Big Data Architecture

Learning objectives

After completing this topic you will:

- Understand integration between services to create an end-to-end data storage solution;
- Understand levels of enterprise architecture and the need for each;
- Understand data flow diagrams and their utility in data engineering, and
- Be able to design a complete data engineering storage solution with special focus on:
  - Creating a data flow diagram for a storage scenario solution, and
  - The creation of architecture diagrams for a scenario storage solution.

Introduction

Thus far in our learning, we've seen various ways in which we can design and architect big data projects. Here we think particularly of the Lambda or Kappa streaming architectures we've previously studied. We've also had a look at best practices for setting up data lakes, and how to properly architect them within large enterprise environments. In this train, our ambition is to pull together the knowledge we've gained up until this point and to use it in the design of a large data system or pipeline. Specifically, we'll consider:

- Thinking about data sources;
- Architecting storage layers and evaluating the relevant design choices upfront;
- Analysing the flow of data; and
- Stringing together services to create an enterprise-grade data engineering implementation.

Integration between services

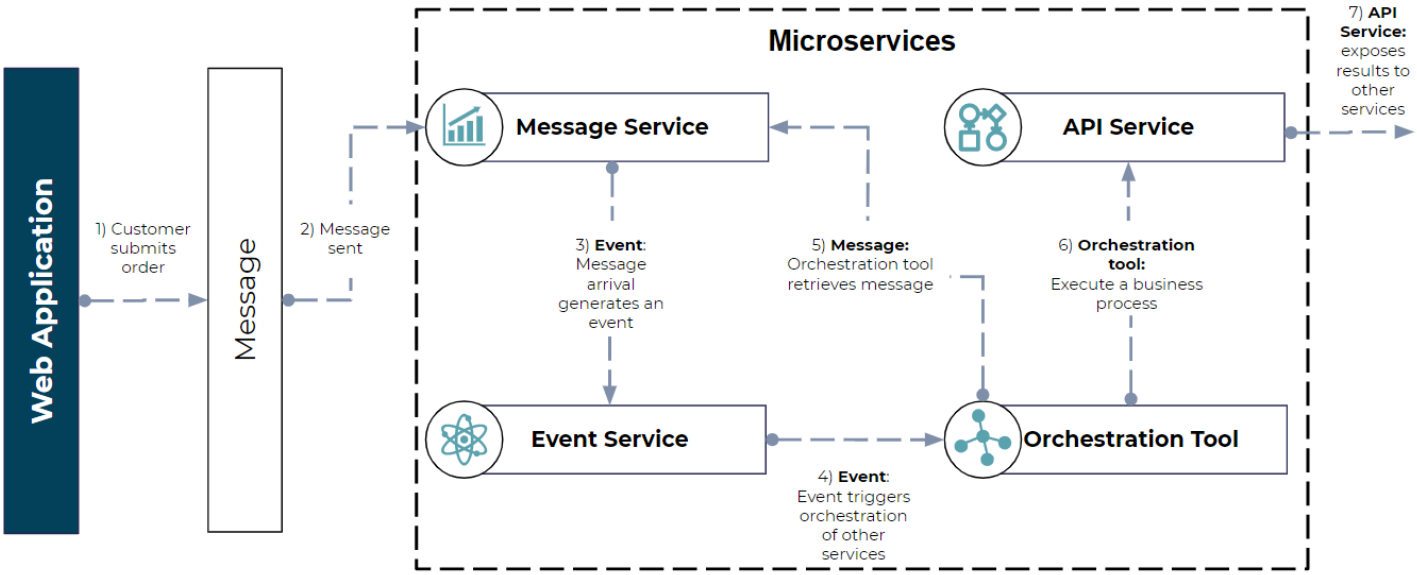

Figure 1: Integration services each have a unique role to play in tying together services across our applications. In the representation above, a messaging service functions to receive messages from an application, which can then be retrieved asynchronously. The messaging service triggers an event, that is then synchronously used to coordinate and kick-off an associated processing pipeline. The event in turn triggers the orchestration tool to execute the pipeline, which respectively goes back to the messaging service to retrieve the originally received message. Finally, the output of the processing step (controlled by the orchestration tool) is handed off to an API service, that integrates with various bespoke or enterprise applications.

