Horizontal Scaling of Databases - Sharding

For relational databases, we use a method called sharding to horizontally scale databases.

No SQL databases are built for scalability. It uses RS (Replica Sets)

ACID Compliance

Atomicity: Either the entire transaction succeeds or the thing fails.
Everything needs to be an atomic operation. If it fails, roll back, we don't need half data

Consistency: All database rules are enforced or the entire transaction is rolled back.
If a rule is set to have a value negative and a positive value is given, the entire transaction needs to be rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

Durability: Once a transaction is committed, its results are permanent, even in the case of a crash or power loss.

Databases like Oracle offer full ACID compliance. Sometimes you need to give some up for scalability

CAP Theorem

Consistency, **Availability**, **Partition Tolerance**
You can have any two of them and not three

3. Methodology (Clustering Algorithms)

Before we move to our clustering algorithms, we need to understand clustering. Clustering is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. (Sebastian Raschka, Vahid Mirjalili, 2019)3.1. K-Means Clustering

KMeans Clustering is an unsupervised machine learning algorithm that groups data into “K” distinct clusters based on similarity from dataset “D”. It does this by minimizing the within cluster sum of squares (WCSS) which is a measure of how compact the clusters are. (Kamil, et al., 2024)

3.1.1. Elbow and Silhouette Analysis.

One of the challenges of unsupervised learning is finding the optimal number of clusters, In this report, we are going to use the elbow method to finding the optimal number of clusters, we will then evaluate the the quality of the clustering with silhouette analysis. We do this because, an inappropriate choice for number of clusters (k) can result in poor clustering performance. (Sebastian Raschka, Vahid Mirjalili, 2019).

By combining both elbow and silhouette analysis, we can confindently select a value of K (number of clusters) that balances compactness, separation and interperability.

Figure 6: Elbow method and silhouette score

In the plot above, there is a visible “elbow” or curve around K=5 to 6. After K = 6, the decrease in inertia starts to flatten, meaning adding more clusters doesn’t really impact the intra-cluster compactness thus making K = 6 an ideal choice.

Also, the silhouette score curves significantly at K = 6 and continued to rise slightly after that making it the best choice. We therefore chose our number of clusters to be 6.

3.1.2. Cluster Profiling and Customer Segmentation.

Once the number of clusters (K=6) was determined, the Kmeans algorithm was applied to the dataset. The resulting clusters were visualised with Annual Income on the X-axis andSpending Score on the Y-axis. This scatter plot reveals a clear segmentation among shoppers based on their income and spending patterns

Figure 7: KMeans Clustering with K=6

3.1.3. Interpretation of Cluster Segments

Figure 8: Cluster ProfilingFrom Figure 7 and 8, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with modest income and high spending, their average income is 48.92 and their average spending score is 48.59. These could be younger or trend sensitive shoppers.

Cluster 1 (Orange): These customers can be grouped as low to mid income and low spending customers. They have an average income of 58.48 however their average spending score is 14.70.

Cluster 2 (Green): Customers with Moderate to high income but high spending. These customers have an average income of 81.69 and average spending score of 38.34

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with low income and low spending, however these customers earn averagely lower than cluster 1..

Cluster 5 (brown): These customers are broadly distributed in income and can be grouped as mid range earners and spenders. They have an average income of 64.53 and average spending score of 45.02.

3.2. Agglomerative Clustering

Agglomerative Clustering is an unsupervised machine learning algorithm that builds clusters using a bottom-up heirarchical approach. Every data point starts as its own cluster and pars of cluster are successively merged based on their similarity until a predefined number of clusteers remains. This process forms a denogram which is a tree-like structure that shows how far or close points and clusters are from one another. (Jain, et al., 1999).

One of the key benefits of Agglomerative Clustering is its flexibility as it does not require predefining the number of clusters (K). Instead we can determine K by visually inspecting the dendrogram or by manually setting K to match other models for comparative analysis.3.2.1. Cluster Profiling and Customer Segmentation.

Figure 9: Agglomerative Clustering with K=6

After applying Agglomerative Clustering with K=6, the resulting customer groups were visualised in the scatter plot where the X-Axis is Annual Income and the Y-Axis is Spending Score. The scatter plot shows clearly separated customer segments thus confirming the model’s ability to group shoppers with similar financial or shopping behaviour.3.2.2. Interpretation of Cluster Segments

Figure 10: Clustering profiling

From Figure 9 and 10, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with low to mid spending distributed across income levels. These may suggest a group with inconsistent engagement

Cluster 1 (Orange): These customers can be grouped as mostly high income and mid to high spending score. They have an average income of 82.25 however their average spending score is 25.10.
Cluster 2 (Green): Customers with lower income and low spending. These customers have an average income of 55.34 and average spending score of 21.48
Cluster 3 (Red): Customers with high income and high spending.
Cluster 4 (Purple): Customers with mostly moderate to high income but low spending.
Cluster 5 (brown): These customers are high spenders yet have low income.

3.3. DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised learning algorithm designed to identify clusters based on density rather than distance fromcentriods or heirarchical merging. It groups points that are closely packed together and labels points in low density regions as noise or outliers. (scikit-learn-developers, 2025)
DBSCAN does not need the number of clusters (K) to be specified in advance, instead it requires two parameters
1. ε (epsilon): The maximum distance between two samples to be considered
2. neighbours.
min_samples: The minimum number of samples in a neighborhood for a point to be considered a core point.
An advantage to using DBSCAN

```
const uploadPDF = useCallback(async (file: File, title?: string) => {
  try {
    dispatch({ type: 'SET_LOADING', payload: true });
    dispatch({ type: 'SET_ERROR', payload: null });
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 0 });

    // Simulate upload progress (since we don't have real progress tracking)
    let currentProgress = 0;
    const progressInterval = setInterval(() => {
      currentProgress = Math.min(currentProgress + 10, 90);
      dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: currentProgress });
    }, 200);

    const pdf = await apiClient.uploadPDF(file, title);

    clearInterval(progressInterval);
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 100 });
    dispatch({ type: 'ADD_PDF', payload: pdf });

    // Reset progress after a short delay
    setTimeout(() => {
      dispatch({ type: 'SET_UPL
```

Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started
Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.

