Hyperspecialized Compilation for Serverless Data Analytics

Leonhard Spiegelberg[1], Tim Kraska[2] and Malte Schwarzkopf[1]

[1] *Brown University, Providence, Rhode Island*
[2] *MIT, Cambridge, Massachusetts*

Abstract
Serverless functions can be spun up in milliseconds and scaled out quickly, forming an ideal platform for quick, interactive parallel queries over large data sets. Modern databases use code generation to produce efficient physical plans, but compiling such a plan on each serverless function is costly: every millisecond spent executing on serverless functions multiplies in cost by the number of functions running. Existing serverless data science frameworks therefore generate and compile code on the client, which precludes specializing this code to patterns that may exist in the input data of individual serverless functions. This paper argues for exploring a trade-off space between one-off code generation on the client, and *hyperspecialized* compilation that generates bespoke code on each serverless function. Our preliminary experiments show that hyperspecialization outperforms client-based compilation on typical heterogeneous datasets in both cost and performance by 2–4×.

1. Introduction

Designing an efficient data analytics framework that utilizes serverless functions is challenging, as it must balance parallelism, communication, and runtime costs. Many modern databases and data analytics systems allow end-users to write queries in familiar languages like SQL or Python, but generate code and compile these queries into native machine code for efficiency [1, 2, 3, 4, 5, 6, 7]. Using compiled code in a serverless setting makes sense, as more efficient code directly lowers costs and avoids merely parallelizing overheads [8]. Code generation, and compilation into machine code naturally fit on the client, which knows the query and can generate code before dispatching hundreds or thousands of parallel serverless functions (“Lambdas” for short in the rest of this paper) that each operate over a part of the input data. Existing serverless frameworks like Starling [9] or Lambada [2] therefore employ code generation on the client machine, and invoke Lambdas with the generated plan in form of a custom runtime executable or shared object, which avoids compilation costs on individual Lambdas. But what if we performed *code generation* and *compilation* on *individual* Lambda functions?

This fine-grained code generation and compilation allows harnessing additional opportunities for performance optimization: as each Lambda processes a small part of the input data (e.g., a day’s worth) and many datasets have shifting distributions and patterns over time, code generation can produce specialized, more efficient code if it knows the input data distribution. This allows a system, for example, to specialize the code to schema changes that occur over time, to constant-fold values that change rarely (e.g., years), or to fit to other patterns in the input data, such as data sorted by categories. In other words, while compiling the same code on each Lambda is wasteful, our idea is to generate different *specialized* code paths on *individual* Lambda functions to offset compilation overheads by obtaining more efficient code for execution. As every millisecond on a Lambda is expensive and comes at a premium over longer-running provisioned resources, it becomes critical to hit the right trade-off between ahead-of-time work on the client and the Lambdas and the runtime reductions realized.

Our approach, *hyperspecialization*, demonstrates that compilation on individual Lambdas is feasible and beneficial to craft efficient data analytics frameworks on top of serverless functions. We present preliminary results from a prototype hyperspecializing system, Viton, built on top of an existing analytics system for Python workloads, Tuplex [1]. Our preliminary findings indicate that compilation for subsets on Lambdas can lead to both cost and efficiency improvements by 2–4×.

2. Motivation

Python became the dominant language for writing modern data science pipelines due to its rich universe of packages and popular data processing frameworks like Pandas or PySpark. Similarly, writing serverless functions in Python is attractive for data scientists, as the benefit of the quick launch of a Python runtime [10] together with the parallelism of thousands of serverless functions makes Python attractive for large-scale data processing when trying to minimize end-to-end runtime.

For example, PyWren [11] is a popular framework that combines Python, AWS Lambda serverless functions, and storage via S3 without the need to provision a cluster first

Joint Workshops at 49th International Conference on Very Large Data Bases (VLDBW’23) — Workshop on Serverless Data Analytics (SDA’23), August 28 - September 1, 2023, Vancouver, Canada
leonhard@brown.edu (L. Spiegelberg); kraska@mit.edu (T. Kraska); malte@brown.edu (M. Schwarzkopf)

to run simple queries that can be expressed as a sequence of `map` operations, with each map operation taking a user-defined function (UDF) as a parameter. PyWren's limited API only allows for simple data analytics workloads that apply a UDF f to each of N input rows stored within S3, but it demonstrates that processing large quantities via serverless functions relying on Python is feasible and scales nearly linearly.

