IBM Cloud Databases: Turning Open Source Databases Into Cloud Services

Tim Waizenegger[1] Thomas Lumpp[2]

1 Abstract

Databases in all their forms are the backbone of most applications, running in the Cloud or on-premise. This creates a large demand for hosted, as-a-service database systems that are used either by Cloud applications, or even by on-premise applications. Through this demand, two types of offerings were created: new Cloud-native multi-tenant database systems and hosted instances of existing database systems. The first of those types, new cloud-native systems, may even have relational and ACID properties. On our IBM Cloud platform, we offer both types: "SQL Query" and "Cloudant" are our Cloud-native multi-tenant database systems. To cover the second type from above, "IBM Cloud Databases" offers popular Open Source databases like PostgreSQL, MongoDB or Redis as a Cloud service.

The IBM Cloud Databases offering includes database provisioning, maintenance and operations, backup and restore, scaling, and other features. As a newly built service, it has unique properties that make it stand out. IBM Cloud Databases is built as a native Kubernetes application that runs containerized versions of Open Source databases. Databases are provisioned as clusters or master/slave configurations, depending on the database, to offer high availability. To further decrease downtime for customers, we implemented a zero-downtime scaling feature for vertical memory scaling. This allows customers to save cost on memory when demand is low, but quickly scale-up without restarting the database when demand increases.

In this presentation, we give an overview of the system architecture and show how we integrated the different database systems. We discuss the advantages and disadvantages of running such a service on Kubernetes and provide insight into how we operate this service.

[1] IBM Deutschland Research & Development GmbH, Böblingen, tim.waizenegger3@ibm.com
[2] IBM Deutschland Research & Development GmbH, Böblingen, thomas.lumpp@de.ibm.com

doi:10.18420/btw2019-21

Fighting Spam in Dating Apps

Uwe Jugel[1] Juan De Dios Santos[2] Evelyn Trautmann[3] Diogo Behrens[4]

Abstract: Online dating allows for interactions between users with a high degree of connectivity. This digital form of interaction attracts spammers and scammers, who try to trick users to visit low-class competitors' websites or steal the users' money. Fortunately, each attacker leaves a footprint of its actions in the network. It is the task of an Anti-Spam system to detect these and punish the culprit.

In this paper, we demonstrate how LOVOO fights such illegal activities using a system of modular, scalable, and fault-tolerant Anti-Spam components. We describe our stream-processing architecture and how it ensures flexibility and facilitates resource-efficient processing of the millions of events produced by hundreds of thousands of users.

1 Introduction

Dating requires interaction and online dating allows interactions from one to many users and vice versa. On a dating platform, the participants belong to a very specific focus group: people looking for love. After a dating platform is established and has created a large user base, it becomes the perfect target for spammers and scammers to trick innocent users.

Our mobile dating app LOVOO allows our users to send *matches*, *likes*, *visits*, *smiles*, and *chat messages* from and to hundreds of thousands of users; 24 hours per day. Consequently, the LOVOO backend is a data-intensive application with near real-time requirements and so is our Anti-Spam system. It needs to oversee large volumes of various kinds of user interactions and detect and punish spammers, scammers, fraudsters, and misbehaving users in real-time.

For this task, the Anti-Spam system needs to be *modular*, *scalable*, and *fault-tolerant*. The individual Anti-Spam components need to collect user state and user history of various kinds to setup and improve machine learning models used to decide if an observed user behavior is regarded as spam or not. When using trained models for spam prediction, the components need live access to the acquired user state and history.

Moreover, training and adjustment of machine learning models must be quick and easy. All training data should be stored and be instantly accessible in a central data warehouse

[1] LOVOO GmbH, uwe.jugel@lovoo.com
[2] LOVOO GmbH, juan.dediossantos@lovoo.com
[3] LOVOO GmbH, evelyn.trautmann@lovoo.com
[4] LOVOO GmbH, Github:@db7

doi:10.18420/btw2019-22

to facilitate automatic training in particular and data analysis and research experiments in general.