If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let's break it down more clearly:

- **"Designed custom reward models"** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.
- **"Leveraging user feedback and prompt scoring"** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).
- **"Contributed to a 20% improvement in human preference ratings"** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Ask ChatGPT

I am going to give you a post I made, i would like you to copy a similar style and give me a post on Caching for my linkedIn Hey guys!, Lets talk about the CAP theorem. When designing distributed systems, it's very easy to choose technologies without thinking of their tradeoffs. Take databases for instance, how do we choose between MySQL, MongoDB, Cassandra or Dynamo db? Choosing the wrong technology for your use case could potentially cause issues in the future. This is why an important system design principle called CAP theorem is crucial for every seasoned engineer to know. What is CAP theorem? The CAP theorem states that in a distributed database system, you can only guarantee two out of three characteristics during a network partition. (Consistency, Availability and Partition Tolerance). Taking systems like social media, inventory, banking systems etc, all these systems trade of a particular characteristic to function efficiently. When you make a social media post, it may not reflect instantly across all users, it takes a while for your post to reflect across. A user in a different country may have to wait for probably a few minutes to see that you have posted but who cares?. Many people actually don't notice or see it as a problem. Do you ever hear a social media platform is down? Hardly, very rarely. This is because many social media platforms trade off Consistency for Availability and Partition Tolerance. They prefer to be up all the time but sacrifice a bit of consistency for this if it means waiting a while to sync data across. (AP system) On the other hand, considering trading or inventory systems, when you transfer money, or want to check stock prices, you expect the most up to date data like prices, stock number i.e. show the latest writes. However these systems are prone to downtimes because they would prefer to be down temporarily rather than show outdated information. These systems usually trade off Availability for Consistency. (CP system) When you have fleet of servers that communicate with each other to keep your app running, Partition tolerance simply means that if there's a network problem, the application should function as expected and can still handle and keep doing its job (at least partially). It's more about surviving communication failures in a distributed system. Databases like Cassandra DB works so well with systems that prioritise Availability. Due to its architecture, it is masterless thereby having no central point of failure. However it may take sometime for writes on the DB to reflect across all nodes. Other databases that have high availability is DynamoDB and Couchbase For high consistency applications where data accuracy is crucial, you can consider working with MongoDB (with strong consistency configs), PostgreSQL and FoundationDB, For CA systems, which are rare and only exist on single node system you can use any you want I guess. Hahaa 😅

ChatGPT said:

Absolutely! Here's a LinkedIn post in a similar engaging and educational style, but focused on **Caching in system design**:

Hey guys! Let's talk about **Caching** 🔁

When building scalable applications, one of the most overlooked but *powerful* tools in your arsenal is caching. It helps speed things up for your users and reduce load on your infrastructure. But as simple as it sounds, using it right requires some thought!

What is Caching?
Caching is the process of storing frequently accessed data in a *temporary storage layer* (usually in-memory) so that future requests for that data can be served faster.

Think of it like this:
Imagine asking your friend every time for a Wi-Fi password — annoying right? So what do you do? You write it on a sticky note next to your desk. That's your cache! 🧠

Now why is this important?

Let's say you're building a weather app. Do you really need to call the API every time someone checks the temperature in Berlin every 10 seconds? Probably not. You can store that result in a cache for 5–10 minutes.
Boom 💥 — faster response time, fewer API calls, and happier users.

Types of Caches:

- **In-Memory**: Super fast, stored in RAM (e.g. Redis, Memcached)
- **CDN Cache**: Edge-level caching for static assets (e.g. Cloudflare, Akamai)
- **Browser Cache**: Stored client-side
- **Application-Level Cache**: Caching inside your app logic (e.g. NestJS interceptors)

But it's not all roses… 🌹
Caching introduces complexity — what happens when the data changes? Do you update the cache? Invalidate it? Let it expire?

This is where strategies like **TTL (Time to Live)**, **Write-through**, **Write-back**, and **Cache Invalidation** come into play.

💡 Fun fact: There's a famous quote that says:

> "There are only two hard things in Computer Science: cache invalidation and naming things."

Use caching when:

- You have data that doesn't change often

- You want to reduce database or API load
- You need faster response times

Don’t use caching blindly though. Caching stale or inconsistent data can break user trust or lead to serious bugs (especially in things like banking, trading, or real-time analytics).

So next time your app is feeling sluggish, consider asking: *“Can I cache this?”* ⚡

Let me know if you want this post adapted for a specific audience like frontend/backend engineers, or add code snippets (e.g. Redis in NestJS).

You said:

Designed custom reward models leveraging user feedback and prompt scoring; contributed to a 20% improvement in human preference ratings. what does this mean

ChatGPT said:

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.
- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).

- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Horizontal Scaling of Databases - Sharding

For relational databases, we use a method called sharding to horizontally scale databases.

No SQL databases are built for scalability. It uses RS (Replica Sets)

ACID Compliance

Atomicity: Either the entire transaction succeeds or the thing fails.
Everything needs to be an atomic operation. If it fails, roll back, we don't need half data

Consistency: All database rules are enforced or the entire transaction is rolled back.
If a rule is set to have a value negative and a positive value is given, the entire transaction needs to be rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

Durability: Once a transaction is committed, its results are permanent, even in the case of a crash or power loss.

Databases like Oracle offer full ACID compliance. Sometimes you need to give some up for scalability

CAP Theorem

Consistency, **Availability**, **Partition Tolerance**
You can have any two of them and not three

3. Methodology (Clustering Algorithms)

Before we move to our clustering algorithms, we need to understand clustering. Clustering is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. (Sebastian Raschka, Vahid Mirjalili, 2019)

3.1. K-Means Clustering

KMeans Clustering is an unsupervised machine learning algorithm that groups data into "K" distinct clusters based on similarity from dataset "D". It does this by minimizing the within cluster sum of squares (WCSS) which is a measure of how compact the clusters are. (Kamil, et al., 2024)

3.1.1. Elbow and Silhouette Analysis.

One of the challenges of unsupervised learning is finding the optimal number of clusters, In this report, we are going to use the elbow method to finding the optimal number of clusters, we will then evaluate the the quality of the clustering with silhouette analysis. We do this because, an inappropriate choice for number of clusters (k) can result in poor clustering performance. (Sebastian Raschka, Vahid Mirjalili, 2019).

By combining both elbow and silhouette analysis, we can confindently select a value of K (number of clusters) that balances compactness, separation and interperability.

Figure 6: Elbow method and silhouette score

In the plot above, there is a visible "elbow" or curve around K=5 to 6. After K = 6, the decrease in inertia starts to flatten, meaning adding more clusters doesn't really impact the intra-cluster compactness thus making K = 6 an ideal choice.

Also, the silhouette score curves significantly at K = 6 and continued to rise slightly after that making it the best choice. We therefore chose our number of clusters to be 6.

3.1.2. Cluster Profiling and Customer Segmentation.

Once the number of clusters (K=6) was determined, the Kmeans algorithm was applied to the dataset. The resulting clusters were visualised with Annual Income on the X-axis andSpending Score on the Y-axis. This scatter plot reveals a clear segmentation among shoppers based on their income and spending patterns

Figure 7: KMeans Clustering with K=6

3.1.3. Interpretation of Cluster Segments

Figure 8: Cluster Profiling

From Figure 7 and 8, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with modest income and high spending, their average income is 48.92 and their average spending score is 48.59. These could be younger or trend sensitive shoppers.

Cluster 1 (Orange): These customers can be grouped as low to mid income and low spending customers. They have an average income of 58.48 however their average spending score is 14.70.

Cluster 2 (Green): Customers with Moderate to high income but high spending. These customers have an average income of 81.69 and average spending score of 38.34

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with low income and low spending, however these customers earn averagely lower than cluster 1..

Cluster 5 (brown): These customers are broadly distributed in income and can be grouped as mid range earners and spenders. They have an average income of 64.53 and average spending score of 45.02.

