Modelling Serverless Function Behaviours

Rafael Tolosana-Calasanz[1], Gabriel G. Castañé[2], José Á. Bañares[1], and Omer Rana[3]

[1] Departamento de Informática e Ing. de Sistemas
Universidad de Zaragoza, Spain
[2] Insight Centre for Data Analytics
University Collegue Cork, Ireland
[3] School of Computer Science and Informatics
Cardiff University, UK

Abstract. The serverless computing model extends potential deployment options for cloud applications, by allowing users to focus on building and deploying their code without needing to configure or manage the underlying computational resources. Cost and latency constraints in stream processing user applications often push computations closer to the sources of data, leading to challenges for dynamically distributing stream operators across the edge/ fog/ cloud heterogeneous nodes and the routing of data flows. Various approaches to support operator placement across edge and cloud resources and data routing are beginning to be addressed through the serverless model. Understanding how stream processing operators can be mapped into serverless functions also offers cost incentives for users – as charging is now on a subsecond basis (rather than hourly). A dynamic Petri net model of serverless functions is proposed in this work, which takes account of the computational requirements of functions, the resources on which these functions are hosted, and key parameters that impact the behaviour of serverless functions – such as warm/cold start up times. The model can be used by developers/ users of serverless functions to understand how deployment optimisation can be used to reduce application time, and to analyse various scenarios on choosing function granularity, data size and cost.

Keywords: Petri nets, Serverless economics, Dynamic models

1 Introduction & Motivation

Cloud computing has seen a transition over recent years, from virtual machines to containers to functions. This transition has mainly been driven by reducing the overhead of deploying user-based applications within a data centre. Increasing demand for short running workloads has also driven this trend towards reducing startup (referred to as *cold start time*) and deployment time. If the startup time is significantly higher than execution time of a user application, understanding how deployment optimisation can be used to reduce application execution time remains an important challenge. Variation in demand (due to dynamically

changing input data streams) requires application resource scaling (up/ down), forcing cloud providers to respond to this within a time-bounded manner.

Serverless computing generally refers to a cloud computing model that *hides* the concept of a server, as a serverless computing platform allows users / developers to build and deploy their code without dealing with computational resources (i.e. resource management activities). The unit of deployment is the code, which is wrapped in several functions, subsequently invoked as a composition of functions that form an application. Serverless computing provides a useful basis for reacting to dynamically changing workloads in a cost-effective manner for a user. User requests for computational resources within sub-second intervals provides a more flexible way to access cloud resources, and enables better budgeting for users. This also provides a useful business model for cloud providers to make more effective use of resources that are not used for long running workloads. Using serverless based resource allocation, cloud providers are also able to utilise their spare (under utilised) capacity in a more effective way.

Serverless computing and fog computing also benefit from increasing capability offered in user devices and sensor/actuators [5]. Cost and latency constraints prevent cloud-only processing, pushing computation closer to the sources of data, and introducing important challenges for dynamically distributing operators across heterogeneous edge/fog/cloud nodes [6], and routing of data flows to the optimum computation node [7].

Understanding how stream processing operations can be mapped into (usually short running) functions at the edge/cloud layer, and the cost incentives for users/ resource providers remains a significant challenge for serverless computing. This paper proposes a dynamic Petri net model of serverless functions, which considers the computational requirements of functions, resources on which these functions are hosted, and key parameters that impact their behaviour – such as warm/ cold start up times. The model can be exploited by developers/ users of serverless functions to understand how deployment optimisation can be used to reduce application execution time, and to explore what-if scenarios for choosing appropriate function granularity, data size and cost.

This paper is structured as follows: section 2 introduces estimated costs of using functions across different vendors. Section 3 presents the key contribution of this work – focusing on developing Petri net models of serverless functions, taking account of the costing approach adopted by various existing cloud vendors. Section 4 includes a description of how these models can be used, with evaluation in section 5, followed by concluding remarks in section 6.

2 Serverless Function Economics

A number of vendors offer serverless functionality – ranging from Amazon AWS, Google, Microsoft – to a number of additional vendors & open source systems such as IBM Cloud Functions, Knative based on Kubernetes deployment, Apache OpenWhisk-based function deployment, Cloudflare workers, Oracle functions, etc. A single mechanism to compare costs across different serverless offerings

