# Task 2, Parts 1 and 2

# (Part 1): Design Document

## Task 2: Non-relational Database Design and Implementation

**By David J Kelley**

(dkelle24@wgu.edu)

### Introduction

This assignment involves walking through the database design process for a given business scenario, based on one of two scenarios. In this case, I have selected the second scenario, as was suggested by the assignment, having completed scenario 1 in Task 1. The second scenario can be summarized thus:

In scenario 2, the company, "EcoMart", is a growing theoretical online e-commerce site focused on sustainability, "connecting eco-conscious consumers with ethically sourced and environmentally friendly products" (quoted from assignment material) To support its expanding platform, "EcoMart" is adopting a database design and has engaged a consultant to design a flexible, scalable, and secure data system. The database must handle diverse product attributes, including pricing, availability, certifications, and reviews, while ensuring high performance (both scaling out and scaling up) and reliability under increasing traffic. Additionally, the system must be secure, including proactive monitoring of various aspects of cybersecurity, both for product data and customer data, as well as providing compliance with global laws applicable to this application and the overall business workflow. The design must implement robust security measures and establish proactive monitoring and maintenance workflows to safeguard data integrity and ensure long-term platform stability.

First, I would never recommend only a “non-relational” or NoSQL (such as MongoDB) database solution for this business scenario at least not entirely. Because of several reasons, but mainly these,

1. In a non-relational db you generally do not have good schema enforcement in these kinds of models, leading to data integrity loss and reporting/complexity problems, and making such problems, even if solved, not very performant. While you can have some version of this or enforce it programmatically it is not built in.
2. Poor ACID Transaction Support: ACID stands for Atomicity, Consistency, Isolation, and Durability. Non-relational databases do not perform well with transactions, as they do not guarantee that records will be written and can lose data and transactions. Given that this is e-commerce, I suspect that 'making money' is essential; therefore, we must not lose even one transaction that generates revenue.

Lastly addressing this assignment, while the title is a design document, based on the details, this assignment is not a design document in how it would might be done in a real company at least in Silicon Valley or the Microsoft technology world which would include ERD's, Solution Architecture, user cases, user stories, information architecture and would be at least 40 pages or more and it seems like that is not what is being asked. I will attempt to model my responses around the details of the rubric assignment description.

To clarify, a NoSQL database solution, specifically MongoDB, is utilized to optimize access to reference data and enhance performance and scalability.

### Part 1.Task A - Selecting Scenario 2

As suggested earlier, the scenario selected is scenario 2.

Now, let us define the problem.

### Part 1.A.1 - Business Problem and how it can be solved by the database design

From the description for "EcoMart's" eco-conscious online marketplace, several key business cases would need to be addressed in real life. These business cases represent the core functionalities and value drivers that the database and platform must support to deliver value to customers, partners, and stakeholders. I'll include just a brief overview of these high-level cases and their database implications, and then break down the solution architecture as applied to the NoSQL-type elements that make business sense.

So at a high level, this includes:

1. **Product Discovery and Filtering**

Business Need: Allow customers to easily discover and compare sustainable products.

Database Implications: Efficient querying on product attributes (e.g., sustainability certifications, price ranges, availability). Should be in a NoSQL (MongoDB) database as well as the underlying RDMS.

1. **Sustainability Certification Tracking**

Business Need: Display verified sustainability credentials (e.g., Fair Trade, Organic, Carbon Neutral) for each product.

Database Implications: Store and validate certification data, link products to certifications. Should be in a NoSQL (MongoDB) database as well as the underlying RDMS in terms of eventual parity. This is not 'key' transactional data, but will get updated.

1. **Dynamic Inventory and Availability Management**

Business Need: Keep real-time stock levels and product availability updated across all listings.

Database Implications: Real-time updates, concurrency control, and transactional accuracy. Must go directly to the RDMS to get Real-time data and use a service bus structure for notifications and related DB workflow processing (ie, emailing, invoicing, etc.) This could be cached using the NoSQL (MongoDB), but with transaction processing, these need to be validated against the RDMS.

1. **User Review and Rating System**

Business Need: Collect and display customer feedback to drive purchase decisions.

Database Implications: Store and aggregate user-generated content, link to products and users. While a copy would be persisted in the RDMS, this is primarily reference data with eventual parity using NoSQL (MongoDB).

1. **Personalized Recommendations**

Business Need: Suggest products based on user behavior, preferences, and values.

Database Implications: Capture and analyze user activity, segment user profiles. This involves multiple components, including possible AI processing on the backend against the RDMS database, with cached values pushed up to the NoSQL database (MongoDB) for high-performance referencing.

1. **Order Management and Transaction Processing**

Business Need: Enable seamless purchase, order tracking, and history management.

Database Implications: Ensure secure transactional integrity across multiple tables (products, orders, users, and payments). Key financial transactional database that must be integrated with the RDMS with no risk of data loss. While past transactions would be cached after the fact in the NoSQL DB (MongoDB)

1. **Partner and Vendor Management**

Business Need: Manage product suppliers and their eco-credentials.

Database Implications: Maintain vendor profiles, certifications, and performance metrics to ensure accurate and up-to-date information. This includes both references and limited transactional data, and much of this would be cached in the NoSQL (Mongo) database. However, management functionality, especially workflow, would be against the RDMS.

1. **Compliance and Data Security**

Business Need: Protect user data and meet GDPR, CCPA, and sustainability reporting standards.

Database Implications: Implement role-based access, audit trails, and encryption. PII data and other sensitive data would need to be stored in the RDMS and encrypted both on disk and in transit. All connections over HTTP would use HTTPS (or SSL) with valid certs and/or would use TLS 1.2+ and AES 256 as appropriate.

1. **Marketing and Campaign Analytics**

Business Need: Track performance of promotions, eco-campaigns, and user engagement.

Database Implications: Support analytics on sales, traffic, click-through rates, and campaign success. Must the tacking, logging, and other instrumentation to solve this be done in a MongoDB/NoSQL database?

1. **Scalable Product Catalog Expansion**

Business Need: Onboard new products with varying attributes across categories.

Database Implications: Flexible schema support, attribute-value modeling. The catalog should be cached to support the highest possible performance, with eventual parity between the NoSQL (Mongo) database and the RDMS and logging systems (CloudWatch or similar).

This more or less covers the basic business needs, but they could be further broken down. I'm sure there are more that didn't occur to me, but that said, there are some key elements that can be focused on, which are elements of the general solution architecture centered on the NoSQL component, where we will use MongoDB in these solution elements.

