IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 19, 2021, accepted January 8, 2022, date of publication January 18, 2022, date of current version January 26, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3144078

TppFaaS: Modeling Serverless Functions Invocations via Temporal Point Processes

MARKUS STEINBACH[1], ANSHUL JINDAL[1], MOHAK CHADHA[1], MICHAEL GERNDT[1], AND SHAJULIN BENEDICT[2], (Senior Member, IEEE)

[1]Chair of Computer Architecture and Parallel Systems, Technical University of Munich, 85748 Garching, Germany
[2]Department of Computer Science and Engineering, Indian Institute of Information Technology, Kottayam, Kerala 686635, India

Corresponding author: Anshul Jindal (anshul.jindal@tum.de)

This work was supported in part by the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus Program, in part by the Technical University of Munich (TUM) in the framework of the Open Access Publishing Program, and in part by the Google Cloud by the Google Cloud Research Credits Program under Award NH93G06K20KDXH9U.

ABSTRACT Serverless computing is a cloud computing paradigm that allows developers to focus exclusively on business logic as cloud service providers manage resource management tasks. Serverless applications based on this model are often composed of several fine-grained and ephemeral Function-as-a-Service (FaaS) functions that implement complex business processes via mutual interaction and interaction with Backend-as-a-Services (BaaS) such as databases. FaaS functions suffer from the cold start problem because of the scale to zero instances feature. In this work, we use neural Temporal Point Processes (TPPs) to model function invocations in FaaS compositions. A probability distribution over the time and class of the following invocations for a given history of invocations is predicted using these probabilistic models. The prediction can avoid cold starts by scaling functions in advance and reduce network load by optimizing the function-server assignment. In this regard, we developed a python-based tool called *TppFaaS* on top of OpenWhisk open-source serverless platform. TppFaaS uses the neural TPPs LogNormMix for modeling the time using a log-normal mixture distribution and TruncNorm for predicting a single value for the time. Furthermore, we built a custom trace data collector for OpenWhisk embedded into TppFaaS and created datasets for multiple FaaS compositions to train and test our models. For datasets without cold starts, the models achieved for most compositions a mean absolute error below 22ms and a percentage of correctly predicted function classes above 94%.

INDEX TERMS Cloud computing, faas, faas composition, function-as-a-service, modeling, serverless computing, temporal point process.

I. INTRODUCTION

With the advent of Amazon Web Services (AWS) Lambda in 2014, serverless computing has gained popularity and more adoption in different application domains such as machine learning [1], [2], linear algebra computation [3], [4], and map/reduce-style jobs [5]. Furthermore, nowadays, it is implemented by every major cloud provider in services like Azure Functions [6] and Google Cloud Functions [7]. Function-as-a-Service (FaaS), a key enabler of serverless computing, allows a traditional application to be decomposed into fine-grained, stateless, and ephemeral functions running isolated in containers with a runtime on a FaaS platform [8]. A FaaS platform is responsible for providing resources, such as the containers, and auto-scaling the functions on demand. Functions are triggered in an event-based fashion, with various sources emitting the events. These include cloud services such as databases or message queues, as well as other FaaS functions of the application emitting events for database or queue updates, as well as HTTP requests [9]. Since FaaS offerings such as AWS Lambda and Azure Functions are based on a pay-per-use pricing policy, running an application on a FaaS platform can therefore reduce costs [10].

A FaaS application is often constructed as a composition of multiple functions that abstracts some business process [11]. An example of such a composition can be seen in Figure 1, in which multiple FaaS functions implement a webshop [12]. In it, each function fulfills a simple modular logic, with the interaction between functions enabling a complex program.

The associate editor coordinating the review of this manuscript and approving it for publication was Sudipta Roy.

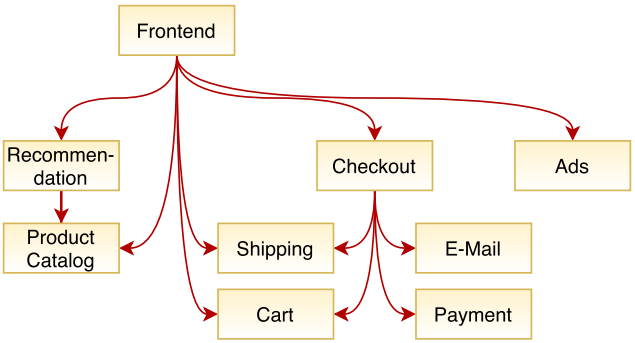


FIGURE 1. A webshop implemented as a composition of FaaS functions [12].

Orchestration tools such as AWS Step Functions [13], Azure Durable Functions [14], or OpenWhisk's Composer [15] facilitate building such compositions. These provide constructs to compose the functions into a control flow, known from any imperative programming language. That is, a developer can arrange the functions sequentially, in parallel, or loops and integrate branching and conditional logic. In addition, the function orchestrator performs other important tasks such as state management, i.e., storing the data communicated between functions, error handling, real-time monitoring, logging, and much more [16]. With all the described characteristics, the migration to FaaS offers an attractive opportunity to break up traditional monolithic applications into a composition of fine-grained and reusable functions that scale independently and automatically and can be arranged in a familiar imperative manner.

Despite having many advantages, serverless computing suffers from some pain points that obstruct its wide adoption [17], [18]. We explain two of them in the following subsections:

A. COLD START PROBLEM

It is mainly connected with loading the FaaS function into the main memory of the executing server and preparing the execution environment for the target code (starting up the VM/container, loading libraries, loading function code, etc. forming the initialization time in Figure 2) [19], [20]. Several influencing factors increase the initialization time of a function [21], [22], one of which is the choice of programming language. While languages such as JavaScript use an interpreter, Java requires a more complex JVM to be set up in the container, leading to higher latency. Also, the size of the function image has a decisive influence on the cold start latency.

B. DATA-SHIPPING ARCHITECTURE

Also, critical in FaaS is the *data-shipping architecture* criticized in [23]. In FaaS, functions are executed in containers isolated from the data they need. In addition, the functions are short-lived, so caching the state to serve multiple requests is limited. Instead, the state is stored in databases that must be queried regularly. Consequently, the data must be transported over the network to the function's location (shipping data to code). However, shipping code to data would be much more efficient. This anti-pattern in FaaS leads to higher latencies, load on the network, and thus higher costs [23].

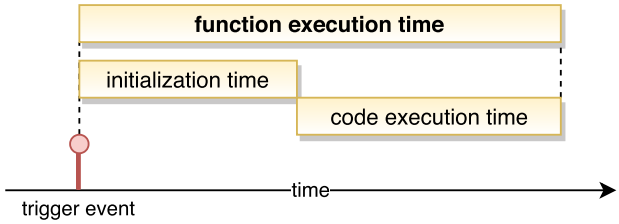


FIGURE 2. A cold start delays the execution of the function code.

A Temporal Point Process (TPP) is a probability distribution over sequences of instantaneous points in time, denoted as events, of variable length in an interval $[0, T]$ [24]. Since FaaS follows an event-based execution model, we can model the events triggering the functions using TPPs. Therefore, TPPs are perfect for modeling invocations in FaaS function compositions by representing an executed function composition by a sequence of events. If we consider that a composition can contain structures such as loops, branches, and conditions, the length of the sequence is also variable. Such modeling of FaaS function compositions and then a prediction can avoid cold starts by scaling functions in advance and reducing network load by optimizing the function-server assignment. Furthermore, it can also help in optimizing the data-function placement. In this regard, this work focuses on modeling FaaS applications in the form of function compositions using neural temporal point processes (TPPs). The key contributions of this work are as follows:

- We present a python-based tool called *TppFaaS*[1] on top of OpenWhisk open-source serverless platform for modeling Serverless Functions Invocations via Temporal Point Processes. *TppFaaS* uses the neural TPPs LogNormMix for modeling the time using a log-normal mixture distribution and TruncNorm for predicting a single value for the time.
- We constructed four FaaS compositions with *TppFaaS* having different characteristics for evaluation of *TppFaaS*. In particular:
  1) the constructed applications exhibit different structural characteristics (sequence, parallel, tree and fanout).
  2) each of the applications is scaled in two variants: small variant and large variant.
  3) for each variant of the application, we implement a randomized and a non-randomized variant. In the non-randomized variant, the duration of the sleep command for all functions is fixed with either 300, 400, or 500ms. In the randomized variant, the

[1] https://github.com/maSteinbach/TppFaaS

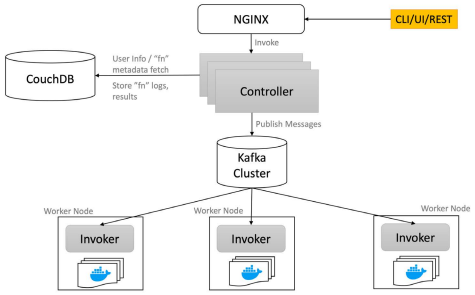


FIGURE 3. Openwhisk high-level workflow [28].

duration is drawn from a gamma distribution for each function invocation (§IV-A).

- We evaluate the prediction performance of LogNormMix and TruncNorm with multiple metrics using the generated datasets without cold starts (§VI-A) and with cold starts (§VI-B).

1) PAPER ORGANIZATION

Section II gives a high level workflow of the OpenWhisk FaaS platform used in this work. We study the theory of TPPs and introduce basic models such as the Hawkes process, and neural TPPs such as introduce RMTPP and LogNormMix in Section III. Our methodology and developed tool *TppFaaS* and its components are described in Section IV. Section V describes the various evaluation settings used in this work, training models hyperparameters, performance quality evaluation metrics and the benchmark applications used in this work. In Section VI, our evaluation results on the introduced performance quality metrics are presented. Section VII describes some prior works in this domain. Finally, Section VIII concludes the paper and presents an outlook.

II. FaaS PLATFORM–OPENWHISK

In this section, we present high level workflow of the OpenWhisk FaaS platform used in this work. OpenWhisk [25] is an open-source FaaS platform developed by IBM that is built on Kubernetes, which provides containers for function invocations. It is also the platform that leverages IBM's FaaS offering IBM Cloud Functions [26]. In OpenWhisk's terminology, a function is called action and an invocation is called activation. Actions can be created using OpenWhisk's CLI, SDK, or UI and invoked using the same tools as well as by event triggers [27].

The procedure of a FaaS function invocation in OpenWhisk starts with an HTTP request entering the OpenWhisk system (shown in Figure 3) through Ngnix [29], an HTTP and reverse proxy server whose primary purpose here is providing the HTTPS protocol. The Ngnix server immediately forwards the request to the controller, which is the system's central component and provides a REST API for creating entities such as actions and for the invocation of them. Since the forwarded request is a request for an invocation, the controller performs authentication and authorization, i.e., it checks whether the user of the request has the privilege to invoke the desired action. To do this, the controller queries the OpenWhisk database CouchDB, where all the users' authorizations are stored. If the authorization was successful, the controller fetches the actual function code from CouchDB along with the default parameters of the action. The default parameters are merged with the dynamic parameters attached to the request. A load balancer integrated into the controller has a global view of the availability of the Invokers and selects one of them to execute the function code. The controller communicates with the invokers via Kafka [30], a distributed publish-subscribe messaging system. Therefore, the controller publishes a message to Kafka addressed to the selected invoker containing the action and its parameters. Kafka ensures the persistence of the message even in case of a system crash. It also buffers the message if the system is under heavy load, in which case the message must wait for other messages to be executed. After Kafka receives the message, it returns the unique *ActivationId* to the user, which can be used to retrieve the invocation's result and meta information from the OpenWhisk API. This immediate termination of the HTTP request after Kafka receives the message describes an asynchronous behavior. OpenWhisk also provides a synchronous behavior where the client is blocked until the invocation is finished. In this case, the complete result of the invocation is returned, rather than just the ActivationId. After the invoker has executed the action, the result is written to CouchDB along with other meta information and logs [28].

OpenWhisk records the invocation's initTime and waitTime in the meta-information. The initTime is only present in case of a cold start and describes the time required for the function initialization. The waitTime describes the time that elapsed from the receipt of the invocation request by the controller to the provision of a container for execution by the invoker. Therefore, the waitTime increases when the system is under heavy load, and the message must wait in the Kafka queue [27].

III. TEMPORAL POINT PROCESSES

A Temporal Point Process (TPP) is a probability distribution over sequences of instantaneous points in time, denoted as events, of variable length in an interval $[0, T]$ [24]. These events are discrete events in continuous time. Discrete means that events can be categorized into classes, often referred to as event type or mark in the literature [31]. A realization of a marked TPP model can be represented as an event sequence $x = \{(t_1, m_1), \ldots, (t_N, m_N)\}$, where $0 < t_1 < \cdots < t_N < T$ represents event's time (see Table 1) with N being the number of events and is itself a random variable, and m_i represents an event type or a mark. In most applications, the marks $(m_1, \ldots, m_N)$ are categorical, such that $m_i = \{1, \ldots, K\}$, although other representations are possible for this. Furthermore, a TPP can also be represented by a list of strictly positive inter-event times $\tau_i = t_i - t_{i-1} \in \mathbb{R}_+$, where

TABLE 1. Symbols and definitions used in this paper.

Symbol	Interpretation
x	sequence of events
N	number of events in the given event sequence x
$t_1, \ldots, t_N$	event's occurrence times
$m_1, \ldots, m_N$	event types (or marks as referred in the literature) at different times
τ_i	inter event time ($t_i - t_{i-1}$)
$\mathcal{H}(t)$	the history of past events for a given event sequence x
$f_i^*(t_i)$	conditional probability density function for modeling the event times of a TPP model
$F_i^*(t_i)$	cumulative distribution function for modeling the event times of a TPP model
$S_i^*(t_i)$	complementary cumulative distribution function also known as survival function for modeling the event times of a TPP model
$\lambda^*(t)$	conditional intensity function for modeling the event times of a TPP model
$\phi_i^*(t)$	*hazard function* for characterizing a TPP model
μ	constant event rate in homogenous Poisson process
$\kappa(t)$	kernel function in the Hawkes process for modeling the dependence on previous events

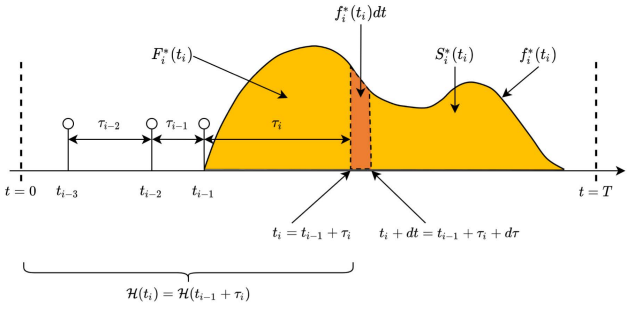


FIGURE 4. The conditional probability density function $f_i^*(t_i)$, the cumulative distribution function $F_i^*(t_i)$, and the survival function $S_i^*(t_i)$ model the time of the next event t_i for a given event history $\mathcal{H}(t_i)$ for a TPP model [32].

$t_0 = 0$ and $t_{N+1} = T$. Both notations are equivalent and can be replaced with each other as desired. Finally, $\mathcal{H}(t) = \{(t_j, m_j) | t_j < t\}$ defines the history of past events for a given event sequence x.

Each event time t_i is a random variable, which is modeled in an autoregressive fashion by the TPP model, i.e., conditioned on past events defined by the history $\mathcal{H}(t_i) = \{t_1, \ldots, t_{i-1}\}$. Modeling t_i is equivalent to modeling the inter-event time τ_i for a given $\mathcal{H}(t_i) = \mathcal{H}(t_{i-1} + \tau_i)$. For the sake of simplicity, in the following subsections we consider an unmarked TPP such that $x = \{t_1, \ldots, t_N\}$. The modeled distribution of t_i and τ_i, respectively, can be characterized for a given $\mathcal{H}(t_i)$ by one of the following three functions, also illustrated in Figure 4:

- The conditional probability density function $f_i^*(t_i) = f_i(t_i|\mathcal{H}(t_i))$ determines the probability that the next event for a given history $\mathcal{H}(t_i)$ occurs in the interval $[t_i, t_i + dt)$. Similarly, the conditional density function $f_i^*(\tau_i) = f_i(\tau_i|\mathcal{H}(t_i))$ defines the probability, that the time until the next event for a given history $\mathcal{H}(t_i)$ is within the interval $[\tau_i, \tau_i + d\tau)$.
- The cumulative distribution function $F_i^*(t_i) = F_i(t_i|\mathcal{H}(t_i)) = \int_{t_{i-1}}^{t_i} f_i^*(u)\,du$ determines the probability that the next event for a given history $\mathcal{H}(t_i)$ occurs before t_i. Similarly, the cumulative distribution function $F_i^*(\tau_i) = F_i(\tau_i|\mathcal{H}(t_i)) = \int_0^{\tau_i} f_i^*(t_{i-1} + u)\,du$ is the probability that the time to the next event for a given history $\mathcal{H}(t_i)$ is less than τ_i.
- The complementary cumulative distribution function $S_i^*(t_i) = S_i(t_i|\mathcal{H}(t_i)) = 1 - F_i^*(t_i) = \int_{t_i}^{\infty} f_i^*(u)\,du$, also known as survival function, defines the probability that the next event for a given history $\mathcal{H}(t_i)$ occurs after t_i. Similarly, the complementary cumulative distribution function $S_i^*(\tau_i) = S_i(\tau_i|\mathcal{H}(t_i)) = 1 - F_i^*(\tau_i) = \int_{\tau_i}^{\infty} f_i^*(t_{i-1} + u)\,du$ is the probability that the time to the next event for a given history $\mathcal{H}(t_i)$ is greater than τ_i [32], [24].

