Exploiting the Cloud Control Plane for Fun and Profit

Josef Spillner

January 11, 2017

Abstract

Cloud providers typically charge for their services. There are diverse pricing models which often follow a pay-per-use paradigm. The consumers' payments are expected to cover all cost which incurs to the provider for processing, storage, bandwidth, data centre operation and engineering efforts, among others. In contrast, the consumer management interfaces are free of charge as they are expected to cause only a minority of the load compared to the actual services. With new service models and more complex and powerful management abilities, it is time to rethink this decision. The paper shows how to exploit the control plane of AWS Lambda to implement stateful services *practically for free* and partially even *guaranteed for free* which if widely deployed would cause a monetary loss for the provider. It also elaborates on the consistency model for AWS Lambda.

1 Motivation

Cloud computing is an economic paradigm as much as it is a technical one. Its popularity can be attributed in part to economic advantages such as a utility pricing model. Applications on the cloud use a defined set of programming interfaces (collectively an API) to settle in, connect to all dependency services and influence certain hosting parameters such as scheduling and scaling. Beyond this use of the control plane, they rely heavily on the computing plane to transmit and store data, to invoke compute services and to interact with other applications and services.

According to a widely used definition, the control plane facilitates the interaction between hosts (or other resources and services) in the cloud infrastructure as well as between the client and the cloud [2]. As such, most cloud providers decided to not put a price tag on it. Given that conventionally no computing can be performed on it, this has been a reasonable decision.

This way of thinking may now be over, however. With the Function-as-a-Service (FaaS) model, tiny stateless computing units at nanoscale are promoted. The cost model is adapted accordingly. One FaaS offering is AWS Lambda for

which convincing cost benefits have been shown [6]. Lambda also allows for tiny storage within its control plane. While not accessible from the computing plane, this combination of free storage and almost free computation will be explored and exploited in this paper for classes of stateful applications with inherent access to the control plane.

After presenting helpful background information, the paper assesses the situation and then proceeds to explain how to exploit it. During this process, it contributes a description of some of the characteristics of AWS Lambda as well as a consistency model. Finally, the findings will be discussed in the direction of how future cloud applications and the underlying programmable platforms and infrastructure may be designed.

2 Background

2.1 Cloud Services and Applications

From an application perspective, the cloud computing service model is defined by an application layer (SaaS) on top of two hosting layers (PaaS/IaaS). Platform and infrastructure services primarily offer computing capabilities such as data transfer and queueing, virtual machine execution and blob storage. Invoking these services may invoke cost or may be for free, with or without authentication in the form of tokens or other credentials. To the application and dedicated cloud management tools, the platforms also offer a control plane which is bound to a user account and which generally only allows authorised access to predefined management interfaces.

The separation of the control plane from other planes originates in the networking domain where other refers to the data plane [5]. In the domain of cloud computing, the set of others naturally extends to other resource types, namely compute and storage services, collectively called the computing plane. Correspondingly, there are management or control interfaces which are either provider-specific, as is the case with AWS, or generic, as for instance OCCI which also includes application-level interfaces [3]. The abstract notion of separated computing and control planes is shown in Fig. 1.

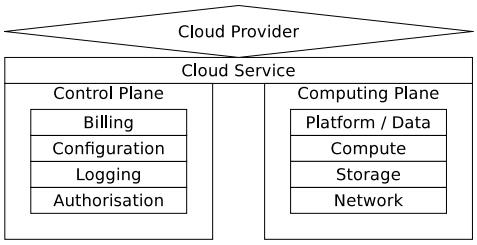

Figure 1: Computing and control planes within a cloud service

2.2 Filesystems and Data Dispersal

The cloud control plane typically limits the amount of data which can be stored to smaller amounts. In order to achieve practical capacities, these small blocks need to be combined. This technique resembles the organisation of storage devices where many fixed-size or variable-sized blocks are combined and organised. The combination can be as simple as striping and as complex as optimal erasure coding; and the organisation can be as simple as a table of linear contents and as complex as a modern filesystem.