To start with I'm going to assume an AWS based engineering stack is going to be used as NoSQL is more likely to be used and is therefore more relevant to the assignment however in real life that might not necessarily be the case depending on details around the needs of the business, current resources, and other constraints that we don't know.

Essentially, what we will look at is a scalable ecom system using something like a federated data model in terms of the database architecture, meaning transactions are fronted with bus or service broker solution going into the master RDMS which could be Postgress or SQL Server etc but still some variation of RDS to lower support requirements. While considering costing is beyond the scope of the assignment, it would also affect technology selection. In this particular case, let's say we are using Postgres SQL and if we setup and main the server we end up needing a database team as well as other support staff, where as we can lower the personal costs and improve RDMS RDS w/o lower risk even if the license costs are some what more then just licensing the containers but end the end it is cheaper and 'safer' for the business to use Postgress RDS which abstracts hosting and other maintenance and performance issues so we don't need to deal with it and provides a reasonable basis for DR or business continuity.

Given that we will recommend a service-based site, using client-side session management and serverless architecture for inventory, with NoSQL without session management, and S3 for static components of the client application. Let us take a look at this diagram.

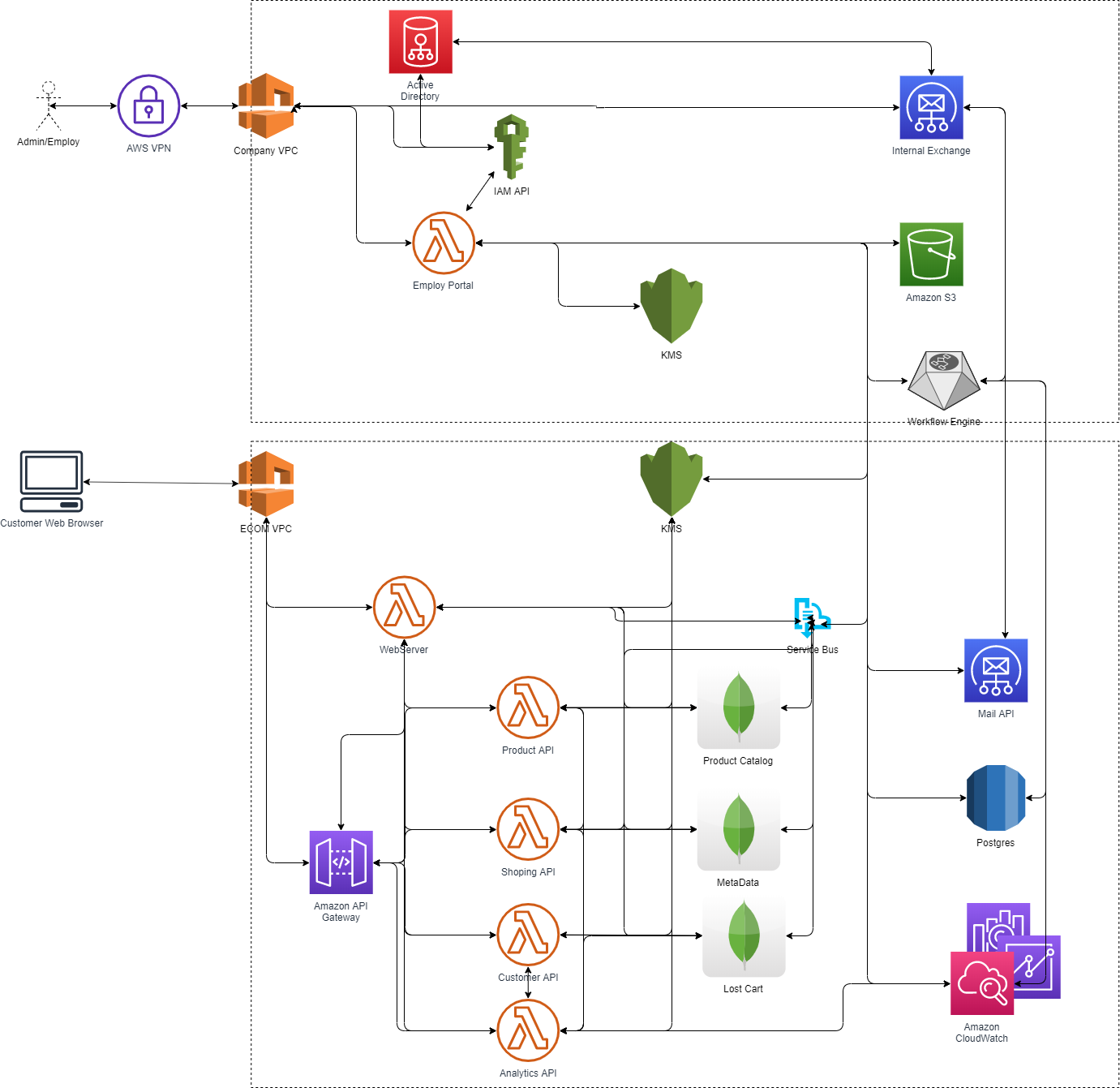


Figure 1A1 – Basic Solution Architecture. See attached document(s) [D597 Task 2 – Solution Arch.drawio] [D597 Task2 – Solution Arch.png]

While possibly overly simplified for a high-end, scalable e-commerce site, this does show the realistic, relative locations of the key database architecture in this case, of both Postgres RDS and MongoDB (AWS Atlas), as might be applied to an AWS engineering stack. A similar approach would apply to Azure or GCP, depending on other technical requirements that are currently unknown; however, the architecture would be fundamentally identical in general. From a theoretical point of view, this demonstrates both a cloud-based company infrastructure and the e-commerce structure. This has a lot to do with security, isolation, redundancy, automation, and many of the elements of the business cases articulated earlier.

### Part 1.A.2 Justify why a NoSQL database solution.

In this case, we are focusing on database architecture, and we are using Postgres RDS as the underlying database. We are implementing a federated model by starting with the service bus or service broker infrastructure, along with the workflow engine, which allows us to scale and perform long-running, stateful operations related to the business. That said, the master database or databases, if federated, is the RDS Postgres that I would design in third normal form, plus probably second normal form type views. Additionally, there would be the service broker infrastructure and associated views feeding all of the read data, which would be fronted with MongoDB. MongoDB would be when looking at the over all data model key read only silo's that would be get updated as needed but don't necessarily need to be up to the second in real time. This would allow the system to scale overall by pushing session management to the client, pending the company's budget. That said, to really push a system like this to that degree, the company is likely to be selling a lot. This is a similar, albeit simplified, version of Amazon, which allows us to scale similarly.

