Sarah Martin

Northwestern University, School of Professional Studies

Foundations of Data Engineering, Professor Lance Levenson

February 22, 2019

Project 1: Business Processes and Requirements Definition

for a Movie Recommender Engine

encoding – ch. 4

* ***At the beginning of this chapter we said that whenever you want to send some data to another process with which you don’t share memory—for example, whenever you want to send data over the network or write it to a file—you need to encode it as a sequence of bytes. We then discussed a variety of different encodings for doing this.***
* ***In this chapter we looked at several ways of turning data structures into bytes on the network or bytes on disk.***
* *Rolling upgrades allow new versions of a service to be released without downtime (thus encouraging frequent small releases over rare big releases) and make deployments less risky (allowing faulty releases to be detected and rolled back before they affect a large number of users). These properties are hugely beneficial for evolvability, the ease of making changes to an application.*
* *During rolling upgrades, or for various other reasons, we must assume that different nodes are running the different versions of our application’s code. Thus, it is important that all data flowing around the system is encoded in a way that provides backward compatibility (new code can read old data) and forward compatibility (old code can read new data).*
* Python uses pickle, but don’t use this
  + But security issue (cause anyone can get you to decode a pickled file and thus instantiate their classes which may try to access your data. Movie preferences are not an issue, but don’t send private personal info in this format
* Json for user data
* Csv (or xml?) for movie db data
* Careful with movie autoencoders:
  + There is a lot of ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string that happens to consist of digits (except by referring to an external schema). JSON distinguishes strings and numbers, but it doesn’t distinguish integers and floating-point numbers, and it doesn’t specify a precision.
* Use schema support if available
* Binary schema–driven formats like Thrift, Protocol Buffers, and Avro allow compact, efficient encoding with clearly defined forward and backward compatibility semantics. The schemas can be useful for documentation and code generation in statically typed languages. However, they have the downside that data needs to be decoded before it is human-readable.
* ??? JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON, and Smile, to name a few) and for XML (WBXML and Fast Infoset, for example). These formats have been adopted in various niches, but none of them are as widely adopted as the textual versions of JSON and XML.
* Apache Thrift [[15](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Slee2007vh)] and Protocol Buffers (protobuf) [[16](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#GoogleProtobuf)] are binary encoding libraries that are based on the same principle. Protocol Buffers was originally developed at Google, Thrift was originally developed at Facebook, and both were made open source in 2007–08 [[17](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Anishchenko2012tx)]. Both Thrift and Protocol Buffers require a schema for any data that is encoded. To encode the data in [Example 4-1](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#fig_encoding_json) in Thrift, you would describe the schema in the Thrift interface definition language (IDL) . The equivalent schema definition for Protocol Buffers looks very similar:
* Thrift and Protocol Buffers each come with a code generation tool that takes a schema definition like the ones shown here, and produces classes that implement the schema in various programming languages [[18](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#ThriftLangs)]. Your application code can call this generated code to encode or decode records of the schema.
* One detail to note: in the schemas shown earlier, each field was marked either required or optional, but this makes no difference to how the field is encoded (nothing in the binary data indicates whether a field was required). The difference is simply that required enables a runtime check that fails if the field is not set, which can be useful for catching bugs.
* Support backward compatibility (less imp to support forward compatibility)
  + Only need forward compatibility if newer db data (pulled and stored with newer code) needs to be read by old/existing code (user recommendation request pre-new code update)
  + Application shouldn’t change that much that frequently
* Apache Avro [[20](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#ApacheAvro)] is another binary encoding format that is interestingly different from Protocol Buffers and Thrift. It was started in 2009 as a subproject of Hadoop, as a result of Thrift not being a good fit for Hadoop’s use cases [[21](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Cutting2009tu)].
* Avro also uses a schema to specify the structure of the data being encoded. It has two schema languages: one (Avro IDL) intended for human editing, and one (based on JSON) that is more easily machine-readable.
* Every new pull for imdb and movielens will have same schema but might be diff from pull to pull
* User input could be more variable over time
* Keep a file in storage containing all old and existing schema version (to read old data?)
* When two processes are communicating over a bidirectional network connection, they can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection. The Avro RPC protocol
* By contrast, if you were using Thrift or Protocol Buffers for this purpose, the field tags would likely have to be assigned by hand: every time the database schema changes, an administrator would have to manually update the mapping from database column names to field tags. (It might be possible to automate this, but the schema generator would have to be very careful to not assign previously used field tags.) This kind of dynamically generated schema simply wasn’t a design goal of Thrift or Protocol Buffers, whereas it was for Avro.
* In dynamically typed programming languages such as JavaScript, Ruby, or Python, there is not much point in generating code, since there is no compile-time type checker to satisfy. Code generation is often frowned upon in these languages, since they otherwise avoid an explicit compilation step. Moreover, in the case of a dynamically generated schema (such as an Avro schema generated from a database table), code generation is an unnecessary obstacle to getting to the data.
* This property is especially useful in conjunction with dynamically typed data processing languages like Apache Pig [[26](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#ApachePig)]. In Pig, you can just open some Avro files, start analyzing them, and write derived datasets to output files in Avro format without even thinking about schemas.
* Many data systems also implement some kind of proprietary binary encoding for their data. For example, most relational databases have a network protocol over which you can send queries to the database and get back responses. Those protocols are generally specific to a particular database, and the database vendor provides a driver (e.g., using the ODBC or JDBC APIs) that decodes responses from the database’s network protocol into in-memory data structures.