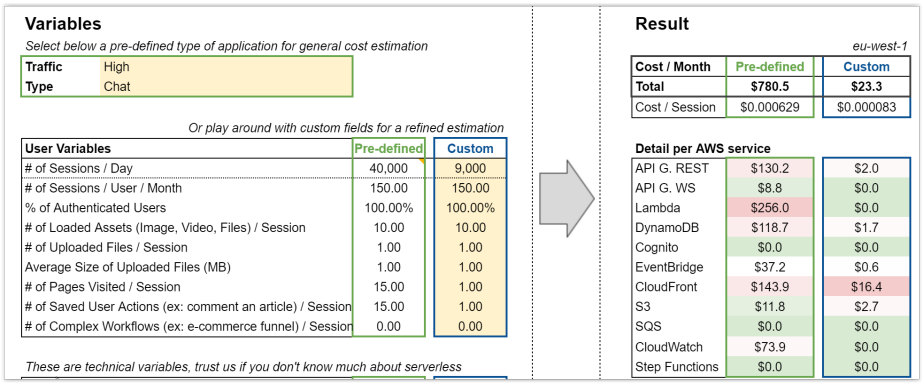
is very challenging, as the type of infrastructure (CPU type, execution speed), memory supported (e.g. 128MB to 8GB), data transfer rates supported etc, differ widely across vendors. Azure and Lambda functions are generally integrated with other services, making it challenging to do a feature to feature comparison across vendors. AWS offers the widest choice, offering serverless functions with differing resource characteristics (different RAM and underlying processor architectures). Figure 1 provides a costing undertaken on AWS Lambda using a number of different variables (e.g. user authentication, number of pages processed), and aligned with other AWS services (e.g. cloud monitoring and CDN (CloudFront)). Google function allocation is based on the size of memory and processor CPU speed, with compute time measured from the time a request is received to the time that the function is signal to be completed (successful termination, failure or a timeout). Compute time is measured in 100ms increments, rounded up to the nearest increment (e.g. a 170ms execution is billed as 200ms). For Microsoft Azure functions, billing is based on a per second resource consumption basis (considering a vCPU) and number of executions carried out within a time window. Consumption plan pricing includes a monthly free grant of 1 million requests and 400,000 GBs of resource consumption per month per subscription in pay-as-you-go pricing across all function apps in that subscription.

Vendor	Billable Unit (US$)	Key considerations
Amazon Lamda (128MB)	$0.0000000021 (1ms)	Pricing based on requests and duration
Amazon Lamda (1024MB)	$0.0000000167 (1ms)	
Amazon Lamda (10240MB)	$0.0000001667 (1ms)	
Google functions (128MB, 200MHz CPU)	$0.000000231 (100ms)	Pricing based on: compute time, use of network capacity, number of invocations
Google functions (8192MB, 4.8GHz CPU)	$0.000006800 (100ms)	
MS-Azure functions	$0.000016/GB-s	Number of invocations

Table 1: Serverless Costs – based on [2–4]

Replacing existing container/VM-based provision with a function-based offering (e.g. AWS Lambda) can lead to significant long term savings for a typical hosting environment. For instance, consider that it takes 2s to serve a page view based on the data from DynamoDB, we can calculate the total cost of serving 100K page requests. Even with a generous 1GB memory allocation and relatively sluggish 2s processing time, the total cost for AWS Lambda would be less than US$5 (calculated based on 1024MB AWS Lamda costs from table 1). A

key challenge in function-based deployments is the *keep-alive* time of these functions between invocations. A cloud service provider may want to use synthetic data to minimise the cold start time associated with starting up a function – an important variable that influences both operational and energy costs for the provider.



Variables

Select below a pre-defined type of application for general cost estimation

Traffic	High
Type	Chat

Or play around with custom fields for a refined estimation

User Variables	Pre-defined	Custom
# of Sessions / Day	40,000	9,000
# of Sessions / User / Month	150.00	150.00
% of Authenticated Users	100.00%	100.00%
# of Loaded Assets (Image, Video, Files) / Session	10.00	10.00
# of Uploaded Files / Session	1.00	1.00
Average Size of Uploaded Files (MB)	1.00	1.00
# of Pages Visited / Session	15.00	1.00
# of Saved User Actions (ex: comment an article) / Session	15.00	1.00
# of Complex Workflows (ex: e-commerce funnel) / Session	0.00	0.00

These are technical variables, trust us if you don't know much about serverless

Result

eu-west-1

Cost / Month	Pre-defined	Custom
Total	**$780.5**	**$23.3**
Cost / Session	$0.000629	$0.000083

Detail per AWS service

API G. REST	$130.2	$2.0
API G. WS	$8.8	$0.0
Lambda	$256.0	$0.0
DynamoDB	$118.7	$1.7
Cognito	$0.0	$0.0
EventBridge	$37.2	$0.6
CloudFront	$143.9	$16.4
S3	$11.8	$2.7
SQS	$0.0	$0.0
CloudWatch	$73.9	$0.0
Step Functions	$0.0	$0.0

Fig. 1: Serverless Costs from Amazon AWS [1]

3 Serverless Models

We develop a serverless function model that can be hosted across different types of resources – from data centre to edge nodes. The model can be used as a basis to support capacity planning for serverless functions, enabling an application designer to investigate their application requirements using the model. Petri nets are a well-established formalism and have been used extensively to model concurrent and distributed systems. Reference nets are a specific type of Petri nets that support greater levels of dynamism than ordinary Petri nets and support Java code inscriptions. In this work, we make use of Reference nets and their interpreter *Renew* to create dynamic serverless function models that can be configured over a real system's deployment. A quick introduction to the ordinary Petri net theory can be found in [12]. An example of how Reference nets can be applied to the modelling of applications and their mapping to cloud resources can be found in [13].