An essential element of creating a complete storage solution is the integration between services within an environment. Fortunately, this is greatly simplified with the advent of cloud solutions. Cloud providers offer native support for integration between services - a term known as *integration Platform as a Service* (iPaaS). Cloud-based integration is enabled through a suite of cloud services that enable customers to develop, execute, and govern integration flows between disparate applications. The advantage of having this integration service is that it removes the requirement to have a middleware or hardware layer between each application or service deployed in the cloud. Since cloud integration breaks down data silos, it is an essential part of creating a conducive big data environment.

There are four core technologies required for cloud-based integration:

- **An API service**, that is responsible for publishing and managing APIs. Most modern applications subscribe to the *microservice architecture, and expose at least a part of their functionality through APIs. This makes software developed and running in the cloud available to other services in the cloud and on premise.
- **An orchestration tool**, this is responsible for creating and executing logic for business processes that rely on several applications. Instead of having to write code to specify the business logic in a specific language (e.g., C# or Java), having a workflow tool greatly simplifies the process, dealing with integration complexity as well as expanding the user base that can use the tool.
- **A messaging service**, providing a way for various applications to communicate in a loosely coupled way. This messaging solution should be asynchronous, allowing services to generate messages at their own pace, while other services consume these messages only as and when required.
- **An event service**, allowing direct synchronous communication between services. This technology allows stronger coupling between

services, with events triggering a string or set of events and down-stream services.

Good to know

Microservice architecture is a design which organises applications into a collection of loosely coupled services. This means that instead of having a massive application and codebase we utilise a series of smaller components, each with its own function, that interact with each other to build out a larger application.

Currently, Amazon Web Services (AWS) and Microsoft Azure occupy more than 50% of the cloud computing market share. For that reason, we undertake to focus on demonstrating their most prominent solutions for the above core integration technologies.

AWS

There are various services that allow application integration on AWS. In the diagram below an example of AWS integration services, and their integration patterns with one another, are depicted. Here we initially have an application the user can interact with. This interaction will create a data point, which is then captured in a Solace PubSub Event Broker - a proprietary tool which has to be integrated with the AWS cloud. Integration is performed by having a Virtual Private Cloud connection to Solace PubSub, which is then integrated using AWS API Gateway. API Gateway then serves as the integration layer that channels the data points into the AWS cloud and its various services.

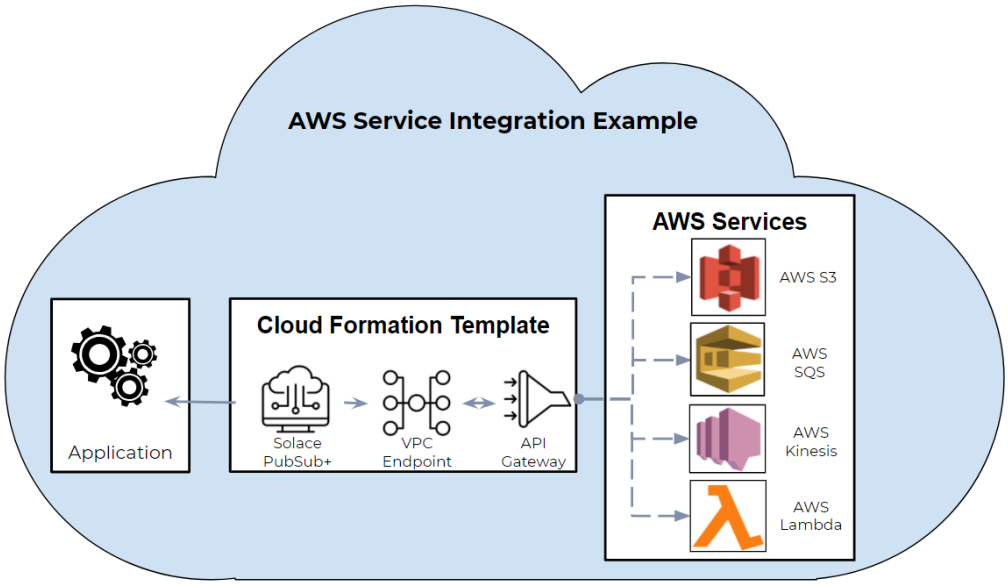

Figure 2: In the above diagram we illustrate how solutions within the AWS environment can interact with each other: a pubsub message queue can send information over a virtual private cloud, that interacts with an API gateway, followed by various interactions with other AWS solutions (e.g., Amazon S3, Amazon Lambda, and others). These interactions are facilitated by integration services such as the API Gateway or messaging queues.

