CS8803-DNS: Reproducing ORION for ML Inference

Aniruddha Mysore
animysore@gatech.edu

Srihas Yarlagadda
syarlagadda37@gatech.edu

1 INTRODUCTION

Function-as-a-service or FaaS is a popular computational model that provides application developers the ability to directly write application logic that, in theory, can scale elastically to adapt to any number of requests without the developer having to explicitly manage resource scaling. FaaS has grown in popularity and is offered by several cloud vendors in the form of services like AWS Lambda and Azure Cloud Functions. These services have several issues that have been explored in past works, such as the need for manual capacity configuration, high data movement due to an absence of stateful abstractions, and the Cold-Start problem [8].

ORION [8] has been proposed to identify dependencies among the individual cloud functions of serverless applications and estimate a performance model that accounts for these dependencies. This information is then used to suggest optimal configurations for each cloud function and pre-warm containers for future functions in the dependency graph with the right lookahead time — optimizing for both low latency and the minimum possible idle time. ORION has been evaluated on three large serverless applications that comprise tasks like model training and chatbots.

In this work, we propose to reproduce the results of ORION for a machine-learning inference pipeline. The characteristics of ML inference naturally lend themselves to be a seemingly great application for the FaaS model — ML inference tasks are embarrassingly parallel and highly latency sensitive. However complex inference workloads can consist of pipelines with chained models; we believe that this workload is a promising candidate for ORION.

2 PROJECT OBJECTIVE

We evaluate the performance of ORION for a machine learning inference pipeline, specifically studying i) the effectiveness of the proposed statistical techniques for end-to-end latency modeling; ii) the impact of cold-start prediction and skew correction in the context of this workload. We also aim to better understand the nature of inference tasks as they relate to serverless environments, and the distribution of their execution profiles; this has can help better quantify the impact of the FaaS model on relevant performance metrics such as tail latency.

We evaluate ORION on the types of prediction pipelines used in current SOTA works [2], like social media applications, image processing, and model cascades. These are reflective of challenging real-world workloads and are currently of interest to the broader research community.

2.1 Falsifiable Hypothesis

State-of-the-art serverless runtime optimizers like ORION can be adapted to complex ML inference tasks like model pipelines and can be demonstrated empirically by comparing tail-latency metrics of ORION against non-serverless inference environments.

3 MOTIVATION

Serverless application deployments have multiple benefits for cloud providers. Virtualizing application logic into an abstract unit allows for flexible placement and scheduling policies, enabling better utilization of cluster hardware. With increasing costs imposed by heterogeneous accelerators and higher-capacity CPU hardware, this elasticity becomes valuable both in terms of monetary implications and evolvability. With the increasing popularity of inference in workflows, there is a natural need to extend this philosophy to a new class of applications.

Inference is particularly relevant as a use-case for a serverless deployment [6] — it is characterized by mostly stateless computation chains and is often used in parallel with web-serving workloads that are canonical serverless target applications. The structure of ML inference pipelines is strikingly similar to the serverless DAG characterization, and the paradigms share similar problem patterns as a result. We further conjecture that the predictability of inference execution — which has been used to guide batching and scheduling in prior work — can potentially lend itself well to the end-to-end latency modeling technique proposed by ORION. Nonetheless, prior work has not explored the effect of a serverless deployment model on inference *pipelines*, motivating our study.

While the authors of ORION have demonstrated the effectiveness of their method on three target workloads (chatbot, ML training, and video analytics), we believe that inference serving is a relevant case study for a serverless orchestration system — such a workload is also more sensitive to latency constraints in a large-scale deployment compared to training.

4 CHALLENGES

The increasing relevance of ML inference on the critical path of production applications has brought several challenges to light.

- **Inference characteristics** Meeting latency SLO targets requires predictable performance. Model heterogeneity and the data-driven nature of inference serving make it difficult to achieve this objective compared to traditional workloads such as web applications and content delivery. The higher configuration burden of inference is another obstacle; several hyperparameters such as batching affect latency.
- **Pipeline factors** Inference pipelines have additional characteristics that affect latency at the tail. There are multiple conditional paths through stages of the model pipeline — combined with model-specific variations this makes end-to-end latency more difficult to predict. Queueing delays between pipeline stages and the replication factor of each model are some additional factors that contribute to this problem.
- **Infrastructure** The effects of inference on latency predictability are compounded by lower-level system factors. Inference is a target for accelerators such as GPUs and

ASICs, with substantially varying on-device scheduling policies. Resource sharing and scheduling are thus made more complex by this inherent variability, often requiring custom policies to ensure scalability.