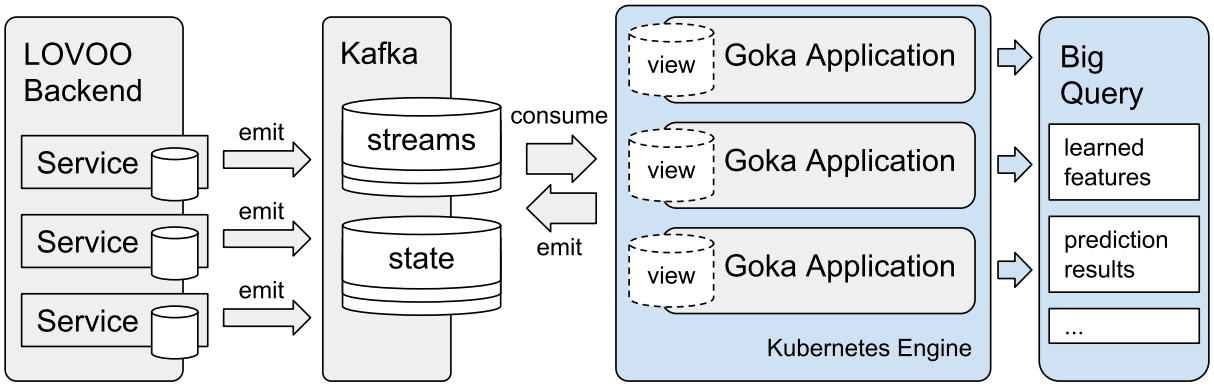

Fig. 1: Architecture Overview: Backend services publish interaction data to Kafka. Independent Goka processors and services asynchronously consume data in tables and emit state changes directly to Kafka. Machine learning features and predictions results are stored in BigQuery.

Paper Overview For fulfilling the aforementioned requirements and implementing the listed features, we combine a variety of cloud products and state-of-the-art technologies as depicted in Figure 1. All our components run in the Google Cloud using a variety of managed products such as Kubernetes Engine and BigQuery (cf. Section 2). We use Apache Kafka as our stream-processing foundation. Since Go is our core programming language for the backend systems, we have built Goka, an open-source library written in Go on top of Kafka that radically simplified our application development (cf. Section 3). We use these technologies to process text, images, user profiles, and user behavior to detect spam in our various Anti-Spam components (cf. Section 4). Eventually, we also evaluate our machine learning models and the quality of the spam-detection in general (cf. Section 5).

2 System Architecture

Our Anti-Spam components need to oversee any form of user interaction in near real-time and without slowing down the core services of the LOVOO backend. Therefore, all communication to and inside the Anti-Spam system is event-driven and thus asynchronous. Similarly, to achieve modularity, the individual Anti-Spam components should not block each other and work asynchronously. Figure 2 shows our system architecture.

Our Anti-Spam system consumes events from Apache Kafka topics. We refer to event topics as *streams*. Kafka is also used to store any kind of spam-related state about the users and devices connected to our platform. State is stored in log-compacted topics in Kafka, which we refer to as *tables*.