Any of the functions f_i^*, F_i^*, and S_i^* can be used to model the distribution of t_i or τ_i. If one of the functions is known, the other two can be derived from it [33]. There are many other functions which can be used to model the distribution of t_i or τ_i, but a prominent one from the literature is the conditional intensity function $\lambda^*(t)$, which is often used in the literature to describe TPP models.

Conditional intensity function $\lambda^*(t) = \lambda(t|\mathcal{H}(t))$, another way to model the event times of a TPP model, indicates the probability of the next event occurring in the interval $[t, t+dt)$ conditioned on no event to have occurred in $[t_{i-1}, t)$, where t_{i-1} is the time of the last event occurring before t [32]. Formally, this means:

$$\begin{aligned}\lambda^*(t)dt &= \mathbb{P}(\text{event in } [t, t+dt) | \text{no event in } [t_{i-1}, t), \mathcal{H}(t)) \\ &= \frac{\mathbb{P}(\text{event in } [t, t+dt) \,\&\, \text{no event in } [t_{i-1}, t) | \mathcal{H}(t))}{\mathbb{P}(\text{no event in } [t_{i-1}, t) | \mathcal{H}(t))} \\ &= \frac{\mathbb{P}(\text{next event in } [t, t+dt) | \mathcal{H}(t))}{\mathbb{P}(\text{no event in } [t_{i-1}, t) | \mathcal{H}(t))} \\ &= \frac{f_i^*(t)dt}{S_i(t)}. \end{aligned} \tag{1}$$

For a better interpretation of the conditional intensity function, we consider an alternative representation of the TPP model in which it is defined as a counting process $N(t)$, counting the number of events up to time t. For an infinitesimally time interval dt it holds that $dN(t) = N(t + dt) - N(t) \in \{0, 1\}$, meaning that at most one event can occur in $[t, t + dt)$ [32]. From this follows:

$$\begin{aligned}\mathbb{E}[dN(t)|\mathcal{H}(t)] &= 1 * \mathbb{P}(\text{next event in } [t, t+dt) | \mathcal{H}(t)) \\ &\quad + 0 * \mathbb{P}(\text{no event in } [t, t+dt) | \mathcal{H}(t)) \\ &= \lambda^*(t)dt. \end{aligned} \tag{2}$$

If the equation (2) is rearranged, the result is equation (3).

$$\lambda^*(t) = \lim_{dt \to 0} \frac{\mathbb{E}[dN(t)|\mathcal{H}(t)]}{dt}, \tag{3}$$

From equation (3), we derive that the conditional intensity function specifies the expected number of events per time unit [33], that is, the frequency rate per time unit, i.e.,

$\lambda^*(t) = events/second$. The intuitive interpretation facilitates the construction of TPP models with desired characteristics by specifying the functional form of $\lambda^*(t)$. When choosing the functional form of $\lambda^*(t)$, the only constraint is that for any t and $\mathcal{H}(t)$, the two terms $\lambda^*(t) \geq 0$ and $\int_t^\infty \lambda^*(u)\,du = \infty$ must be satisfied. In contrast, the conditional probability density function $f_i^*(t)$ must be specified as a valid probability distribution, such that $\int_{t_{i-1}}^\infty f_i^*(u)\,du = 1$ is satisfied [32], [33].

If the conditional intensity function $\lambda^*(t)$ is given, the conditional probability density function $f_i^*(t)$ can be derived from it. From the definition of the survival function $S_i^*(t)$ we know that $S_i^*(t) = 1 - F_i^*(t)$, thus

$$\begin{aligned} \frac{dS_i^*(t)}{dt} &= \frac{d}{dt}(1 - F_i^*(t)) \\ \iff -\frac{dS_i^*(t)}{dt} &= f_i^*(t). \end{aligned} \tag{4}$$

Plugging equation (4) into (1) then yields

$$\lambda^*(t) = \frac{f_i^*(t)}{S_i^*(t)} = -\frac{1}{S_i^*(t)}\frac{dS_i^*(t)}{dt} = -\frac{d\log S_i^*(t)}{dt}. \tag{5}$$

The integration of both sides of equation (5) leads to

$$\begin{aligned} \log S_i^*(t) &= -\int_{t_{i-1}}^t \lambda^*(u)\,du \\ \iff S_i^*(t) &= \exp(-\int_{t_{i-1}}^t \lambda^*(u)\,du). \end{aligned} \tag{6}$$

The derived equation for the survival function from (6) is plugged into (1), leading eventually to the formula for the conditional probability density function [32]

$$f_i^*(t) = \lambda^*(t)\exp(-\int_{t_{i-1}}^t \lambda^*(u)\,du). \tag{7}$$

Furthermore, we introduce the *hazard function* $\phi_i^*(t) = \phi_i(t|\mathcal{H}(t))$, another function to characterize a TPP and which is related to the conditional intensity function $\lambda^*(t)$ [24], [33]. While $\lambda^*(t)$ describes the global intensity in the time interval $[0, T]$, the hazard function $\phi_i^*(t)$ is limited to the time interval between two events $(t_{i-1}, t_i]$, which is why the index i is required. That is, for a sequence of N events, we obtain the global intensity $\lambda^*(t)$ by concatenating the hazard functions $\phi_1^*, \phi_2^*, \ldots, \phi_{N+1}^*$, i.e.,

$$\lambda^*(t) = \begin{cases} \phi_1^*(t) & \text{if } 0 \leq t \leq t_1 \\ \phi_2^*(t - t_1) & \text{if } t_1 < t \leq t_2 \\ \ldots & \\ \phi_{N+1}^*(t - t_N) & \text{if } t_N < t \leq T. \end{cases} \tag{8}$$

A. BASIC TEMPORAL POINT PROCESSES MODELS

In this section, we describe about the basic Temporal Point Processes (TPPs) models.

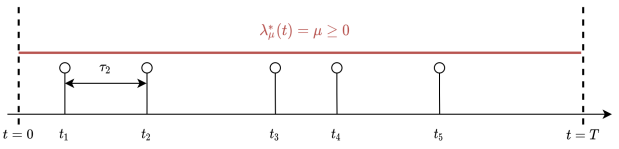


FIGURE 5. The homogenous poisson process is the simplest TPP and is defined by a constant conditional intensity function [34].

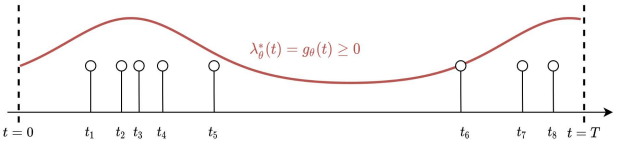


FIGURE 6. The inhomogeneous poisson process has a varying intensity capable of reflecting global patterns [34].

1) HOMOGENEOUS POISSON PROCESS

The homogeneous Poisson process depicted in Figure 5 is the simplest TPP model with a positive and constant event rate μ whose event times are independent of the history $\mathcal{H}(t)$, i.e.

$$\lambda_\mu^*(t) = \mu \geq 0.$$

The conditional probability density function $f_i^*(t)$ of the process can be derived using equation (7), so that

$$\begin{aligned} f_i^*(t) &= \lambda^*(t)\exp(-\int_{t_{i-1}}^t \lambda^*(u)\,du) \\ &= \mu\exp(-\mu(t - t_{i-1})) \\ \iff f_i^*(\tau) &= \mu\exp(-\mu\tau). \end{aligned} \tag{9}$$

From equation (9), it follows that the inter-event times τ follow an exponential distribution with parameter μ. Therefore, we can alternatively define a homogeneous Poisson process as a sequence of N i.i.d. exponentially distributed random variables, i.e. $(\tau_i)_{i\in\{1,\ldots,N\}}$ [32], [35].

2) INHOMOGENEOUS POISSON PROCESS

Another basic TPP is an inhomogeneous Poisson process depicted in Figure 6, where the event rate varies as a function of time. The conditional intensity function $\lambda^*(t)$ is specified by any function parameterized by θ, resulting in

$$\lambda_\theta^*(t) = g_\theta(t) \geq 0.$$

As with the homogeneous Poisson process, the event times are independent of the history $\mathcal{H}(t)$ [32]. Therefore, these TPP models (homogeneous and Inhomogeneous Poisson Process) are suitable for modeling global trends such as food orders in a restaurant where the event rate is spiked around lunch and dinner time. We can model this pattern by choosing an appropriate function g_θ whose shape reflects these trends [33].

3) HAWKES PROCESS

For both the homogeneous and inhomogeneous Poisson processes, the event rate is independent of past events.

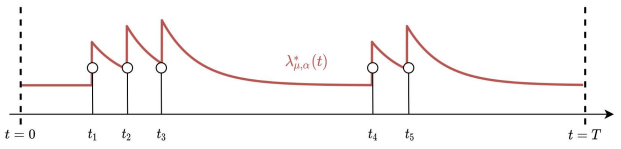


FIGURE 7. The conditional intensity function of the Hawkes process increases with each event that occurs and then slowly decays, resulting in a temporally clustered distribution of events [34].

However, for many applications, the occurrence of an event increases the probability of other events occurring immediately afterward. So-called *self-exciting processes* can simulate this behavior by increasing the conditional intensity function $\lambda^*(t)$ for each event that occurs by a certain amount. The most famous of these processes is the *Hawkes* process, whose intensity is defined by

$$\lambda^*(t) = \lambda_0(t) + \sum_{t_i \in \mathcal{H}(t)} \kappa(t - t_i),$$

where $\lambda_0(t) > 0$ is the base intensity and $\kappa(t) > 0$ is the *kernel* function. The base intensity captures events triggered by external sources and is thus independent of previous events. The kernel function models the dependence on previous events and gives the Hawkes process its characteristic self-excitation. The kernel computes for each past event $t_i \in \mathcal{H}(t)$ the quantitative influence on the intensity at time t. To ensure that events further back in time have less influence than recent events, a monotonically decreasing function is usually chosen, such that the influence of a past event on intensity decays with increasing temporal distance. The most popular function for this is the exponential function, such that the kernel function is defined by

$$\kappa(t) := \alpha \exp(-\omega t),$$

where $\alpha \geq 0$, $\omega > 0$ and $\alpha < \omega$ applies [32], [35], [36]. The parameter α scales the self-exciting behavior of the Hawkes process, eliminating it with $\alpha = 0$. The parameter ω affects how fast the influence of a past event on the intensity at time t decays with growing temporal distance, with a high value for ω leading to faster decay.

Figure 7 shows a realization and the corresponding intensity function of a Hawkes process with constant base intensity, i.e. $\lambda_0^*(t) = \mu$, and exponential kernel function. We can see that the intensity jumps with each event by the amount α and decays exponentially with increasing time until it returns to the value of the base intensity.

B. NEURAL TEMPORAL POINT PROCESSES MODELS

Neural TPP models autoregressively predict the time t_i and mark m_i of the next event by conditioning the prediction on the history of past events $\mathcal{H}(t_i)$. In [24], the authors partition the prediction process of neural TPP models (shown in Figure 8) into the following three steps:

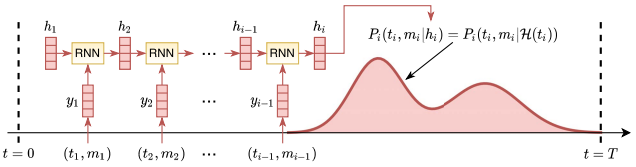


FIGURE 8. In a neural TPP, the distribution over the next event $P_i(t_i, m_i|\mathcal{H}(t_i))$ is parameterized with the RNN's hidden state vector h_i, which encodes the event history $\mathcal{H}(t_i)$ (therefore also called history embedding) [24].

1) Each event (t_i, m_i) is mapped to a feature vector $\boldsymbol{y}_i$.
2) The history $\mathcal{H}(t_i)$ is encoded by the history embedding vector $\boldsymbol{h}_i$, which is computed by sequentially feeding $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{i-1}$ into an RNN.
3) The conditional distribution over the next event $P_i(t_i, m_i|\mathcal{H}(t_i)) = P_i^*(t_i, m_i)$ is parameterized by $\boldsymbol{h}_i$, so $P_i(t_i, m_i|\mathcal{H}(t_i)) = P_i(t_i, m_i|\boldsymbol{h}_i)$. P_i^* can be defined by f_i^*, F_i^*, S_i^* or ϕ_i^* (see Table 1) [24].

While the first and second steps are similar for prominent neural TPP implementations such as *RMTPP* [37], *FullyNN* [38], and *LogNormMix* [31], they differ significantly in the third step. Therefore, in the following subsections, we present the neural TPP models *RMTPP* [37] and *LogNormMix* [31] in more detail.

1) RECURRENT MARKED TEMPORAL POINT PROCESSES (RMTPP)

The RMTPP model was the first TPP to encode event history by the hidden state $\boldsymbol{h}_i$ of an RNN, thereby parameterizing the distribution over the next event $P_i^*(\tau_i, m_i)$, i.e., $P_i(\tau_i, m_i|\boldsymbol{h}_i)$. The model assumes conditional independence between the mark and inter-event time, such that $P_i(\tau_i, m_i|\boldsymbol{h}_i) = P_i(\tau_i|\boldsymbol{h}_i)P_i(m_i|\boldsymbol{h}_i)$. The mark distribution $P_i^*(m_i)$ is defined as a categorical distribution. The time distribution $P_i^*(\tau_i)$ is characterized by the hazard function $\phi_i(\tau_i|\boldsymbol{h}_i) = \exp(w\tau_i + \boldsymbol{v}^T\boldsymbol{h}_i + b)$, where the vector $\boldsymbol{v}$ and the scalars b and w are learnable parameters and the exp transformation guarantees the positivity constraint of the hazard function [37]. By applying equation (7), we can express $\phi_i^*(\tau_i)$ as a conditional probability density function $f_i^*(\tau_i)$, which in this case is a Gompertz distribution [31]. Because of the simplicity of the hazard function, the integral $\int_0^{\tau_i} \phi_i^*(u)\,du$ of the likelihood can be computed analytically. Unfortunately, no closed-form formula exists for computing the mean of the distribution, i.e., $\mathbb{E}\left[f_i^*(\tau_i)\right]$. Instead, an integral must be solved numerically for its computation. However, the model allows to draw samples analytically from the distribution [37].

2) LogNormMix

As with RMTPP, the TPP LogNormMix [31] assumes conditional independence between the mark and time such that $P_i(\tau_i, m_i|\boldsymbol{h}_i) = P_i(\tau_i|\boldsymbol{h}_i)P_i(m_i|\boldsymbol{h}_i)$. Similarly, the mark distribution $P_i^*(m_i)$ is defined as a categorical distribution. The unique feature of LogNormMix is that it characterizes the distribution over τ_i with the conditional probability density

function $f_i^*(\tau_i)$, whereas most other TPP models use the intensity for this purpose. This offers the advantage that we can specify f_i^* with any positive PDF, thereby automatically satisfying the condition of a valid distribution. LogNormMix uses a mixture model to specify f_i^* as they are well suited for low-dimensional density estimations [39] and therefore in particular for modeling the one-dimensional inter-event time τ_i. As a mixture distribution defined in $(0, \infty)$, LogNormMix uses a mixture of K log-normal distributions defined by

$$f_i(\tau_i|\boldsymbol{w}_i, \boldsymbol{\mu}_i, \boldsymbol{s}_i) = \sum_{k=1}^{K} \frac{w_{ik}}{\tau_i s_{ik}\sqrt{2\pi}} \exp\left(-\frac{(\log \tau_i - \mu_{ik})^2}{2s_{ik}^2}\right) \tag{10}$$

The parameters of the mixture distribution are computed using the hidden state $\boldsymbol{h}_i$ of the RNN, i.e.

$$\begin{aligned} \boldsymbol{w}_i &= \text{Softmax}(\boldsymbol{V}_w \boldsymbol{h}_i + \boldsymbol{b}_w) \\ \boldsymbol{s}_i &= \exp(\boldsymbol{V}_s \boldsymbol{h}_i + \boldsymbol{b}_s) \\ \boldsymbol{\mu}_i &= \boldsymbol{V}_\mu \boldsymbol{h}_i + \boldsymbol{b}_\mu \end{aligned} \tag{11}$$

where $\boldsymbol{V}_w$, $\boldsymbol{b}_w$, $\boldsymbol{V}_s$, $\boldsymbol{b}_s$, $\boldsymbol{V}_\mu$, and $\boldsymbol{b}_\mu$ are learnable parameters and the softmax and exp transformations enforce the parameter constraints of the distribution. The model allows the computation of the survival function $S_i^*(T)$ of the likelihood with a closed-form formula. The mean of the distribution, i.e., $\mathbb{E}\left[f_i^*(\tau_i)\right]$, can also be computed analytically by taking the weighted mean of the component means. In addition, we can also analytically draw samples from the distribution [31].

