converting the functional code to a container image, burning code or logic, the runtime, dependencies, configuration, and environment variable, etc. to OCI standard Image, pushing it to an internal Kubernetes registry.

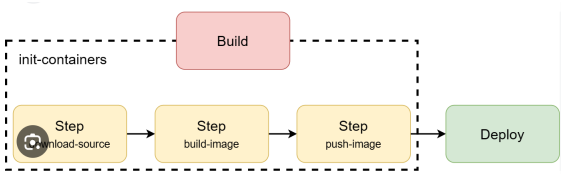
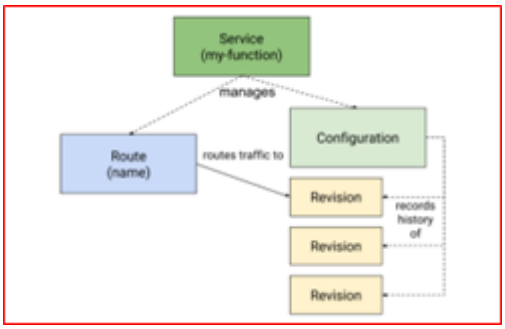


Figure 2: Knative Build Stage

Knative Serving is a request-driven provisioning component, responsible for auto-scaling Serverless containerized instances. It also takes care of load balancing. Knative Serving also abstracts the Kubernetes service to administer the provisioning of new service versions. It also manages configuration updates, by rolling out a new immutable revision representing the updated functional state. Each service saves specific routing information to redirect traffic to a specific instance version.

> *Knative Serving is ideal for running your application services inside Kubernetes by providing a more simplified deployment syntax with automated scale-to-zero and scale-out based on HTTP load. The Knative platform will manage your service's deployments, revisions, networking, and scaling.* (Sutter and Sampath 2020)



4.2 Leveraging Service Mesh

Knative leverage the service mesh to efficiently manage traffic routing between services which maintain immutable revision of Serverless application. It aids in satisfying the Serverless framework requirement of dynamic scaling, scale to zero and ease of deploying new versions. Various Service Mesh technologies are available that can be used in Knative example: Istio, kourier, etc.

4.3 Knative Eventing

Knative Eventing provides sets of primitives for consuming and producing events, assisting in satisfying the Serverless requirement of event-driven applications.

> *Knative Eventing was to enable late binding of producers and consumers. It should be possible to connect consumers to producers without either component needing to know the configuration of the other.*
>
> *CNCF to host CloudEvents in the Sandbox, by Kristin Evans* 2023

Just like any other message-driven system, the Knative Eventing component is composed of:

- an Event-Source, which produces events from various sources.
- a Channel, which saves and buffers the event between and producers and consumers.
- and Subscriptions, which forwards events from Channel to services or other channels

The message-driven approach makes services loosely coupled and independent. Knative

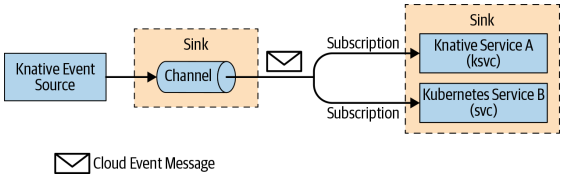


Figure 3: Knative Eventing Channel, Ref: Sutter and Sampath 2020

eventing framework makes Serverless applications event-driven and asynchronous. Knative being open source also allows the plugging of third-party Event Sources like Kafka Topics.

4.3.1 Using Broker and Triggers

Brokers
Knative also offers Broker, which are Kubernetes Custom Resources and services, defining the event mesh, collecting pool of events. Knative Eventing will implicitly create a Knative Eventing Channel for the Broker. The Brokers expose an endpoint, using ingress. Event Producers can POST event directly to broker using the ingress based endpoint.
Triggers
The Custom resources for Knative Eventing, also includes Triggers, which is responsible to send events to a configured *Event Sinks*. Triggers subscribe to Broker's underlying event channel, consuming events as they arrive.
The downstream Sink service, supports a CloudEvent response, which is routed back to the Broker, and finally to event channel.
Filters
Knative Broker also supports event filtering.

> *"Event filtering is a method that allows the Subscribers to show an interest in a certain set of messages that flows into the Broker." (Sutter and Sampath 2020)*

Trigger is also responsible for Filtering of events before dispatching it to configured Sink. It subscribes to the broker to consume events, then apply filter on it.

In the easiest term, Knative Broker is an abstraction over event channel, subscription and triggers.

The developers leverage for Knative API and Eventing platform to consume events in their functional code without writing any infrastructure code for consuming and parsing it. The API makes it possible to consume events in the standard "CloudEvent" format.

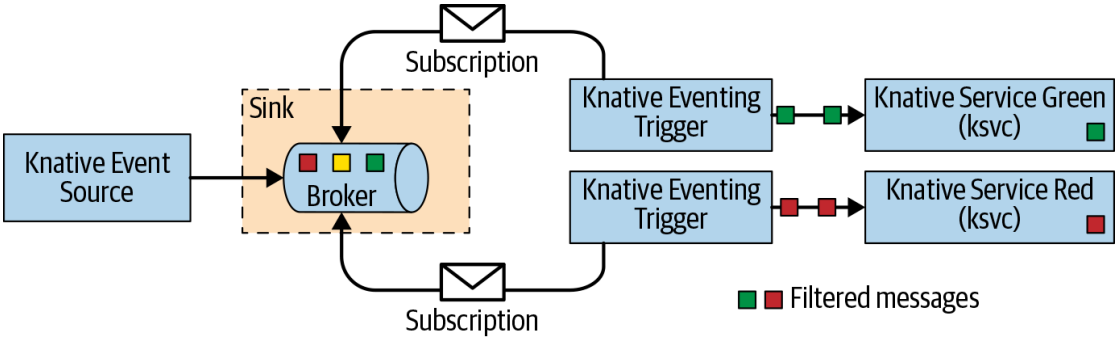


Figure 4: Knative Event Source, Broker Trigger and Sink
"The filters are applied on the CloudEvent attributes of the messages, before delivering the message to the interested Sink Services"
(Ref : Sutter and Sampath 2020)

5 Azure Function

Azure Function is a Serverless offering by Microsoft. It allows developers to deploy their FaaS application on Azure Cloud using a Pay-as-you-go model. It followed all the features of Serverless, like the scale to zero, horizontal scalability, complete abstraction from the Platform, and eventing framework. It offers SDK and CLI to build and Deploy Serverless applications to the cloud, using various programming languages and frameworks like Java, .Net, NodeJS, etc.

Azure Functions adopts the Azure Eventing framework to provide a comprehensive set of event-driven triggers and bindings. This binding connects Azure functions to other services without having to write extra code, providing the capability of Azure function to be asynchronous and event-driven.

Azure Serverless Functions are grouped in *Azure Function App*, which provide convenience in deployment, management, configuration, scalability and resource sharing, represented as one logical unit. Azure deploys its Serverless code, runtime dependencies and configuration information under its *Functional App*'s file storage.

6 Azure Eventing

Azure Functions leverages Azure eventing framework to create Event-Driven, Asynchronous Serverless Functions. The common use case for this feature is creating Workflow.

> *Serverless is a great fit for event-driven architectures because each message acts an individual independent unit of work that can be processed Serverlessly.* ANDERSON 2024

Azure Eventing framework supports event-sourcing and event-subscription. It consumes events from the Eventing Channel, triggers, and routes it to the Functional instances. If the instance is scaled to Zero, a new instance is started. Azure Serverless framework abstracts all infrastructure and platform complexity from the code, making the function API consume CloudEvent formatted data. The configuration also allows the Azure function to push data back to the Event Channel.

