Data Engineering Assessment

[redacted]

[redacted]

1 Database Design for Hockey Analytics

1. How would you gather, store, and update information from the EHP API?

This is a great use case for Extract, Transform, Load (ETL) pipelines. ETL pipelines split the responsibilities for extracting data from external systems, transforming that data into a standardized format, and loading the data into your internal systems into distinct pipeline stages. The pipeline structure is useful because the stages are dependent on each other and must be executed in a predefined order - if the transformation stage fails we need to skip loading that data into our internal system.

I would build independent services for each of these stages - and potentially for each external system. This architecture is commonly referred to as “micro-services” and is beneficial because many small systems with clear boundaries and consistent contracts are much easier to build, reason about, test, scale, and replace than one large monolithic system. Another benefit is that it allows you to use the correct tool for each stage - even if those tools vary drastically.

The extract stage is responsible for extracting data from external systems, by whichever means are necessary. I would build an extract service in Java or another object-oriented strongly typed programming language. The schema of the responses from each external system are going to be unique and fairly complex. They are also likely to change over time. The benefit of using an object-oriented strongly typed language for this stage is that our classes and objects will map 1-to-1 to each entity, and their structure is fixed at the time of instantiation. Java is commonly used for enterprise applications and has ample library support, such as the Spring framework, to assist in building out a scalable system that interacts with many APIs. We can pull in Spring libraries to fetch and store access tokens, make requests to external APIs, and assist with the development of integration tests. If we choose to build out a separate extract service for each external system, we could build out a set of reusable tooling and common functionality that can be pulled in as a common dependency for every service.

The transform stage is responsible for transforming the data from the external representations into the standard internal representation we have defined. For this stage, I would build a new service for each external system. I would either choose to use Java for these services so that we can re-use the classes from the extract service, or I would choose Python in order to leverage it’s excellent data manipulation and validation tools. We can define a REST API controller for these services to act as our standard contract, and then operate on all data in a JSON or other flexible format inside the service. The transform stage will rely heavily on reading from our internal databases in order to restructure the external data for use internally. Whichever tool we choose for this job will need to be able to perform many database queries in a performant way. I anticipate that this service will need to be changed and iterated on more than the other services, so it would be beneficial to build extensive unit and integration tests for all features.