The following list details some salient AWS core integration services:

- **API service:** To manage APIs, AWS provides two core services:Amazon API Gateway, and AWS AppSync. Amazon API Gateway provides a way to create, publish, manage, monitor and secure APIs. These can be bespoke APIs that we create within our applications, serverless APIs, or through web application based APIs. AWS AppSync is an API service that allows us to create an API to access, manipulate, or combine data from one or more sources.
- **Event service:** To build event-driven architecture, AWS providesAmazon EventBridge, a service that connects data from AWS services, applications, tools, or other SaaS services.
- **Messaging:** AWS has many services that can provide messaging capabilities, includingAmazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS) and Amazon MQ. Amazon SQS is a message queue that sends, stores, and receives messages between application components. SNS can send notifications through publish/subscribe, SMS, email, and mobile push notifications services. Amazon MQ is a message broker for Apache ActiveMQ and RabbitMQ. Additionally, AWS also provides implementations for most popular message brokers, e.g., Apache Kafka, and AWS Kinesis.
- **Orchestration:** Amazon AppFlow provides a way to automate the flow of data between SaaS applications and AWS services in a code-free manner. AWS Step Functions also allow us to create a serverless workflow for multiple AWS services. AWS also has an implementation of Apache Airflow which is fully managed; Amazon Managed Workflows for Apache Airflow. Otherwise, AWS Data Pipeline and AWS Glue allow for simple orchestration of data pipelines in a cloud environment.

Azure

Microsoft has always been a pioneer in ensuring that their services integrate well across platforms (or at least they have tried to do this). On of the ways in which they have done this in the past is through Microsoft Active Directory, a centralized domain management system that effectively takes charge of authenticating and authorizing users on a Windows domain network. Microsoft expanded their active directory to Azure Active Directory, which greatly embedded and enabled their Azure Integration Services. This allows extending domain management to the cloud, greatly simplifying the experience of authentication and authorization for users accessing a wide array of technologies, as well as authenticating the interaction of these services with one another.

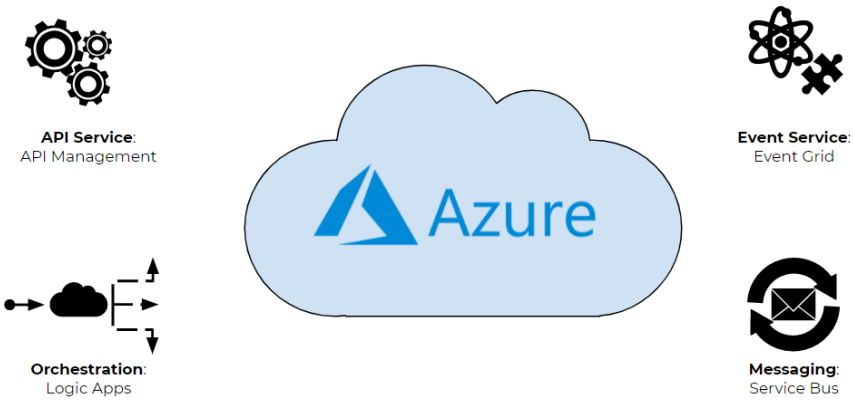

Figure 3: The four core cloud-based integration services on Azure.

At the core of cloud-based integration on Azure are four services, represented in the diagram above and described in the list below:

- **API service:** Azure provides API Management - a service that allows us to expose REST and SOAP APIs from any backend or serverless application, or from on-premise services. This means *any application* written in *any language* can be exposed in this way, or even other Azure services, e.g., Logic Apps. API Management creates proxies of the APIs, which can in turn be called, and points to the API endpoint of our application. As a result, anyone (on the client side) trying to access our API, just has to be able to make HTTP requests, and the internal routing functionality of API Management will ensure that our applications are accessible.

  More than just unifying APIs within our applications, API Management lets us track the usage and health of our APIs. Additional functionality also exists to enable us to set policies to control our APIs by:

  - Restricting the number of calls that a single user can make;
  - Specifying authentication;
  - Blocking certain IP ranges;
  - Turning data caching on or off;
  - Converting user provided SOAP requests to RESTful requests, and
  - Converting XML to JSON and vice-versa.

- **Orchestration:** Regarding orchestration, Azure offers Logic Apps - a service capable of configuring workflows that implement business processes. These could be system-to-system (connecting two services in the cloud) or user-to-system (connecting people to an application or software). These workflows are a series of actions, each which implement a logical step in the process. Actions can be conditional (if statements etc.), loops, calls to external software or services, or calls to Azure services. In fact, Logic Apps provides connectors to more than 200 external services, creating a simple way to interface with these applications. Connectors include: Office 365, Salesforce, SharePoint, SQL Server, Oracle, SAP, and link to standard network protocols such as SMTP and FTP. While Logic Apps is a no-code solution, it stores its configuration in JSON format, meaning that its workflows can be versioned, as well as be created from code.