Let's take a look at the following Database ERD (Entity Relationship Diagram):

A diagram of a computer

AI-generated content may be incorrect.

Figure 1A2 – High Level ERD for explanation purposes only (Incomplete). See attached document(s) [D597 Task 2 - ERD.drawio] [D597 Task 2 – ERD.png]

This is by no means a comprehensive ERD of the solution I'm proposing. Still, it serves as an example of key elements I'll use to explain the solution architecture. You will note that the tables are color-coded to indicate their type, and a legend is provided in the upper right.

Most of the tables in this database are not concerned with transactions against them. For example, green is reference data that will very rarely change. The yellow-colored tables are relational, and white and orange are key elements, but they are unlikely to change often. All of these tables, as long as they eventually have parity with caching, are likely not to cause any issues.

The red tables, however, are key transactional tables that cannot be risked, no matter what; we hope they will change often if the business is to succeed. These will need to be sent to the bus and will be pushed to the database, where they will be checked for a successful right, and the workflow is likely to be fired off.

The last color views are not really tables as such but are processed from various queries and populated as the underlying data changes. These also do not need to change in real-time and are not transactional. That means that everything but the red tables and the white table can be cached on the API and web servers in Mongo database directly and as changes happen pushed to mongo and mongos format makes it easier to process in the client as JSON so less work means faster rendering, faster process and generally a better user experience.

### Part 1.A.3 Identify a Specific Type of DB.

It appears that the previous sections were not considered as part of the A3 response so I’m redoing this to included all the relevant information about the type of database we are talking about even though the previous sections largely answer this it seems little detailed is needed in this section but apparently that assumption was wrong.

The “A3” segment of Part 1 says, “Identify a NoSQL database type to solve the identified business problem.”  
  
As I understand this, we need to first break down the business problem, then identify the solution architecture and reasoning for why we select this specific type of database. However, a more distinct summary will be first and then I’ll restate that at the end.

To summarize and focus on the non-relational, NoSQL components of the proposed solution:

I selected MongoDB which is a document-oriented NoSQL non-relational database, meaning it stores data in flexible, semi-structured documents using JSON-like (BSON, binary JSON) format. This makes it well-suited to manage data with varying attributes, such as EcoMart’s (See assignment Scenario 2) diverse product catalog, user-generated reviews, and dynamic metadata. Unlike key-value stores or column-family NoSQL database types, document databases support rich querying and indexing on embedded fields, enabling high performance filtering on attributes like certifications, ingredient tags, or product types. MongoDB’s flexible schema also supports rapid iteration and scaling without the need for rigid schema migrations (this does tend to cause data integrity problems from a normalization standpoint), which is especially beneficial for startups or evolving e-commerce platforms. Additionally, MongoDB supports horizontal scaling via sharding (Siloing) and replica sets for high availability, making it a strong choice for non-transactional, high-volume read-heavy use cases such as product discovery, customer reviews, and real-time caching of inventory data etc..

I, of course, have identified MongoDB as our type of NoSQL, non-relational document DB we will use because the class is focused on it and in the previous sections I justified it as I will do again. However, even if that were not the case, there is a strong case for MongoDB among NoSQL databases in general. For me, selecting Mongo or Redis or some other database examples would generally be determined by the company's needs. Still, I would not normally recommend something other than MongoDB unless I'm rolling my own or there is some other business need that would make something other than Mongo a good idea. For example, Azure Cosmo DB can use JSON as the underlying format. I might utilize it as a graph database and NoSQL, but I wouldn't use it for caching cases or other high-performance read use cases like what I would use Mongo for. Further Cosmo for example also is specific to Azure so it is not very multi cloud or cloud agnostic friendly.

To restate, Part 1 justifies why I would use the type of database (non-relational, NoSQL, Document database specifically MongoDB) in specific areas of the overall solution as well as some other RDBMS’s to complete the business case.

The summary business case as it relates to this database type selection:

From the description for "EcoMart's" eco-conscious online marketplace, several key business cases would need to be addressed in real life. These business cases represent the core functionalities and value drivers that the database and platform must support to deliver value to customers, partners, and stakeholders. I'll include just a brief overview of these high-level case only so much as they are related to the NoSQL selection, but you can review the rest in the previous sections of this paper. The ones specific to this selection and their database implications applied to the MongoDB Document JSON (BSON) Database NoSQL-type elements that make business sense from a solution architecture standpoint.

So at a high level, this includes:

1. **Product Discovery and Filtering**

Business Need: Allow customers to easily discover and compare sustainable products.

Database Implications: Efficient querying on product attributes (e.g., sustainability certifications, price ranges, availability). Should be in a NoSQL (MongoDB) database as well as the underlying RDMS (Relational Database Management System).

1. **Sustainability Certification Tracking**

Business Need: Display verified sustainability credentials (e.g., Fair Trade, Organic, Carbon Neutral) for each product.

Database Implications: Store and validate certification data, link products to certifications. Should be in a NoSQL (MongoDB) database as well as the underlying RDMS in terms of eventual parity. This is not 'key' transactional data, but will get updated.

1. **Dynamic Inventory and Availability Management**

Business Need: Keep real-time stock levels and product availability updated across all listings.

Database Implications: Real-time updates, concurrency control, and transactional accuracy. Must go directly to the RDMS to get Real-time data and use a service bus structure for notifications and related DB workflow processing (ie, emailing, invoicing, etc.) This could be cached using the NoSQL (MongoDB), but with transaction processing, these need to be validated against the RDMS.

1. **User Review and Rating System**

Business Need: Collect and display customer feedback to drive purchase decisions.

Database Implications: Store and aggregate user-generated content, link to products and users. While a copy would be persisted in the RDMS, this is primarily reference data with eventual parity using NoSQL (MongoDB).

1. **Personalized Recommendations**

Business Need: Suggest products based on user behavior, preferences, and values.

Database Implications: Capture and analyze user activity, segment user profiles. This involves multiple components, including possible AI processing on the backend against the RDMS database, with cached values pushed up to the NoSQL database (MongoDB) for high-performance referencing.

1. **Order Management and Transaction Processing**

Business Need: Enable seamless purchase, order tracking, and history management.

