Costradamus: A Cost-Tracing System for Cloud-Based Software Services

Jörn Kuhlenkamp[1,2] and Markus Klems[1,2(✉)]

[1] Technische Universität Berlin, Berlin, Germany
jk@ise.tu-berlin.de
[2] Information Systems Engineering Research Group, Berlin, Germany
mk@ise.tu-berlin.de

Abstract. Cloud providers offer a range of fully managed infrastructure services that enable a "serverless" architecture and development paradigm. Following this paradigm, software services can be built on compositions of cloud infrastructure services that offer fine-granular pay-per-use pricing models. While this development and deployment approach simplifies service development and management, it remains an open challenge to make use of fine-granular pricing models for improving cost transparency and reducing cost of service operations. As a solution, we present Costradamus, a cost-tracing system that implements a generic cost model and three different tracing approaches. With Costradamus, we can derive cost and performance information per API operation. We evaluate our approach and system in a smart grid context and discuss unexpected performance and deployment cost tradeoffs.

Keywords: Tracing · Cloud computing · Deployment costs · Performance

1 Introduction

Serverless computing [14] is an emerging architecture and development paradigm for building cloud-based software services that promises to reduce cost of service development and operations. A serverless service relies entirely on fully managed cloud infrastructure services that offer fine-granular pay-per-use pricing models.

Despite these detailed usage and pricing models, actual capacity usage and billing information is usually presented to users as aggregates, in terms of time (e.g., monthly bills) and resource usage (e.g., per infrastructure service category). This makes it difficult for software service developers to determine the actual capacity usage and associated cost of a single software service and to obtain a cost breakdown per API operation of a single service.

In this paper, we propose an approach and system prototype that solves this problem by enabling **per-request cost-tracing**. Potential applications of our approach are:

1. Cost-debugging tools [7] for developers who thereby gain insight into cost changes that are caused by small source code or deployment changes.

2. Systems for improving cost awareness across different teams, in particular, if cross-functional teams work independently on their own microservices. With a cost-tracing system, teams could communicate their service cost to other teams more easily.
3. Software-as-a-Service providers could calculate marginal cost of operations and, based on detailed cost information, design pay-per-use pricing models for their software services that guarantee a stable profit margin.

Our approach enables per-request cost-tracing by using a mix of analytic and experiment-based techniques. We use an analytic cost modeling approach and supply the cost model with input data that is derived through a tracing system which augments each request trace with resource capacity consumption data along its invocation path.

In the following Sect. 2, we give a short overview of serverless computing and distributed tracing. Then, in Sect. 3, we introduce the scenario of a smart grid metering application that is realized with serverless infrastructure. Section 4 shows our first contribution, a generic cost model for serverless infrastructure. In Sect. 5, we present our second contribution, Costradamus, a cost-tracing system that can determine the per-request capacity usage and cost of infrastructure service compositions. In Sect. 6, we present results of our cost measurements and discuss interesting effects that we observed when applying our cost-tracing approach to the smart grid metering application.

2 Background

Serverless computing is a paradigm that introduces a new system architecture approach as well as a new programming, runtime, and deployment model. Serverless architecture is characterized by the extensive use of fully managed cloud services and the absence of self-managed system components, in particular the absence of self-managed servers. Serverless computing is characterized by the use of Serverless Functions (SF), also known as Function-as-a-Service, Cloud Functions, or Serverless Microservices. Examples of SF services include AWS Lambda, Google Cloud Functions, and Azure Functions.

An SF encapsulates business logic and exposes an RPC handler interface for remote procedure calls. Typically, an SF is stateless and can be invoked through events, such as HTTP request events or events by other cloud infrastructure services. The SF lifecycle typically looks like this: a developer bundles the SF business logic source code and uploads it to a storage service. When the SF is invoked, the source code is loaded and executed in a managed, container-based runtime environment. The lifespan of an SF is relatively short, often below one second. For performance-optimization, SF containers are typically re-used for subsequent invocations and only destroyed if no new events have arrived for a prolonged period of time (several minutes), thereby saving infrastructure cost on the provider side.

Although SF deliver on the promise of low operational effort (NoOps), a service that is composed of many small functions creates new management challenges. For this purpose, distributed tracing and debugging solutions are needed, such as Google's Dapper [17]. AWS X-Ray is a similar tracing service that can be used to debug and analyze service compositions that comprise AWS Lambda functions and other AWS infrastructure services. For a sample of requests that clients send to an API Gateway, load balancer, or Lambda function, X-Ray adds a trace id to the request context and then passes the request to the destined service. A trace segment (also known as trace record or span) for the traced service is sent to an X-Ray daemon which buffers segments and uploads batches of segments to the X-Ray API. Downstream service incovactions, such as AWS DynamoDB or AWS Kinesis, can be traced by instrumenting the AWS SDK client that makes the request, e.g., from an EC2 instance or a Lambda function.

