Bricklayer: Resource Composition on the Spot Market

Walter Wong*, Lorenzo Corneo†, Aleksandr Zavodovski*, Pengyuan Zhou*, Nitinder Mohan§, Jussi Kangasharju*

*Department of Computer Science, University of Helsinki, Finland
Email: {walter.wong, aleksandr.zavodovski, pengyuan.zhou, jussi.kangasharju}@helsinki.fi
†Department of Information Technology, Uppsala University, Sweden
Email: lorenzo.corneo@it.uu.se
§Department of Informatics, Technical University Munich, Germany
Email: nitinder.mohan@tum.de

Abstract—AWS offers discounted transient virtual instances as a way to sell unused resources in their data-centers, and users can enjoy up to 90% discount as compared to the regular on-demand pricing. Despite the economic incentives to purchase these transient instances, they do not come with regular availability SLAs, meaning that they can be evicted at any moment. Hence, the user is responsible for managing the instance availability to meet the application requirements. In this paper, we present Bricklayer, a software tool that assists users to better use transient resources in the cloud, reducing costs for the same amount of resources, and increasing the overall instance availability. Bricklayer searches for possible combinations of smaller and cheaper instances to compose the requested amount of resources while deploying them into different spot markets to reduce the risk of eviction. We implemented and evaluated Bricklayer using 3 months of historical data from AWS and found out that it can reduce up 54% of the regular spot price and up to 95% compared to the standard on-demand pricing.

Index Terms—Cloud computing, spot instances, availability

I. INTRODUCTION

Cloud computing offers many benefits to users, such as flexible on-demand resource allocation and a pay-as-you-go pricing model. One of the main drivers for cloud computing adoption is the reduction of upfront capital investment on infrastructure (CAPEX) by leasing servers in the public cloud as the service grows or scaling servers in the cloud to handle seasonal peak workloads. Cloud providers offer a wide range of virtual server options and subscription models, for instance, reserved, on-demand, and transient instances (known as spot instances in AWS). These spot instances can be up to one order of magnitude cheaper compared to regular prices and, as such, many research works have been done to explore the possible benefits on them in different scenarios, from batch processing [14], [16], [22] to web-serving [11], [13].

Despite the attractiveness of the spot instances, they come with some risks attached, for example, the threat of eviction and the financial risk. Spot instances are sold as instantaneous spare capacity from a cloud provider; thus, they do not enjoy the same Service Level Agreement (SLA) on availability as regular on-demand instances. Cloud providers revoke spot instances to fulfill requests for on-demand servers, which have a higher price point compared to the spot instances. Hence, spot instances can be evicted with a 2-minute notice and, therefore, users need to be aware of this characteristic. The financial risk is due to the nature of the spot market, where prices are driven by supply and demand. Whenever a user bids for a spot instance, the user pays for the current market price of the spot instance rather than the bid price. Therefore, any variations on the price between the market price to the bid price is paid by the user. Our observation of historical spot market data of a 3-month period shows that, on average, the user ends up paying double the lowest market price over the period.

Many research has been done to optimize the use of spot instances, for example, better bidding strategies to get the best pricing and reduce the eviction risk [10], [12], [23], [25], increase application availability by proactively migrating applications between spot and on-demand instances [17], [18], [20] and also strategies to mix-and-match spot and on-demand instances to trade availability for some cost reduction [7]. However, one overlooked issue is the bulk eviction problem, where all spot instances of the same type are evicted simultaneously due to the same bid price. This occurs because many bidding strategies optimize for the best bidding price, and they use multiple instances of the same type and price, thus, whenever the market price goes over the bidding price, all instances are simultaneously evicted from the cloud provider. Another overlooked problem when using spot instances is the application running on top of them, which has different availability requirements and can be broadly divided into two main types: data-intensive applications such as big data and machine learning which can tolerate time delay and some failures [14], [16], [22], [24], and always-on interactive applications, e.g., e-commerce and social networks, which need to be always online.