Figure 2 depicts Petri net (shorted to *net* in the description below) patterns to model a physical, hierarchical edge / cloud infrastructure. On the left, the net represents a node that contains computational resources to execute functions. The net in the center models data movement activity that connects two nodes, or a data source (sink) and a node. The two nets on the right represent a data source and a data sink. All these patterns can be combined to generate a model of a hierarchical (layered) physical edge infrastructure, where the lowest layer

comprises IoT sensors and other user devices, and the topmost layer will represent the cloud data center. All intermediate nodes between the data sources and the cloud data center represent different fog nodes, aligning with the systems architecture proposed by the Open Fog Consortium[4].

In our model, data is always transmitted across the infrastructure along with a processing plan (or user application). The plan specifies a composition of functions in the form of a Directed Acyclic Graph (DAG) that need to be applied to its associated data. Furthermore, both the data and its plan are modelled as tokens in the physical edge model. In our model, the plan specifies the orchestration of the execution inside a node and across the nodes.

When data chunk and its associated processing plan arrive at a node, then Transition $t1$ from Figure 2 is fired (triggering the invocation of the synchronous channel *begin* of the node). Once all function invocations in the graph are accomplished, Transition $t2$ is fired and the processing plan (Variables *app* and the data chunk d are obtained from that transition). Transitions $i11$, $i12$ and $i13$ of the node are involved in its initialization. Transition $i11$ creates an instance of the underlying serverless node components ($faasnode$ net). The computational resources of the node are initialized in Transition $i12$. The model could be parameterized from a configuration file, but to enhance the readability purpose, we made the textual configuration visible within the model: We can see that 7 Raspberry Pi 2 devices with 1000MB and 1000MIPS are available for the node. Transition $i13$ initializes the functions that a particular user wants to place at that node. In this example, user id 1 places two functions $f1$ and $f2$ with the following parameters (from left to right): the first 4 numbers in the tuple represent the function execution time, the function warm invocation time, the function cold start time, and the time that the function will be idle in memory. The next number is the cost per millisecond of invoking the function (aligning with costs identified in Table 1), the last two numbers represent the computational requirements, expressed in MIPS and the size of memory required, respectively.

The three main internal components of a node are depicted in Figure 3: user application (the composition of functions to be applied to a data chunk), the user functions available at the node that are managed by the function manager component (Variable fm in the model), and the machine on which the functions will be executed. As stated previously, when the pair: data chunk and its processing plan arrive at a node, Transition $t1$ of Figure 2 is fired. During the firing, by means of the synchronous channel *begin*, Transition $t1$ of Figure 2 synchronizes with Transition $t21$ of Figure 3, and the token data chunk-plan is moved inside the node for processing. Once all the required functions are invoked, Transition $t22$ of Figure 3 will be synchronized with Transition $t2$ of Figure 2, taking the data chunk and its processing plan out of the node. The invocations of functions are accomplished by means of Transitions $t23$, which represents the start of an invocation, and $t24$ which represents the end of an invocation in Figure 3. In these two transitions, using synchronous channels, a composition plan (*app* in the model) is paired with the function manager component (fm in the model).

[4] `https://opcfoundation.org/about/opc-technologies/opc-ua/`

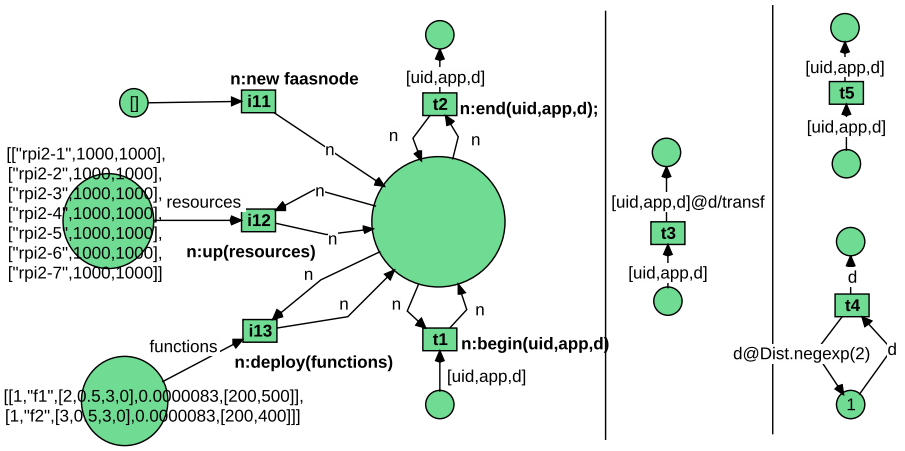


Fig. 2: Node Infrastructure Net Patterns: (left) FaaS-based edge / cloud node, (center) data transfer representation, and (right) data source modelling

As these two transitions can be fired concurrently, a processing plan can invoke functions concurrently in the model.