Database Implications: Ensure secure transactional integrity across multiple tables (products, orders, users, and payments). Key financial transactional database that must be integrated with the RDMS with no risk of data loss. While past transactions would be cached after the fact in the NoSQL DB (MongoDB)

1. **Partner and Vendor Management**

Business Need: Manage product suppliers and their eco-credentials.

Database Implications: Maintain vendor profiles, certifications, and performance metrics to ensure accurate and up-to-date information. This includes both references and limited transactional data, and much of this would be cached in the NoSQL (Mongo) database. However, management functionality, especially workflow, would be against the RDMS.

1. **Compliance and Data Security**

Business Need: Protect user data and meet GDPR, CCPA, and sustainability reporting standards.

Database Implications: Implement role-based access, audit trails, and encryption. PII data and other sensitive data would need to be stored in the RDMS and encrypted both on disk and in transit. All connections over HTTP would use HTTPS (or SSL) with valid certs and/or would use TLS 1.2+ and AES 256 as appropriate.

1. **Scalable Product Catalog Expansion**

Business Need: Onboard new products with varying attributes across categories.

Database Implications: Flexible schema support, attribute-value modeling. The catalog should be cached to support the highest possible performance, with eventual parity between the NoSQL (Mongo) database and the RDMS and logging systems (CloudWatch or similar).

As noted, this is most of the basic use cases implied in the scenario that have direct implications as to the database type selection. For there elements of the over all solution architecture MongoDB BSON (JSON) based Document NoSQL database is very good at high-speed reference data and scalability and helps the solution architecture articulated in sections A1 and A2 effective.   
That said as implied MongoDB’s BSON (JSON) based Document database NoSQL system doesn’t fill all the requirements to address the core business cases but it helps solve a lot of them as part of a comprehensive ecommerce solution architecture.

**Database Type Classification:**

MongoDB is a NoSQL, non-relational, document-oriented database. In the NoSQL family, “document store” databases organize and store data as collections of flexible, semi-structured documents, rather than rows and columns. Documents are encoded in BSON (Binary JSON), allowing storage of complex, nested data structures. Unlike key-value stores, which associate a single value with a key, or column-family stores, which store data in a distributed table-like format, document databases allow each record (document) to have a unique structure, making them ideal for handling evolving, heterogeneous data. MongoDB supports indexing on fields within documents, ad-hoc queries, and aggregation pipelines, distinguishing it from other NoSQL types such as graph databases like Neo4j or time-series databases like InfluxDB or ones like Cosmo mentioned earlier.

**Summary for A3**

EcoMart’s eco-conscious online marketplace requires a database solution capable of handling diverse and dynamic product data, sustainability certifications, customer reviews, and personalized recommendations, all while maintaining high performance and scalability. The chosen solution is MongoDB, a document-oriented NoSQL, non-relational database that stores data in flexible JSON-like (BSON) documents. Its schema flexibility supports rapid changes and varied product attributes, while rich querying and indexing on embedded fields enable efficient product filtering and discovery. MongoDB also provides horizontal scaling via sharding, high availability through replica sets, and is particularly suited for high-volume, read-heavy, non-transactional workloads. Compared to alternatives like Redis (primarily for caching) or Azure Cosmos DB (vendor-locked), MongoDB offers a cloud-agnostic, general-purpose NoSQL BSON (JSON) based Document database system option that integrates well with an RDBMS for hybrid architectures.

Within the overall solution architecture, MongoDB addresses several key business needs. It powers fast product searches, tracks and links sustainability certifications, caches real-time inventory data from the RDBMS, stores and aggregates user reviews, and serves personalized product recommendations. It also supports scalable catalog expansion by accommodating products with varied attributes and serves as a cache for partner/vendor information and historical order data, while the RDBMS handles transactional integrity and sensitive information storage for compliance with GDPR and CCPA. In this hybrid setup, MongoDB operates as a high-speed reference data store, improving performance and scalability while complementing relational systems to meet EcoMart’s operational and business goals.

### Part 1.A.4 How the business data will be used within the database solution

This was also articulated earlier, with a key model on the backend. Still, all product, reference data, and custom view data are preprocessed and loaded into MongoDB for high-speed access. Even these I would go so far as to silo into functional groups of metadata, product data, customer data, and analytics. Analytics is actually a special case and would not necessarily be pushed into RDS/Postgres; however, MongoDB would be used to support analytics references. Log data, on the other hand, would be pushed back to AWS CloudWatch or another analytics system. This is not critical transactional data, even though it is transactional, and there is a lot of it. However, I'd be more likely to store it in a data lake or in the analytics system for analysis, separate from RDS. This would allow for greater flexibility in processing using various solutions, such as Snowflake and Datadog.   
I also mentioned a federated model, which is also applied to the RDMS. There are some logistical issues with this, such as the fact that some foreign keys and references would need to be programmatically enforced at scale; however, this is probably not necessary for this business initially. I would very much doubt they have that much transaction data to make it essential in a solution like this for a medium to small business. That said, if the scale was needed, then the ERD would be split among things like customer data, order data, and product data, and they would be in their own RDS instances in a federated model.

### Part 1. B. Explain how the proposed database design…

To really explain the proposed database design and how this solution architecture addresses database scalability and performance, there are several strategies:

1. Normalized RDS RDMS layer that can be set to automatic scale and ensure transactional integrity.
2. Catching data that is not transient or critical, including product data.
3. Siloing NoSQL/Cashing to increase the ability to scale
4. Ensure the ability to federate the underlying model if needed to scale out and up.

There is a lot more to the particular type of solution architecture in general. Still, it follows best practices for this kind of system, meaning dynamic scaling and session state managed on the client, so APIs, Services, and the like can be scaled on the fly. Additionally, features like RDS ensure continuity in the event of a failure.

Most of the data, as long as we get 'eventual consistency', we are not too concerned about manipulating it. Hence, we need to scan it, which is where NoSQL and MongoDB, in this case, really come in handy. The fact that it is in JSON by default makes it faster and easier for the JavaScript (ECMA, plus React.js in this case) in the UI to process it, thereby making the UI fast and easy to use.

### Part 1. C. Output the privacy and security measures

Privacy and security in this solution architecture and database design adhere to best practices. Basic rules include:

1. No Service Accounts.
2. Only SSL
3. Backend mapped to IPs inside the VPCs
4. Getting to the backend requires IAM-based authentication
5. Using Security secrets for everything (ie, connection strings, etc. )
6. Pushing authentication down through the stack based on the user. For example, if I log in as a user and I somehow figure out how to make a call to the database via an API, the API will check to see if I have the right to whatever data I'm trying to get to
7. Would we need to consider other requirements, such as GDPR or SOC2, based on business needs that were not clear in this regard? For example, if we are selling in Europe, GDPR would be an issue, and we would have to either extend the data to live in the EU or move the entire VPC to the EU to be compliant.