- **Messaging:** Service Bus is an Azure service that allows asynchronous communication between programs, preventing one service blocking another. Service Bus provides enterprise messaging among many kinds of software; cloud applications, and on-premise and Azure services. Some of the prominent features are:

  - *Queue semantics* supporting all of the standard features of message broker services including message persistence, message ordering and conforming to 'exactly-once' delivery guarantees.
  - *Atomic transactions* allowing a queue to read or write as part of a larger operation that succeeds or fails as a unit.
  - *Poison message handling* that removes any troublesome messages that might cause problems, or put a receiver into an ongoing loop.

  - *High availability* including geo-redundancy and out of the box disaster recovery.

    Service Bus allows the creation of multiple topics that receivers can subscribe to as a means to receive messages, with senders being able to send messages to one or more of these topics. Receivers can in turn also filter topics to only receive a subset of messages from a specific topic.

- **Event service:** Event Grid is an Azure service used when, instead of polling a queue or waiting for an API call, we can register to an event handler to listen for an event to take place on an event source we are interested in. The event handler is invoked when the specified event occurs. As with all other services, Event Grid can integrate with many Azure services that produce events, e.g., a file being created in Azure Blob storage. To enable Event Grid-based interaction, the receiver subscribes to a standard topic for a service, which is typically for an Azure service. We can also create custom topics for other cloud or on-premise services. Event Grid is massively scalable, being able to scale to up to 10'000'000 events per second in a single Azure region. This, however, may come at the cost of losing some of the order of the messages. Event Grid also provides near real-time performance, having a service-level agreement (SLA) to delivery 99% of messages within less than a second.

Enterprise architecture (Conceptual, Logical, or Physical)

Infrastructure can be modelled at various levels of detail. These levels have been solidified as being*Conceptual*, *Logical* and *Physical*, with conceptual models providing the least detail, and physical models providing the most. These levels of modelling have been designed to ensure that the correct focus on detail is employed for a specific task or at a specific moment in the architectural design process. When these levels are mixed or confused, low quality systems are designed and models fail to fully capture or communicate the ideas presented in them.

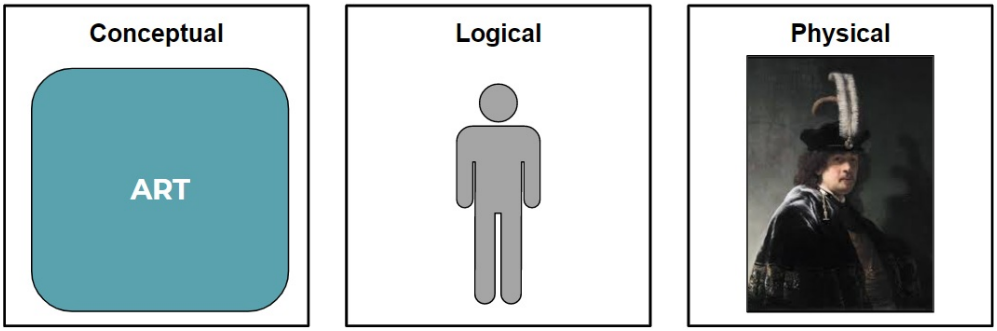

Figure 4: Higher levels of architecture modelling are there to convey concepts (such as 'art'), whereas lower levels provide extensive detail (such as an instance of a painting).

Conceptual architecture

The Conceptual modelling level can be considered as the bird's-eye view of a system, providing detail from which we can understand general system functioning. This model should highlight the most important concepts, capturing structural components of the system, relationships between concepts, and the order of different parts of the system. It is crucial that this level of architecture modelling *focuses on the relationships between components and not how they function*, and should provide a clear and concise overview of the architecture and project, only exposing the most fundamental concepts.

This view is most useful in communicating work done by an architectural or data engineering team to a non-technical audience. Having this communication in place allows for alignment with the business strategy, and establishment of a clear baseline of understanding.

Let's use a running example to illustrate the various levels of architecture that we're describing. Imagine we're designing an enterprise-grade data lake for a large engineering company. This company consumes data from many sensors that are spread around several of its processing plants. These sensors are used for understanding and optimising their processing capability through the use of data science-based techniques. We are tasked with setting up an data acquisition application, central data lake, and analytical capability for data scientists to process the data on.