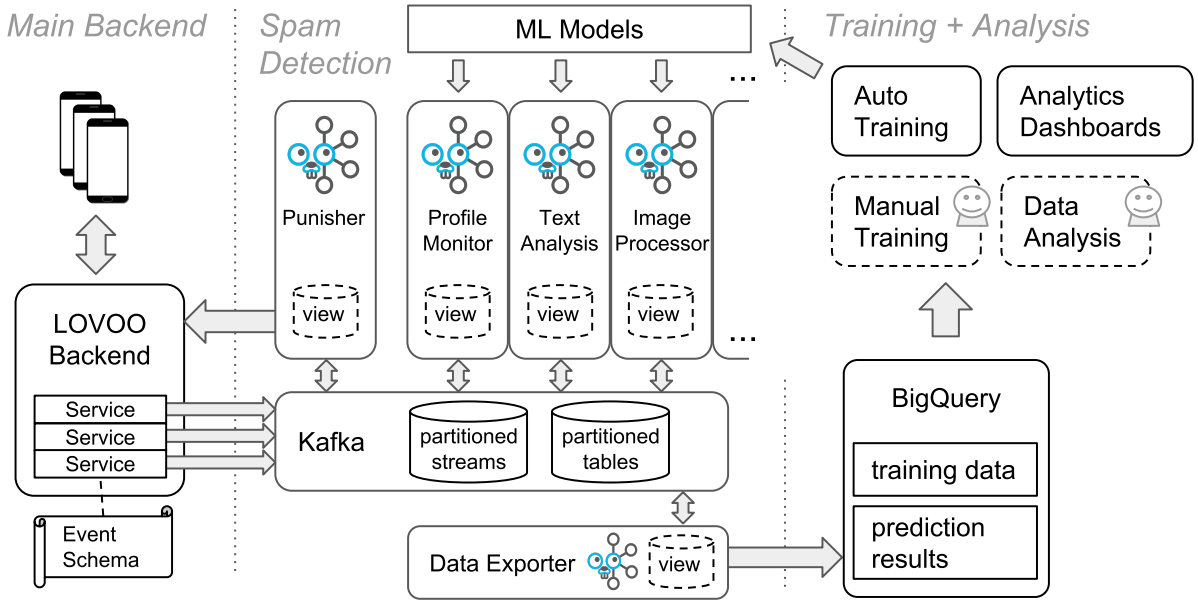

Fig. 2: LOVOO Anti-Spam Architecture.

The schema of events and state is defined with Protocol Buffers and shared in source code repositories. Any kind of service supporting Protocol Buffers and Kafka drivers can be used to publish or consume events and state updates from or to Kafka.

The majority of Anti-Spam components employ Goka to consume and update streams and tables. Goka is a stream-processing library written in Go that helps us building highly scalable applications with Kafka (cf. Section 3). Each component processes incoming events using rule sets, manually trained models, automatically trained models, or additional cloud services to learn about and detect spam in texts, images, user profiles, and user behavior. Learned data and prediction results are exported to BigQuery, i.e., our data warehouse in the Google Cloud. The warehouse data is used for data analysis and for training our machine learning models. The new models are tested against the current models in the individual spam-detection components to eventually replace the old models after a successful evaluation.

The Anti-Spam components (cf. Sections 4) are completely independent of each other and communicate exclusively via Kafka. For instance, our Image Processor extracts text from images and emits the obtained text back to Kafka, from where our Text Analyzers consume and analyze the text. If they detect spam in the text, they emit their findings back to Kafka, from where our Punisher component picks them up and applies some final safety measures before telling the LOVOO Backend to actually punish a user or device.

All time-critical components are written in Go using our Goka library that handles issues of scalability, local persistence, fault-tolerance, monitoring, data parallelism, and data consistency in Kafka.

For training our machine learning systems, we use scikit-learn, Keras and TensorFlow. The trained models are transferred to the time-critical components by dumping the learned coefficients and intercept value from scikit-learn and recreating the model in a Go data structure. The models trained on Keras and TensorFlow are hosted on Cloud ML Engine.

3 Goka: Go Stream Processing with Kafka

Our first version of Anti-Spam was built using a combination of event queues, off-the-shelf databases, and a caching layer. The system was hard to scale, caches needed to be carefully invalidated, and a lot of different technologies needed to be known and touched during each development cycle.

In the search for a cleaner and more scalable architecture, we redesigned the Anti-Spam system as a set of stateful stream processors, following the rationale of „I heart logs“ [**Kr14**] by Jay Kreps and „Making sense of stream processing“ [**Kl16**] by Martin Kleppmann.

To support this redesign, we developed and open-sourced Goka, a stream processing library written in Go that exploits several features of Apache Kafka, radically simplifying the implementation of applications. In fact, the new architecture turned out to be so flexible and performant that a number of our systems have been re-implemented with Goka. From user search to machine learning, Goka now powers several applications that handle large volume of data and have near real-time requirements.

The main features of Goka are the following.

