Architecture

January 23, 2019
Version 1.1.0

STRICTLY PRIVATE AND CONFIDENTIAL

1PLUSX

1plusX architecture

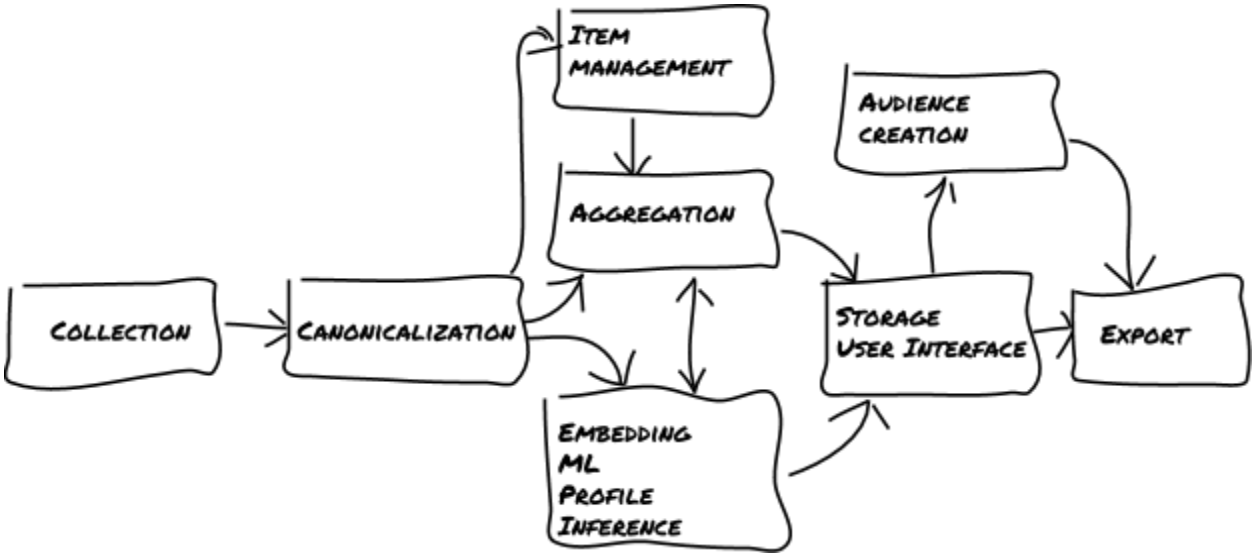
Introduction

Reference documents

- 1plusX Product Manual

Overview

1plusX processes hundreds of millions of events every day to build user segments from profiles and predictions about the users they originate from. Our infrastructure is built from a pipeline of batch processing jobs using Apache Spark. For data collection and serving, we run web services. Furthermore, we have a realtime version of the processing pipeline with reduced functionality (see appendix for diagram).



The infrastructure roughly breaks down into the following components:

Collection: Data enters our system: web tracking events, 1st and 3rd party logs, or custom files.

Canonicalization: All events are brought into a 'canonical form' such that downstream systems can process them in a unified way and don't need specialized code depending on e.g. the origin or type of data. This step also takes care of assigning a global "canonical" 1plusX user ID to all events independent of where and from what id space the events are coming from

Aggregation: This step aggregates all event data per user. The output of the aggregation are per-user attributes which can be used to build user segments. Several attributes from different sources can also be blended together to produce a new attribute in this stage.

Embedding/ML/Profile Inference: The historical per-user event data is used to create an embedding ("semantic fingerprint") of the user in a high-dimensional space. This embedding can be used to infer demographic properties of the user, as well as predicting other behaviors and attributes of the user. These embeddings are also used for the audience expansion feature that allows reach extension based on certain attributes or behaviours ("lookalike modeling")

Item Management: This component handles the meta information about items (websites/articles/e-commerce items/campaigns). What keywords match their content, which user interacted with it and when. This component enables the keyword-based targeting feature that allows to build audiences based on items the users have interacted with.

Storage / User Interface: The data is then stored as a master profile on disk. It is also imported into an elasticsearch instances which serves as the backend for the user interface.

Audience Creation: The UI allows for specification of audiences, by interactively placing restrictions on the attributes of the users. The audience definitions are stored in a table, can be edited by the user and are used to create user segments to be exported to ad servers

Export: We support a variety of 3rd party adserver integrations such as DFP and AppNexus. Furthermore, we offer an API for user and item information (Profile API).

Note that these components cover both batch and realtime: The realtime system hooks into the same collection and export components, but has its own canonicalization, profile inference, and audience creation components (microservices). Item management, aggregation, and user interface are batch only at the moment. Eventually, batch and realtime systems should be merged.

Architectural considerations

Our production system is built on a few design pillars that allow for stable operation.

Batch processing with Spark Scala - this allows us to have fault-tolerant and easily scalable system. If a job gets slow or runs out of memory, we can simply add more

resources. We use Luigi to manage the operation of the pipeline itself. Luigi understands dependencies between jobs and can rerun failed tasks etc.
Generally, we try to keep as few state as possible in databases (idmatcher and item manager below are the only ones) to remove the bottleneck and single point of failure. All state is computed from scratch, or incrementally on top of the past batches. This way, we get atomicity on the level of the full task and this allows tolerant rerunning of components in case they fail.