***Conceptual architecture example**: Focus on elements such as users of the system, organisations, technologies and how they interact.*

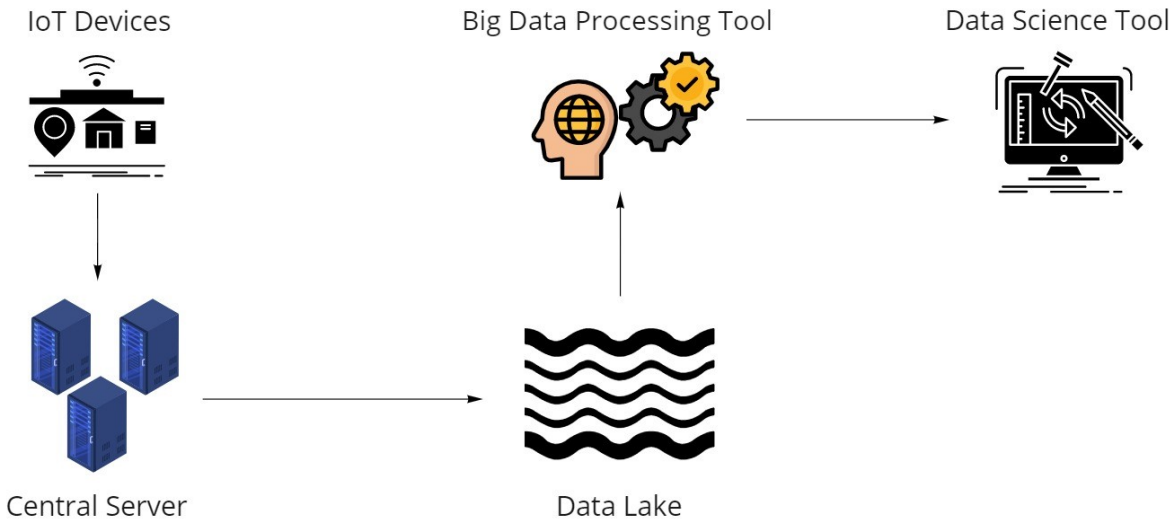

Figure 5: Using the above example, the IoT devices will submit data to a central server, we will then access and ingest the data into a central data lake. Once we have data in a data lake we will process and clean the data using a big data processing tool. Finally we will add a data science processing tool.

Having this clear baseline allows for the next step; creating a logical architectural diagram.

Logical architecture

Logical architectural models describe how a solution works with regards to function or logical information. Sitting neatly between conceptual and physical levels, logical models allow for the modelling of architecture at both high and low levels of detail depending on the specific use of the diagram. Logical models can be a very powerful tool of persuasion during the architecting process.

When designing a solution, we typically start off by drawing up a conceptual diagram. As we move to convince investment committees to invest funds or allocate resources to the specific project, we can draw up a logical diagram that includes more information and concepts familiar to the stakeholders we are engaging. This may include jargon, e.g., *Risk*, *Competition*, or *Revenue*, or could see the application of logical filters or placeholders for elements such as data or components of the system. Including these details can greatly increase the probability of getting stakeholder buy-in. Logical architecture allows for stakeholders and decision makers to see a clear link between strategy, capabilities and business benefits.

***Logical architecture example**: Here there is more focus on specific inputs, outputs, functions and decisions. This provides us a deeper view into the physical details to ensure we understand and can address all key technical issues.*