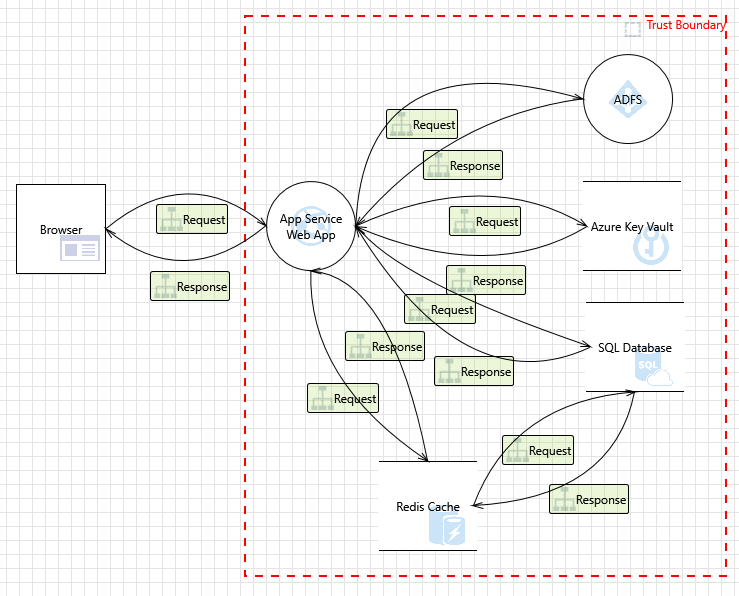
Generally, as part of the solution architecture, I would have a threat model and assessment done. Take a look at this example: 

Figure 1C – Example website security threat model. See attached document (s) [D597 Task 2 – Threat Model.png] [(\*.tm7 file not included but available on request.)]

This is a screenshot of a threat model I generated for the last assignment using the Microsoft Threat Modeling tool. The model essentially generates a checklist of things we would need to look at, based on the complexity and previous experience. There are at least 100 attack services in the solution architecture I articulated. Issues include, but are not limited to:

1. Data Breaches from unauthorized access
2. Misconfiguration
3. Data Leakage from Insecure Backups
4. Insecure data transmission
5. Compliance Violations

That said between isolating the databases and even isolated the company AD in a VPC, limited public access to single endpoints, ip white-listing, enforcing SSL/TLS only, enforcing IAM based least privilege service account access, known security key storage (KMS), rich logging and monitoring (maybe use cloud trail or something like that or a third party monitoring service, but also WAF's. A lot of this will also depend on the culture and setup of the company's team supporting this, including DevOps and the like. If the company is not going to have a team handling this, then they should at least have a third-party support contract.

# Part 2 - Introduction

While the title of the assignment for Task 2 Part 2 implies creating the entire solution, there are only two source data files, and the sub-instructions are only creating elements of the proposed design. Aligning with the detailed aspects of Part 2, Items D 1 through 4, are the following outcomes.

### Part 2.Task D. Items 1 and 2 - Writing script, creating DB, and importing

The description here is unclear about what the script should be written in, so I thought I would use something different from what I did in Task 1. In Task 2, I used C#, and here I used an old-fashioned batch file, or rather, two of them. Here is the source:

@echo off

REM "C:\Program Files\MongoDB\Server\7.0\bin\mongoimport.exe"

set DB\_NAME=D597Task2

set COLLECTION\_NAME=cosmetics

set FILE\_PATH=C:\Scenario2\Task2Scenario2Dataset1\_cosmetics.json

"C:\Program Files\MongoDB\Server\7.0\bin\mongoimport.exe" --db %DB\_NAME% --collection %COLLECTION\_NAME% --file %FILE\_PATH% --jsonArray

pause

@echo off

REM "C:\Program Files\MongoDB\Server\7.0\bin\mongoimport.exe"

set DB\_NAME=D597Task2

set COLLECTION\_NAME=groceries

set FILE\_PATH=C:\Scenario2\Task2Scenario2Dataset2\_Groceries\_dataset.json

"C:\Program Files\MongoDB\Server\7.0\bin\mongoimport.exe" --db %DB\_NAME% --collection %COLLECTION\_NAME% --file %FILE\_PATH% --jsonArray

pause

This completes writing a script as seen above. Below, we will examine the scripts before and after running. See the attached \*.bat files named [D597 Task 2 - Part 2 D1n2.bat] [D597 Task 2 - Part 2 D2plus.bat]

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2A – Shows MongoDB before Batch files are run to do the imports.  
  
And here are the screenshots in the WGU environment, note the creation of the database and successful imports:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2B- First Batch file used to create the db and do the first import

A computer screen with a computer screen

AI-generated content may be incorrect.

Figure 2C – Showing the results of the first batch file import with the data in the db after the db was created.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2D – Second Batch file with import of the second source file.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2E – Batch file import complete into the NoSQL DB (MongoDB), showing results

See files attached files: [D597 Task 2 – Running 1.png][ D597 Task 2 – 1 Complete.png][ D597 Task 2 – Running 2.png][ D597 Task 2 – 2 Complete.png]

Now we will look at 3 queries

### Part 2.D.3 - Queries

The following queries will display the code and usage in the MongoDB GUI tool, Compass. The queries are JavaScript, whereas the usage in Compass is more JSON-like than anything else. They demonstrate three variations of the types of queries that would be required to implement the system as described earlier.

**Query 1: Top Moisturizers for all skin types**

db.cosmetics.find({

Label: "Moisturizer",

Combination: 1,

Dry: 1,

Normal: 1,

Oily: 1,

Sensitive: 1

}).sort({ Price: -1 }).limit(5)

In Compass:

Filter:

{

"Label": "Moisturizer",

"Combination": 1,

"Dry": 1,

"Normal": 1,

"Oily": 1,

"Sensitive": 1

}

Project:

{

"Brand": 1,

"Name": 1,

"Price": 1,

"\_id": 0

}

Sort:

{ "Price": -1 }

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3A – Running Query

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3B – Query 1 Performance Summary

**Query 2: Top Grocery Items by Frequency**

db.groceries.aggregate([

{

$group: {

\_id: "$itemDescription",

count: { $sum: 1 }

}

},

{ $sort: { count: -1 } },

{ $limit: 5 }

])

"Aggregations" tab

Stage 1: $group

{

\_id: "$itemDescription",

count: { $sum: 1 }

}

Stage 2: $sort

{

count: -1

}

Stage 3: $limit

5

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3C – Running Query 2

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3D – Performance stats on Query 2

**Query 3: Cosmetics with natural ingredients**

Note: that this is using REGEX, which is a string search markup/syntax

{

"Ingredients": /fruit|seed|leaf|extract|algae/i

}

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3E – Query 3 Results

A screenshot of a computer

AI-generated content may be incorrect.