In this paper, we will make use of simple tree-shaped table-of-content (ToC) structures where a root block contains pointers to subordinate blocks and collections of blocks are numbered to avoid an excessive amount of pointers. The model maps well to environment variables in the following form: $ROOT = A, B, C; LEN_A = \{3\}; LEN_B = \{19\}; LEN_C = \{1\}; A_1 = *; A_2 = *; A_3 = *; B_1 = ...$

3 Situation Analysis

Our work focuses on AWS Lambda, a service offered since late 2014 by Amazon Web Services [1]. AWS Lambda is a service which hosts a set of Lambda functions per region. There are 16 regions available although new ones are added every few months by the provider. Out of these, 14 are general-purpose regions (exempting the US government cloud and the Chinese region) and 10 offer Lambda. Each region is characterised by a human-readable name (e.g. US West / Northern California) and an internal name which is also reflected in the domain name (e.g. $us - west - 1$).

Each Lambda function consists of a name, a code implementation, a set of environment variables (since late 2016), a trigger and further configuration. The trigger could be an AWS API Gateway which translates external HTTP requests into Lambda invocations. Seven other triggers exist, most of which listen to events within other services of the same provider. Upon triggered invocation, the code is executed and any data input and output is realised through services, for instance AWS S3 for persistent blob storage. Despite the name, Lambda function code objects may encompass multiple actual functions.

3.1 Function Code

Lambda functions are implemented in a supported programming language and follow an expected interface which requires at least one function, named entry point, to adhere to a certain signature. When invoked, a context object and a JSON-formatted event structure is given as parameter, and a JSON structure is likewise expected as return value. Supported programming languages include JavaScript in the Node.js flavour, Python, Java and C#. The entry point function in Python is defined as follows: $def\ lambda\_handler(event, context)$: $return\ \{\}$. The name of the function can be configured through the configuration setting *handler*. For brevity, we will use f from now on. The code

implementation is mandatory and will be checked according to a basic parsing and code analysis upon deployment. The minimal accepted, albeit invalid and not executable, implementation in Python would thus read as follows: $def\ f() : pass$. The net code size is then 12 bytes, even though each Lambda function adds an unavoidable overhead of 136 bytes presumably for configuration, leading to a minimum possible size of 148 bytes for each Lambda function. It should be noted that the name of the Lambda function itself does not influence this size; thus, a Lambda function verbosely and uniquely called $square\_root$ can call a subordinate programming language function ambiguously called f.

The limits of AWS Lambda mandate a per-region maximum of 75 GB of deployed code. This implies that 544125924 or roughly 544 million minimal Lambda functions could be deployed in each region. We assign this value to the constant $\#F$.

3.2 Function Environment

Lambda functions can be annotated with a set of key-value strings which are made available as environment variables to function instances. While the same information could be represented in the function code, this model cleanly separates the concern of implementation and configuration, and allows for faster updates in particular for larger Lambda functions. Each variable needs to adhere to the following syntactical regular expression: $[a-zA-Z]([a-zA-Z0-9_]*[a-zA-Z0-9])+]$. This implies that a two-character name (e.g. AA) is the shortest possible one. Furthermore, each value needs to adhere to the expression $[\wedge,]*$ which implies that commas are not allowed.

The limits of AWS Lambda concerning the environment mandates a maximum size of 4096 per Lambda function. This limit is enforced over the JSON representation of the set of variables which reads as follows: $\{"AA" : "..."\}$. This implies a loss of 9 bytes so that 4087 bytes remain usable for the value in the extreme case of only having one variable. We assign this value to the constant S. For completeness, it should be mentioned that further environment variables are provided by AWS Lambda itself which do not count into the quota.

3.3 Combination

The environment variables configuration of AWS Lambda could be used to store arbitrary binary data. In this case, it would have to be represented as chunks in a suitable encoding. A safe encoding resulting in only uppercase and lowercase characters, numerals and equal signs is Base64 which reduces the usable net size by 25% of $\#F$. Due to padding on multiples of three bytes of input or four bytes of output respectively, this means that 3063 arbitrary bytes can be stored. We assign this value to the constant B.