3   Application Scenario: Smart Grid Metering

We consider an application scenario in the context of smart grid management. The application scenario is inspired by the PolyEnergyNet project[1]. Continuously, the power grid becomes increasingly dynamic and decentralized in nature. To make timely and knowledgeable decisions for strategic grid expansion and day-to-day grid operation, it becomes increasingly important to meter the state of smart grids with fine granularity. This includes meters in the infrastructure of distributed network operators (DNO) and meters at individual consumers and prosumers.
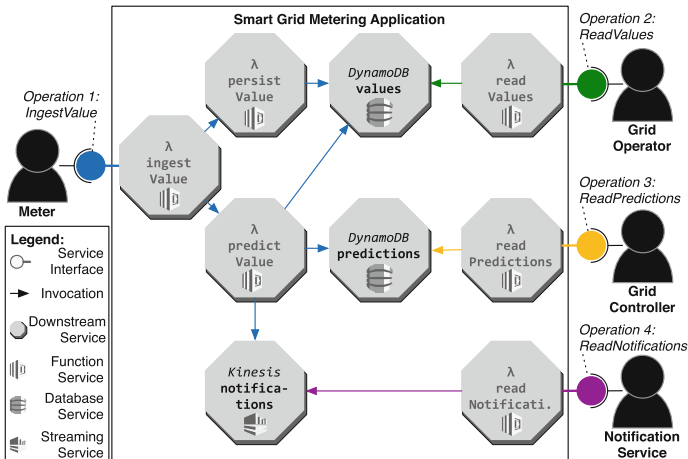

Fig. 1. Application scenario: smart grid metering application.

---

[1] http://www.polyenergynet.de.

A software service provider within the organization of a DNO offers the Smart Grid Metering application (SGMApp) (see Fig. 1). The SGMApp exposes an API comprising four operations that are each backed by cloud services. A request to an API operation triggers a cascade of invocations to downstream services in the back-end. Meters use the `IngestValue` operation to periodically push new metered values. The `ingestValue` service checks parameters and forwards values to `persistValue` and `predictValue`. The `persistValue` service stores values in a database for later analysis. The `predictValue` service uses historical data of the corresponding time series to calculate predictions. Historical data is retrieved from a local cache or from the `values` database service. Predictions are stored in a dedicated database service. If a predicted value significantly deviates from a current value, a notification is sent to a streaming service. Grid operators use the `ReadValues` operation to display the current state of the smart grid. The `readValues` service checks query parameters and retrieves values from `values`. Similarly, automated grid controllers use in addition the `ReadPredictions` operation. The `ReadNotifications` operation allows to consume critical notifications.

4   Software Service Cost Model

In this section, we present a generic cost model for cloud-based software services, and propose metrics to quantify cost and capacity waste per API request.

4.1   Service Model

An *API request R* cascades through a set of downstream services. Therefore, we model R as a set of *invocations $I \in R$* of downstream services. Each invocation consumes capacity of the corresponding downstream service. Precisely, a single invocation I consumes capacity of a set of *capacity types $t \in I$*. We denote the *measured consumption* of a capacity type $t \in I$ by u_t. Capacity for each capacity type is provisioned in provisioning units with a *provisioning unit size c_t*, and billed with a *provisioning unit price p_t*, respectively. However, a cloud provider meters consumption for each capacity type in full coarse-grained metering units with a *metering unit size m_t*, $m_t \leq c_t$. Provisioning units, metering units, and measured consumption are each specified as a tuple of amount and duration. We refer to the amount by the superscript 0 and to the duration by the superscript 1. Figure 2a illustrates different parameters used to model consumption of a single capacity type of an downstream service.

As an example, we model an invocation of a service function implemented on top of AWS Lambda. The service function uses a single capacity type `mem` denoting memory time. At the creation time of the service function, a service provider configures the service function to use provisioning units of $c_{mem} = (128\,\mathrm{MB}, 100\,\mathrm{ms})$. The provisioning unit price of a single provisioning unit is $p_t = 208\,\mathrm{n\$}$. Provisioning units equal metering units $c_{mem} = m_{mem}$. An example invocation utilizes a constant amount of $60\,\mathrm{MB}$ memory over a total runtime of
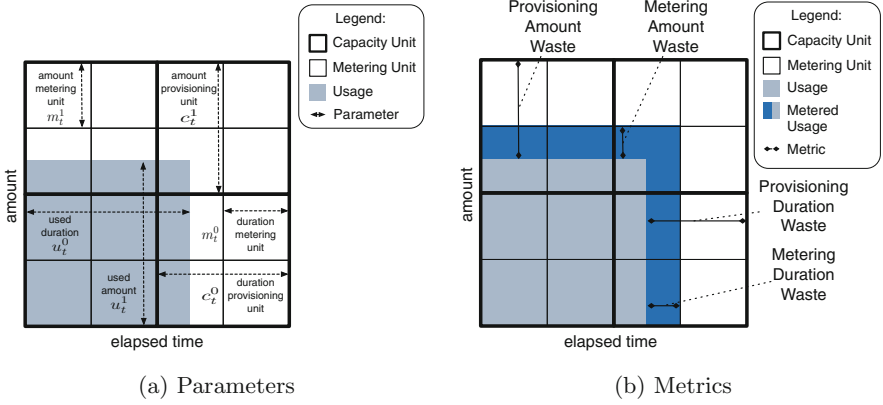
(a) Parameters   (b) Metrics

Fig. 2. Cost model parameters and metrics for a single capacity type.