3.2. Agglomerative Clustering

Agglomerative Clustering is an unsupervised machine learning algorithm that builds clusters using a bottom-up heirarchical approach. Every data point starts as its own cluster and pars of cluster are successively merged based on their similarity until a predefined number of clusteers remains. This process forms a denogram which is a tree-like structure that shows how far or close points and clusters are from one another. (Jain, et al., 1999).

One of the key benefits of Agglomerative Clustering is its flexibility as it does not require predefining the number of clusters (K). Instead we can determine K by visually inspecting the dendrogram or by manually setting K to match other models for comparative analysis.

3.2.1. Cluster Profiling and Customer Segmentation.

Figure 9: Agglomerative Clustering with K=6

After applying Agglomerative Clustering with K=6, the resulting customer groups were visualised in the scatter plot where the X-Axis is Annual Income and the Y-Axis is Spending Score. The scatter plot shows clearly separated customer segments thus confirming the model’s ability to group shoppers with similar financial or shopping behaviour.3.2.2. Interpretation of Cluster Segments

Figure 10: Clustering profiling

From Figure 9 and 10, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with low to mid spending distributed across income levels. These may suggest a group with inconsistent engagement

Cluster 1 (Orange): These customers can be grouped as mostly high income and mid to high spending score. They have an average income of 82.25 however their average spending score is 25.10.

Cluster 2 (Green): Customers with lower income and low spending. These customers have an average income of 55.34 and average spending score of 21.48

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with mostly moderate to high income but low spending.

Cluster 5 (brown): These customers are high spenders yet have low income.

3.3. DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised learning algorithm designed to identify clusters based on density rather than distance fromcentriods or heirarchical merging. It groups points that are closely packed together and labels points in low density regions as noise or outliers. (scikit-learn-developers, 2025)

DBSCAN does not need the number of clusters (K) to be specified in advance, instead it requires two parameters

1. ε (epsilon): The maximum distance between two samples to be considered
2. neighbours.

min_samples: The minimum number of samples in a neighborhood for a point to be considered a core point.

An advantage to using DBSCAN

```
const uploadPDF = useCallback(async (file: File, title?: string) => {
  try {
    dispatch({ type: 'SET_LOADING', payload: true });
    dispatch({ type: 'SET_ERROR', payload: null });
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 0 });

    // Simulate upload progress (since we don't have real progress tracking)
    let currentProgress = 0;
    const progressInterval = setInterval(() => {
      currentProgress = Math.min(currentProgress + 10, 90);
      dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: currentProgress });
    }, 200);

    const pdf = await apiClient.uploadPDF(file, title);

    clearInterval(progressInterval);
```

```
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 100 });
    dispatch({ type: 'ADD_PDF', payload: pdf });

    // Reset progress after a short delay
    setTimeout(() => {
      dispatch({ type: 'SET_UPL
```

Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started
Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)

4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let's break it down more clearly:

- **"Designed custom reward models"** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.
- **"Leveraging user feedback and prompt scoring"** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).
- **"Contributed to a 20% improvement in human preference ratings"** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Ask ChatGPT

I am going to give you a post I made, i would like you to copy a similar style and give me a post on Caching for my linkedIn Hey guys!, Lets talk about the CAP theorem. When designing distributed systems, it’s very easy to choose technologies without thinking of their tradeoffs. Take databases for instance, how do we choose between MySQL, MongoDB, Cassandra or Dynamo db? Choosing the wrong technology for your use case could potentially cause issues in the future. This is why an important system design principle called CAP theorem is crucial for every seasoned engineer to know. What is CAP theorem? The CAP theorem states that in a distributed database system, you can only guarantee two out of three characteristics during a network partition. (Consistency, Availability and Partition Tolerance). Taking systems like social media, inventory, banking systems etc, all these systems trade of a particular characteristic to function efficiently. When you make a social media post, it may not reflect instantly across all users, it takes a while for your post to reflect across. A user in a different country may have to wait for probably a few minutes to see that you have posted but who cares?. Many people actually don’t notice or see it as a problem. Do you ever hear a social media platform is down? Hardly, very rarely. This is because many social media platforms trade off Consistency for Availability and Partition Tolerance. They prefer to be up all the time but sacrifice a bit of consistency for this if it means waiting a while to sync data across. (AP system) On the other hand, considering trading or inventory systems, when you transfer money, or want to check stock prices, you expect the most up to date data like prices, stock number i.e. show the latest writes. However these systems are prone to downtimes because they would prefer to be down temporarily rather than show outdated information. These systems usually trade off Availability for Consistency. (CP system) When you have fleet of servers that communicate with each other to keep your app running, Partition tolerance simply means that if there’s a network problem, the application should function as expected and can still handle and keep doing its job (at least partially). It’s more about surviving communication failures in a distributed system. Databases like Cassandra DB works so well with systems that prioritise Availability. Due to its architecture, it is masterless thereby having no central point of failure. However it may take sometime for writes on the DB to reflect across all nodes. Other databases that have high availability is DynamoDB and Couchbase For high consistency applications where data accuracy is crucial, you can consider working with MongoDB (with strong consistency configs), PostgreSQL and FoundationDB, For CA systems, which are rare and only exist on single node system you can use any you want I guess. Hahaa 😅

ChatGPT said:

Absolutely! Here's a LinkedIn post in a similar engaging and educational style, but focused on **Caching in system design**:

Hey guys! Let's talk about **Caching** 🔁

When building scalable applications, one of the most overlooked but *powerful* tools in your arsenal is caching. It helps speed things up for your users and reduce load on your infrastructure. But as simple as it sounds, using it right requires some thought!

What is Caching?
Caching is the process of storing frequently accessed data in a *temporary storage layer* (usually in-memory) so that future requests for that data can be served faster.

Think of it like this:
Imagine asking your friend every time for a Wi-Fi password — annoying right? So what do you do? You write it on a sticky note next to your desk. That's your cache! 🧠

Now why is this important?

Let's say you're building a weather app. Do you really need to call the API every time someone checks the temperature in Berlin every 10 seconds? Probably not. You can store that result in a cache for 5–10 minutes.
Boom 💥 — faster response time, fewer API calls, and happier users.

Types of Caches:

- **In-Memory**: Super fast, stored in RAM (e.g. Redis, Memcached)
- **CDN Cache**: Edge-level caching for static assets (e.g. Cloudflare, Akamai)
- **Browser Cache**: Stored client-side
- **Application-Level Cache**: Caching inside your app logic (e.g. NestJS interceptors)

But it's not all roses… 🌹
Caching introduces complexity — what happens when the data changes? Do you update the cache? Invalidate it? Let it expire?

This is where strategies like **TTL (Time to Live)**, **Write-through**, **Write-back**, and **Cache Invalidation** come into play.

💡 Fun fact: There's a famous quote that says:

> “There are only two hard things in Computer Science: cache invalidation and naming things.”

Use caching when:

- You have data that doesn’t change often
- You want to reduce database or API load
- You need faster response times

Don’t use caching blindly though. Caching stale or inconsistent data can break user trust or lead to serious bugs (especially in things like banking, trading, or real-time analytics).

So next time your app is feeling sluggish, consider asking: *“Can I cache this?”* ⚡

Let me know if you want this post adapted for a specific audience like frontend/backend engineers, or add code snippets (e.g. Redis in NestJS).

