Enhancing Performance Modeling of Serverless Functions via Static Analysis

Runan Wang$^{[0000-0001-9245-6096]}$, Giuliano Casale$^{[0000-0003-4548-7951]}$, and Antonio Filieri$^{[0000-0001-9646-646X]}$

Department of Computing, Imperial College London, UK
{runan.wang19, g.casale, a.filieri}@imperial.ac.uk

Abstract. Serverless computing leverages the design of complex applications as the composition of small, individual functions to simplify development and operations. However, this flexibility complicates reasoning about the trade-off between performance and costs, requiring accurate models to support prediction and configuration decisions. Established performance model inference from execution traces is typically more expensive for serverless applications due to the significantly larger topologies and numbers of parameters resulting from the higher fragmentation into small functions. On the other hand, individual functions tend to embed simpler logic than larger services, which enables inferring some structural information by reasoning directly from their source code. In this paper, we use static control and data flow analysis to extract topological and parametric dependencies among interacting functions from their source code. To enhance the accuracy of model parameterization, we devise an instrumentation strategy to infer performance profiles driven by code analysis. We then build a compact layered queueing network (LQN) model of the serverless workflow based on the static analysis and code profiling data. We evaluated our method on serverless workflows with several common composition patterns deployed on Azure Functions, showing it can accurately predict the performance of the application under different resource provisioning strategies and workloads with a mean error under 7.3%.

Keywords: Serverless Computing · Performance Modeling · Layered Queueing Networks · Static Analysis · Code Profiling

1 Introduction

Serverless computing is a novel cloud computing paradigm that aims at making operations concerns transparent to developers and cloud users [9, 13]. It has recently gained increasing attention in industry due to the potential for significant cost savings and on-demand billing modes. Function-as-a-Service (FaaS) is a cloud computing execution model introduced within serverless computing that allows developers to deploy single functions as basic building blocks [9]. Compared to monolithic applications and microservice-based architectures, FaaS-based applications can be triggered and served by events (e.g., HTTP requests) and executed on-demand. There are several cloud vendors providing FaaS capabilities like AWS Lambda, Google Cloud Functions and Microsoft Azure Functions, as well as open-source alternatives such as OpenFaaS or KNative.

Developers can write individual serverless functions and compose them in complex workflows deployed on the FaaS platforms. FaaS platforms enable automatic management, scaling, and billing of the execution of FaaS-based workflows to take over most operational efforts from developers and users. However, maintaining Quality-of-Service (QoS) requirements and meeting service-level agreements (SLAs) of FaaS applications remains an outstanding concern [26].

Performance models provide analytical prediction and simulation results to help to reason about and improve the quality of FaaS-based applications. Accurate and efficient performance modeling benefits not only the developers and operators, but also FaaS providers. On the one hand, with performance models, the developers have a better understanding and prediction capabilities of the quality of the application under different workloads and deployment configurations, which may also help direct development decisions. On the other hand, FaaS providers can take advantage of accurate cost prediction and resource management, inferring related metrics from performance models. There are well-established stochastic models such as queueing networks [16], layered queueing networks (LQNs) [14], Petri nets [22] that can describe the system with a simplified abstraction. Among them, LQNs are particularly suitable for capturing the dependencies and interactions between different FaaS functions.

Building performance models for FaaS applications accurately and efficiently is a non-trivial problem. However, differently from monolithic or service-based applications that aggregate larger functionalities behind each endpoint, the source code of individual serverless functions is usually more focused and succinct, rendering it amenable to static code analysis to infer additional information about the internals of FaaS applications. Our insight is to exploit established control and data flow analysis methods [23] to improve the granularity of performance models for FaaS-based applications, ultimately improving the accuracy of models and performance predictions. However, building LQN models for FaaS functions and workflows is still challenging due to the information gap between modeling and monitoring granularity compared to the classical performance modeling for web applications and microservice-based applications.

The first challenge in building LQN models for FaaS applications is learning the topological graph representing the application behavior on both inter- and intra-function levels. Attempting to accurately and completely reconstruct this structure only from traces or monitoring data may be difficult because it relies on the test inputs capable of covering all the relevant execution traces. However, when functions are observed as black-boxes, i.e., without knowing which parts of their code have been exercised, there is no reliable way to ascertain whether any behavior has remained uncovered. In turn, the LQN model inferred from such partial traces may itself be incomplete.

Additionally, appropriate model parameterization is critical to define effective and efficient parameter estimation methods. Estimating service demand for individual endpoints from system monitoring measurements, like utilization or response time, is particularly challenging, with most methods typically resorting to regression algorithms to combine different measurements [25]. However, these methods estimate service demand based on queueing theory and may lead to in-accurate results due to the uncertainty introduced by the approximation based on the queueing theory.

This paper proposes to build performance models for FaaS workflows combining static analysis and code profiling. We assume that the source code and configuration metadata of FaaS functions and workflows are accessible. To learn the topology of the model, we apply static analysis on the source code to obtain the inter-procedural call graph of the orchestrator defining the workflow composing the individual functions, and the intra-procedural control flows for each function. To more accurately characterize the model parameterization, we propose to inject code to hook system function calls during the profiling stage and capture the distribution of the service demand based on profiling data instead of estimating based on system measurement. The profiling data, being measured within the process executing the function, depends only on the function inputs, while it is largely workload-independent since queueing time does not affect the measures. Then, we derive the LQN models for serverless workflows by mapping the static graphs and code profiling data. In the experiments, we implement FaaS-based workflows representing different function compositing patterns to evaluate our proposed method. We compare the results by solving LQN to the data collected from the workflow execution. The experimental results yield model predictions with a mean error under 7.3% in all the evaluated scenarios.