$600\,\text{ms}$ $u_{mem} = (60\,\text{MB}, 150\,\text{ms})$. Optionally, an invocation that includes data transfer to or from a data center requires modelling of a second capacity type `bw` denoting network bandwidth.

4.2 Metrics

Based on the service model in Sect. 4.1, we define metrics to quantify per-request cost. On a more fine granular level, waste metrics characterize individual invocations of downstream services. Precisely, waste metrics characterize provisioned and not utilized capacity for invocations. Waste metrics serve the main purpose of supporting a service provider in cost debugging and optimization of software services. Figure 2 illustrates different metrics.

Marginal Request Cost. One of the main motivations behind Costradamus is the quantification of per-request cost. The Marginal Request Cost (MRC) metric (Eq. 1) does exactly that. For a request R, we add costs over all invocations $I \in R$. For each invocation, we add costs over all capacity types $t \in I$. To derive cost per capacity type t, we calculate metered amount and metered duration based on measured amount u_t^0 and measured duration u_t^1 per capacity type. Finally, we obtain MRC by calculating the relative share of provisioning units multiplied by the provisioning unit price p_t.

$$MRC(R) = \sum_{I \in R} \sum_{t \in I} \left\lceil \frac{u_t^0}{m_t^0} \right\rceil * \frac{m_t^0}{c_t^0} * \left\lceil \frac{u_t^1}{m_t^1} \right\rceil * \frac{m_t^1}{c_t^1} * p_t \tag{1}$$

Metering Duration Waste. Cloud providers typically measure how much and for how long a certain capacity is used by an invocation. However, for metering purposes, measured usage is usually rounded up to coarse-grained units. Metering Duration Waste (MDW) (Eq. 2) describes the difference between metered

and measured duration for the consumption of a capacity type. In our example, MDW quantifies the 50 ms of runtime that is metered for the invocation.

$$MDW(t) = m_t^0 - (u_t^0 \mod m_t^0) \tag{2}$$

Metering Amount Waste. Similar to MDW, Metering Amount Waste (MAW) (Eq. 3) describes the difference between metered and measured consumed amount of a capacity type. In our example, MAW quantifies the 68MB memory that is metered and never used by the invocation.

$$MAW(t) = m_t^1 - (u_t^1 \mod m_t^1) \tag{3}$$

Provisioning Duration Waste. A downstream service can provision per-invocation capacity or shared capacity for multiple invocations. Provisioning waste metrics characterize provisioned and unused capacity for an invocation in the absence of other invocations. Therefore, Provisioning Duration Waste (PDW) (Eq. 4) describes the difference between provisioned and metered usage duration for a capacity type. In our example, PDW equals 0 ms.

$$PDW(t) = c_t^0 - (u_t^0 \mod c_t^0) \tag{4}$$

Provisioning Amount Waste. Similar to PDW, Metering Amount Waste (PAW) (Eq. 5) describes the difference between provisioned and metered consumed amount of a capacity type.

$$PAW(t) = c_t^1 - (u_t^1 \mod c_t^1) \tag{5}$$

5   Cost-Tracing System

In this section, we present Costradamus, our end-to-end cost-tracing system for software services. In analogy to performance-tracing [15], we define cost-tracing as the activity of collecting detailed cost information of causally-related events in a distributed service-oriented system.

An application consists of multiple services that expose operations through an API. A tracing system collects data that relates to operation calls, including all downstream service invocations. In the next section, we identify design goals for a **cost-tracing** system.

5.1   Design Goals

Costradamus enables users to retrieve performance and cost information. In more detail, we propose and motivate the following design goals for our cost-tracing system.

(D1) **Per-request tracing.** The tracing system should provide fine-granular cost information for individual API operations, such as a single HTTP request.

(D2) **Cost composition.** Operations might invoke complex compositions of services with heterogeneous pricing models. A cost trace should cover the entire service composition and provide measurements in a normalized cost metric.

(D3) **Non-intrusiveness.** Making an application traceable should not have negative side-effects on other non-functional properties, such as availability, reliability, performance, and security.

Our first design goal (D1) is motivated by agile software development methodologies and DevOps best practices which advocate short and continuous cycles in which small software changes are pushed from development to production. With per-request cost traces, small software changes can be evaluated in isolation. Thereby, a developer can inspect performance and cost of a new feature or compare the performance and cost change that accompanies a feature change.