You said:

Designed custom reward models leveraging user feedback and prompt scoring; contributed to a 20% improvement in human preference ratings. what does this mean

ChatGPT said:

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).
- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Horizontal Scaling of Databases - Sharding

For relational databases, we use a method called sharding to horizontally scale databases.

No SQL databases are built for scalability. It uses RS (Replica Sets)

ACID Compliance

Atomicity: Either the entire transaction succeeds or the thing fails.
Everything needs to be an atomic operation. If it fails, roll back, we don't need half data

Consistency: All database rules are enforced or the entire transaction is rolled back.
If a rule is set to have a value negative and a positive value is given, the entire transaction needs to be rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

Durability: Once a transaction is committed, its results are permanent, even in the case of a crash or power loss.

Databases like Oracle offer full ACID compliance. Sometimes you need to give some up for scalability

CAP Theorem

Consistency, **Availability**, **Partition Tolerance**
You can have any two of them and not three

3. Methodology (Clustering Algorithms)

Before we move to our clustering algorithms, we need to understand clustering. Clustering is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. (Sebastian Raschka, Vahid Mirjalili, 2019)

3.1. K-Means Clustering

KMeans Clustering is an unsupervised machine learning algorithm that groups data into “K” distinct clusters based on similarity from dataset “D”. It does this by minimizing the within cluster sum of squares (WCSS) which is a measure of how compact the clusters are. (Kamil, et al., 2024)

3.1.1. Elbow and Silhouette Analysis.

One of the challenges of unsupervised learning is finding the optimal number of clusters, In this report, we are going to use the elbow method to finding the optimal number of clusters, we will then evaluate the the quality of the clustering with silhouette analysis. We do this because, an inappropriate choice for number of clusters (k) can result in poor clustering performance. (Sebastian Raschka, Vahid Mirjalili, 2019).

By combining both elbow and silhouette analysis, we can confindently select a value of K (number of clusters) that balances compactness, separation and interperability.

Figure 6: Elbow method and silhouette score

In the plot above, there is a visible “elbow” or curve around K=5 to 6. After K = 6, the decrease in inertia starts to flatten, meaning adding more clusters doesn’t really impact the intra-cluster compactness thus making K = 6 an ideal choice.

Also, the silhouette score curves significantly at K = 6 and continued to rise slightly after that making it the best choice. We therefore chose our number of clusters to be 6.

3.1.2. Cluster Profiling and Customer Segmentation.

Once the number of clusters (K=6) was determined, the Kmeans algorithm was applied to the dataset. The resulting clusters were visualised with Annual Income on the X-axis andSpending Score on the Y-axis. This scatter plot reveals a clear segmentation among shoppers based on their income and spending patterns

Figure 7: KMeans Clustering with K=6

3.1.3. Interpretation of Cluster Segments

Figure 8: Cluster Profiling

From Figure 7 and 8, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with modest income and high spending, their average income is 48.92 and their average spending score is 48.59. These could be younger or trend sensitive shoppers.

Cluster 1 (Orange): These customers can be grouped as low to mid income and low spending customers. They have an average income of 58.48 however their average spending score is 14.70.

Cluster 2 (Green): Customers with Moderate to high income but high spending. These customers have an average income of 81.69 and average spending score of 38.34

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with low income and low spending, however these customers earn averagely lower than cluster 1..

Cluster 5 (brown): These customers are broadly distributed in income and can be grouped as mid range earners and spenders. They have an average income of 64.53 and average spending score of 45.02.

3.2. Agglomerative Clustering

Agglomerative Clustering is an unsupervised machine learning algorithm that builds clusters

using a bottom-up heirarchical approach. Every data point starts as its own cluster and pars of cluster are successively merged based on their similarity until a predefined number of clusteers remains. This process forms a denogram which is a tree-like structure that shows how far or close points and clusters are from one another. (Jain, et al., 1999).
One of the key benefits of Agglomerative Clustering is its flexibility as it does not require predefining the number of clusters (K). Instead we can determine K by visually inspecting the dendrogram or by manually setting K to match other models for comparative analysis.3.2.1. Cluster Profiling and Customer Segmentation.

Figure 9: Agglomerative Clustering with K=6

After applying Agglomerative Clustering with K=6, the resulting customer groups were visualised in the scatter plot where the X-Axis is Annual Income and the Y-Axis is Spending Score. The scatter plot shows clearly separated customer segments thus confirming the model's ability to group shoppers with similar financial or shopping behaviour.3.2.2. Interpretation of Cluster Segments

Figure 10: Clustering profiling

From Figure 9 and 10, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:
Cluster 0: (Blue): These are customers with low to mid spending distributed across income levels. These may suggest a group with inconsistent engagement
Cluster 1 (Orange): These customers can be grouped as mostly high income and mid to high spending score. They have an average income of 82.25 however their average spending score is 25.10.
Cluster 2 (Green): Customers with lower income and low spending. These customers have an average income of 55.34 and average spending score of 21.48
Cluster 3 (Red): Customers with high income and high spending.
Cluster 4 (Purple): Customers with mostly moderate to high income but low spending.
Cluster 5 (brown): These customers are high spenders yet have low income.

3.3. DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised learning algorithm designed to identify clusters based on density rather than distance fromcentriods or heirarchical merging. It groups points that are closely packed together and labels points in low density regions as noise or outliers. (scikit-learn-developers, 2025)
DBSCAN does not need the number of clusters (K) to be specified in advance, instead it requires two parameters

1. ε (epsilon): The maximum distance between two samples to be considered
2. neighbours.

min_samples: The minimum number of samples in a neighborhood for a point to be considered a core point.
An advantage to using DBSCAN

```
const uploadPDF = useCallback(async (file: File, title?: string) => {
  try {
    dispatch({ type: 'SET_LOADING', payload: true });
    dispatch({ type: 'SET_ERROR', payload: null });
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 0 });

    // Simulate upload progress (since we don't have real progress tracking)
    let currentProgress = 0;
    const progressInterval = setInterval(() => {
```

```
    currentProgress = Math.min(currentProgress + 10, 90);
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: currentProgress });
  }, 200);

  const pdf = await apiClient.uploadPDF(file, title);

  clearInterval(progressInterval);
  dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 100 });
  dispatch({ type: 'ADD_PDF', payload: pdf });

  // Reset progress after a short delay
  setTimeout(() => {
    dispatch({ type: 'SET_UPL
```

Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started
Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
  - Each PDF should be split into logical sections (e.g., by pages or chapters)
  - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
  - A list view showing available PDFs with their metadata
  - A detail view showing:
    - PDF metadata
    - A paginated list of chunks/sections
    - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:

    - How users rated responses (thumbs up/down, star ratings, etc.)

    - Scores given to responses from prompts (possibly by humans or heuristics).

- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Ask ChatGPT

I am going to give you a post I made, i would like you to copy a similar style and give me a post on Caching for my linkedIn Hey guys!, Lets talk about the CAP theorem. When designing distributed systems, it’s very easy to choose technologies without thinking of their tradeoffs. Take databases for instance, how do we choose between MySQL, MongoDB, Cassandra or Dynamo db? Choosing the wrong technology for your use case could potentially cause issues in the future. This is why an important system design principle called CAP theorem is crucial for every seasoned engineer to know. What is CAP theorem? The CAP theorem states that in a distributed database system, you can only guarantee two out of three characteristics during a network partition. (Consistency, Availability and Partition Tolerance). Taking systems like social media, inventory, banking systems etc, all these systems trade of a particular characteristic to function efficiently. When you make a social media post, it may not reflect instantly across all users, it takes a while for your post to reflect across. A user in a different country may have to wait for probably a few minutes to see that you have posted but who cares?. Many people actually don’t notice or see it as a problem. Do you ever hear a social media platform is down? Hardly, very rarely. This is because many social media platforms trade off Consistency for Availability and Partition Tolerance. They prefer to be up all the time but sacrifice a bit of consistency for this if it means waiting a while to sync data across. (AP system) On the other hand, considering trading or inventory systems, when you transfer money, or want to check stock prices, you expect the most up to date data like prices, stock number i.e. show the latest writes. However these systems are prone to downtimes because they would prefer to be down temporarily rather than show outdated information. These systems usually trade off Availability for Consistency. (CP system) When you have fleet of servers that communicate with each other to keep your app running, Partition tolerance simply means that if there’s a network problem, the application should function as expected and can still handle and keep doing its job (at least partially). It’s more about surviving communication failures in a distributed system. Databases like Cassandra DB works so well with systems that prioritise Availability. Due to its architecture, it is masterless thereby having no central point of failure. However it may take sometime for writes on the DB to reflect across all nodes. Other databases that have high availability is DynamoDB and Couchbase For high consistency applications where data accuracy is crucial, you can consider working with MongoDB (with strong consistency configs), PostgreSQL and FoundationDB, For CA systems, which are rare and only exist on single node system you can use any you want I guess. Hahaa 😅

ChatGPT said:

Absolutely! Here's a LinkedIn post in a similar engaging and educational style, but focused on **Caching in system design**:

Hey guys! Let's talk about **Caching** 🔁

When building scalable applications, one of the most overlooked but *powerful* tools in your arsenal is caching. It helps speed things up for your users and reduce load on your infrastructure. But as simple as it sounds, using it right requires some thought!

What is Caching?
Caching is the process of storing frequently accessed data in a *temporary storage layer* (usually in-memory) so that future requests for that data can be served faster.

Think of it like this:
Imagine asking your friend every time for a Wi-Fi password — annoying right? So what do you do? You write it on a sticky note next to your desk. That's your cache! 🧠

Now why is this important?

Let's say you're building a weather app. Do you really need to call the API every time someone checks the temperature in Berlin every 10 seconds? Probably not. You can store that result in a cache for 5–10 minutes.
Boom 💥 — faster response time, fewer API calls, and happier users.

Types of Caches:

- **In-Memory**: Super fast, stored in RAM (e.g. Redis, Memcached)
- **CDN Cache**: Edge-level caching for static assets (e.g. Cloudflare, Akamai)
- **Browser Cache**: Stored client-side
- **Application-Level Cache**: Caching inside your app logic (e.g. NestJS interceptors)

But it’s not all roses… 🌹
Caching introduces complexity — what happens when the data changes? Do you update the cache? Invalidate it? Let it expire?

This is where strategies like **TTL (Time to Live)**, **Write-through**, **Write-back**, and **Cache Invalidation** come into play.

💡 Fun fact: There’s a famous quote that says:

> “There are only two hard things in Computer Science: cache invalidation and naming things.”

Use caching when:

- You have data that doesn’t change often
- You want to reduce database or API load
- You need faster response times

Don’t use caching blindly though. Caching stale or inconsistent data can break user trust or lead to serious bugs (especially in things like banking, trading, or real-time analytics).

So next time your app is feeling sluggish, consider asking: *“Can I cache this?”* ⚡

Let me know if you want this post adapted for a specific audience like frontend/backend engineers, or add code snippets (e.g. Redis in NestJS).

You said:
Designed custom reward models leveraging user feedback and prompt scoring; contributed to a 20% improvement in human preference ratings. what does this mean

ChatGPT said:

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).
- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Horizontal Scaling of Databases - Sharding

For relational databases, we use a method called sharding to horizontally scale databases.

No SQL databases are built for scalability. It uses RS (Replica Sets)

ACID Compliance

Atomicity: Either the entire transaction succeeds or the thing fails.
Everything needs to be an atomic operation. If it fails, roll back, we don't need half data

Consistency: All database rules are enforced or the entire transaction is rolled back.
If a rule is set to have a value negative and a positive value is given, the entire transaction needs to be rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

Durability: Once a transaction is committed, its results are permanent, even in the case of a crash or power loss.

Databases like Oracle offer full ACID compliance. Sometimes you need to give some up for scalability

CAP Theorem

Consistency, **Availability**, **Partition Tolerance**
You can have any two of them and not three

3. Methodology (Clustering Algorithms)

Before we move to our clustering algorithms, we need to understand clustering. Clustering is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. (Sebastian Raschka, Vahid Mirjalili, 2019)3.1. K-Means Clustering

KMeans Clustering is an unsupervised machine learning algorithm that groups data into "K" distinct clusters based on similarity from dataset "D". It does this by minimizing the within cluster sum of squares (WCSS) which is a measure of how compact the clusters are. (Kamil, et al., 2024)

3.1.1. Elbow and Silhouette Analysis.

One of the challenges of unsupervised learning is finding the optimal number of clusters, In this report, we are going to use the elbow method to finding the optimal number of clusters, we will then evaluate the the quality of the clustering with silhouette analysis. We do this because, an inappropriate choice for number of clusters (k) can result in poor clustering performance. (Sebastian Raschka, Vahid Mirjalili, 2019).

By combining both elbow and silhouette analysis, we can confindently select a value of K (number of clusters) that balances compactness, separation and interperability.

Figure 6: Elbow method and silhouette score

In the plot above, there is a visible "elbow" or curve around K=5 to 6. After K = 6, the decrease in inertia starts to flatten, meaning adding more clusters doesn't really impact the intra-cluster compactness thus making K = 6 an ideal choice.

Also, the silhouette score curves significantly at K = 6 and continued to rise slightly after that making it the best choice. We therefore chose our number of clusters to be 6.

3.1.2. Cluster Profiling and Customer Segmentation.

Once the number of clusters (K=6) was determined, the Kmeans algorithm was applied to the dataset. The resulting clusters were visualised with Annual Income on the X-axis andSpending Score on the Y-axis. This scatter plot reveals a clear segmentation among shoppers based on their income and spending patterns

Figure 7: KMeans Clustering with K=6

3.1.3. Interpretation of Cluster Segments

Figure 8: Cluster ProfilingFrom Figure 7 and 8, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with modest income and high spending, their average income is 48.92 and their average spending score is 48.59. These could be younger or trend sensitive shoppers.

Cluster 1 (Orange): These customers can be grouped as low to mid income and low spending customers. They have an average income of 58.48 however their average spending score is 14.70.