Fig. 2 summarises the allocation of 4096 bytes when the goal is the maximisation of the payload area for binary data.

Eq. 1 combines our analytical results on the function code and on the respective environment.

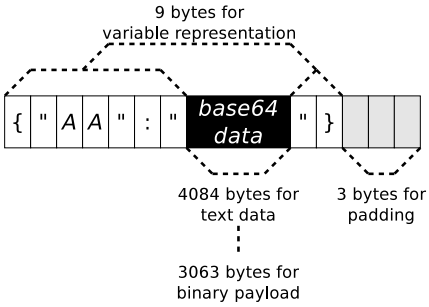

Figure 2: Byte-level representation of environment variables configuration in AWS Lambda

$$\begin{aligned} \#F &= 544125924 \\ B &= 3063\ bytes \\ \#FB &= 1666657705212\ bytes \\ &\approx 1.51\ TB \end{aligned} \tag{1}$$

Hence, we conclude that in total about 1.51 TB of data can be stored per region in the unaccounted control plane of AWS Lambda which corresponds to a typical notebook solid state disk capacity.

When the function should be invocable, its code needs to be adapted. A minimal implementation is $importos|def(e,c) : returnos.getenv("AA")$ which increases the code size by 31 bytes. When the data should furthermore be globally accessible, for instance without authentication, the AWS API Gateway needs to be switched on with a dedicated resource in front of each Lambda function. The code then grows by another 39 bytes due to the necessary JSON format of the return value: $importos|def(e,c) : return"statusCode" : 200,"headers" : ,"body" : os.getenv("AA")$. The function then becomes accessible by POST requests to the resource $https : // < api - id > .execute - api. < region > .amazonaws.com/prod/ < lambdaname >$. One important limitation of the gateway is that only 3 (fixed, not increasable) $CreateDeployment$ requests per minute are permitted which contradicts the rapid instantiation of micro- or nanoservices requiring dozens or more resources in a short period of time. Likewise, the number of resources is limited. Not considering these limitations, the increased function size reduces the theoretic maximum of retrievable data to 1.25 TB and 1.03 TB, respectively.

...

4 Situation Exploitation

We argue that by placing computational elements into the control plane of a cloud service, the provider's pricing model can be effectively forgone. Our con-

tribution to demonstrate this hypothesis is a set of two software applications both of which benefit from the unmetered network connection. The first application is a backup tool for storing and retrieving data which furthermore exploits the storage capabilities to achieve a *guaranteed for free* service level. The second application is a special-purpose database which exploits both storage and compute capabilities and due to the pricing model achieves at least a *practically for free* service level.

4.1 Backup Application

Lambackup is a shell script which splits files into chunks to store them into the environment variables section of generated Lambda functions. Furthermore, the script can retrieve and reassemble the files. The length of a file can be determined deterministically by storing one more chunk of information containing the file length on position zero (LEN), or heuristically by retrieving chunks until an error occurs. The script can also maintain a growing list of filename records including a counter chunk as table of contents (TOC).

...

Fig. 3 gives an impression of the transmission performance of Lambackup when storing two files which take 4 and 48 functions for storage, respectively, one time in minimal configuration and one time in the default LEN+TOC configuration. It is evident that the throughput is extremely low but stabilises for files larger than just a few kilobytes to a predictable $1kB/s$. The overhead of using LEN+TOC becomes then negligible.

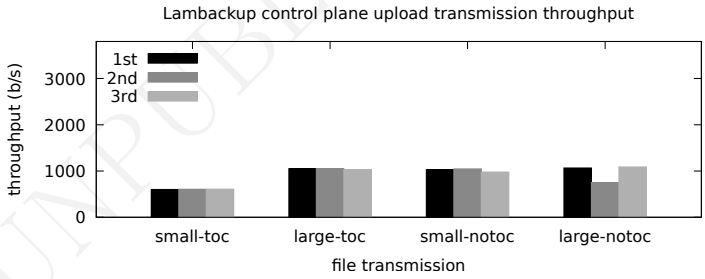

Figure 3: Lambackup backup file transmission throughput with and without recording of file length and table of contents; 3 consecutive runs

The restore operation is faster as no Lambda functions need to be created and no overhead due to transmitting the function code itself occur. Fig. 4 demonstrates the corresponding restore throughput. The reachable practical limit is about $2kB/s$.