Profile Aspects, rather than a framework - all generated data about users and items is stored in a common format with a single join key (the user ID) and are materialized in Avro (similar to a column family in nosql databases). This allows for different systems to utilize whatever processing infrastructure they choose, rather than providing a framework that coerces all processing into a common structure. Any job can read profile aspects and emit profile aspects. At the end of the pipeline all profile aspects are joined into a single master profile. This design choice has given individual teams enough freedom to progress quickly and minimized inter-team and infrastructure dependencies.

No customer specific code beyond parsing of events - We allow ourselves to write custom parsers per customer because data ingestion can be very peculiar depending on the customer. But beyond this, all processing needs to be configurable. We do not allow any customer specific code anywhere else. Rebucketing of data, aliases of labels, and other mappings are built in a way that they can be configured using per customer configurations. We do run pipelines per customer, but the code is customer-agnostic and is parametrized by the per-customer configuration. This allows for quick scaling onto new customers.

Backbones

The backbones for our processing are:

Batch All input and output data and intermediate results are stored as files in S3 buckets. The individual components are batch jobs, mostly Spark. They are orchestrated using the Python Luigi framework. The “orchestrator” machine starts the daily pipelines via cron. The only communication between pipelines is waiting for another pipeline’s output at an agreed location and using it as an input.

Realtime All data travels through Kafka as Avro events. The individual components are microservices that consume events from one or multiple Kafka topics and output to one or multiple other topics, potentially maintaining an internal state using a database. Note that currently, a subset of the batch system is replicated in this way: It supports consuming data from. Eventually the batch and realtime systems should merge.

Data Collection

Data flows into our system on the left via different channels. We run our own tagger server that allows tracking of users across the web. We also consume custom logs from our customers, these can be ad logs from their ad servers, or other data provided by them. Finally, we need to match the ids between the different data sources which is done via cookie syncing.

Tagger System

Technology:

- Webserver
  - scala
  - play framework
- Client code
  - vanilla javascript
- Databases
  - Files transfered to S3 (for batch processing)
  - Kafka (for realtime processing)
- Deployment
  - ECS

1plusX maintains a proprietary tagger system that allows for seamless integration with the customer web properties. This tagger uses pixel tracking technology to collect user events. Both, customized javascript tags or simple tracking pixels are generated per customer and these tags are placed in the webpages to enable event tracking on browsers.
As a result of this system, 1plusX is able to track among other attributes – cookie ID, URL, and timestamp of the event. For web events, the data can be collected in either the 3rd party cookie space or the 1st party cookie space or both. The tagging system also supports collection of custom predefined events like conversions, scroll down, CRM data, etc. This is supported via a custom event endpoint.
Our system also supports selective wiping of PII attributes at logging time (IPs, CGI parameters) so sensitive data can be guaranteed to never enter our system.

Currently, our tagger endpoint processes about 100M events per day, but scales easily because it parallelizes trivially.

Cookie Syncs

Technology:

- Webserver
  - scala
  - play
- Client code
  - vanilla javascript
- Database
  - DynamoDB as a cache to remember which cookiesyncs are already done

In order to associate users in our id-space (cookie) to user data generated in a different id-space (e.g. 3rd party data provider, or adserver) we perform a cookie-sync procedure with the 3rd parties. This is usually done by implementing a two-way sync protocol with the partner on page load, whereby the user's browser first reads out our id via the cookie, redirects to the partner and then vice versa. This is a standard procedure in the industry. Our tagger system supports both partner initiated, 1st party initiated as well as piggybacking cookie syncs.

Datafile ingestion 1st and 3rd party

Technology:

- File fetching
  - golang
- Driver
  - python

We can import data in various formats from different sources. Currently we support download from S3, Google Cloud storage, (s)ftp. More sources can easily be added on demand. Our customers can provide additional user or event data via such files. We also read 3rd party event and panel data this way.

Canonicalization

Technology:

- Processing:
  - Spark / Scala
- Materialization
  - Avro

All ingested data sources are brought into a canonical form so downstream systems can consume them in a unified way. There are three types of canonical events:

1. Cookie sync events
2. Interaction events
3. Custom events

All events from one day are processed as batch in a spark job. There, they are parsed, written into the canonical structure and then materialized into the Avro format. The parsers are designed generically and configurable, but we resort to customer-specific parsers for select datasources. Once the parsing is done and all data is in canonicalized form, subsequent processing does not require customer specific code.

In the realtime system, there is a microservice that consumes tagger events from a Kafka topic, canonicalizes them by executing the same parsers we also use in batch, and outputs the canonical events to another topic. Note that for realtime processing, we don't support id matching nor other input channels than our tagger right now.

Cookie Sync Events / idMatching / Canonical IDs

Technology:

- Processing
  - Spark / Scala
- Key-Value store
  - Couchbase
- idMatcher code
  - Spark / Scala

Cookie Sync events are tuples of IDs that link a single user to different id-spaces. These events are parsed and canonicalized like all other events. But we also feed them into our ***idMatcher*** which keeps track of them.

The input to the canonicalization, Log and data transfer files are written with unique identifier from different spaces. E.g., the tagger uses the user-ID that 1plusX stores in the cookie under the `opecloud` domain. Customer specific files may be written in the ID space of the customer (the customer 1th party ID). If we are serving the tagger javascript snippet from a 1st party domain, that id-space will be different from `opecloud` and thus will require a cookie sync.

