A white circle and blue circle

Description automatically generated

**Version: draft**

**Last Updated: 01/2025**

**Author(s): Benjamin Roberts**

End Point Assessment -

Level 4 Software Development

Benjamin Roberts

# Table of Contents

[Table of Contents 2](#_Toc188360828)

[Table of Figures 3](#_Toc188360829)

[Introduction 4](#_Toc188360830)

[Discovery 5](#_Toc188360831)

[Relational Databases 5](#_Toc188360832)

[Keys 5](#_Toc188360833)

[Data Relationships 6](#_Toc188360834)

[Normalisation 6](#_Toc188360835)

[Non-Relational (NoSQL) Databases 7](#_Toc188360836)

[Vs Relational Databases 10](#_Toc188360837)

[Current Architecture 11](#_Toc188360838)

[Overview 11](#_Toc188360839)

[Databases 12](#_Toc188360840)

[Connecting Data to User Interfaces 12](#_Toc188360841)

[Strengths and Weaknesses 16](#_Toc188360842)

[Alternative – Microservice Architecture 17](#_Toc188360843)

[Proposed Architecture 18](#_Toc188360844)

[How to Implement 19](#_Toc188360845)

[Conclusion 21](#_Toc188360846)

[Bibliography 22](#_Toc188360847)

# Table of Figures

[Figure 1 - Example of a Monolith platform (Atlassian, 2024) 4](#_Toc188360744)

[Figure 2 - Example of tables in a relational database, where a Foreign Key "EmployeeId" is used to relate the tables (Pluralsight, 2020) 5](#_Toc188360745)

[Figure 3 - Example of data storage in a Document store, a type of non-relational database (Microsoft Learn, 2024) 8](#_Toc188360746)

[Figure 4 - Visual representation of a Key-Value store (Microsoft Learn, 2024) 9](#_Toc188360747)

[Figure 5 - Visual representation of a column database (Amazon AWS, 2024) 9](#_Toc188360748)

[Figure 6 - Representation of a graphical database (Microsoft Learn, 2024) 9](#_Toc188360749)

[Figure 7 - Current Architecture for calling External Data Providers 11](#_Toc188360750)

[Figure 8 - Snippet of the Decisioning Engines HTTP Response, which the front-end references to control the next screen shown to applicants 12](#_Toc188360751)

[Figure 9 - Front-end screen shown to applicants for a "Progress" result. 13](#_Toc188360752)

[Figure 10 - Check Record in the Front-End system, saving a record of the decision(s) received from the decisioning system 13](#_Toc188360753)

[Figure 11 - Multiple Check records created in the Front-End system 13](#_Toc188360754)

[Figure 12 - Additional blocks of data within the decisioning engine HTTP Response 14](#_Toc188360755)

[Figure 13 - Check record created from the Scorecard data from the decisioning engine's HTTP response 14](#_Toc188360756)

[Figure 14 - Check record created from the External Info data from the decisioning engine's HTTP Response 14](#_Toc188360757)

[Figure 15 - Visual representation of a Microservice Architecture (Atlassian, 2024) 16](#_Toc188360758)

[Figure 16 - Proposed changes to Architecture, focusing on reusing API Integrations 17](#_Toc188360759)

[Figure 17 - JSON Schema for an Experian API Product as seen within our low-code decisioning engine 18](#_Toc188360760)

[Figure 18 - Example of a RegEx match function being used on a variable within this parsed JSON schema 18](#_Toc188360761)

[Figure 19 - Proposed JSON Schema for the Microservice Payloads 19](#_Toc188360762)

# Introduction

Portfolio Formatting

- Cover page - name, date, professional design, company branding

- Table of contents

- Overall introduction - Introduce yourself, your role, your company, the apprenticeship etc.

- Apply activities inserted one after another

- Any further evidence they wish to include (communication, problem solving, creative thinking etc if it is relevant to the workplace)

- Overall reflection - talk about what went well, the challenges, future plans etc...

Consider the following;

- Page numbers

- Header, footer, subheadings

- Consistent text font, size, style

# Apply 1 - Java Decisioning Program

# Apply 2 - React Web App – Decisioning Dashboard

# Apply 3 - Application Testing

# Apply 4 and 5 – DevOps Framework End 2 End

# Apply 6 - Microservices, Databases and API’s

## Introduction

Currently our Credit Decisioning system is set up as standalone “**monolithic**” platforms, a traditional architecture approach where all components of the software are compiled into a single network and code base, tightly coupling it all together (Atlassian, 2024). The figure below shows this visualised:



Figure 1 - Example of a Monolith platform (Atlassian, 2024)

This has been useful to date for getting our platforms ready for production on tight project timelines but has led to issues. Specifically, we see occurrences where we integrate with multiple HTTP API’s provided by 3rd parties across multiple platforms, but each platform uses its own code base to do this, leading to small differences in implementation.