Figs. 5 and 6 show the decreased backup throughput but in turn increased restore throughput when applying the FST options for downloading chunks through the API Gateway and through an aggregation function invoked via

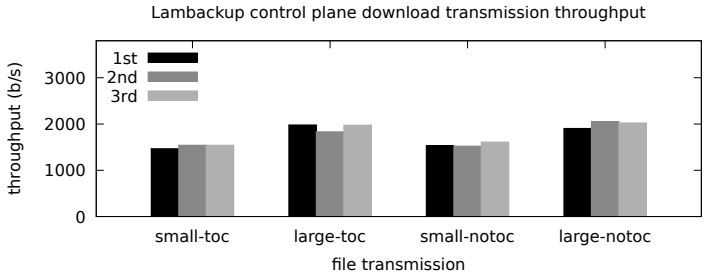

Figure 4: Lambackup restore file transmission throughput associated to Fig. 3

the control plane. Compared to Figs. 3 and 4, both experiments also use LEN+TOC.

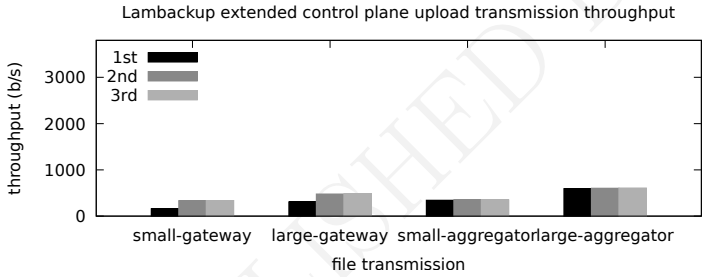

Figure 5: Lambackup fast backup file transmission throughput; 3 consecutive runs

Table 1 summarises the results and compares the characteristics of the environment data transmission options.

Table 1: Comparison of Lambda environment data transmission.

Option	**Upload Perf.**	**Download Perf.**	**Cost**	**Limits**
Default	939–1016 b/s	1608–2023 b/s	0	... xxx
LEN+TOC	604–1043 b/s	1541–1975 b/s	0	... xxx
API Gateway	336–481 b/s	2366–4324 b/s	$\approx 0^*$	... xxx
Aggregator	349–608 b/s	5234–35507 b/s	$\approx 0^{**}$	... xxx

*: ...; **: ...

4.2 Database Application

LaMa is a pseudo-relational database management system consisting of a client tool and a set of Lambda functions. As unique design criterion, all tables,