Data flow:

* We have been looking at the different ways encoded data flows from one process to another. So far, we’ve discussed REST and RPC (where one process sends a request over the network to another process and expects a response as quickly as possible), and databases (where one process writes encoded data, and another process reads it again sometime in the future).
* Via databases (see [“Dataflow Through Databases”](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#sec_encoding_dataflow_db))
  + When you deploy a new version of your application (of a server-side application, at least), you may entirely replace the old version with the new version within a few minutes. The same is not true of database contents: the five-year-old data will still be there, in the original encoding, unless you have explicitly rewritten it since then. This observation is sometimes summed up as *data outlives code*.
  + Rewriting (*migrating*) data into a new schema is certainly possible, but it’s an expensive thing to do on a large dataset, so most databases avoid it if possible. Most relational databases allow simple schema changes, such as adding a new column with a null default value, without rewriting existing data.[v](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#idm140417569794096) When an old row is read, the database fills in nulls for any columns that are missing from the encoded data on disk. LinkedIn’s document database Espresso uses Avro for storage, allowing it to use Avro’s schema evolution rules [[23](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Auradkar2015wz)].
* Via service calls (see [“Dataflow Through Services: REST and RPC”](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#sec_encoding_dataflow_rpc))
  + a typical web app server acts as client to a database
  + This way of building applications has traditionally been called a *service-oriented architecture*(SOA), more recently refined and rebranded as *microservices architecture* [[31](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Newman2015wq), [32](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Richardson2014wv)].
  + In some ways, services are similar to databases: they typically allow clients to submit and query data. However, while databases allow arbitrary queries using the query languages we discussed in [Chapter 2](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch02.html#ch_datamodels), services expose an application-specific API that only allows inputs and outputs that are predetermined by the business logic (application code) of the service [[33](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Helland2005tc_ch4)]. This restriction provides a degree of encapsulation: services can impose fine-grained restrictions on what clients can and cannot do.
  + A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable. For example, each service should be owned by one team, and that team should be able to release new versions of the service frequently, without having to coordinate with other teams. In other words, we should expect old and new versions of servers and clients to be running at the same time, and so the data encoding used by servers and clients must be compatible across versions of the service API—precisely what we’ve been talking about in this chapter.
  + Also a ‘web’ service: A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable. For example, each service should be owned by one team, and that team should be able to release new versions of the service frequently, without having to coordinate with other teams. In other words, we should expect old and new versions of servers and clients to be running at the same time, and so the data encoding used by servers and clients must be compatible across versions of the service API—precisely what we’ve been talking about in this chapter.
  + As WSDL is not designed to be human-readable, and as SOAP messages are often too complex to construct manually, users of SOAP rely heavily on tool support, code generation, and IDEs [[38](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Lacey2006ul)]. For users of programming languages that are not supported by SOAP vendors, integration with SOAP services is difficult.
  + Even though SOAP and its various extensions are ostensibly standardized, interoperability between different vendors’ implementations often causes problems [[39](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#Tilkov2006tb)]. For all of these reasons, although SOAP is still used in many large enterprises, it has fallen out of favor in most smaller companies.