The remainder of the paper is structured as follows. In Section 2, we give background on static analysis and LQN. In Section 3, we discuss the methodology of building LQN model based on static analysis and code profiling. In Section 4, we conduct experiments with different FaaS workflows to evaluate the effectiveness and efficiency of our proposed modeling method. In Sections 5 and 6, respectively, we discuss related work and draw conclusions.

2 Background

Static Analysis of Source Code. Static analysis is widely used to infer information about a program by reasoning on the structure and features of its source code, or convenient intermediate representations, without actually executing the program [23] (as opposed to dynamic analyses that require executing the program). For example, it can infer which statements in a program affect the value of a variable at a specific line. Two widely used representations that can be statically extracted are control and data flow graph. A control flow graph (CFG) captures (a superset of) all the possible paths that can be executed at runtime. A CFG represents how the evaluation of conditional statements (e.g., branches and loops) determines the next code block to be executed. At intra-procedural level, CFGs represent the dependencies between code blocks and all the possible execution orderings, subject to the decisions at conditional nodes. Intra-procedural CFGs can be related to one another via the program call graph. A call graph (CG) captures for each caller function all the callee functions it can invoke, providing an inter-procedural representation of the dependencies and interactions among functions. Instead, a data flow graph represents the propagation of information throughout program statements and variables [18]. Static data flow analysis, for example, can compute execution paths that propagate

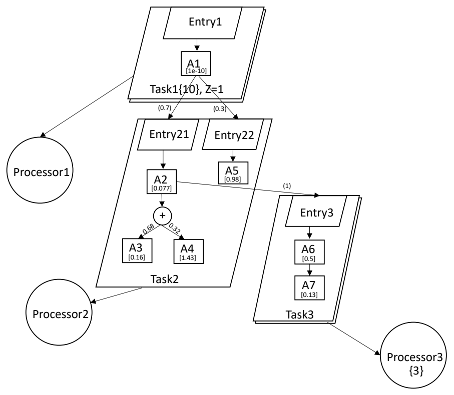


Fig. 1: An example LQN model

values of interests from their sources to sinks. Data can flow through dependent nodes of a CFG, e.g., through the arguments of a function invocation or the decision at a conditional node. Taint analysis is a data flow analysis that can track which program variables at which code locations are affected by the values of a function's inputs. While typically used for security purposes [27], taint analysis can capture what input information can flow to other function invocations.

Layered Queuing Networks. LQNs are an extended queuing network formalism that has been widely used to abstract web applications [17]. The main components of LQN models covered in this paper are shown with an example in Figure 1. The large parallelograms, denoted as *Task*, represent software and hardware entities. There are mainly two types of tasks: a task representing the clients, and tasks representing the servers processing incoming requests. Tasks are hosted on resources that are denoted as processors in the circle, and multiprocessor hosts can be specified with a multiplicity figure. Smaller parallelograms inside a task are called *Entry* and represent different service classes provided by a task (e.g., different endpoints). The detailed operations inside each entry can be described with a set of *Activity* specified with their execution order (rectangular nodes for activities and circular "+" nodes representing probabilistic choices). Each activity is parameterized with service demand, for example, specifying the mean value of the exponential demand distribution. Activities can make requests to different entries by sending synchronous or asynchronous calls. For instance, in the top task, 10 concurrent clients are sending synchronous requests to $E21$ and $E22$ and the arcs are labelled with the value of the mean number of requests.

3 Methodology

Our methodology for modeling serverless applications combining static analysis and code profiling includes three main phases: a static analysis to learn the topology for LQN structure modeling, a dynamic analysis with code profiling to collect data for model parameterization, and LQN model generation.

Overview. The overview of the proposed methodology is shown in Figure 2. Assume that after the development of individual serverless functions or FaaS-based workflows, we are able to access and instrument the source code and configuration metadata either from developers or cloud providers. (Black-box functions

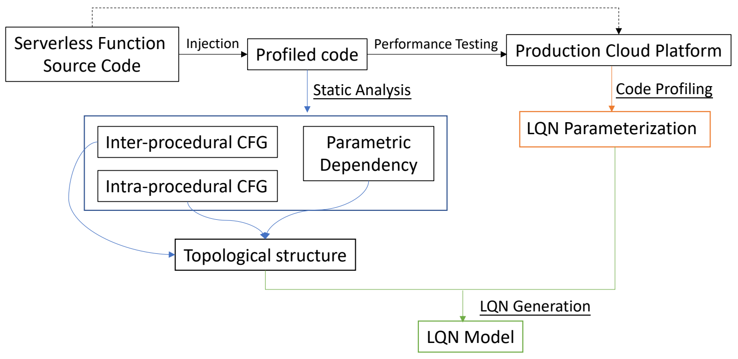


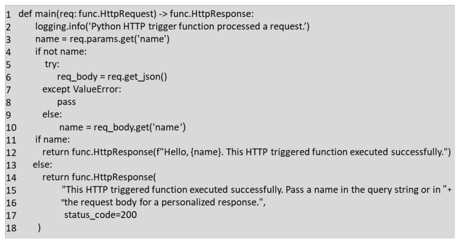
Fig. 2: The overview of the proposed methodology

whose performance models have been constructed with alternative methods can in principle be included in the LQN model as well, with possible increase of the overall uncertainty of the model. However, in this paper, we focus on modeling functions whose source code is accessible.) First, to define the topological structure of the workflow, we apply static analysis to extract the intra-procedural and inter-procedural control flow graphs from the source code of FaaS-based workflows, including individual functions and orchestration code composing them. Besides, we try to infer the dependencies between the input parameters of each individual serverless function and which function calls they affect; this can help to reduce the number of nodes in the topological graph. We then inject profiling instructions into the source code to enable code-level profiling during performance testing (we will refer to the instrumented code as *profiled code*). Next, the profiled code can be deployed on the production platform as required for performance testing and data collection. After exercising the test inputs, the service demand distribution is captured with the profiling data. The availability of the static graph also allows inspecting if any static execution path has not been covered, enabling the developer to decide whether 1) the static path is effectively not executable (e.g., the FaaS application does not require all the features of a library function, thus using only some of its possible behaviors), 2) the static path implements features not relevant to ensure the SLAs thus it was deliberately not exercised during performance testing, or 3) the performance test suite needs improvement to cover more missing relevant application behaviors. Finally, we can generate LQN models using the topological graph for components specification, and accurately characterize the model parameters with code-profiling data.