We can efficiently train both models due to their likelihood in closed-form. However, the multimodal log-normal mixture distribution of LogNormMix provides much higher flexibility in modeling $f_i^*(\tau_i)$ than the unimodal Gompertz distribution of RMTPP. Using a log-normal mixture distribution allows us the approximation of any distribution [31].

IV. TppFaaS–DEVELOPED SYSTEM

In this section, we introduce our developed system called *TppFaaS* (see Figure 9), for modeling function invocations in FaaS applications using temporal point processes (TPPs). It is designed for applications running on OpenWhisk [25] FaaS platform, which underneath uses Kubernetes cluster. For modeling, trace data of application functions is collected and is used for training TPP models. Based on these models, predictions are carried out.

Additionally, for creating the dataset for training TPP models, we created a component within *TppFaaS* called *Sampler*. The Sampler is an automated pipeline for creating simulated FaaS applications by specifying configuration parameters. Here, an application is a function composition in which sleep commands simulate the execution times of the functions following a distribution. This simulated application is deployed on the OpenWhisk FaaS platform. For generating the traces, *Sampler* send user requests to the deployed application (Step 1 in Figure 9). The OpenWhisk executes the application. The *OpenTelemetry* library [40] instruments the application's functions, and exports a span for each function invocation to the post-processing service *Trace Collector* (Step 2). The *Trace Collector* enriches the span with meta-information retrieved from the OpenWhisk API (Step 3) and subsequently exports it to *Zipkin* [41] (Step 4). Here, the spans are aggregated into traces and then fetched by the *Sampler*, which transforms the traces into a data format suitable for the TPP models (Step 5). Once a trace dataset is generated, we split it into a training and test dataset. We use the training data to optimize the parameters of the TPP model (Steps 6-7), which we then evaluate using the test data (Step 8).

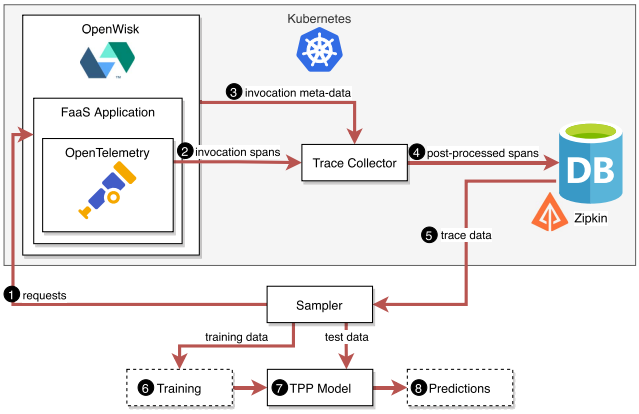


FIGURE 9. TppFaaS is a system for modeling FaaS applications using temporal point processes. For this purpose, trace data is collected from an artificial FaaS application that the user can easily create via configuration. The trace data is then used to train a TPP that models the function invocations of the FaaS application.

In the following subsections, we first describe how the simulation of FaaS applications is carried out (§IV-A), and then describe each component of *TppFaaS* in more detail.

A. SIMULATION OF FaaS APPLICATIONS

1) FaaS APPLICATIONS FUNCTION's DURATION SIMULATION

A FaaS application is a composition of multiple functions. To simplify the construction of different compositions, each function is simulated by a sleep command whose duration is drawn from a probability distribution for each function invocation. This approach was also followed in [11], in which a composition was built that consisted of a sequence of 16 functions. The execution duration of each function was drawn from an exponential distribution, where each function had an individual parameterization of the distribution. The parameters were chosen so that the distribution of the first function had an expected value of 500ms and was increased by 50ms for each subsequent even position and decreased by 50ms for each subsequent odd position. The expected values of the sequence were thus 500, 550, 450, 600, 400, ..., 150, 900ms. The function duration in this work is simulated using a gamma distribution, since this distribution is the

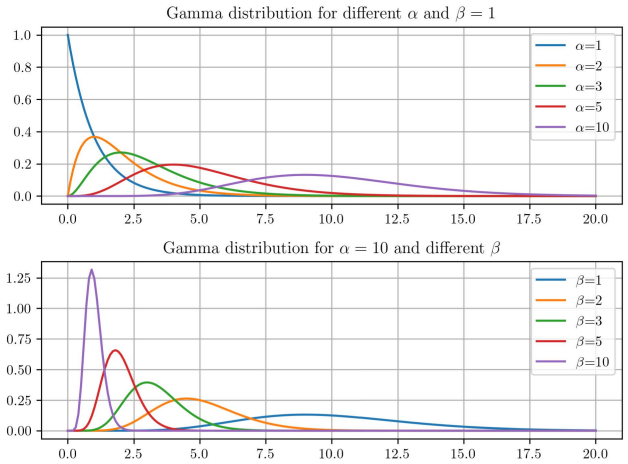


FIGURE 10. Effects of α and β on the gamma distribution.

TABLE 2. Effects of α and β on the entropy of the gamma distribution.

α	1	2	5	10	10	10	10
β	1	1	1	1	2	5	10
Entropy	1.00	1.58	2.15	2.54	1.84	0.93	0.23

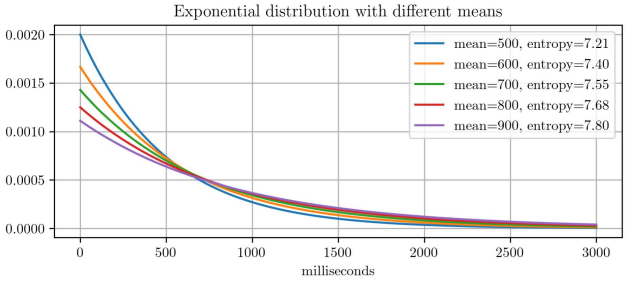


FIGURE 11. Exponential distributions based on [11] for the simulation of the function duration having mean between 500ms and 900ms.

generalization of the exponential distribution and thus offers us higher flexibility with its two parameters α and β. For $\alpha > 0$, $\beta > 0$ and $x \in (0, \infty)$ the probability density function of the gamma distribution is defined as

$$f(x) = \frac{\beta^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

The exponential distribution is a special case of the gamma distribution at $\alpha = 1$ and $\Gamma(1) = 1$. To better understand the effect of α and β, the gamma distribution is visualized in Figure 10 for different parameterization. In the top figure, α varies while β remains fixed, and we note that, the lower the α, the more right-skewed the distribution becomes. That is, the density mass shifts to the left and gets a long flat sloping tail on the right side. Thus, if the function duration is simulated with a strong right-skewed gamma distribution, this leads increasingly to high outlier values, which negatively affect the later evaluation of the TPP model with respect to the mean absolute error (MAE). In general, the more concentrated the distribution of function durations is, the more accurately the duration until the next function invocation τ_i (see Table 1) can be predicted. This is because τ_i depends on the execution time of the previous function. That is, if the distribution of the function duration of the previous function is very flat, this will lead to an equally flat distribution for $f_{i,\text{true}}(\tau_i)$, the true distribution for the duration until the next function invocation. If the TPP model is evaluated using the absolute mean error between the expected value of the modeled $f_i^*(\tau_i)$ (see Table 1) and the true duration until the next function invocation τ_i, the more spread out realizations of $f_{i,\text{true}}(\tau_i)$ result in a larger error. This highlights the dilemma between parameterization of the functions and the success of the findings. However, if the model is evaluated using the negative log-likelihood, i.e., how similar the modeled distribution $f_i^*(\tau_i)$ is to the true distribution $f_{i,\text{true}}(\tau_i)$, this problem does not apply because the TPP can also model a flat distribution due to its flexibility.

In the lower figure of Figure 10, we can observe the effect of the parameter β on the distribution. We can see that a lower β causes a shift of the density mass to the right, but the skewness remains constant. Moreover, the distribution becomes flatter and spreads out more. For the predictive ability of the TPP model, this leads to the same issue as described previously.

We can measure the concentration of the distribution's mass using the distribution's entropy [42]. The higher the entropy, the more uncertain is the value of a possible realization of the distribution. That is, flat and widespread distributions have higher entropy than spiky distributions. We see the influence of the parameters α and β on the entropy of the gamma distribution in Table2. If α increases and β remains fixed, the entropy increases. This observation also aligns with the graphs in Figure 10, in which the distributions flatten for a higher α. In the opposite case, α remains fixed and β increases, the entropy decreases and the distributions become more peaked. In conclusion, we can assume that using low entropy distributions for simulating the function duration will reduce the MAE.

Figure 11 shows the exponential distributions used in [11] for the simulation of the function duration, as well as their entropies. Among all continuous distributions defined in $[0, \infty)$, the exponential distribution for a given expected value has the highest entropy [43]. For this reason, we will not use this distribution for modeling the function duration. Instead, we will rely on gamma distributions as in Figure 12, whose parameterization result in lower entropies than in the case of the exponential distribution. In [11], high entropies are not an issue because their mechanism for reducing cold starts is not based on predicting function invocations using probabilistic models. Moreover, in our work, high entropies are only problematic when the TPP is evaluated using the MAE. In contrast, it should not be a problem for the TPP to model the distribution of the duration until the next function invocation $f_i^*(\tau_i)$ such that it resembles the true distribution $f_{i,\text{true}}(\tau_i)$.

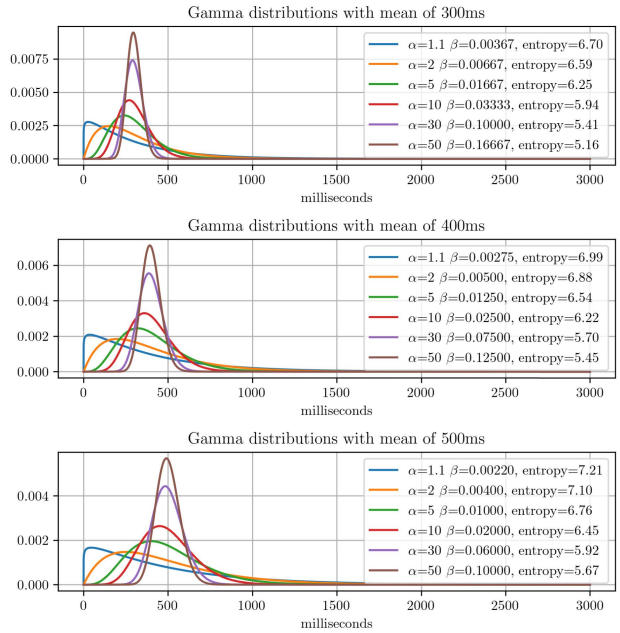


FIGURE 12. Gamma distributions with mean fixed at 300, 400 or 500ms. It can be seen that the distribution deforms to a normal distribution and the entropy decreases.

In the topmost figure in Figure 12, all distributions have an expected value of 300ms. For an increasing α and β, the distribution becomes more symmetric and peaked, while at the same time, the entropy decreases. For a high α and β, the distribution resembles a normal distribution. Analogously, we can see this in the below figures, where the distributions have an expected value of 400ms and 500ms, respectively. We will simulate the functions of the FaaS applications used in this work with gamma distributions with expected values of 300ms, 400ms, and 500ms. Higher expected values again lead to higher entropies and are therefore not used in this work.

Besides the gamma distribution, the log-normal distribution, also defined in $(0, \infty)$, would be another candidate for modeling the function duration. As mentioned above, the duration until the next function invocation τ_i depends on the function duration of the previous function. For example, if we consider a FaaS application in which the functions are invoked sequentially one after another, and if we also neglect the overhead of OpenWhisk, then the end of a function execution also signifies the start of the following function. This would result in $f_{i,\text{true}}(\tau_i)$ being equal to the distribution of the function duration of the previous function. Thus, the TPP that attempts to model $f_{i,\text{true}}(\tau_i)$ indirectly models the distribution of the function duration of the previous function. If the function duration is simulated by a log-normal distribution, then it would be too easy for our TPP to model $f_{i,\text{true}}(\tau_i)$, since it also uses a log-normal mixture distribution for modeling (see §III-B2). However, is this work, we also showcase the flexibility of the log-normal mixture distribution. With this, it should be possible to model any distribution, such as a gamma distribution.

2) FUNCTION COMPOSITION

We implement the FaaS applications to generate trace data using JavaScript functions that we instrument with the OpenTelemetry library [40]. This library provides tools for generating and modifying spans by code and the ability to post-process the generated spans and export them to multiple destinations. An application is constructed as a composition of $n + 1$ functions, where the function *main* is the entry point of the composition and invokes one or more subsequent functions. We denote the remaining functions of the composition with $f_1, \ldots, f_i, \ldots, f_n$. A function can invoke one or more successor functions, allowing the construction of simple or complex compositions such as sequences or trees. A span represents the duration of a function execution, making a trace a representation of all the function executions in the composition. A composition is finished after all branches of the composition have been executed. All function invocations are made asynchronously, meaning that a function does not wait for the completed execution of the invoked following functions. Instead, it is terminated upon invocation of the following functions. As discussed in §IV-A, the logic of the functions $f_1, \ldots, f_i, \ldots, f_n$ consists only of a sleep command whose duration is drawn from a gamma distribution. This simple logic enables the functions to share the same function code.

It is to be noted that, we can find conditional function invocations in real-life applications expressed with an if-else syntax. For example, a condition evaluated in function f_1 can decide whether we invoke function f_2 or function f_3 next. For the sake of simplicity, we do not include such conditional function invocations in this work. To cover conditional function invocations, the input parameters to the function would be required as another feature for modeling.

We create compositions using the generic functions and are individualized by different parameterizations. Using the *serverless framework* [44] we can simplify the parameterization as well as the lifecycle management of the composition. We configure the functions and their parameters in a YAML file named *serverless.yaml*. We show an example of such a configuration in Listing 1, with a composition consisting of five functions: *main*, f_1, f_2, f_3, and f_4. The *handler* attribute (in line 8 or 15) specifies the function code. Here, we reference the generic and configurable JavaScript implementations, *main* and *f*. The two implementations receive different parameters specified under the *parameters* attribute. These parameters are passed to the function as default parameters via the *params* parameter object upon invocation. For the *main* function, the specification of the address and authentication of OpenWhisk is required (in line 10 and 11). These parameters are needed to initialize the OpenWhisk client, which invokes the successor function. The *f* functions also use this client, but get the required parameters propagated from the *main* function via *params*. Analogously, the *collectorHost* parameter (in line 12) is propagated to the *f* functions. This parameter specifies the

```
1  service: tree
2  provider:
3    name: openwhisk
4  functions:
5    main:
6      handler: funcs.main
7      parameters:
8        owHost: ...
9        owAuth: ...
10       collectorHost: ...
11       nextFn: f1
12   f1:
13     handler: funcs.f
14     parameters:
15       alpha: 30
16       beta: 0.075
17       nextFn: [f2, f3]
18   f2:
19     handler: funcs.f
20     parameters:
21       alpha: 30
22       beta: 0.06
23   f3:
24     handler: funcs.f
25     parameters:
26       alpha: 30
27       beta: 0.075
28       nextFn: f4
29   f4:
30     handler: funcs.f
31     parameters:
32       alpha: 30
33       beta: 0.06
```

LISTING 1. Example configuration file for creating a variety of compositions.

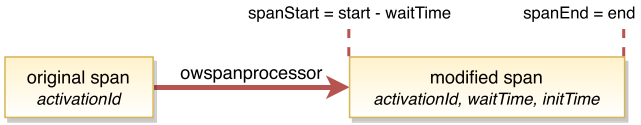


FIGURE 13. The owspanprocessor adapts start and endpoint of the original span and adds further attributes.

address of the Trace Collector to which we export the spans for post-processing. The *main* and *f* functions share the *nextFn* parameter. This parameter specifies the successor functions that the OpenWhisk client invokes after the function execution finishes. If we specify multiple functions here as in line 19, the client invokes these functions in parallel. The specification of multiple functions enables the construction of compositions with a tree-like function hierarchy, as is the case in Listing 1. The function *f*, whose function duration is simulated with a gamma distribution, is additionally configured with the parameters of the distribution, *alpha* and *beta*. In summary, the approach shown provides an easy way to create diverse compositions.

B. TRACE COLLECTOR

We used the OpenTelemetry Collector Library [45] to implement a custom collector[2] that post-processes the spans produced by the instrumented FaaS application. We configure the components of the collector within the YAML file. The collector consists of one receiver, three processors, and one exporter.

The spans produced by the instrumented FaaS application are received over HTTP by the pre-implemented **OLTP Receiver** [46], and forwards them to the first processor in the pipeline, the batch processor. The **Batch Processor** [47] aggregates the data to minimize later outgoing connections from the exporter. It is configured with the parameters *send_batch_size* and *timeout*. The former specifies the maximum batch size. The parameter *timeout* specifies the time after the batch is forwarded to the pipeline's next step, regardless of its size. The next processor in the pipeline, the **owspanprocessor**, receives the aggregated spans. This processor is developed by us using the OpenTelemetry collector library and is configured with the host address of OpenWhisk. The processor extracts the span's *activationId* attribute to retrieve meta-information about the span's associated function invocation from the OpenWhisk API. The attributes extracted by the owspanprocessor measured in milliseconds are: *start*, *end*, *waitTime*, and *initTime*.