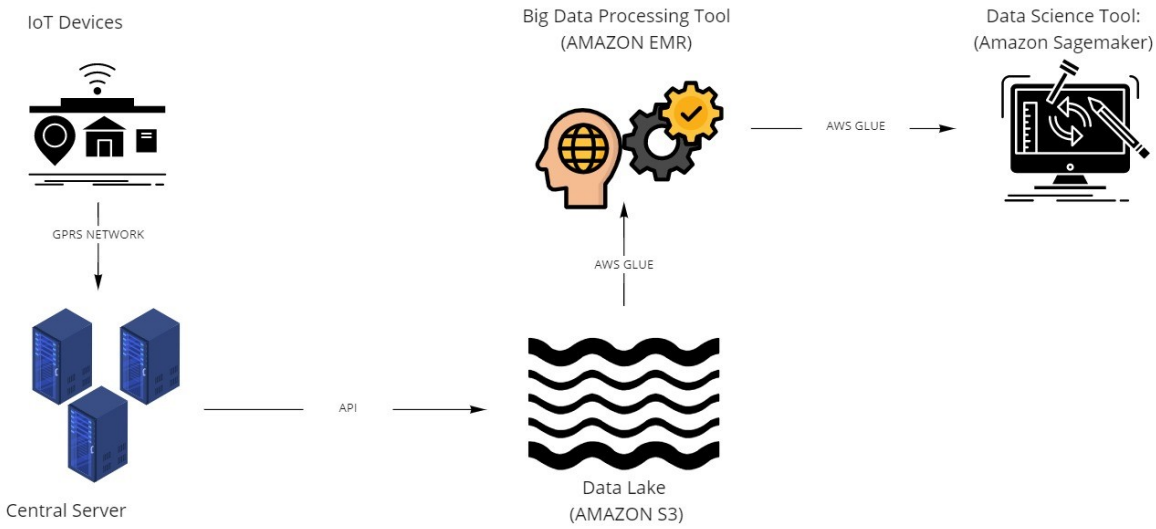

Figure 6: Continuing with the above example, at a logical architectural level we start filling in the detail. Here we'll look at the interface we have with the central server that stores the data from the IoT devices, e.g., is it an API, do we have to set up an FTP site to receive data, or can we stream it in? We will look into the needed orchestration methods and tools to use for moving data around, and the steps where we will require translation or cleansing of the data.

Finally, to see how we actually go about implementing these changes to the business, we look at modelling at a physical architecture level.

Physical architecture

Physical architecture is the lowest level of modelling abstraction, and is specifically detail-orientated; containing information around tools to be used, data representations, and other technical details that provide sufficient context to enable practical system implementation. Here, a comprehensive model ensures that all details are accounted for - with no ambiguities in the system design that may cause implementation uncertainty or unaccounted time delays/expenses being incurred.

Having a complete physical architecture of the entire system, which address data, infrastructure, the application, and the business, can serve as a roadmap for stakeholders. This roadmap can be used to visualise dependencies, find risks, track spending, and budget for any changes in the system design.

***Physical architecture example**: The most detailed view gives enough context for actual implementation. This diagram should show how actual objects move through the physical infrastructure.*

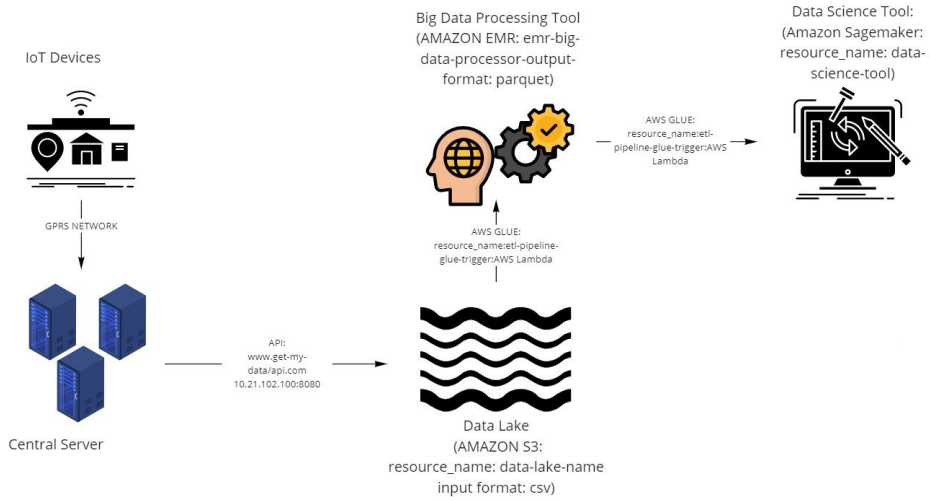

Figure 7: Completing the above example, physical architecture modelling provides enough detail to allow for system implementation. We need to include detail of the central IoT server, such as IP addresses and protocols to use to access the data. We also need to have detail on the specific file formats we expect the input data to arrive in, and the expected output formats. We should also have a sense of how we use integration services to combine a suite of cloud services to build our application.