In the remainder of this section, we will detail each of the three phases.

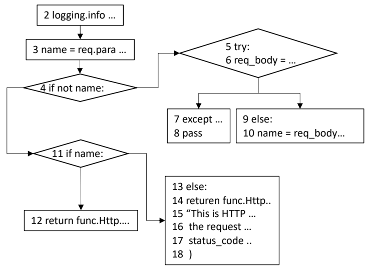
3.1 Static Analysis for Structure Extraction

We assume that there are two major components in a given FaaS-based application: an orchestration function defining workflows to compose individual serverless functions with suitable patterns (e.g., sequential or parallel execution), and a set of individual serverless functions implementing different functionalities. We first construct the topology of the LQN from static source code analysis. The static analysis provides a fast way to capture the internal control flows



```
1  def main(req: func.HttpRequest) -> func.HttpResponse:
2      logging.info('Python HTTP trigger function processed a request.')
3      name = req.params.get('name')
4      if not name:
5          try:
6              req_body = req.get_json()
7          except ValueError:
8              pass
9          else:
10             name = req_body.get('name')
11     if name:
12         return func.HttpResponse(f"Hello, {name}. This HTTP triggered function executed successfully.")
13     else:
14         return func.HttpResponse(
15             "This HTTP triggered function executed successfully. Pass a name in the query string or in "+
16             "the request body for a personalized response.",
17              status_code=200
18         )
```

(a)



(b)

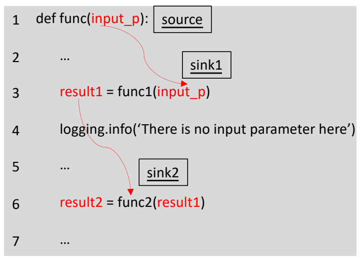
Fig. 3: An exampled source code in Python from Azure Functions (a) and the corresponding control flow graph (b).

by identifying (a superset of) all feasible paths of the programs. This can help build a complete topological graph, whereas certain parts could be missed in monitoring data if the inputs used to test the system do not cover all of its features extensively.

Static Graph Generation. In this paper, we use both inter- and intra-procedural static graphs to derive the topological structure of the LQN models. A call graph (CG) is mainly responsible for extracting the calling relationships on the workflow, which is extracted from the orchestration function. While intra-procedural information can be obtained by generating a control flow graph (CFG) for each serverless function. Both CG and CFG are constructed by traversing nodes in the abstract syntax tree (AST) on the profiled code, resulting in a collection of code blocks and control nodes representing conditional execution [23]. Combining both CG and CFG of the serverless workflow, we obtain an inter-procedural static description of the system that we call the static graph (SG) as $SG = \{CG, CFG\}$.

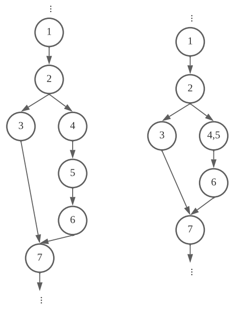
Given the control graph of an individual serverless function as $CFG = (\boldsymbol{N}, \boldsymbol{E})$, the control flow is formalized by conditions, loops, function calls, and sequential code blocks. In CFG, $\boldsymbol{N}$ and $\boldsymbol{E}$ denote the nodes and edges, respectively. In order to enable further analysis of the CFG, we then define each node N_i as a tuple (i, l_s, l_e), where l_s and l_e are the starting and end lines of the i^{th} code block. A directed edge in $\boldsymbol{E} = \{(N_i, N_j), \dots\}$ describes the relationship between nodes N_i and N_j. The example source code in Figure 3a is available at [1], and Figure 3b shows the control flow for the example serverless function. The resulting CFG is represented with $\boldsymbol{N} = \{N_1, N_2, \dots, N_9\}$, $\boldsymbol{E} = \{(1,2),(2,3),(3,4),(4,5),(4,6),(4,7),(3,7),(7,8),(7,9)\}$ and, for example, $N_9 = (9, 13, 18)$.

Data Flow Analysis for Parameter Dependency Inference. The static graph of serverless applications could be large due to redundant nodes representing the statements whose execution times are not influenced by a function's input parameters (e.g., constant time initialization). Thus, from a performance perspective, these statements could be aggregated into single blocks to reduce the size and fragmentation of the static graph. We apply a static data flow analysis to the profiled code to infer the potential parametric dependencies between input parameters and function calls using taint analysis. This can help reduce



```
1  def func(input_p):
2      ...
3      result1 = func1(input_p)
4      logging.info('There is no input parameter here')
5      ...
6      result2 = func2(result1)
7      ...
```

(a)



(b)

Fig. 4: An example excerpt of code with taint analysis results highlighted (a), and the control flow graph reduction process (b).

the number of nodes in a static graph by aggregating input independent nodes, in turn lowering the computation complexity of solving the inferred LQN models. Taint analysis in this phase works by marking a statement in the source code as tainted if its execution or assigned value is affected by function input values. The parametric dependency inference can be formulated as detecting any existing tainted statement in the nodes of the CFG. If there are input parameters used by the statement of any node, we then deduce that the execution times of the detected node N_i are potentially dependent on such input parameters. Formally, potential parametric dependencies can be described as a set of $PD_i = (source, sink, lineno)$, where $source$ and $sink$ are the identifiers of a function input parameter and a function call whose arguments are affected by the input parameter, while $lineno$ is the line number identifying the call site of the sink function to distinguish possible multiple calls.