- The *start* attribute is a Unix timestamp and is computed by $start := executionStart - initTime$, where *executionStart* is a Unix timestamp specifying the start time of the function code execution. That is, *start* already specifies the start of function initialization for a cold function invocation. This is unfortunately not evident from the OpenWhisk documentation, but can be derived from the source code [48] of OpenWhisk.
- The *end* attribute is a Unix timestamp and specifies the end of function execution.
- The *initTime* attribute specifies the duration of function initialization which applies only to cold function invocations, making the attribute optional.
- The *waitTime* attribute specifies the OpenWhisk caused delay occurring before the function initialization/execution [27].

As illustrated in Figure 13, the processor uses the extracted attributes to adjust the start and end time of the span. On the one hand, *waitTime* and *initTime* should be included in the duration of the span. On the other hand, the start and end time of the received span does not match the true start and end time of the function invocation and should be adjusted with *start* and *end*. Thus, the processor modifies the start and end time of the span as follows:

$$\begin{aligned} spanStart &:= start - waitTime \\ spanEnd &:= end \end{aligned} \tag{12}$$

Additionally, the *waitTime* and *initTime* are added to the modified span as attributes. In the pipeline's next step, the **owspanattacher** processor receives the spans. As shown in Figure 14, the processor creates a child span for each of the *waitTime* and *initTime* attributes, as well as a child span *executionTime* that represents the function code execution. The start time of the child span *executionTime* is computed

[2]https://github.com/maSteinbach/owtracecollector.git

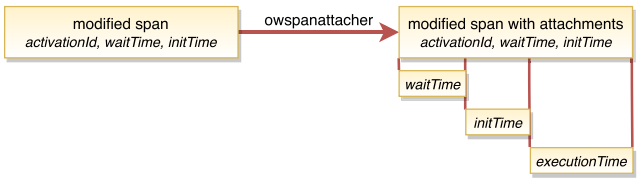


FIGURE 14. The owspanattacher adds child spans for waitTime, initTime, and executionTime.

with *executionStart* = *start* + *initTime*. The child spans are not required for the later modeling and serve an exclusively visual purpose. Therefore, this processor is optional and can be removed from the pipeline if desired.

In the pipeline's last step, the spans are exported to Zipkin, a backend service that receives, validates, indexes, and stores them aggregated into traces [49]. For exporting, we specify a **Zipkin Exporter** with the address of Zipkin in the collector configuration. The exporter transforms the spans into the Zipkin data format and sends them to the given address. We can use Cassandra as a backend database for Zipkin.

C. SAMPLER

The Sampler is an automated end-to-end pipeline that contains all the necessary steps for trace datasets generation used to train and evaluate TPP models, such as deploying the application, sending requests and collection of data. The Sampler creates the datasets by sending n requests to the FaaS application's *main* function at irregular time intervals. The *main* function represents the entry point of the application. The time intervals between requests are drawn from a continuous uniform distribution with an interval specified by the user, who thus determines the load on OpenWhisk and, indirectly, the number of cold starts. Another feature of the Sampler is performing requests in batches, pausing requesting after each batch for a user-specified duration. The result is a dataset consisting of n traces whose format is compatible with training a TPP model.

The first step of the pipeline validates the user-input arguments, such as that the interval of the uniform distribution is in the positive range. Next, it verifies that Node.js and the Serverless Framework CLI are available. Using Node.js, the pipeline installs the FaaS application's dependencies, such as the OpenTelemetry library. In the next step, the application is deployed using the Serverless CLI, where the OpenWhisk credentials are read from a configuration file and provided to the CLI as environment variables. After deploying the application, the sampler sends n requests to the FaaS application's *main* function at irregular time intervals, whose durations are drawn from a uniform distribution each time. For each request, OpenWhisk returns the unique *activationId* of the *main* function invocation, which is collected in the *unfetched_ids* array. Once the sampler has sent all n requests, it may take some time to execute all function invocations, depending on OpenWhisk's load. With

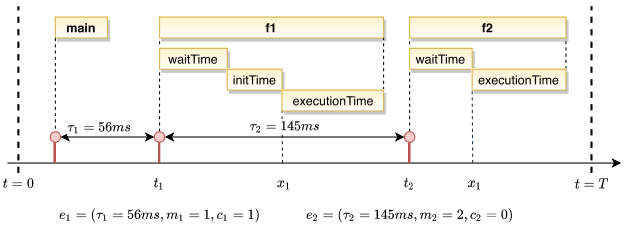


FIGURE 15. The spans of the invoked functions *f1* and *f2* are mapped to the 3-tuple events (e_1) and (e_2), which carry the inter-event time (τ_i), the function class (m_i), and the cold start feature (c_i). Given the cold invocation of *f1*, we have ($c_1 = 1$).

the *activationIds* returned by OpenWhisk, we can reference any span associated with a *main* function invocation of the generated n traces. The Zipkin API provide the ability to filter traces by a single span attribute. Thus, iterating over the *activationIds* of the *unfetched_ids* array and setting the ID as a filter criterion, we fetch each trace of the n requests from the Zipkin API. For each fetched trace that is complete, the respective *activationId* is removed from the *unfetched_ids* array. If the trace is incomplete, we keep the ID in the array so that the trace can be retrieved again in the next loop. The iteration stops if either the array *unfetched_ids* is empty or the number of IDs in the array stagnates after several iterations. The latter happens upon runtime errors of OpenWhisk so that some traces are never completed. Zipkin returns the traces as JSON, from which the sampler extracts the necessary information and converts it to a format compatible with the TPP model.

In order to convert the extracted spans into the TPP model format compatible, we first decompose the span of a function invocation into the three time ranges *waitTime*, *initTime* (for a cold start), and *executionTime* (see §IV-B). We map the span to an instantaneous point in time, denoted as an event in the context of TPPs. For the next invocation, we want to predict the time at which its request arrived at the FaaS platform. The FaaS platform could use the predicted time to upscale the function upfront, allowing it to begin its execution without delay. In reality, however, a cold start or platform-specific issues, such as the creation of a Docker container, might delay the function execution, which OpenWhisk captures through the *waitTime* and *initTime*. So, to predict the time when the request for the next function invocation arrives at the FaaS platform, we need to subtract these delays from the actual start of the function. Let w_i be the *waitTime*, i_i be the *initTime*, and x_i be the start time of the function execution of the i^{th} invocation, then we define $t_i = x_i - w_i - i_i$ as the mapping of the span to an instantaneous point in time. The mapping is visualized in the example in Figure 15, where the functions *main*, *f1*, and *f2* are invoked sequentially, with a cold start occurring on *f1*. We represent the event of the i^{th} invocation, which we denote by e_i, as either the 2-tuple (τ_i, m_i) or 3-tuple (τ_i, m_i, c_i). The attribute $\tau_i = t_i - t_{i-1} \in \mathbb{R}_+$ describes the inter-event time from §III. We use $m_i \in \mathbb{N}_0$, denoted as a mark in §III, to specify the class of the invoked function. The binary attribute $c_i \in \{0, 1\}$ is an optional feature intended to enhance

the predictive ability of the TPP model, indicating whether the i^{th} function invocation was a cold start. We compute the feature with $c_i = w_i > 0$.

In the final steps of the pipeline, the sampler saves the formatted trace dataset as a pickle and shuts down the application.

D. TPP MODELS

In this section, we briefly describe the TPP models and their purposes within *TppFaaS* for modeling functions invocations.

1) LogNormMix: τ_i AS A LOG-NORMAL MIXTURE DISTRIBUTION

We use the TPP model LogNormMix (§III-B2) to model the duration until the next function invocation with the conditional probability distribution $f_i^*(\tau_i)$, where f_i^* is defined as a log-normal mixture distribution. For this, we compute the duration until the next function invocation, i.e., the inter-event time τ_i, from the time points of the function invocations t_i. Since the inter-event times may take high values, they are logarithmized and centered. The inter-event time is combined with the function class attribute m_i and the optional cold start feature c_i to yield the 3-tuple event $e_i = (\tau_i, m_i, c_i)$, which represents the function invocation and is input to the RNN. We represent each function class by a trainable 32-dimensional embedding vector. The vectors are concatenated into an embedding matrix indexed by m_i. Analogously, we represent the two values of the cold start feature, c_i each, by a trainable 32-dimensional embedding vector. The RNN ingests the event e_i and produces a hidden state vector $h_i \in \mathbb{R}^{64}$ that encodes the history of past invocations. An affine transformation and subsequent softmax operation maps the vector h_i to the parameters of the log-normal mixture distribution. The softmax operation forces the component weights of the mixture distribution to sum to 1.

2) TruncNorm: τ_i AS A SINGLE VALUE

Instead of an entire probability distribution $f_i^*(\tau_i)$, a single value for the inter-event time τ_i is sufficient for some applications. For example, if the FaaS platform must initialize the function in advance to avoid a cold start, only the single value τ_i is required. Thus, we need a point estimate of $f_i^*(\tau_i)$ that maps the distribution to a single value. There are two methods to obtain this point estimate. First, as in §IV-D1, we can model $f_i^*(\tau_i)$ with LogNormMix, which provides us with a log-normal mixture distribution for it. The expected value of this mixture distribution, i.e. $\mathbb{E}\left[f_i^*(\tau_i)\right]$, can be computed analytically and quickly, representing the desired point estimate of $f_i^*(\tau_i)$. In the second method, we map the hidden state vector h_i of the RNN to a positive real number representing the inter-event time τ_i using an affine transformation and subsequent softplus operation. Instead of softplus, we can use any other operation that enforces $\tau_i > 0$, such as the logarithm. We may also interpret this method as a TPP that models the conditional probability distribution $f_i^*(\tau_i)$ with a truncated normal distribution with constant variance [50]. The normal distribution is "truncated" as it is not defined in $\mathbb{R}$ as usual, but only in $\mathbb{R}_+$. The single value for τ_i, obtained by the affine transformation of h_i and the softplus operation, is the expected value of this distribution. In this work, we selected the second method (which we refer to as *TruncNorm*) since, for a simple point estimate, the high flexibility of the log-normal mixture distribution is unnecessary for modeling f_i^*. Moreover, we experienced more stable training with TruncNorm and a faster decrease of the loss function, i.e., the mean absolute error.

3) MARK MODELED WITH A CATEGORICAL DISTRIBUTION

We assume that the mark or function type m_i and the inter-event time τ_i of the i^{th} function invocation are independent. We define the distribution over m_i as the categorical distribution $f_i^*(m_i) = f_i(m_i|\mathcal{H}(t_i))$ parameterized by the vector $\boldsymbol{\pi}_i$. The value $\pi_{i,c}$ describes the probability that m_i is of class c. We obtain $f_i^*(m_i)$ by an affine transformation of the hidden state vector h_i produced by the RNN and a subsequent softmax operation.

V. EVALUATION SETTINGS

This section describes the various evaluation settings used in this work. First, we describe the benchmark applications used in §V-A. Then we present the infrastructure settings on which the evaluation is conducted in §V-B. Furthermore, we explain the various datasets generated for evaluation in §V-C, and training models hyperparameters in §V-D. Lastly, in §V-E, we define the performance quality evaluation metrics used in this work for evaluation of the results.

A. BENCHMARK APPLICATIONS

To generate trace datasets, we construct several instrumented FaaS applications using the method described in §IV-A2. That is, the applications are a composition of several artificial functions whose execution time is simulated by a sleep command. We configure the application in the YAML file of the Serverless Framework. With it, we specify the call graph, i.e., the structure of the composition that dictates in which order the functions invoke each other. In addition, we use the configuration to specify the duration of the sleep commands of the individual functions. By adjusting these two hyperparameters, the structure of the composition, and the distribution of the function duration, we build applications with unique characteristics that complicate the modeling of the function invocations for the TPP model. In particular, 1) the constructed applications exhibit different structural characteristics (sequence, parallel, tree and fanout), 2) each of the applications are scaled in two variants: small variant and large variant, and 3) for each variant of the application, we implement a randomized and a non-randomized variant. In the non-randomized variant, the duration of the sleep command for all functions is fixed with either 300, 400, or 500ms. In the randomized variant, the duration is drawn from a

TABLE 3. To simulate the functions, three gamma distributions with mean values of 300, 400 and 500ms are used for the randomized FaaS applications. The respective values of the parameters for α and β of the gamma distribution are listed in this table.

mean	300ms	400ms	500ms
α	30	30	30
β	0.1	0.075	0.06

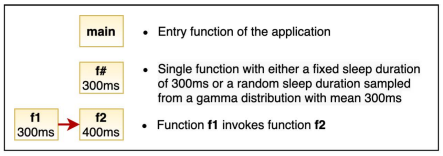


FIGURE 16. The application *sequence* is a sequential composition of 11 functions invoked one after another.

gamma distribution for each function invocation (see §IV-A). During configuration, we, therefore, assign each function one of three gamma distributions with expected values of either 300, 400, or 500ms. The associated parameters α and β of these distributions are listed in Table 3 and the associated distributions are shown in Figure 12.

In the rest of the paper, we label the randomized applications with the substring *rand* in the application name. However, we do not distinguish between randomized and non-randomized applications in the following figures of the section. Therefore, when we refer to the duration of the sleep command in the following, we also mean the expected value of the gamma distribution. In the following subsections, we present the four constructed FaaS applications: sequence, parallel, tree and fanout. For each of these applications, except for *sequence*, four variants exist. These include the two variants for scaling the structural characteristic, i.e., *small* and *large*, as well as a randomized and non-randomized variant for each. The application *sequence* exists only in the versions *sequence* and *sequence_rand*, since scaling the simple structure does not provide any added value. All applications consist of exactly one *main* function and multiple *f#*. We study the applications with respect to the following aspects: the structural characteristic, what challenges do this pose for the TPP model, how is the structural characteristic scaled, and how are the sleep commands of the functions configured.

1) SEQUENCE APPLICATION

The **sequence** application shown in Figure 16 is the simplest of the four applications. Its structure consists of 11 functions that are invoked one after another. Due to the simplicity of the structure, scaling of the sequence is not required. If the TPP can model a sequence of 10 functions, it should also be possible for 20 and more. Furthermore, there are no parallel function executions in the structure. Therefore, the function invocations mapped to the time axis always have the same order. Thus, it should be straightforward for the model to predict the class of the next invocation. The sleep commands are configured with the cyclically increasing sequence of 300, 400, and 500ms. However, except for the sleep command of the last function f10, which has a duration of 0ms. Since no functions follow here, a simulated function duration is unnecessary, and we omit it in favor of resource optimization.

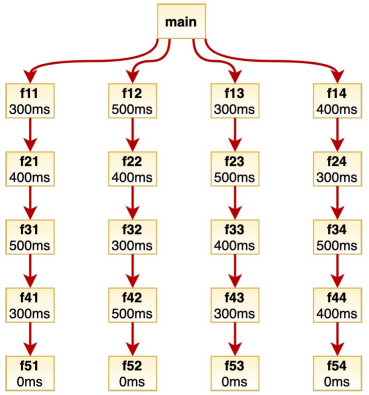


FIGURE 17. The application *parallel_large* consists of four function branches that are executed in parallel.

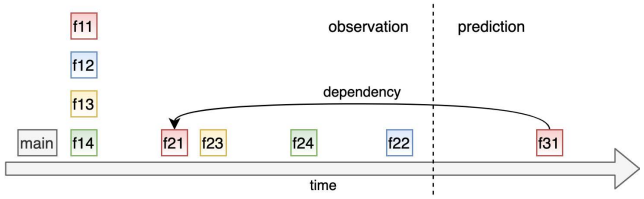


FIGURE 18. A potential time sequence of the function calls of the application *parallel_large*. The next function call f31 depends on the call of f21, but between f21 and f31 the other function calls of f22, f23 and f24 occur. This means, in order to model the call of f31, the model must be able to store the information of f21's call across multiple function invocations. This is challenging for RNN-based models.

2) PARALLEL APPLICATION

Figure 17 shows the large variant of the **parallel** application, i.e. *parallel_large*. The structural characteristic here is the parallel execution of four function branches. If we map the function invocations to the time axis, their order may differ for different traces. For example, a cool start for one function may delay the executions of the successor functions of the affected function branch, which would eventually affect the order of the invocations of the entire composition. Smaller deviations in the function duration, which we simulate with the randomization of the application, might also cause this effect. Uncertainty in the invocation order is a challenge for the TPP model. We illustrate another challenge caused by parallel function execution using Figure 18. This figure shows a possible sequence of invocations of *parallel_large* mapped to the time axis. Here, invocations from different function branches are colored differently. In the scenario shown, the functions from *main* to f22 have already been invoked, and the next function invocation, here f31, should be modeled. As illustrated in Figure 17, f31 is a successor function to f21,

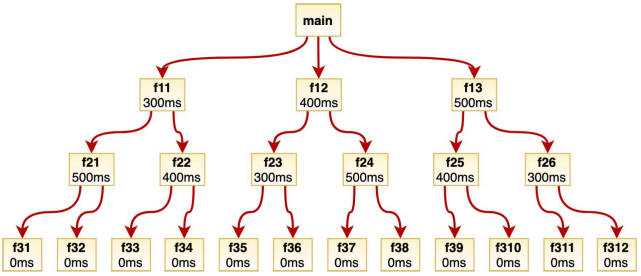


FIGURE 19. The application *tree_large* is a tree-shaped function composition with a height of three.