However, this scalability comes at a cost: for increased dataset sizes, the benefit of the Python runtime's low startup times gets eclipsed by the slow execution speed for the actual processing work in the Python UDFs. A data scientist might be tempted to simply increase the parallelism level to reduce runtime, but this could be an expensive mistake: each millisecond wasted due to slow execution rapidly multiplies by the number of Lambda functions invoked—e.g., spending an extra second on $5,000$ Lambdas on AWS with 1GB memory each translates to an added \$0.08 cost. Reducing end-to-end runtime by scaling up the parallelism may therefore end up merely parallelizing Python overhead, hiding a higher-than-necessary total compute cost (in cycles and dollars).

A possible answer is to instead generate efficient machine code, similar to what an optimizing C/C++ compiler may produce. This is a tried-and-tested approach in a single-machine setting, but making it work for interactive queries on Lambdas poses new challenges.

3. Code generation for Lambdas

Code generation improves runtime efficiency for queries at the expense of a one-time compile cost, which amortizes when running over sufficiently large input data. Indeed, code generation (either fine-grained, or by templating and combining query fragments) and subsequent compilation are a standard way to produce an efficient physical plan. In the serverless setting, this raises the question where and how to generate and compile a query.

Code generation blocks query execution. Compiling on the client machine (or via a dedicated compilation service) is cost-effective, as no Lambda functions are invoked, but also limits the parallelism to the client machine and blocks query execution until this machine finishes compiling the plan. Generating C/C++ code is a popular choice because it makes code generation easy, but C/C++ compilers like Clang or GCC take a long time to generate code with optimizations enabled. For example, Meta reports that its unified execution engine, Velox, which uses C/C++ templating and code generation, takes tens of seconds to generate code, invoke a C/C++ compiler, and produce a shared library to load into the execution engine [12]. While ahead-of time code generation for queries can be cost-effective, as shown in proof-of-concept engines like Starling [9], it can become the dominant cost in query execution as the serverless function parallelism increases and per-function runtime shrinks, making it harder to amortize long compile times.

Vectorized execution engines that rely on pre-compiled primitives trade-off shorter compile time against missed optimization potential for generated code and larger code size compared to fully-compiling, fine-grained execution engines. Thus, it becomes difficult to provide both efficient code and low, interactive end-to-end query latency by relying on a classic compiler.

Heterogeneity and marginal optima. In cases where the data distribution varies across subsets of the input data, compiling different code paths may be beneficial. Generating individual code for subsets of the data would allow a system to *locally* specialize and emit optimized code that may outperform a single, globally optimized code path. By compiling different code paths in parallel on individual Lambdas, the system can also prevent stalling execution when all Lambdas would otherwise need to wait on the physical plan to compile on the client machine.

Given the HTTP request model of Lambdas, existing techniques involving multiple code-paths—such as on-stack replacement, where an existing code-path is replaced on-the-fly with a more performant version [13]—are challenging to realize, as serverless environments allow only for limited communication and synchronization between individual Lambdas (or require extensive effort to overcome network limitations [14]), and provide no bidirectional communication channel to the client.

Pre-baking code in the form of specialized primitives, as proposed in micro-adaptivity [15], may benefit long-running queries, but could also lead to high runtime costs when swapping between paths too often, or miss out on optimization potential when relying on primitives that are too coarse-grained.

Low startup times come from light runtimes. To guarantee fast startup times, images for Lambda functions should be as small as possible.[1] A common optimization is to use warmed-up instances by keeping "hot" containers around, via warmup calls or by paying a premium to the vendor (e.g., AWS Lambda provisioned functions). Caching techniques on the service side [16, 17, 18, 19] or loading only necessary application code during runtime [20] can also help to drive down overheads.

Frameworks that are able to compile most of the user-supplied logic reduce the image size by including only minimal runtime and compile logic. This much reduces startup time compared to including a full language interpreter and all dependencies, even though it may require shipping compiled code from the client to individual Lambdas, or compiling code on them.