Downstream processes should not have to worry about matching IDs from different id-spaces when joining data sources together. To this end, we introduced the idMatcher. It manages all IDs, builds per-user clusters and elects a canonical unique ID per user and attaches it to all canonical events from this user, independent of the source.

The idMatcher works as follows: It consumes cookie syncs and builds a graph of ids. The edges correspond to ids that are associated with each other based on cookie-sync events. Each user is identified by a connected component of this graph. The canonical user ID is generated deterministically for each user from their connected component. The determinism in the ID election guarantees the stability of IDs even during backfilling of cookie-syncs which may become necessary when e.g. corrupt data has been accidentally inserted previously. The canonical ID and all associated IDs from different spaces (sometimes several for a single space) are then stored per user in a high-throughput and scalable key-value store (Couchbase), to enable us to look up the canonical ID in the canonicalization process.

Interaction Events

Interaction events are events that describe the interaction of a user with an item. The item is specified in the form of a URI, e.g. a website. We support different kinds of *InteractionTypes* (e.g. impression, click, conversion, purchase) which can also be user defined.

```
record Interaction {
   Event event;           // source and timestamp
   User user;             // canonical id
   string uri;            // item id
   InteractionType kind;

   // fields from the http header
   union { null, Browser } browser = null;
   union { null, DeviceType } deviceType = null;
   union { null, string } ip = null;
   union { null, string } language = null;
   union { null, string } referrer = null;
   ...
}
```

Custom Events

We provide customers with the option of sending us extra data through custom events. They can pass in any information they like which we will then parse into dynamic attributes. Those will make their way to the system and eventually will be made available as attributes per user to be used for e.g. audience building.
The custom events can be passed to us via file transfer, or packed as payload on http requests to our tagger.

We support a set of semantic types for custom attributes:

- Age
- Gender

- Income
- Distribution
- Int
- String
- StringSet

Age, Gender and Income are essentially Distribution types but with an attached semantic such that downstream systems (demographics inference, UI) can use them specifically based on their semantics. If 1plusX should not interpret the field at all, the basic types Int, String and StringSet can be used.

Aggregation

In the next step, canonical events are aggregated by user. We call these *per-user facts*, or *user aspects*. They contain information about the user in aggregated form. We support a number of different aggregation functions which can be configured per attribute

- Latest: override the attribute with the value of the latest event. This is useful when providing ground truth data for the user.
- Histogram: collect a histogram of the values of all events. This is useful when collecting usage statistics of the user
- SetMerge: build a StringSet with unique values from all events. This is useful for collecting labels such as e.g. interests of a user.
- Custom: We can also define any custom aggregation function in code.

The aggregation step consumes events and emits user aspects, it can operate on dynamic types that are injected via Custom Events.

User Aspects

User aspects are our primary data structure. They are essentially simple key-value tables stored in Avro, similar to column families. The key for every row is the canonical ID of the user, the value is a specific attribute, or a collection of attributes of the user. These tables reside on S3, in daily dated directories.

We found this data structure to be very useful. Any process can easily generate them with Spark. They can be consumed by other processes that derive and emit new user aspects. When reading them, we can limit reading to the attributes we are actually interested in, thus greatly reducing IO.

Profile Join

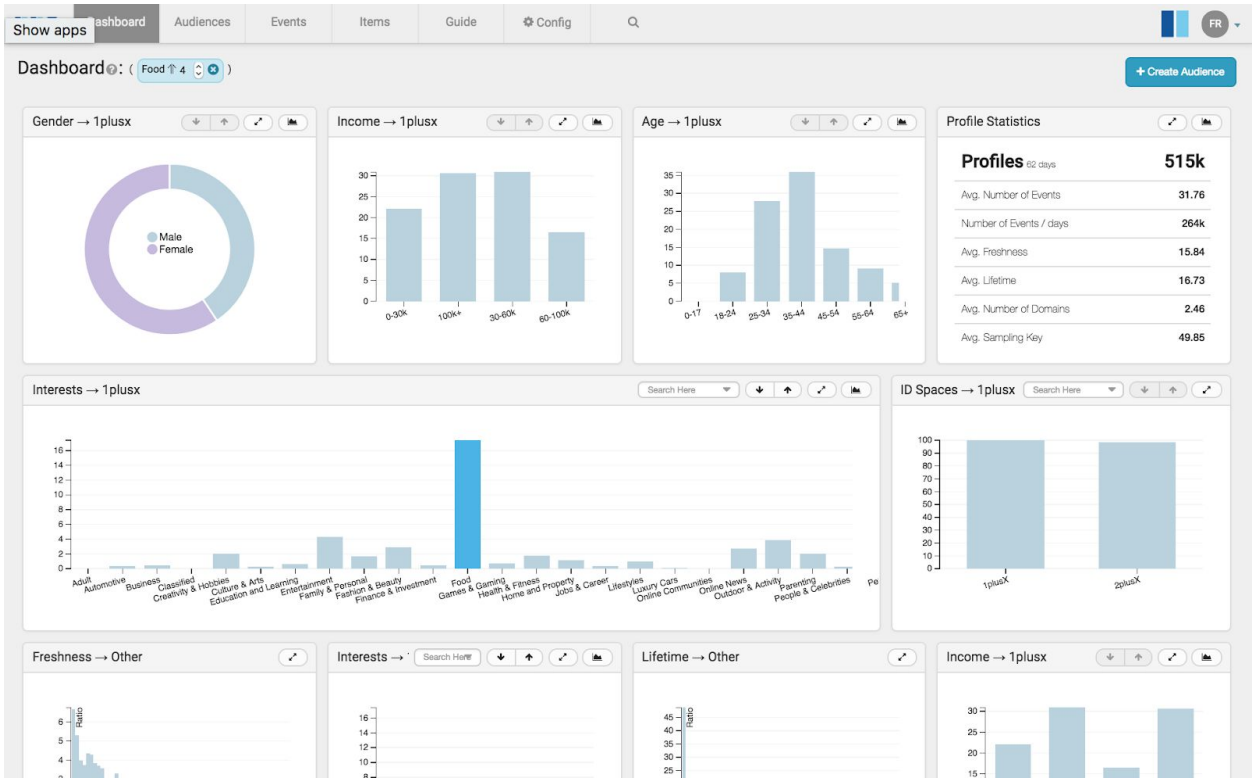
At the very end of all aggregations we join (configurable) all per-user aspects together into a master profile that contains all aggregated information per user, for all users. This data structure can then be used by downstream systems (UI, exports) to reference all information about any given user.