Stream Processing Model. Goka allows us to easily define Kafka stream processors in our Go code, providing abstractions for setting up Kafka consumer groups, processor inputs, outputs, and state. Goka internally automates any handling of Kafka partitions.

Local Persistence. Goka provides a local storage model for caching compacted topics locally in LevelDB, for near real-time lookups.

Monitoring. Goka provides built-in metrics to monitor the processing state of various kinds of data consumers.

We now briefly describe the Goka concepts required to unserstand the remainder of this paper. For more details, please see our article on the LOVOO engineering tech blog [**Be17**].

Tables and Processors. Most of our applications employ one or more key-value *tables* to store the application state. The goal of Goka is to allow for manipulation of such tables in a composable, scalable, and fault-tolerant manner.

Each table is owned by one single component, we call it *processor*. The processor consumes input streams and produces state changes. State changes can only be applied to a table by the processor that owns the table. A single table owner is essential to achieve key-wise sequential updates, eliminating the need for complex and error-prone synchronization of multiple writers.

Each table is stored in a log-compacted Kafka topic as well as in a local LevelDB storage. The combination allows for fast reads and writes. Reads access the local storage, not requiring any network hop. Writes are applied locally and remotely, but remote writes occur asynchronously to the computation.

Keeping the table stored in Kafka provides not only fault tolerance by default, but also allows for easy processor migration by recovering the local storage from the Kafka table.

Views. A table stored in Kafka has a single owner processor that writes to it, but other processors may also read the table using views. A *view* is a local storage that continuously consumes state updates from a table in Kafka. Views are eventually consistent and allow processors to join input streams with tables to perform updates to their own tables or to emit new events into output streams.

Messages and Partitions. Messages are key-valued. Every topic in Kafka is partitioned, and messages are routed to partitions by hashing the message key. If input streams and tables are co-partitioned, i.e., they have the same number of partitions and the keys have the same domain, then the processors can be split into multiple instances, each owning one or more partitions of the table.

The Kafka consumer-group protocol assigns partitions to consumers. Goka exploits the same mechanism to assign related stream and table partitions to processor instances. For instance, consider a processor owning a table T and consuming two input streams A and B. The streams and the table each have, e.g., two partitions. When starting two processor instances, partition 0 of A, B, and T is assigned to one processor instance, whereas partition 1 is assigned to the other. Since keys are routed to partitions using a global hash function, partition 0 of A, B, and T contain the same key subdomain, which is disjunct from the other key subdomain of partition 1.

Scaling a Goka processor consists in simply starting multiple instances. When a new instance connects to Kafka, Kafka reassigns the partitions and distributes the plan to the instances. Once the reassigned table partitions are fetched from Kafka and stored locally in LevelDB, the instance starts processing the input streams from where they stopped before the reassignment.

4 Anti-Spam Processors

In Sections 2 and 3 we described how to set up a modular and scalable Anti-Spam architecture using Kafka as its backbone (also cf. Figure 2).

However, independent of our use of Kafka and Goka for stream processing, we designed our Anti-Spam system as a lambda architecture that supports batch and streaming processing alike. The individual Anti-Spam components can make use of any kind of external service required to effectively perform their job. In particular, we combine Goka-based stream processing with batch processing using, e.g., Google BigQuery and Cloud Vision.

This design has proven to be highly beneficial for all the different problems that Anti-Spam seeks to solve when detecting and stopping the proliferation of fake or disrupting profiles.

Each Anti-Spam component employs an algorithm that is particular to its domain and that detects spammers in a way distinct from the others. In the following, we describe our core spam-detection components and how they leverage Kafka and Goka, and make efficient use of external service in the Google Cloud. These components are the following.

Spotter	detects text in images.
Photographer	extracts text from images.
Terminator	classifies text based on rules.
Rosetta	classifies text using a machine learning model.
Sheriff	oversees user behavior.
Tracer	detects sequential interaction patterns.
Entity Reputation	detects spam based on the reputation of specific entities.