A function needs to meet some conditions to be invoked: (i) it needs to find a computational resource with enough memory and CPU capacity, (ii) the function needs to be loaded in memory. Transition $t26$ binds a computational resource to a function, a computational resource that matches the function memory and CPU requirements. It should be noticed that these constrains are enforced by the inscription: $guard mips >= mipsrq \& mem >= memrq$. This inscription will only enable Transition $t26$ when there is a machine whose MIPS and memory are enough to host the function. Transition $t27$ frees the computational resources, it means that the function was removed from memory and placed back on disk.

From Figure 3, transition $t27$ enables users to deploy functions, transitions $t28$ and $t29$ enable allocation and deallocation of computational resources respectively. Computational resources are represented by a tuple comprising: (i) resource identifier, (ii) resource CPU performance (in MIPS) and (iii) the memory size.

The dynamic behaviour is achieved through the function manager component, which is inside the node (Figure 3). While multiple data chunks and their processing plans can exist simultaneously inside the node model in Figure 3, there is only one instance of the function manager component. This component controls the life cycle of functions and manages their invocations. It consists of two concurrent processes, the function as a service life cycle process (specified in Figure 4) and the function invocation process (in Figure 5). A function is deployed in the model at Transition $t41$ in Figure 4. This transition synchronizes with Transition $t25$ in Figure 3 and with Transition $i13$ in Figure 2 simultaneously, by chaining different synchronous channels that enable the functions

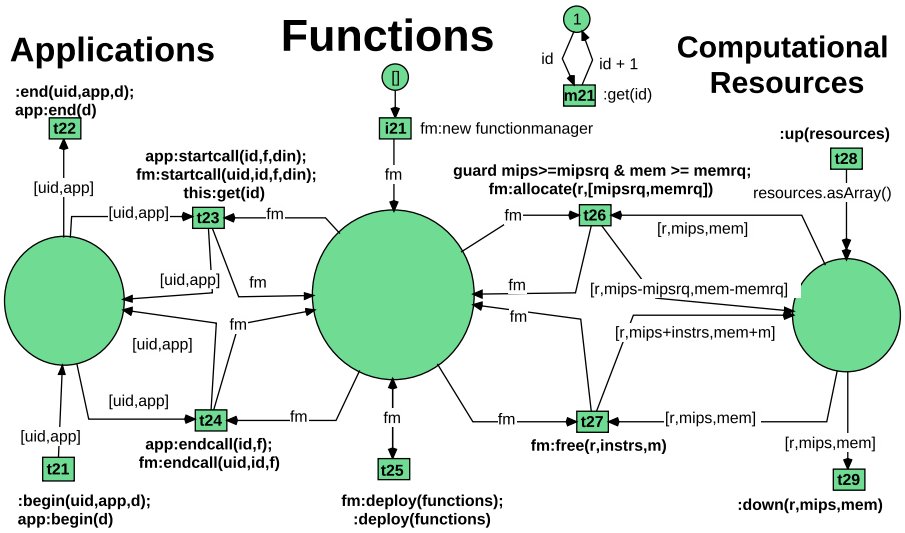


Fig. 3: Node Modelling: a node consists of applications, functions and computational resources

to arrive from the model in Figure 2 to Figure 4. The deployed functions will eventually arrive at the place "Compiled & Idle in Disk" in Figure 4, waiting for an invocation. Once an invocation occurs, a function instance is loaded in memory (Transition $t43$ fires), this is called a cold start invocation. At that point, a function in memory is ready to be called, the actual invocation happens when Transition $t45$ is fired, and the invocation finishes when Transition $t46$ is fired. In the model, after the call, the function remains in memory idle for some period of time (function parameter $t3$), ready to be invoked again. In such a case, the invocation is called warm function invocation, and it involves firing Transition $t44$. If the period of time elapses without an invocation Transition $t47$ will be fired.

Cold and warm function invocations can have significant impact on function performance, and cold invocations typically have a higher time than warm invocations. This is reflected in the model by the time inscriptions on the arcs – the output arc of Transition $t43$ (cold invocation) has the time inscription $[uid, f, [tex, t1, t2, t3], ecost, hwrq]@t2$, while the output arc of Transition $t44$ (warm invocation) has $[uid, f, [tex, t1, t2, t3], ecost, hwrq]@t1$. In both cases, a token will be available after $t2$ and $t1$ units of time after the firing. The model allows cold and warm times for each function to be parameterized. Similarly, the actual function execution is modelled by the output arc of Transition $t45$, $[uid, f, [tex, t1, t2, t3], ecost, hwrq]@tex$, which indicates that after the invocation, the token will be available after tex units of time. While all these time inscriptions are on output arcs, the model also uses time inscriptions at input