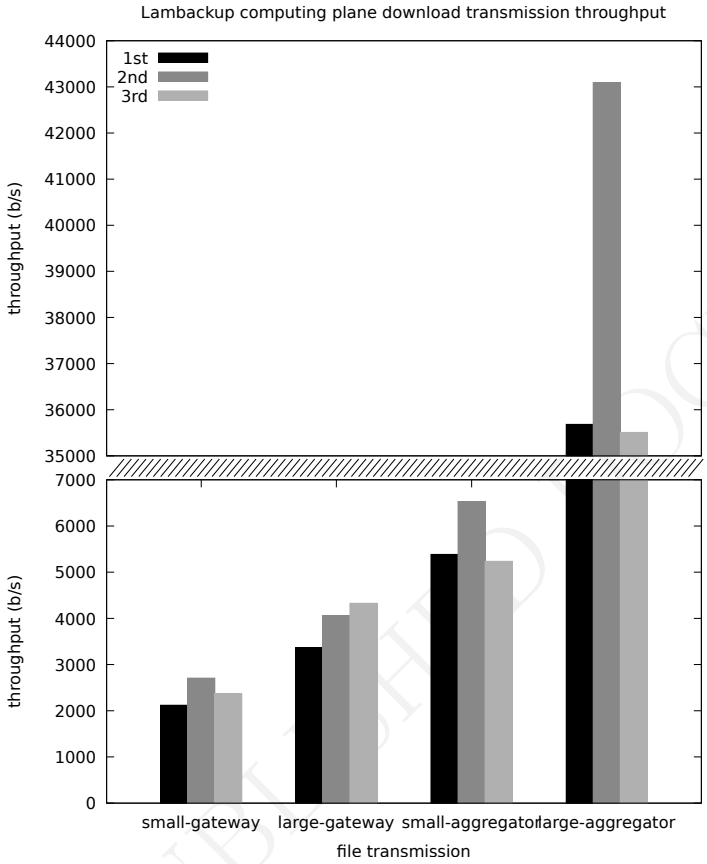

Figure 6: Lambackup fast restore file transmission throughput associated to Fig. 5

columns and data as content of the columns are stored in the environment variables section of Lambda's configuration. Only read-only queries are performed through Lambda functions whereas all modifications are required to use the configuration interface due to the service's security model which does not allow access to the control plane from the computing plane.

LaMa's data model is dynamic, supporting only one data type ($DYNAMIC$) with a basic subset of the Structured Query Language (SQL). Of interest in such a design are two properties, performance and correctness. The implementation, whose query frontend is shared with StealthDB [4], has been measured by running a sequence of SQL commands for 100 times. The sequence drops the table if it exists, (re-)creates it as single-column table, inserts 100 values and queries the entire data without $WHERE$ clause.

Fig. 7 shows the results. The execution performance is relatively stable

around an average of 16.4s on the test system with a few slower outliers having up to 58% higher time requirements. More interesting is the deviation in the query results which in about one third of all cases ranges from 2 to 9 instead of the expected 10.

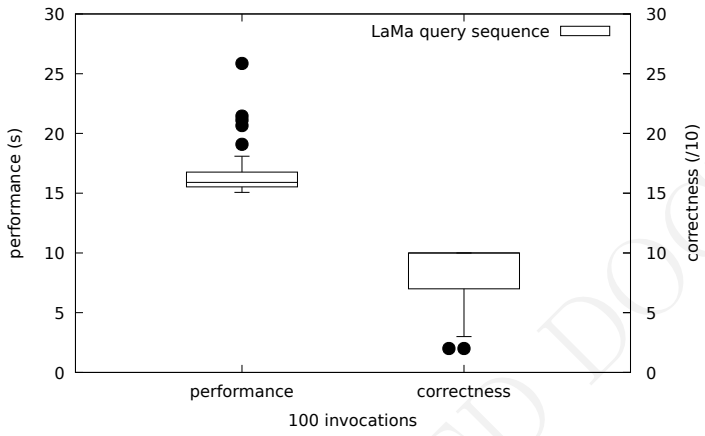

Figure 7: Benchmark for performance and correctness of LaMa

The limited correctness can be explained by another, more focused experiment which runs 200 rounds of random environment variable assignments, each followed by 10 consecutive reads. In total, 67 out of 2000 reads fail, about half of them in the first read, about one third in the second, and the remainder in the third and fourth. Five failures appear twice with or without a successful read in between. No error occurred after the 4th read attempt which coincides with a maximum error interval of 9.87s. We conjecture that the internal synchronisation of Lambda's configuration is set to 10s. Indeed, a second run of the same experiment with a 10s delay after the write yielded no read failures which helps application authors in making their applications less eventual and more consistent. Fig. 8 contains the consistency measurement results in a graph with clustered read failures.

The presence of eventual consistency and its boundary characteristics are thus far not documented for AWS Lambda. Other services of the same provider are described more completely, for instance EC2 whose control plane eventual consistency is mentioned, albeit without details about boundary conditions. We argue that expressive service descriptions would help to make such characteristics explicit. In the absence of a clear documentation, we assume that Lambda instances have lazy-synchronised configuration copies. Fig. 9 refines the introductory figure adapted to how AWS Lambda is known to work.