[1] Image size restrictions (e.g., 250MB on AWS Lambda) can be overcome using a container registry at the cost of higher startup time.

4. Hyperspecialization

The central idea of *hyperspecialization* is to generate bespoke, specialized code for each input slice rather than to rely on a single, global specialization. As emitting different code paths benefits only heterogeneous datasets, we focus on such in the following. For homogeneous datasets, a system would automatically disable hyperspecialization, or let users do so explicitly.

4.1. Challenges

The overall challenge of hyperspecialization is that any cost to perform hyperspecialization weighs against the performance benefits of better-fitted code. In particular, a hyperspecializing query compiler must avoid situations where hyperspecialization performs worse than just a single, globally-generated code path.

Balancing optimization cost. One key challenge is to balance where the system generates, optimizes, and executes code. Typically, the client machine issuing the query to each Lambda executor has limited parallelism and a slow connection to a blob service like S3. However, any compute time spent on the client machine is essentially free, whereas every single millisecond spent on a Lambda multiplies by the parallelism employed. Keeping overheads low on each Lambda is crucial, but spending too much time on the client to generate and optimize code results in a slow query and a bad user experience.

In Viton, our hyperspecializing query compiler, we find a compromise: Viton performs a raw global optimization using a cheap sample on the client that it uses to split a query into stages, to project an initial set of columns, and to perform logical optimizations (like pushing filters through joins). Re-optimization on the Lambdas then resolves any initial sampling errors Viton may have incurred on the client and addresses heterogeneity within the input data. With this design choice, Viton balances the cost of too much optimization and code generation on a Lambda versus increased end-to-end time.

Balancing sampling cost. To generate a new specialized code-path, a Lambda must draw an input data sample for its specific input slice from S3. Controlling the sampling cost here is challenging, as the system must avoid issuing too many S3 requests and spending cycles parsing many rows, but must also ensure that the sample is representative. For example, sorted input data easily provokes sampling errors where using randomized sampling or sampling the first and last rows only.

Viton issues two S3 requests to get a block of fixed size of the start and end of a file to base the initial sample on. To further reduce sampling cost, Viton uses stratified sampling instead of parsing all available rows in the received data blocks. With stratified sampling, Viton partitions the input data into groups (strata) of equal size, and draws

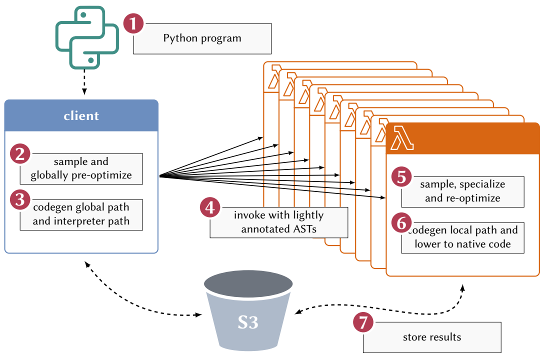

1
Python program
client
2
sample and globally pre-optimize
3
codegen global path and interpreter path
4
invoke with lightly annotated ASTs
5
sample, specialize and re-optimize
6
codegen local path and lower to native code
S3
7
store results

Figure 1: Viton system architecture: the client performs initial sampling and code generation, but each serverless Lambda function further samples and specializes to its particular input.

an identical number of random samples from each group. Picking random samples within a group avoids sampling errors. Viton then detects whether the Lambda's input data distribution differs from the global distribution. If so, Viton triggers re-optimization of the complete stage assigned to the Lambda, which fits both logic and data representation tightly to the concrete input data the Lambda is about to process.

4.2. Design

We base the design of Viton on a setting in which a single client machine issues AWS Lambda requests for data stored in S3. Viton divides query execution into two steps when it comes to planning, reflected in the overall system architecture (Figure 1). In a first step, which executes on the client, Viton draws a small initial sample from S3 to estimate an initial data distribution for the query to perform initial query planning steps, like detecting the schema, deciding which stages to generate, and collecting globally helpful statistics to derive a global physical plan. Viton intentionally keeps the sampling on the client cheap, as it expects hyperspecialization to adapt the query during execution. Viton also generates and compiles a general code path that is globally optimized and serves as a fallback on each Lambda executor when subsets are similar in distribution or hyperspecialization on an executor fails.