If there is no potential parametric dependency detected in N_j, we can infer that the demand for executing N_j is not impacted by function inputs, resulting in a reduction of N_j by aggregating it with its predecessor. Figure 4a shows an example source code with the taint analysis results highlighted. Let the left graph in Figure 4b be the original CFG of the example code, with $N_4 = (4, 3, 3)$ and $N_5 = (5, 4, 4)$. The right graph in Figure 4b shows the reduction of node N_5 into node $N_{4,5}$ due to no detected parametric dependencies in node N_5.

Algorithm 1 formulates the control flow graph reduction process based on taint analysis. The algorithm takes the intra-procedural control flow graph and the detected parametric dependencies as inputs. The algorithm traverses N_i in the original CFG and checks if any potential parametric dependencies occurred at N_i. If no dependency exists, the current node N_i is added to the untainted set $\boldsymbol{r_N}$. At Line 6 and 7, the algorithm first finds all predecessors and successors of the untainted node and then revises the end line number of all predecessors with l_e of N_i. Then at Line 10, the untainted node is removed from the original graph. From Line 11 to Line 16, the algorithm iterates on the nodes and removes the edges containing the affected nodes. By fully connecting the nodes in $\boldsymbol{succ}$ and $\boldsymbol{pred}$, the new edges are generated to form the untainted edge set $\boldsymbol{E_r}$. After iterating on all the nodes in the original graph, the reduced graph $CFG_r =$

Algorithm 1 Control flow graph reduction with potential parametric dependencies

```
Input: $CFG \leftarrow$ Control flow graph of the source code $CFG = (\boldsymbol{N}, \boldsymbol{E})$
    $\boldsymbol{PD} \leftarrow$ Set of potential parametric dependencies $[PD_1, PD_2, \ldots, PD_n]$
Output: $CFG_r \leftarrow$ Optimized CFG with reduction on nodes
 1: Initialize $\boldsymbol{r_N} = \emptyset$, $\boldsymbol{E_r} = \emptyset$
 2: for $N_i$ in $CFG$ do
 3:   for $PD_i$ in $\boldsymbol{PD}$ do
 4:     if $lineno$ is not in the range of $[l_s, l_e]$ then
 5:       $\boldsymbol{r_N} \leftarrow \boldsymbol{r_N} \cup N_i$
 6:       $\boldsymbol{succ} \leftarrow$ all successors of $N_i$, $\boldsymbol{pred} \leftarrow$ all predecessors of $N_i$
 7:       update $l_e$ of $\boldsymbol{pred}$ to include $N_i$
 8:     end if
 9:   end for
10:   $\boldsymbol{N_r} \leftarrow \boldsymbol{N} \setminus \boldsymbol{r_N}$
11:   for $succ_i$ in $\boldsymbol{succ}$ do
12:     for $pred_i$ in $\boldsymbol{pred}$ do
13:       $\boldsymbol{E_r} \leftarrow \boldsymbol{E} \setminus (pred_i, N_i)$, $\boldsymbol{E_r} \leftarrow \boldsymbol{E} \cup (pred_i, succ_i)$
14:     end for
15:     $\boldsymbol{E_r} \leftarrow \boldsymbol{E} \setminus (N_i, succ_i)$
16:   end for
17: end for
18: return $CFG_r \leftarrow (\boldsymbol{N_r}, \boldsymbol{E_r})$
```

$(\boldsymbol{N}_r, \boldsymbol{E}_r)$ is generated by combining nodes in which there are no function calls or parametric dependencies.

It can be noticed that the parametric dependency inference is only capable of detecting potential relationship between function calls and input parameters from the code syntax. For example, consider `y = 0*x; f(y)`; most taint analyses would conclude that the invocation of `f` may depend on the input parameter `x`. This may lead to a conservative over-approximation, with possibly only a subset of the statically detected dependencies satisfied during runtime. In this case, the static graph could have been further reduced, realizing that `0*x` is identically 0. Nonetheless, even when non-optimal, taint analysis may still help to reduce the size of the static graph.

3.2 Code Profiling for Model Parameterization

Application-level monitoring data may be too coarse-grained to accurately infer service demand parameters of LQN activities, representing the operations inside individual serverless functions. We instead propose instrumenting the code of a function to obtain fine-grained measurements that can bridge the information gap between the granularity of the topology extracted via static analysis and the data used for model parameter inference.

Code-level profiling. To avoid changing any functionality of the source code and try to instrument the code as less as possible, we only wrap the *MAIN* function block into a wrapper function and inject a decorator to record the execution times with a standard line-level profiler [2] (while we refer mainly to Python code in this work, similar profiler utilities exist for all mainstream programming languages). Here, we take the assumption that the performance test inputs are representative of all relevant production behaviors. If executable paths in the static graph are not covered by the current test inputs, while they may affect the application's SLAs, the developer has the opportunity to identify the gap and produce additional performance tests.

We denote a sample from the collected profiling data as $s = (lineno, dt, iter)$, where dt is the execution duration of the statement at line $lineno$ and $iter$ is the iteration counter to distinguish different iterations in a loop. We can then map the profiling data into the static graph according to line numbers $lineno$ in s and (l_s, l_e) of nodes in CFG_r to extend the static graph with profiling data; we will refer to this extended structure as profiled static graph.