User Interface + Backend

User Interface

Technology:

- UI Backend:
  - NodeJS
- UI Client:
  - Angular



A screenshot of our UI. The UI allows our customers to:

- Get a global overview of their user base (profiles)
- Slice and dice by user attributes, predicted, as well as deterministic (1st party data via data consolidation) in real time.
- Audiences can be created from restrictions on the profiles
- Export channels for audiences are defined
- Topics / KW based targeting features can be managed from here
- The UI provides an account management system that allows granular control over audience creation, sharing and other administrative features.

Audiences

Technology

- Customer metadata / Audiences
    - PostgreSQL

Audiences are restrictions on the attributes of the users and can be combined in arbitrary Boolean expressions. Below is an example of an audience that targets “females with a high interest in science”. It is defined by finding all profiles that have a female probability greater than 85% and requiring the interest “Science and Technology” to appear in the top 2 interests of the profile.

```
{
  "operator": "and",
  "uuid": "afc6296c-14be-4371-b008-cd444cea7ede",
  "nodes": [
    {
      "type": "Probability",
      "id": "1",
      "name": "Female",
      "threshold": 85,
      "operator": "gt"
    },
    {
      "type": "MaxRank",
      "id": "324",
      "name": "Science and Technology",
      "threshold": 2,
      "operator": "maxRank"
    }
  ],
}
```

These audience definitions are stored together with other customer metadata in a PostgreSQL table (frontend database).

UI backend / profileExplorer

Technology:

- Processing

    - Spark / Scala / Akka
- Store
    - elasticsearch 5

The UI is able to show aggregations of profiles on arbitrary restrictions in real time: the user of the UI can edit the restrictions by merely clicking on attributes and the UI updates immediately. Since the restrictions and aggregations on the data need to happen interactively in real time, we need to be able to serve these with very low latency. We aim at serving below 500ms in the 95th percentile. To accommodate these requirements we use an ElasticSearch instance as the backend. We sample 5% of the data to achieve a good latency / accuracy tradeoff.

We import the user profiles computed from the Aggregation and Inference pipelines into ElasticSearch in batch, once per day. The profileExplorer binary serves as both, importer for the data, as well as UI backend. It receives queries from the UI and translates them into ES queries which it executes.

Export

Audience Aspect Creation

Technology:

- Spark/Scala

A daily job reads all audiences from the database and creates audience aspects. This means it iterates over all user profiles and aspects and applies each audience definition to them to see which one matches. This job emits an ***audience aspect***, a record that stores for each user ID the list of audiences which it is part of. Note that this step happens on the full, unsampled data. This aspect can then further be used for exporting audiences, or retrieving the list of audiences for a given user.

In the realtime system, there is a microservice that consumes profile events from a Kafka topic, applies the same audience filters we also use in batch, and stores the matching audience IDs for the user at hand in a Kafka state store (in Kafka, cached on the machine). We only maintain audiences for users that were active in the last 24 hours. Every night, the state is wiped. Audience data for users who haven't been active lately come from the batch system.

profileAPI export

Technology

- Couchbase
- Scala/Akka
- ECS

We support returning all audiences for a user in real-time via a web request to our profileAPI. The request needs to happen from the user itself via browser. That way, the cookie can be read out and the profile retrieved for the given ID from a high performance key-value store (Couchbase) which can serve these with sub-10ms latency at the 99th percentile, not including network round-trip.

The profile API is the point where batch and realtime data is merged. It has two backends: a Couchbase which contains the audiences computed in batch keyed by user id, and the audience service (see Audience aspect creation), which provides the audiences computed in realtime. For a given user, the profileAPI returns the batch audiences if available, and otherwise the realtime audiences.

AdServer / DSP exports

Technology

- Scala

We offer integrations with a variety of ad servers and DSPs and are constantly adding more. Currently, we support

- Google DBM and DFP
- SmartAd
- AppNexus
- Adition

The exports run once per day. They include creation of new user segments updated every day via incremental updates when supported by the activation channel.