> *CloudEvents, an industry initiative that members of the CNCF Serverless Working Group contribute to, provides a consistent set of metadata to make events easier to work with for publishers, middleware, subscribers, and applications.*
> (*CNCF to host CloudEvents in the Sandbox, by Kristin Evans* 2023)

6.0.1 Three Patterns of Knative Eventing

- Source To Sink: It is a very basic connection between the Event Source to service that receives the event unconditionally (or SINK). There is no support for filtering, queuing, etc.
- Channel and Subscription: Knative defines a Channel and subscribes it to other back-end messaging services like In-Memory, Kafka, etc. for event sourcing. Each message in the channel is in Cloud-Events format. There is no message filtering in the Channel.
- Triggers and Broker: Knative provide a Broker which is an abstraction over underlying Event Channel and Event Triggers. Trigger is responsible for subscription and filtering of events, based on Cloud-Events attributes. Triggers configures a Sink, which is the destination endpoint for the filtered events.

7 Building and Deploying Serverless Function

7.1 Building and Deploying in Azure

Azure SDK and CLI provide templates for creating Azure Functions, giving the choice of programming language and event schema.

Azure CLI archives and publishes the code to the File System under the Function App. The deployed Serverless function will remain in a dormant state (i.e. scaled to zero). The function performs a "Cold Start" when the event is called by Event Triggers.

7.2 Build Automation

There are various option to achieve automation in deployment of Azure Serverless Functions. Source Control integration connects the Function App to a GIT Repository, which will trigger deployment whenever updated code is committed and pushed.

7.3 Build and Deployment in Knative

The Knative CLI offers a build utility which uses the project name and the image registry name to construct and push a fully qualified container image, in OCI format, for the function. The container image has the application's language specific runtime and functional code. Knative Serving deploys the container image as Kubernetes application, pulling the image from registry for first pod initialization.

```
# build Knative functional image and push it to docker repository
func build --registry docker.io/x00159358 \
           --image docker.io/x00159358/invoicemgrimg:latest \
           --push \
           --verbose
```

Azure function do not create a containerized image. Azure CLI provide utility to package the artifacts and archive to Azure File Storage. Azure functional loads the application runtime and code into memory and executes it.

8 Cold Start

Defining Cold Start :
This phenomenon is associated with a delay occurring in provision a runtime container to execute the functions.
Vahidinia, Farahani and Aliee 2020

A cold start in Serverless refers to an increase in latency for Functions which haven't been called recently. The Serverless framework will need to scale its instance from zero to one, before it can process the request. The cold start latency will be passed to request, which need to wait until an becomes fully ready to serve.

The startup delay is influenced by various factors including bootstrapping application runtime, Serverless code and its dependent libraries. This bootstrapping startup delay is also witnessed when Auto-Scalar provisions instances to support the traffic. This initialization time is unavoidable and lasts until the application starts responding. Depending on the application use case, this latency caused by Cold-start can create a request backlogs, which may eventually result in timeout.

Consider a traffic request bust of 3000 in 1 sec. If the cold start delay is 3 sec, there will be a backlog of nearly 9000 requests. Backlog will eventually get cleared after Serverless application is fully started. If the request timeout is 1 sec, nearly 80 percent of the request might be timed out.

8.1 Azure Cold Start

Azure Functions has a longer cold start time, because of its uses of a **Dynamic Provisioning Model**.

> *Dynamic concurrency automatically determines optimal per trigger concurrency settings for your workloads and adjusts as your load patterns change over time.* (Cachai and Gailey 2022)

In short, the function is only created and scaled when needed. This can lead to longer cold start times, especially for functions which is not called frequently.

Serverless offering from cloud vendors like AWS Lambda use Static Provisioning Model, which ensures that a minimum functional instance are always running, reducing the cold start time even for less frequently called Functions.

There are many factors which effect the latency of the cold start when using Cloud Vendors like Azure. Programming Runtime environment plays a significant role in Cold Start of Serverless functions.

The Research study titled "An Investigation of the Impact of Language Runtime on

the Performance and Cost of Serverless Functions", conducted series of tests and analysis measuring Cold and Warm Startup latency of AWS and Azure Serverless functions, using NodeJS and DotNet runtimes.

Test was conducted using :
" There were a total of 144 Cold-Start tests (for both runtimes) over a 6-day period. These were performed at the same 1-hour intervals as in AWS Lambda testing. "

(Jackson and Clynch 2018)

Serverless Platform	Language Runtime	Warm Start Average (ms)	Cold Start Average (ms)
AWS	.NET C#	6.32	2500.09
AWS	NodeJS	11.46	23.67
Azure	.NET C#	0.93	16.84
Azure	NodeJS	4.91	276.42

Table I: Summary of Average Performance Between Azure and AWS

Figure 5: test result showing performance of Serverless startup
(Jackson and Clynch 2018)

8.2 Knative Cold Restart

Knative Serving instructs the Kube API to provision the Serverless application, when it needs to scale up the instances.

There are different distinct stages of execution leading to provisioning of the POD.
Cold Start: When the application is being started for the first time
Warm Disk Start: When the docker image is already cached. The process starts from mounting of image, starting the container, application runtime and pods
Warm Memory Start: The container is resumed from the paused state. The costly, time consuming process of image pull, mounting and starting of container is avoided, thus speeding up the startup time.
Warm CPU start: In this case the container is already running as such the only latency that will apply will be the time application takes to respond to the request. In this is a case application was not scaled to zero.

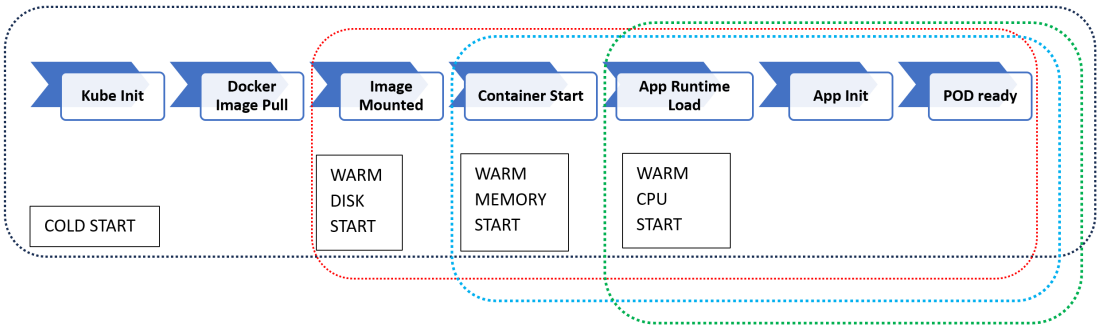


Figure 6: Knative Lifecycle

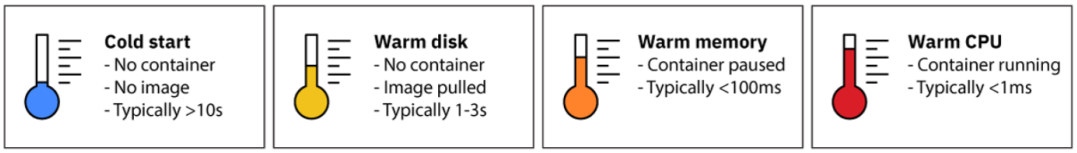


Figure 7: Knative Guage for Cold Start

When a cold start or a warm disk start is used during a scale-from-zero, there is a noticeable delay between when a user makes a request and when the application responds to it, resulting in a poor user experience and possibly other consequences, such as missed service level objectives (SLOs).
In a worst-case scenario, when a cold start is used during a scale-from-zero, it is even possible to encounter client timeouts if the application doesn't respond in time.
(Schweigert and Hadas 2022)

9 Mitigation to Cold startup

9.1 Azure Mitigation to Cold startup

Azure handles this issue by keeping a pool of server in a "warm state", and drawing these servers as "workers" from pool. At any point of time, there are servers in the pool which are idle and pre-configured with Configuration and Functional runtime in a ready and running state. This feature is only available in Azure's Premium Plan.