This paper will examine the current architecture and tooling, look for potential issues and what alternatives are possible.

## Discovery

Before exploring existing architecture, topics relevant to examining the current decisioning system architecture are explored in the sections below.

### Relational Databases

A Relational Database (full name “Relational Database Management System", or RMDBS when abbreviated) is a storage solution that stores data in rows and columns, across various tables within the database. Examples of these types of databases are:

* Microsoft SQL Server
* MySQL
* OracleDB
* SQLite

The rows and columns make up “records” within the tables. Columns are used to capture specific data points, while rows hold the value for those columns. The table structure allows the data to be structured logically e.g. columns that all relate to an order on an e-commerce site (e.g. identifier, datetime) would all be in 1 table.

#### Keys

Tables relate to each other through “keys”. Different types of keys are used to structure data:

* **Primary Keys** – Column(s) in the table that uniquely identify a record. Ideally this would be a single column within the table, but it can be made up of multiple fields (these types being called “Composite Keys”
* **Foreign Keys** – A column within a table that references a record within another table. These are used to join tables to each other as needed e.g. analytics. When a column is declared as a foreign key, it references another column in a different table to establish a relationship with it. Once in place, values for the foreign key much match the references column in the other table.

The figure below visualises how these are used to relate tables. In it, “EmployerID” is the Primary Key of the “Employees” table. This is also referenced in the “Sales” table.

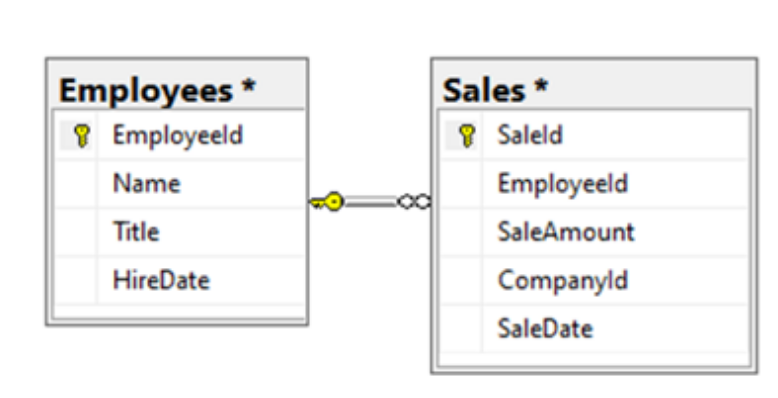


Figure 2 - Example of tables in a relational database, where a Foreign Key "EmployeeId" is used to relate the tables (Pluralsight, 2020)

#### Data Relationships

Tables within a relational database can have many types of relationships, which dictate the result you’d expect when joining tables together. At the high level, these relationships are:

|  |  |
| --- | --- |
| **Type** | **Description** |
| One-to-One | Records within Table A will join onto exactly one record from Table B, and no more than that.  Where it is possible there is no matching record in Table B, we call that a One-to-Zero or One relationship. |
| One-to-Many | One record within Table A will join onto ether one or multiple records from Table B (dependent per record).  The “to-Many” part of this relationship is strictly 1 way e.g. the multiple Records in B will only join onto 1 Record in Table A. |
| Many-to-Many | Multiple records within Table A will join onto multiple records from Table B.  This type of relationship should be discouraged, as they lead to complex joins that require significant computational power to calculate.  One way to avoid this relationship is to create a table between the 2 tables, each with a One-to-Many relationship to this new table. For example: Table A and Table B have a Many-to-Many relationship. To avoid this, we create a Table C and create a One-to-Many relationship to it for both Table A and B. Doing so creates an intermediary that pre-stores the matching records between Tables A & B, minimising the computational resources needed to calculate the otherwise complex join when querying the database. |

#### Normalisation

Splitting the data like this allows “**normalisation**” to be achieved, meaning the data is organised to eliminate data anomalies (DataCamp, 2024). Examples of these anomalies are:

* Duplicate data points across several tables e.g. employee details. Without normalisation, any update would have to be done in multiple tables
* Redundancy – The same duplication means redundant data is in the system, which requires more storage space (and cost) to store

Normalisation is important to a relational database as it helps achieve referential integrity i.e. data is consistent across tables. Organising the data into structured tables prevents these data anomalies from occurring by ensures that updates, deletions, or additions in one table are reflected consistently across related tables. Attempting such an operation will return an error, preventing orphaned records or inconsistencies in the data. This help support reliable data management and data quality.