Viton then executes each stage using parallel Lambda executors. With hyper-specialization mode active, Viton assigns each Lambda a specialization unit. While there may be different strategies on how to identify and assign specialization units, in Viton, each input file serves as a specialization unit. We base this choice on the assumption that data sets are often partitioned by initial attributes, such as time. Thus, individual files marginalize the data distribution such that marginal distributions have overall lower variance. For historical data, this is typically the time of collection, but other schemes exist (e.g., categorical grouping or sorted data).

In the second step, each Lambda draws a new sample and re-optimizes the stage if the data distribution differs from the global sample. In order to re-optimize a stage on a Lambda executor, Viton ships logical operators together with associated UDFs in the form of lightly annotated abstract syntax trees (ASTs).

Specializing code on each Lambda on the new sample allows the specialization to combine *logical* with *compiler* optimizations, with each potentially benefiting the other. For example, a UDF may require different input columns to be parsed for input data from different years, but a globally optimized pipeline would always parse all the union of all required input columns. By re-optimizing the code locally and detecting common branches (a compiler optimization), Viton avoids parsing unnecessary columns in the first place (a logical pushdown optimization). Likewise, Viton could remove operators that become dead code, or reorder filters based on patterns in the data.

To make hyperspecialization work, the cost of executing all these steps has to be low enough to be offset by a performance gain through a more efficient code path. Viton uses aggressive optimizations, which may work for a subset of the data, but would likely fail if applied globally.

4.3. Optimizations

Viton adds two additional, aggressively-specializing speculative optimizations to those already in Tuplex [1].

Constant folding applies when an input data column is constant (e.g., a year or month), and allows Viton to remove deserialization of constant data and eliminates unnecessary code. While constant folding is a well-known compiler optimization, Viton applies it as a logical optimization to avoid deserialization.

Filter promotion assumes that a filter condition always holds or fails, which reduces code complexity by eliminating any future checks on the filter condition and allows Viton to base other optimizations only on sample rows that pass the filter. In the best case, filter promotion fully collapses individual operators.

These two optimizations are examples of a broader class of speculative optimizations that may be effective locally on subsets of a dataset. They also benefit logical optimizations when, e.g., they reduce the set of input columns required.

4.4. Implementation

We implemented our Viton prototype on top of Tuplex [1]. Creating Viton required adding support for more aggressive optimizations that can exploit properties of marginal distributions, and extending the early-stage Lambda backend of Tuplex to support shipping stages in the form of abstract syntax trees (ASTs) to Lambda executors. For this, we implemented a custom AWS Lambda runtime as this was more efficient in micro-benchmarks than building on top of existing runtimes in AWS Lambda. In addition to implementing per-Lambda, per-input file sampling, and hyperspecialized code generation, Viton also adds support for semi-structured JSON files with a parser built on top of simdjson [21].

5. Preliminary Results

We configure each Lambda to run a single Viton executor that uses up to 10 GB of memory and a maximum of three threads. As of June 2023, a Lambda instance with 10 GB of memory has six vCPUs, three of which we use for processing and three for S3. We run the client on a single `r5d.xlarge` EC2 instance. For our preliminary evaluation, we evaluate two queries.

Flights query. This query performs data cleaning over the flights dataset [22], but imputes missing values for delay factors prior to 06/2003, and retrieves a cleaned result for the years 2002–2005. Due to a schema change, delay information prior to 06/2003 was collected only as a single, aggregate delay factor in the form of one column which then changed into collecting detailed information breaking down delays into several delay factors using additional columns. The input data consists of 410 files (83.51 GB total) with sizes from 177–284 MB, each containing data for one month between 10/1987 to 11/2021.

Github query. The second query analyzes historical data in the Github Archive dataset collected from Github since February, 2011 [23], which contains raw information about 20+ events. Within this dataset, data is organized as newline-delimited JSON files for each day. Schema changes due to introduction of new fields are frequent (e.g., there are 3,748 changes over 417 days [24]). In addition, the schema of each row varies depending on the event type and time of collection, as data collection used multiple APIs with different response schemas over time. Due to resource constraints, we limit out experiment to a subset of eleven files for October 15th of each year (35.5GB total). We run a query that, for each fork event, extracts the number of commits, original repository ID, and when a fork happened.