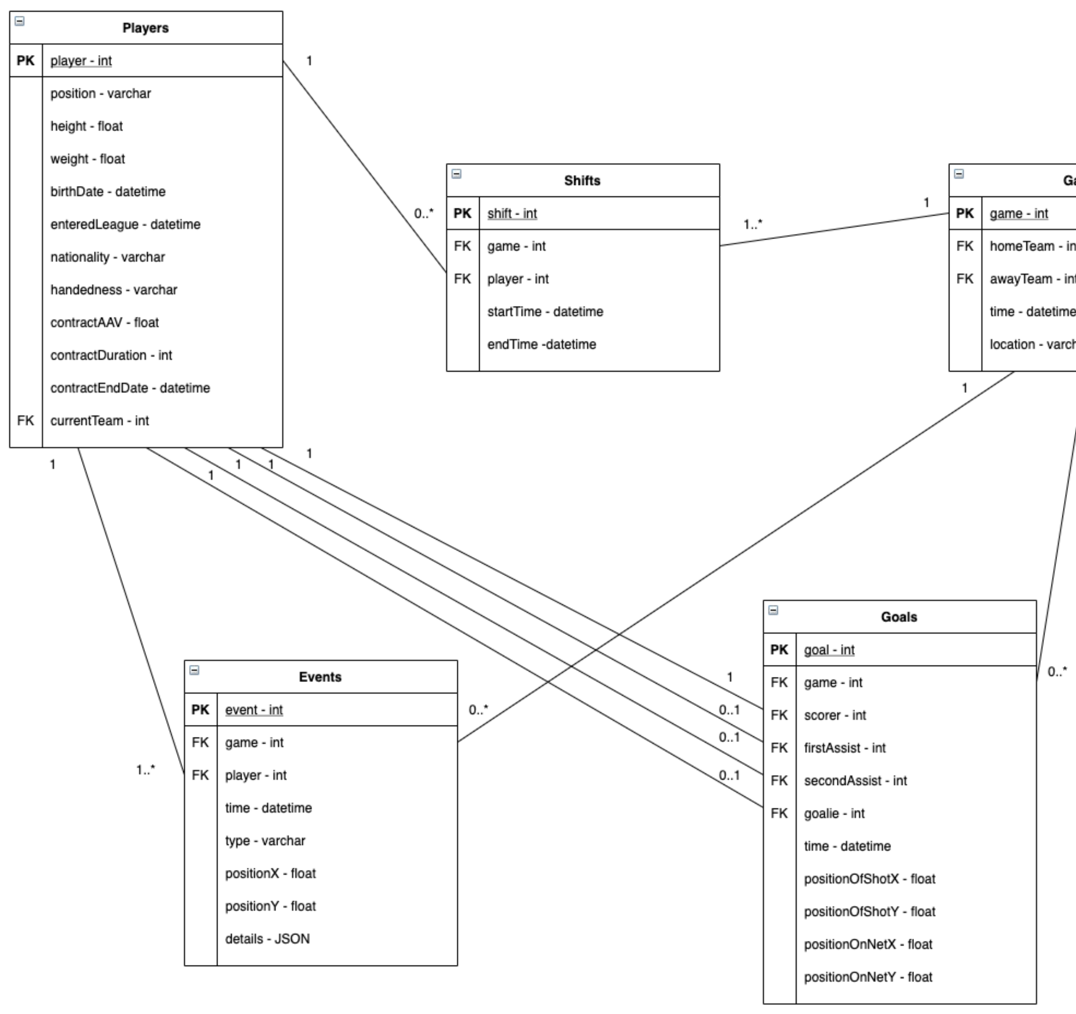
The load stage needs to take the standardized data and load it into our internal databases. I would build a single load service capable of taking the output from the transform stage and inserting it into or updating the correct database tables. I would probably use Java for this service and an ORM tool like Hibernate to provide performant, predictable, and convenient access to our databases. This stage needs to be especially well-tested and monitored because it is the only stage capable of creating and mutating data in our internal database. We should follow defensive programming practices to guarantee that any problematic data coming from the transform phase is properly identified and rejected. Any bug here will manifest in problems for downstream users.

Finally, we need to stitch together all of these services using a pipelining tool. There are many, many possible technologies we could use, depending on how flexible we want the pipeline to be, and how much control we want over it's operation. A fully managed pipeline tool built for ETL pipelines, such as AWS Data Pipeline is a good first choice because it will be reliable, easy to configure, and feature rich.

Due to the scale of this problem, I would advocate for using cloud-based services to run these pipelines so that we are flexible to scale the underlying hardware, quickly iterate on our implementations, and deploy new versions of our software as needed.

2. What data tables do you think would be useful for hockey analytics purposes? Pick 2-5 potential data tables that you would recommend including in an internal version of the EHP database. Describe the information in each table. Describe your proposed schema for the tables in this database.

The 5 core tables I would recommend using for an internal hockey analytics database are players, games, shifts, events, and goals. An entity relationship diagram is included below. Each box represents a table, with PK and FK designating primary and foreign keys, and lines showing the multiplicity of relationships between the tables.



<u>Players</u>
The players table includes information about each player's physical characteristics such as height and weight, background such as nationality and birthdate, and contract information such as the team they are under contract with, the current contract AAV, and the contract end date. The players table can be used to attribute goals and other on-ice events to a specific player, and can be used to make comparisons between players based on physical traits and contract terms. Each player will be represented by exactly one row.

<u>Games</u>
The games table has basic information about each game including the home team, away team, time of game, and the location. Each game will be represented by exactly one row.