### Non-Relational (NoSQL) Databases

A non-relational database (also called NoSQL) also stores data, but does not use a table, row, identifying and relational keys structure like a traditional relational database does (Pluralsight, 2020). There are instead various forms of non-relational databases that are tailored to specific requirements but the data not being stored in a relational manner is the commonality between them. These database types have gained the term “NoSQL” due to this, as the lack of a relational model removes the traditional use case for the SQL language (Microsoft Learn, 2024).

Examples of these are:

|  |  |  |  |
| --- | --- | --- | --- |
| **Types** | **Details** | **Use Cases** | **Example solutions** |
| **Document** | Contains data in a set of named string fields and object data values, referred to as a document (Microsoft Learn, 2024). Typically, this will take the form of a semi-structured format of data e.g. JSON, XML, YAML.  Multiple data entities can be combined into this document (e.g. customer details, order details etc), unlike a relational database which would separate these entities into tables. These documents also do not need to have the same structure.  Specific documents can be retrieved by a key, which uniquely identifies the document.  See Figure 3 for a visualisation of this. | * E-commerce data where items attributes differ product to product * Customer Data Management * Real-time analytics | * MongoDB * Azure Cosmos DB * Google Cloud Firestore |
| **Key-Value** | A simple table structure that stores data in 2 columns: the “key” and the “value”. The “key” column is used as the identifier for the data and the “value” columns stores the data point. Unlike a document store however, this is just a simple value.  The data is stored using a hashing function on the “key” value (hashing functions are one-way transformations, hence why this is not done on the “value” as well). Using the hash function allows for an efficient lookup.  These are optimised for simple lookup operations, where data need not be compared across multiple tables e.g. configuration options.  See Figure 4 for a visualisation of this. | * Caching session data on an application * Storing user preferences * Shopping cart data | * Couchbase * Amazon DynamoDB * Azure Table Storage |
| **Column-oriented** | These are like traditional Relational databases in that data is stored within rows & columns. Unlike relational databases however, data is stored within the columns of the table. Each record is still stored with a unique identifier, but the columns are grouped into “families” and these store logically related data points, which can be retrieved together. (Amazon AWS, 2024). This allows large amounts of data to be stored together and retrieved quickly.  These columns are not fixed. They can scale as needed and do not need to have pre-defined data types, removing the normalisation constraints of a typical relational database.  See Figure 5 for a visualisation of this. | * Logs * Read-heavy workloads e.g. analytics * Time-stamped data | * Amazon DynamoDB * Azure Cosmos DB * Google Cloud Datastore |
| **Graph** | Data is stored in the form of “nodes” and “edges”:   * “Nodes” represent entities e.g. a person, department * “Edges” represent the relationship between these nodes e.g. 1 person reports to another   (Microsoft Learn, 2024)  This storage allows the data to be represented graphically, hence the name.  See Figure 6 for a visualisation of this. | * Social Networks, for relationships between users * Supply chain management, via mapping out dependencies | * Azure Cosmos DB * ArangoDB |

*Example solutions sourced from* (Strauch, n/a)

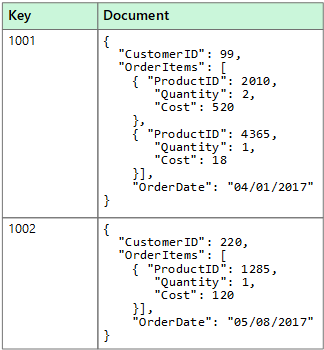


Figure 3 - Example of data storage in a Document store, a type of non-relational database (Microsoft Learn, 2024)

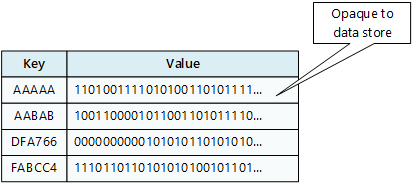


Figure 4 - Visual representation of a Key-Value store (Microsoft Learn, 2024)