Item Management Subsystem

Technology

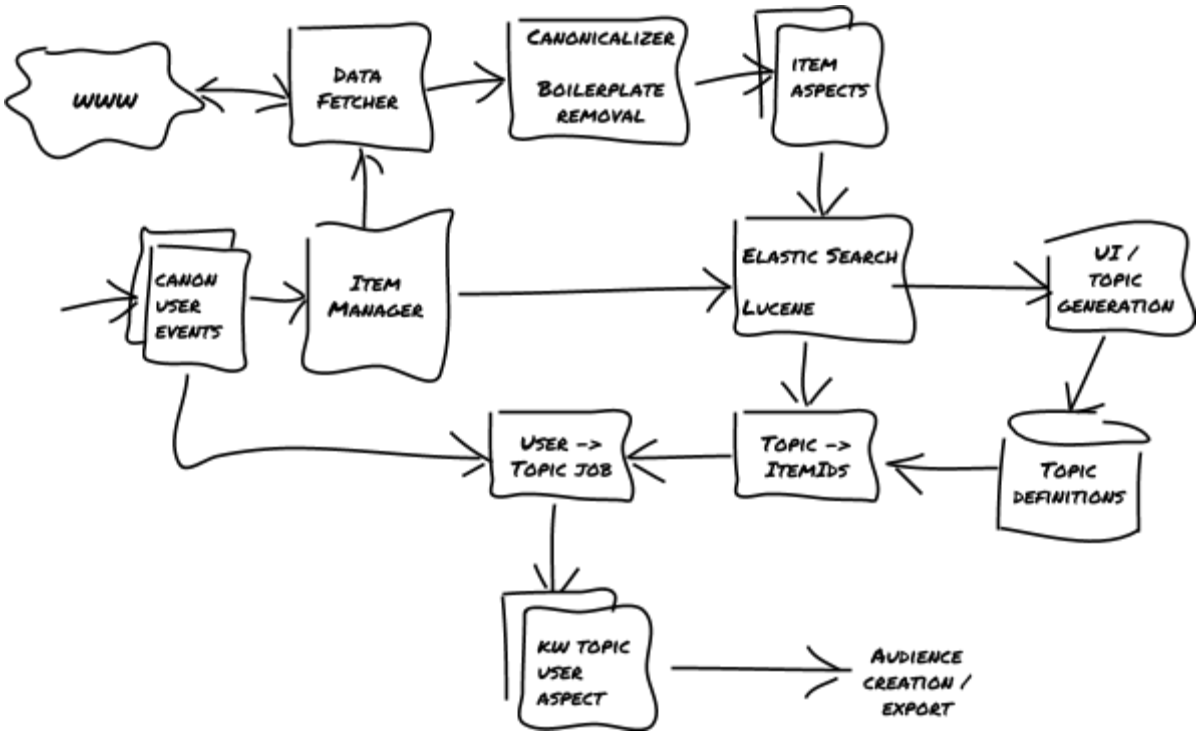
- elasticsearch 5
- Boilerpipe
- Scala

- Go

We offer ‘Keyword based targeting’ (KW based targeting) as a feature. It allows building user segments based on pages they have visited. The visited pages are defined by matching a set of keywords which the customer specifies.

This feature is built on top of an ElasticSearch instance for document retrieval and a bunch of nice tricks using data sketches to do audience size estimation in real time.

Here is a rough overview of the architecture:



At the center of the pipeline is the ***Item Manager***: It consumes (incrementally) all interaction events for the customer for a given time window (usually 90 days) and extracts all ***URIs*** on which they happened. These URIs build the corpus of websites over which we can do KW based targeting. These URIs are sent on the fetcher and periodically fetched (freshness). Fetched pages are ingested into the system via the standard canonicalization pipeline which here plays the role of boilerplate removal to extract the textual content of the website.
This content is then imported into and indexed by ElasticSearch where URIs for a given keyword can be retrieved in real time. The UI allows the creation of topics (sets of keywords). Definitions of these are stored in the frontend database. For purposes of the UI,

audience sizes for the topics are estimated in real time using the datasketch HyperLogLog (count distinct on these large joined datasets would be too expensive).

A daily job builds a table “topic to items” that maps all topics to their URIs. A second job then builds a per-user aspect (like any other aspect, see above) that maps each user to the set of topics it has interacted with via a join of the “topic -> items” with the interaction events (“user to items”).

The “user -> topic” aspect is no different to any other per-user aspect and can deterministically be selected from for audience building. This way the KW based targeting fits nicely into the existing infrastructure.

Note that the item manager, fetching, and indexing infrastructure is currently being redesigned as a realtime-capable system based on Couchbase, Kafka, and microservices.

ElasticSearch

We use Elastic Search 5 for the retrieval of the documents that match a KW based audience. We use BM25, ES’s default algorithm for scoring the documents against the keyword. To determine the cutoff threshold we have performed human evaluations.

Probabilistic audiences and Predictive attributes

Technology

- GenSim/tensorFlow
- Python+SciKitLearn
- Spark/Scala
- SparkML
- Parquet

We offer probabilistic profiling of users (cookieIDs, deviceIDs, UUID) based on browsing behaviour. This means we are able to infer/predict attributes such as age and gender based on browsing behaviour of the users with additional ground truth input that can be received via multiple sources. Additionally, we use the Item management subsystem in order to extract semantic information about the items that we see in our event logs. As an input to this probabilistic profiling component, we learn a representation of the items and users. This representation learning model is designed in a scalable and flexible manner in order to allow us to use the same representation for multiple learning tasks including but

not limited to audience expansion, audience optimisation, semantic targeting, content recommendation etc.