<u>Shifts</u>
The shifts table is responsible for keeping track of when each player is on the ice. It includes columns for the game, player, and start and end times. This table will be used to calculate

summary metrics for a player like time on ice, average shift length, and average shifts per game. It can be used to understand how players perform when on the ice together to influence future line composition decisions. It will also be used to understand how a player performs when they are not directly involved in the play, as we will know which players were on the ice at the time of each goal, penalty, takeaway, etc. Each row in the table represents exactly one shift for a single player in a single game. For a single game, the shifts table will have many rows representing each shift by each player on both teams.

Goals

The goals table contains information about every goal scored in every game. This table has columns for the game, players credited with the goal, first assist, and second assist, time of the goal in the game, and positional information about where the shot was taken on the ice and where the shot entered the goal. This is an important table because we need to know who was involved in every scoring play and where that play took place. The game and time columns in this table can be compared with the game and time columns in the shifts table to understand which players were on the ice for both teams when the goal was scored. One row represents one goal in a given game.

Events

The events table is intended to capture information about all non-scoring events that we care about. It has columns for the game, time, player involved, and position on the ice - similar to the other tables. It also has important type and details columns intended to distinguish exactly what this event was and other important details about it. Some example event types include “shot”, “giveaway”, “takeaway”, “injury”, “penalty”, “hit”, “faceoff win”, and “broken stick”. The details column is JSON in order for it to be flexible depending on the event type. For a penalty, we may want to record the duration of the penalty and the affected player, whereas for an injury we may wish to record which body part was injured and if the injury was caused by contact. Each row represents a single event that we care about in a specific game, so each game will likely have tens-to-hundreds of events.

Conclusion

I believe this simple schema would enable basic analysis about the most critical events in hockey - goals. However, a real analytics system would need many more tables and relationships to be useful for an NHL team.

This schema has some obvious limitations. In practice, a single events table is a bad idea because its rigid schema means that we will be recording data that we wish to query later in the JSON details column. JSON columns are inefficient and hard to query. This table is also likely to be much, much larger than the rest of the tables which will result in worse query performance as our dataset grows.

Another limitation is the lack of representation for data about player and puck position on ice over the full duration of the game. We have no way of evaluating the effectiveness of a specific defensive strategy, tracking how often a player is generating scoring chances without scoring, or picking up on a goalie’s tendencies that we may be able to exploit. It would be incredibly valuable to gather all of this data from game video or another source, but we have no way of storing it with this schema.

The choice of database system for storing this data would depend greatly on the expected use cases, amount of data to be stored, and performance requirements. My initial recommendation would be to use a well-supported Relational Database Management System (RDBMS) designed for Online Analytical Processing (OLAP) applications. PostgreSQL is one popular option that I’m familiar with and is supported by all major cloud vendors.

3. How would you deal with the name-standardization and data linkage issues (so that all player information can be quickly linked across data sources)?

I would split this problem into 2 subproblems - first time linkage, and subsequent data pulls. The solution proposed below can be generalized to work for any system, but I will focus on the EHP API.

When we pull data from the EHP API for a given player for the first time, we need to find this player in our database. I would build a matching service to handle this problem that uses as much data as we can from the external system so that we can determine which internal player this data belongs to with some level of certainty. This matching service may need to be unique for each external system, depending on how much the response fields vary from system-to-system. Some example criteria I would use in this matching service for the EHP API are:

- Fuzzy match on the player’s **firstName**, **lastName, alternativeFirstName,** and **alternativeLastname** using a library such as FuzzyWuzzy. This fuzzy match algorithm should be fairly complex in order for it to handle nicknames, differing translations, and typos.
- Exact match on **birthdate**
- Exact match on **nationality**
- Approximate match on **height** (+/- 2 inches)
- Approximate match on **weight** (+/- 15 pounds)
- Exact match on **nhlRights**

I would also generate a confidence level for each match using a heuristic. A player that matches on first name, last name, birthdate, and nationality should be close to a 100% confidence level whereas a player that only matches on birthdate and nationality should be much lower. I would set thresholds for how to handle the matches. If the confidence level is high, we can automatically link the players as described below. If the confidence level is below an acceptable threshold, I would notify a human to complete the verification process and approve the match before the players are linked.