Non-trivial applications consist of many services with heterogeneous pricing models which can be invoked through non-deterministic events, motivating our second design goal (D2). Each trace should contain performance and cost information that allow users to drill down into the cost of all service invocations that are causally related to an API operation.

Our third design goal (D3) relates to general design goals of low performance overhead and application-level transparency [18]. Trace records can either be explicitly or implicitly related to a specific request. An explicit approach adds a reference to a specific entry event, e.g., the entry event id, to each trace record at runtime. An implicit approach assigns trace records to an operation offline and based on statistical correlation. The explicit approach simplifies (D2), however, applications must be instrumented to obtain a trace id reference, with potential negative effects on (D3). We favor (D2) over (D3) and use instrumentation points that can be disabled for production workloads.

5.2 Capacity Usage Tracing Approaches

Costradamus supports three tracing approaches for collecting capacity usage data (T1–T3). We discuss each tracing approach in the context of an example application as illustrated in Fig. 3. The example shows a software service that is composed of four infrastructure services: two function services, a messaging service and a database service. Each function service integrates an instrumentation point for capturing performance and cost data of each infrastructure service invocation. Part of the tracing system is a trace record store (which is a specialized *Message Store* [8]) that persistently stores trace records for later analysis. Whenever an infrastructure service is invoked, an instrumentation point produces a trace record and sends it to the trace record store. In the following, we describe three tracing approaches that take into consideration different types of infrastructure services.
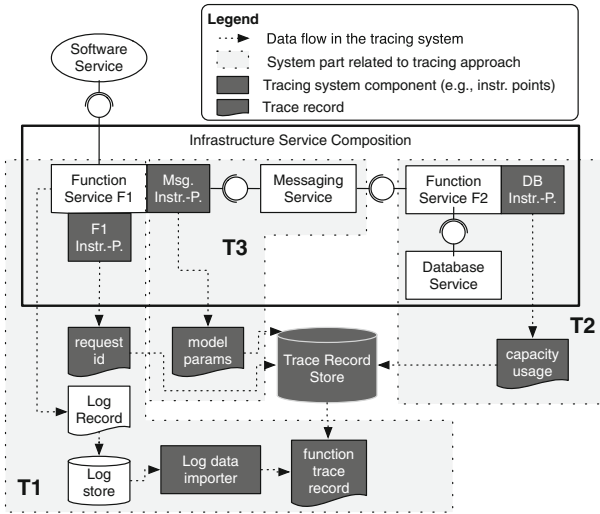
Fig. 3. Example setup of Costradamus showing tracing approaches T1–T3.

T1: Log Import. Tracing approach T1 constructs a capacity usage trace record from data in a (distributed) log store by sending a reference to a log entry to the trace store and later use the reference to query the log store. The example in Fig. 3 shows a function service F1 that writes logs to a log store. A log contains capacity usage and performance information, such as the start and end time of the function invocation, used memory, and billed duration. The log entry for each request in the log store can be identified by a unique request id. The instrumentation point of F1 extracts the request id from the function invocation context and adds it to the meta-data of F1's trace record. Thereby, a log data importer can be used for augmenting function service trace records during the trace collection phase. First, the trace record for F1 is retrieved and the request id is extracted from the trace record. Then, the log store is queried using the request id, and capacity usage information, such as memory and billed duration, are retrieved.

T2: Response Recording. Tracing approach T2 requires capacity usage information from a service invocation response message to construct a trace record. In the example shown in Fig. 3, function service F2 invokes another infrastructure service via a remote procedure call or API request. In the example, F2 invokes a database service. Approach T2 relies on information delivered in the response message of the invoked infrastructure service. For example, the database service AWS DynamoDB returns capacity usage information in provider- and service-specific capacity units. The capacity usage information is extracted from the response and added as meta-data to the trace record that is associated with the database service invocation. For simplifying Fig. 3, we do not show the

instrumentation point for tracing function service invoications of F2, which is performed as shown for F1 using the tracing approach T1.

T3: Modeling. Tracing approach T3 creates trace records at runtime with estimated capacity usage values that are modeled by Costradamus based on runtime measurements. This approach can be used if T1 is infeasible because there are no logs or if T2 is infeasible because the service does not send usage data in a response message. Similar to T2, T3 augments a trace record with meta-data. However, instead of capacity usage data from the service response, the trace record meta-data contains service request parameters that can be used for offline capacity usage estimation.

5.3   Prototype

We have implemented Costradamus, a cost-tracing system for AWS cloud infrastructure with Node.js based Lambda functions. The project is available as open source software [1]. For using Costradamus, instrumentation points must be added to the Lambda functions of the software service that should be traced. Furthermore, Amazon's distributed tracing service X-Ray must be activated for these Lambda functions.

Instrumentation Points. Costradamus uses special-purpose instrumentation points to add capacity consumption meta-data to trace records in X-Ray. For adding these instrumentation points, the developer needs to add the costradamus software library as a dependency to the Lambda function source code.