Figure 3F – Query 3 Performance Results

Screen shots are included in files: [D597 Task 2 - Query 1.png] [D597 Task 2 - Query 2.png] [D597 Task 2 - Query 3.png] [D597 Task 2 - Query 1 Results.png] [D597 Task 2 - Query 2 Results.png] [D597 Task 2 - Query 3 Results.png]

### Part 2.D.4 – Query Optimization

Referring to the queries from the last section in order, the following are the basic optimizations I would do for them if they were used by the website, for example.

Query 1, I would add a compound index for the query fields and a sort. So, maybe filter by skin type and label, and sort the 'price' field in descending order. The trick is really avoiding sorting in memory.

Example:

db.cosmetics.createIndex({

Label: 1,

Combination: 1,

Dry: 1,

Normal: 1,

Oily: 1,

Sensitive: 1,

Price: -1

})

A screenshot of a computer

AI-generated content may be incorrect.

Figure 4A - Query 2 Optimizations Results

Query 2, here I would ad an index on itemDescription to reduce scanning and I would ensure 'allowDiskUse: true' to enable large aggregations.

Example:

db.groceries.createIndex({ itemDescription: 1 })

db.groceries.aggregate([...], { allowDiskUse: true })

A screenshot of a computer

AI-generated content may be incorrect.

Figure 4B – Query 2 Results Post Optimizations

Query 3: I might avoid Regex if possible. Even full-text searching is faster if it can be done. That said, I could parse the ingredients at ingest and then do something like this:

db.cosmetics.createIndex({ IngredientTokens: 1 })

And then run the query like this (via command prompt):

db.cosmetics.find({

IngredientTokens: { $in: ["fruit", "seed", "leaf", "extract", "algae"] }

})

A screenshot of a computer

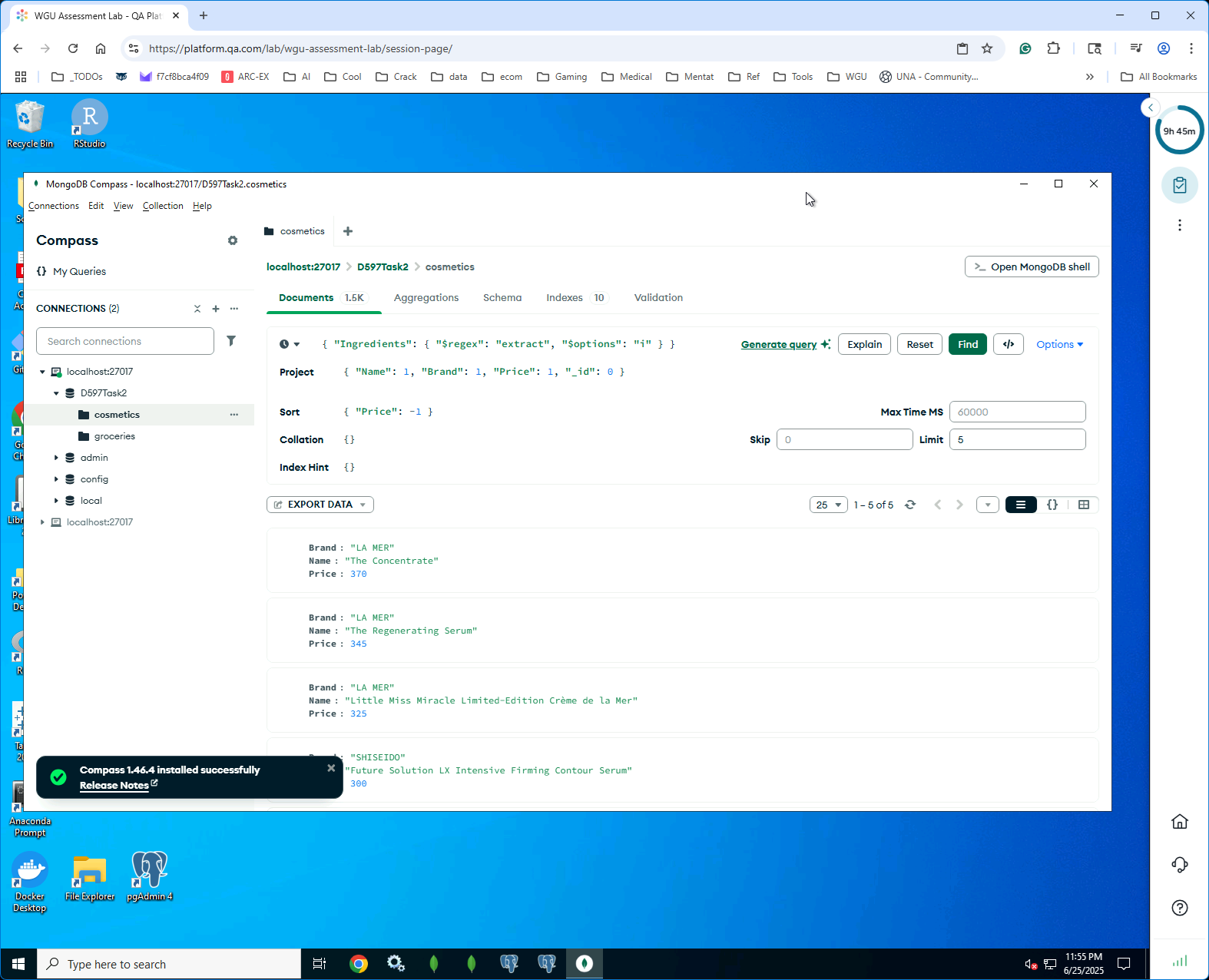
AI-generated content may be incorrect.