When execution of function is triggered by incoming traffic:
Azure will allocate a pre-configured server from the pool. The server already has a functional runtime and configuration on it.
After allocation, the server re-configures the functional runtime based on the application requirements.
The Server now resets the Functional runtime and load any required extensions, using the *function.json* file, specified the application manifest.
In the last stage the Function is now loaded into memory and executed. Time take to complete the last stage depends on language used, size of application etc. The function is now in a ready state serving request.

> *"Making these 'pre-warmed sites' happen has given us measurable improvements on our cold start times. Things are now on the order of three to four times faster" - (* Understanding serverless cold start *2018*)

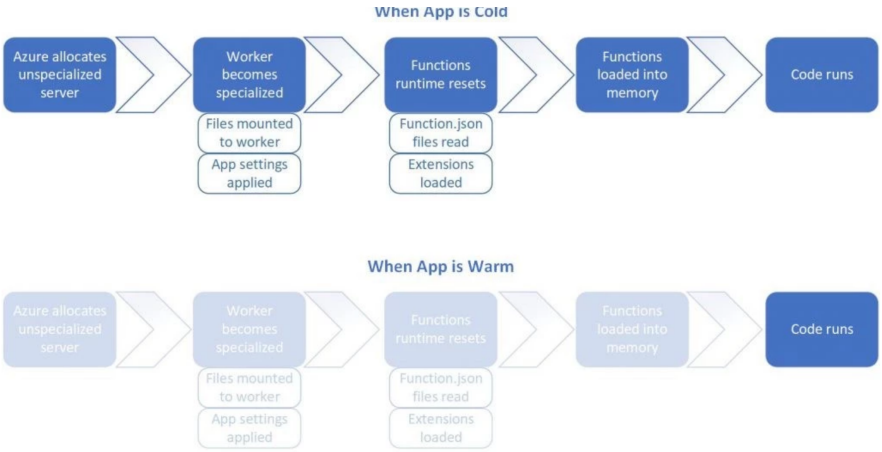


Figure 8: Azure Lifecycle
Ref: *Understanding serverless cold start* 2018

9.2 Knative Mitigation to Cold startup latency

There are a number of ways to reduce Cold Stat-up Latency (scale from zero latency).

- Choose better Runtime and programming language : Some programming language are faster to startup, while other have extra overheads. Optimized runtime can reduce startup time. Java runtime has higher initialization time than other programming languages. As such choosing correct programming languages will contribute in lowering down Cold-Startup time.

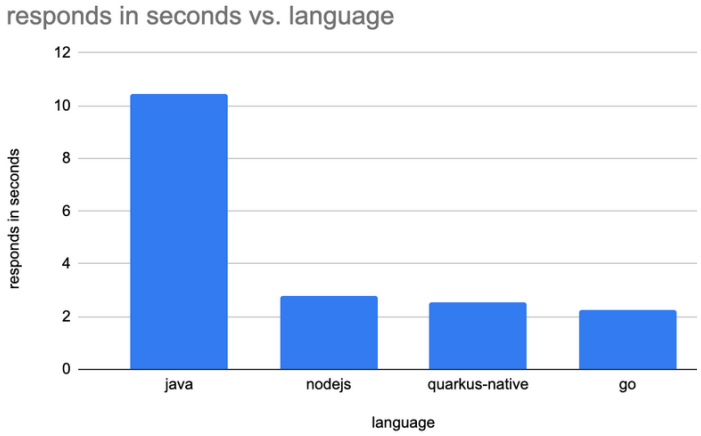


Figure 9: Knative App Runtime Startup
Ref: Schweigert and Hadas 2022

- Speedup image pull. Image Pull effects initial Cold Startup time,because of network latency in downloading container image from repository.
  - Use small size of Base image e.g. "alpine base image".
  - Keep dependencies and additional libraries to base image to bare minimum.
  - Use Stratergy to do a lazy pull of image
  - Avoid POD's image pull policy as "Always"
  - Use specific image digest rather than using the latest tag
  - Cache image to all nodes.
  - try pull image to Kubernetes docker registry through different deployment channel
- Avoid Cold-startup completely: Keep at-least one container to be always in a

running state. This can be adjusted using deployment flag *"min-scale"* to 1. But this options destroys the basis if Serverless concept, where application should be scaled to zero if there is no traffic.

- Reduce the frequency of your application scaling down The deployment flag *"scale-down-delay"*, causes increase the time window before Knative service decides to scale down the container.
- retention period for Scale to zero, increases minimum amount of time for the last pod to remain after a scale-to-zero decision has been made. This is controlled by the deployment flag *"scale-to-zero-pod-retention-period"*.

10 Autoscaling in Knative

Figure 10 explains the full process of Autoscaling in Knative Serving Infrastructure.

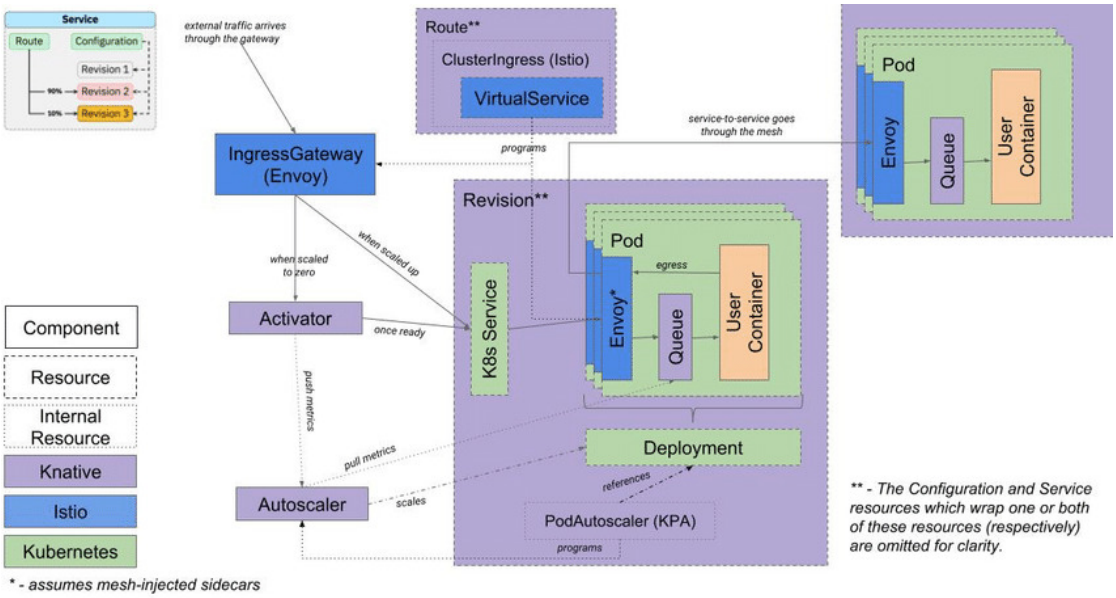


Figure 10: Knative Service Architecture
Kaviani, Kalinin and Maximilien 2019

The interesting case is when scaling from Zero to One:

- Traffic comes to Ingress Gateway, then to the Serverless Services.
- since the Pods are scaled to zero, requests are routed to the Activator. (i.e. Inactive Case).
- Activator passes metrics to Autoscaler which obliged for scaling up the Pods, (from zero to 1).
- Activator calls Knative Serving Deployment to spin up a pod, with all its containers and sidecars.
- Pod's cluster IP is now updated in Serverless Services.
- Once Pods are scaled above zero, Serverless Services updates the Route, disconnecting the Autoscaler. The traffic is now routed directly to Pod's service Proxy.
- The Knative Pod's sidecars and Envoy provide metrics which is now scared to Autoscaler directly, so that it can now decide to need to scale further based on incoming traffic.