Representation Learning - Items and Users

The input to this ML subsystem is the set of events per userID that we have seen in the last 60-90 days. The collection of items (URIs) per user is considered as a document that describes the user. And the collection of all the users is considered as a document collection. This allows us to treat the URIs as words in a language model and use results from deep representation learning techniques as applied to Natural Language Processing (NLP). Using these models, we are able to learn a robust representation (embeddings) of both items and users which satisfy various desirable properties such as a complementarity, supplementarity, and compositionality.

The item embedding computation module returns item embedding vectors of a configurable prespecified dimensionality. This module is built from components such as word2vec, gensim and also allows easy plug and play into deep learning frameworks such as tensorFlow. This allows us to scale to hundreds of thousands of items. In the event that we have more items we could handle it via parallel execution of embedding computations (offered in tensorFlow for example) or we could downsample items accordingly (this is done already in all the popular frameworks) by thresholding on popularity.

Once the item embeddings are learnt - the user embedder module uses the user history and the learnt item embeddings to produce a representation of the users. In its simplest case, the user embedder simply averages the embeddings of items in the user's history. Other models that we use include weighted averaging, attention based mechanisms and other combination modules for learning more complex task specific user embeddings.

Feature Evaluator

The goal of the feature evaluator module is to produce coverage and quality metrics on the user embeddings produced by the embedder module. The evaluator constructs multiple tasks that test various components of the input signal and the efficiency of the embeddings in encoding these signals. Some of the tasks that are tested are Socio-Demographic predictions (including Age and Gender), temporal activity prediction, semantic content/interest prediction, etc.

This allows us to measure multi-task performance of general purpose user embeddings or specific task performance of specialized embeddings. Candidates for production embeddings are chosen from the best performing embeddings on each task.

Supervised Learning pipeline

Standard attributes like age, income and gender are predicted using a supervised learning approach which uses ground truth labels from various sources. The ground truth could be aggregate sources which provide URL/Domain level aggregate statistics or individual user level ground truth from CRM, Panel or any other first/second/third party data.

The learning pipeline itself takes the ground truth dataset which consists of datapoints of the form *(ID, embedding, label)*. The ID could be either denoting a user or an item and the label is either a single valued user level ground truth, or distributions over labels for either users or items.

The type of model to use and the corresponding set of hyper-parameters to tune are provided in a configuration file. The default models are linear and logistic regression provided by sparkML in order to do multi-class or binary classification. In addition, we have extended sparkML to handle learning from label distributions.

Each run of the supervised learning pipeline automatically splits the data into a pre-specified training and test split, performs cross-validation on the training set in order to select the best model parameters and outputs the model file with the best parameters based on test set evaluation along with the metrics. At serving time, the serving module can easily instantiate a model with the parameters set from the latest (and best) output from the supervised learning pipeline.

User Socio-Demographics

As part of the standard product offering - 1plusX computes predictions for gender and age of the users. These computed socio-demographics are stored as user aspects discussed earlier in this whitepaper. In addition to gender and age, our systems are also capable of predicting other attributes provided there is enough ground truth labels (around 2-10% of the user population) for the same or enough aggregate audience information at the item ID level.

Each attribute is computed via its own supervised learning pipeline. The attributes could be binary, categorical, ordinal or continuous valued. The input at the user level is represented

as a [feature, label] pair and at the item level as [feature, label distribution] pair. The features of the items and users are the output of the embeddings pipeline.

In the realtime system, there is a microservice that consumes canonical events from a Kafka topic, aggregates them to a user-embedding and executes the inference. It outputs the prediction/user profile aspect to another Kafka topic where it is consumed by the audience service.

User Interests

In addition to socio-demographic attributes, we also compute user interests based on browsing behaviour. Here, we piggyback on the item management subsystem and the data fetcher module to extract content and semantic information from the fetched content. Each URI is then assigned a set of interest values from a pre-defined taxonomy (IAB v2) based on the content using semantic content classification tools. Post this, each user is then assigned the aggregated interests resulting from the browsing behaviour of the user. By default, the IAB content classification scheme is used and we classify content into the top level of this taxonomy.

The key components here are the semantic classification tool and the user level aggregation modules. The semantic classification should also work across different languages. We use a classification algorithm built in-house that achieves state of the art performance on item level classification. This is evaluated based on human judgement labels collected via our item ground truth collection game. The user aggregation, in its simplest form, could be a sum of all the interest scores from the content consumed by the user. More complex techniques for user aggregation are also possible within the framework of our aggregation system provided there is user level ground truth on the interests of the users.

Audience Expansion

Another feature that uses the embeddings framework and the supervised learning module is Audience Expansion. Using this feature, a campaign manager (of our customer) could select a few users as the ideal candidates for a campaign. This set could be decided based on various qualifying attributes for the current campaign or based on actions (clicks, conversions, etc.) on a previous similar campaign. The audience expansion feature then finds more users from the entire population that are most similar to the current selected set of ideal users (seed set).