In this paper, we present Bricklayer, a software tool to assist users in getting the best combination of spot instances that meets the application requirements on cost and availability in the cloud. Rather than receiving a specific spot instance type and finding out the best bidding strategy for it, Bricklayer receives the application resource requirements and constraints and looks for the best spot instance set that can fulfill those requirements, either at optimizing for overall cost or improved availability, allowing for horizontal scaling. Under the hood, Bricklayer checks what is the cheapest AWS EC2 Computing Units (ECU) price for a given virtual hardware family, e.g., CPU or GPU, calculate the price volatility and eviction rates for all instances available in the spot market and return the composed set of spot instances that fulfill the user’s requirements. The main benefit of breaking up
larger resources into smaller ones is that each individual
spot instance can be allocated to different spot markets, thus,
having different price variations and eviction rates as they
now belong to different spot markets. We implemented and
evaluated Bricklayer using 3 months of historical spot market
data from AWS and evaluated the spot instance composition
for the two main categories of application, namely, batch-type
and always-on applications. The experimental results show
that Bricklayer can achieve up to 54% discount over the
spot instance pricing and up to 95% discount as compared
to regular on-demand pricing on AWS while maintaining the
same level of availability.

The paper is divided as follows. Section II presents the
background information on AWS spot market. Section III
describes the design of Bricklayer. Section IV presents the
implementation and the main components of the tool. Section
V shows the evaluation scenarios and the experimental results.
Section VI presents the related work and compares them with
our work. Finally, Section VII concludes the work.

II. SPOT INSTANCES BACKGROUND

All major cloud providers offer transient instances as a
way to sell under-utilized resources to the customers, for in-
stance, Amazon AWS Spot Instances [3], Google preemptible
instances [9], and Azure low priority instances [8]. AWS Spot
model uses a dynamic pricing model while Google and Azure
use a fixed price model; thus, we will focus our model on the
AWS spot market in this paper.

A. Amazon EC2 Spot Market

AWS offers three subscription models for virtual instances:
reserved, on-demand, and spot instances. Reserved instances
provide up to 75% discount compared to the on-demand
price [2]. However, the subscriber is bound to a 1 to 3-year
contract (which may or may not be paid upfront). The second
option is on-demand, where the user pays for the resource
consumption (pay-as-you-go model). In both models, the
subscriber has an SLA with AWS where the latter guarantees
minimum availability for the virtual machines and also that
they are non-renovable, i.e., they cannot be preempted by the
cloud provider and have their resources taken back without
the subscriber’s consent. Lastly, the spot market is where
resources are offered with big discounts (up to 90% off)
comparing to the on-demand price but without the regular
SLA guarantees. Another drawback is the cloud provider can
reclaim the resources back after giving just a short notice to
the subscriber.

In order to get a spot instance, the customer places a bid
in the AWS Spot market. If the bid price is higher than the
current market price of the resource, the user will get the
instance and pay for the market price. Therefore, the user
places a bid with the maximum acceptable price to pay. Once
the bid is placed in the spot market, it cannot be changed, and
the bid will remain active until the spot instance is evicted
or terminated. In case the instance spot market price goes
above the bid price, the user will receive a 2-minute notice
about the virtual machine preemption, and then it will be
terminated. There are some works [21], [25] that explore the
best bidding strategy at the spot market, leveraging the spot
pricing distribution, and choosing the least volatile one. AWS
recommends that customers bid for the on-demand price to
reduce the chance of eviction [4].

One complexity in the spot market is the number of
available options to choose from. The spot market is divided
into multiple regions (21 regions), availability zones, instance
families and types, resulting in more than ~6k separate spot
markets, where each instance type in a different availability
zone has a different spot pricing. Therefore, the same instance
in the same region but deployed in a different availability zone
may have different prices. In addition to that, spot instances
are also divided into several instance families, e.g., general-
purpose (A1, T1) and within each family, their sizings (large,
x2.large, etc.).

Despite the attractive pricing for the spot instances, most of
the time, they cannot be used out-of-box for most applications.
Some instances can be preempted over 10 times a day,
affecting application performance and task completion times.
Another limitation is that the spot instances can be preempted
in groups, i.e., if the market price for an instance type goes
over a bid that requested a group of instances, they will be
preempted all at once.