In sum, unpredictability is induced by the nature of ML inference applications, pipeline factors such as conditional paths, and the systems underlying the deployment. Despite these domain-specific challenges, modeling inference latency has been extensively researched — the efficacy of a more general approach like ORION will, however, be interesting to observe and characterize.

5 APPROACH

We reproduce the results of ORION on AWS Lambda, and review the codebase to identify which components require modification to fit our test case. This primarily includes the lambda deployment scripts and associated container images hosted on ECR, as well as the registered dependencies.

We have deployed custom Docker images and the associated Lambda functions for the social media pipeline in Fig:1. Pretrained models from huggingface https://huggingface.co/ were used - i) ReNet-18 for image classification, ii) A multilingual cased Bert model for language detection, iii) Allen-AI's MarianMT based WMT19 submission for German-to-English translation and iv) A fine-tuned version of Helsinki-NLP's opus transformer for Spanish-to-English translation. This represents a fairly varied set of models; this was a conscious choice to help generate a distribution of latencies.

Each model was deployed using TorchServe. This allows us to wrap the inference model with a REST interface that allows for reconfiguration — enabling us to modify bundle size dynamically at runtime using management API calls. The use of TorchServe imposes an additional startup latency on the first cold Lambda request — this is largely unavoidable due to process start overhead and extracting the model files from storage. Subsequent Lambda calls resume all TorchServe processes from the exit point of the previous invocation (a feature of the Lambda Runtime), allowing them to complete without this delay.

We have used the latency measurement method from ORION to extract the latency of each inference request. We use profiles of latency distribution to model both end-to-end and per stage-latency as continuous distribution functions (CDF). In our case, since ORION does not support conditional execution of pipeline branches, we used a single stage for the image translation. We believe this is an acceptable compromise since both traslation models are similar in size and performance. We use the Convolution operator to join the PDFs of the Language detection and Language translation functions which are executed in sequence and use the Max operator and to join the Probability Density Functions (PDF) of the Image and Language portions of the pipeline since they run in parallel, and compute the CDF from this combined PDF. These models function as the input to the best-first-search algorithm used to pick ideal container sizes given a latency target. We compare key metrics including tail latency against an equivalent testbed in handpicked deployment configurations to assess the effectiveness of the E2E latency model adopted by ORION. Optimizations such as cold-start mitigation using prewarming can be selectively enabled, allowing us to study them separately.

5.1 Artifacts and Platform

The codebase for ORION is open-sourced on GitHub (https://github.com/icanforce/Orion-OSDI22) and forms the basis for our experimentation. The platform used is AWS Lambda for serverless FaaS deployment, with Step Functions as a DAG orchestrator. The repository includes scripts for profiling and image deployment via Docker and the AWS CLI, as well as exporting Lambda function definitions and Step Function graphs. The codebase is a mix of Python and C#. Our fork of the codebase is available at https://github.com/SrihasY/DNS_Spring_2023/.

6 RELATED WORK

Serverless functions have been studied increasingly in recent years, owing to the elasticity and flexibility they afford for cloud providers to optimize costs. Previous work has sought to characterize and better understand the nature of serverless workloads [10] [1], aiming to improve placement and scheduling policies that improve resource utilization and minimize latency overheads. Models that have been developed for latency estimation of serverless functions rely on techniques such as regression [4] or mixture density networks [5] — these methods are fairly sample intensive and not directly extensible to DAGs, which is the target of ORION. The closest comparable work targeting serverless DAG scheduling includes Sequoia [11], SONIC [7] and Caerus [12]; the authors of ORION aim to extend these by accounting for the correlation between the latency distribution of stages in the DAG and deriving their impacts on cost and utilization. The focus on serverless DAGs is a specific feature of ORION that is relevant to ML inference pipelines — this enables workflows that use multi-model inference for stage-wise tasks.

Modeling the latency of ML inference has proven effective in prior work, forming the basis for inference serving systems such as Clipper [3] and Inferline [2]. Production use cases often involve serving many models where choosing the right model from a model zoo is non-trivial as shown in works like INFaaS[9]. Models can also form complex DAGs with the output of one model feeding into the input of another [2]. These prediction pipelines can be seen in varied applications from image processing (basic image preprocessing + image classification) and social media (image and text classification of a social media post along with hate speech detection), to cascade models where heavier models in the cascade are used only when necessary for certain workloads.

7 IMPLEMENTATION

7.1 Code

We have implemented the wrapper code needed to execute each of the models in the pipeline - language detection, image classification and language translation. The model and model weights were taken from PyTorch's preexisting model library and from HuggingFace. The model, model weights, and this wrapper code have been packaged into Model Archives that are used by the torchserve package.