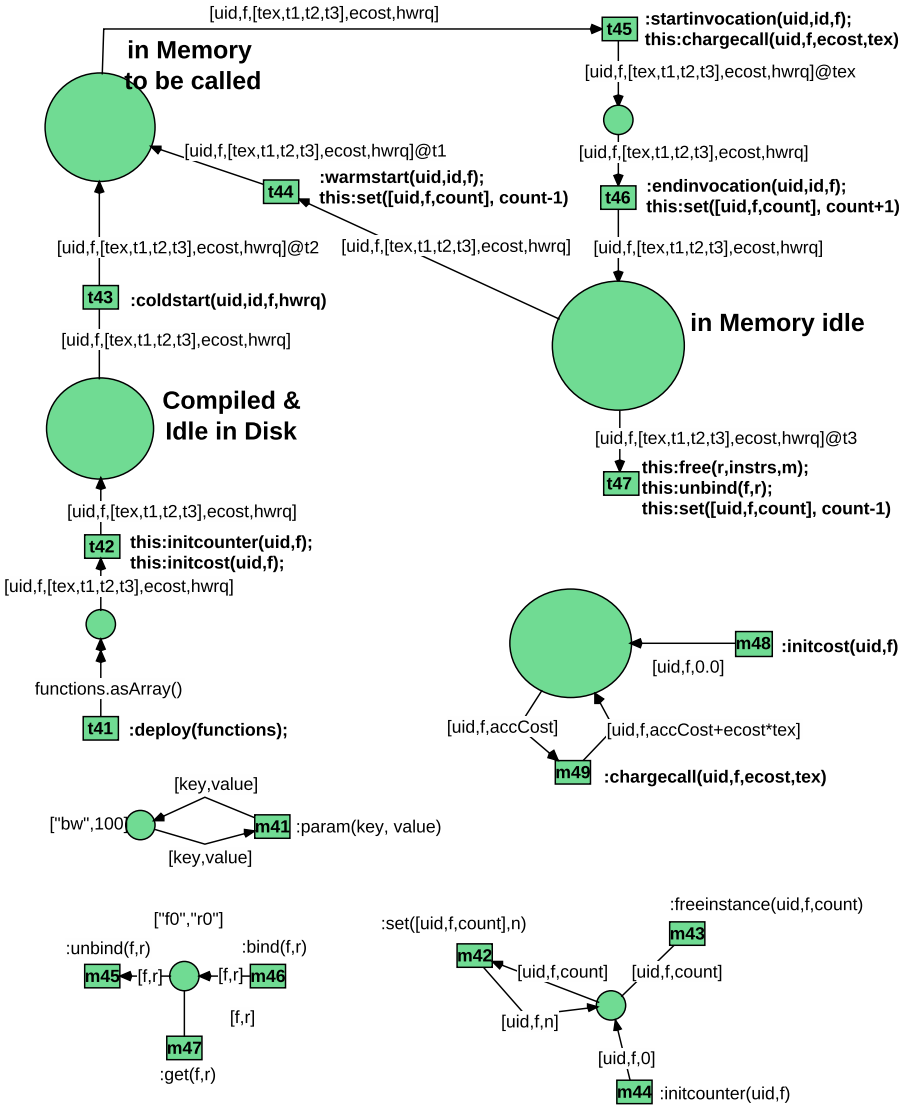


Fig. 4: Function as a Service life cycle

arcs. Once a function is idle in memory, it will remain for a period of $t3$ units of time. If no invocation occurs, the function will be removed from memory and the computational resources freed. This is modelled by the input arc of Transition $t47$: $[uid, f, [tex, t1, t2, t3], ecost, hwrq]@t3$. The effect of the time inscription at the input arc is that once a function instance is idle in memory, Transition $t47$ will be only enabled after $t3$ units of time.

Another important aspect of the model in Figure 4 is the consideration of the economic cost. Serverless infrastructures typically charge users on a per millisecond basis. This is reflected in the model in Transition $t45$ that, once fired, invokes Synchronous Channel $this : chargecall(uid, f, ecost, tex)$. The callee channel is in Transition $m49$, which retrieves the accumulated cost for function f of user uid ($[uid, f, accCost]$) and adds the incurred cost for the actual invocation $[uid, f, accCost + ecost * tex]$, where $ecost$ is the cost associated with function f, and it is a parameter of the function in the model. This pricing model is based on computing time on per millisecond basis, which one of the models described on Section 2. Other models can be implemented by updating that cost formula.