  + All of these factors mean that there’s no point trying to make a remote service look too much like a local object in your programming language, because it’s a fundamentally different thing. Part of the appeal of REST is that it doesn’t try to hide the fact that it’s a network protocol (although this doesn’t seem to stop people from building RPC libraries on top of REST).
  + Thrift and Avro come with RPC support included, gRPC is an RPC implementation using Protocol Buffers, Finagle also uses Thrift, and Rest.li uses JSON over HTTP.
  + However, a RESTful API has other significant advantages: it is good for experimentation and debugging (you can simply make requests to it using a web browser or the command-line tool curl, without any code generation or software installation), it is supported by all mainstream programming languages and platforms, and there is a vast ecosystem of tools available (servers, caches, load balancers, proxies, firewalls, monitoring, debugging tools, testing tools, etc.).
  + For these reasons, REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organization, typically within the same datacenter.
  + The backward and forward compatibility properties of an RPC scheme are inherited from whatever encoding it uses:
    - Thrift, gRPC (Protocol Buffers), and Avro RPC can be evolved according to the compatibility rules of the respective encoding format.
    - RESTful APIs most commonly use JSON (without a formally specified schema) for responses, and JSON or URI-encoded/form-encoded request parameters for requests. Adding optional request parameters and adding new fields to response objects are usually considered changes that maintain compatibility.
* Via asynchronous message passing (see [“Message-Passing Dataflow”](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch04.html#sec_encoding_dataflow_msg))
  + Using a message broker has several advantages compared to direct RPC:
    - It can act as a buffer if the recipient is unavailable or overloaded, and thus improve system reliability.
    - It can automatically redeliver messages to a process that has crashed, and thus prevent messages from being lost.
    - It avoids the sender needing to know the IP address and port number of the recipient (which is particularly useful in a cloud deployment where virtual machines often come and go).
    - It allows one message to be sent to several recipients.
    - It logically decouples the sender from the recipient (the sender just publishes messages and doesn’t care who consumes them).
    - However, a difference compared to RPC is that message-passing communication is usually one-way: a sender normally doesn’t expect to receive a reply to its messages. It is possible for a process to send a response, but this would usually be done on a separate channel. This communication pattern is *asynchronous*: the sender doesn’t wait for the message to be delivered, but simply sends it and then forgets about it.
  + Kafka?
    - open source implementations such as RabbitMQ, ActiveMQ, HornetQ, NATS, and Apache Kafka have become popular. We will compare them in more detail in [Chapter 11](https://learning.oreilly.com/library/view/designing-data-intensive-applications/9781491903063/ch11.html#ch_stream).

Replication – ch. 5

* semi-synchronous
* not leaderless
* how often to re-fit/encode? What data does it take
* multiple simultaneous ‘writes’ of new user input preferences?
* High availability: Keeping the system running, even when one machine (or several machines, or an entire datacenter) goes down
* Disconnected operation: Allowing an application to continue working when there is a network interruption
* Latency: Placing data geographically close to users, so that users can interact with it faster
* Scalability: Being able to handle a higher volume of reads than a single machine could handle, by performing reads on replicas

**Section 1: Management Overview and Recommendations**

The task at hand is to define general requirements for the technical system and related business processes for a new movie recommender engine. Ideas for technical solutions that might help fulfill these requirements are also occasionally suggested. However, additional clarity around the product’s needs would be required to align on any final technical solutions for adoption.