We implemented Dockerfiles for each lambda task and a JSON file that expresses our pipeline DAG in the format required by AWS StepFunctions.

Nearly all the code in the DAG Modeler relating to loading profiler data and specifying the latency distributions as CDFs were hard-coded for the three benchmark applications in the original paper. We added a few hundred lines (C Sharp) to the original codebase to evaluate the right-sizing algorithm on our inference pipeline.

7.2 Experiments

We deployed and ran the DAG Profiler component of Orion on our pipeline - allowing us to collect end-to-end latency as well as the individual latencies of each function call. The results are summarized in the Evaluation section below. The startup delays were excluded from the aggregate latency results. After profiling, we augmented the DAG Modeler component to work with our inference pipeline, and ran the bread-first-search algorithm to obtain optimal memory sizes for each lambda in our deployment.

7.3 Unforeseen Challenges

- Containerizing Lambda applications requires support to be built-in for the AWS Lambda Runtime Environment (RIE), which is the agent responsible for pausing and resuming function invocations. TorchServe is Python-based but relies on a Java backend for deploying model threads — this meant that we could not use the base images provided by AWS, since they provide RIE-supported runtimes for only one programming language. We finally resorted to building a custom image based on Amazon Linux Corretto, installing Python and building support for RIE manually.
- We found that the performance of TorchServe on Lambda was significantly slower than a local test run, likely due to virtualization delay. This, in addition to the fairly high memory requirements imposed by TorchServe, meant that we needed to downsize models in order to enable them to run with lower memory configurations. This has implications for bundling experiments, since multiple parallel workers will be spawned.
- Expressing the pipeline as a StepFunctions DAG took longer than expected, partially due to the StepFunctions UI which made debugging any change challenging.
- Due to the DAG latency distribution functions being hard-coded, we could not directly use our DAG as an input to test Orion's modeling strategy. This required us to implement the CDF of our pipeline using details from the paper and base classes defined in the code.

8 EVALUATION

The baseline latency statistics for the overall inference pipelines as well as each individual function are reported in Tables 1, 2, 4. Tables 3 and 5 show the same statistics when Orion is applied to find optimal memory configurations, directly comparing with Tables 2, 4 respectively; running with a target of 3.1s resulted in the same configuration as manual, with all lambda memory maxed. The startup latencies (latency of the first cold invocation) have been ommitted in these calculations, a sample set of observed startup latencies is in the Appendix (Table 8). Finally, Table 6 shows the effect of bundling function invocations for the ResNet lambda.

Table 1: Manual baseline results, achieved P95 = 3.1s

Function Name	Mean	Variance	Tail (P95)	Memory (MB)
langdetect	1025.5	51.71	1108	4000
es2en	1302.27	371.12	1947.5	2048
de2en	1547.72	244.82	1861.4	2048
resnet	321.12	50.21	407.1	2048
End-to-End	2603.24	336.16	3108.35	-

Table 2: Manual baseline results, achieved P95 = 2.1s

Function Name	Mean	Variance	Tail (P95)	Memory (MB)
langdetect	758.58	47.47	834.35	6000
es2en	825.21	223.93	1205.15	4096
de2en	925.42	126.27	1069.4	4096
resnet	264.75	32.17	324	4096
End-to-End	1796.01	208.02	2148.5	-

Table 3: Orion performance, target P95 = 2.1s

Function Name	Mean	Variance	Tail (P95)	Memory (MB)
langdetect	748.44	62.99	822.4	5632
es2en	844.35	231.60	1237	4096
de2en	916.42	127.92	1065.8	4096
resnet	314.46	62.86	450.6	2048
End-to-End	1778.06	197.42	2139.05	-

Table 4: Manual baseline results, achieved P95 = 1.5s

Function Name	Mean	Variance	Tail (P95)	Memory (MB)
langdetect	531.87	21.97	570	10240
es2en	591.8	140.66	845	10240
de2en	623.4	79.15	728.7	10240
resnet	239.95	43.2	317.2	10240
End-to-End	1302.93	127.85	1505.2	-

Table 5: Orion performance, target P95 = 1.5s

Function Name	Mean	Variance	Tail (P95)	Memory (MB)
langdetect	563.79	69.59	664.7	10240
es2en	591.8	140.66	845	8640
de2en	623.4	79.15	728.7	8640
resnet	315.85	60.5637447	440.65	2048
End-to-End	1333.89	150.61	1613.45	-

Table 6: Bundling ResNet

Bundle Size	Mean	Variance
2	267.1	45.69
3	593.9	157.13
4	1087	223.95

All latency statistics were calculated over 100 profiling runs, while 10 runs were used for generating bundling data in accordance with the paper.