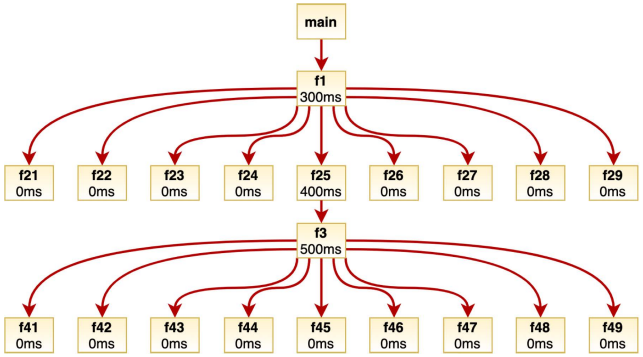


FIGURE 20. The distinctive feature of the application *fanout_large* are the functions f1 and f3 with nine subsequent functions each.

so f31 is invoked as soon as f21 finishes execution. However, we see in the figure that the invocation of f21 is relatively far in the past, and in the meantime, the functions f23, f24, and f22 have been invoked. Thus, the TPP model must memorize the invocation of f21 over multiple invocations to model the long-time dependency. This "looking far into the past" challenges TPP models based on an RNN architecture [51].

We scale the structure of the *parallel* application by the number of parallel function branches. While *parallel_large* has four of them with a total of 21 functions, *parallel_small* has two with a total of 11 functions. The sleep commands are configured to distribute the invocations of the different branches evenly over time. As with the *sequence* application, the last functions of the composition have a simulated function duration of 0ms for resource optimization reasons.

3) TREE APPLICATION

Figure 19 shows the large variant of the **tree** application, i.e. *tree_large*, whose structure is a tree of height three. Except for the *main* function, each function invokes two successor functions in parallel. That is, the number of functions executed in parallel doubles with each lower level of the tree, giving us 12 parallel function executions at the deepest level. With this high number of parallel executions, the temporal order of the function invocations is particularly uncertain. Moreover, the invocations for deeper levels occur at increasingly shorter time intervals. We scale the structure by the height of the tree. While *tree_large* has a tree height of three with a total of 22 functions, *tree_small* has a tree height of two with a total of 10 functions. An advantage is that relatively many functions of the application do not have a successor function; thus, we can specify their sleep commands with 0ms.

4) FANOUT APPLICATION

Figure 20 shows the large variant of the application **fanout**, i.e. *fanout_large*. Characteristic for the structure are the two highly parallel function executions at functions f1 and f3. However, the functions are not invoked exactly simultaneously. Instead, f1 invokes the functions from f21 to f29 sequentially, with a time gap of about 20ms. Nevertheless, many invocations occur with short time intervals, which might be challenging for the TPP. Moreover, the FaaS platform may not follow the invocation order of functions f21 to f29 defined by us. We scale the structure by changing the degree of parallelism of functions f1 and f3. While in *fanout_large*, functions f1 and f3 each have nine subsequent functions, in *fanout_small* they each have five. Thus, the total number of functions is 21 for *fanout_large* and 13 for *fanout_small*. As with *tree* application, a relatively large number of functions have no successor and, therefore, a simulated execution time of 0ms.

B. INFRASTRUCTURE SETTINGS

Generating trace data with cold starts imposes high demands on the infrastructure. To meet these, we host the performance-critical components of the system architecture, i.e., OpenWhisk, the Trace Collector, and Zipkin, on Google's Kubernetes Engine.[3] Our Kubernetes cluster consists of nine nodes, each with 32 GiB of memory and a CPU (Intel Skylake architecture) with nine virtual cores. So, in total, we have 72 CPU virtual cores and 288 GiB of memory at our disposal. The sampler service requires only a few resources and runs on a separate VM with two virtual cores.

We train our TPP models on a single-node cluster with 754 GiB of memory and two Intel Cascade Lake processors (Intel(R) Xeon(R) Gold 6238 CPU @ 2.10GHz) with 22 cores each.

C. DATASET GENERATION

We generate datasets with 1000 traces each for all variants of the four applications sequence, parallel, tree, and fanout described in §V-A. The parallel, tree, and fanout applications each exist in a small and large variant, identified by the substring *small* and *large*, respectively, in the application name. In addition, each small and large variant and sequence exist in a randomized and non-randomized variant. For each of these variants, we generate a dataset with and without cold starts. In the former, the cold invocations account for exactly 30% of the total invocations. To generate such a dataset, we create 400 traces with almost exclusively cold invocations and 1000 traces with almost exclusively warm

[3]https://cloud.google.com/kubernetes-engine

invocations. We then incrementally substitute the warm traces with cold traces until the 30% of cold invocations is reached. We generate the datasets using the Sampler from §IV-C, which sends requests to a given application. The duration between requests is drawn each time from a continuous uniform distribution whose interval bounds are specified by the parameters l (lower bound) and u (upper bound). Thus, the specification of the interval influences the request rate and thus the load on OpenWhisk. A higher request rate increases the load on OpenWhisk, which responds by scaling up the functions, causing cold starts. These interval limits are accordingly to generate datasets with or without cold starts.

For the datasets without cold starts, the choice of l and u is simple. Here, we just need to ensure that the request rate specified by l and u does not exceed the capacity of OpenWhisk. The maximum request rate depends on the nature of the FaaS application. Therefore, we choose a higher rate for the small application variants, consisting of fewer functions, than for the large ones with more functions. If the request rate is below capacity, OpenWhisk can prevent requests from queuing by scaling the functions to adapt to the load. OpenWhisk performs the scaling relatively fast, so that the proportion of cold starts of the total invocations is negligible, with less than one percent.

Generating datasets with almost exclusively cold starts is more challenging because we have to choose the request rate with l and u so that OpenWhisk scales the functions. If the sampler sends requests at a constant rate to a FaaS application, OpenWhisk starts scaling up the functions until it can serve each future request with a warm function instance without delay. As a result, even with a high request rate, cold starts only occur at the beginning, until OpenWhisk has adapted to the load. To avoid it, we set the request rate high enough that OpenWhisk cannot adapt to the load over the entire request period, and therefore also scale the functions over this period. However, we reach the limit of our hardware resources after a certain time, which prevents further scaling. From this point on, more and more requests start to queue up at OpenWhisk, which thus have to wait longer and longer for a free function instance. The *waitTime*, which captures this waiting at OpenWhisk, therefore increases continuously and reaches high values of up to several minutes. To mitigate the problem of a continuously increasing *waitTime* for a constant high request rate, we execute the Sampler's requests in batches. That is, the Sampler sends b requests to the FaaS application at a constant rate and then waits for w seconds before sending the next b requests. The size of the batch b and the duration of the pause between batches w are parameters of the Sampler. This approach prevents the accumulation of more and more requests at OpenWhisk and thus the continuous increase of the *waitTime*. If we choose b not too large and w not too small, all requests are already served by a function execution before the next batch of requests arrives at OpenWhisk. However, a too small batch size will prevent OpenWhisk from scaling, and thus no cold starts will occur.

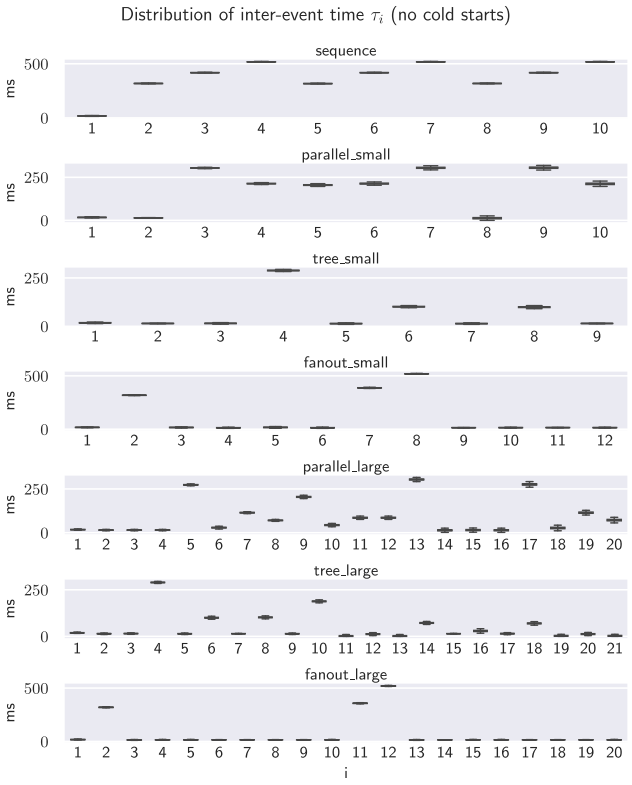


FIGURE 21. Distribution of the inter-event time τ_i for $i \in \{1, \ldots, N-1\}$ in the generated datasets without cold starts, where N is the number of functions of the FaaS composition.

Therefore, we choose the batch size b large enough to trigger scaling.

The chosen parameters of the l, u, w, and b for each dataset are shown in the Table 4. For the datasets without cold starts, no batching of the requests is required, so we do not need to specify the w and b parameters here. Based on the selected parameters, distribution of the inter-event time τ_i for $i \in \{1, \ldots, N-1\}$ in the generated datasets for the applications without cold starts is shown in Figure 21, Figure 22 for random variant and with 30% cold starts is shown in Figure 23, where N is the number of functions of the FaaS composition.

D. TRAINING DETAILS AND MODEL PARAMETERS

We partition the 1000 traces of each dataset into 600 for training and 200 each for validating and testing the TPP model. The training set is used to optimize the model parameters, the validation set is used for evaluation during training, and the test set is used for the final evaluation. To obtain averaged results, we train and evaluate with each dataset using ten different dataset splits. For each split, we train two TPP models, LogNormMix and TruncNorm (see §IV-D2). We optimize the former with the loss function $\mathcal{L}_{\text{NLL}}$ and the latter with $\mathcal{L}_{\text{MAE}}$. Both loss functions evaluate the prediction of the next function class m_i with the negative log-likelihood (NLL), but differ in the evaluation of the predicted τ_i. As shown in §IV-D1 and §IV-D2, LogNormMix

TABLE 4. The parameter configurations of the Sampler from IV-C which were used to generate the datasets for the 14 applications. For each application, one data set without and one data set with almost exclusively cold function calls were generated.

	no cold starts		almost only cold starts			
application	l (seconds)	u (seconds)	l (seconds)	u (seconds)	b	w (seconds)
sequence	0.3	0.6	10^{-4}	10^{-4}	50	120
parallel_small	0.3	0.6	10^{-4}	10^{-4}	50	120
tree_small	0.3	0.6	10^{-4}	10^{-4}	50	120
fanout_small	0.3	0.6	10^{-4}	10^{-4}	20	120
parallel_large	0.9	1.4	0.01	0.01	20	120
tree_large	0.9	1.4	0.01	0.01	20	120
fanout_large	0.9	1.4	0.01	0.01	20	120
sequence_rand	0.3	0.6	10^{-4}	10^{-4}	50	120
parallel_small_rand	0.3	0.6	10^{-4}	10^{-4}	50	120
tree_small_rand	0.3	0.6	10^{-4}	10^{-4}	50	120
fanout_small_rand	0.3	0.6	10^{-4}	10^{-4}	20	120
parallel_large_rand	0.9	1.4	0.01	0.01	20	120
tree_large_rand	0.9	1.4	0.01	0.01	20	120
fanout_large_rand	0.9	1.4	0.01	0.01	20	120

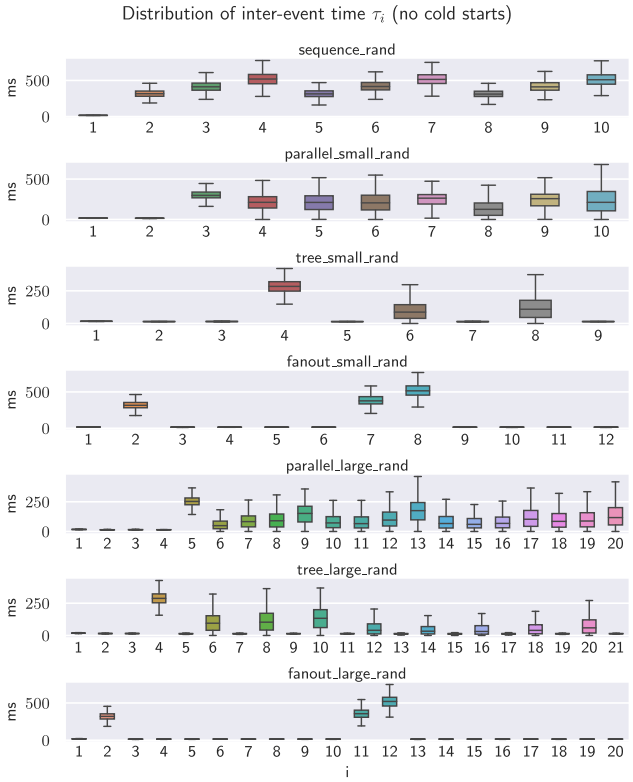


FIGURE 22. Distribution of the inter-event time τ_i for $i \in \{1, \ldots, N-1\}$ in the generated random variant datasets without cold starts, where N is the number of functions of the FaaS composition.

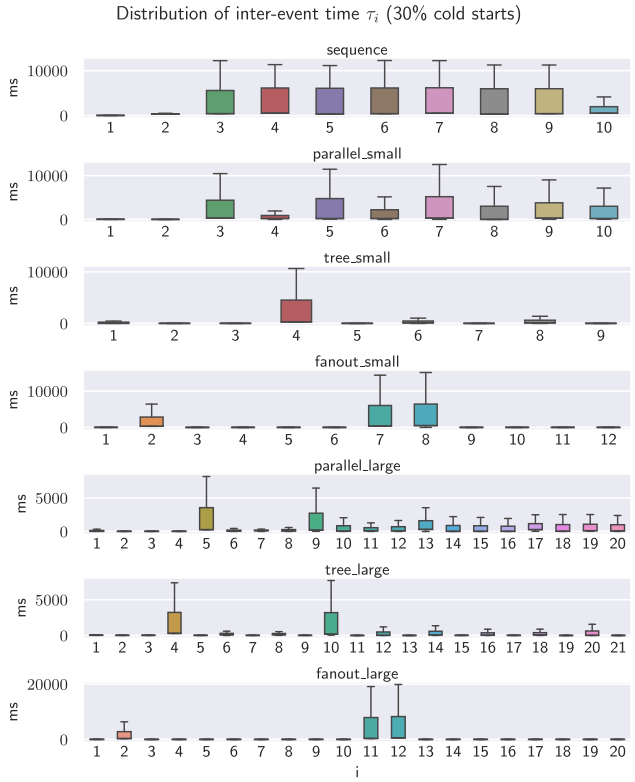


FIGURE 23. Distribution of the inter-event time τ_i for $i \in \{1, \ldots, N-1\}$ in the generated datasets with 30% cold starts, where N is the number of functions of the FaaS composition.

predicts τ_i with the conditional probability distribution $f_i^*(\tau_i)$, whereas TruncNorm provides a concrete value for τ_i, which we denote with τ_i^{pred}. The loss function $\mathcal{L}_{\text{NLL}}$ evaluates the distribution $f_i^*(\tau_i)$ using the NLL, whereas the loss function $\mathcal{L}_{\text{MAE}}$ computes the mean absolute error (MAE) for τ_i^{pred}. To derive $\mathcal{L}_{\text{NLL}}$, we denote by $x = \{e_1 = (\tau_1, m_1), \ldots, e_N = (\tau_N, m_N)\}$ an event sequence representing a trace of

invocations. The likelihood of the trace is defined by

$$p(x|\theta) = \prod_{i=1}^{N} \left[f_i^*(\tau_i, m_i)\right] S_{N+1}^* \tag{13}$$

Assuming that the inter-event time τ_i and mark m_i are independent, we obtain our loss function:

$$\begin{aligned} p(x|\theta) &= \prod_{i=1}^{N} \left[f_i^*(\tau_i, m_i)\right] S_{N+1}^* \\ &= \prod_{i=1}^{N} \left[f_i^*(\tau_i) f_i^*(m_i)\right] S_{N+1}^* \\ \mathcal{L}_{\text{NLL}}(\theta) &= -\log p(x|\theta) \\ &= -\sum_{i=1}^{N} \left[\log f_i^*(\tau_i) + \log f_i^*(m_i)\right] - \log S_{N+1}^* \end{aligned} \tag{14}$$

The model parameters are optimized by minimizing the loss function. For this, we use the optimization algorithm Adam [52] with a learning rate of 10^{-3} and minibatch size of 64. We train LogNormMix and TruncNorm up to 2000 and 4000 epochs, respectively, where an epoch describes the iteration over the entire training data. If the loss does not decrease after 100 and 200 epochs, respectively, with respect to the validation set, we abort the training and pick the model with the lowest loss with respect to the validation set. To reduce the effect of overfitting, we apply L2 regularization with 10^{-5} on the model parameters. To model $f_i^*(\tau_i)$, LogNormMix uses a log-normal mixture distribution with $K = 64$ components. According to [31], the parameter K does not have much impact on the performance of the model, which is why we do not test any other values. As RNN architecture, we use a gated recurrent unit (GRU) [53] with a hidden state vector in $\mathbb{R}^{64}$. As described in §IV-D1, we represent the mark m_i and the cold-start feature c_i with embedding vectors in $\mathbb{R}^{32}$.