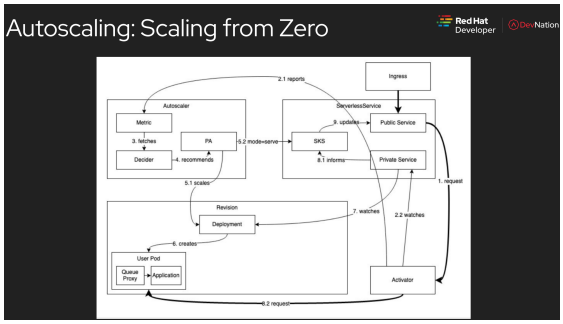


Figure 11: Autoscaler : Cold Start Case, Zero to One

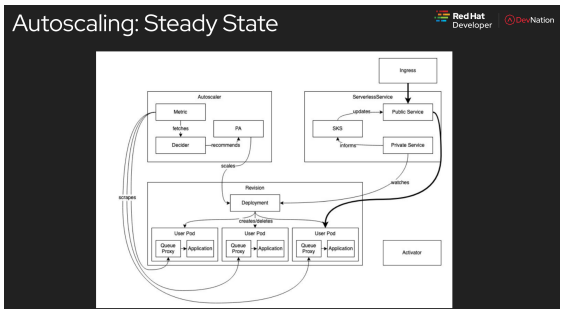


Figure 12: Autoscaler : for Normal Traffic

ref: *Morie 2020*

11 Deployment overhead of Knative

Knative is offered over Kubernetes as set of CRDs and Services. Knative framework includes Knative Serving, Eventing. It will also need a Service Mesh. These are the required components that must be deployed before deploying any Knative Serverless function.

- Service Mesh: It enables secure service-to-service communication and authorization, using sidecars processing secure communication controlled by a central plane. The deployment contains set of CRDs, Services, Gateway components etc.
- Knative Serving: Manages deployment, configuration, revision and routing of the Knative application in Kubernetes.
- Knative Build: The Knative Build framework is read the Function code and manifest, to prepare a container image, pushing it to the Container Registry.
- Knative Eventing: Knative Eventing provides the framework and API to configure and integrate Messaging Channels, Brokers, and Triggers. It facilitates the integration of third-party message brokers like Apache Kafka and Rabbit MQ as event sources.

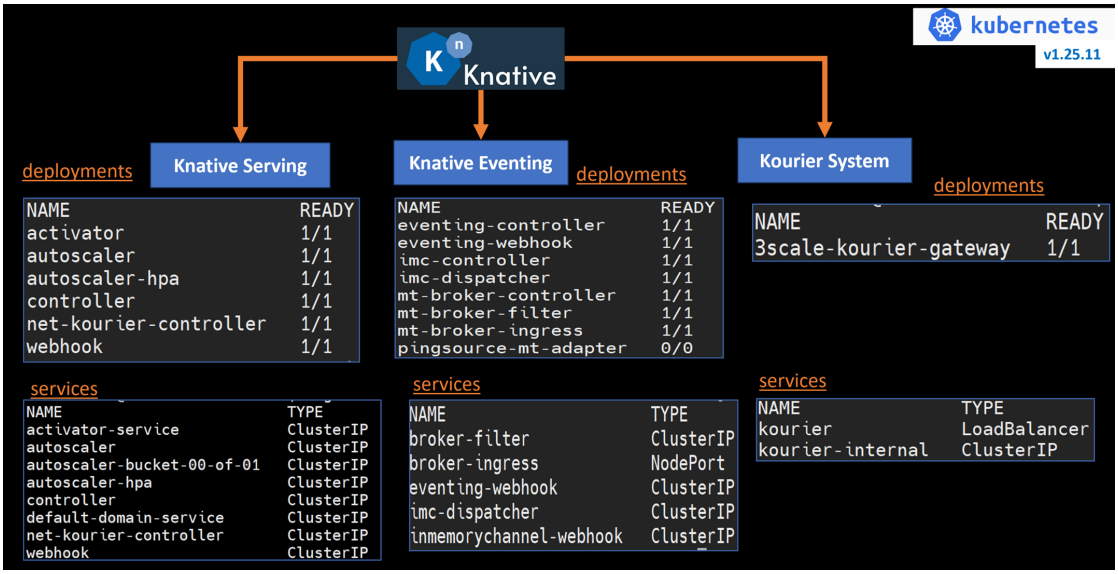


Figure 13: Knative Core Components , PODs and Services

The initial deployment of Knative frameworks and components offers complexity and expert understanding of Kubernetes and Knative components. However, these steps can be considered as One of the Admin jobs, automated using CI jobs. The controlled deployment using the YAML configuration file offers greater control and flexibility in preparing the Serverless Infrastructure.

11.0.1 Higher Infrastructure cost

Since the Knative framework requires many mandatory components and services to be deployed and running, it increased the physical hardware requirement of the Kubernetes Nodes to a minimum of "Standard D2 V2" (, which is 4 Core CPU and 14 GB Ram).

There are at-least 14 different core components, that must be deployed as part of Knative Serving and Eventing. The number of components that will be deployed as part of Knative, will further depend of type of *Service Mesh* deployed.

12 Developing and Event Driven Application

Azure Eventing framework provide a production grade messaging service like Event-Grid and Event-Hubs, which supports easy integration to Azure Functions. Event Grid enables clients to publish and subscribe to messages over the MQTT, HTTP and other protocols. Azure Messaging Service also allows subscription and filtering of events and triggers to registered endpoint or Sink. The Sink can by Azure Serverless Function, Azure Webhook and Services.

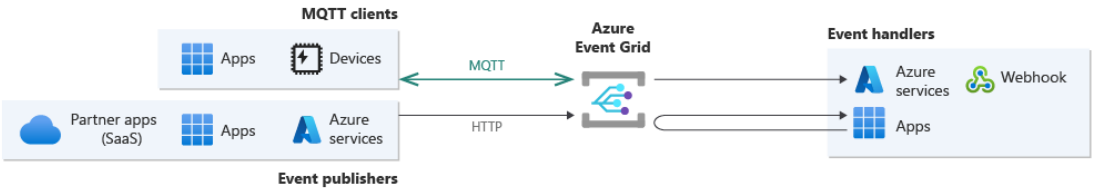


Figure 14: Azure Event Grid

Knative Eventing, already explained in section 5.3, provide the framework to develop an event driven application. Knative eventing provide integration with other third-party messaging frameworks, like Kafka, NATs, etc. It distinguished itself from Azure function by providing a Broker, providing an abstraction over different event channels.
Both Azure and Knative intend to isolate the functional code and configuration from messaging infrastructure. Knative provide event "Brokers", which is an abstraction over different messaging channels.
The Event Triggers in both Azure and Knative, subscribe to events, filters and routes it to the SINK. These subscription are externally configured without impacting the functional code. Messaging formats like Cloud-Events is used to keep the function code independent and portable. Filtering is done on Cloud-Event attributes.

13 Workflow Orchestration

Both Azure Function and Knative provide options to create a Workflow. These workflows are created using Eventing Framework which includes Event Source, Sink, Event Brokers, and Triggers.
The workflow orchestration consumes events from the message broker and triggers events which invokes Serverless function. The Serverless function receives event in CloudEvent format, it can return a transformed event back to the broker or channel for next stage of processing
There are 2 types of Workflows created in Knative

- Parallel
- Sequence

Parallel Workflow, offered by Knative is a set of CRD that defines a set of branches. Each branch contains filters and subscriptions. Knative Workflows creates channels and Subscriptions under the hood. Both Filter and Subscription refer to a Knative Service Function.
Sequence Workflow, offered by Knative is also a set of CRD that defines an order list of functions that must be invoked. Each step creates a filter and transforms events as they are passed. Knative Sequence workflows also create Channels and Subscriptions under the hood.