Results. We evaluate the potential of hyperspecialization by measuring the runtime improvements that specialized code paths provide. We keep files in each dataset partitioned as they were in the original dataset, including heterogeneous input file sizes, and measure performance without hyperspecialization (i.e., vanilla Tuplex [1]), with hyperspecialization using only Tuplex's existing optimizations (e.g., speculating on NULL values), and with aggressive hyperspecialization, which adds the two new optimizations from §4.3. A good result would show hyperspecialization reducing the querys' end-to-end runtime and monetary cost.

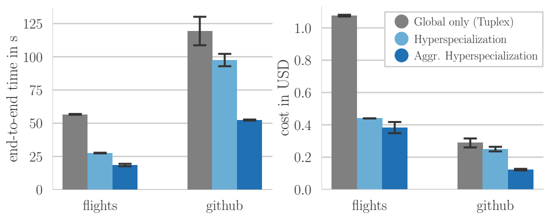
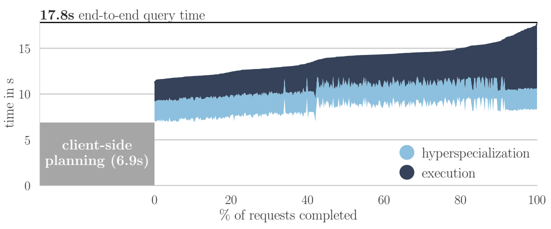
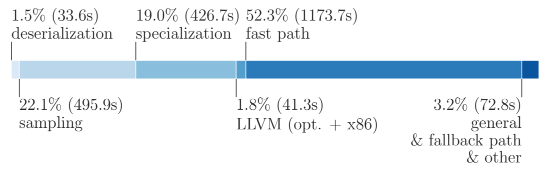


Figure 2: Hyperspecialization reduces end-to-end runtime by 3.05× and 2.27× for the *flights* and *github* queries, and reduces cost by 2.8× and 2.3×. The *github* query's runtime is larger due to longer-running executors, but cost is lower than the *flights* query's, which uses higher parallelism. Times are averages of 10 runs after a single warm-up run.



(a) Overall query breakdown over time.



(b) Cumulative time spent on Lambda executors.

Figure 3: Breakdown for a single execution of the flights query: Lambda executors spent 44% of time on hyperspecialization overheads, but with hyperspecialization enabled still better end-to-end performance and cost is achievable here.

Figure 2 shows the results. Hyperspecialization both makes existing optimizations more impactful ("hyperspecialization"), reducing runtime by 1.25–2×, and enables extra, aggressive optimizations ("aggr. hyperspecialization") that further reduce runtime for a total runtime gain of 2.3–3×. These reduced runtimes translate into 2.3–2.8× lower cost-per-query. The *github* query sees larger improvements from the extra optimizations, as it benefits from both filter promotion and constant folding, while the *flights* query only benefits from constant folding. These early results indicate that generating hyperspecialized code paths for sufficiently large specialization units can yield overall improvements in cost and performance, amortizing any overheads incurred.

Breakdown. We now break down an individual run of the *flights* query to understand the overheads of hyperspecialization. Figure 3 shows a timeline of the query. Viton spends 6.9 seconds on the client retrieving data from S3, sampling globally, and generating the global code path.

	Total	Total $	per λ	ct per λ
Lambda exec. time	2231.6s	$0.370	5.44s	0.091ct
deserialization	33.6s	$0.006	0.08s	0.001ct
sampling	495.9s	$0.083	1.21s	0.020ct
specialization	426.7s	$0.071	1.04s	0.017ct
LLVM (opt. + x86)	41.3s	$0.007	0.10s	0.002ct
Σ overheads	997.6s	$0.166	2.43s	0.041ct
fast path	1173.7s	$0.196	2.86s	0.048ct

Figure 4: Breakdown of an example flights query execution which took 17.8s end-to-end with 410 parallel Lambdas.