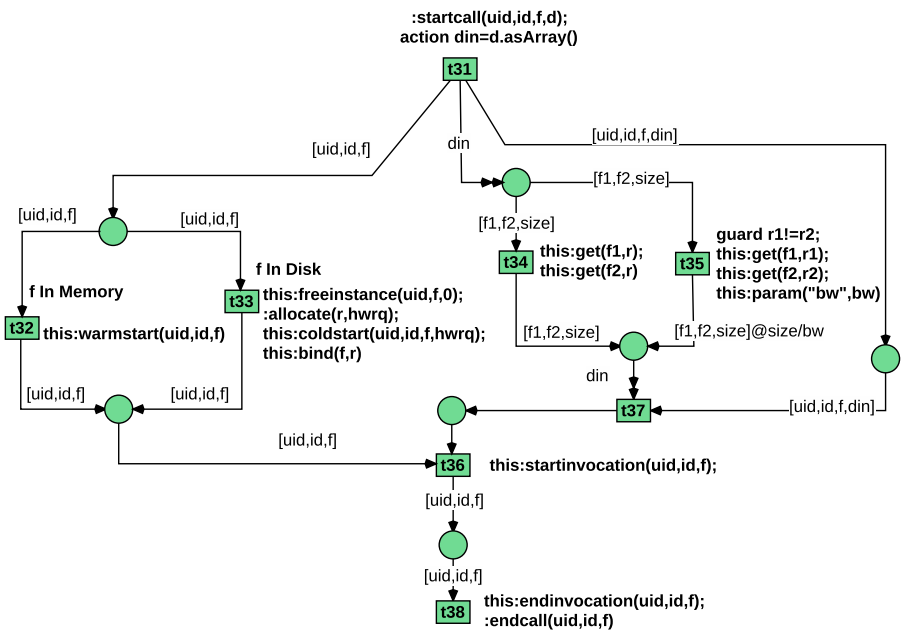


Fig. 5: Function Invocation and Data Movement

When a function invocation occurs in the model of Figure 3, Transition $t23$ is fired. It involves invoking the synchronous channel *startcall* of the function composition, and the synchronous channel "startcall" of the function manager component. In Figure 5, it corresponds to Transition $t31$, where the invoca-

tion process starts. At this point, the model considers two *concurrent* activities: the function invocation, and the transmission of the arguments of the function through the local area network links of the node. It is important to highlight that the time elapsed in both activities will overlap.

The function invocation involves Transitions $t32$ and $t33$. Transition $t32$ will only be enabled if there is any function instance f idle in memory in the net of Figure 4 (warm invocation). Otherwise, Transition $t33$ will be enabled (cold invocation). In case of a cold invocation, computational resources need to be allocated. Transition $t33$ will allocate the required computational resources by means of synchronous channel : $allocate(r, hwrq)$. It will synchronize with Transition $t26$ of Figure 3, which was described previously.

The data movement activity may only have an impact on the function performance time if the transmission time is significant, considering the argument size and the LAN bandwidth. There is no need to move data if the functions are hosted in the same machine. From the user function composition, the model obtains the data dependencies for function f. All these dependencies are placed at the input place of Transitions $t34$ / $t35$. If the origin function ($f1$) and the destination function ($f2$) are hosted on the same machine (r), then there is no data movement required, and Transition $t34$ will be fired. In contrast, if they are in different machines, Transition $t35$ will be fired. This transition has an output arc time inscription $[f1, f2, size]@size/bw$ modelling that the data argument will require $size/bw$ units of time to arrive from $f1$ to $f2$, where $size$ is the data argument size and bw is the LAN effective bandwidth. Once all the data arguments are available in f, the actual invocation can start, and Transition $t38$ will be enabled. This transition synchronizes with Transition $t48$ of the net model of Figure 4, which was described previously.

Therefore, when tokens move across all these net paths, time and economic cost derived from processing accumulates, allowing the model to obtain end-to-end latency for a user data chunk, and the processing bill. The economic cost derived from data transmissions inter layers of the edge can also be easily computed by counting the number of messages.

4 Related Work, Model Usage & Characterisation

Simulators are a widely used tool to explore what-if scenarios and costs - energy, resource provider or user - in computer systems, however the core models that the simulation engines allow to use to extract results is radically different. For instance, different network simulator libraries and frameworks include NS-3 [**?**], OMNET++ [**?**], and OPNET [**?**]. These tools focus on different network aspects but lacking on details to simulate fine gran applications and workflows between these on the application layer.

Similarly, and focused on computer-system architecture research are Simics [**?**], Gem5 [**?**], and Graphite [**?**] where the focus is given to multiple architectures for CPU and GPU processors architecture and memory systems.

Different cloud and edge simulation frameworks also exist and there are two strands. The network-based components layout focused on simulating the user applications but supported by detail models of hardware and software interactions in order to also estimate multiple parameters on energy, and resource utilisation of the nodes. Foremost among these are: CloudSim [**?**], and iFogSim [**?**] - as plugin of the former, iCanCloud [**?**], FogNetSim++ [**?**]. These simulators, in spite to provide a high accuracy at every level, are build on top of network simulators and it is not possible to deattach the hardware part easily from the application level, therefore adding a additional overhead - from computation and memory requirements - that in case of the serverless applications simulations is unnecessary to the final user.

A second strand of cloud and edge simulators focuses on layered architectures representing data center components. These model primarily the interactions between components and abstracting hardware and/or network resources. Some relevant simulators on this category are: Simulizar [**?**], model driven simulator designed for self-adaptative systems to explore transient phases on load balancing of workloads; and focused on the application layer, but the impact of bags of tasks on heterogeneous large scale data centres, Cloudlightning Simulator provide high scalable simulations simplifying the hardware and application models as authors explain in their paper [**?**].

Modelling and simulation to support capacity planning for serverless and fog systems provides significant benefit, to the best of our knowledge, this is the first study that proposes a model for the serverless computing paradigm. The work in [8] provides a survey of modelling and simulation tools for Fog systems, covering mathematical models, including Petri nets and Markov Chains, and and various cost parameters that need to be considered. The survey concludes that only a few simulation tools take account of cost metrics, and that this aspect is still in its infancy.