Once we confidently know which internal player this data belongs to, we should store the unique identifier from the external system in our internal system. All mature APIs should provide end users with a unique identifier for all entities. The unique identifier is usually either a Universally Unique Identifier string (UUID) or an integer for a database primary key. We should maintain an internal table in our database that maps the combination of an external system identifier and the external unique identifier for the player to our internal unique identifier for the player.

For the EHP API, all responses for player data contain the player's ID. The Sabre's internal database table for **externalUsers** could look something like the table below:

externalPlayerId	**externalSystem**	**externalId**	**sabresId**
1	EHP API	30900	80
2	EHP API	901100	5
3	NHL	47dcaea1-13a1-4f43-843b-ebf61172ab89	9
4	NHL	9d2fcfee-b7f6-448f-8a55-016849886ec8	80

For all future interactions with the EHP API, we should ignore the player name in the response and instead use the **id** field and the **external system** identifier to find the **sabresId** for this player. Once we have the **sabresId,** we know which player this data belongs to and can update our database tables as needed with confidence.

4. How would you design this system to handle the streaming nature of the data (i.e. updating databases to include the most recent player statistics, handling new players in each underlying data source, etc)?

All mature APIs should have a simple way for users to pull only the data that has changed since the last time that user interacted with the API. For the EHP API, this is implemented as an **updatedAt** query parameter. I would set up a task to run the ETL pipeline proposed above on a regular schedule, such as every night at 2am, using a CRON job or similar tool.

If we are able to successfully pull data from the EHP API, I would record the current date and time in a table in our database. On the next execution of this scheduled job, it will query the database to find the last update timestamp, and then pass that along to the EHP API on every request as the updatedAt query parameter valu. This will allow us to receive all new entities as well as updates to entities in the EHP API without doing any sorting or filtering on our end. In general, if an API allows date based filtering we should rely on that functionality instead of building it ourselves.

In the Extract stage of the pipeline, I would be sure to pull data in a consistent order each time, depending on the relationships for the data we pull from the EHP API. For example, playerStats references a player entity so I would be sure to pull player data before playerStats data. This will guarantee that our internal entities are updated in the required order. I would store this unique entity order as a tree data structure for each system, so that when we need to pull new data, we can simply traverse the tree in a breadth-first order and query the API for the entity at each node.

5. What other issues do you expect to encounter along the way here? What measures would you put in place to prepare for these potential obstacles (and/or other unexpected issues), to ensure that there no downstream issues in our hockey research systems?

The two major issues I would expect to encounter are data integrity and system health.

As we are ingesting large amounts of data from external systems, we should be careful to protect the quality of the data in our internal system. Since we do not control these external systems, we have no way to be sure that the data we're retrieving from them is valid. What happens if a developer for the EHP API pushes out buggy code that assigns goals to the wrong team? We need to be defensive of our internal systems by catching these error scenarios and preventing the bad data from entering our internal system.

I would add a step in the ETL pipeline for data validation - creating an ETVL pipeline. The pipeline can (E)xtract data from the external system, (T)ransform the data into our internal representation, (V)alidate the integrity of the data, and then finally (L)oad the data into our internal database if the validation passes. The validation step may be different for each data source. For example, if we are pulling shift information about each player in a game from one system, we may choose to validate that each player actually plays for the team that the shift was assigned to by using our internal players table, or by using another external system. This validation process should err on the side of rejecting data if there is any question about it's integrity - at which point it can notify a human for further analysis. If we err on the side of accepting the data, then we may not catch an issue before it's been used downstream in our analytics processes.