E. PERFORMANCE QUALITY MEASURES

The TPP **LogNormMix** predicts the conditional probability distribution $f_i^*(\tau_i)$ over the inter-event time τ_i and the conditional categorical distribution $f_i^*(m_i)$ over the marks m_i. We use the negative log-likelihood (NLL) to evaluate the predicted distributions with respect to the test dataset $x = \{(\tau_1, m_1), \ldots, (\tau_N, m_N)\}$. Using NLL_{time}, NLL_{mark}, and $\text{NLL}_{\text{total}}$, we evaluate the distribution over τ_i, m_i, and both variables, respectively. The NLL quality measures are defined as follows:

$$\begin{aligned} \text{NLL}_{\text{time}} &= -\frac{1}{N} \sum_{i=1}^{N} \log f_i^*(\tau_i) - \log S_{N+1}^* \\ \text{NLL}_{\text{mark}} &= -\frac{1}{N} \sum_{i=1}^{N} \log f_i^*(m_i) \\ \text{NLL}_{\text{total}} &= \text{NLL}_{\text{time}} + \text{NLL}_{\text{mark}} \end{aligned} \tag{15}$$

It is worth noting here that a single NLL value has little explanatory power. That is, we cannot evaluate whether a value is "good" without referring to other values. For this reason, the relative differences between the NLL values for different datasets is analyzed [31].

The *accuracy* is another quality measure that evaluates LogNormMix's predictive capability of the mark m_i. It describes the fraction of correctly predicted marks, such that 1.0 is the optimal and 0.0 is the worst value for this metric. We obtain the predicted class c^{pred} of the mark m_i with

$$c^{\text{pred}} = \arg\max_{c} \pi_{i,c}, \tag{16}$$

where $\pi_{i,c}$ describes the probability that the i^{th} function invocation is of class c (see §IV-D3). We expect a correlation between the measure NLL_{mark} and the accuracy. The accuracy measure evaluates the TPP according to its capability to predict a single class for the next function invocation. The FaaS platform can use the prediction to scale the corresponding class in advance.

The TPP **TruncNorm** predicts a single value for the inter-event time τ_i and also, like LogNormMix, a conditional categorical distribution over m_i (see §IV-D2). We evaluate the predicted value for the inter-event time, denoted as τ_i^{pred}, by computing the mean absolute error

$$\text{MAE} = \frac{1}{N} \sum_{i=1}^{N} |\tau_i - \tau_i^{\text{pred}}| \tag{17}$$

for the test dataset. Besides the mean value of the absolute errors, the distribution of the errors is interesting. This give us information if the time predicted for the invocation was too early or too late. Therefore, we compute for the test dataset the errors using the following equation:

$$E_i = \tau_i^{\text{pred}} - \tau_i, \; i \in \{1, \ldots, N\} \tag{18}$$

and visualize their distribution. Here, a negative value indicates that the predicted time was too early.

Like LogNormMix, TruncNorm also predicts a distribution over the mark m_i. However, in contrast to LogNormMix, we do not evaluate this distribution because the results of the two TPPs would be similar. The reason for this is that both predict their mark distribution conditionally independent of the time. Therefore, the distribution is only conditioned on the history embedding h_i produced by an RNN in both TPPs.

VI. RESULTS

We evaluated our TPP models LogNormMix (§IV-D1) and TruncNorm (§IV-D2) with respect to various applications (§V-A), which differ in structure, number of functions, and randomization of the function's sleep command. In this section, we present the results of both the datasets (with and without cold starts). We evaluated predicted distributions with the negative log-likelihood (NLL) and predicted single values with the mean absolute error (MAE). For both quality measures, lower values are better and zero is

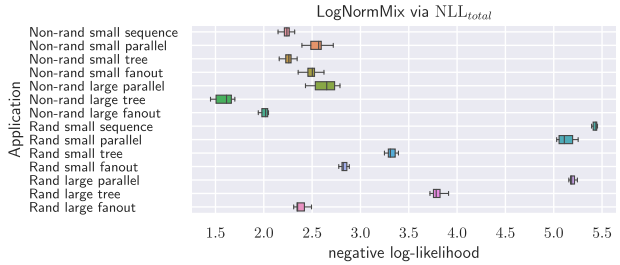


FIGURE 24. LogNormMix evaluated via the negative log-likelihood of the mark (function class) and inter-event time prediction (NLL_{total}) with respect to the test dataset, with a lower value being better and zero being optimal. The datasets have no cold starts.

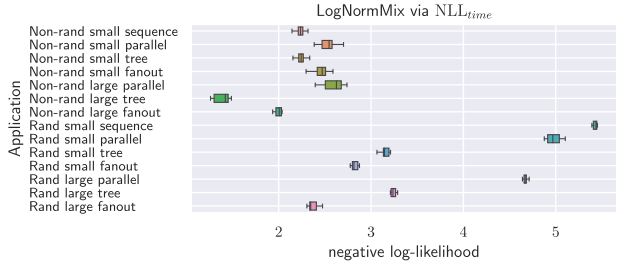


FIGURE 25. LogNormMix evaluated via the negative log-likelihood of the inter-event time prediction (NLL_{time}) with respect to the test dataset, with a lower value being better and zero being optimal. The datasets have no cold starts.

optimal. As described in §V-E, a single NLL value has little explanatory power. Instead, the differences between values for different applications are of interest. In contrast, a single MAE value is meaningful and evaluable even without comparison to other values.

A. PREDICTIONS ON DATASETS WITHOUT COLD STARTS

In this section, we present the results of prediction on dataset without cold starts.

1) LogNormMix VIA NLL_{total}

LogNormMix predicts a distribution for the inter-event time τ_i and for the mark m_i, i.e., for the functional class. Using NLL_{total} from equation (15), we evaluated both distributions combined and present the results in Figure 24. Looking at the NLL measures NLL_{total},NLL_{time}, NLL_{mark} in Figure 24, Figure 25 and Figure 26, we notice that NLL_{time} has a much higher proportion of NLL_{total} than NLL_{mark}. For example, the application tree_large_rand has a value of about 3.8 for NLL_{total}. From this value, about 3.25 accounts for NLL_{time} and about 0.55 for NLL_{mark}. Thus, we can infer that it is much more challenging for LogNormMix to predict the time than the functional class. Future research should therefore prioritize improving the prediction of the inter-event time τ_i.

2) LogNormMix VIA NLL_{time}

We evaluated the inter-event time with NLL_{time} from equation (15) and show the results in Figure 25. We draw the following conclusions:

a: DIFFERENCES BETWEEN RANDOMIZED AND NON-RANDOMIZED APPLICATIONS

A look at the metric NLL_{time} in Figure 25 shows that LogNormMix performed better for non-randomized applications than for randomized ones. This was expected since the function duration was drawn from a gamma distribution instead of being constant. When we look at the distributions of the inter-event time τ_i in Figure 21 and Figure 22, we see that the distributions for randomized applications have a higher variance than for non-randomized ones. This higher variance makes prediction more challenging for the TPPs. The fact that the function duration is drawn independently of the gamma distribution also impairs the prediction. In a real-world application, some dependence between function execution times can be assumed. For example, if the execution time of a function is longer than usual due to a high load on the FaaS platform, it is likely that subsequent functions will also execute longer than usual. The information about the overload would be encoded in the resulting higher inter-event times, thus improving time prediction of the TPP.

Furthermore, we see in Figure 25 that the results for applications with a small proportion of parallel functions suffered particularly from randomization. For example, this is evident for the applications sequence_small, which has no parallel functions, and fanout_large, which has a high proportion of parallel functions. While the TPP performed marginally worse in the non-randomized case for sequence_small than for fanout_large (difference of approximately 0.25), this difference is much more significant in the randomized case (difference of approximately 3). This is because in sequence, each function has a successor that is invoked after a sleep command completes. This means that there is a randomized sleep command between every two invocations, which makes the predictions more difficult. In contrast, the parallel functions in fanout (e.g., functions f21 to f29 in 20) are invoked as a sequence without any intermediate randomized sleep commands, so the results in fanout are less affected by the randomization. This is also illustrated in 22, where we see in the diagram of sequence_rand that the inter-event time distributions for all i have high variance. In comparison, in the diagram of fanout_large_rand, only the distributions of τ_2, τ_{11} and τ_{12} have high variance. The other distributions with a low variance refer to parallel functions (e.g., f21 to f29). Transferring this knowledge to real-world scenarios, we can say that applications with parallel functions without uncertainty between invocations, e.g., caused by a database query with high variance in execution time, facilitate time prediction.

b: DIFFERENCES BETWEEN SMALL AND LARGE APPLICATIONS

It is interesting that in Figure 25 the result for the applications parallel_small and parallel_large are equal in the non-randomized case, but the result for parallel_large is slightly better in the randomized case. This contradicts our

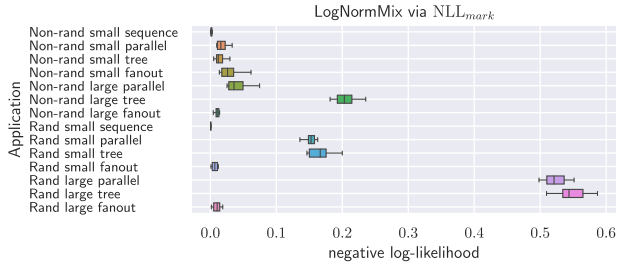


FIGURE 26. LogNormMix evaluated via the negative log-likelihood of the mark (function class) prediction (NLL_{mark}) with respect to the test dataset, with a lower value being better and zero being optimal. The datasets have no cold starts.

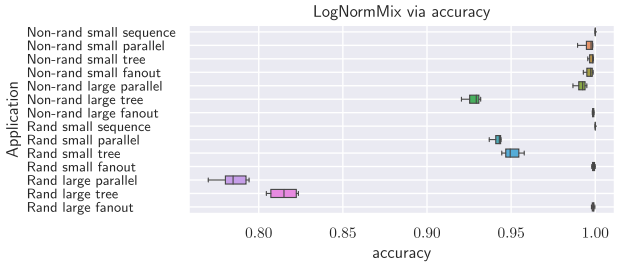


FIGURE 27. LogNormMix evaluated via the accuracy of the mark (function class) prediction with respect to the test dataset, with a higher value being better and 1.0 being optimal. The datasets have no cold starts.

assumption (presented in Figure 18) that a higher number of parallel function branches would affect the prediction performance for the inter-event time. Moreover, we see in Figure 25 that LogNormMix performed better for tree_large than for tree_small in the non-randomized case and that the results of both applications are equal in the randomized case. This indicates that a higher tree depth has no negative influence on the prediction performance. In addition, we see in Figure 25 that the prediction performance for fanout_large was better than for fanout_small, which is due to the higher proportion of parallel functions. This also shows us that scaling the number of parallel functions in the application structure does not harm the time prediction performance of the TPP.

3) LogNormMix VIA NLL_{mark}

In addition, we evaluated the function class distributions separately with NLL_{mark} from equation (15) and show the results in Figure 26. The NLL_{mark} measure in Figure 26 shows that LogNormMix performed well for the majority of the applications, i.e., the values are close to zero. However, exceptions are the results for tree_large and the randomized versions of parallel and tree. A drop in performance between the small and the large versions can be observed for the two latter applications, parallel and tree. Characteristic for the structure of these applications is a high number of parallel function branches. This indicates that the function class prediction is challenging for applications with this structure. Thus, the assumption in Figure 18 holds for function class prediction, in contrast to the time prediction as described previously. Since the function class order is the same for all traces, LogNormMix performed best for the application sequence with a near-zero NLL value.

4) LogNormMix VIA ACCURACY

Another measure that evaluates the performance in terms of the function class prediction is the accuracy from equation (16). The measure is defined in the range [0.0, 1.0], where 1.0 is the best (all classes were predicted correctly) and 0.0 is the worst. We show the results of LogNormMix with respect to this measure in Figure 27. The results of the accuracy in Figure 27 reflect the results of the NLL_{mark} measure, though the values are more interpretable. We see that LogNormMix achieved an accuracy close to 1.0 for the majority of the applications, meaning that almost all invocations were classified correctly. Analogous to NLL_{mark}, LogNormMix achieved worse results for the randomized versions of parallel and tree. However, an accuracy of above 0.93 was still achieved for tree_large, parallel_small_rand, and tree_small_rand, which is acceptable. An accuracy of about 0.8 for parallel_large_rand and tree_large_rand, on the other hand, could further be improved by collecting more data.

5) TruncNorm VIA MAE

TruncNorm predicts a single value for the inter-event time τ_i. We evaluated this prediction using the MAE from equation (17) and show the results in Figure 28. Figure 28 shows the results of TruncNorm's inter-event time predictions in terms of the mean absolute error. The results are similar to those for the NLL_{time} measure, i.e., they exhibit the same patterns: better results for non-randomized applications than for randomized ones, smaller drop in performance due to randomization for applications with a higher proportion of parallel functions (e.g., tree and fanout), and no negative impact on the results when scaling the application structure from small to large. For the non-randomized applications, all MAE values are below 20ms, which is reasonable given the duration of the sleep command of 300s to 500ms. This also applies to the randomized applications, excluding the applications parallel and sequence. For these two applications, the values of about 40 and 60ms can be improved by providing more features for the TPP in future work.

6) TruncNorm VIA E_i

In addition, we calculated the errors E_i from equation (18) for the entire test dataset and visualize their distribution in Figure 29. Here, lower absolute values are better and zero is optimal. The error distributions of the inter-event time predictions in Figure 29 show that TruncNorm performed well for most applications. However, analogous to the results for the mean absolute error, the performance for the randomized versions of sequence and parallel was relatively poor. Here, the error distributions have higher variances

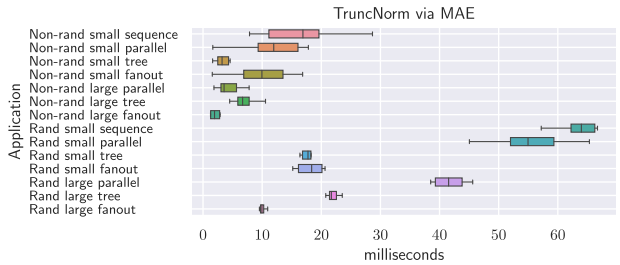


FIGURE 28. TruncNorm evaluated via the mean absolute error (MAE) of the inter-event time prediction with respect to the test dataset, with a lower value being better and zero being optimal. The datasets have no cold starts.

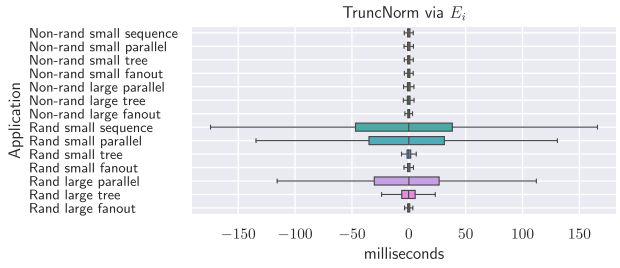


FIGURE 29. TruncNorm evaluated via the distribution of the errors between the predicted and true inter-event time ($E_i = \tau_i^{\text{pred}} - \tau_i$) with respect to the test dataset, with a lower absolute value being better and zero being optimal. A negative value indicates that the predicted time for the invocation was too early. The datasets have no cold starts.

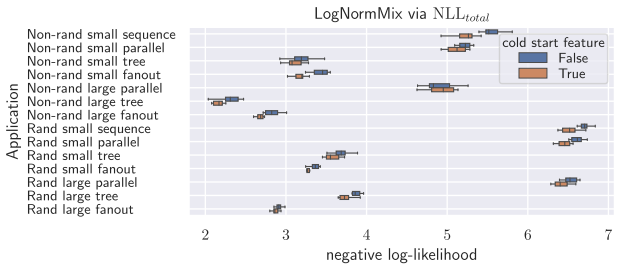


FIGURE 30. LogNormMix evaluated via the negative log-likelihood of the mark (function class) and inter-event time prediction ($\text{NLL}_{\text{total}}$) with respect to the test dataset, with a lower value being better and zero being optimal. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.

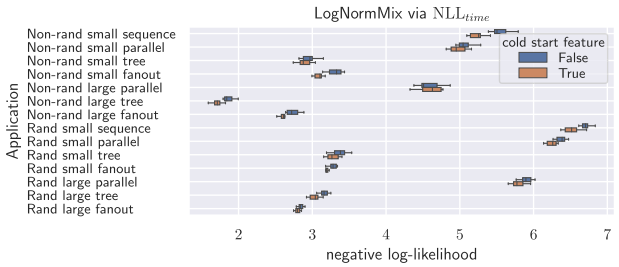


FIGURE 31. LogNormMix evaluated via the negative log-likelihood of the inter-event time prediction (NLL_{time}) with respect to the test dataset, with a lower value being better and zero being optimal. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.

than for the other applications. Notably, all distributions are symmetric and centered in zero.

B. FUNCTION PREDICTION ON DATASETS WITH COLD STARTS

This section repeats the evaluation from §VI-A but with the difference that 30% of the function invocations were cold starts. Another difference is that we trained and evaluated the models twice for each application. Once the cold start feature $c_i \in \{0, 1\}$ was included in the event representation, i.e. (τ_i, m_i, c_i), and once it was not, i.e. (τ_i, m_i). The feature indicates whether the i^{th} invocation was a cold-start.