4.1 Spotter and Photographer

One of the means spammers use for distributing spam on our platform are images. The spammer writes a message, e.g., an advertisement for a pornographic or low-class competitor website on top of the image and uploads the image to LOVOO. Detecting this text requires a combination of computer vision and machine learning to detect and extract the text and subsequently analyze it to determine the adequacy of the content.

The text detection component is called Spotter and is written in Python. It consumes `picture_uploaded` messages from Kafka that are sent every time a picture is uploaded on LOVOO. The message contains the location of the user image in Cloud Storage and metadata about the image and the upload event. For every incoming message, Spotter retrieves the image from Cloud Storage, and, using OpenCV, it will apply a technique known as Class-specific Extremal Region [**NM12**] to detect the regions of the image that contain text. If Spotter detects a text region, a message is emitted to another Kafka topic that is read by the Photographer.

The text extraction component is called Photographer and is a Goka processor. It consumes messages from the aforementioned text detection topic populated by Spotter. Each message again contains the storage location of a now suspicious image. The image is retrieved a second time from Cloud Storage and sent to the Google Cloud Vision API to extract the text. The extracted text is emitted as another message to Kafka and picked up by the Terminator that verifies if the text violates any rules and thresholds.

Splitting up the task of image processing into a low-cost but less versatile text detection (OpenCV) and a more versatile but also more expensive text extraction step (Cloud Vision) saved us a lot of costs. Google charges for every image uploaded and less than 1% of our images actually contain text.

4.2 Sheriff

In comparison to most spammers, our regular users behave in particular ways. For example, a female user who is 32 years old, usually interacts with the app in a different way than a 21 years old male user. Moreover, spammers usually try to reach a broad audience very quickly and thus use the app in very uncommon ways. We track and oversee this user behavior in our Sheriff, where we employ machine learning to distinguish regular user behavior from the malign behavior of spammers.

The Sheriff is a Goka processor that keeps track of the frequency of the actions executed by our users, as well as individual user characteristics, such as gender, age, and their search settings, i.e., the gender and age of the users they are looking for at LOVOO.

The Sheriff uses a machine learning model trained using logistic regression, an algorithm that uses a logistic function for modeling the relationship between a dependent variable and a set of independent or predictors variables.

The model training is done in Python using scikit-learn. The training dataset contains over 1 million observations of over 30 features. These features comprise the user characteristics mentioned above and the relative execution ratios of particular user actions. For instance, given the three types of user actions `message_sent`, `like_created`, and `message_received`, and absolute executions of 3 sent messages, 7 likes, 2 received messages, the features values for the user actions are 0.25, 0.58, and 0.17 respectively. To train the model, we use k-fold cross-validation with $k = 5$, and to find an optimal set of hyperparameters, we fit the model using grid search [**BB12**] in the following search space:

`alpha:`	$0.0, 10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}, 10^{-6}$
`penalty:`	$L1$, $L2$, elastic net [**ZH05**]
`max_iter:`	500, 1000

4.3 Tracer

The Tracer is similar to the Sheriff, as it also consumes user actions to analyze user behavior. But instead of tracking execution ratios, it collects and evaluates the order in which the user executed the actions.

The Tracer processor consumes the most important user action topics from Kafka and encodes the topic name and the time span between subsequent events using a sequence of characters, e.g., as follows.

- `0:` Marks the start of the sequence.
- `S:` User has started the app and is logged in.
- `4:` User is idle for 4 seconds.
- `L:` User created a like on a another users' profile.

The sequence not only includes the user's active actions, but also passive events, i.e., other users' actions that affect the current user. For instance, for every created like there is a user who received the like.

In Goka, processing passive events may require reading and writing data in other partitions. This is done using Goka's loopback API. A loopback is an operation that emits a message back to the current processor, but with a different key. The message is then picked up by the correct processor instance to update the correct partition of the corresponding table.

In the Tracer, the loopback is used to update another user's action sequence. When consuming a `like_created` message it has as key the current user and one of its values is the liked user's key. This key is used for the loopback to send a message to the correct processor instance and update the correct sequence.