Besides obtaining the execution times of nodes in the graph, we also need to learn the probabilities of branches and the number of iterations for loops to infer the remaining parameters of an LQN model. For the branch probabilities, we define the executed path of each test input request $\boldsymbol{eP}$. Each $eP_i \in \boldsymbol{eP}$ represents one of the feasible paths in the static graph that has been executed according to the profiling data $\boldsymbol{s}$. Therefore, the probability of a given selection path^i on each conditional statement can be derived as the fraction of eP_i taking each branch over the number of eP_i evaluating the corresponding condition. For loop iterations, we represent the body of *for* or *while* loop as an entire activity and infer the expected number of iterations from the profiling data, which is consistent with the typical specification of LQNs. This can be further optimized by considering the branch probabilities inside loops to indicate a probabilistic loop, however, this is out of the scope of our current modeling method. In our proposed method, the number of iterations of loops in each execution can be directly extracted from the profiled data with $iter$.

LQN Activity Service Demand Distribution. Service demands are critical parameters for the specification of activities, as they represent the cumulative computation time the activity requires to run. To capture the demand of the activities in LQN, we model the service demand distribution with acyclic phase-type (APH) distributions and Erlang distributions by moment matching. Based on the execution duration dt in the profiling data, we first try to fit an APH distribution by matching the first three moments of dt. If there is no solution for APH distribution with the current data, we then fit an Erlang distribution with mean value and squared coefficient of variation (scv). In this way, the service demand of each activity can be directly characterized by the profiling data.

3.3 LQN Model Generation

To construct an LQN model from the profiled static graph, we first define a reference task to represent the incoming workload and an orchestration task to abstract the workflow logic composing individual serverless functions. Then, each individual serverless function can be modeled with a single task hosted on a separate processor, since it can be deployed with different configurations of resources and even to different platforms.

The entry node of the LQN is then specified according to the entry point of each function. We further assume that the sequential or parallel (fork-join) composition of the functions is specified in the orchestration function, e.g., using Azure Functions code constructs. The degree of concurrency allowed to each function is specified in the configuration metadata. The scheduling policy of the processor can be specified as either First-come-first-serve (FCFS), if the source code is with single-thread implementation, or Processor-sharing (PS) if function invocations can be interleaved on the same processor. Both scheduling policies are supported in LQN modeling [15].

LQN Activity Graph Characterization. The static graphs and profiling information collected so far allows for a systematic construction of the LQN model. First, we consider that the orchestration function is allowed to specify the workflow patterns with HTTP calls to invoke the individual serverless functions. Each activity inside the orchestration entry can be defined according to the nodes in the call graph and takes the role of sending synchronous and asynchronous calls for parallel execution to the entries of individual functions in the lower layers of the LQN. Whereas, the skeleton of the activity graph for an individual function can be directly derived from the reduced CFG. The activity graph representing the set of activities $\boldsymbol{act}$ is defined as $AG = \{\boldsymbol{act}, \boldsymbol{sd}, \boldsymbol{prec}\}$, where $\boldsymbol{sd}$ presents the service demand and the precedence relation among activities is denoted as $\boldsymbol{prec}$.

Now we discuss the procedure of activity graph specification for the serverless function f following the approach in [14]. For each activity representing $N_i^{'}$, all the successors and predecessors of $N_i^{'}$ are computed. There are mainly 4 types of activity precedence included in our method: (1) If the current node is included in its predecessors, it indicates that loops are occurring at $N_i^{'}$ which can be extracted with the number of iterations $iter_i$ derived from profiled data. (2) If the current node only has one successor and one or fewer predecessors, it means that $N_i^{'}$ is sequentially connected to its successor. (3) When there is more than one successor of $N_i^{'}$, there are branches with IF or $SWITCH$ statements for jumping to different nodes, whose branching probabilities have previously been computed from profiling data. (4) If there is more than one predecessor, different conditional blocks can be merged at $N_i^{'}$. From the orchestration function, we also capture parallelism and synchronization among the execution of different serverless functions. Combining all the listed cases, our method can describe the operator precedence in the activity graph including sequential interactions, conditioning and merging on branch nodes, as well as fork-join synchronization.

4 Evaluation

In this section, we first introduce the experimental setup and metrics to evaluate the accuracy of performance models constructed with our method. The comparison of LQN model predictions against execution monitoring traces for serverless workflows with different composition patterns is presented afterwards.

4.1 Experimental setup

To evaluate the proposed method for automatically building LQN models based on static analysis and code profiling, we first implement 4 serverless workflows including sequential, branching, parallel and complex execution scenarios. The source code of the serverless workflow implementation is available at [1].

We create 13 serverless functions and 4 orchestration functions to define a collection of common workflow patterns on Azure Functions Service. The in-

Table 1: FaaS-based workflow patterns

	$wf1$	$wf2$	$wf3$	$wf4$
Number of functions	8	9	8	14
c^2 of execution times	0.26	5.71	0.84	7.44

dividual serverless functions are adapted from public examples that use TensorFlow with Azure functions [3] and models from Onnx Model Zoo [4]. The functionality of different workflows includes preprocessing of input images and classification based on machine learning algorithms or pre-trained models. Some metrics for the composition workflows implemented by the 4 orchestration functions are shown in Table 1, where c^2 is the squared coefficient of variation of the execution times.

To evaluate the accuracy of our modeling method, we conduct several experiments with different workloads and compare the performance predictions from the LQN models against the application-level monitoring data of the serverless workflows. The experimental environment is as follows. All the individual serverless functions are developed with Python 3.7 and deployed with Azure Functions 3.0. We take the response times of requests from the real traces as ground truth to evaluate our model-based predictions. To collect the real traces, we use Azure Application Insights as the monitoring tool and expose the code profiling data on the same service. As workload-independent execution times can be profiled in isolation, we can perform offline profiling on the production platform as required to collect profiling data, and then deploy the non-instrumented functions to the target cloud service (without the profiling instructions) to collect application-level runtime monitoring data.