Internally, we cast this problem as one of similarity learning on the universe of users employing the embeddings again as the features of the user. The idea is to find a function

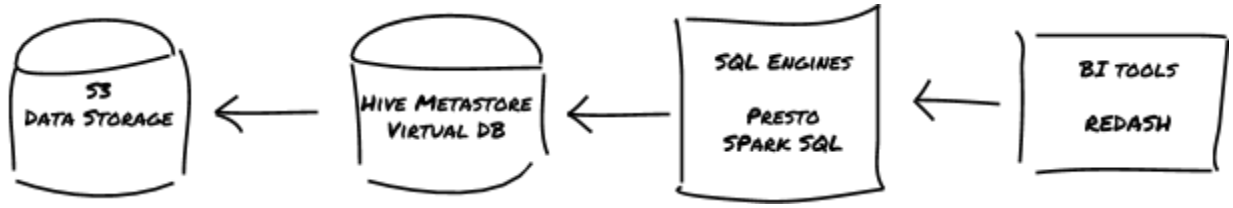
that discriminates those users in the seed set from a random set of users in the general population. Once this function is computed, it can be used to generate sets of different sizes in decreasing similarity to the seed set. The amount of expansion is determined by the required reach of the campaign and the size of the seed set that qualifies for the current campaign.

Data Lake

Technology

- Processing
  - PostgreSQL
  - Hive metastore
  - Presto
  - Redash
- Materialization
  - Parquet, Avro, JSON

In order to make our data easily accessible, we built a data lake. This adds a virtual database schema on top of our data (which sits on S3) and allow us to quickly run SQL queries on it allowing us to easily run reporting and analytics. Moreover, to make it more efficient, we decided to convert some of our data to Parquet and optimize it for faster queries.



Parquet file format

Parquet has multiple advantages over Avro and JSON for querying. The main ones are that it is a columnar file format and it allows you to optimize the compression of the data. This is a great benefit for querying because both can greatly reduce the IO. Less IO, means faster queries. The simplest example is that reading a column would entail reading the full dataset in a row-oriented storage format like Avro or JSON. With columnar storage, only the desired column would be read. Therefore, this becomes an obvious choice when running queries on a large input file (many columns) with a small output - which is the case of most analytics queries.

Hive Metastore

Hive Metastore allows you to build a virtual SQL database layer on top of unorganized files. When working on many files with the same schema, this allows you to build a virtual table on top of all these files which will then enable SQL engines to run queries on your files. A lot of our data is in a canonical format meaning that it shares the same schema. Therefore, this metastore allows us to represent our data into tables accessible to the SQL engine. Moreover, it can also manage the tables and the files itself. This allows for easy optimization on the file structure which would be painful to maintain manually. For example, a lot of our data is daily and Hive can maintain a partitioning of the table on the day. This will allow SQL engines when filtered on specific dates to dive directly to the files for these days, significantly minimizing the amount of data to read. We also make use of more advanced optimization like splitting this daily data in a specific way (sharding/bucketing) optimizing the lookup of certain crucial columns even more.
All this Hive metastore metadata is stored in a PostgreSQL database.

Presto (SQL Engine)

Presto is an all in-memory SQL engine fully compatible with Hive metastore. This is a very fast SQL engine developed by Facebook which is optimized for fast and concurrent queries. The benefit of this SQL engine over others is that it takes full advantage of optimization performed in Hive (e.g. partitioning and bucketing as described in the previous section).
It is also a supported data source in many BI tools

Redash (BI tool)

Redash is a BI tool that offers a nice UI to perform, version and share queries. It also offers quick ways to build charts out of a query's output and build dashboard.

Metrics and Monitoring

Technology:

- InfluxDB
- Grafana

All our services send metrics to InfluxDB using collectd or other means.

An example screenshot for some ElasticSearch metrics.

All our production systems emit metrics to a central place. We roughly group our metrics in the following way

1. Operational / system health metrics, running on all our continuously running hosts
    a. CPU load
    b. free memory
    c. free disk
    d. latency
2. Business metrics
    a. number of events processed
    b. number of profiles
    c. ...
3. Quality metrics
    a. accuracy of our gender inference
    b. precision of our interest classifier against ground truth
    c. ...

The vital metrics are guarded by alerts that will fire if metrics are showing anomalous behavior. Our on-call engineer will handle these alerts.

Cloud Deployment

The whole system runs on AWS in the eu-west-1 (Ireland) and eu-central-1 (Frankfurt) regions. We mostly use lower-level cloud-services (EC2, ELB/ALB, EB, ECS, S3, EMR, RDS), but not AWS-proprietary databases and queues (with a few exceptions). This means that our application code is cloud-agnostic, but the deployment is cloud-specific.

Provisioning We provision resources like databases (Couchbase, Elasticsearch) and queues (Kafka) in AWS using Terraform and Ansible, and sometimes the Console, but we are

migrating towards using Terraform exclusively as far as possible. A big chunk of our resource usage also comes from EMR-clusters and clusters of EC2 machines that we spawn dynamically from our pipelines. To this end, we have built a library that extends Luigi with cluster management functionality.

Service Deployment We have been using EC2 (with Ansible), Elastic Beanstalk, and ECS for service deployment. We are now migrating to a unified solution based on ECS. This solution uses Cloudwatch for log collection.

Continuous Integration/Deployment We run our own Jenkins instance on EC2. On every master build, Jenkins builds one Docker image for the batch pipelines as well as one per per service. The nightly build is used automatically for running the pipelines on the next day. Services are deployed by triggering special deployment jobs on Jenkins manually. These jobs invoke the ECR and ECS CLIs.