14 Simple Use-Case

To demonstrate and test the features of Knative and Azure Function, a simple Use Case is created demonstrating an Event-Driven Microservices and Workflow.
The use case starts with generation of an invoice by the clients, making a REST call to the Event Grid or Broker. The Application will consume the event and update the status of invoice as **NEW**. The updated invoice is published back to the eventing system, triggering the other Microservices to consume and register it to the system, republishing the invoice with status as **PENDING**. The client can pay for the pending invoice by POSTing a cloud event with invoice data and status as **PAID**. the Application will now consume the paid invoice and validate it. The result of validation is republished with invoice updated status. If the balance left after payment is zero, the status is updated as **CLOSED**, if the balance is greater then zero, the status is updated back to **PENDING**

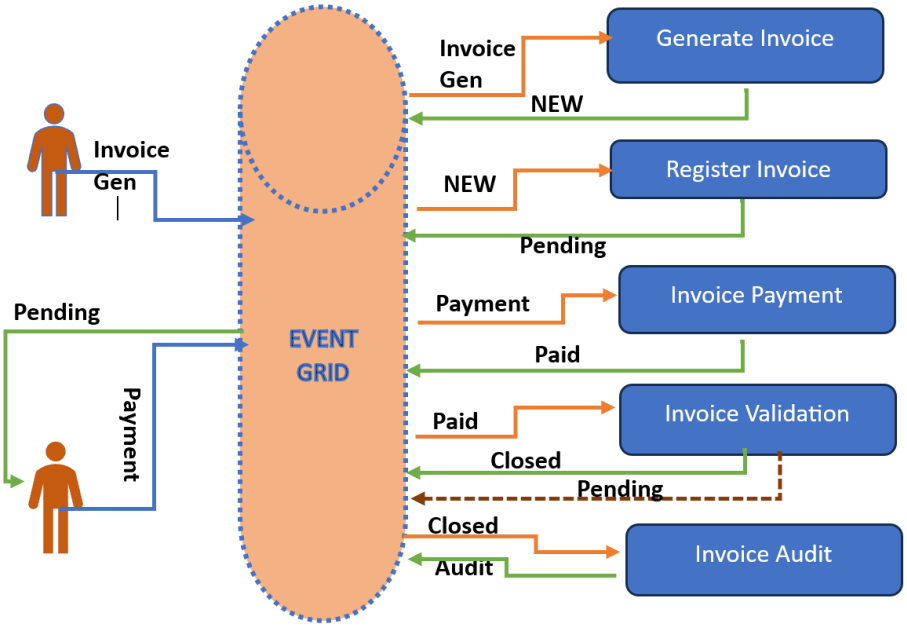


Figure 15: Use Case

All these stages will require unique stateless, Serverless, Event Driven Microservices. The Status and Type of Invoice will act as filter, triggering different Microservices to react.

14.1 Application Code base

We have 2 code bases one for Azure another for Knative. Both involves CloudEvent as intermediate data structure. The Logical code is same for Knative and Azure. Deployment process will be different.
Azure Function:
https://github.com/AINULX00159358/InvoiceManagerAzure
Knative Code base
https://github.com/AINULX00159358/InvoiceMgrKnative
Test Scheduler code base
https://github.com/AINULX00159358/InvoiceMgrTestScheduler

14.2 Application Design

The Use Case for Invoice Management System requires an Event Driven Application Design, where each stage managed and processed by stateless, singular function Microservices.

Invoice status will be updated in every stage. The cloud-event's attribute "Type" will be mapped to Invoice Status. Cloud-Event's attribute "Type" is used for routing and filtering events to specific Microservices, by the configured Trigger.

Event Grid or Knative broker (with event channel) with its Trigger and filter will act as Orchestration of invoice data to different Microservices. Both Event Grid and Knative broker exposes an web accessible URL, allowing client to POST Cloud-Event data using HTTP protocol.

Serverless application, perfectly fits the Microservices requirement. It is stateless, isolated, event driven and only process single function. Their endpoints are exposed and configured in the Event Triggers, with Invoice Status acting as a filter.
Serverless offering from either Knative or Azure can be utilized for this design.

14.2.1 Test Code

A Test Code is deployed as Simple Kubernetes Pod, which will start by POSTing client Invoice Request to the Broker Endpoint at different TPS like 30, 60, 100 etc.

14.3 Execution of Test code

A Test Code is created to request generation of Invoice by making and REST calls to Knative Broker or Azure Event-Grid, using cloudevent schema. The test code is capable of sending Cloudevent at the required rate to simulate regression.

The Test code is packaged and pushed as Container Image, in OCI format, to the Docker repository. The Test application can be deployed as Kubernetes POD or Azure Container Application, with required arguments and properties.

15 Metrics

Both Azure Functions and Knative provide default metrics which is collected and exposed by tools like Prometheus or Application Insight. The application code also provide E2E latency metrics, calculating time taken from generation of invoice to completion of payment.

Azure Portal provides options to view live metrics of Azure Functional App, as shown in figure 16

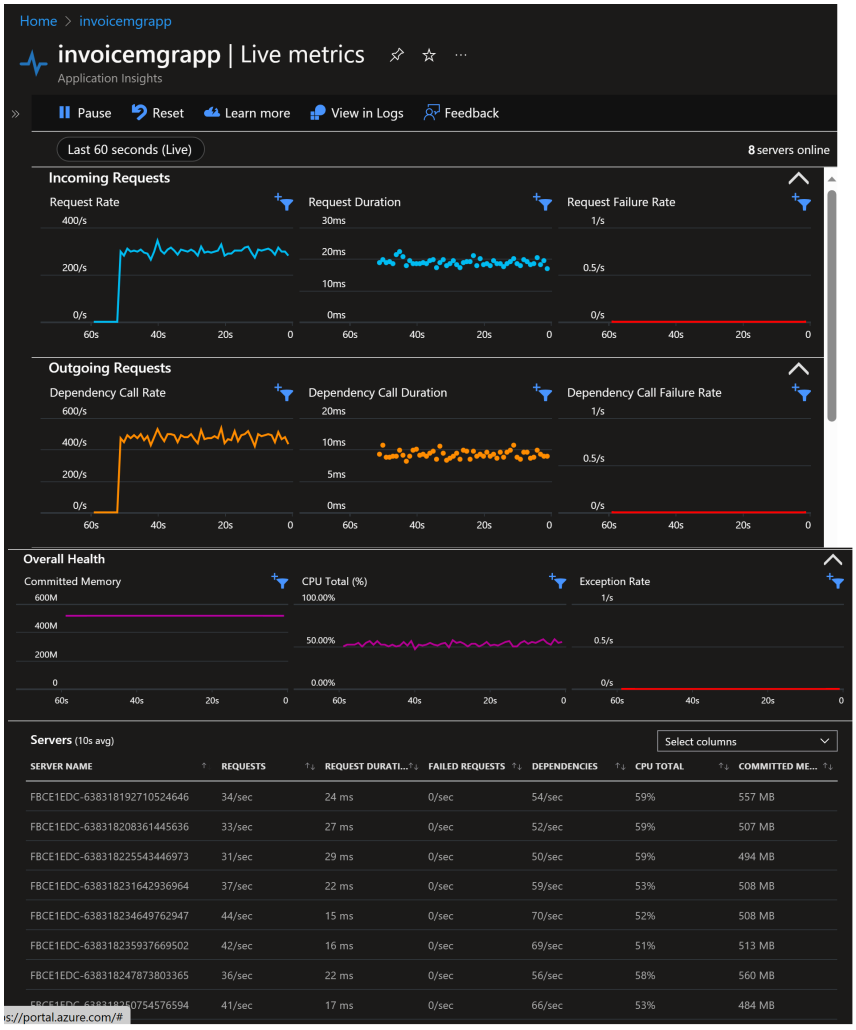


Figure 16: Metrics: Azure Live Metrics

Figure 16 , shows Azure Function App is subjected to average 400 TPS request. There no failure of request. Request latency for each calls was between 15 to 20 milliseconds.
The metrics shows average resource usage for all Serverless containers initialized during the Test.