We also need to make sure our ET(V)L pipelines are functioning as expected at all times, or we risk losing data or delaying downstream usages. If a server crashes, a database becomes unreachable, or a developer pushes out broken code, we need a way to identify the scenario and recover gracefully. If these systems are built out in a cloud provider like AWS, we can easily add alarms, monitoring, logging, and notifications so that we can attempt to recover from any event in a graceful way, and alert a human if the system can't recover to a healthy state. We should also follow modern software development practices by writing unit and integration tests for all features, running those tests on all new code changes, and deploying all new changes to a protected sandbox environment before deploying to a live production environment. In this way, we can ensure that the downstream analytics tools are unaffected by any problems to earlier parts of the pipeline.

2 Cloud vs. On Premise

Would you recommend housing our internal databases (and data analysis systems) in the cloud or on premise? What are the pros and cons of each approach?

The decision of cloud vs on premise is important to get right the first time. The cost (real and opportunity) of correcting the wrong decision could be massive. The criteria I would use to evaluate these options are <u>functionality</u>, <u>difficulty of operation</u>, <u>availability</u>, <u>security</u>, and <u>disaster recovery</u> in the context of both databases and data analysis servers. In the comparison below, I will use AWS as a reference cloud provider because it's the vendor I have experience with, but the other major cloud vendors have similar features so the general arguments hold regardless of vendor choice.

<u>Functionality</u>
The most important consideration for this decision comes down to functionality. Are we able to build the services we need on-premise? What about in the cloud?

On-premise installations have the ultimate flexibility. It's possible to choose any hardware we wish, and we have full access to that hardware to install or build software services. We are in full control, so if it can be built, we can build it. However, this flexibility comes at a cost. I'm fully responsible for building the functionality - nothing comes for free. If we want extensive logging and monitoring, we need to build it or find and install an existing service on our own. If something breaks along the way, it's on us to fix.

Cloud vendors like AWS offer nearly the same flexibility while freeing us from the burden of managing all of the individual components. AWS has a remarkably diverse set of hardware configurations we can choose from - from the physical servers running our services, to the network load balancers routing all of the network traffic, to the operating system image (AMI) running on each server. We will likely be able to find a combination of hardware and operating system that will work for any service we wish to run.

Cloud vendors offer additional functionality by providing fully-managed services in addition to simple servers. For example, in AWS we could choose to run our databases on Elastic Compute Cloud (EC2) servers which are similar to an on-premise server. We will have root level access to these servers and can install whatever software services we need to run. A better option might be to use the managed database service - AWS Relational Database Service (RDS). RDS trades flexibility for functionality and ease of use. We are able to choose the database vendor we wish to use and the hardware configuration we need from a set of preconfigured options. While this limits our initial decisions somewhat, we will benefit greatly from the features of this managed service. RDS provides automated backups, extensive monitoring, replication services, and data encryption features for free. All of these can be configured with a few clicks in the AWS console and can be changed at any time to fit our needs.

AWS does a great job of providing enough flexibility in hardware and software choices for us to be confident that we can run any service we need. The managed services in particular are extremely valuable because they come standard with common cross-cutting features. This allows us to quickly build, deploy, and iterate on new services.

Difficulty of Operation

Building a data center on-premise capable of running analytics services is no easy task. It will require large amounts of human labor by a team with many areas of specialty. Once the data center is built, it'll need continuous maintenance and periodic upgrades. If anything goes wrong at any point in time, we're responsible for fixing it.

To build an on-premise data center we would need network experts to build out a robust intranet, data center experts to ensure our servers are properly installed, powered, and cooled, hardware experts to choose the correct and compatible hardware, and physical security experts to make sure nobody can walk away with one of our hard drives.

By contract, cloud vendors handle all of that for us, freeing us to focus only on configuration and the software we are running. One major benefit of this is that the engineers building services are capable of deploying and operating those services themselves. The experts on the software and use cases can take full responsibility of their services without needing to rely on data center technicians or network experts to implement their desired changes.

Lastly, scaling an on-premise operation is a massive undertaking. If we need to double the RAM on all of our servers, someone needs to order the RAM, replace the physical hardware, and reboot all services without any interruption. Cloud providers excel at scaling dynamically. In fact, they are so good at this that we can choose to scale our services up and down on a regular schedule in order to control costs while meeting peak demand. If we expect this operation to grow and mature overtime, the scalability provided by cloud vendors should be highly valued.