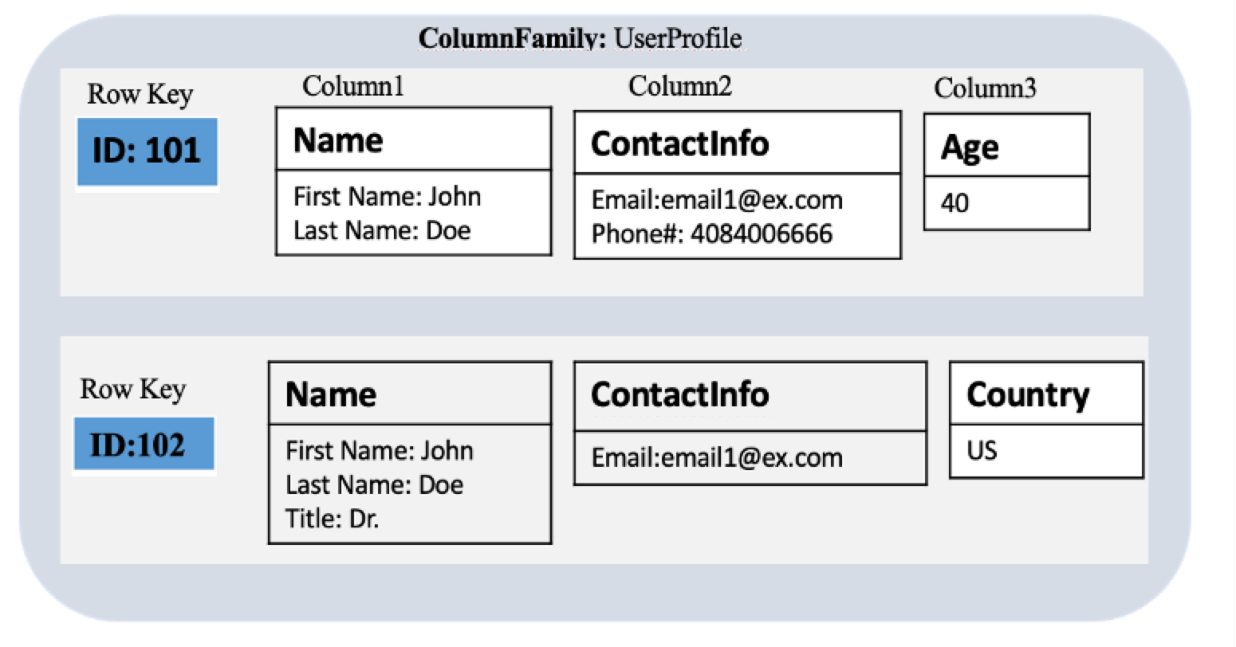


Figure 5 - Visual representation of a column database (Amazon AWS, 2024)

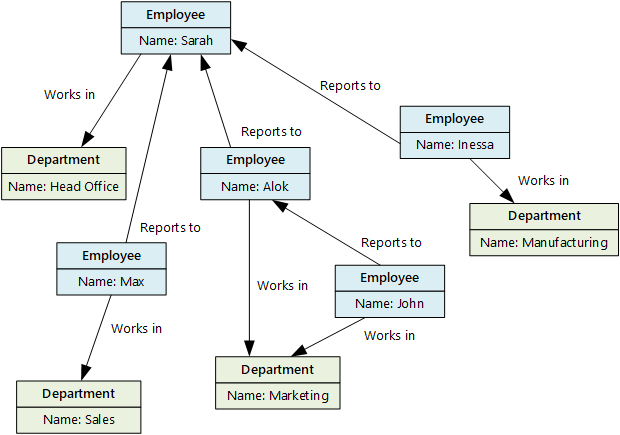


Figure 6 - Representation of a graphical database (Microsoft Learn, 2024)

#### Vs Relational Databases

There are some common differences between the various types of non-relational databases compared to relational ones. These are explored below:

|  |  |
| --- | --- |
| **Non Relational** | **Relational** |
| Non-Relational databases do not specifically have to confirm to a set schema the same way a relational model does.  This makes non-relational databases more reliable for data without fixed structures, or data that frequently changes | Data must be structured e.g. a table in a relational database must define columns with set datatypes. |
| Data is de-normalised, leading to potentially redundant/duplicate data that is unsuitable for analytics or reporting | Data is normalised, creating referential integrity in the data. |
| Data retrieval is simple and normally does not require joining other tables | Complex joins may be required |
| Excel at high availability and performance, where latency times are measured in milliseconds while performing millions of transactions per seconds |  |

(Microsoft Learn, 2022)

A summary takeaway I see form this is that non-relational databases are preferred for applications that require fast read times and do not require complex operations. For use cases that do require this however (e.g. analytics and reporting on key business performance indicators), a traditional relational database would be more suitable.