15.1 Azure Function Cold Start metrics

Since request for generation of invoice was POST'ed to the Event Grid by the Test Code, a view of Event Grid's Topic latency can give some hint on Cold Start latency. This is presented in Figure 17

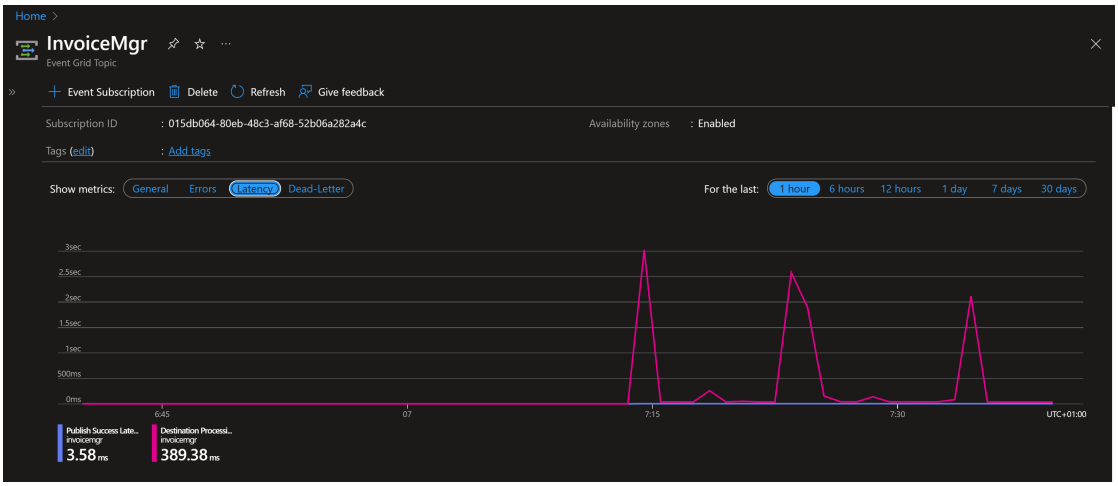


Figure 17: Metrics:Event Grid Latency Metrics

The Max latency was between 2 seconds to 3 seconds, which could be attributed to cold start. This is echoed by the Latency experienced in Knative Function "invoicegenerator" , as shown in figure

15.2 Metrics Recorded for Knative Application

These metrics are captured from Prometheus, These startup metrics is evaluated by taking a delta of time in Seconds when POD is scheduled and achieved a ready state. This gives the "Cold-start" latency.

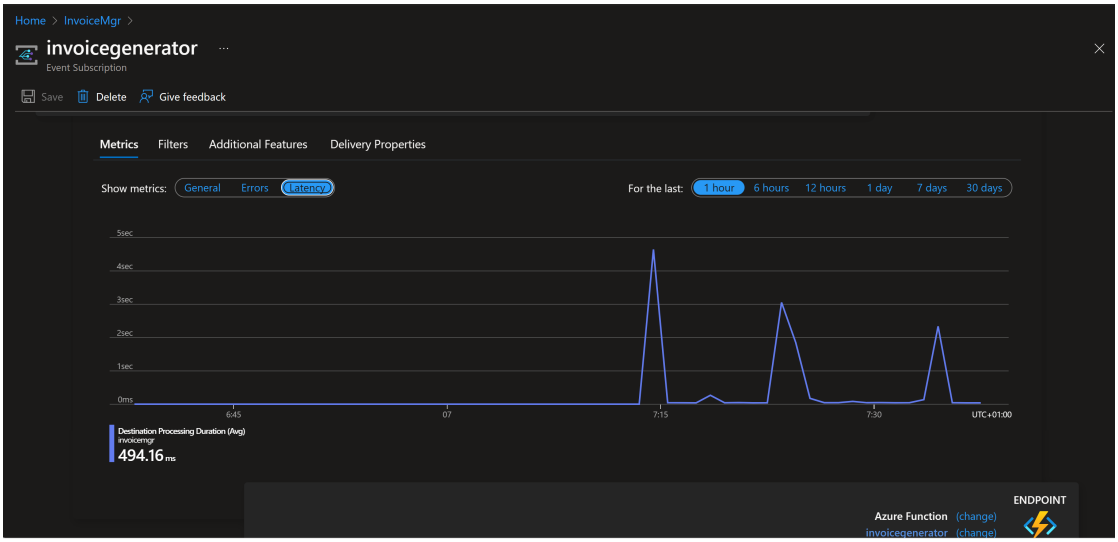


Figure 18: Metrics:invoice generator Latency Metrics

15.3 Metrics: First Initial Deployment

First initial deployment involves pulling docker images from the repository and start the pod. First deployment will also have time spent in creation of services and service-proxy, registration of public endpoints to registry.

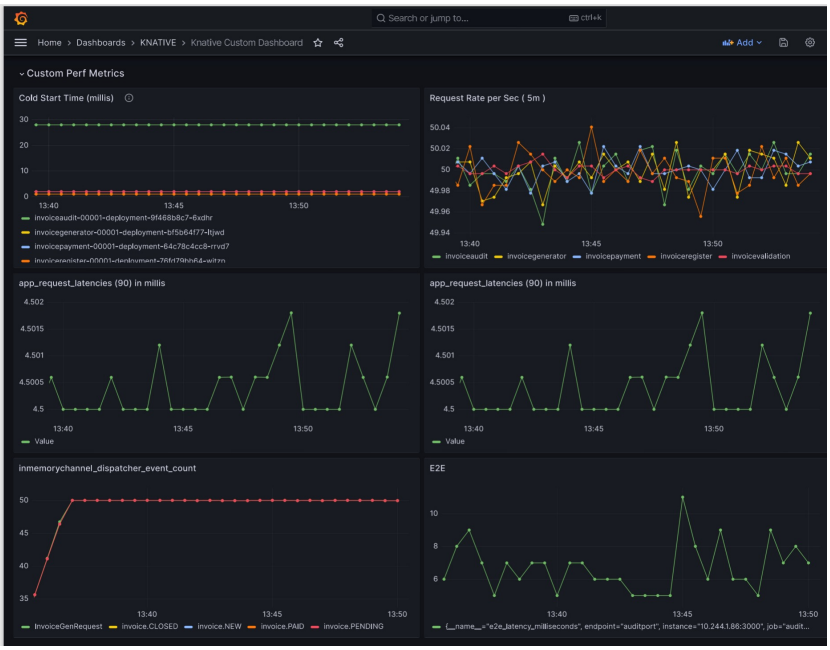


Figure 19: Metrics: First Initial Deployment

Metrics shows higher cold Start time of 30 sec on first initial deployment, because of time spend in pulling image from Repository. Since we are using the same image for all other services, subsequent initialization of services will undergo "Warm Disk Start" with Cold Start time of 1 sec.

15.4 Metrics: Warm Disk Start

Once the image is pulled, subsequent deployment will reuse the image. This stage can be considered as "Warm Disk start".