Availability

What good is it to build databases and analysis services, if they are unavailable when you need them? A good set of analytics tools should always be available for the end-users - regardless of the time of day or location of the user.

Building highly available systems on-premise is not easy. It takes an expert in network configuration, and data center operations to build highly available systems. Some strategies include duplication of services in a hot/hot or hot/cold configuration which will allow the services to be available in the event of planned or unplanned downtime. Having duplicate services is required, but how do you know when the services need to failover to the duplicate? In order to ensure services are always available, we need to build out monitoring to detect when services are unreachable and automate the failover process. These processes need to be well-tested so that we can be confident that they will work when needed.

Another problem is the availability of our services across the globe. If we have scouts working in Russia, will the services be available for them? And if they are available, will the latency be low enough to actually use the services effectively? An on-premise solution by definition is in a single location - which makes this problem unavoidable.

Cloud vendors have unparalleled availability. The uptime for individual services is normally at least “Five 9s” - or 99.99999%. Every cloud vendor has built-in support for hot/hot and hot/cold deployments and has simple ways to configure the failover process. In AWS, an admin can configure a health check for each service with a few clicks and can then instruct AWS to failover to a specific backup service in the event that the health check fails. This is all highly customizable and can be configured by a software engineer familiar with the services, without the need to understand the exact network and data center operations required to make this happen.

Cloud vendors also have physical data centers all over the globe. We can configure all of our services to be deployed to multiple physical data centers in different geographic regions to ensure that our services will be available and performat for users even if there is a problem with one of the data centers. If we have a need to have low-latency and highly available systems for users in Russia, we can simply deploy a duplicate set of these services to a physical data center in Russia with a few clicks. The scale of cloud vendors allows us to build highly available services with much less labor and almost certainly better performance than on-premise.

Security

Data security should be a major concern for an NHL team. The data we will be collecting is most valuable if it’s only available to us. There are many layers to data security, from physical security to access-level controls.

On-premise implementations benefit from having physical security - the physical hard drives and servers with access to the data can be stored in a secure location on-site. With an on-premise installation, we can also prevent any data from ever leaving our trusted intranet, meaning we will not be at risk for man-in-the-middle or other network based attacks.

On the other hand, cloud vendors like AWS will never give you physical access to the devices your data is stored on, and by the nature of the cloud that data will need to be transmitted over the internet for our use. However, both of these issues can be mitigated quite easily. In AWS, you can require all communication internal to your cloud network and external to use HTTPS with a few button clicks, largely mitigating the risk of any network based attack. In addition, we can easily opt-in to encrypting our data at rest so that if any were to ever get access to one of these data centers, our data would be useless without the private key that only we know.

AWS really shines with the ease of configuration. They make it easy to have a secure implementation by allowing users to configure security settings, removing the need to configure all levels of security controls like in an on-premise installation. For example, access-level controls using the Identity and Access Management Service (IAM) are simple and easy to configure, and allow an administrator to restrict access to a given database, server, or network configuration tools to only trusted users. With a few clicks in the AWS console, an admin can require 2-factor authentication for access to the AWS console, require secure SSH keys to be used to access any server, and prevent any user from ever having ssh access to our database servers.

Another area of concern is security patches. What happens when it's discovered that the database system we are using on-premise has a potential attack vector that must be fixed with a software patch? If we're lucky, we may be notified to apply this patch and then we'll need to actually apply the patch ourselves. With AWS, we can configure a maintenance window and opt-in to all security patches. In this situation, we can be sure that we are staying secure without any additional manpower.

Lastly, AWS maintains access and change logs for all services - something that would require a tremendous amount of work to build for an on-premise installation. In the event of a security breach, we will know information about the user that accessed our resources, and which resources they had access to. By following best practices in AWS, such as configuring unique virtual private clouds (VPC) for each unique service, we can further limit the blast radius that any one security breach can have.