The static analysis is built on top of the *ast module* in Python 3.7. For the taint analysis on the static data flow, we use the open-source tool Pyre shipped with Pysa [6] to infer the potential parametric dependencies. To obtain the analytical results, we use LQNS via LINE to solve the generated LQN models [11]. For performance testing, we generate closed workloads with different intensities using Locust [5].

We compare the model prediction accuracy of mean response times to the collected traces, using mean relative error (MRE) as our comparison metric, where $MRE = |m - m'|/m$ is computed with the mean response times m of the monitored execution traces and m' for LQN predicted response times.

4.2 Experimental Result

We first evaluate the static graph reduction based on inferring static parametric dependencies. Next, to evaluate the accuracy of parameterization for LQN models, we conduct extensive experiments with different settings of the number of processors and the dynamic auto-scaling to simulate two resource provision scenarios. Here, we regard these two experimental settings as limited resources and sufficient resources in the following discussion.

LQN Model Node Reduction. In Section 3.2, we introduced Algorithm 1 to reduce the size of the static graph by aggregating code blocks independent of input parameters, with the ultimate goal of further reducing the size and complexity of the generated LQN models. We here compare the accuracy and

Table 2: The comparison results of based on LQN node reduction

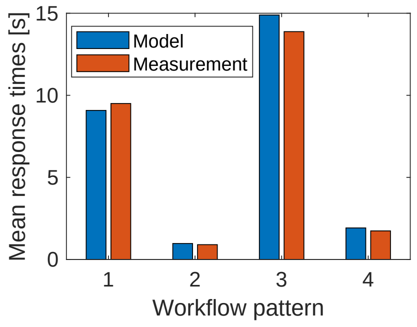
	Number of Activities		Execution times (s)		MRE	
	Original	Reduced	Original	Reduced	Original	Reduced
$wf1$	91	67	2.793	2.092	0.044	0.030
$wf2$	96	69	2.804	2.133	0.029	0.038
$wf3$	27	21	1.545	1.542	0.222	0.236
$wf4$	122	89	3.682	3.257	0.103	0.117

efficiency of the original LQN models to the reduced LQN models. From Table 2, we can observe that after node reduction, for example, the number of activities of $wf2$ is reduced to 69, which indicates that nearly 30% of nodes have been merged according to the static parametric dependencies. Besides, it can be noticed from the table that the execution times of solving model are decreased by up to 25% for $wf2$, while the MRE increases only to a small degree for all the subjects, which is likely an acceptable trade-off between prediction accuracy and model complexity in most situations. We can conclude that the reduction of nodes in static graphs can directly help to reduce the number of activities in the LQN model, thus saving analysis costs. The savings come with a marginal increase in the MRE for three out of four subjects, while the MRE marginally decreased for $wf1$. Overall, the impact of reduction on the MRE appears marginal.

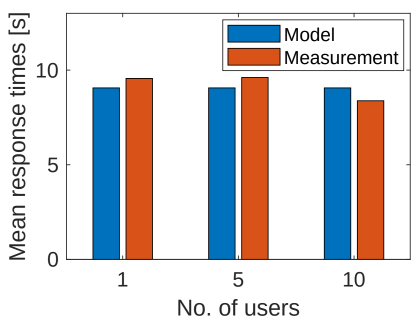
Sufficient Resource. In the following experiments with sufficient resources, we assume that dynamic auto-scaling is enabled for each FaaS function and there is no need to operate on the configuration of the resources. In LQN models, we set the multiplicities of each processor to 100 to simulate sufficient resource provision not to limit the scaling out of the individual functions.

We evaluate the above 4 different workflow patterns and take $wf1$ as an example under different workloads. The comparison of LQN model predictions against the real traces is shown in Figure 5, and the details of model accuracy evaluation are in Table 3. It can be seen from Figure 5a that the mean response times among different workflow patterns vary in a range of 0 to 15 seconds, while all prediction results are close to the measurements. Figure 5b, which zooms on $wf1$, shows that there is no obvious increase in response times as the number of clients grows. This is because under sufficient provision, all required resources can be allocated and there are no significant queueing times for each request. Therefore, the LQN modeling results capture the correct trend of response times changing with workloads. Besides, it can be observed from Table 3 that the prediction of the LQN model yields good accuracy with an average MRE over the four workflows of 5.5% (min=2.9% for $wf2$, max=10.3% for $wf4$), indicating a fairly accurate characterization of the performance of the FaaS workflows.

Limited Resource. For the limited resources experiments, we tune the configuration of each serverless function to variate the number of cores for the processor. Practically, we first identify the most resource-demanding serverless function as the bottleneck function and then study the accuracy of our model for different values of the maximum number of instances on Azure and, coherently, of the multiplicity parameter of the corresponding LQN activity.



(a)



(b)

Fig. 5: Mean response times of different workflow patterns (a) and of $wf1$ under different workloads (b), comparing model prediction and real trace measurements.

Table 3: MRE of compared results in Figure 5

	workflow pattern				workload		
	$wf1$	$wf2$	$wf3$	$wf4$	1	5	10
Model	9.081	0.927	10.144	1.923	9.081	9.081	9.081
Measurement	9.501	0.901	9.737	1.744	9.501	9.613	8.390
MRE	0.044	0.029	0.042	0.103	0.044	0.055	0.082

However, when a single processor is allowed, Azure Function Consumption plan limits the allocated memory to 1.5Gb, which in the case of $wf3$ and $wf4$ is not sufficient to serve 10 users without scaling strategies for the most resource-demanding serverless function. Therefore, we selected the second most resource-demanding serverless function as the bottleneck function for $wf3$ and $wf4$. The comparison of results on LQN model prediction and monitoring traces measurements is shown in Tables 4 and 5.