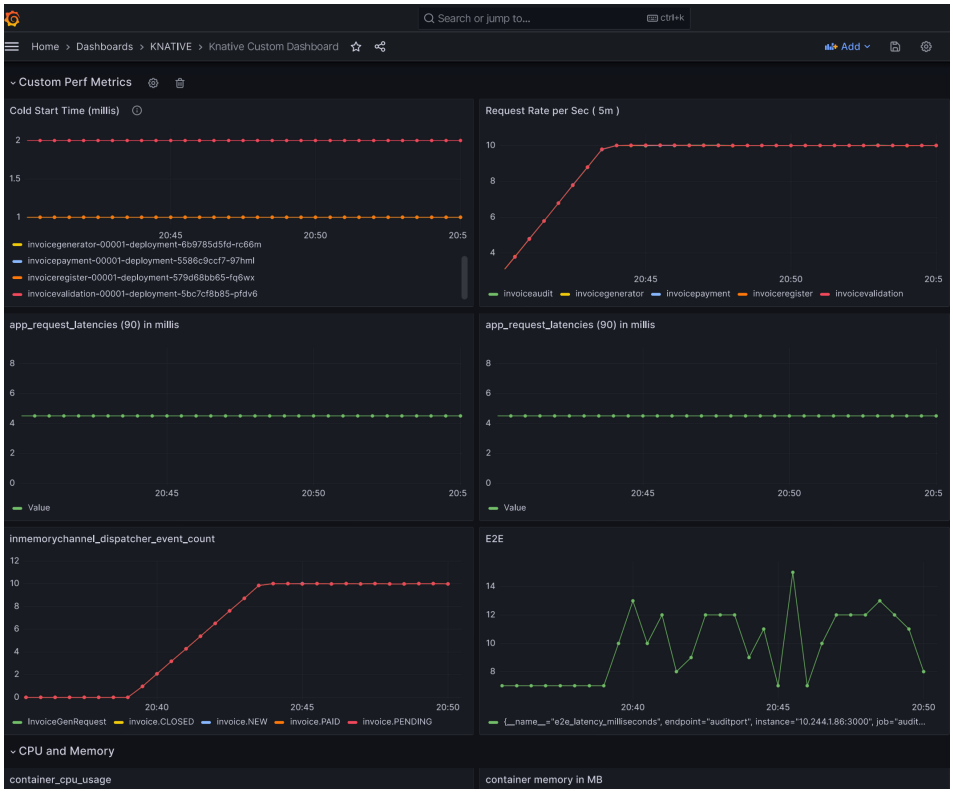


Figure 20: Warm Disk Start

This metrics is collected after sending first Traffic, causing the Pods to Scale from Zero to One. This is the stage of "Warm Memory Start". Application is faster to start avoiding cost of Address registration etc.

Metrics shows reduced cold Start time to one seconds, compared to Cold Start during first deployment, (ref figure 19). Other metrics like application and e2e latency tends to remain consistent over time.

15.5 Metrics: Application Auto-scaling

This metrics is collected after traffic bust, which triggers auto-scaling. Traffic Bust challenged the capacity and capability of Current Kubernetes nodes, triggering scaling of pods.

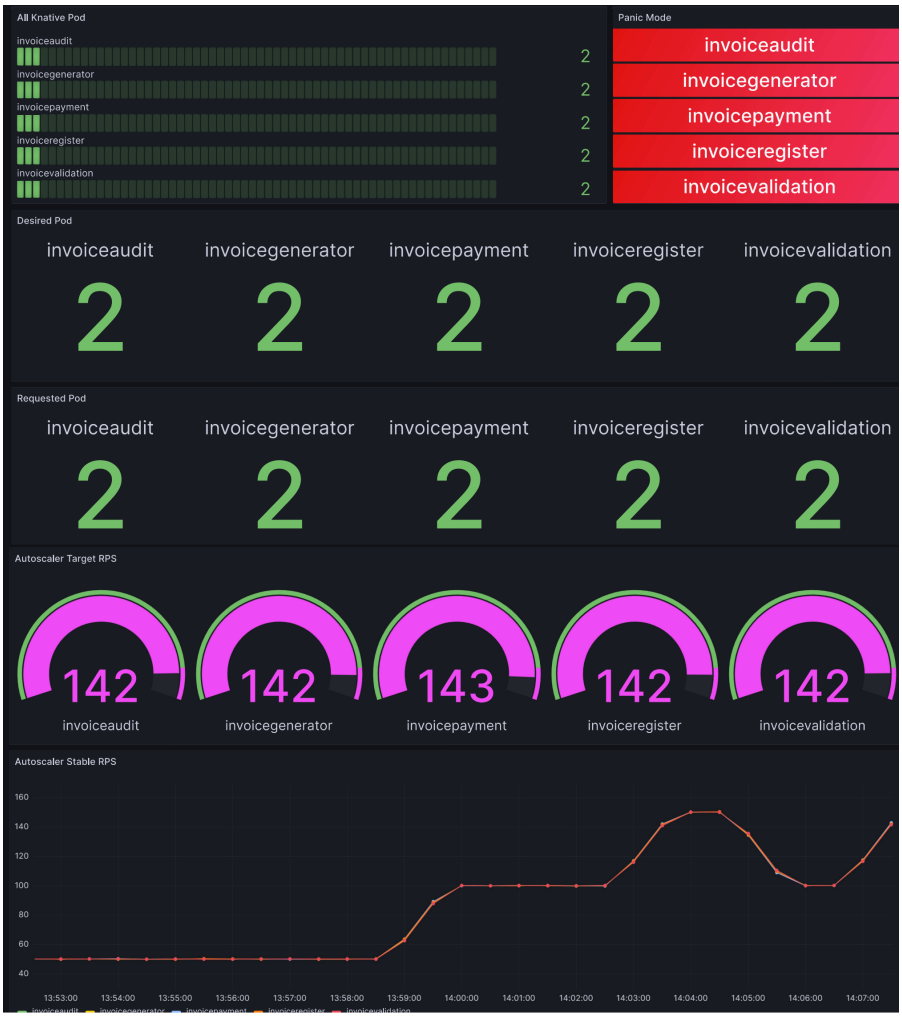


Figure 21: Knative : Pod Auto scaling in Panic mode

The default setting for Autoscaler in Knative is "request concurrency", but for test it was modified to "request per second" (RPS). When the RPS touched the 70% of the threshold value (i.e 200), Autoscaler went in a panic state, requesting Knative Serving to spin more POD(s).

Above figure (22) shows how desired pod for Knative Function

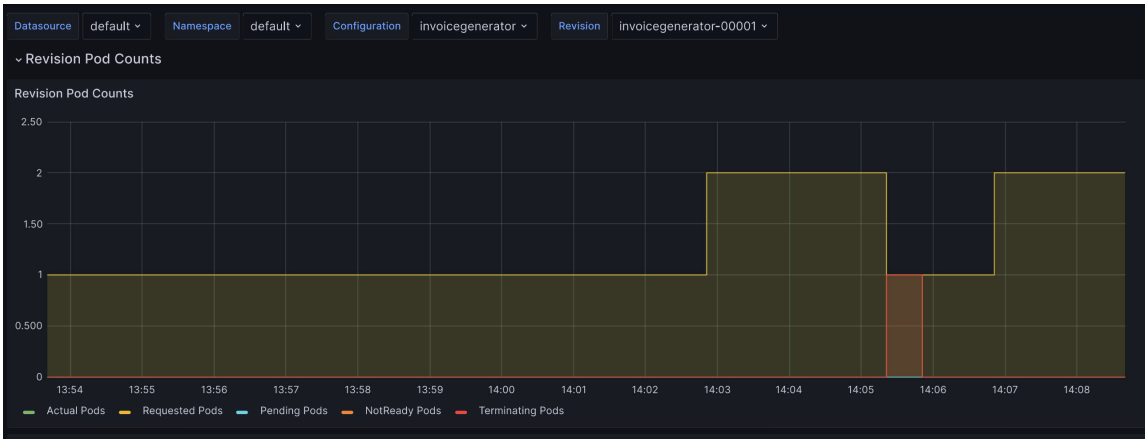


Figure 22: Knative : Pod Auto scaling in Knative Serving mode

(invoicegenerator) increased from one to two, then stabilized back to 1 when traffic normalized. It increased to 2 again when RPS increased back to 140+.

16 Analysis

16.1 Deployment

Knative has an overhead of setting up the Kubernetes cluster, Service Mesh and its required Infrastructure. In production environment a reliable messaging infrastructure like Kafka must be deployed.