Disaster Recovery

Disaster recovery is often overlooked but should be one of the most important considerations when evaluating on-premise and cloud offerings. The data we collect is incredibly valuable and needs to be long-lived. Disasters happen - whether it's a natural disaster like an ice storm taking out Buffalo, or a catastrophic hardware failure in the data center. We need to be resilient to these disasters in order to avoid catastrophic loss of data, human time, and money.

On-premise installations need to spend a great deal of time ensuring they can recover from any type of disaster. Processes need to be put into place to make physical copies of all data and store those in many different geographic locations. Systems need to be built to automate the generation of backups, and need to be regularly tested to ensure that the data can be recovered in the event of a disaster. This work should be done by an expert - as a failure in your disaster recovery tools can be very costly.

Cloud vendors like AWS can use their massive scale to provide effective and easy to use disaster recovery mechanisms. From the AWS console, we can choose to create new snapshots of our databases every day or even every hour. By default, these backups are stored in multiple physical locations (availability zones). Recovering from a disaster is as simple as instructing AWS to recreate our databases from the latest snapshot. This process can even be automated by configuring alarms to automatically fail-over to a different database in a different availability zone without any human intervention.

When it comes to disaster recovery, on-premise installations will never be able to match the capabilities of the large cloud providers.

Conclusion

I would recommend using cloud over on-premise implementations because the cloud services will require far less human labor and cost less to build functional, highly-available, secure, and fault-tolerant systems. It is feasible for a small team of software engineers to build and maintain complex systems in a cloud environment while an on-premise installation will require dedicated data center experts. In addition, cloud vendors free us from the need to purchase and configure hardware for our services which allows us the flexibility to scale, change services, and experiment with different hardware configurations quickly and cheaply. All cloud-based services are designed, built, tested, and maintained by domain specific experts. We should trust that AWS's data encryption processes will be more reliable and secure than anything we could implement ourselves on-premise. Unless an organization needs a specific hardware configuration not available on the cloud, or has strict business policies about the physical location of their data, I believe all new services should be built in the cloud.

3 Data Science Discussion Questions

1. What is the difference between questions of the form "who performed best in the past" vs. "who do you predict will perform best in the future"? How does your method for answering these questions change, if at all?

Questions like "who performed best in the past" can be answered with simple statistical analysis because we can use the collected data to measure performance with absolute certainty. There is a definitive answer to these questions given a set of defined criteria. The player that scored the most goals in the last decade can be found by tallying up the goals for each player over that time period. We can define the criteria we think best represents a "good performance" and use that criteria to come to a definitive answer.

Questions like "who do you predict will perform best in the future" require predictive analytics. There is no definitive answer, no matter how well our criteria is defined. There's a large amount of uncertainty in answering these questions, and that uncertainty should be accounted for in some way. We can combine historic data and predictions of likely future performance to *guess* which player will perform best, but we can never be certain. No matter how complex our predictive model is, if Connor McDavid has a career ending injury next season our prediction will likely be wrong! By contrast, a mediocre goalie could get LASIK surgery next offseason and outperform any possible prediction.

My method for solving questions about the past would rely on statistical analysis of the data we have already collected. To make predictions on the future, we need to build predictive models that combine historical performance and likely outcomes in the future to make reasonable predictions. The field of machine learning aims to solve these predictive analytics questions by using a wide variety of techniques.

In general, any predictive model will benefit from large and varied data sets. In order to predict the top goal scorer in 2025, I would want data about each player's historical goal totals, but I would also want information on the player's age and physical stature, the level of competition that those goals were scored in, the player's preferred playing position, and the team that player will be under contract with in 2025. We could also combine multiple models to build a more robust prediction. We could first predict likely line combinations in 2025 and use that data to influence our predicted goal totals. Answering predictive analytics questions requires deep understanding about the application (hockey), large amounts of trusted, high-quality, and preprocessed data, and models that are trained and validated to the best of our abilities. Finding answers to these questions is as much art as it is science - there's no perfect answer but there's lots of bad answers!