B. Spot Market Analysis

In this section, we will analyze the main metrics that we
will use to drive the instance selection decision. For that, we
will use 3 months of historical data (March 28th to June 28th,
2019) and analyzed the behavior of three metrics: the EC2
computing unit (ECU) pricing, instance volatility regarding
price and the overall instance availability over time.

ECU is a metric provided by AWS to ease the comparison
between instance capacity through AWS internal benchmarks.
In our case, we will use it to calculate the ECU unit price (1
ECU) for each instance type and pick the cheapest one on the
list. Table I compares the ECU unit price for each instance
type in different instance families. The results show that ECU
unit price has a high variance within the same instance family
(up to 10 times) and between different instance families (13
times). Another interesting finding shown in Fig. 1 is that there
is no economy of scale on bigger machines (due to higher
ECU density) or price increase (as more powerful machines
would be more expensive). For that, there is a minimum ECU
price that is roughly the same for all machines (except for
FPGA and GPU instances). Note that those prices are dynamic

<table>
<thead>
<tr>
<th>Instance Family</th>
<th>Min. ECU (¢/h)</th>
<th>Max. ECU (¢/h)</th>
<th>Diff. (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute optimized</td>
<td>0.002600</td>
<td>0.008958</td>
<td>344.54</td>
</tr>
<tr>
<td>General purpose</td>
<td>0.003154</td>
<td>0.011854</td>
<td>325.81</td>
</tr>
<tr>
<td>Storage optimized</td>
<td>0.012476</td>
<td>0.025212</td>
<td>176.60</td>
</tr>
<tr>
<td>Memory optimized</td>
<td>0.003560</td>
<td>0.037692</td>
<td>1105.77</td>
</tr>
<tr>
<td>FPGA instances</td>
<td>0.019038</td>
<td>0.021064</td>
<td>11.06</td>
</tr>
<tr>
<td>GPU instances</td>
<td>0.022979</td>
<td>0.043650</td>
<td>189.95</td>
</tr>
</tbody>
</table>

TABLE I: ECU unit price vs. instance family on AWS.
and vary throughout the day, and our goal is to pick the cheapest option to minimize the cost on AWS.

Another important conclusion is that there is no difference between getting a bigger machine or a set of smaller machines to compose to a bigger machine. Although the cost is the same, selecting a set of smaller machines from different spot markets to compose to a bigger machine provides higher availability as it reduces the bulk evictions and also the individual eviction rate. Therefore, the main takeaways from the ECU unit price analysis are: (i) select the cheapest ECU unit price instance within the same hardware family, (ii) there is almost no difference between minimum ECU unit price between a small or a large instance on AWS, and (iii) a set of smaller instances are preferred to compose to a bigger instance as they provide higher availability due to different spot markets.

The second metric to be analyzed is the price volatility of each instance. The goal is to select the instances with lower price volatility, so the user doesn’t end paying much more than the spot base price. Fig. 2 shows the price volatility of the top 10 instances in the us-east-1 region. Some prices can hike up to 10x (e.g., m2.2xlarge) and also be the same as the on-demand pricing, affecting the overall discounted price that the user may have. Fig. 3 shows instances that did not have changes in the price over the 90 day period, and those are the instances that Bricklayer should preferentially select. Therefore, the main takeaway for the second metric analysis is that not all instances are equal in price volatility, and we should focus on those with lower prices and volatility (or higher durability).

The third metric is the instance eviction rate, i.e., the number of times an instance is shutdown by AWS. This metric is important to differentiate the instances with a higher eviction rate (we should avoid them) and the lower eviction rate. Fig. 4 shows the number of evictions for the top 10 most evicted instances. Some instances can be evicted up to 6 times per day, on average. Although some instances may have smaller number of evictions, they can actually present lower availability due to longer periods of downtime. Therefore, the key takeaway is to select instances with lower/lowest eviction rate from the AWS list.