Infrastructure cost of setting a proper Knative environment which can support high traffic bandwidth, messaging infrastructure and scalability of Serverless will require provisioning of large capacity nodes.

16.2 Building as Container Image

The Knative code must be converted and pushed as docker image to the container registry.
Kubernetes POD will pull the container image from the registry for initial deployment. Cost of downloading the container image do not effect "Cold Start" as long as Image Pull Policy is set to "IfNotPresent".

In Azure the function build, do not convert the code to a docker image, instead it stores the function files in its file system. The Functional code and runtime is loaded to the Container during execution.

16.3 Azure's Hosting plan

Azure's Basic Serverless Hosting Plan is **"Consumption Plan"**. The Consumption plan scales application on traffic load factor. Customers are charged based on the compute resources utilized. The Compute instances for Functions hosts are dynamically added and removed based on the traffic load factor. Customers are also charged based on function execution. In production it is recommended to use **Premium Plan**, which provided larger physical resources, for higher scalability and traffic bandwidth

16.4 Serverless Costing

16.4.1 Knative

> *Knative is best for applications that generate a variable number of trigger events that, over time, stay within established limits. While a single application can meet this requirement, it's more likely met by a combination of three or more applications whose triggers aren't synchronized.* (Nolle 2020)

When there is a large number of Serverless function, any inflation in trigger caused by high or sudden bust of traffic may exponentially increase Serverless cost due to sudden increase in resource utilization and scaling of instances. This against the traditional container applications, which had fixed number of instances, keeping the processing cost constant.

A proper mix of Knative application, can offer better Serverless performance at lower cost than public cloud. But wrong application or poorly sized resource pool can destroy the benefit.

Knative is based on stateless Microservices approach. Business transaction and data is processed by various steps, involving different stateless Microservices. Since stateless Microservices do not preserve state, so do the transaction context. IT teams need to provide their own means to preserve the context which can add to cost of transaction and latency.

The area of responsibility of SRE increases when using Knative framework. The SRE's are also responsible for containers, Kubernetes, Service Mesh (like Istio) and Knative Eventing Components (like Brokers or channels). SRE's are free from these challenges when using Serverless in public cloud.

If the companies already have Kubernetes and "Service Mesh" in their Technology Stack, Knative is just an added extension on it. So for those companies Knative offers a better and affordable solution than their Serverless public cloud provider alternatives.

16.4.2 Azure Functions

Azure customer using Serverless Functions, under *Consumption Plan*, is billed based on per-second resource consumption and their executions. Most billing is done using "Pay As You Go Model", where customers pay for compute capacity by second, without any long-term commitments or subscriptions. Customer budget is directly proportional to the increase and decrease in their demand and traffic.

Billing for Azure Function Application is subject to the total number of requested processing for all functions in a month. Execution is counted for each time a function is called by the event trigger. There are free exemptions, like 1 million executions per month is free

Customers choose *Premium Plan* for better scalability and no cold-start, with some enhancement in performance. Azure's *Premium plan* for Function App is based on consumption of Core vCPU (in seconds) and memory allocation across the running instances.

Execution charges are exempted under Premium Plan, but it will require at-least one instance to be live and running at all times.

Different pricing tires are available for Premium Plan customers, which provides different combination of vCPU and Memory allocated.

- **V2**: 4 vCPU, 14 GB RAM $0.532/hour
- **V3**: 8 vCPU, 28 GB RAM $1.064/hour
- **V4**: 16 vCPU, 56 GB RAM $2.128/hour
- **V5**: 32 vCPU,112 GB RAM $4.256/hour

Conclusion

Knative is a Cloud-Native and Open-Source alternative to traditional Serverless offered by public cloud providers like Azure Functions.

Knative becomes a better alternative, for companies who have Kubernetes and Service Mesh in their technology stack, and has an active SRE team managing the Kubernetes Cluster.

Azure Function becomes a choice when companies want to avoid the management overhead of Kubernetes Cluster and Service Mesh. In production, it is recommended to use the "Premium plan" to avoid cold start, which comes with added cost to the infrastructure.

To summarize, Both Knative and Azure Function are better solutions for Microservices requiring Serverless computing, but Knative offer is diluted by complexity of managing Kubernetes cluster. Azure Function keeps the customers "vendor locked", but are easily managed because Azure takes the responsibility of underlying infrastructure and resources.

References

ANDERSON, EVAN (2024). *Building serverless applications on Knative: A Guide to designing and writing serverless cloud... applications*. O'REILLY MEDIA.

Cachai and Glenn Gailey (Nov. 2022). *Concurrency in azure functions*. en-us. URL: https://learn.microsoft.com/en-gb/azure/azure-functions/functions-concurrency#dynamic-concurrency-preview.

CNCF to host CloudEvents in the Sandbox, by Kristin Evans (May 2023). URL: https://www.cncf.io/blog/2018/05/22/cloudevents-in-the-sandbox/.

Global Market Insights Inc. Report ID: GMI3796 (Dec. 2022). URL: https://www.gminsights.com/industry-analysis/serverless-architecture-market.

Jackson, David and Gary Clynch (2018). 'An Investigation of the Impact of Language Runtime on the Performance and Cost of Serverless Functions'. In: *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pp. 154–160. DOI: 10.1109/UCC-Companion.2018.00050.

Kaviani, Nima, Dmitriy Kalinin and Michael Maximilien (Oct. 2019). 'Towards Serverless as Commodity: a case of Knative'. In: ISBN: 978-1-4503-7038-7. DOI: 10.1145/3366623.3368135.

Killalea, Tom (May 2016). 'The Hidden Dividends of Microservices: Microservices Aren't for Every Company, and the Journey Isn't Easy.' In: *Queue* 14.3, pp. 25–34. ISSN: 1542-7730. DOI: 10.1145/2956641.2956643. URL: https://doi.org/10.1145/2956641.2956643.

Lin, Ping-Min and Alex Glikson (2019). 'Mitigating cold starts in serverless platforms: A pool-based ap-proach'. In: *arXiv preprint arXiv:1903.12221*.

Morie, Paul (Nov. 2020). *Autoscaler knative backstage with the - Red Hat Developer*. URL: https://developers.redhat.com/sites/default/files/2020-09/DevNation%20Day%20-%20Knative%20Backstage%20w_%20Autoscaler.pdf.

Nolle, Tom (July 2020). *An overview of knative use cases, benefits and challenges: TechTarget*. URL: https://www.techtarget.com/searchitoperations/tip/An-overview-of-Knative-use-cases-benefits-and-challenges.

Opara-Martins, Justice, Reza Sahandi and Feng Tian (2014). 'Critical review of vendor lock-in and its impact on adoption of cloud computing'. In: *International Conference on Information Society (i-Society 2014)*, pp. 92–97. DOI: 10.1109/i-Society.2014.7009018.

Schweigert, Paul and David Hadas (Sept. 2022). URL: https://developer.ibm.com/articles/reducing-cold-start-times-in-knative/.

Serverless Architectures by Mike Roberts (May 2018). URL: https://martinfowler.com/articles/serverless.html.

Sutter, Burr and Kamesh Sampath (2020). *Knative Cookbook: Building effective serverless applications with Kubernetes and OpenShift*. O'Reilly Media.

Understanding serverless cold start (Feb. 2018). URL: https://azure.microsoft.com/en-us/blog/understanding-serverless-cold-start.

Vahidinia, Parichehr, Bahar Farahani and Fereidoon Shams Aliee (2020). 'Cold start in serverless computing: Current trends and mitigation strategies'. In: *2020 International Conference on Omni-layer Intelligent Systems (COINS)*. IEEE, pp. 1–7.

Wikipedia : Function as a service (Sept. 2023). URL: https://en.wikipedia.org/w/index.php?title=Function_as_a_service&oldid=1166791869.