Data flow diagrams

Data Flow diagrams are graphical representations of how data or information will flow within business information systems or software-system processes. They serve as an alternative to the enterprise architecture diagrams described above. A data flow diagram provides an understanding of the inputs and outputs of each entity within a system, in addition to detailing the functions performed by these entities. The aim of a data flow diagram is to make its described system easy to understand by end users, which can be specialised or non-technical audiences. As with architectural diagrams, data flow diagrams can range in the detail they provide - from representing high-level abstractions of a system, to being able to serve as a template for implementation. In a *level 0 data flow diagram*, less thought is given to the specific services at each point in a process, and even less to the specifics of how these services will interact with each other. In contrast, in a *level 2* and above data flow diagram,

enough detail is given to implement a solution. This might range from specific services that will be involved in implementing the solution, to giving detail on which authentication methods will be used.

At a high level, data flow diagrams consist of processes, data-flows, external-entities, and data stores:

- **Processes** change data, taking in a data input and producing an output. This can be business logic, some aggregations, or mathematical operations.
- **Data-flows** describe the path data takes between processes, external-entities, and data stores (think of these as the arrows between a set of blocks).
- **External-entities** are exactly that - an external system that sends or receives data. This can be a business system, person, or computer system, and will typically be regions where data enters or exits the system being draw.
- **Data stores** are repositories that hold data, and include data lakes, databases, or other methods for storing data.

As mentioned above, data flow diagrams can have increasing levels of detail, from *level 0* (referred to as *context diagrams*) to *level 3* and beyond (containing more and more detail of the underlying processes). However, it is uncommon to go to levels 3 and beyond as these levels will introduce unnecessary complexity that can be difficult to represent visually.

Let's illustrate these levels of flow diagram using an example. Imagine that we are in a team building out a smart budgeting application. This application connects to a banking application gateway, allowing us to retrieve user information and transactions for a range of banks. Transaction history for a specific user gives us the basic information that a user will need to build a budget from. The application should then allow the user to classify and sort these transactions into various categories. To assist the user, we build a machine learning model, as well as rule-based models which pre-classify transactions into certain budgeting categories. Being able to connect to multiple financial institutions, we also deduplicate records between these systems (e.g., when we transfer money between accounts, we will not want to double count this transaction information). Finally, the transaction history and categorised transactions will be displayed in the form of a mobile application.

Having described the system, let's realise it with flow diagrams of varying levels:

Level 0 data flow diagram

This level is called a *context diagram* - providing the most basic overview of a system. The aim here is to show the system as a single process, with connections to external systems. A question we should ask ourself here is: "What is our system trying to do?"

An example for our budgeting app would be:

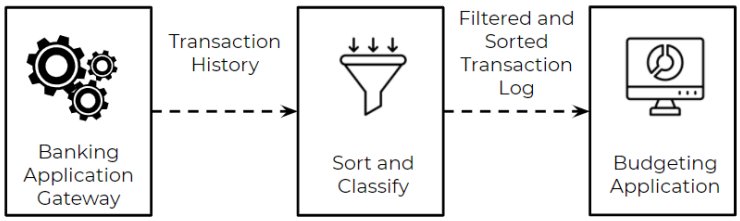

Figure 5: A context, or level 0, diagram of our described system.

In the above diagram, we simply try to get a sense of the flow of information, starting with a specific external-entity, going through a process (which in this case is the whole application), and ending in an output as another external-entity.

Level 1 data flow diagram

Here we start to get to the detail, breaking out some of the functions that are required by the system. This means breaking down the main process into some sub-processes.

Lets continue with the above example:

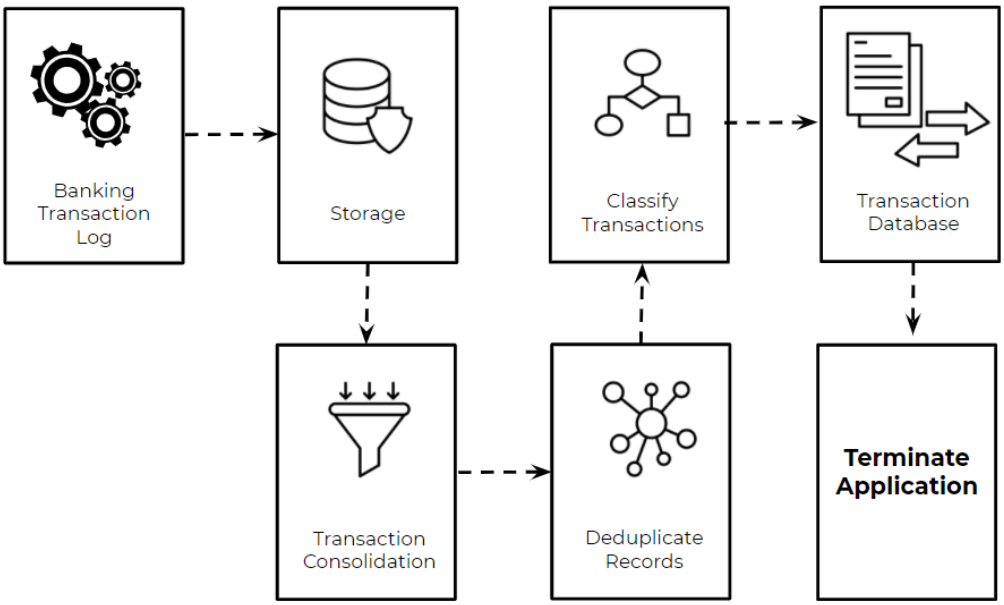

Figure 6: A level 1 flow diagram of our described system.

In the above diagram, we start to look into storage, where we have some logic or intelligence layers built in, and begin developing a sense of the flow within the application. We specifically show where data will be stored in the application, spell out three processes that will be involved in

transforming the data, and end with the data in a transactional database which will be used to display records on a front-end application. At this level there is enough detail to start involving the development team in the system's design decisions in order to improve its expected operation.

Level 2 data flow diagram

One more layer of complexity. Here we break the processes down even further, with each sub-process requiring more text and explanation than the previous level 1 diagram.

Let's carry on:

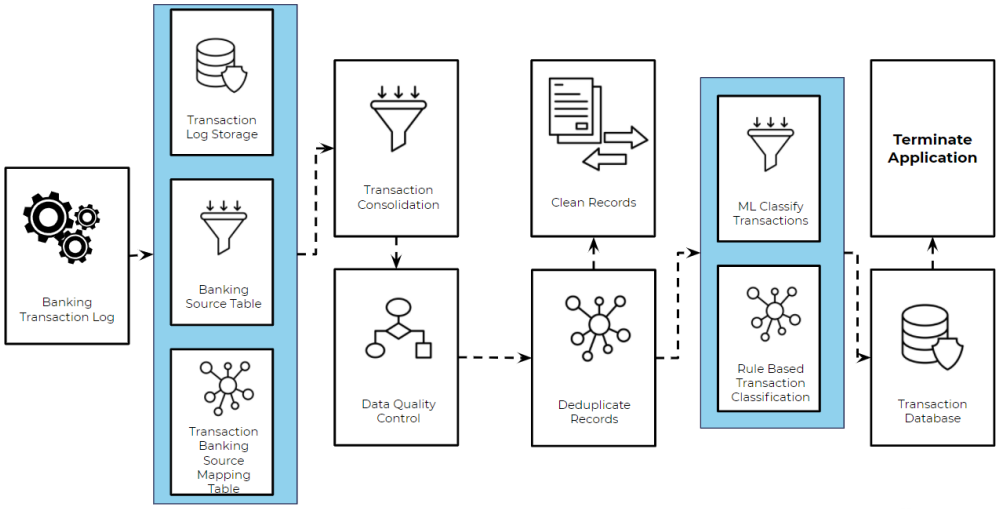

Figure 7: Level 2 flow diagram.

At this point we should be close to a diagram that can guide developers to build the application. We now not only provide storage as a category, but start breaking down the key data sources and tables that will be required as input data. We also create intermediary datasets, which can be used to ensure idempotency and validate our logic. Finally, we also include the models that we will be using within the application.

As mentioned above, levels 3 and beyond continue by adding additional details to our diagram, specifying further details around our data. If we want to be very fancy, we could use software that allows us to collapse the diagrams to level 0, while also allowing us to drill down to lower levels (an example tools here is Whimsical).

If done correctly, developers can use these data flows to directly write pseudocode, which inturn will simplify the system's implementation.

Conclusion

We had a deeper look into how we can integrate services withing the cloud and what it takes to design a large data engineering system or application. Designing systems and services are no longer only the responsibility of systems architects, but as data engineers, we'll also be responsible to think about best practices for designing data-intensive systems. This is especially true in using design principles to effectively illustrate the concepts and components that will form part of our systems. Using differing levels of complexity, we can effectively communicate to business stakeholders, as well as the technical development teams that will implement our solutions. There will be various ways in which we can model these systems, and it is important to find the technique most suitable to our specific use case.

Resources

- How to Use Architecture Levels Effectively
- Tools for drawing and deeper explanations of Data Flow Diagrams
- A deeper dive into various notations used to draw Data Flow Diagrams
- Everything you possibly want to know about Data Flow Diagrams