Cluster 2 (Green): Customers with Moderate to high income but high spending. These customers have an average income of 81.69 and average spending score of 38.34
Cluster 3 (Red): Customers with high income and high spending.
Cluster 4 (Purple): Customers with low income and low spending, however these customers earn averagely lower than cluster 1..
Cluster 5 (brown): These customers are broadly distributed in income and can be grouped as mid range earners and spenders. They have an average income of 64.53 and average spending score of 45.02.

3.2. Agglomerative Clustering

Agglomerative Clustering is an unsupervised machine learning algorithm that builds clusters using a bottom-up heirarchical approach. Every data point starts as its own cluster and pars of cluster are successively merged based on their similarity until a predefined number of clusteers remains. This process forms a denogram which is a tree-like structure that shows how far or close points and clusters are from one another. (Jain, et al., 1999).
One of the key benefits of Agglomerative Clustering is its flexibility as it does not require predefining the number of clusters (K). Instead we can determine K by visually inspecting the dendrogram or by manually setting K to match other models for comparative analysis.3.2.1. Cluster Profiling and Customer Segmentation.

Figure 9: Agglomerative Clustering with K=6

After applying Agglomerative Clustering with K=6, the resulting customer groups were visualised in the scatter plot where the X-Axis is Annual Income and the Y-Axis is Spending Score. The scatter plot shows clearly separated customer segments thus confirming the model's ability to group shoppers with similar financial or shopping behaviour.3.2.2. Interpretation of Cluster Segments

Figure 10: Clustering profiling

From Figure 9 and 10, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:
Cluster 0: (Blue): These are customers with low to mid spending distributed across income levels. These may suggest a group with inconsistent engagement
Cluster 1 (Orange): These customers can be grouped as mostly high income and mid to high spending score. They have an average income of 82.25 however their average spending score is 25.10.
Cluster 2 (Green): Customers with lower income and low spending. These customers have an average income of 55.34 and average spending score of 21.48
Cluster 3 (Red): Customers with high income and high spending.
Cluster 4 (Purple): Customers with mostly moderate to high income but low spending.
Cluster 5 (brown): These customers are high spenders yet have low income.

3.3. DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised learning algorithm designed to identify clusters based on density rather than distance fromcentriods or heirarchical merging. It groups points that are closely packed together and labels points in low density regions as noise or outliers. (scikit-learn-developers, 2025)
DBSCAN does not need the number of clusters (K) to be specified in advance, instead it requires two parameters
1. ε (epsilon): The maximum distance between two samples to be considered
2. neighbours.
min_samples: The minimum number of samples in a neighborhood for a point to be considered a core point.
An advantage to using DBSCAN

```
const uploadPDF = useCallback(async (file: File, title?: string) => {
```

```
try {
  dispatch({ type: 'SET_LOADING', payload: true });
  dispatch({ type: 'SET_ERROR', payload: null });
  dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 0 });

  // Simulate upload progress (since we don't have real progress tracking)
  let currentProgress = 0;
  const progressInterval = setInterval(() => {
    currentProgress = Math.min(currentProgress + 10, 90);
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: currentProgress });
  }, 200);

  const pdf = await apiClient.uploadPDF(file, title);

  clearInterval(progressInterval);
  dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 100 });
  dispatch({ type: 'ADD_PDF', payload: pdf });

  // Reset progress after a short delay
  setTimeout(() => {
    dispatch({ type: 'SET_UPL
```

Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.

If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started
Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:
   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let's break it down more clearly:

- **"Designed custom reward models"** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:
    - How users rated responses (thumbs up/down, star ratings, etc.)
    - Scores given to responses from prompts (possibly by humans or heuristics).
- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Ask ChatGPT

I am going to give you a post I made, i would like you to copy a similar style and give me a post on Caching for my linkedIn Hey guys!, Lets talk about the CAP theorem. When designing distributed systems, it’s very easy to choose technologies without thinking of their tradeoffs. Take databases for instance, how do we choose between MySQL, MongoDB, Cassandra or Dynamo db? Choosing the wrong technology for your use case could potentially cause issues in the future. This is why an important system design principle called CAP theorem is crucial for every seasoned engineer to know. What is CAP theorem? The CAP theorem states that in a distributed database system, you can only guarantee two out of three characteristics during a network partition. (Consistency, Availability and Partition Tolerance). Taking systems like social media, inventory, banking systems etc, all these systems trade of a particular characteristic to function efficiently. When you make a social media post, it may not reflect instantly across all users, it takes a while for your post to reflect across. A user in a different country may have to wait for probably a few minutes to see that you have posted but who cares?. Many people actually don’t notice or see it as a problem. Do you ever hear a social media platform is down? Hardly, very rarely. This is because many social media platforms trade off Consistency for Availability and Partition Tolerance. They prefer to be up all the time but sacrifice a bit of consistency for this if it means waiting a while to sync data across. (AP system) On the other hand, considering trading or inventory systems, when you transfer money, or want to check stock prices, you expect the most up to date data like prices, stock number i.e. show the latest writes. However these systems are prone to downtimes because they would prefer to be down temporarily rather than show outdated information. These systems usually trade off Availability for Consistency. (CP system) When you have fleet of servers that communicate with each other to keep your app running, Partition tolerance simply means that if there's a network problem, the application should function as expected and can still handle and keep doing its job (at least partially). It's more about surviving communication failures in a distributed system. Databases like Cassandra DB works so well with systems that prioritise Availability. Due to its architecture, it is masterless thereby having no central point of failure. However it may take sometime for writes on the DB to reflect across all nodes. Other databases that have high availability is DynamoDB and Couchbase For high consistency applications where data accuracy is crucial, you can consider working with MongoDB (with strong consistency configs), PostgreSQL and FoundationDB, For CA systems, which are rare and only exist on single node system you can use any you want I guess. Hahaa 😅

ChatGPT said:

Absolutely! Here's a LinkedIn post in a similar engaging and educational style, but focused on **Caching in system design**:

Hey guys! Let's talk about **Caching** 🔁

When building scalable applications, one of the most overlooked but *powerful* tools in your arsenal is caching. It helps speed things up for your users and reduce load on your infrastructure. But as simple as it sounds, using it right requires some thought!

What is Caching?
Caching is the process of storing frequently accessed data in a *temporary storage layer* (usually in-memory) so that future requests for that data can be served faster.

Think of it like this:
Imagine asking your friend every time for a Wi-Fi password — annoying right? So what do you do? You write it on a sticky note next to your desk. That's your cache! 🧠

Now why is this important?

Let's say you're building a weather app. Do you really need to call the API every time someone checks the temperature in Berlin every 10 seconds? Probably not. You can store that result in a cache for 5–10 minutes.
Boom 💥 — faster response time, fewer API calls, and happier users.

Types of Caches:

- **In-Memory**: Super fast, stored in RAM (e.g. Redis, Memcached)
- **CDN Cache**: Edge-level caching for static assets (e.g. Cloudflare, Akamai)
- **Browser Cache**: Stored client-side
- **Application-Level Cache**: Caching inside your app logic (e.g. NestJS interceptors)

But it's not all roses… 🌹
Caching introduces complexity — what happens when the data changes? Do you update the cache? Invalidate it? Let it expire?

This is where strategies like **TTL (Time to Live)**, **Write-through**, **Write-back**, and **Cache Invalidation** come into play.