To build and support this recommender engine, data should be collected from three sources, and technical solutions to properly store and analyze this data would need to be built and/or bought. Components would also have to be built that would allow users to interact with our product by submitting their own movie preferences, requesting recommendations from our engine, having their data applied to our model to generate predictions, and receiving results.

The next section goes into more detail about the data that could be extracted from each of the data sources, and our needs and initial ideas for storing the data. The following section covers our needs and initial suggestions for tools to use in analyzing the data, fitting models, and producing user-specific recommendations. The last section touches on our relative computing needs to support the above functions and potential solutions for those computing needs.

**Section 2: Data and Database Software**

* Continue to answer functional design questions: What should the system do? How can it best be defined to serve movie consumers? How shall data be stored, searched, and processed?
* includes specification of server-side architecture and software systems.
* Specify components for databases and analytics software and hardware systems satisfying the following:
  + Open source software and systems.
  + Work from complete IMDb and MovieLens.org databases, updated at least every 10 days.
  + Relational, graph, network, or other data model (justify choice).
  + Train neural network autoencoders.
  + Support initial system prototype, but also scalable to support hundreds of thousands of users in production.

The movie recommender engine currently requires input data from three sources: The Internet Movie Database (IMDb) owned by Amazon, MovieLens.org hosted by GroupLens Research ([GroupLens](#grouplens)), a group of students, staff, and visitors engaged in social computing research at the University of Minnesota, and users of the recommender engine themselves.

Section 2.1: IMDb Data

While IMDb does not provide an API for public queries, it offers seven daily updated downloadable compressed tab-separated UTF-8 encoded plain text files that contain much of the data found on IMDb via this link: <https://datasets.imdbws.com/>. The data is listed as available for personal and non-commercial uses only per their Non-Commercial Licensing (“[Can I Use IMDb Data in My Software?](#imdbdatause)”). Immediate research is required to determine if we can publish modeling results leveraging the data. If we eventually intend to monetize our recommender engine, we must first determine the viability of Commercial licensing with IMDb ([“Content Licensing”](#imdblicensing)). If Commercial licensing is an option, technical and legal resources will be required to coordinate with IMDb to develop and maintain a relationship to ensure continued use of their data.

We will need to explore and manipulate the IMDb data at will and feed it into our analytics tooling of choice for Neural Network encoding. As such the data should be stored in-house. Along with command-line interface tools and a Java-based GUI that allows searches and displays of the information provided, IMDb offers a Python package, “IMDbPY”, that can process and load the files into a number of different SQL-based databases. Thus, an initial assessment suggests the IMDbPY package may be well suited for accessing this data and feeding it into a SQL-based database.

Our system should also allow for regularly scheduled pulls of new versions of the files on IMDb and batch processing to load them into the database and/or data warehousing solutions we choose. The appropriate cadence of these data pulls will be highly dependent on the performance of the recommender results, both in terms of model accuracy and in terms of user affinity. As such, the precise cadence of IMDb data acquisition may change over time and should be reassessed periodically leveraging input from the modelers as well as business decision makers.

A technical resource[[1]](#footnote-1) should be put in charge of developing, executing, and monitoring data acquisition, basic pre-processing, and batch loading into data storage. This resource must easily communicate with those managing any IMDb relationship *and* with the modelers to ensure that expectations regarding the data are in alignment.

Section 2.2: MovieLens.org Data

GroupLens offers downloads of 7 UTF-8 encoded csv structured zipped files sampling the

MovieLens data. Some of these datasets are static and others seems to be periodically, though infrequently, updated. Each dataset comes with a README file outlining proper uses of the data. Specifically, “the user may not use this information for any commercial or revenue-bearing purposes without first obtaining permission from a faculty member of the GroupLens Research Project.” (“[GroupLen ml-latest-small-README](#readme)”) If we eventually intend to monetize our recommender engine, immediate research is required to assess the viability of using the MovieLens.org data. If viable, resources will again be required to coordinate with GroupLens to develop and maintain a relationship to ensure continued use of their data.