During the execution of a Lambda function and invocation of downstream Lambda, DynamoDB, and Kinesis services from within the function's business logic, we need to capture capacity consumption information that is not included in the plain X-Ray trace records. This is realized by adding a Costradamus subsegment to each parent segment (Lambda, DynamoDB, or Kinesis service invocation). Each Costradamus subsegment contains meta-data that is needed according to the respective tracing approach (T1, T2, or T3).

The instrumentation points require between 1–3 additional lines of code in the Lambda function source code, for each downstream service invocation, and 2 lines for making the Lambda function itself cost-traceable. The code for implementing an instrumentation point is between ca. 10–60 lines of code and should not be much larger for other infrastructure services, besides Lambda, DynamoDB, and Kinesis.

Cost-Tracing Process. After a client invokes a Lambda function that is activated for tracing with X-Ray, the client receives the trace id in the HTTP response header. This trace id is used in a next step to retrieve the corresponding trace record, consisting of segments and subsegments with performance data and some meta-data, from X-Ray (our trace record store). As described in the previous section, for tracing approach T1, in addition to the trace records stored in X-Ray, we retrieve Lambda function logs from CloudWatch. Each Lambda function segment contains a Costradamus meta-data field with the Lambda request

id which we extract and use to query the CloudWatch logs within a specific time window, between the start and end times of the Lambda function invocation. Since the logs in CloudWatch materialize with a longer delay than the X-Ray trace records, according to our observations, this operation might need to be repeated several times. The other trace records of DynamoDB and Kinesis invocations already contain all required capacity usage information as Costradamus meta-data.

In the next step, the X-Ray traces are augmented with cost meta-data which is generated on the client by using capacity meta-data as input for the cost modeler. In a further step, the Costradamus consumption subsegments are removed (pruned) from the trace record as they are not needed any more. Each of the three processing steps results in a new file that contains the trace record, so that, after processing, one trace record maps to three trace record files: plain, augmented, and pruned. We use a batch script to process multiple traces with a single command. The pruned files from the last processing step are used as input by a helper tool that creates a CSV file for performance and cost data analysis.

6   Evaluation

We investigate performance/cost tradeoffs of the SGMApp in four experiments.

6.1   Experiment Setup

Implementation. We implement the SGMApp (Sect. 3) with AWS. We use DynamoDB tables for the `values` and `predictions` services and Kinesis streams for the `notifications` service. All other downstream services are implemented as AWS Lambda functions. Operations are published via the Amazon API Gateway service. Invocations of service functions for the `IngestValue` operation are event-based, other invocations are request-response-based. All service functions parallelize invocations of tables and streams. We use a 10 s timeout for all service functions, and exponential backoff as retry strategy.

Workload and Measurements. For each experiment, we run a *Load* phase followed by a *Run* phase. The Load phase writes 600 historical values per meter to the `ValuesTable`. The Run phase issues 100 requests to each of the four API operations with a 1 s wait time between subsequent requests. For brevity, we refer to the operations by O1 (`IngestValue`), O2 (`ReadValues`), O3 (`ReadPredictions`), and O4 (`ReadNotifications`).

Metrics. For each request, we record a trace with segments and subsegments. Figure 4 shows an excerpt of a trace for the `IngestValue` operation. We measure all metrics presented in Sect. 4. In addition, we measure request-execution latency (REL) and invocation-execution latency (IEL) for each invocation of a downstream service. A relation exists between REL and IEL. For a single request, the REL is equal or larger than the sum of all corresponding IELs. We plot traces sorted by MC in ascending order.
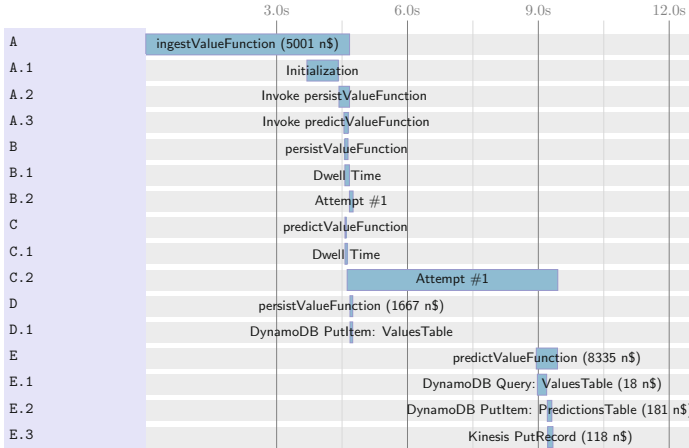
Fig. 4. Excerpt of an ingestValue operation trace with trace segments (for readability named A–E) and subsegments (A.1, A.2, etc.).