### Current Architecture

The below sections contain details of the existing architecture of our decisioning systems and how the data they generate is stored and used by front-end systems.

#### Overview

Below, I have summarised the architecture into a diagram:

A diagram of a data flow

Description automatically generated

Figure 7 - Current Architecture for calling External Data Providers

Our Decisioning Systems is currently split on a by-product basis i.e. for every product we offer, we use an individual instance of the decisioning system e.g. one for Loans, one for Cards etc. This is done so that individual decisioning logic that only applies to 1 product can be developed and maintained without the risk of introducing breaking changes into other products.

The decisioning system itself takes the form of a HTTP API. Front-end systems will send POST HTTP Requests to the system containing necessary data collected from the customer to make an informed credit decision. This data includes:

* Name
* Address
* Income
* Expenditure

The system uses this data to call Data Provider services offered by Credit Bureaus (e.g. Experian, TransUnion & Equifax) to obtain additional data. All this data is then used to execute a Credit Decisioning strategy to judge the credit risk of lending to an applicant. The system will then send a Response back to the front end with this decision, allowing the front-end to handle ether the onboarding steps (if the applicant is approved) or the decline process (if they are not).

#### Databases

Each instance also contains its own database; a MySQL relational database which is used for various purposes:

* Each HTTP Request/Response is logged for auditing
* Previous requests from the same applicant can be looked up e.g. if someone changes their details between applications, to get a preferable decision to them
* **Responses from external data providers are cached**, in case the same applicant re-enters the system. This allows the data to be reused, saving costs on calling these services (which are chargeable per call).

MySQL was not a specific tool choice by us. It was chosen by our 3rd Party partner & is deeply integrated into the Zoot system. As such, we are unlikely to be able to change it. MySQL however is a flexible solution, which provides some mitigation against this risk e.g. it can store and parse JSON documents, allowing it to effectively function as a Document-type non-relational database.

* Include backend DB and MI DB (K10)
* Explain Relational DB is an Architecture choice (MySQL has options for Horizontal Scaling)
* How Backend looks in front end (S2)?
* Past work on linking Datasets (Snowflake Copy of Live best bet)

#### Connecting Data to User Interfaces

The HTTP Response created by the decisioning systems is critical. By itself, this is just a data set. The front-end system interprets this data to direct where the applicant journey should go next.

Figure 8 below for example shows part of the response that contains the decision made by the decisioning system on an application. The “finalDecisionReason” and “decisionReasonCodes” are the critical fields here, as differing values can be used to control what the next steps for an application should be.

A screenshot of a computer code

Description automatically generated

Figure 8 - Snippet of the Decisioning Engines HTTP Response, which the front-end references to control the next screen shown to applicants

The “Progress” values seen here are an instruction for the front-end to show a “Congratulations” screen to the applicant, to indicate we can provisionally offer then a credit product (subject to other checks in the journey). An example of this screen is shown in the figure below (note that this is for an older build of the front-end system).

A screenshot of a credit card

Description automatically generated

Figure 9 - Front-end screen shown to applicants for a "Progress" result.

That only covers the parts the customer sees. In the background, the front-end system also creates several records within its own system to record relevant information on the application. This is to cover audit scenarios where a full view of how the applicant went through the application journey is needed & what decisions are made. Another use case is that this data can be referenced by the Operations teams when an application needs to be manually approved before a credit product can be offered.

Decision records like the one shown in Figure 8 above are saved in what the front-end calls “Check” records. The figure below shows an example:

A screenshot of a computer

Description automatically generated

Figure 10 - Check Record in the Front-End system, saving a record of the decision(s) received from the decisioning system

That is not the only record created however. Multiple check records are created based on data received in the HTTP Response from the decisioning system, covering other data points seen besides the main decision, as shown in the figure below:

A screenshot of a computer

Description automatically generated

Figure 11 - Multiple Check records created in the Front-End system

Figure 11 shows additional check records. These are based on other data blocks received in the response e.g. the “scorecards” data maps to the data seen in the Check record show in Figure 13, while one of the entries in the “externalInfo” data maps to the record shown in Figure 14. Data mapping logic within the front-end system parses the JSON response received to create as many records as required, allowing the front-end to maintain a complete record of the application, in a cleaner User Interface then a raw JSON can provide.