Afterwards, Viton's Lambdas spend about 44% of their 7–10 second execution time on hyperspecialization (Figure 3a), and the remainder of execution time processing data. Figures 3b and 4 further break down the time spent on Lambdas. Sampling takes about one second and code generation and compilation take about 1.2 seconds per Lambda. Importantly, it was necessary to restrict ourselves to a set of cheap LLVM optimizations in order to achieve this quick optimization time. Combined with other overheads, the total overheads of hyperspecialization come to 2.43 seconds, while 2.86 seconds are spent running the specialized fast path, and 0.15 seconds on the general compiled code path or in the interpreter. The initial time spent on the client could be reduced by caching information about files stored in S3, as the client spends most of the time accessing S3.

These results indicate that hyperspecialization is effective and can amortize its overheads sufficiently to provide end-to-end runtime reduction and cost savings.

6. Conclusion and Outlook

In this paper, we introduced the idea of hyperspecialization. Our preliminary results indicate that hyperspecialization is a promising direction. Further work will need to investigate several research questions.

What specialization unit size to pick? We want to quickly identify large, distinct subsets of input data and compile efficient code for them. However, optimizing too narrowly may fail to amortize the overheads of hyperspecialization despite improvements in performance. New techniques to identify regions for which hyperspecialization is a good idea and for a query optimizer to utilize this information are needed.

How to handle scenarios where compilation cost is high? Interpreters with JIT-compilation support typically compile only small code regions like individual loops or functions. But query compilation for a full query can become prohibitively expensive. Automating the process of detecting when to perform costly compilation within a serverless setting, and what optimizations are affordable, is part of the set of research questions we are just starting to understand better.

Acknowledgments

This research was supported by a Meta PhD fellowship. We thank Ben Givertz, Yunzhi Shao, Andrew Wei, Rhea Goyal, Shreeyash Gotmare, Khemarat (March) Boonyapaluk and Rahul Yesantharao for their contributions to Viton's implementation. This research was supported by NSF awards DGE-2039354 and IIS-1453171, and by funding from Google and VMware.

References

[1] L. Spiegelberg, R. Yesantharao, M. Schwarzkopf, T. Kraska, Tuplex: Data Science in Python at Native Code Speed, Association for Computing Machinery, New York, NY, USA, 2021, p. 1718–1731. URL: https://doi.org/10.1145/3448016.3457244.

[2] I. Müller, R. Marroquín, G. Alonso, Lambada: Interactive data analytics on cold data using serverless cloud infrastructure, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 115–130. URL: https://doi.org/10.1145/3318464.3389758. doi:10.1145/3318464.3389758.

[3] J. Sompolski, M. Zukowski, P. Boncz, Vectorization vs. compilation in query execution, in: Proceedings of the Seventh International Workshop on Data Management on New Hardware, DaMoN '11, Association for Computing Machinery, New York, NY, USA, 2011, p. 33–40. URL: https://doi.org/10.1145/1995441.1995446. doi:10.1145/1995441.1995446.

[4] T. Neumann, Efficiently compiling efficient query plans for modern hardware, Proc. VLDB Endow. 4 (2011) 539–550. URL: https://doi.org/10.14778/2002938.2002940. doi:10.14778/2002938.2002940.

[5] K. Krikellas, S. D. Viglas, M. Cintra, Generating code for holistic query evaluation, in: 2010 IEEE 26th International Conference on Data Engineering (ICDE 2010), IEEE, 2010, pp. 613–624.

[6] R. Y. Tahboub, G. M. Essertel, T. Rompf, How to architect a query compiler, revisited, in: Proceedings of the 2018 International Conference on Management of Data, SIGMOD '18, Association for Computing Machinery, New York, NY, USA, 2018, p. 307–322. URL: https://doi.org/10.1145/3183713.3196893. doi:10.1145/3183713.3196893.

[7] W. Zhang, J. Kim, K. A. Ross, E. Sedlar, L. Stadler, Adaptive code generation for data-intensive analytics, Proc. VLDB Endow. 14 (2021) 929–942. URL: https://doi.org/10.14778/3447689.3447697. doi:10.14778/3447689.3447697.

[8] F. McSherry, M. Isard, D. G. Murray, Scalability! but at what COST?, in: Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS), 2015.