2. Suppose you had 20 years of goaltender statistics. Describe how would you answer the following question: Does a goalie's ability change over time?

In order to answer this question, we need to control for several factors to make a fair comparison of a goalie's performance from year-to-year. We would need to control for league wide changes and trends during this time period - like goalie pad sizes decreasing in size, and more goalie interference penalties. We must control for the amount of playing time each goalie gets. And should also control for the quality of shots a goalie faces during each season - maybe in 2002 a specific goalie played a limited number of games played against the worst teams in the league whereas in 2003 they played every game.

We can create a new metric for "goals allowed above expected per 60 minutes" (GAaE/60).

To calculate this metric, we first need to calculate the expected goals allowed for each goalie per 60 minutes which is dependent on the quantity and quality of shots that goalie faced. To do this, we would need to know each shot every goalie faced during a season and the likelihood that that shot would have resulted in a goal assuming league-average goaltending for the year.

Once we know the expected goals allowed for each goalie every year, we can calculate how many goals above the expected average our goalie allowed for each year. A negative number for this metric is better - meaning during a specific season, this goalie had fewer goals scored against them than expected.

Finally, I would compare a single goalie's GAaE/60 over the whole time period. If this number trends downwards over time, that means that this goalie improved, while an upward trend means that the goalie regressed. A significant change in either direction indicates that an individual goalie's ability changed over time.

3. Last season, the NHL forwards with highest rates of giveaways (per 60 minutes) include names like Jack Hughes, Evgeni Malkin, Johnny Gaudreau, Leon Draisaitl, Sebastian Aho, and Mitch Marner – players typically thought of to be good with the puck on their stick. On the surface, this doesn't make sense: Why are some of the league's most talented players among the worst in the league when it comes to giving the puck away? What are the issues with a metric like "giveaways per 60"? How would you design a better metric to capture how responsible a player is with the puck?

The major problem with a metric like "giveaways per 60" is that it's standardized to 60 minutes of ice time, instead of standardizing for the time that the player actually possesses the puck. A giveaway is a mistake by a player possessing the puck that causes the other team to gain possession. A takeaway is similar, but the turnover must be caused by the defense instead of a mistake by the offensive player.

A player that never touches the puck in a season will have a GV/60 of 0, but is that valuable? No, a player must possess the puck to drive offense. By contrast, a player like Evgeni Malkin frequently drives offensive plays by having the puck on his stick and tends to maintain possession of the puck for a significant amount of the time that he's on the ice. His GV/60 is likely going to be fairly high, but that doesn't mean he's irresponsible with the puck. It may even be desirable to have a player with a high GV/60, because he had to have possession of the puck in order to give it away! In the case of Evegeni Malkin, I think most coaches and teammates want the puck on his stick in important game situations - so GV/60 isn't the most valuable way to evaluate how responsible a player is with the puck.

Another consideration is shots that miss the goal. Obviously we want shots on goal, but should our metric penalize a player that shoots, misses the goal, and then loses possession to the other team? Shots that miss the goal generally end up deep in the other team's defensive zone and we need players to be taking shots to drive offense, even if they occasionally miss the goal. I would want Johnny Gaudreau to take as many shots as he can from high percentage areas of the ice, even if that occasionally results in a giveaway. The GV/60 will likely penalize high volume shooters.

A better metric should standardize for time of puck possession instead of time on ice. This could be represented by giveaways per 60 minutes of puck possession, or seconds of puck possession per giveaway. This will correct for the problem mentioned above, and I suspect Evegeni Malkin's "seconds of puck possession per giveaway" will look much better than his GV/60 when compared to other centers in the league.

I suspect that tracking seconds of puck possession for each player might be quite difficult. If the puck is not on any player's stick, is anyone possessing it? Maybe, depending on how possession is defined. A simpler metric that should be a close approximation could be “touches per giveaway”. We could count a “touch” as anytime a player makes a play on the puck for any amount of time without another player touching the puck. This would be much simpler to track and would be a great improvement over GV/60.