Similar to the system requirements for use of the IMDb data, our data-flow system must be able

to retrieve the MovieLens files and load the data into storage. Again, we likely want our system to allow for periodic pulls of new versions of the some of the files provided by GroupLens and batch processing to load them into the database and/or data warehousing solutions we choose. However, in this case, we can likely perform these data acquisition tasks at more infrequent intervals as compared with the IMDb data acquisition schedule. A technical resource will have to developing, executing, and monitoring MovieLens.org data acquisition, basic pre-processing, and batch loading into storage. This resource could easily be the same resource handling the IMDb data acquisition and also, should have direct lines of communication with technical resources at GroupLens. Since it is a small research organization, developing the relationship with GroupLens will likely be far less demanding than doing so with IMDb, a subsidiary of one of the most powerful companies in the world.

Section 2.3: User Input Data

The last data source required for the recommender engine is input from users. For a prototype, any API can serve this function. To make the product user friendly (and to leverage previous user movie preferences in model validation and periodic re-training), we should update existing users’ movie preferences every time input is received. Thus, user input should be delivered to a server that will load the data into storage for future use.

The last data input to create the personalized movie recommender engine is input from users. Any API can serve this function. However, movie recommender engines have broad potential appeal among anyone who watches movies. Since most people watch movies, and most have developed strong preferences for modern well-designed visually enticing websites, depending on our desired use case of the recommender engine, it is likely we would develop a website to handle user requests, as well a recommendation output.

Regardless of the front-end that users will interact with, ultimately, information should be sent to a server resource that we manage. Until we more fully develop the business use case for the recommender engine, we should assume that inputs could arrive in the form of single movie preferences or a bulk set of movie preferences.

A technical resource should develop and monitor the functionality of the user input API ensuring the condition of the data arriving at our server is as expected and that it is successful delivered into data storage. (ALT): whatever front-end we develop for the user input API. In addition, a (potentially) different resource or set of resources should be responsible for monitoring the data arriving at our server and its successful delivery into our databases and/or data warehouses in the proper format. This resource should have direct lines of communication with the modelers. Ideally, the API should have some data checks in place and be able to automatically abandon data ingestion and produce appropriate error messages to relieve burden on our technical resources and the rest of our back-end system when faulty data is submitted.

Section 2.4: Storage Systems

Most of our input data is already structured in relational-friendly tables, so a relational (SQL) data model is a compelling option for storage needs. The exact database to use depends on existing infrastructure, technical expertise required and available, and cost. SQLite, a free serverless (cloud-based) database, would keep costs low assuming it can handle our load appropriately. Since the volume of writes will likely be higher than the volume of reads, another option is MySQL which offers a log-structured (LSM) alternative to its default B-tree solution. MySQL would give us the freedom to test LSM versus B-tree indexing within the confines of a relational. Many engineers are familiar with SQLite and MySQL making maintainability easier to fulfill.

A potential exception to the single relational data model would be the user input data. While storing user preference alongside preferences from our other data sources in a relational database, additionally storing user data in a document data model with each document representing an individual user would enable optimizations from locality when producing recommendations for a user. However, we would need to conduct a cost-benefit analysis once more architecture decisions have been made to determine if the added benefit of locality is worth costs including any required licensing as well as resource costs for monitoring and maintenance. If we chose to supplement the relational data model with document-based storage, we may also choose to store the parameters of all previous model formulas in the document storage system as well, in the rarer cases reversions or investigations are required.

Regardless of the products used, a database administrator will need to monitor database software, track issues, instantiate data checks, ensure that any changes to the structure of incoming data can be handled appropriately, and monitor read/write volume and performance to ensure database tuning parameters are set properly.

users

To do this, we would require a unique identifier for the user ideally based on an email account but potentially based on an IP address if we do not want to require users to create accounts with us. In addition, users should input user-friendly movie identification information, such as title and release year, as well as some sort of preference rating, the nature of which will depend on how we choose to encode final inputs into our modeling.

Note database systems that will likely be used for implementing the application or system.