[9] M. Perron, R. Castro Fernandez, D. DeWitt, S. Madden, Starling: A scalable query engine on cloud functions, in: Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 131–141. URL: https://doi.org/10.1145/3318464.3380609. doi:10.1145/3318464.3380609.

[10] D. Jackson, G. Clynch, An investigation of the impact of language runtime on the performance and cost of serverless functions, in: 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), 2018, pp. 154–160. doi:10.1109/UCC-Companion.2018.00050.

[11] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, B. Recht, Occupy the cloud: Distributed computing for the 99%, in: Proceedings of the 2017 Symposium on Cloud Computing, 2017, pp. 445–451.

[12] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, B. Chattopadhyay, Velox: meta's unified execution engine, Proceedings of the VLDB Endowment 15 (2022) 3372–3384.

[13] G. M. Essertel, R. Y. Tahboub, T. Rompf, On-stack replacement for program generators and source-to-source compilers, in: Proceedings of the 20th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2021, Association for Computing Machinery, New York, NY, USA, 2021, p. 156–169. URL: https://doi.org/10.1145/3486609.3487207. doi:10.1145/3486609.3487207.

[14] M. Wawrzoniak, I. Müller, R. Fraga Barcelos Paulus Bruno, G. Alonso, Boxer: Data analytics on network-enabled serverless platforms, in: 11th Annual Conference on Innovative Data Systems Research (CIDR 2021), 2021.

[15] B. Răducanu, P. Boncz, M. Zukowski, Micro adaptivity in vectorwise, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, SIGMOD '13, Association for Computing Machinery, New York, NY, USA, 2013, p. 1231–1242. URL: https://doi.org/10.1145/2463676.2465292. doi:10.1145/2463676.2465292.

[16] D. Du, T. Yu, Y. Xia, B. Zang, G. Yan, C. Qin, Q. Wu, H. Chen, Catalyzer: Sub-millisecond startup for serverless computing with initialization-less booting, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems,

ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 467–481. URL: https://doi.org/10.1145/3373376.3378512. doi:10.1145/3373376.3378512.

[17] A. Singhvi, A. Balasubramanian, K. Houck, M. D. Shaikh, S. Venkataraman, A. Akella, Atoll: A scalable low-latency serverless platform, in: Proceedings of the ACM Symposium on Cloud Computing, SoCC '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 138–152. URL: https://doi.org/10.1145/3472883.3486981. doi:10.1145/3472883.3486981.

[18] D. Mvondo, M. Bacou, K. Nguetchouang, L. Ngale, S. Pouget, J. Kouam, R. Lachaize, J. Hwang, T. Wood, D. Hagimont, N. De Palma, B. Batchakui, A. Tchana, Ofc: An opportunistic caching system for faas platforms, in: Proceedings of the Sixteenth European Conference on Computer Systems, EuroSys '21, Association for Computing Machinery, New York, NY, USA, 2021, p. 228–244. URL: https://doi.org/10.1145/3447786.3456239. doi:10.1145/3447786.3456239.

[19] L. Ao, G. Porter, G. M. Voelker, Faasnap: Faas made fast using snapshot-based vms, in: Proceedings of the Seventeenth European Conference on Computer Systems, EuroSys '22, Association for Computing Machinery, New York, NY, USA, 2022, p. 730–746. URL: https://doi.org/10.1145/3492321.3524270. doi:10.1145/3492321.3524270.

[20] X. Liu, J. Wen, Z. Chen, D. Li, J. Chen, Y. Liu, H. Wang, X. Jin, Faaslight: General application-level cold-start latency optimization for function-as-a-service in serverless computing, ACM Trans. Softw. Eng. Methodol. (2023). URL: https://doi.org/10.1145/3585007. doi:10.1145/3585007.

[21] G. Langdale, D. Lemire, Parsing gigabytes of json per second, The VLDB Journal 28 (2019) 941–960.

[22] Bureau of Transportation Statistics, United States Department of Transportation, Reporting carrier on-time performance (1987-present), 2020. URL: https://www.transtats.bts.gov/Fields.asp?Table_ID=236.

[23] I. Grigorik, Github archive, https://www.gharchive.org/, 2023.

[24] Github, Changelog - github docs, 2022. URL: https://docs.github.com/en/graphql/overview/changelog.