Case Study and Experiment Setups. We conduct experiments with four different experiment setups E1–E4 (see Table 1) to quantify performance and costs per API request. E1 serves as a baseline scenario. We compare the other experiments with E1 to investigate the impact of changes to the provisioned infrastructure (E2), business logic (E3), and target data center (E4). Precisely, we use three parameters: provisioned lambda memory in MB [128, 1024], the number of historical meter values and prediction horizon used by the predictValues function [3, 60] and the AWS region [us-east-1, eu-west-1].

6.2 Results and Discussion

E1. Figure 5 illustrates the results for E1. For O2, most traces are subject to a constant cost of 1.8 µ\$, this is due to a IEL < 100 ms for readValues and a constant number of meter values that are queried from values. The measurements show increased costs for trace ids >#87 due to an increased 100 < IEL < 400 ms of readValues resulting in a stepwise increase of MRC up to 6.8 µ\$. However, the increased IEL should result in a steeper cost increase. Detailed analysis of

Table 1. Summary of parameters for experiment setups E1–E4.

Experiment	Region	Memory [MB]	Interval [s]	Scenario
E1	us-east-1	1024	3	Baseline
E2	us-east-1	128	3	Infrastructure sizing
E3	us-east-1	1024	60	Business logic refinement
E4	eu-west-1	1024	3	Multi-region role-out

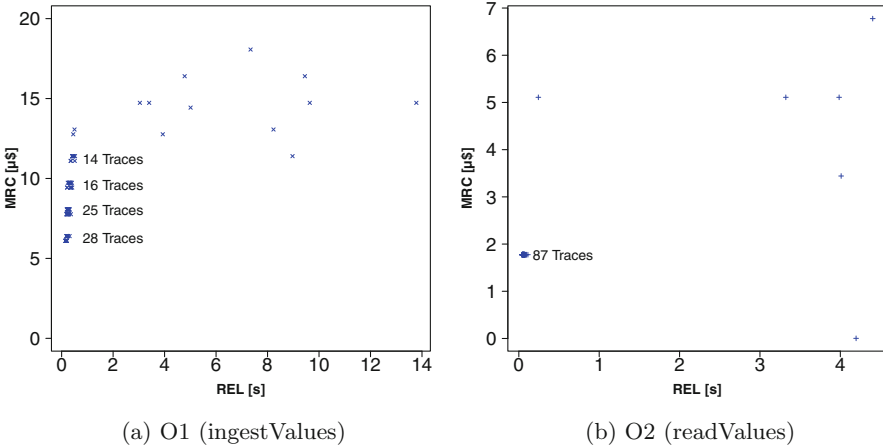(a) O1 (ingestValues)   (b) O2 (readValues)

Fig. 5. Cost and performance comparison of O1 and O2 for E1 (Baseline).

the corresponding traces reveals that this behavior is caused by failed executions of `readValues` that are not charged by the cloud provider but increase REL. For O1, we observe a higher variation of costs compared to O2. This behavior is caused by a variable runtime of service functions and variable size of data that is queried from `values` and written to `notifications`.

For O1 and O2, around 10% of the traces show a REL > 600 ms and, therefore, can be considered performance outliers. However, compared to O2, we observe that O1 performance outliers show an up to three times higher REL. Detailed analysis shows that the increased REL is caused by failed executions of multiple service functions for the same request. Furthermore, all performance outliers are scheduled in the beginning of a workload. A consistent explanation is a startup time for new containers that back a service function [7].

E2. In comparison to E1, E2 provisions only 128MB memory for containers that back service functions. Therefore, we expect REL to increase and MRC to decrease. We exclude performance outliers and summarize our results in Table 2. For O1, we observe a 200% increase in 95th-percentile REL and a 58% decrease in median MRC. Thus, the results indicate that the REL is bound to the IEL of the three service functions and, therefore, reducing provisioned memory results in a significantly lower performance. For O3, results indicate that REL is not bound to the `readPredictions` and reduced memory does not result in lower performance. Therefore, our experiment suggests that informed decisions on infrastructure sizing can help to identify new deployments that are strictly dominating in terms of performance and deployment costs.

E3. For E3, we change the implementation of `predictValues`. Precisely, the data resolution for the prediction is increased from 3 to 60 m values. Therefore, we expect additional (i) reads on `values` and (ii) writes on `predictions` and `notifications`. Figure 6 compares results for O1. We observe an unexpected
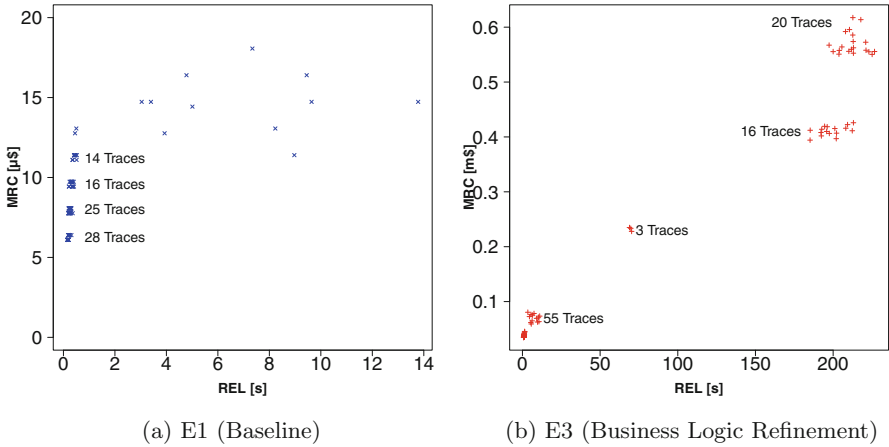
(a) E1 (Baseline)