III. SYSTEM DESIGN

The first step of Bricklayer design is to define which kind of applications are going to run on top of it. In one end of the supported application spectrum, we have the time-sensitive, always-on type of application. These always-on applications need minimum downtime as some of them may result in financial losses, e.g., e-commerce and social networks. For example, an outage of 5 minutes costs half-million to Google, and a 10-minute downtime on Amazon may cost 2 million [1], [5]. Therefore, the paramount of always-on applications running over spot instances is to minimize the eviction rate on spot instances.

On the other end of the application spectrum, we have the delay, fault-tolerant, and batch-type applications, where users can tolerate some delays in the job processing time. Some examples in this category include big data processing and machine learning workloads. These kinds of applications can trade some delay for possible reduced costs in the processing; thus, we want to minimize the overall cost of applications running over spot instances.

A. Metrics

In order to be able to compare and decide which instances are a better fit for each kind of application, we elected the following criteria for spot instance comparison:

ECU unit price. The ECU unit price is the baseline metric to compare the computing power between different instances within the same AWS family type (thus, share the same underlying hardware). It is calculated by dividing the instance's...
hourly price by the number of instance’s ECUs. Bricklayer aims to prioritize the selection of instances with the cheapest ECU unit price within the same AWS instance family.

**Price volatility.** Price volatility represents the amount of variation in the instance price over time, and it is calculated as the relative standard deviation of the instance price. Bricklayer aims to select instances with lower price volatility, which means a lower financial risk of overpaying and more price predictability.

**Availability.** Availability measures the instance uptime over a given period of time, discounting the number of times the instance has been evicted due to price eviction, i.e., spot instance price going over the on-demand pricing. Bricklayer will choose instances that have lower eviction rates due to price eviction.

With those 3 metrics, Bricklayer can optimize for always-on type of applications (maximize for availability) or batch-type of applications (minimize cost). Next, we will formalize the Bricklayer model, metrics, and the problem statement.

**B. Model**

Let be \( M = \{ m : m < N \land m \in \mathbb{N}_+ \} \) the set of all the available instances provided by AWS, where \( N \in \mathbb{N}_+ \) is their availability upper bound. Additionally, we model the set of the \( K \) spot markets as \( S = \{ s : s \in \mathbb{N}_+ \land s < K \} \). Consequently, we indicate the set of all the available instances in the spot market \( s \in S \) with \( M_s \), that is, \( M_s \subseteq M \) and therefore \( M = \bigcup_{s \in S} M_s \). We refer to the \( i_{th} \) virtual machine in set \( M \) with the notation \( M_t \). Additionally, \( \forall m \in M \) there exists a set of three functions that return, respectively, the instance type, the available number of ECUs and the price per hour. We define instance type function as \( t : M \to \mathbb{N} \) where the return type could be seen as an integer number that maps to a particular instance type defined by AWS, e.g., \( m2.xlarge \). The function \( e : M \to \mathbb{N} \) simply returns the number of ECUs available in a particular instance in \( M \). Finally, the hourly price, at time \( \tau \), of a spot instance is provided by the function \( p : (M, \tau) \to \mathbb{R}_+ \). For convenience, we define the set of all the instances of type \( i \) as \( T_i = \{ m : m \in M \land t(m) = i \} \) and the set of all active instances as \( A = \{ m : m \in M \land on(m) = 1 \} \) where \( on : M \to \{ 0, 1 \} \) a binary function returning 1 if the instance is active or 0 otherwise. To be noticed, we use the notation \( A_s \) to refer to the set of all the active instances in spot market \( s \) and, whenever not indicated, it is assumed the set of all active applications regardless of their spot market.

**ECU unit price.** The ECU unit price, \( x \), is a metric defined as the ratio between the hourly price of an instance \( m \in M \) and its number of ECU. To put it formally:

\[
x(m, \tau) = \frac{p(m, \tau)}{e(m)}
\]

**Price volatility.** We calculate the price volatility of \( x \) over time by means of its relative standard deviation, that is, the ratio between the standard deviation and the mean. We define the price variability, \( v \), of an instance, \( m \), as follows:

\[
v(m, \tau) = \frac{\sigma(P)}{\mu(P)}
\]

where \( P = \{ p(m, \tau) : \tau \in [\tau_{MIN}, \tau_{MAX}] \} \), and \( [\tau_{MIN}, \tau_{MAX}] \) is the time period under investigation subject to price variability calculation.