* store movie encodings in database
  + leverage search indexes - hashing
  + anything to cache?
* duplicate imp data on multiple systems to account for crashes
  + especially previous user inputs
  + can always re-pull from imdb, etc.
  + + software fault-tolerance techniques
* Diff data
  + 1 table with movie id & movie info (title + year?)
  + 1 table with training preference data??: user type??? move ids? Ratings?
  + 1 table with user preferences to predict?: user id, movie Id, rating
    - Else document for each user with their preferences – updated via web-api. LOCALITY.
      * Relational dbs that support: postgresql, mysql
  + Sql lends itself to parallism
* Indexing = yes since reads more and more imp than writes
  + Bloom filters for movie lookup prior to the imdb update every 10 days?
* Cassandra? (nosql)
* Or….. etl into data warehouse? Maybe not.
  + Sap hana (optimized for both storage and warehousing) for new incoming preferences?
  + More recently, a plethora of open source SQL-on-Hadoop projects have emerged; they are young but aiming to compete with commercial data warehouse systems. These include Apache Hive, Spark SQL, Cloudera Impala, Facebook Presto, Apache Tajo, and Apache Drill

**Section 3: Analytics and Modeling Software**

Normally, data warehousing in addition to data storage can provide valuable optimizations for an analytics product. However, in this case, further evaluation is needed. To *develop* the recommender engine, analysis would require large infrequent queries of the data, an ideal use case for data warehousing. The IMDb data is particularly predisposed to a star-schema structure. As most data warehousing solutions are not cheap, several unknowns -- including the size of queries for analysis, their frequency, our database read performance, and the impact that slow database read times have on engine development -- need answering before deciding to purchase a data warehouse solution.

However, the most frequently performed analytics for the recommender engine post-launch would involve model predictions from a pre-fitted model, requiring only the pre-fitted model parameters, and a single user’s input data to produce the model output. Since standard data warehousing solutions are not cheap, this use case would not warrant data warehousing on its own. The data being queried for the recommender engine would have no place in any consistent dashboard reporting for frequently used BI tools, a use case that data warehousing is built to support. The only case for consistent BI reporting post-launch would be on model performance and maybe load volume to assess growing interest in the new produce. As such, a more formal assessment of the costs and benefits of data warehousing solutions as we learn more specifics about the load and expected future load of incoming data, the size of queries for analytics purposes, their frequency, our database read performance, and the impact that slow database query performance has on the progress of the recommender engine development, launch, and adoption should be conducted before deciding to purchase a data warehousing solution.

To execute analyses such as model fitting and prediction generation, other analytical tooling is required. Python is free, well known to engineers and analysists alike, and may be used in other areas of the data pipeline. To leverage Python, or any scripting language that supports neural networks, whenever a recommendation request is submitted, the back-end server would process any new data submitted with the request, consolidate that data with the user’s existing data sourced from storage, and run a Python script leveraging the fitted model (also from data storage) on that data. There are also several options for Python development environments for data exploration and model fitting, such as Jupyter Notebooks.

The modelers should be in charge of conducting model validation and tuning to ensure optimal accuracy of the engine (ALT:) periodically monitoring recent incoming data for statistical consistency and conducting frequent model validation and tuning to ensure our recommendation engines continually maintains or increases its efficacy. This resource should have direct communication with decision makers in case of model failure or required changes to the prototype.

* Should be able it identify when user asks for recommendation with a LOT of previous preference inputs. May take up more resources so potential alternate processes/resources should be leveraged upon large model identification
* Prepare for faulty input data and produce appropriate ‘error’ messages when this happens
* Some way of confirming that each request is dealt with and output is sent
* Separate (copied) sandbox environment for periodic (and frequent) model testing & evolution
* Phased roll-outs of new models
* Monitoring of inputs/output
  + Business processes for maintenance checks, quick escalation, easy methods for roll-backs/fixes

**Section 4: Computing and Communications Systems**

The computing infrastructure for this project will have to support data storage (and maybe warehousing), ETL process flows, model development, web hosting (most likely), and model predictions. With over 600,000 to 700,000 movie titles on IMDb plus upwards of 20 million movie ratings applied to 27,000 movies in a single file from MovieLens, storing and exploring the data, and fitting neural network models to it will take a great deal of computing power, requiring distributed and/or cloud computing. Since neural network fitting is notoriously slow but will not be a continuous process, the elasticity of a cloud system would allow us to dynamically and efficiently scale computing to accommodate model fitting when it is being conducted. To guide our understanding of the computing power required for prediction generation (post model fitting), the average response time for delivering recommendations upon user request for the prototype should be less than 1 second.

load estimation will not be exact and load may be highly unpredictable upon product launch. An elastic cloud computing infrastructure will allow us to respond to various load more quickly and efficiently.