The Tracer model is an recurrent neural network (RNN) [**Ho82**] based on long short-term memory cells (LSTM) [**HS97**] trained on Keras and TensorFlow. Our Tracer network consists of three layers. The first one is an embedding layer, with an input dimension of 10, output dimension of 32 and maximum length of 1000. The second layer is an LSTM layer of 100 units. In the last layer, we classify spammers vs. non-spammers, using a dense layer with a single unit and a sigmoid activation function. The model also uses binary cross entropy loss and an Adam optimizer [**KB14**].

4.4 Rosetta

Rosetta is the component of Anti-Spam that deals with text-related features. In the LOVOO app, there are two text fields that the user can fill with freetext, the username and the "about me" section. Spammers use these parts of the app as another medium to advertise and spread their content. Thus, we trained various logistic regression models in scikit-learn

using usernames and "about me" data to classify them into spam content and non-spam content.

For the username model, we used around 13000 usernames, of which 60% were labeled as non-spammer and the remainder 40% as spam. For the training, we process all text as lowercase bigrams on character level. Then we produced our feature vectors by using a bag of words approach. Like the Sheriff model, we found the best set of hyperparameters using grid search and trained using k-fold cross-validation with $k = 5$. The scoring metric was the $F1$ score.

Because LOVOO supports a diverse number of languages, for the "about me" problem, we fit several models, each for a particular language. The size of the training datasets and pipelines are similar across all languages. As training data we used around 40000 records with a spam to non-spam message ratio of $1 : 1.5$. The training pipeline is similar to the one used for the usernames, except that in this case, we remove stop words from the text, before creating the bag of words. Afterwards, we fit a logistic regression model using grid search and 5-fold cross-validation; the scoring metric used is the accuracy.

4.5 Entity Reputation

The Entity Reputation component determines scores for the reputation of various users characteristics. These so-called entities are, e.g., the user's IP address, email domain, and uploaded images. If many users share an entity, e.g., the same image, and some of the users have been already flagged as spammers, the reputation score of that entity is decreased.

The component is a Goka processor that maintains the scores for such entities in a table. It consumes all topics from Kafka that contain updates for these entities.

The spam detection itself is a threshold-based geometrical separation between spammers and non-spammers in terms of user counts and share of received reports and other detected spammers associated with this entity. The scorer exploits that spammers in contrast to individual users share more entities just by being not an individual person. The ensemble of entities describes a spam probability.

To achieve a clear separation between scores of spammers and non-spammers, we model the spam-related features, such as the number of user-received spam reports and number of flagged spammers for that entity, as a sigmoid function per entity.

$$\frac{1}{1 + exp(-\beta(x - \lambda))} \in [0, 1]$$

The sigmoid functions range between 0 and 1 with a scaling factor β and a separation threshold λ. The separation threshold is obtained from our historical data. The ensemble of single entity scores contributes to a total reputation score given to every user. If a predefined score is reached, the user has been identified as a spammer.

Scoring entities with sigmoid scores resemble a logistic regression model. However, since the various entities have very different significance and reliability, we cannot score them with the same model. Instead, we fit the sigmoid function to every single scalar x per entity.

5 Evaluation and Results

We publish the progress and results of our spam-fighting activities at tech.lovoo.com in the form of regular transparency reports. In the following, we summarize the results published in our June 2018 edition of the report.