Performance and cost modeling of cloud computing has been extensively covered, but only recently the modelling of Serveless Function has received attention. In [9], limited user control over resources on FaaS platforms on the cloud is emphasised, and a formal model of serverless workflows to estimate performance and cost is proposed. However, Fog computing nodes have limited resources, which can introduce an added complexity in the modelling of serverless function behaviors that now must consider the heterogeneity of cloud and edge/fog nodes – with varying resource capacity. The need to represent the dependency of serverless applications on data storage and other resources on the cloud is identified in [10]. In this paper, authors present a dependency graph for serverless applications that helps to optimize an existing system by identifying hot spots, supports the generation of test cases and can be used to monitor an existing system. The problem of scheduling operators between the Cloud and the Fog is also the focus of several research efforts – these consider both computational and network resource usage costs [6, 11], and propose analytical models and operator placement strategies to reduce end-to-end latency, data transfer times and messaging costs between edge and cloud systems.

The Reference/ Petri net models presented in Section 3 enables us to support capacity and cost planning for deployment of serverless functions. By varying costs and times associated with execution of functions, and the types of resources on which these functions are hosted, it is possible for a user to plan their application design and deployment. The models we propose go beyond existing cost calculators provided by cloud providers, as we are able to derive a finer grained analysis taking account of actual deployment and use – achieved by combining the modelling and simulation capability made possible by the use of a Petri net model.

5 Evaluation

In addition to the formal semantics provided by Petri nets, another advantage of using Reference nets is that they can be interpreted by the Renew tool[5]. In order to show how the model can be exploited, we provide an example of usage here in this section. From two synthetic applications, we conducted different simulations to analyse the impact of cold and warm function invocations on its performance and the amount of computational resources required. We made use of simple Reference nets, and the time inscriptions where simulated with action delays[6]. For the physical infrastructure, we modeled two nodes: an edge node with up to 5 Raspberry Pi 2 devices, connected to a Cloud data center, with multiple Intel Xeon servers. The edge node is connected to two data sources that generate data continuously at constant rates (every 5s to 6.75s), having each data chunk a size of 1 MB and remaining constant through the simulation.

We designed two synthetic streaming applications: f and g. Application f is a sequential composition of 5 functions (f_1 to f_3, at the edge and f_4 to f_5 at the cloud) and it consumes data generated from a data source. The other source of data is processed by application g, which consists of 3 functions (g_1, g_2 at the edge and g_3 at the cloud). Table 3 summarizes the characterization and requirements of the functions: the average execution time, the average memory size requirements and the average amount of instructions that require its execution. We simulated 4 different scenarios with different combinations of warm / cold function invocation times, and keep alive periods, as well as the number of computational resources allocated to each application, as specified in Table 2.

Figure 6 depicts a graphical representation of the main idea of our simulations, a chronograph corresponding to the edge of two possible scenarios of applications f ($f_1 \rightarrow f_2 \rightarrow f_3$) and g ($g_1 \rightarrow g_2$): (a) on top, without keep alive periods of time for functions, whenever a function finishes, the involved computational resources are released. Therefore, as the two applications are sequential compositions, each application only requires one computational resource at a time. In contrast, the case (b) reflects that when introducing keep alive periods

[5] `http://renew.de/`

[6] The models in Reference nets and the simulation environment are made available through a Docker container with the aim of enhancing the reproducibility of experiments: `https://github.com/rtolosana/fog-modelling`

of functions, as functions are kept in memory, the number of computational resources required increases significantly (five at this early stage of execution of applications f and g). Choosing between one or another option, or intermediate alternatives will depend on the actual cold / warm function invocation times, on the QoS to be enforced and on the computational capacity available, which might be scarce at the edge.

Case	Warm Invocation	Cold Invocation	Keep Alive
1	0.01	10	10
2	0.01	0.05	10
3	0.01	10	0
4	0.01	0.05	0

Table 2: Simulation Parameters (in secs)

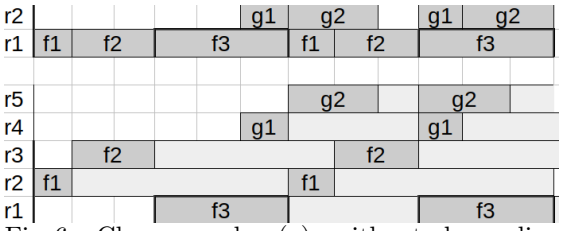


Fig. 6: Chronograph: (a) without keep-alive (top) (b) with keep alive (down)

The results of the simulations can be seen in Figure 7. Although the simulations involve the execution of compositions f and g, for clarity purposes and space, Figure 7 only depicts the performance of composition f. On the x-axis, the timeline, and on the y-axis, the end-to-end latency both in seconds. The end-to-end latency includes processing times, waiting times, overheads, and data transmissions. When the cold function invocation time is higher than the execution time (cases 1 and 3), it has a significant impact on performance time, unless the function is kept alive in memory (case 3). Therefore, in case 3, the impact of high cold invocations on performance only appears the first time the functions are invoked. However, this is at the expense of consuming more computational resources. As our models do not consider invocation overheads for the economic cost, and they only include the actual invocation time, the 4 scenarios show the same cost. For composition f for all the simulation time, the economic cost is 0.24 USD, as each function pricing tariff costs 0.0000083 USD per msec.