💡 Fun fact: There's a famous quote that says:

> "There are only two hard things in Computer Science: cache invalidation and naming things."

Use caching when:

- You have data that doesn't change often
- You want to reduce database or API load
- You need faster response times

Don't use caching blindly though. Caching stale or inconsistent data can break user trust or lead to serious bugs (especially in things like banking, trading, or real-time analytics).

So next time your app is feeling sluggish, consider asking: *"Can I cache this?"* ⚡

Let me know if you want this post adapted for a specific audience like frontend/backend engineers, or add code snippets (e.g. Redis in NestJS).

You said:
Designed custom reward models leveraging user feedback and prompt scoring; contributed to a 20% improvement in human preference ratings. what does this mean

ChatGPT said:

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let's break it down more clearly:

- **"Designed custom reward models"** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **"Leveraging user feedback and prompt scoring"** → You trained or tuned these models using data such as:
  - How users rated responses (thumbs up/down, star ratings, etc.)
  - Scores given to responses from prompts (possibly by humans or heuristics).
- **"Contributed to a 20% improvement in human preference ratings"** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Horizontal Scaling of Databases - Sharding

For relational databases, we use a method called sharding to horizontally scale databases.

No SQL databases are built for scalability. It uses RS (Replica Sets)

ACID Compliance

Atomicity: Either the entire transaction succeeds or the thing fails.
Everything needs to be an atomic operation. If it fails, roll back, we don't need half data

Consistency: All database rules are enforced or the entire transaction is rolled back. If a rule is set to have a value negative and a positive value is given, the entire transaction needs to be rolled back.

Isolation: No transaction is affected by any other transaction that is still in progress.

Durability: Once a transaction is committed, its results are permanent, even in the case of a crash or power loss.

Databases like Oracle offer full ACID compliance. Sometimes you need to give some up for scalability

CAP Theorem

Consistency, **Availability**, **Partition Tolerance**
You can have any two of them and not three

3. Methodology (Clustering Algorithms)

Before we move to our clustering algorithms, we need to understand clustering. Clustering is a technique that allows us to find groups of similar objects that are more related to each other than to objects in other groups. Examples of business-oriented applications of clustering include the grouping of documents, music, and movies by different topics, or finding customers that share similar interests based on common purchase behaviors as a basis for recommendation engines. (Sebastian Raschka, Vahid Mirjalili, 2019)3.1. K-Means Clustering

KMeans Clustering is an unsupervised machine learning algorithm that groups data into “K” distinct clusters based on similarity from dataset “D”. It does this by minimizing the within cluster sum of squares (WCSS) which is a measure of how compact the clusters are. (Kamil, et al., 2024)

3.1.1. Elbow and Silhouette Analysis.

One of the challenges of unsupervised learning is finding the optimal number of clusters, In this report, we are going to use the elbow method to finding the optimal number of clusters, we will then evaluate the the quality of the clustering with silhouette analysis. We do this because, an inappropriate choice for number of clusters (k) can result in poor clustering performance. (Sebastian Raschka, Vahid Mirjalili, 2019).

By combining both elbow and silhouette analysis, we can confindently select a value of K (number of clusters) that balances compactness, separation and interperability.

Figure 6: Elbow method and silhouette score

In the plot above, there is a visible “elbow” or curve around K=5 to 6. After K = 6, the decrease in inertia starts to flatten, meaning adding more clusters doesn’t really impact the intra-cluster compactness thus making K = 6 an ideal choice.

Also, the silhouette score curves significantly at K = 6 and continued to rise slightly after that making it the best choice. We therefore chose our number of clusters to be 6.

3.1.2. Cluster Profiling and Customer Segmentation.

Once the number of clusters (K=6) was determined, the Kmeans algorithm was applied to the dataset. The resulting clusters were visualised with Annual Income on the X-axis andSpending Score on the Y-axis. This scatter plot reveals a clear segmentation among shoppers based on their income and spending patterns

Figure 7: KMeans Clustering with K=6

3.1.3. Interpretation of Cluster Segments

Figure 8: Cluster ProfilingFrom Figure 7 and 8, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with modest income and high spending, their average income is 48.92 and their average spending score is 48.59. These could be younger or trend sensitive shoppers.

Cluster 1 (Orange): These customers can be grouped as low to mid income and low spending customers. They have an average income of 58.48 however their average spending score is 14.70.

Cluster 2 (Green): Customers with Moderate to high income but high spending. These customers have an average income of 81.69 and average spending score of 38.34

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with low income and low spending, however these customers earn averagely lower than cluster 1..

Cluster 5 (brown): These customers are broadly distributed in income and can be grouped as mid range earners and spenders. They have an average income of 64.53 and average spending score of 45.02.

3.2. Agglomerative Clustering

Agglomerative Clustering is an unsupervised machine learning algorithm that builds clusters using a bottom-up heirarchical approach. Every data point starts as its own cluster and pars of cluster are successively merged based on their similarity until a predefined number of clusteers remains. This process forms a denogram which is a tree-like structure that shows how far or close points and clusters are from one another. (Jain, et al., 1999).

One of the key benefits of Agglomerative Clustering is its flexibility as it does not require predefining the number of clusters (K). Instead we can determine K by visually inspecting the dendrogram or by manually setting K to match other models for comparative analysis.3.2.1. Cluster Profiling and Customer Segmentation.

Figure 9: Agglomerative Clustering with K=6

After applying Agglomerative Clustering with K=6, the resulting customer groups were visualised in the scatter plot where the X-Axis is Annual Income and the Y-Axis is Spending Score. The scatter plot shows clearly separated customer segments thus confirming the model’s ability to group shoppers with similar financial or shopping behaviour.3.2.2. Interpretation of Cluster Segments

Figure 10: Clustering profiling

From Figure 9 and 10, we can interprete the cluster segments, we observe that we can give an interpretation to each clusters as follows:

Cluster 0: (Blue): These are customers with low to mid spending distributed across income levels. These may suggest a group with inconsistent engagement

Cluster 1 (Orange): These customers can be grouped as mostly high income and mid to high spending score. They have an average income of 82.25 however their average spending score is 25.10.

Cluster 2 (Green): Customers with lower income and low spending. These customers have an average income of 55.34 and average spending score of 21.48

Cluster 3 (Red): Customers with high income and high spending.

Cluster 4 (Purple): Customers with mostly moderate to high income but low spending.

Cluster 5 (brown): These customers are high spenders yet have low income.