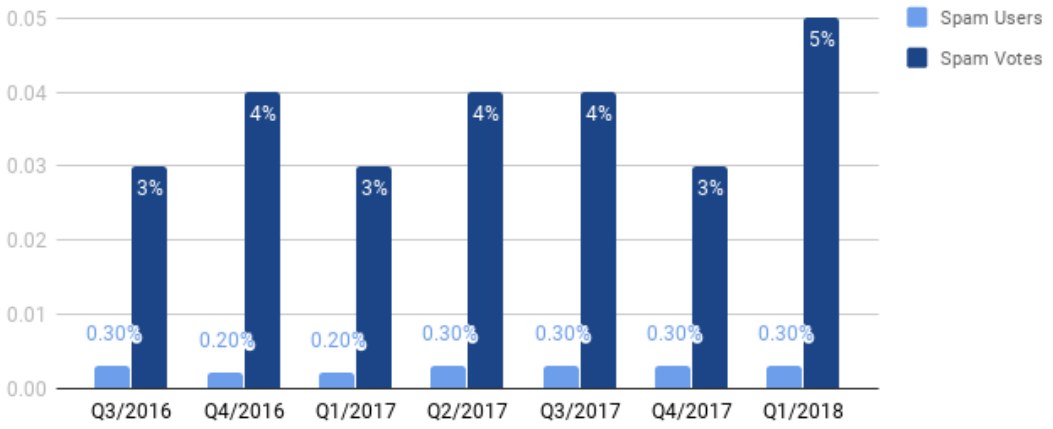

Fig. 3: Spam activity at LOVOO

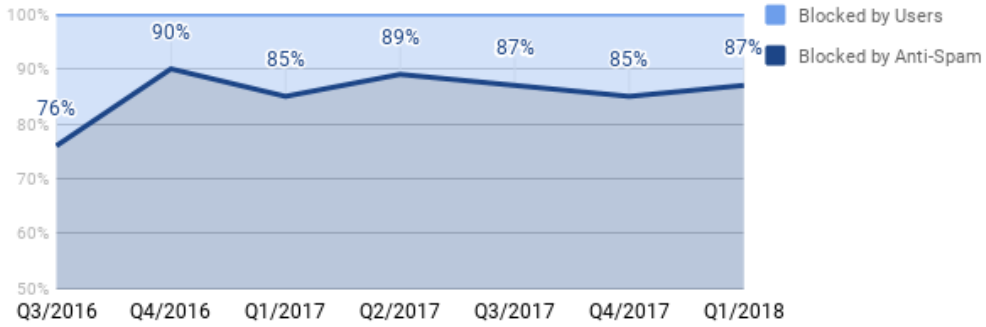

Fig. 4: Automatically detected vs. user-reported spam at LOVOO.

As shown in Figure 3, we currently (as of June 2018) have about 0.3% registered LOVOO profiles that are spammers and these spammers are responsible for 5% of the likes and dislikes, i.e., the votes on our platform, which is one of the main forms of digital interaction provided by the app. Figure 4 shows that of those spammers and scammers, over 87% are automatically detected and blocked by our Anti-Spam system. The other 13% are reported by other LOVOO users.

We are aware that these numbers alone are not the perfect measure of the quality of our spam detection. However, they show that our Anti-Spam system was able to maintain its

Fig. 5: Average hours before a spammer is blocked.

good spam-detection ratios over the last years, even though spammers and scammers are always coming up with new ways to work around our system and to trick our users.

Another indicator of the quality of a spam detection system is the duration before a spammer is blocked, as shown in Figure 5. Currently, we are able to block spammers in less than 42 minutes after they become active on the platform, with a significant improvement over previous quarters.

While the above numbers provide us with a very general metric how our Anti-Spam system performs, we also need to look at the details to see if our specific machine learning models are working as expected.

This validation of new detection models is one of our biggest challenges. Simple cross validation with historical data will always struggle with newly detected spam patterns, since the positive findings of the new models are not recognized by the former models and thus cannot contribute to the datasets used for validation.

We actually want to know two key metrics about the new models.

1. Does the new model detect spammers earlier?
2. Does the new model detect new spam patterns?

While the first question can be answered by cross validation with historical data, the second requires different, i.e., external metrics. Some of these external metrics are, e.g., the number of user-reported spam profiles, the number of requested user verifications, or the number and type of unblocks, i.e., when automatically or manually marking a blocked user as regular users again and no longer as spammer. Figure 6 depicts the trend for these unblocks on our platform, showing the number of unblocks over time for five different (undisclosed) block categories. An increase or decrease in a specific category tells us how well specific parts of the Anti-Spam system, and thus the new detection models, are performing.

In total, the number of unblocks of falsely detected users is decreasing, which suggests that our models and rules are actually blocking the correct profiles and less of our regular users.