1) LogNormMix VIA NLL_{time}

We evaluated the inter-event time with NLL_{time} from equation (15) and present the results in Figure 31. The results regarding NLL_{time} in Figure 31 are similar to the results for this measure without cold starts in Figure 25, yet with slightly poorer performance. However, one difference is that LogNormMix also performed relatively poorly for the non-randomized versions of the applications sequence and parallel. At the same time, this was not the case for the datasets without cold starts. Looking at the inter-event time distributions in the cold start datasets, illustrated in Figure 23, we see that they have a high variance for the applications sequence, parallel_small, and parallel_large. This obviously affects the prediction performance. In comparison, the inter-event time distributions in the datasets without cold starts, illustrated in Figure 21, have almost no variance. The high variance of the inter-event time distributions is caused by the high variance of the waitTime distributions shown in Figure 32. These very high waitTime values, up to 10 seconds, are caused by the high load imposed on OpenWhisk to enforce cold starts. Furthermore, it can be seen in Figure 31 that the enabled cold start feature slightly improved the prediction results. However, the improvement is marginal as the major uncertainty in inter-event time prediction comes from the waitTime values with high variance. The feature does provide the information that a cold start occurred and that a higher inter-event time can be expected, but due to the high variance of the waitTime, prediction is still challenging.

2) LogNormMix VIA NLL_{mark}

Looking at the results of the NLL_{mark} measure in Figure 33, it is noticeable that they are slightly worse than the results for the datasets without cold starts in Figure 26. This implies that the function class prediction was also affected by the higher variance of the inter-event time (see Figure 23) caused by the higher variance of the waitTime (see Figure 32). Similar to the results without cold starts, LogNormMix performed worse for the applications parallel and tree due to their structure with parallel function branches. Enabling the cold start features led to improvements, but as with the results for the NLL_{time} measure, these were marginal.

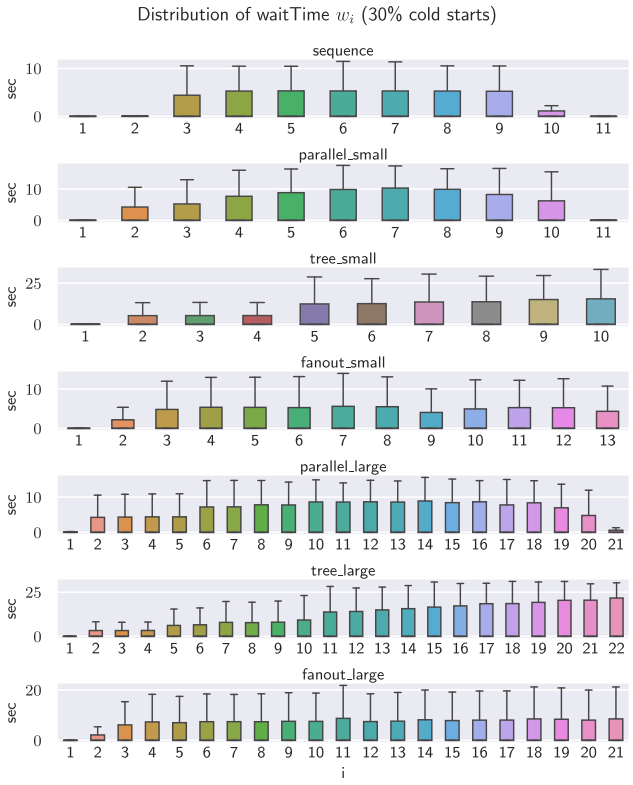


FIGURE 32. Distribution of the waitTime w_i for $i \in \{1, \ldots, N\}$ in the generated datasets with 30% cold starts, where N is the number of functions of the FaaS composition.

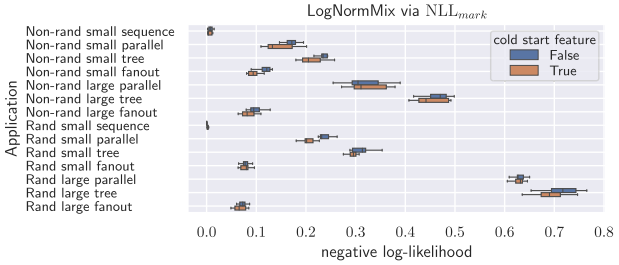


FIGURE 33. LogNormMix evaluated via the negative log-likelihood of the mark (function class) prediction ($\mathrm{NLL}_{\mathrm{mark}}$) with respect to the test dataset, with a lower value being better and zero being optimal. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.

3) LogNormMix VIA ACCURACY

Analogous to the drop in performance for $\mathrm{NLL}_{\mathrm{mark}}$ due to cold starts, this is also the case for the results with the measure accuracy in Figure 34. Especially, the results for the non-randomized versions of parallel and tree were affected by the high variance of waitTime. For example, the results for parallel_small and tree_small decreased by approximately 0.06 and 0.08, respectively, compared to the results for the datasets without cold starts in Figure 27. The highest decrease in accuracy of approximately 0.11 was experienced for the application parallel_large. Even though the performance generally decreased due to the cold starts,

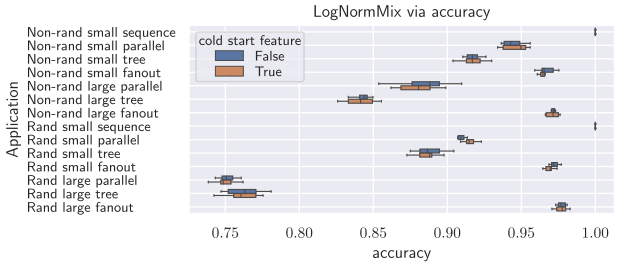


FIGURE 34. LogNormMix evaluated via the accuracy of the mark (function class) prediction with respect to the test dataset, with a higher value being better and 1.0 being optimal. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.

the results are still good. Thus, the accuracy for parallel_large_rand and tree_large_rand decreased by only about 0.03 and 0.05, respectively. Similarly, parallel_small_rand and tree_small_rand decreased by about 0.03 and 0.06, respectively. The accuracy for all versions of fanout decreased by at most 0.03.

4) TruncNorm VIA MAE

Similar to the decrease in performance with respect to $\mathrm{NLL}_{\mathrm{time}}$ due to the cold starts, a decrease in performance with respect to the mean absolute error in Figure 28 is also observed. The high variance of the waitTime in the cold start datasets significantly affected the prediction performance of TruncNorm, resulting in MAEs of more than 400 ms. Compared to the results for the datasets without cold starts in Figure 28, where the MAE was below 20 ms for most applications, this is a significant increase. The MAE is especially high for the applications sequence and the small versions of parallel, with values between 1000 and 1500ms. This could be related to the fact that the structures of these applications have a low proportion of parallel functions. Therefore, as seen in Figure 23, most of the inter-event time distributions have a high variance. In contrast, the performance of TruncNorm is relatively good for the large versions of parallel. This is surprising since LogNormMix struggled to predict the time for these applications, as can be seen in Figure 31. We can also observe that the cold start feature improves the prediction performance, especially for the small versions of sequence and parallel.

5) TruncNorm VIA E_i

The error distributions of the inter-event time predictions in Figure 36 show that TruncNorm achieved good results for most applications, i.e., absolute values close to zero. Analogous to the results with mean absolute error in Figure 35, the performance for the small versions of parallel and especially sequence is relatively poor as the error distributions have high variance. Furthermore, the error distributions show that most of the errors were negative. By the definition $E_i = \tau_i^{\mathrm{pred}} - \tau_i$, a negative error signifies that the predicted time for the invocation was too early. This is since the high waitTime delayed the invocation.

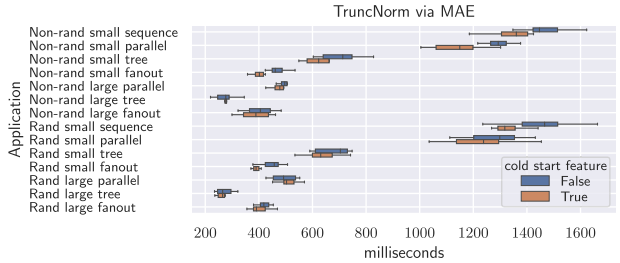


FIGURE 35. **TruncNorm evaluated via the mean absolute error (MAE) of the inter-event time prediction with respect to the test dataset, with a lower value being better and zero being optimal. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.**

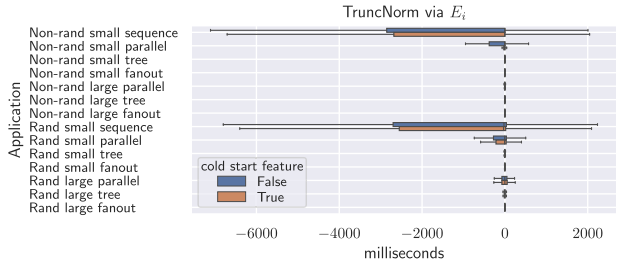


FIGURE 36. **TruncNorm evaluated via the distribution of the errors between the predicted and true inter-event time ($E_i = \tau_i^{\text{pred}} - \tau_i$) with respect to the test dataset, with a lower absolute value being better and zero being optimal. A negative value indicates that the predicted time for the invocation was too early. 30% of the invocations were cold starts, where for each application, the TPP was trained and evaluated once with the cold start feature c_i enabled and once with it disabled.**

C. TRAINING DURATION

Table 5 and Table 6 show the training duration of the TPP models LogNormMix (§IV-D1) and TruncNorm (§IV-D2) for different datasets using the technical infrastructure from §V-B. The training duration was measured in process time (see Table 5) and in number of epochs (see Table 6), where an epoch describes one iteration over the entire training dataset. Each measured value is the average of ten training iterations with different train/validation/test dataset splits (§V-D). We trained LogNormMix using the loss function $\mathcal{L}_{\text{NLL}}$ from equation (14), which evaluates the prediction for the inter-event time τ_i with the negative log-likelihood, and TruncNorm with the loss function $\mathcal{L}_{\text{MAE}}$ from equation (17), which evaluates the prediction for τ_i with the mean absolute error. We trained both models on datasets that contained no cold function invocations and on datasets with 30% of the invocations being cold starts (see V-C). We can provide the TPP with the optional feature c_i, which indicates whether the current invocation was a cold start. We trained with each cold start dataset twice, once with the feature enabled and once with it disabled. In this way, we also separated our measurements of the training duration.

VII. RELATED WORK

With the advent of serverless computing, there is a significant amount of research aimed at optimizing cloud computing resource utilization [54]–[57]. There has been some work on the performance profiling of various FaaS platforms. Wang *et al.* [58] performed an in-depth study of resource management and performance isolation with three popular serverless computing providers: AWS Lambda, Azure Functions, and Google Cloud Functions. Their analysis demonstrates a reasonable difference in performance between the FaaS platforms. Furthermore, Shahrad *et al.* [59] studied the architectural implications of serverless computing and pointed out that exploitation of system architectural features like temporal locality and reuse are hampered by the short function runtimes in FaaS. Chadha *et al.* [60] examine the underlying processor architectures for Google Cloud Functions (GCF) and determine the optimization of FaaS functions using Numba can improve performance by and save costs on average.

Many approaches have been proposed to reduce the occurrences of cold starts. A solution to reduce cold starts is presented in [11]. The developer defines here in a configuration file which function classes communicate with each other. A middleware acts as an early warning mechanism and is deployed together with the FaaS application. When a function is invoked, the middleware sends so-called hinting messages to the subsequent functions defined in the configuration file. Thus, these are initialized early and cold starts are avoided. The FaaS orchestration platform OpenWhisk [25] also follows a pooling approach by consistently providing a pool of so-called stem cell containers. These consist only of a base image, i.e., without function code and libraries, enabling faster function initialization. AWS Lambda employs a fixed-time "keep-alive" policy to keep resources in memory after function executions [61]. Lin and Glikson [22] reduces cold starts by providing a pool of already initialized functions of a certain class. This pool can be used in case of upscaling. Oakes *et al.* [62] reduces the size of the function deployment by separating function and library code. Libraries are deployed separately and can be used by multiple functions concurrently. Defuse in [63] leverages the dependencies among serverless functions to schedule them directly. While these described approaches are often based on pooling or faster function initialization, our approach tries to reduce cold starts by predicting function invocations using TPPs. This approach also has the advantage that the predictions can be used to optimize the function-server assignment.

Pawlik *et al.* [64] state that to assess the feasibility of running an application on the FaaS platform, we have to determine the SLO of the application. This can be achieved by constructing a reliable performance model capable of analyzing a function performance, which requires knowledge about the performance of the infrastructure. Cloud service providers abstract details such as the number of cores, memory available, and network I/O capacity in the underlying hardware, usually limiting the available information to function time limit, maximum memory. The allocated memory also affects the provisioned CPU quota [65]. In our previous work [66], we developed a tool for estimating the maximum number of

TABLE 5. Training times of the TPP models LogNormMix and TruncNorm measured in process time (seconds).

process time (seconds)	LogNormMix with loss $\mathcal{L}_{\text{NLL}}$			TruncNorm with loss $\mathcal{L}_{\text{MAE}}$		
	no cold starts	30% cold starts		no cold starts	30% cold starts	
	feature disabled	feature disabled	feature enabled	feature disabled	feature disabled	feature enabled
sequence	68s	37s	36s	314s	592s	607s
parallel_small	111s	67s	61s	451s	593s	686s
tree_small	72s	72s	55s	355s	638s	650s
fanout_small	28s	40s	41s	287s	615s	652s
parallel_large	143s	80s	58s	465s	641s	681s
tree_large	38s	87s	94s	185s	489s	587s
fanout_large	59s	42s	39s	295s	651s	684s
sequence_rand	65s	57s	41s	249s	619s	841s
parallel_small_rand	40s	50s	39s	257s	659s	607s
tree_small_rand	50s	75s	48s	152s	588s	602s
fanout_small_rand	99s	79s	50s	276s	614s	648s
parallel_large_rand	34s	33s	31s	142s	603s	697s
tree_large_rand	43s	67s	59s	165s	575s	630s
fanout_large_rand	104s	79s	51s	356s	644s	686s

TABLE 6. Training times of TPP models LogNormMix and TruncNorm measured in the number of epochs.

number of epochs	LogNormMix with loss $\mathcal{L}_{\text{NLL}}$			TruncNorm with loss $\mathcal{L}_{\text{MAE}}$		
	no cold starts	30% cold starts		no cold starts	30% cold starts	
	feature disabled	feature disabled	feature enabled	feature disabled	feature disabled	feature enabled
sequence	428	237	230	2102	4000	4000
parallel_small	698	422	379	3034	4000	4000
tree_small	450	425	318	2395	4000	4000
fanout_small	177	245	247	1859	4000	4000
parallel_large	852	478	339	2855	3953	3992
tree_large	228	516	544	1134	2987	3433
fanout_large	354	256	226	1815	4000	4000
sequence_rand	345	360	217	1646	4000	4000
parallel_small_rand	240	294	246	1303	4000	4000
tree_small_rand	323	479	302	1029	4000	4000
fanout_small_rand	599	478	301	1756	4000	4000
parallel_large_rand	202	199	184	872	3715	4000
tree_large_rand	259	395	346	998	3523	3669
fanout_large_rand	620	475	301	2179	3981	4000

requests a microservice can handle when it is sandboxed. This capacity estimation of microservices enables us to ensure the flexibility of the capacity planning for a microservices application. While these approaches work at the level of function modeling, but with TppFaaS we are modeling the FaaS application and user-workload invocations, which allows to better optimize the function-server assignment and anomalies detection.

VIII. CONCLUSION AND FUTURE SCOPE

This work has shown that neural temporal point processes (TPPs) effectively model the time and class of function invocations in FaaS compositions. For this purpose, we introduced TppFaaS, a system for implementing FaaS compositions and generating data from them that can be used to train and test neural TPPs. In this data, function invocations are represented by the timing of their function trigger events. In addition, the data contains meta-information such as the function class and the cold start initialization time. Using TppFaaS, we implemented multiple versions of FaaS compositions with a sequential, parallel, and tree-shaped structure. The versions differed in the randomization of the function's sleep command and the number of functions. Based on the compositions, datasets with and without cold starts were generated. With these datasets, we trained and tested the two TPPs LogNormMix and TruncNorm.

It was shown that both models managed to capture the latent temporal dynamics of the different FaaS compositions. We observed that the performance of the time prediction was not affected by scaling the composition structure. Moreover, the function class prediction proved to be more challenging for compositions involving parallel executed function branches. For datasets without cold starts, the TPPs performed well with respect to all measures. Here, LogNormMix achieved an accuracy of over 0.94 for most applications,

i.e., the class of 94% of the invocations was predicted correctly. Also, the mean absolute error of TruncNorm's time prediction was below 22ms for most applications. However, the predictions for the datasets with cold starts were more challenging. Here, TruncNorm achieved a mean absolute error between 200 and 750ms for most applications. The high errors resulted from the high variance of the waitTime, which measures the time an invocation request waits for execution in the internal OpenWhisk system. In addition, LogNormMix's function class prediction performance declined for the cold start datasets. Nevertheless, an accuracy above 0.85 was achieved for most applications, which is still satisfactory. The cold start feature, which indicates whether a cold start occurred, improved the results only marginally. This is because the most uncertainty in the prediction is caused by the high variance of the waitTime and not by the variance of the cold start initialization time. Future work may therefore provide additional features to the TPP to assist in the estimation of the waitTime. Such as the number of invoker resources or the number of invocation requests waiting in the OpenWhisk system. In general, predicting the time was more difficult than predicting the functional class for datasets with and without cold starts. Therefore, future work should prioritize improving the prediction of the time rather than the functional class.