3.3. DBSCAN Clustering

DBSCAN (Density-Based Spatial Clustering of Applications with Noise) is an unsupervised learning algorithm designed to identify clusters based on density rather than distance fromcentriods or heirarchical merging. It groups points that are closely packed together and labels points in low density regions as noise or outliers. (scikit-learn-developers, 2025)

DBSCAN does not need the number of clusters (K) to be specified in advance, instead it requires two parameters

1. ε (epsilon): The maximum distance between two samples to be considered
2. neighbours.
min_samples: The minimum number of samples in a neighborhood for a point to be considered a core point.
An advantage to using DBSCAN

```
const uploadPDF = useCallback(async (file: File, title?: string) => {
  try {
   dispatch({ type: 'SET_LOADING', payload: true });
   dispatch({ type: 'SET_ERROR', payload: null });
   dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 0 });

   // Simulate upload progress (since we don't have real progress tracking)
   let currentProgress = 0;
   const progressInterval = setInterval(() => {
    currentProgress = Math.min(currentProgress + 10, 90);
    dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: currentProgress });
   }, 200);

   const pdf = await apiClient.uploadPDF(file, title);

   clearInterval(progressInterval);
   dispatch({ type: 'SET_UPLOAD_PROGRESS', payload: 100 });
   dispatch({ type: 'ADD_PDF', payload: pdf });

   // Reset progress after a short delay
   setTimeout(() => {
    dispatch({ type: 'SET_UPL
```

Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:

- A list view showing available PDFs with their metadata
- A detail view showing:
  - PDF metadata
  - A paginated list of chunks/sections
  - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started
Create a feature that allows users to:
1. Upload a PDF file through a web interface
2. Parse the PDF content on the backend into chunks/sections:
   - Each PDF should be split into logical sections (e.g., by pages or chapters)
   - Store basic metadata (upload date, title, total pages) with the PDF
3. Store the parsed content in a database (PDF and its chunks)
4. Display the parsed content in a structured way on the frontend:

   - A list view showing available PDFs with their metadata
   - A detail view showing:
     - PDF metadata
     - A paginated list of chunks/sections
     - The ability to search through chunks

Note: Build as far as you get in a few hours, focus on the parts you consider most important to show your skills and best practices.
If you run out of time, feel free to add your thoughts and ideas as comments or notes.

And feel free to improve our code base ! :)

Tech Stack
- Frontend: React + TypeScript
- Backend: FastAPI + Python

Getting Started

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:

    - How users rated responses (thumbs up/down, star ratings, etc.)

    - Scores given to responses from prompts (possibly by humans or heuristics).

- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.

Ask ChatGPT

I am going to give you a post I made, i would like you to copy a similar style and give me a post on Caching for my linkedIn Hey guys!, Lets talk about the CAP theorem. When designing distributed systems, it’s very easy to choose technologies without thinking of their tradeoffs. Take databases for instance, how do we choose between MySQL, MongoDB, Cassandra or Dynamo db? Choosing the wrong technology for your use case could potentially cause issues in the future. This is why an important system design principle called CAP theorem is crucial for every seasoned engineer to know. What is CAP theorem? The CAP theorem states that in a distributed database system, you can only guarantee two out of three characteristics during a network partition. (Consistency, Availability and Partition Tolerance). Taking systems like social media, inventory, banking systems etc, all these systems trade of a particular characteristic to function efficiently. When you make a social media post, it may not reflect instantly across all users, it takes a while for your post to reflect across. A user in a different country may have to wait for probably a few minutes to see that you have posted but who cares?. Many people actually don't notice or see it as a problem. Do you ever hear a social media platform is down? Hardly, very rarely. This is because many social media platforms trade off Consistency for Availability and Partition Tolerance. They prefer to be up all the time but sacrifice a bit of consistency for this if it means waiting a while to sync data across. (AP system) On the other hand, considering trading or inventory systems, when you transfer money, or want to check stock prices, you expect the most up to date data like prices, stock number i.e. show the latest writes. However these systems are prone to downtimes because they would prefer to be down temporarily rather than show outdated information. These systems usually trade off Availability for Consistency. (CP system) When you have fleet of servers that communicate with each other to keep your app running, Partition tolerance simply means that if there's a network problem, the application should function as expected and can still handle and keep doing its job (at least partially). It's more about surviving communication failures in a distributed system. Databases like Cassandra DB works so well with systems that prioritise Availability. Due to its architecture, it is masterless thereby having no central point of failure. However it may take sometime for writes on the DB to reflect across all nodes. Other databases that have high availability is DynamoDB and Couchbase For high consistency applications where data accuracy is crucial, you can consider working with MongoDB (with strong consistency configs), PostgreSQL and FoundationDB, For CA systems, which are rare and only exist on single node system you can use any you want I guess. Hahaa 😅

ChatGPT said:

Absolutely! Here's a LinkedIn post in a similar engaging and educational style, but focused on **Caching in system design**:

Hey guys! Let's talk about **Caching** 🔁

When building scalable applications, one of the most overlooked but *powerful* tools in your arsenal is caching. It helps speed things up for your users and reduce load on your infrastructure. But as simple as it sounds, using it right requires some thought!

What is Caching?
Caching is the process of storing frequently accessed data in a *temporary storage layer* (usually in-memory) so that future requests for that data can be served faster.

Think of it like this:
Imagine asking your friend every time for a Wi-Fi password — annoying right? So what do you do? You write it on a sticky note next to your desk. That's your cache! 🧠

Now why is this important?

Let's say you're building a weather app. Do you really need to call the API every time someone checks the temperature in Berlin every 10 seconds? Probably not. You can store that result in a cache for 5–10 minutes.
Boom 💥 — faster response time, fewer API calls, and happier users.

Types of Caches:

- **In-Memory**: Super fast, stored in RAM (e.g. Redis, Memcached)
- **CDN Cache**: Edge-level caching for static assets (e.g. Cloudflare, Akamai)
- **Browser Cache**: Stored client-side
- **Application-Level Cache**: Caching inside your app logic (e.g. NestJS interceptors)

But it's not all roses… 🌹
Caching introduces complexity — what happens when the data changes? Do you update the cache? Invalidate it? Let it expire?

This is where strategies like **TTL (Time to Live)**, **Write-through**, **Write-back**, and **Cache Invalidation** come into play.

💡 Fun fact: There's a famous quote that says:

> "There are only two hard things in Computer Science: cache invalidation and naming things."

Use caching when:

- You have data that doesn't change often
- You want to reduce database or API load
- You need faster response times

Don't use caching blindly though. Caching stale or inconsistent data can break user trust or lead to serious bugs (especially in things like banking, trading, or real-time analytics).

So next time your app is feeling sluggish, consider asking: *"Can I cache this?"* ⚡

Let me know if you want this post adapted for a specific audience like frontend/backend engineers, or add code snippets (e.g. Redis in NestJS).

You said:

Designed custom reward models leveraging user feedback and prompt scoring; contributed to a 20% improvement in human preference ratings. what does this mean

ChatGPT said:

That sentence means:

"I built specialized models that evaluate responses (reward models), using feedback from users and scoring prompts, which led to responses that humans liked better — improving preference ratings by 20%."

Let’s break it down more clearly:

- **“Designed custom reward models”** → You created machine learning models that assign a score (reward) to outputs (like AI-generated responses) based on certain criteria.

- **“Leveraging user feedback and prompt scoring”** → You trained or tuned these models using data such as:
  - How users rated responses (thumbs up/down, star ratings, etc.)
  - Scores given to responses from prompts (possibly by humans or heuristics).
- **“Contributed to a 20% improvement in human preference ratings”** → After implementing your models, people preferred the AI's responses 20% more often (measured in A/B tests or other evaluation methods).

In short: You improved the quality of AI responses significantly by training models on what people liked or disliked.