Figure 4D – Query 3 Post Op Results  
  
These all work to some degree, but without testing in detail, it would take a while to land on the best optimizations. However, this is the kind of approach I would take by creating queries based on the case, then indexing based on the needed queries, and further optimizing under performance testing.

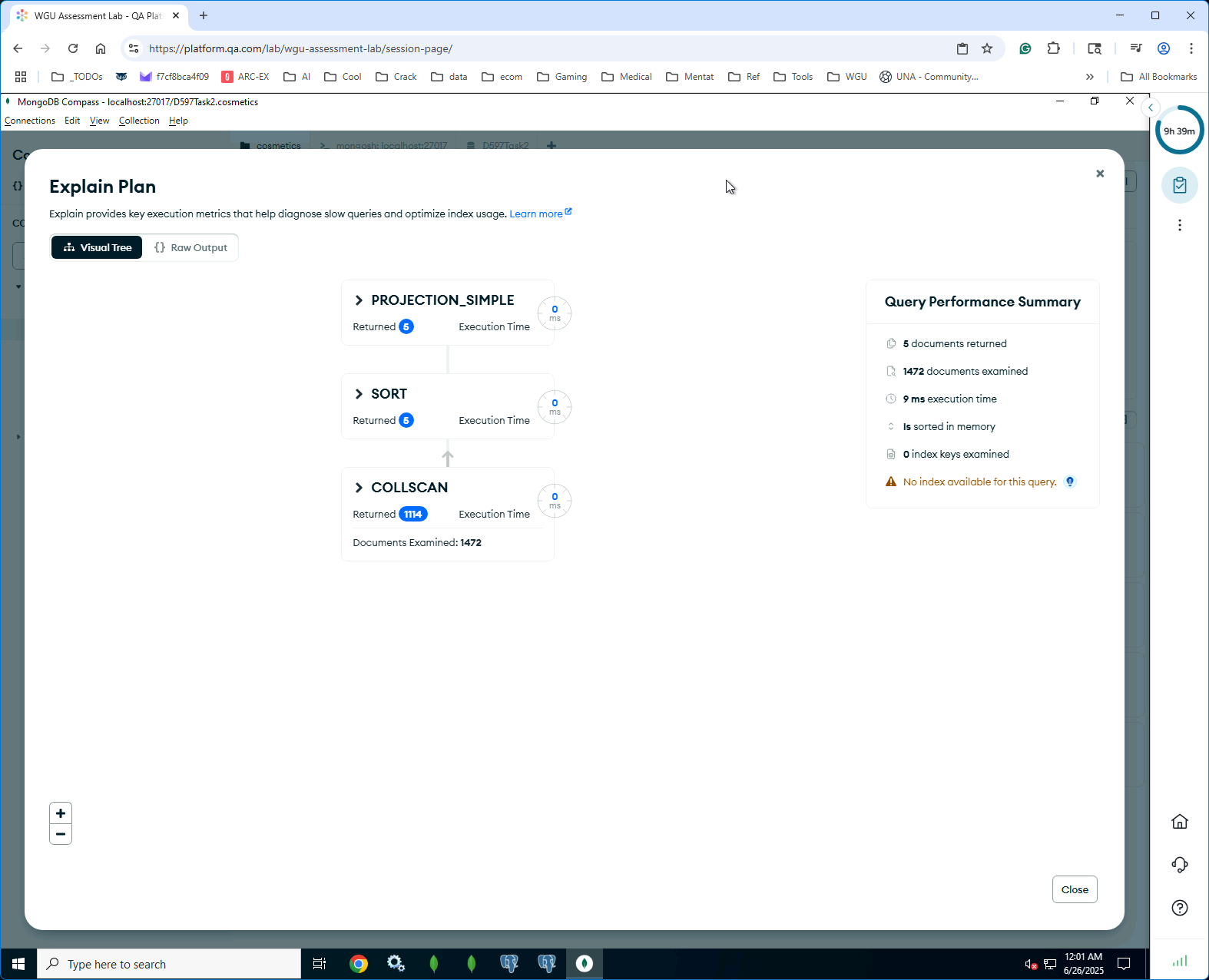
In terms of what I actually achieved with these optimizations on Query 1, the execution time doubled, probably because there are too many indexes and not enough data to make it worthwhile. It scans fewer documents, and there are elements in the query plan that are faster, but that notwithstanding, this is only partially successful.   
Looking at query 2, this result was interesting because I was performing a number of aggregations in the query, but that notwithstanding, the optimizations still allowed it to execute faster. On query 3, it was also faster to not use REGEX in the query, so times were much quicker.