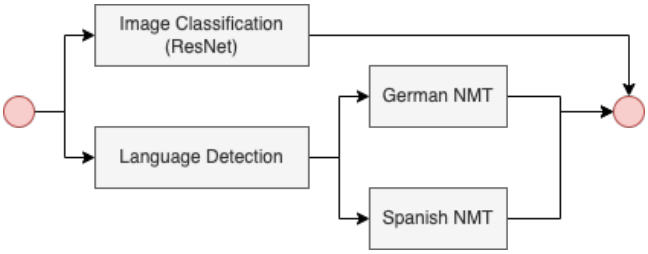

Figure 1: Social Media ML Pipeline

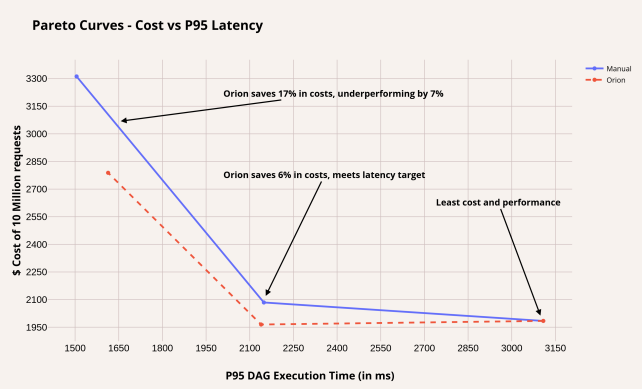

Figure 2: Pareto Curves - Orion vs. manual baselines

8.1 Observations

- Orion's modeling strategy successfully applies to ML inference. For both latency targets, we observe that the critical path of the DAG is identified and scaled appropriately, whereas the ResNet is kept at its base value to minimize cost.
- Using the optimal memory sizes, we observe a 6-17% cost reduction. The performance is close to the target P95 - displaying a drop of around 7% in the worst case (Figure 2). This loss can be compensated by decreasing the latency target at slightly higher additional cost.
- The right bundle size can have benefits in utilization of Lambda resources, potentially translating to cost savings — for the anecdotal ResNet instance, a bundle size of 2 provides similar results to a bundle size of 1. However, bundling shows higher variance across the board. This is likely due to contention between worker processes of the same Lambda, showing that the choice is non-trivial in latency-sensitive scenarios.
- The exercise has helped us narrow down the downsides of Orion for inference, and more generally for any serverless DAG.
  (1) There is no explicit support for dynamic control flow - all potential paths need to be modeled as a single stage. This is not suitable for paths with substantially varying durations.
  (2) The profiling data required is quite significant, despite being low compared to other SOTA works as the paper claims.
  (3) Bundling does not account for interference effects, and simply scales memory sizes with number of bundled workers.

8.2 Timeline

We have completed the following tasks:

- **Feb 24** | Progress Report 1: Artifact reproduced, Evaluation pipelines and baselines determined
- **March 18** | Mid-Semester: Evaluation pipeline and baseline implementation
- **April 1** | Progress Report 2: Preliminary analysis after evaluating ORION with social media pipeline
- **April 25** | Final Presentation: Results compiled, tested bundling and latency modeling

8.3 Required Resources

We used the AWS free-tier to experiment with ORION. Free-tier limits were exhausted running the target ML inference pipelines, but the total cost was negligible.

9 RESPONSIBILITIES OF DIFFERENT TEAM MEMBERS

- **Inference Models** - Both
- **Modifying Orion** - Both
- **Containerization and Lambda Deployment** - Srihas
- **StepFunctions DAG Definition + Testing** - Aniruddha
- **Updating DAG Modeling** - Aniruddha
- **Implementing bundling and collecting data** - Srihas

REFERENCES

[1] Robert Cordingly, Wen Shu, and Wes J Lloyd. 2020. Predicting performance and cost of serverless computing functions with SAAF. In *2020 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/CyberSciTech)*. IEEE, 640–649.