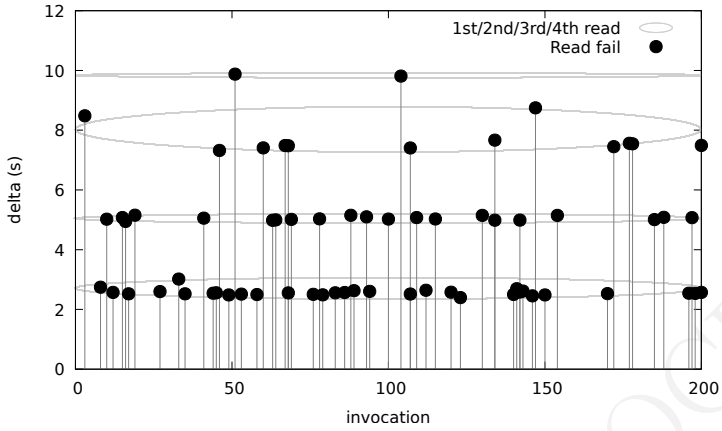

Figure 8: Eventual consistency results

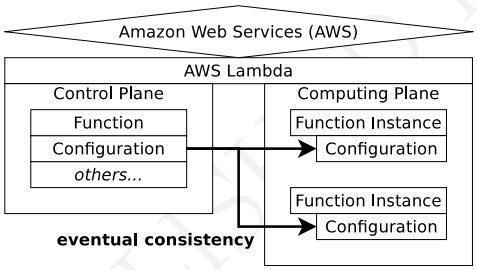

Figure 9: Lambda-specific computing and control planes

5 Discussion

It appears unusual to massage a cloud service's control plane into a useful service in itself upon which meaningful applications are built. Whether this becomes a sound technique or remains a singular proof of concept needs to be answered by future work. Concerning AWS Lambda, both the advantages (zero cost) and disadvantages (severe limits and hardly predictable eventual consistency) of the control interface are now palpable for future system designs.

Stateful Lambda functions can only be implemented with explicit access to the control plane. The service model is therefore not suitable for holding state per function in the general case or per function instance in any case. Among other effects, this limitation affects ongoing research work to transform object-oriented code to functional code which requires stateful instance attributes.

The requirements for future programmable platforms and infrastructure include better discoverability (through manuals and processable descriptions), better composability and programmability (refer to the non-trivial integration between AWS Lambda and API Gateway) and better procedures. A suitable

model would be working on a disconnected configuration which is model-checked and then enacted or rolled back during a single controllable transaction. The role of the control plane would then change considerably.

Repeatability

The implementations of Lambackup and LaMa as well as the scripts used in the referred experiments and the obtained reference results are made publicly available. We encourage the creative use of these artefacts for repeatability of the results and for confirmability of the presented findings. The corresponding repository is `http://github.com/serviceprototyping/...TODO`.

Acknowledgements

This research has been supported by an AWS in Education Research Grant which helped us to run our experiments on AWS Lambda as representative public commercial FaaS.

References

[1] D. Bernstein. Is Amazon Becoming the New Cool Software Company for Developers? *IEEE Cloud Computing*, 2(1):69–71, 2015.

[2] S. Butt, V. Ganapathy, and A. Srivastava. On the Control Plane of a Self-service Cloud Platform. In *ACM Symposium on Cloud Computing (SoCC)*, pages 10:1–10:13, Seattle, Washington, USA, November 2014.

[3] A. Ciuffoletti. Application level interface for a cloud monitoring service. *Computer Standards & Interfaces*, 46:15–22, 2016.

[4] J. Spillner, M. Beck, A. Schill, and T. M. Bohnert. Stealth Databases: Ensuring User-Controlled Queries in Untrusted Cloud Environments. In *8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 261–270, Limassol, Cyprus, December 2015.

[5] K. Thimmaraju, B. Shastry, T. Fiebig, F. Hetzelt, J.-P. Seifert, A. Feldmann, and S. Schmid. Reigns to the Cloud: Compromising Cloud Systems via the Data Plane. arχiv CoRR abs/1610.08717, October 2016.

[6] M. Villamizar, O. Garces, L. Ochoa, H. E. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, and M. Lang. Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *CCGrid*, pages 179–182, 2016.