A screenshot of a computer code

Description automatically generated

Figure 12 - Additional blocks of data within the decisioning engine HTTP Response

A screenshot of a computer program

Description automatically generated

Figure 13 - Check record created from the Scorecard data from the decisioning engine's HTTP response

A screenshot of a computer

Description automatically generated

Figure 14 - Check record created from the External Info data from the decisioning engine's HTTP Response

## Strengths and Weaknesses

From reviewing the architecture and including reflections on recent work experience, I have identified the following strengths and weaknesses of the architecture:

|  |  |
| --- | --- |
| **Strengths** | |
|  | Using isolated instances of the decisioning system per product helps make the business logic implementation more maintainable.  For example, if changes are only needed in the business logic for evaluating Cards credit risk, this split means we would not risk creating breaking changes in the business logic for evaluating credit risk for a Vehicle Finance product. The split platforms mean the logic is isolated from one another. |
|  | The MySQL database solution can support both relational and non-relational database design paradigms, making it flexible for different use cases.  It also specifically has options for horizontal scaling, allowing the database to gain more processing power as demand increases. |
| **Weaknesses** | |
|  | HTTP Integrations with 3rd Parties are currently being built per monolithic product platform. These effectively means we are duplicating code per platform and as these are independent, they can differ.  This has been a risk that has materialised in recent history e.g. in 2024 we found a critical issue where the integration between an Experian API differed between 2 of our platforms. The way address data was being mapped into the request to the Experian API differed and resulted in the Experian API not consuming these properly. This negatively impacted the data we received back from that API and presented regulatory and compliance implications, requiring us to roll back the impacted build i.e. revert to an earlier production build. |
|  | Data stored in the Database cannot be re-used per platform, as they are all isolated from each other.  For example, the Response Cache use case stores responses from external data providers and attempts to re-use them before making new API calls. As each call is chargeable, this reduces costs. The isolation however means it is theoretically possible for an applicant to apply for each individual product we offer & we’d call the Data Provider service each time. Assuming the request data is the same (i.e. name, address, date of birth), then we’d get the same response from the data provider for each call.  This would be a barrier if we wanted to build a customer management process that checks a customer’s eligibility for each product (i.e. cross sell). |
|  | If there is Business Logic that is shared across platforms, we’d currently need to implement this on each individual platform.  This could be mitigated however by importing the necessary business logic between platforms (e.g. with version control). A design doing this however must be modular i.e. not have any dependencies on processes specific to a single platform. |

## Alternative – Microservice Architecture

The weaknesses identified in the above section highlight pain points in our current architecture. These weaknesses are symptoms of the monolithic approach we’ve taken so far. To resolve these, we can look at using a “microservice” approach.

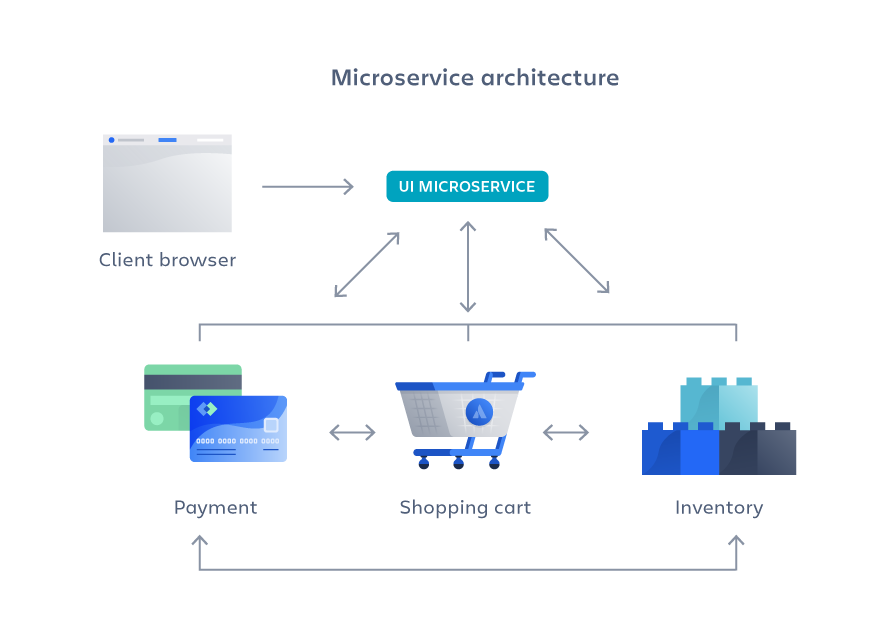


Figure 15 - Visual representation of a Microservice Architecture (Atlassian, 2024)

The Microservice architecture differs from monoliths in that instead of packaging all components into a single code base, multiple independent services are created and split into their own code bases, each performing a specific business logic. The resulting services then communicate with each other as needed e.g. via HTTP calls (Atlassian, 2024).