(b) E3 (Business Logic Refinement)

Fig. 6. Cost and Performance Comparison of O1 (`ingestValue`) for E1 and E3.

significant increase in the 95th-percentile REL by ∼466 times and median MRC by ∼7 times, respectively. We further investigate this behavior, and find that invocations of `predictValues'` require up to three attempts to succeed due to a timeout of 10s. While failed attempts do not increase usage of Lambda, usage increases for invocations of `values`, `predictions` and `notifications` during failed attempts. Therefore, failed executions of service functions can significantly increase MRC. We refer to this effect as *retry cost effect*. Besides the retry cost effect, we observe that `predictValues'` issues more invocations of `values` and `predictions`. Therefore, `predictValues'` results in increases consumption of other downstream services. We refer to this effect as *ripple cost effect*. One implication of the ripple cost effect is that cost testing in an iterative development process should not only rely on isolated tests of single downstream services but also incorporate end-to-end cost testing.

E4. We compare the `us-east-1` and `eu-west-1` regions. Increased prices apply to the `eu-west-1` region for DynamoDB and Kinesis. Thus, we expect an increase MRC under similar REL in comparison to E1. Counterintuitively, we observe lower MRC and REL for the `eu-west-1` region (Table 2). Higher region prices are accompanied by better performance. Therefore, shorter runtimes and lower usage of service functions compensate for higher region prices. One implication of our findings is that cost calculations should not exclusively be based on analytical models but include real measurements.

Design Goals. Costradamus enables us to perform fine-granular per-request tracing (D1), as demonstrated with the experiment results above. We can also measure the cost of complex service compositions (D2), however, each downstream service must be instrumented, requiring the implementation of service-specific instrumentation points. The third design goal of non-intrusiveness (D3) is less prioritized and therefore also not evaluated comprehensively, as we propose

Table 2. Comparison of experiment setups E1/E2 and E1/E4.

Experiment	Metric	O1	O2	O3	O4
E1	95th-prec REL [ms]	452	84	231	303
	Median MC [n\$]	8063	1775	1685	5103
E2	95th-prec REL [ms]	1358	424	157	589
	Median MC [n\$]	3384	732	226	936
Δ E2, E1	95th-prec REL [ms]	+906(200%)	+340(403%)	−74(32%)	+285(94%)
	Median MC [n\$]	−4679(58%)	−1043(59%)	−1459(87%)	−4167(82%)
E4	95th-prec REL [ms]	250	101	78	271
	Median MC [n\$]	6242	1790	1687	5110
Δ E4, E1	95th-prec REL [ms]	−202(45%)	+17(21%)	−153(66%)	−33(11%)
	Median MC [n\$]	−1821(23%)	+15(1%)	2(0%)	7(0%)

to apply our tracing approach only during development and disable it during production. However, we observed low performance overhead when comparing client-side latency of requests with tracing toggled on/off. This can be explained by the fact that the Costradamus prototype builds on AWS X-Ray which runs as a separate daemon that sends data batches over UDP. A more comprehensive evaluation, in particular of security and availability implications, would be needed to use Costradamus in a production environment, and is a task for future work.

7 Related Work

A large number of tracing frameworks exist to model performance of distributed, server-based applications [3,5,11,18]. We extend this work by providing a cost-tracing system that addresses two unique challenges. First, tracing of consumptions for heterogeneous infrastructure services and capacity types. Second, tracing under highly restricted options to add instrumentation points due to the high abstraction of serverless infrastructure.

Per-request cost is determined by consumed resources in downstream services. Therefore, our work is related to existing research in the area of cloud resource management [9] from the perspective of a cloud user with a focus on application resource demand profiling and application pricing [16]. Existing approaches for resource demand profiling model resource consumption for a given workload to model performance [2,6,12,20], cost [19] or energy consumption [10] as a function of resource consumption. The work by [4] evaluates the tradeoff between profit and customer satisfaction for a traditional virtual machine based infrastructure setup in a compute cloud. In contrast, we do not assume traditional infrastructure services, e.g., virtual machines, but serverless infrastructure services that expose resources on a higher abstraction level, e.g., function, messaging, and database services.

Leitner *et al.* [13] provide closely related work by modeling overall costs of microservice-based cloud applications. In contrast, our work models marginal per-request costs and provides a cost-tracing system. Thereby, we can identify and study performance and cost effects in isolation and greater detail than related experiments on serverless microservice-based applications [7,21].