**Availability.** The life-cycle of an instance is dictated by the previously defined on function so that we can detect 1-to-0 transitions, meaning active-to-evicted, and increment the eviction rate. We introduce the falling edge detection function as \( \delta : (\{0,1\}) \to (\{0|1\}) \), taking as input the values returned by the on function and returns either 0 (evicted) or 1 (active). It is possible to define the eviction rate, \( d \), as:

\[
d = \sum_{\forall m \in A} \delta(on(m))
\]

**C. Problem statement**

We now formalize the problem statement for the always-on and batch-type category of applications. These two types have different objectives and, therefore, two different optimization problems.

**Batch-type applications.** This category of applications can tolerate some delays and, therefore, the objective is to reduce the ECU unit price of the overall system, \( x \), while still satisfying the requirement on the number of ECUs, \( ECU_{req} \).

\[
\text{minimize} \quad x(m, \tau) \quad \text{s.t.} \quad \sum_{\forall m \in A} e(m) = ECU_{req}
\]

**Always-on applications.** This category of applications have strict uptime requirements and, therefore, the objective is to maximize the overall system availability by selected as many as possible different instances from different spot markets.

\[
\text{maximize} \quad |\hat{S}| \quad \text{s.t.} \quad \hat{S} \subseteq S \quad A_s = \bigcup_{s \in \hat{S}} A_s \quad \sum_{\forall m \in A} e(m) = ECU_{req}
\]

The result of this maximization is the best set of active instances from different spot markets, \( \hat{A} \), delivering the best possible availability.

**IV. IMPLEMENTATION**

We implemented the core Bricklayer features in Python, as shown in Fig. 5. Bricklayer has four main components: (a) Bricklayer engine, responsible for providing the best set of spot instances based on the application type; (b) Bricklayer analytics, responsible for analyzing the spot market pricing; (c) the historical data parser, which fetches AWS spot pricing and stores in the database; and (d) the MongoDB which stores all pricing information and analytics data.

The Bricklayer Engine uses Google Operation Research Tools for the spot instance set combination and optimization based on the application type (always-on vs. batch-type) [6]. The Bricklayer Analytics uses Pandas framework [15] to analyze the historical data and perform the analytics over the metrics. The Historical Data Parser fetches the pricing data from AWS.
Bricklayer will recommend multiple instances of the same instance type, for instance, 4 instances of the `ml.xlarge`, if it finds that this is the cheapest price for the ECU over all spot instances, but this may result in lower availability compared to picking up different instance types to compose a resource type. The reason is that the price increase in a given instance type may lead to bulk evictions of all instances. The price optimized resource set is more suited for the fault-tolerant batch type of applications, which can re-run jobs in case of failure without a higher impact on the system. In case the application requires higher availability, Bricklayer allows for trading off cost for higher availability, which we will analyze next.

Fig. 7 shows the spot instance price variation vs. the number of different spot markets. By increasing the number of different spot markets, we reduce the risk of bulk eviction. The reason for that is that the chance of price increase for all virtual instances at a given time is lower compared to a single one, and the change decreases as the number of spot markets increases. For example, in the first figure, having one instance type (e.g., `ml.xlarge`) will yield the cheapest resource set (in this case, $0.155/h). By having two instance types (or instances in two different spot markets), the price increases to $0.19/h for the same amount of processing power. Therefore, users can pay a premium in order to have a resource set with higher availability. On average, choosing more distinct instance types have an increase of 24% in the first additional distinct instance, and further distinct instances add up roughly 3%. The reason is that Bricklayer is not selecting the cheapest instance type available for the set, but adding the second, third and so forth cheapest instances, resulting in a higher composed price.