This approach allows for business logic to be decoupled from each other, bringing benefits over traditional monoliths:

* The independent services can be re-used across other solutions as needed e.g. if a later business project required a payment service, we would not need to build a new one
* The services are independent of each other, allowing changes to be made without the risk of breaking another service
* Splitting the services creates smaller code bases, making the solution easier to maintain and test, and therefore minimising time to market for changes
* The small code base also creates less opportunity to be “locked in” to specific technology choices, bringing more freedom in upgrading

(Atlassian, 2024)

### Proposed Architecture

Based on this idea of Microservices, I propose the to split off our 3rd party data provider integrations into Microservices, where our product platforms would instead call these services instead of using integrations built within the platform itself. Figure 16 shows this visualised:

A diagram of a database

Description automatically generated

Figure 16 - Proposed changes to Architecture, focusing on reusing API Integrations

Instead of integrating 3rd party APIs per product platform, we instead set up a new platform solely for calling these services. This will allow for **a consistent integration between these services to be used**, resolving the 1st weakness identified previously.

Using a single platform with a single Response Cache database also resolves the 2nd weakness, as this singular entry point to the integrations means we can share a database for these services**, allowing cached data to be re-used across platform**s.

Strictly speaking, a **true microservice architecture would use separate platforms for each 3rd party service we’d use**, however **doing so increases the costs** (as our 3rd party partner bills us per platform). Having a singular platform allows the above benefits while minimising this platform cost overhead, trading off not fully isolating each 3rd party service.

### How to Implement

Currently, our decisioning engine instances parse the JSON/XML responses from these 3rd party services into a Schema, allowing the data within to be used by the platform. This schema must pre-declare every object and variable within the structure. Anything not captured is dropped. An example of one such schema is shown below:

A screenshot of a computer program

Description automatically generated

Figure 17 - JSON Schema for an Experian API Product as seen within our low-code decisioning engine

Making this architecture change could have implications on how this is referenced. For example, variables/objects using this schema reference the exact path, with the below figure showing a variable called “MatchTo” being used in an IF-condition RegEx logic:

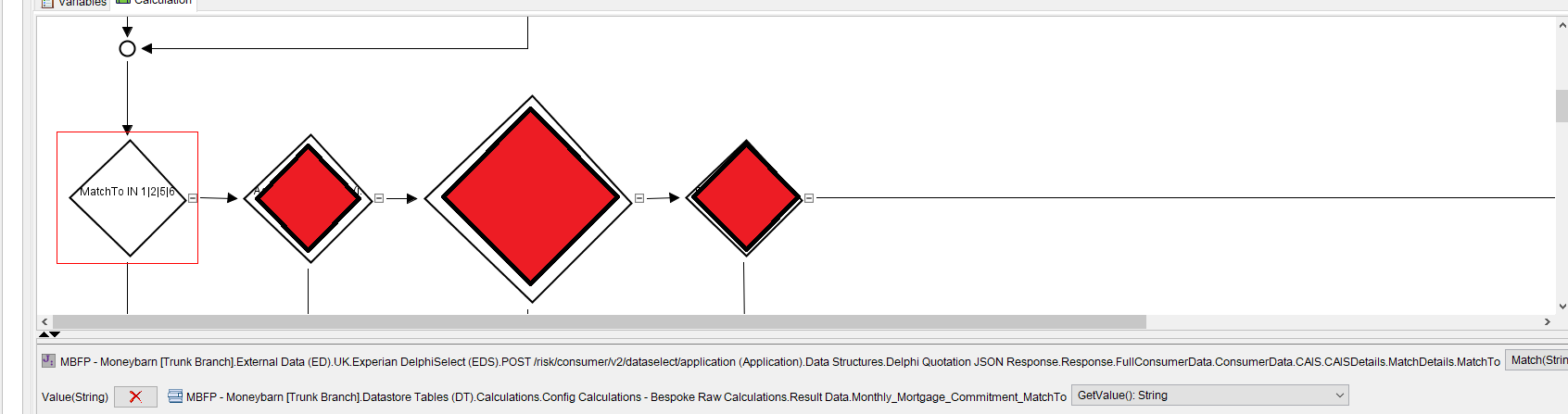


Figure 18 - Example of a RegEx match function being used on a variable within this parsed JSON schema

To avoid having to refactor any code, we’d have to pass the response from the 3rd party data provider exactly as is when designing our microservice payloads, so that it would still conform to this schema. This could become a limitation, as it limits our ability to create useful logging information from the HTTP call to the 3rd party service.