8   Conclusions

We present Costradamus, a cost-tracing system that enables per-request cost-tracing for cloud-based software services. Costradamus includes a generic cost model and three tracing approaches: log import, response recording, and model-based tracing. We use Costradamus to investigate performance and deployment cost tradeoffs in a smart grid application context. In our experiments, we observe unexpected effects. First, the retry cost effect: In the case of function service invocations that call downstream services, failed attempts to invoke the upstream function, e.g., due to a timeout, can lead to increased cost, even if the failed upstream function invocation itself is not charged. Second, the cost ripple effect: more invocations of an upstream function service can lead to a multiplication of downstream service invocations. These effects illustrate that cost testing should not only rely on isolated tests of single services but consider comprehensive end-to-end cost traces.

References

1. Costradamus. https://github.com/markusklems/costradamus. Accessed 15 Jun 2017
2. Björkqvist, M.: Resource management of replicated service systems provisioned in the cloud. Ph.D. thesis, Universita della Svizzera Italiana (2015)
3. Braun, B., Qin, H.: ddtrace: Rich performance monitoring in distributed systems. Technical report, Stanford University (2015)
4. Chen, J., Wang, C., Zhou, B.B., Sun, L., Lee, Y.C., Zomaya, A.Y.: Tradeoffs between profit and customer satisfaction for service provisioning in the cloud. In: Proceedings of the 20th International Symposium on High Performance Distributed Computing, pp. 229–238. ACM (2011)
5. Chen, M., Kiciman, E., Fratkin, E., Fox, A., Brewer, E.: Pinpoint: problem determination in large, dynamic Internet services. In: Proceedings of International Conference on Dependable Systems and Networks, pp. 595–604. IEEE Computer Society (2002)
6. Gandhi, A., Harchol-Balter, M., Raghunathan, R., Kozuch, M.A.: AutoScale: dynamic, robust capacity management for multi-tier data centers. ACM Trans. Comput. Syst. **30**(4), 1–26 (2012)

 7. Hendrickson, S., Sturdevant, S., Harter, T., Venkataramani, V., Arpaci-dusseau, A.C., Arpaci-dusseau, R.H.: Serverless computation with OpenLambda. In: USENIX Workshop on Hot Topics in Cloud Computing (2016)
 8. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman Publishing Co. Inc., Boston (2004)
 9. Jennings, B., Stadler, R.: Resource management in clouds: survey and research challenges. J. Netw. Syst. Manag. **23**(3), 567–619 (2015)
10. Kansal, A., Zhao, F., Liu, J., Kothari, N., Bhattacharya, A.A.: Virtual machine power metering and provisioning. In Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, p. 39. ACM Press, New York (2010)
11. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Bhoedjang, R.A.F.: MAGPIE. In: ACM SIGPLAN Notices, number 8, pp. 131–140. ACM, August 1999
12. Kuhlenkamp, J., Rudolph, K., Bermbach, D.: AISLE: assessment of provisioned service levels in public IaaS-based database systems. In: Barros, A., Grigori, D., Narendra, N.C., Dam, H.K. (eds.) ICSOC 2015. LNCS, vol. 9435, pp. 154–168. Springer, Heidelberg (2015). doi:10.1007/978-3-662-48616-0_10
13. Leitner, P., Cito, J., Stöckli, E.: Modelling and managing deployment costs of microservice-based cloud applications. In: Proceedings of of the 9th International Conference on Utility and Cloud Computing, pp. 165–174. ACM Press, New York (2016)
14. Roberts, M.: Serverless Architectures (2016)
15. Sambasivan, R.R., Fonseca, R., Shafer, I., Ganger, G.R.: So, you want to trace your distributed system? Key design insights from years of practical experience. Technical report, Parallel Data Laboratory, Carnegie Mellon University, Pittsburgh, PA 15213–3890 (2014)
16. Sharma, B., Thulasiram, R.K., Thulasiraman, P., Garg, S.K., Buyya, R.: Commodities, pricing cloud compute: a novel financial economic model. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID 2012), pp. 451–457. IEEE, May 2012
17. Sigelman, B.H., Andr, L., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Google Research, 14 April 2010
18. Sigelman, B.H., Barroso, L.A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., Shanbhag, C.: Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google (2010)
19. Smith, J.W., Khajeh-Hosseini, A., Ward, J.S., Sommerville, I.: CloudMonitor: profiling power usage. In: 2012 IEEE Fifth International Conference on Cloud Computing, pp. 947–948. IEEE, June 2012
20. Urgaonkar, B., Shenoy, P., Chandra, A., Goyal, P., Wood, T.: Agile dynamic provisioning of multi-tier Internet applications. ACM Trans. Autonom. Adapt. Syst. **3**(1), 1–39 (2008)
21. Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., et al.: Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In: 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 179–182. IEEE (2016)