First, we investigate the model performance with increasing concurrent users between $N = 1$ to 10 with only one processor ($P = 1$) available for the bottleneck function. It can be seen from Table 4 that, with increased workload intensity, the mean response time grows with different trends. For example, in $wf1$, the response time with 10 users is nearly 5 times higher than with 1 user due to the contention on the bottleneck function that forces the users to wait. Nevertheless, regardless of the variation trends in the response time, the model predicts accurately the performance measurements from monitoring traces in all four workflows, with average MRE across all the experiments of about 6.2% (min=0.8% for $wf1$ with $N = 10$, max=12.2% for $wf4$ with $N = 10$).

Next, we evaluate model prediction accuracy using an intense workload (N=10) and varying the number of processors available for the bottleneck function. Table 5 shows the comparative data for the number of processors P between 1 and 10. As expected, increasing the number of processors reduces the response time for all the workflows, albeit with different trends. The average MRE across all

Table 4: Comparison results on different number of processors on 4 workflow patterns with limited resource $P = 1$

	$wf1$			$wf2$		
	$N = 1$	$N = 5$	$N = 10$	$N = 1$	$N = 5$	$N = 10$
Model	9.081	22.718	49.016	0.927	1.454	3.085
Measurement	9.501	24.397	48.606	0.901	1.394	2.952
MRE	0.044	0.069	0.008	0.029	0.043	0.045
	$wf3$			$wf4$		
	$N = 1$	$N = 5$	$N = 10$	$N = 1$	$N = 5$	$N = 10$
Model	10.144	10.664	12.545	1.923	2.229	3.215
Measurement	9.737	12.125	13.626	1.744	2.3	3.655
MRE	0.042	0.120	0.079	0.103	0.031	0.122

Table 5: Comparison results on different workloads on 4 workflow patterns with limited resource $N = 10$

	$wf1$			$wf2$		
	$P = 1$	$P = 5$	$P = 10$	$P = 1$	$P = 5$	$P = 10$
Model	49.016	10.325	9.083	3.085	0.928	0.927
Measurement	48.606	13.27	8.018	2.952	1.149	0.879
MRE	0.008	0.222	0.133	0.045	0.192	0.054
	$wf3$			$wf4$		
	$P = 1$	$P = 5$	$P = 10$	$P = 1$	$P = 5$	$P = 10$
Model	12.545	10.144	10.144	3.215	1.923	1.923
Measurement	13.626	8.600	10.489	3.655	2.031	1.895
MRE	0.089	0.180	0.033	0.120	0.053	0.015

the experiments is in this case about 9.5% (min=0.8% for $wf1$ with $P = 1$, max=22.2% for $wf1$ with $P = 5$). While average MRE remained under 10%, we observed a performance deterioration for $P = 5$. By observing the execution traces, we conjecture this deterioration may be caused of some implicit optimization or automation happening on the serverless platform around $P = 5$ which is not accurately captured by our models and may require additional investigation.

Summary. The evaluation of our LQN modeling strategy for serverless functions based on static analysis and code profiling may be summarized with the following two observations. First, node reduction on the static graph leads to smaller LQN models, saving computation time for both LQN model generation and model-based performance prediction, with negligible impact on prediction accuracy. Second, model-based performance prediction achieved a close fit to the measurements from monitoring traces (average MRE=7.3%), under different workload intensity and in both sufficient and limited resources. Finally, we remark that the availability of the static graph also allows assessing the coverage of the performance test inputs, highlighting possible execution paths relevant to the satisfaction of the application's SLAs that are not exercised (enough), thus driving the refinement of the performance test suite.

5 Related Work

The question of how to predict the performance of serverless functions is closely followed by researchers. However, fine-grained analytical performance modeling for serverless functions still lacks investigations to our knowledge. Eismann et al. [12] propose to use mixture density networks to predict the response time distribution of a single serverless function and then estimate the cost of serverless workflow execution by Monte-Carlo simulation. In [7], the authors develop a framework called COSE for serverless function configuration with a trace-based performance model. Based on the performance model, they apply Bayesian Optimization into obtaining the optimal serverless function configuration. These works can be identified as data-driven performance predictions for serverless functions, which cannot give an explicit, interpretable abstraction of a serverless application.

On the other hand, model-driven performance prediction can help developers and providers to better understand different performance prediction and reason about performance issues or design alternatives. Boza et al. [10] propose to use $M(t)/M/\infty$ queues to model serverless functions, enabling the calculation of performance and cost. Mahmoudi et al. [20, 21] propose an analytical performance model by using a continuous-time semi-Markov process to accurately predict the performance metrics. However, this work mainly focuses on modeling aspects of the computing platform to support tuning its configuration, and does not directly relate to the internals of serverless functions. Lin et al. [19] use probabilistic directed acyclic graph abstractions to predict the end-to-end response times of serverless applications. The smallest representable unit in this work is a whole serverless function, which may limit the performance prediction accuracy due to the coarse modeling granularity.

Finally, the generation of LQN models for software performance prediction have also been investigated starting from higher-level, architectural specifications, e.g., from UML [24] or Palladio Component Models (PCM) [8]. Recently, TOSCA specifications have been extended to specify several concerns of serverless applications [28] and can be used to generate LQN performance models. However, most of these approaches require expert knowledge to define accurate architectural models in the first place. This is typically expensive and error-prone due to the need to keep the models consistent with the actual implementation, which also requires manual instrumentation and adequate performance test suite to measure the implementation's performance.