**Query 4 – Cleaner version of Query 1**

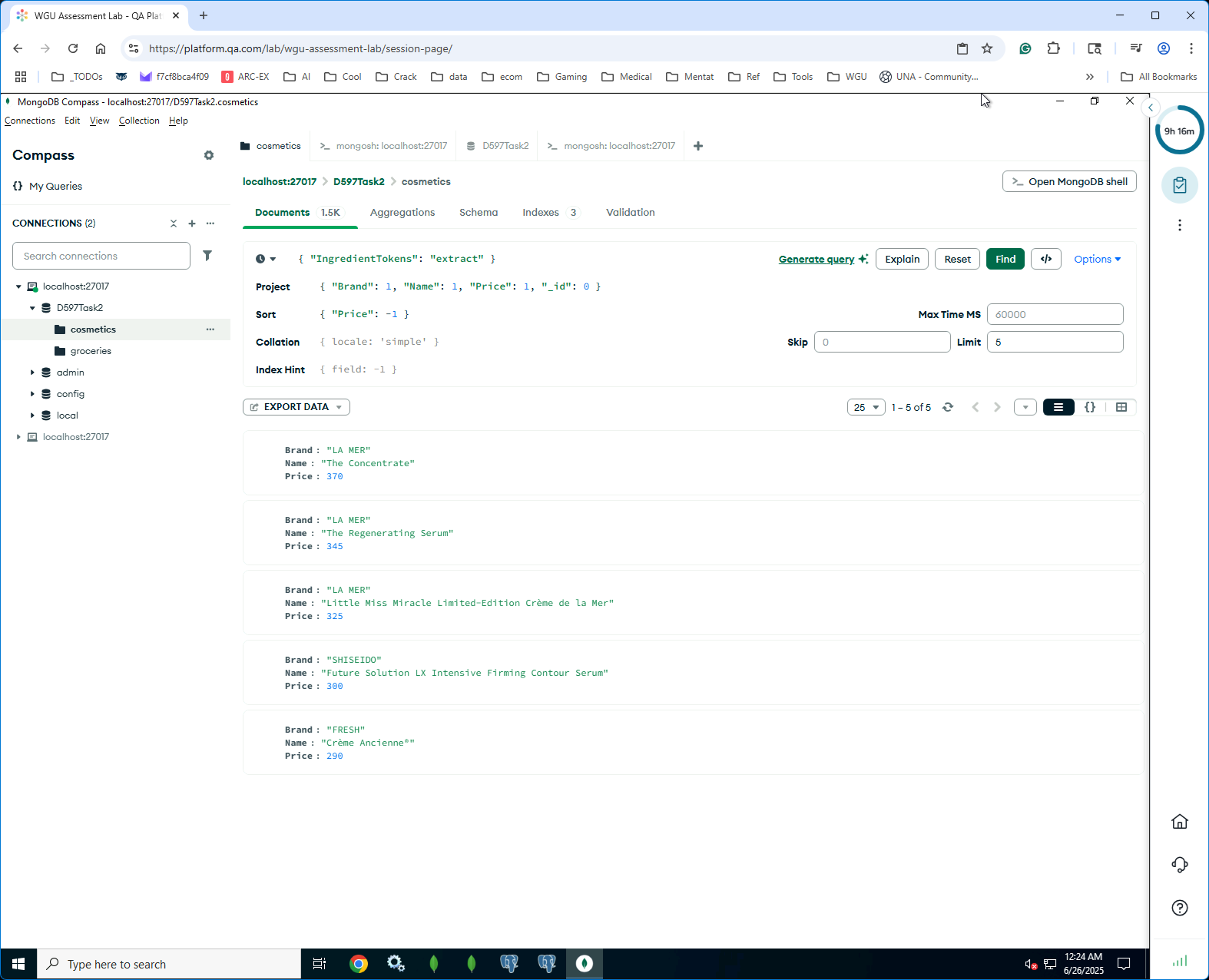
Examining these results seemed somewhat ambiguous, so I made some additional queries that are less ambitious but yielded clear results. This one went back to the beginning and worked on a query that searched for ingredients again. The first one in Compass was this:

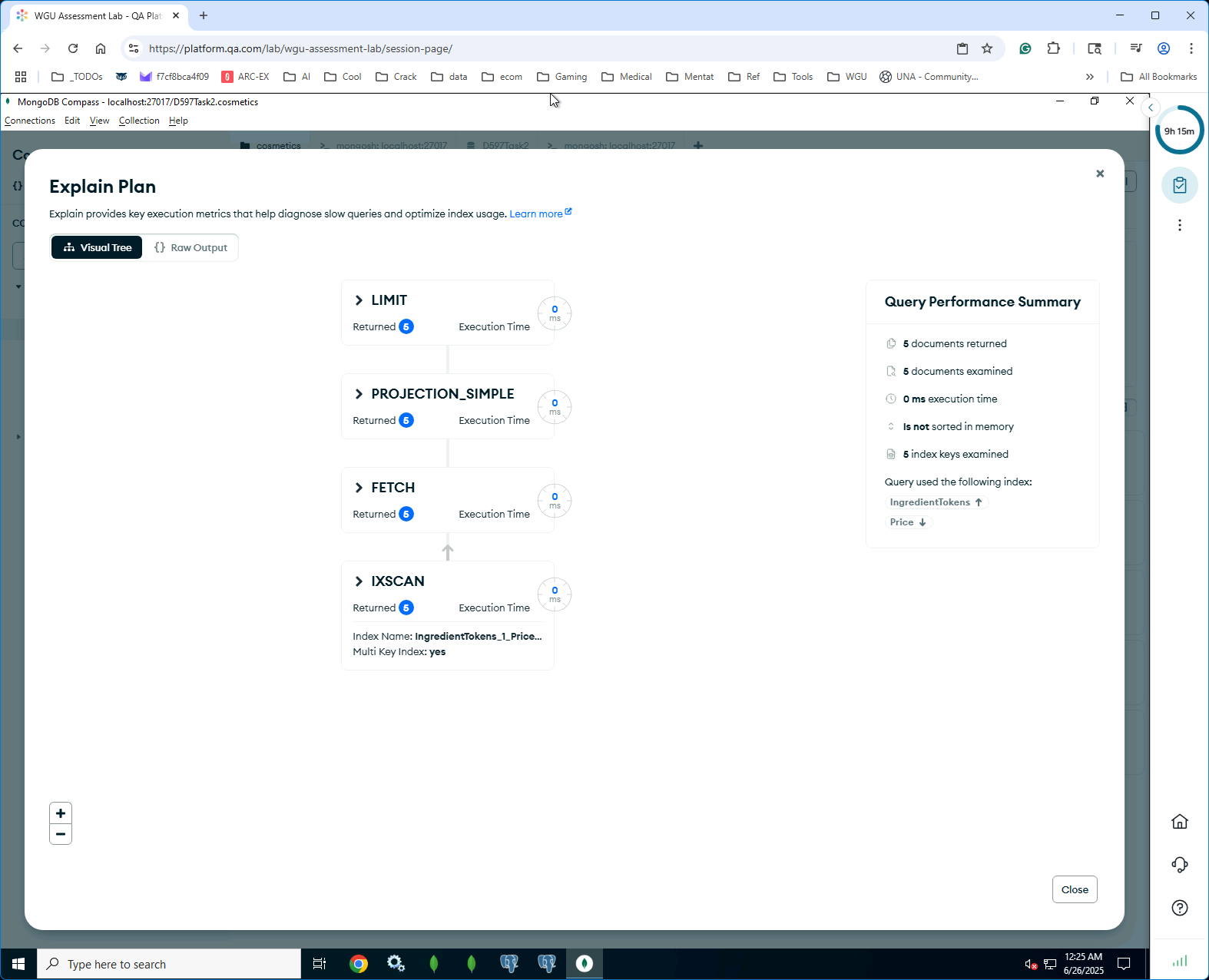


. Looking at the performance, we get:



This is clearly 9ms in terms of execution time. By creating just one index for price and a compound index. I ended up having to do this at the command line and then when updating the query like this:

  
  
And we get our 5 results and look at the details, we have an execution time of 0 ms, which was much clearer to me.



Overall, it is clear that the queries that were actually faster 'after' optimization using indexes and better queries, rather than forcing table scans and the like.

These screen shots are also attached 1 through 4 as PNG’s.

## References:

No references were used in this document.