We could instead wrap the raw response from the data providers inside a “providerPayload” object, which opens the schema for additional metadata to be passed. A proposal for how this could look is below:

A screenshot of a computer

Description automatically generated

Figure 19 - Proposed JSON Schema for the Microservice Payloads

This structure will:

* Allow the payload from the data providers to be captured exactly as is, meaning if new data is added, we don’t need to update the schema for it to be captured. The exact format that can accomplish this however needs investigating. I have a concern that system limitations within our 3rd party decisioning engine tool mean that this may not be true if the raw payload from the data provider is saved as an object. It may be required to save the raw payload as a string instead to avoid this (preferably as a Base64 encoded value to avoid issues with special characters).
* Additional data from the microservice platform can be recorded e.g. call duration to the data provider in milliseconds, allowing us to record this for MI Purposes e.g. measuring the overhead using the microservice architecture adds compared to the current direct integration done per platform

## Conclusion

There are some painful weaknesses within our existing monolithic platforms around how they integrate with 3rd party services. The single code base nature of monolithic platforms means we duplicate the code required for these integrations to each platform. Each platform then makes changes to this code, resulting in differing implementations. We’ve already seen a critical bug in production because of this.

Moving the 3rd party data integrations to a separate platform in a microservice fashion would mitigate this and provide us with re-usable services. Ideally, we’d split off each data provider into its own microservice, however this would incur additional cost, as the core architecture of the decisioning engine software provided by our 3rd party partner means that a separate platform would have to be created for each service. They bill us per platform, so this approach would significantly increase costs. As a compromise, the proposed solution outlined in this document uses a shared platform to host these integrations, allowing the benefits to be realised with minimal cost.

There is one potential problem with the proposed solution, where the need for a predefined schema means we could lose data if the 3rd party provided add new variables or objects. How to mitigate this is an action to work with our 3rd party partner with. There is however a workaround identified here, where we could encode the raw payload from the providers in Base 64 and provide in the microservice response as a string, avoiding the risk at the cost of requiring decoding within the platforms.

# Additional Evidence

# Final Reflections

## Future Development

# Appendices

# Bibliography

Amazon AWS, 2024. *Types of NoSQL databases.* [Online]   
Available at: https://docs.aws.amazon.com/whitepapers/latest/choosing-an-aws-nosql-database/types-of-nosql-databases.html  
[Accessed 17 01 2024].

Atlassian, 2024. *Microservices vs. monolithic architecture.* [Online]   
Available at: https://www.atlassian.com/microservices/microservices-architecture/microservices-vs-monolith#:~:text=A%20monolithic%20architecture%20is%20a%20singular%2C%20large%20computing%20network%20with,of%20the%20service%2Dside%20interface.  
[Accessed 16 01 2024].

DataCamp, 2024. *Normalization in SQL (1NF - 5NF): A Beginner’s Guide.* [Online]   
Available at: https://www.datacamp.com/tutorial/normalization-in-sql  
[Accessed 20 01 2024].

Microsoft Learn, 2022. *SQL vs. NoSQL data.* [Online]   
Available at: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/relational-vs-nosql-data  
[Accessed 21 01 2024].

Microsoft Learn, 2024. *Non-relational data and NoSQL.* [Online]   
Available at: https://learn.microsoft.com/en-us/azure/architecture/data-guide/big-data/non-relational-data  
[Accessed 16 01 2024].

Pluralsight, 2020. *Relational vs Non-Relational Databases.* [Online]   
Available at: https://www.pluralsight.com/resources/blog/software-development/relational-vs-non-relational-databases#:~:text=There%20are%20five%20popular%20non,used%20for%20a%20single%20application.  
[Accessed 16 01 2024].

Strauch, C., n/a. *NoSQL Databases.* [Online]   
Available at: https://www.christof-strauch.de/nosqldbs.pdf  
[Accessed 17 01 2024].