6 Conclusion and Future Work

We presented a new method to build LQN performance models for serverless applications using information from static analysis to enhance model-based prediction accuracy. We exploit the relatively smaller size of serverless function implementations, together with advances in static analysis methods for modern programming languages, to extract intra- and inter-procedural control and data dependencies among functions and their invocation parameters at different call sites. The topological structures identified by these dependencies then drives both code-level performance profiling and the automatic generation of a succinct LQN model to reason about the performance of the application. Experimental results indicate that our method can accurately capture the characterization of FaaS workflows and yield accurate prediction results under different workloads and resource provisions.

Among the possible future research directions, we aim to explore the integration of performance modeling of FaaS-based applications with performance issues diagnosis. Intra- and inter-function LQN models can help to relate performance bottlenecks to code artifacts, potentially helping to locate the root causes of SAL violations.

References

1. https://anonymous.4open.science/r/Enhancing-Performance-Modeling-of-Serverless-Functions-via-Static-Analysis-828D
2. https://github.com/pyutils/line_profiler
3. https://github.com/Azure-Samples/functions-python-tensorflow-tutorial
4. https://github.com/onnx/models
5. https://locust.io/
6. Pyre. https://pyre-check.org/
7. Akhtar, N., Raza, A., Ishakian, V., Matta, I.: Cose: configuring serverless functions using statistical learning. In: IEEE INFOCOM 2020-IEEE Conference on Computer Communications. pp. 129–138. IEEE (2020)
8. Altamimi, T., Petriu, D.C.: Incremental change propagation from uml software models to lqn performance models. In: CASCON. pp. 120–131 (2017)
9. Baldini, I., Castro, P., Chang, K., Cheng, P., Fink, S., Ishakian, V., Mitchell, N., Muthusamy, V., Rabbah, R., Slominski, A., et al.: Serverless computing: Current trends and open problems. In: Research advances in cloud computing, pp. 1–20. Springer (2017)
10. Boza, E.F., Abad, C.L., Villavicencio, M., Quimba, S., Plaza, J.A.: Reserved, on demand or serverless: Model-based simulations for cloud budget planning. In: 2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM). pp. 1–6 (2017)
11. Casale, G.: Integrated performance evaluation of extended queueing network models with line. In: Winter Simulation Conference (WSC). pp. 2377–2388. IEEE (2020)
12. Eismann, S., Grohmann, J., Van Eyk, E., Herbst, N., Kounev, S.: Predicting the costs of serverless workflows. In: Proceedings of the ACM/SPEC International Conference on Performance Engineering. pp. 265–276 (2020)
13. Eismann, S., Scheuner, J., Van Eyk, E., Schwinger, M., Grohmann, J., Herbst, N., Abad, C.L., Iosup, A.: Serverless applications: Why, when, and how? IEEE Software **38**(1), 32–39 (2020)
14. Franks, G., Al-Omari, T., Woodside, M., Das, O., Derisavi, S.: Enhanced modeling and solution of layered queueing networks. IEEE Transactions on Software Engineering **35**(2), 148–161 (2008)
15. Franks, G., Maly, P., Woodside, M., Petriu, D.C., Hubbard, A., Mroz, M.: Layered queueing network solver and simulator user manual. Dept. of Systems and Computer Engineering, Carleton University (December 2005) pp. 15–69 (2005)

16. Garetto, M., Cigno, R.L., Meo, M., Marsan, M.A.: A detailed and accurate closed queueing network model of many interacting TCP flows. In: Proceedings IEEE INFOCOM 2001. vol. 3, pp. 1706–1715. IEEE (2001)
17. Israr, T.A., Lau, D.H., Franks, G., Woodside, M.: Automatic generation of layered queuing software performance models from commonly available traces. In: Proceedings of the 5th international Workshop on Software and Performance. pp. 147–158 (2005)
18. Khedker, U.P., Sanyal, A., Karkare, B.: Data flow analysis: theory and practice. CRC Press (2017)
19. Lin, C., Khazaei, H.: Modeling and optimization of performance and cost of serverless applications. IEEE Transactions on Parallel and Distributed Systems **32**(3), 615–632 (2020)
20. Mahmoudi, N., Khazaei, H.: Performance modeling of serverless computing platforms. IEEE Transactions on Cloud Computing (2020)
21. Mahmoudi, N., Khazaei, H.: Temporal performance modelling of serverless computing platforms. In: Proceedings of the 2020 Sixth International Workshop on Serverless Computing. pp. 1–6 (2020)
22. Marsan, M.A., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic Petri nets, vol. 292. Wiley New York (1995)
23. Nielson, F., Nielson, H., Hankin, C.: Principles of Program Analysis. Springer (2015)
24. Petriu, D.C., Shen, H.: Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications. In: International Conference on Modelling Techniques and Tools for Computer Performance Evaluation. pp. 159–177. Springer (2002)
25. Spinner, S., Casale, G., Brosig, F., Kounev, S.: Evaluating approaches to resource demand estimation. Performance Evaluation **92**, 51–71 (2015)
26. Tariq, A., Pahl, A., Nimmagadda, S., Rozner, E., Lanka, S.: Sequoia: Enabling quality-of-service in serverless computing. In: Proceedings of the 11th ACM Symposium on Cloud Computing. p. 311–327. SoCC '20, Association for Computing Machinery (2020)
27. Tripp, O., Pistoia, M., Fink, S.J., Sridharan, M., Weisman, O.: Taj: effective taint analysis of web applications. ACM Sigplan Notices **44**(6), 87–97 (2009)
28. Zhu, L., Giotis, G., Tountopoulos, V., Casale, G.: Rdof: Deployment optimization for function as a service. In: 2021 IEEE 14th International Conference on Cloud Computing (CLOUD). pp. 508–514. IEEE (2021)