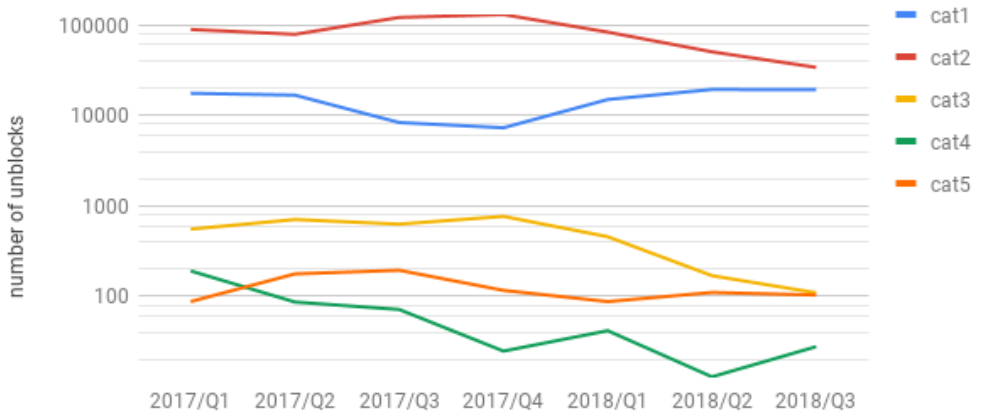

Fig. 6: Number of unblocks of falsely reported or detected user profiles in different (undisclosed) spam categories.

6 Conclusion

In this paper, we described LOVOO's Anti-Spam architecture and how we use stream-processing, batch-processing, and machine learning in production and in the cloud to fight spammers and scammers on our platform. In particular, we described how our open-source Goka framework helped us setting up a stream-processing solution on top of Kafka to create a scalable, fault-tolerant, and modular architecture.

For our main Anti-Spam components, we described how they interact asynchronously via Kafka, which kind of rules and machine learning they apply in which domain, and also how they interact with external services. In this regard, we also provided a real-world example of combining simple and sophisticated machine learning systems to save costs in the cloud.

Spammers are good at adapting their techniques to our systems and are constantly finding new approaches to circumvent our filters. In this regard, malicious content on user images is one of our biggest challenges, which we are trying to win using new self-developed detection models, but also pretrained models provided by machine learning services in the cloud.

References

[Kr14] Kreps, Jay: I Heart Logs: Event Data, Stream Processing, and Data Integration. O'Reilly Media, 2014.

[Kl16] Kleppmann, Martin: Making Sense of Stream Processing: The Philosophy Behind Apache Kafka and Scalable Stream Data Platforms". Report. O'Reilly Media, 2016.

[Be17] Behrens, Diogo: Goka: Go stream processing with Kafka. LOVOO Engineering, LOVOO GmbH, 2017, (online: https://tech.lovoo.com/2017/05/23/goka).

[NM12] Neumann, Lukáš; Matas, Jiří. Real-time scene text localization and recognition. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 281–305, IEEE, 2012.

[BB12] Bergstra, James; Bengio, Yoshua: Random Search for Hyper-parameter Optimization. The Journal of Machine Learning Research 13(1), pp. 281–305, ACM, 2012.

[ZH05] Zou, Hui; Hastie, Trevor: Regularization and variable selection via the elastic net. Journal of the Royal Statistical Society: Series B (Statistical Methodology) 67(2), pp. 301–320, Wiley Online Library, 2005.

[HS97] Hochreiter, Sepp; Schmidhuber, Jürgen: Long short-term memory. Neural Computation 9(8), pp. 1735–1780, MIT Press, 1997.

[KB14] Kingma, Diederik P. ; Ba, Jimmy L.: Adam: A Method for Stochastic Optimization. arXiv preprint for ICLR 2015, arXiv:1412.6980, 2014.

[Ho82] Hopfield, John J.: Neural networks and physical systems with emergent collective computational abilities. In: Proceedings of the National Academy of Sciences 79(8), pp. 2554–2558, National Acad Sciences, 1982.