[2] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: Latency-Aware Provisioning and Scaling for Prediction Serving Pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 477–491. https://doi.org/10.1145/3419111.3421285

[3] Daniel Crankshaw, Xin Wang, Giulio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2016. Clipper: A Low-Latency Online Prediction Serving System. https://doi.org/10.48550/ARXIV.1612.03079

[4] Simon Eismann, Long Bui, Johannes Grohmann, Cristina L. Abad, Nikolas Herbst, and Samuel Kounev. 2020. Sizeless: Predicting the optimal size of serverless functions. https://doi.org/10.48550/ARXIV.2010.15162

[5] Simon Eismann, Johannes Grohmann, Erwin van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the Costs of Serverless Workflows. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering* (Edmonton AB, Canada) *(ICPE '20)*. Association for Computing Machinery, New York, NY, USA, 265–276. https://doi.org/10.1145/3358960.3379133

[6] Jiawei Jiang, Shaoduo Gan, Yue Liu, Fanlin Wang, Gustavo Alonso, Ana Klimovic, Ankit Singla, Wentao Wu, and Ce Zhang. 2021. Towards Demystifying Serverless Machine Learning Training. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 857–871. https://doi.org/10.1145/3448016.3459240

[7] Ashraf Mahgoub, Karthick Shankar, Subrata Mitra, Ana Klimovic, Somali Chaterji, and Saurabh Bagchi. 2021. SONIC: Application-aware data passing for chained serverless applications. In *USENIX Annual Technical Conference (USENIX ATC)*.

[8] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[9] Francisco Romero, Qian Li, Neeraja J. Yadwadkar, and Christos Kozyrakis. 2021. INFaaS: Automated Model-less Inference Serving. In *2021 USENIX Annual Technical Conference (ATC)*. USENIX Association, 397–411. https://www.usenix.org/conference/atc21/presentation/romero

[10] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. *arXiv preprint arXiv:2003.03423* (2020).

[11] Ali Tariq, Austin Pahl, Sharat Nimmagadda, Eric Rozner, and Siddharth Lanka. 2020. Sequoia: Enabling Quality-of-Service in Serverless Computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 311–327. https://doi.org/10.1145/3419111.3421306

[12] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: Task Scheduling for Serverless Analytics. 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

A APPENDIX

Table 7: Reproduced E2E Latency (in ms) of 10 runs, ML training pipeline

CherryPick	Max Memory	Orion
130448	20575	51081
124772	20969	32600
121249	18932	27412
115815	18451	27829
121262	18913	28517
118253	19356	25631
129741	19692	25649
135916	18707	27582
117543	18528	27925
120839	18413	29511

Figure 3: AWS Step function DAG for the training pipeline

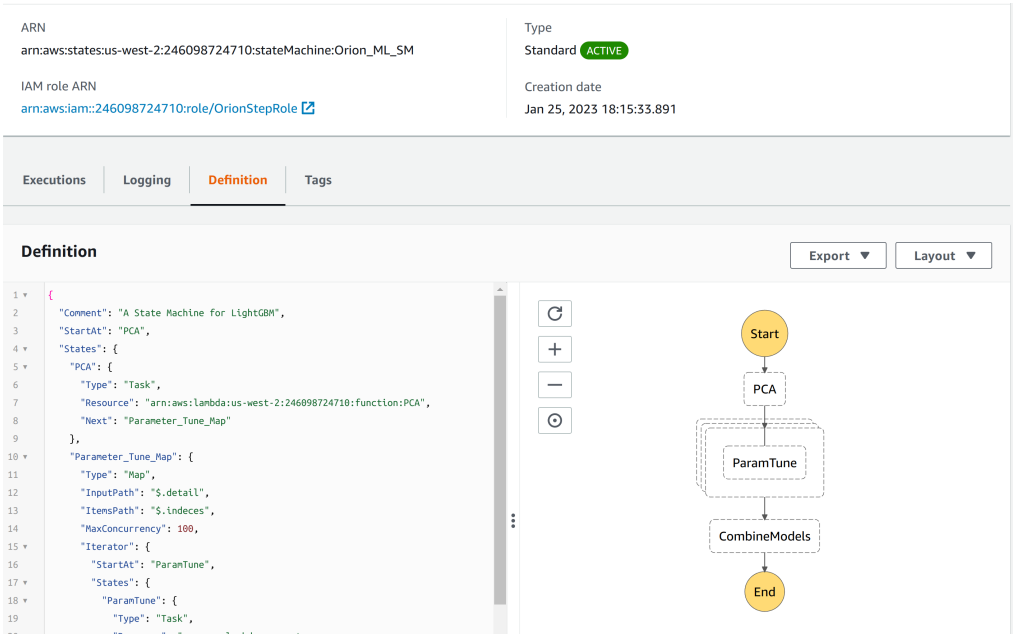

Table 8: Startup latencies observed, sample (2048,4000)

Function Name	Latency (ms)
langdetect	27451
es2en	27899
de2en	85809
resnet	11939
End-to-End	86975

Figure 4: AWS Step function DAG for the inference pipeline