Multi-region setup The multi-region setup we have right now has mostly historical reasons: We started in Ireland and our batch pipelines still run there. Since our customers are mostly in the DACH region and the invocations of our services are latency-sensitive, we moved the data collection and serving services (tagger and profileAPI) to Frankfurt. We move data either via S3 cross-region-replication or VPC peering connections. Also, the services making up the realtime pipeline (which replicates a subset of the batch pipeline) and the Kafka cluster run in Frankfurt, since it would be a latency hit to transfer the data to Ireland and back. In the mid-term, we will need to provide good latency from all over Europe, so a proper multi-region setup is up for discussion.

Networking Setup We run our production system in one VPC per region. The datalake product, which is especially sensitive since it allows customer code execution, runs in a separate VPC per customer. In Frankfurt, we have multiple subnets with NACLs and endpoints for various services. However, we have decided to go with a simpler setup (public, private, and protected subnets) and will implement this in Ireland soon.

Appendix

Diagram of import, ingestion and aggregation architecture

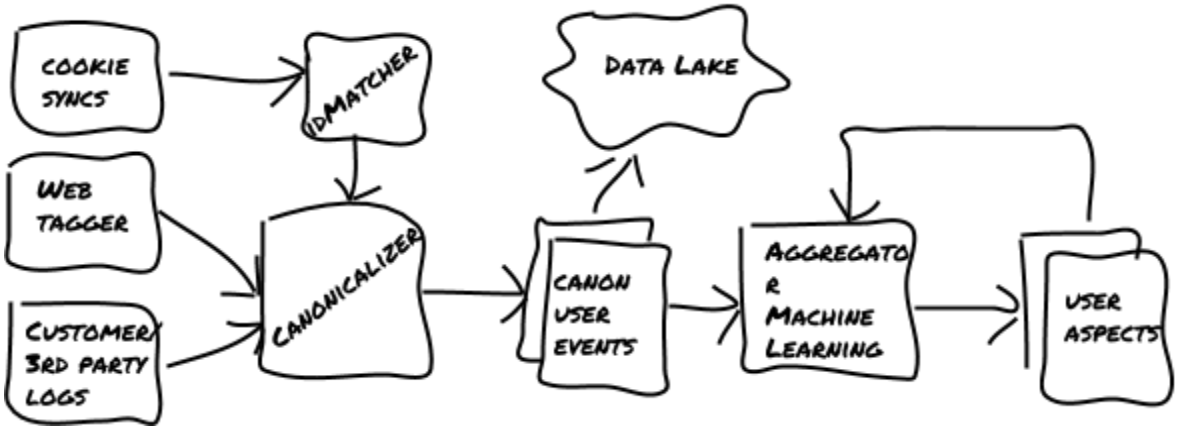


Diagram of audience building and export architecture

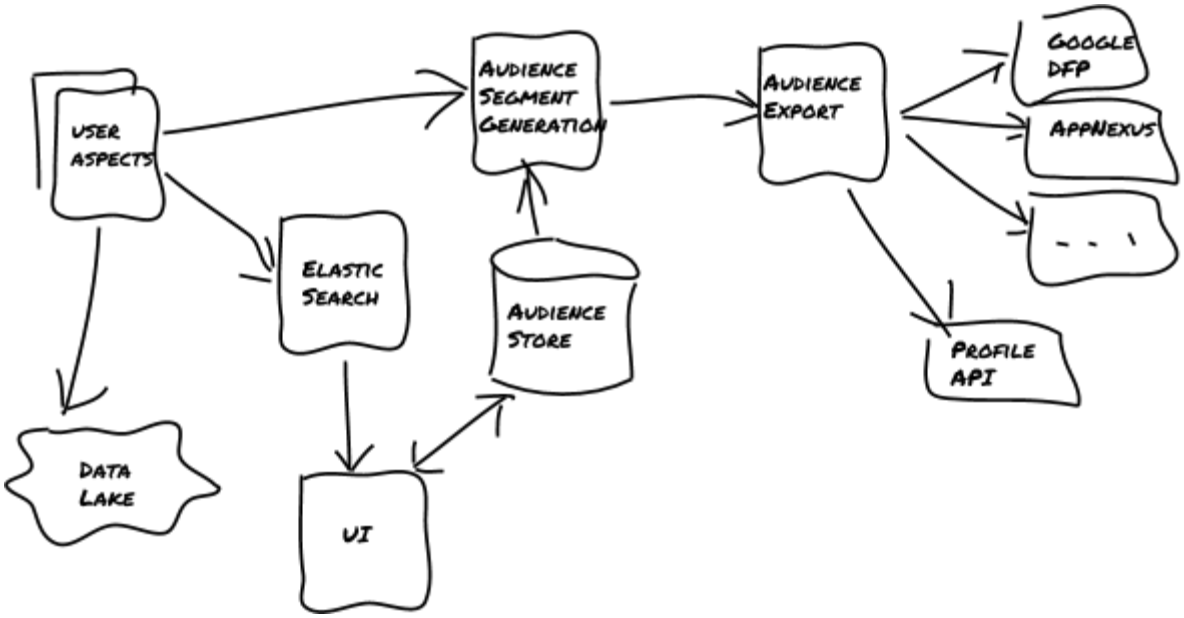


Diagram of realtime subsystem