6 Conclusions

We develop a dynamic Petri net model of a serverless function, demonstrating how a combination of these functions can be hosted across edge and cloud data centre-based resources. With the significant flexibility that a serverless model offers to users, in both costs of use and deployment, many see a transition to serverless as a natural progression from VM and container based invocations. The model proposed in this work includes a number of parameters that can be characterised from a practical deployment, and can be used for designing an application across different types of resources. Our proposed approach can be used

F	Tex(secs)	Mem(MB)	MIPS
f1	0.2	600	600
f2	0.4	600	700
f3	0.6	600	800
f4	0.04	600	600
f5	0.04	600	600
g1	0.2	200	700
g2	0.4	300	500
g3	0.05	600	600

Table 3: Function Characterization

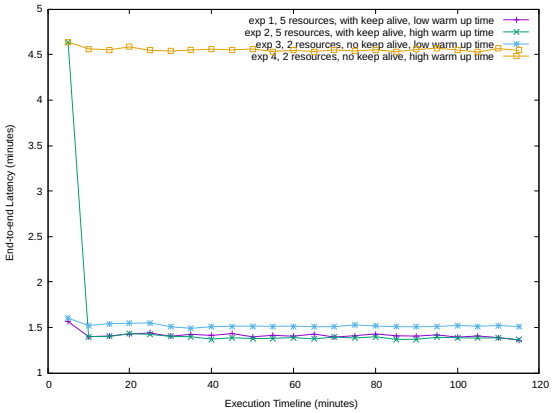


Fig. 7: End-to-End Latency for f over time

to undertake a number of *what-if* scenarios to explore various configuration options available to a developer, ranging from: computational complexity of hosting nodes (characterised as MIPS and memory), cold and warm start times associated with initiating a function, data size associated function execution, time to move function executable to/from disk and computational requirements (also modelled as MIPS and memory) of the function itself. This approach can also be used to undertake comparison of executing the same function across different cloud vendors – who may offer different pricing/power tradeoffs for function developers.

References

1. X. Lefevre, "Is serverless cheaper for your use case? Find out with this calculator" – Available at: `https://medium.com/serverless-transformation/is-serverless-cheaper-for-your-use-case-find-out-with-this-calculator-2f8a52fc6a68`. Last accessed: June 2021.
2. Microsoft, "Azure Function Pricing". Available at: `https://azure.microsoft.com/en-gb/pricing/details/functions/`. Last accessed: June 2021.
3. Amazon, "AWS Lambda Pricing". Available at: `https://aws.amazon.com/lambda/pricing/`. Last accessed: June 2021.
4. Google, "Google Function Pricing". Available at: `https://cloud.google.com/functions/pricing`. Last accessed: June 2021.
5. L. M. V. González and L. Rodero-Merino, "Finding your way in the fog: Towards a comprehensive definition of fog computing," *Comput. Commun. Rev.*, vol. 44, no. 5, pp. 27–32, 2014.
6. E. Gibert Renart, A. Da Silva Veith, D. Balouek-Thomert, M. D. De Assuncao, L. Lefevre, and M. Parashar, "Distributed operator placement for iot data analytics across edge and cloud resources," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 459–468.

7. M. A. L. Peña and I. M. Fernández, "Sat-iot: An architectural model for a high-performance fog/edge/cloud iot platform," in *5th IEEE World Forum on Internet of Things, WF-IoT 2019, Limerick, Ireland, April 15-18, 2019*. IEEE, 2019, pp. 633–638.
8. S. V. Margariti, V. V. Dimakopoulos, and G. Tsoumanis, "Modeling and simulation tools for fog computing a comprehensive survey from a cost perspective," *Future Internet*, vol. 12, no. 5, 2020.
9. C. Lin and H. Khazaei, "Modeling and optimization of performance and cost of serverless applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 3, pp. 615–632, 2021.
10. S. Winzinger and G. Wirtz, "Model-based analysis of serverless applications," in *2019 IEEE/ACM 11th International Workshop on Modelling in Software Engineering (MiSE)*, 2019, pp. 82–88.
11. P. Ntumba, N. Georgantas, and V. Christophides, "Scheduling continuous operators for iot edge analytics," in *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking*. Association for Computing Machinery, 2021, pp. 55-60.
12. T. Murata, "Petri nets: Properties, analysis and applications," in Proceedings of the IEEE, 77(4), 1989, pp.541-580.
13. R. Tolosana-Calasanz, J.Á. Bañares, C. Pham, and O.F. Rana, "Enforcing qos in scientific workflow systems enacted over cloud infrastructures". *Journal of Computer and System Sciences*, vol. 78, no. 3, pp.1300–1315, 2012.