Fig. 8 shows the total price of the composed resource set over the 90 day period vs. the number of distinct spot markets (or instances types) and the number of required instances. For this evaluation, Bricklayer only selected the spot instances with the lowest volatility, and eviction rate; thus, all selected instance types in this evaluation have not been evicted during the 3 month period due to price increase. In Fig. 8(a), we compared the total price between regular off-the-shelf standalone spot instances (`r4.4xlarge`, `m5.4xlarge`, and `r5.4xlarge`) with the Bricklayer composite set, shown as opt-<number of distinct instance types>. For instance, if the user wants to get the cheapest price option from Bricklayer, she will use 8 instances of the same type (opt-1 model). However, if she wants to reduce the change of bulk eviction, she can choose instances from different spot markets, which results in higher prices. However, the number of instance may reduce as Bricklayer will get bigger instances to fulfill the requested resource requirements.

VI. RELATED WORK

HotSpot [18] uses containers within spot instances to migrate the applications between different instances within the same availability zone. It periodically computes the lowest spot price in the market and proactively migrates to a new instance to avoid forced preemption. The main benefit of
Fig. 6: Cost comparison between the naïve selection of spot instances vs. optimized spot instance selection.

Fig. 7: Cost comparison between number of different spot markets.

Fig. 8: Comparison between total cost over period of 3 months vs. number of distinct spot markets and number of instances of the same spec. All instances have availability of 100% as they have not been evicted by price.

HotSpot is it doesn’t require any changes to the application inside the container. Compared to Bricklayer, the latter spreads the eviction risk among a set of spot instances, which can be located in different spot markets. With that, the user can reduce even more the risk of eviction compared to the single instance model in HotSpot.

SpotOn [20] is a batch computing service running on top of spot instances, allowing for automatic selection of spot instances and implementing fault tolerance mechanism to mitigate data loss without modifying the application. SpotOn uses containers to encapsulate jobs with their dependencies, and it may use either reactive or proactive container state checkpointing in the disk. In the latter case, the batch computing engine periodically checkpoints the job information in the disk, and, in case of an instance termination, SpotOn can start a container migration to a new instance or deploy a new spot instance and use the last saved checkpoint data. Although Bricklayer does not implement the checkpointing mechanism available on SpotOn, Bricklayer can leverage such mechanism to save intermediate application state during runtime and restoring it in case of failure, resulting in reduced processing time.

SpotCheck [17] describes a derivative cloud market where SpotCheck purchases cloud resources from the big cloud
providers and resell it with customizations to smaller customers. The contribution is to provide with different solutions and specialized customizations to support customers specific use-cases that are not supported by the original cloud provider. SpotCheck allows running customers’ applications within nested VMs inside spot servers whenever possible and transparently migrate to on-demand servers whenever the spot server is revoked. In the similar nested virtualization solution, SuperCloud [19] is a cloud architecture running over OpenStack that integrates multiple cloud providers and allows for live application migration across those providers using Xen virtualization. Supercloud uses nested virtualization to enable the complete VM migration from one server to another, thus allowing users to relocate virtual machines from one cloud provider to another without disrupting the running application.

VII. CONCLUSION

In this paper, we presented Bricklayer, a resource composition tool over AWS spot instances. Bricklayer gets the resource requirements, constraints, and application type from the user and optimizes for cost or availability. In order to accomplish the optimization process, Bricklayer computes the ECU unit price, price volatility, and overall instance availability for each instance in the spot market and selects the ones that fulfill the requirements. Additionally, Bricklayer can select instances from distinct spot markets to reduce the bulk eviction risk. We implemented and evaluated Bricklayer with AWS historical data, and the results show that the costs can be reduced up to 54% on average, compared to regular spot instances, and up to 95% compared to regular on-demand instance prices without compromising the availability aspect.

ACKNOWLEDGMENTS

This work was supported by the Academy of Finland in the BCDC (314167), AIDA (317086), WMD (313477) projects and by the Swedish Foundation for Strategic Research in the project “Future Factories in the Cloud” (GMIT14-0032).

REFERENCES