Further future work could involve the application of the probability distribution predicted by LogNormMix for anomaly detection. In FaaS compositions, anomalies can occur in the form of abnormally short or long function executions or unexecuted functions. A trace containing such an anomaly would have a lower probability with respect to the probability distribution predicted by LogNormMix. In order to capture anomalies in execution duration, the time at which a function ends must also be recorded. Thus, each function invocation is represented by the trigger event and the event marking the end of execution. Another idea for future research might be the inclusion of conditional function invocations. Here, the input object of the function could be encoded as a vector representation provided to the TPP as a feature. Similarly, when a function performs a database query, the SQL statement could be encoded as a vector using natural language processing. This could assist the TPP in estimating the execution time of the query.

REFERENCES

[1] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using FaaS fabric," in *Proc. 6th Int. Workshop Serverless Comput.*, New York, NY, USA, Dec. 2020, pp. 49–54.

[2] J. Carreira, P. Fonseca, A. Tumanov, A. Zhang, and R. Katz, "Cirrus: A serverless framework for end-to-end ML workflows," in *Proc. ACM Symp. Cloud Comput.*, Nov. 2019, pp. 13–24.

[3] V. Shankar, K. Krauth, K. Vodrahalli, Q. Pu, B. Recht, I. Stoica, J. Ragan-Kelley, E. Jonas, and S. Venkataraman, "Serverless linear algebra," in *Proc. 11th ACM Symp. Cloud Comput.*, New York, NY, USA, Oct. 2020, pp. 281–295.

[4] R. Chard, T. J. Skluzacek, Z. Li, Y. Babuji, A. Woodard, B. Blaiszik, S. Tuecke, I. Foster, and K. Chard, "Serverless supercomputing: High performance function as a service for science," *CoRR*, vol. abs/1908.04907, 2019. [Online]. Available:http://arxiv.org/abs/1908.04907

[5] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proc. Symp. Cloud Comput.*, Sep. 2017, pp. 445–451.

[6] Microsoft. *Azure Functions*. Accessed: Sep. 4, 2021. [Online]. Available: https://azure.microsoft.com/de-de/services/functions/

[7] Google. *Cloud Functions*. Accessed: Sep. 4, 2021. [Online]. Available: https://cloud.google.com/functions

[8] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[9] P. Castro, V. Ishakian, V. Muthusamy, and A. Slominski, "The rise of serverless computing," *Commun. ACM*, vol. 62, no. 12, pp. 44–54, Nov. 2019.

[10] D. Taibi, J. Spillner, and K. Wawruch, "Serverless computing-where are we now, and where are we heading?" *IEEE Softw.*, vol. 38, no. 1, pp. 25–31, Dec. 2021.

[11] D. Bermbach, A.-S. Karakaya, and S. Buchholz, "Using application knowledge to reduce cold starts in FaaS services," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 134–143.

[12] M. Grambow, T. Pfandzelter, L. Burchard, C. Schubert, M. Zhao, and A. D. Bermbach, "Befaas: An application-centric benchmarking framework for faas platforms," 2021, *arXiv:2102.12770*.

[13] Amazon Web Services. *AWS Step Functions*. Accessed: Sep. 4, 2021. [Online]. Available: https://aws.amazon.com/step-functions/

[14] Microsoft. *Azure Durable Functions*. Accessed: Sep. 4, 2021. [Online]. Available: https://docs.microsoft.com/en-us/azure/azure-functions/durable/

[15] OpenWhisk. *Apache Openwhisk Composer*. Accessed: Sep. 4, 2021. [Online]. Available: https://github.com/apache/openwhisk-composer

[16] Amazon Web Services. *AWS Step Functions Features*. Accessed: Sep. 4, 2021. [Online]. Available: https://aws.amazon.com/step-functions/features/

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. Rabbah, A. Slominski, and P. Suter, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Singapore: Springer, 2017, pp. 1–20.

[18] A. Eivy, "Be wary of the economics of "Serverless' cloud computing," *IEEE Cloud Comput.*, vol. 4, no. 2, pp. 6–12, Apr. 2017.

[19] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and A. V. Sukhomlinov, "Agile cold starts for scalable serverless," in *Proc. 11th USENIX Workshop Hot Topics Cloud Computing (HotCloud)*, 2019, pp. 1–6.

[20] R. Byrro. (2019). *Can we Solve Serverless Cold Starts?*. Accessed: Apr. 17, 2020. [Online]. Available: https://dashbird.io/blog/can-we-solve-serverless-cold-starts/

[21] J. Manner, M. EndreB, T. Heckel, and G. Wirtz, "Cold start influencing factors in function as a service," in *Proc. IEEE/ACM Int. Conf. Utility Cloud Comput. Companion (UCC Companion)*, Dec. 2018, pp. 181–188.

[22] P.-M. Lin and A. Glikson, "Mitigating cold starts in serverless platforms: A pool-based approach," 2019, *arXiv:1903.12221*.

[23] M. J. Hellerstein, J. Faleiro, E. J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless computing: One step forward, two steps back," *Proc. 9th Biennial Conf. Innov. Data Syst. Res. (CIDR)*, 2019, pp. 1–8.

[24] O. Shchur, A. C. Türkmen, T. Januschowski, and S. Günnemann, "Neural temporal point processes: A review," in *Proc. 30th Int. Joint Conf. Artif. Intell.*, Aug. 2021, pp. 4585–4593.

[25] OpenWhisk. *Openwhisk: Open Source Serverless Cloud Platform*. Accessed: May 16, 2021. [Online]. Available: https://openwhisk.apache.org

[26] *IBM Cloud Functions*. Accessed: Sep. 7, 2021. [Online]. Available: https://cloud.ibm.com/functions/

[27] OpenWhisk. *Openwhisk Annotations*. Accessed: Apr. 30, 2021. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/annotations.md

[28] OpenWhisk. *How Openwhisk Works*. Accessed: Sep. 7, 2021. [Online]. Available: https://github.com/apache/openwhisk/blob/master/docs/about.md

[29] W. Reese, "Nginx: The high-performance web server and reverse proxy," *Linux J.*, vol. 2008, p. 2, Sep. 2008.

[30] Apache Software Foundation. *Apache Kafka*. Accessed: Sep. 7, 2021. [Online]. Available: https://kafka.apache.org/

[31] O. Shchur, M. Biloš, and S. Günnemann, "Intensity-free learning of temporal point processes," 2019, *arXiv:1909.12127*.

[32] A. De, U. Upadhyay, and M. Gomez-Rodriguez, "Lecture notes for human-centered ML: Temporal point processes," Saarland Univ., Saarland, Germany, Tech. Rep. nil, 2019. [Online]. Available: http://courses.mpi-sws.org/hcml-ws18/lectures/TPP.pdf
[33] O. Shchur. (2020). *Temporal Point Processes 1: The Conditional Intensity Function*. Accessed: Jun. 16, 2021. [Online]. Available: https://shchur.github.io/blog/2020/tpp1-conditional-intensity/
[34] O. Shchur and S. Günnemann, "Lecture notes in machine learning for graphs and sequential data," Tech. Univ. Munich, Munich, Germany, Tech. Rep. nil, 2020.
[35] M.-A. Rizoiu, Y. Lee, and S. Mishra, "Hawkes processes for events in social media," *Frontiers Multimedia Res.*, vol. 17, pp. 191–218, Dec. 2017.
[36] A. G. Hawkes, "Spectra of some self-exciting and mutually exciting point processes," *Biometrika*, vol. 58, pp. 83–90, Apr. 1971.
[37] N. Du, H. Dai, R. Trivedi, U. Upadhyay, M. Gomez-Rodriguez, and L. Song, "Recurrent marked temporal point processes: Embedding event history to vector," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, New York, NY, USA, Aug. 2016, pp. 1555–1564.
[38] T. Omi, N. Ueda, and K. Aihara, "Fully neural network based model for general temporal point processes," in *Proc. 33rd Int. Conf. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2019, pp. 1–13.
[39] G. J. McLachlan and D. Peel, *Finite Mixture Models* (Wiley Series in Probability and Statistics). Hoboken, NJ, USA: Wiley, 2004.
[40] OpenTelemetry. *Opentelemetry Javascript*. Accessed: May 5, 2021. [Online]. Available: https://opentelemetry.io/docs/js/
[41] Zipkin. *Zipkin*. Accessed: Sep. 4, 2021. [Online]. Available: https://zipkin.io/
[42] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Comput. Commun. Rev.*, vol. 5, no. 1, pp. 3–55, Jan. 2001.
[43] S. Y. Park and A. K. Bera, "Maximum entropy autoregressive conditional heteroskedasticity model," *J. Econometrics*, vol. 150, no. 2, pp. 219–230, 2009.
[44] Serverless Framework. *Serverless Apache Openwhisk Plugin*. Accessed: May 11, 2021. [Online]. Available: https://github.com/serverless/serverless-openwhisk
[45] OpenTelemetry. *Opentelemetry: Data collection*. Accessed: Apr. 29, 2021. [Online]. Available: https://opentelemetry.io/docs/concepts/data-collection/
[46] OpenTelemetry. *Opentelemetry Collector: Oltp Receiver*. Accessed: Apr. 30, 2021. [Online]. Available: https://github.com/open-telemetry/opentelemetry-collector/tree/main/receiver/otlpreceiver
[47] OpenTelemetry. *Opentelemetry Collector: Batch Processor*. Accessed: Apr. 30, 2021. [Online]. Available: https://github.com/open-telemetry/opentelemetry-collector/tree/main/processor/batchprocessor
[48] OpenWhisk. *Openwhisk: Inittime in Duration*. Accessed: Apr. 30, 2021. [Online]. Available: https://github.com/apache/openwhisk/pull/3053/files/
[49] Zipkin. *Zipkin Architecture*. Accessed: May 12, 2021. [Online]. Available: https://zipkin.io/pages/architecture.html
[50] O. Shchur. (2021). *Loss With NLL of Mark and MAE of Inter-Event Time*. Accessed: Sep. 13, 2021. [Online]. Available: https://github.com/shchur/ifl-tpp/issues/14
[51] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Netw.*, vol. 61, pp. 85–117, Jan. 2015.
[52] P. Diederik Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *Proc. 3rd Int. Conf. Learn. Represent. (ICLR)*, 2015, pp. 1–15.
[53] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proc. Conf. Empirical Methods Natural Lang. Process. (EMNLP)*, 2014, pp. 1724–1734.
[54] M. Akin. *How Does Proportional CPU Allocation Work With AWS Lambda? | Opsgenie Engineering*. Accessed: Jul. 26, 2021. [Online]. Available: https://engineering.opsgenie.com/how-does-proportional-cpu-allocation-work-w% ith-aws-lambda-41cd44da3cac
[55] S. Kulkarni. *Optimize AWS Lambda Memory | Towards Data Science*. Accessed: Jul. 26, 2021. [Online]. Available: https://towardsdatascience.com/optimize-aws-lambda-memory-more-memory-doesnt% -mean-more-costs-51ba566fecc7
[56] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Softw., Pract. Exper.*, vol. 51, no. 9, pp. 1936–1963, Mar. 2021.
[57] A. Jindal, J. Frielinghaus, M. Chadha, and M. Gerndt, "Courier: Delivering serverless functions within heterogeneous FaaS deployments," in *Proc. 14th IEEE/ACM Int. Conf. Utility Cloud Comput.*, New York, NY, USA, Dec. 2021, pp. 1–10.
[58] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *Proc. USENIX Annu. Tech. Conf.*, 2018, pp. 133–146.
[59] M. Shahrad, J. Balkind, and D. Wentzlaff, "Architectural implications of Function-as-a-Service computing," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, Oct. 2019, pp. 1063–1075.
[60] M. Chadha, A. Jindal, and M. Gerndt, "Architecture-specific performance optimization of compute-intensive FaaS functions," in *Proc. IEEE 14th Int. Conf. Cloud Comput. (CLOUD)*, Sep. 2021, pp. 478–483.
[61] M. Shilkov. *Cold Starts in AWS Lambda*. Accessed: May 1, 2021. [Online]. Available: https://mikhail.io/serverless/coldstarts/aws/
[62] E. Oakes, L. Yang, K. Houck, T. Harter, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Pipsqueak: Lean lambdas with large libraries," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. Workshops (ICDCSW)*, Jun. 2017, pp. 395–400.
[63] J. Shen, T. Yang, Y. Su, Y. Zhou, and M. R. Lyu, "Defuse: A dependency-guided function scheduler to mitigate cold starts on FaaS platforms," in *Proc. IEEE 41st Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jul. 2021, pp. 194–204.
[64] M. Pawlik, K. Figiela, and M. Malawski, "Performance evaluation of parallel cloud functions," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–2.
[65] W. Lloyd, S. Ramesh, S. Chinthalapati, L. Ly, and S. Pallickara, "Serverless computing: An investigation of factors influencing microservice performance," in *Proc. IEEE Int. Conf. Cloud Eng. (ICE)*, Apr. 2018, pp. 159–169.
[66] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proc. ACM/SPEC Int. Conf. Perform. Eng.*, Apr. 2019, pp. 25–32.

MARKUS STEINBACH was born in Künzelsau, Baden-Württemberg, Germany, in 1992. He received the B.S. degree in business informatics from the University of Jena, Germany, in 2018, and the M.S. degree in data engineering and analytics from the Technical University of Munich, Germany, in 2021.

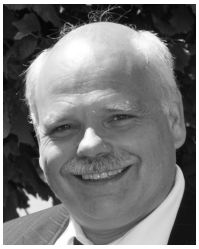
From 2020 to 2021, he worked as a Data Engineer at Allianz Deutschland AG. His research interests include temporal point processes and serverless computing.

ANSHUL JINDAL was born in India, in 1992. He received the B.Tech. degree in computer science and engineering from the National Institute of Technology, Hamirpur, India, in 2014, and the M.Sc. degree in computer science from the Technical University of Munich, Germany, in 2018, where he is currently pursuing the Ph.D. degree with the Chair of Computer Architecture and Parallel Systems, Department of Computer Science.

From 2014 to 2016, he worked at Samsung Semiconductors, Bengaluru, India, as a Senior Software Engineer. There, he worked on the development of firmware for NVMe-based PCIe SSDs. His research interests include cloud computing, specifically focusing on serverless computing for heterogeneous systems, edge computing, and AIOps.

MOHAK CHADHA received the B.E. degree (Hons.) in computer science and the M.Sc. degree (Hons.) in mathematics from the Birla Institute of Technology and Science, Pilani, India, in 2017, and the M.Sc. degree in computer science from the Technical University of Munich, Germany, in 2020, where he is currently pursuing the Ph.D. degree with the Chair of Computer Architecture and Parallel Systems, Department of Computer Science.

During his studies, he interned at CEERI Pilani, Centre for Information System and High Performance Computing at TU Dresden, Nvidia, and Airbus. His research interests include cloud computing, specifically focusing on serverless computing, high-performance computing, and federated learning.



MICHAEL GERNDT received the Ph.D. degree in computer science from the University of Bonn, Germany, in 1989, and the Habilitation degree from the Technical University of Munich, in1998.

He developed SUPERB the first automatic parallelizer for distributed memory parallel machines. For two years, in 1990 and 1991, he held a postdoctoral position at the University of Vienna and joined research center Juelich, in 1992, where he concentrated on program languages and implementation issues of shared virtual memory systems. Since 2000, he has been a Professor for architecture of parallel and distributed systems at the Technical University of Munich. He is leading the development of the periscope tuning framework for automatic performance analysis and tuning of HPC application. Within the Transregional Collaborative Research Centre InvasIC (TR 89) funded by the German Science Foundation, he investigates programming models for elastic HPC applications. He is also researching resource management for cloud and edge computing in joint projects with industry. In collaboration with Huawei in Munich, he is investigating smart cloud operations, especially with the focus on automatic anomaly detection. His current research interests include programming models and tools for scalable parallel architectures.

Dr. Gerndt is a member of the Advisory Board of Euro-Par and a member of the steering committees of several international workshops.

SHAJULIN BENEDICT (Senior Member, IEEE) graduated (Hons.) from Manonmaniam Sundaranar University, India, in 2001. He received the M.E. degree in digital communication and computer networking from A.K.C.E, Anna University, Chennai, in 2004, and the Ph.D. degree in the area of grid scheduling from Anna University.

After his Ph.D. degree, he joined a research team in Germany to pursue postdoctoral under the guidance of Prof. Gerndt. He worked as a Professor at SXCCE Research Center, Anna University. He currently works at the Indian Institute of Information Technology, Kottayam, Kerala, India. He also serves as the Director/PI/Representative Officer for AIC-IIIT Kottayam (Sec.8 Company) for nourishing young entrepreneurs of India. His research interests include HPC/cloud/grid scheduling, performance analysis of parallel applications (including exa-scale), cloud computing, the IIoT, blockchain, and parallel compilers. He was the University Second Rank Holder for his master's degree.