Requesting additional information on the resources required for our model fitting and estimating load (and the growth of load) in request per minute from users, as well as the ratio of reads to writes of a production system will be key in determining the precise specifications of any cloud computing or distributed system. In addition, frequently monitoring throughput will allow us to prepare for growth in the product and anticipate and remedy scaling challenges before they impact the system.

Specify the computing infrastructure, resource requirements (in terms of processing power and memory?)

information systems and connections between systems.

Content Notes:

* reliability - must be able to tolerate faults (hardware, software, human)
  + not a finance thing, enjoyment vs necessity so the world will still turn if the system fails, but at the same time, the more it fails, the less users will like, use, and recommend our new product
* scalability - handling load, performant
* maintainability - should be easy/simple to evolve
* batch processing of imdb updates
* stream processing of new user movie preferences from MovieLens.org?
* api + queues for new recommendation requests
* break down that application into what types of tools to use for each step/part, must be used in combination with each other
* automated & comprehensive testing
* phased rollout for launch of product
* save old files in case we need to roll-back newer modeling version
* load
  + requests per second from client to a web server
  + ratio of reads to writes in a database
* what is the throughput—the number of records we can process per second, or the total time it takes to run a job on a dataset of a certain size
* Use systems and languages that are known by many to foster maintainability.
  + Operations team to monitor health, track down issues, keep software/platforms updated, anticipating changes to movie/preference db sources, setting up processes and preserving knowledge
  + Want data systems that will provide visibility into load, runtime, support automation – this use case is pretty perfect for automation,
  + Keep documentation up to date and clean + consistent terminology and naming
  + Limit special casing, abstraction, sql, python for modeling

Works Cited

“Can I Use IMDb Data in My Software?” IMDb, IMDb.com, help.imdb.com/article/imdb/general-information/can-i-use-imdb-data-in-my-software/G5JTRESSHJBBHTGX?pf\_rd\_m=A2FGELUUNOQJNL&pf\_rd\_p=3aefe545-f8d3-4562-976a-e5eb47d1bb18&pf\_rd\_r=A4RHXCC91B96D7392C05&pf\_rd\_s=center-1&pf\_rd\_t=60601&pf\_rd\_i=interfaces&ref\_=fea\_mn\_lk1#.

Choudhury, Sid. “A Busy Developer's Guide to Database Storage Engines - The Basics.” YugaByte DB, 15 Aug. 2018, blog.yugabyte.com/a-busy-developers-guide-to-database-storage-engines-the-basics/.

“Content Licensing.” IMDb, IMDb.com, www.imdb.com/licensing/?ref\_=helpms\_ih\_gi\_license.

Designing Data-Intensive Applications: the Big Ideas behind Reliable, Scalable, and Maintainable Systems.”  Martin Kleppmann, O'Reilly Media, 2018, pp. 3–107.

“GroupLens.” GroupLens, grouplens.org/.

GroupLens. "ml-latest-small-README."  *MovieLens Datasets*. http://files.grouplens.org/datasets/movielens/ml-latest-small-README.html.

Hooks, Ivy. Requirements Experts, reqexperts.com/resources/requirements-articles/articles-what-is-the-difference/.

“IMDb Datasets.” IMDb, IMDb.com, www.imdb.com/interfaces/.

“MovieLens.” GroupLens, 14 Jan. 2019, grouplens.org/datasets/movielens/.

1. Resource may always refer to a single individual or a team of people [↑](#footnote-ref-1)