



WORKSHOP

Data Modeling and Network Source of Truth

AutoCon 4 - November 2025



About me: Damien Garros



Co-Founder and CEO of OpsMill

Focused on infrastructure as code, automation & observability for 12+ years

Previously led Technical Architecture at Network to Code



damiengarros



@dgarros

OpsMill team



Benoit Kohler



Wim Van Deun



Alex Gittings



Mikhail Yohman

Agenda 1/2

Introduction	9:00am	10min
Part 1 Data Management <ul style="list-style-type: none">• Introduction to data management• Key schema concepts• Different type of databases• Lab 1	9:10am	90min
BREAK <ul style="list-style-type: none">• Advanced schema concepts• Beyond the schema	10:45am	15min

Agenda 2/2

Part 2 Network infrastructure modeling	11:30am	90min
<ul style="list-style-type: none">• Data in layers• Business & operational context• Data federation / aggregation• Design for idempotency• Lab 2		
End	1:00pm	

Targets and goals for this workshop

This workshop is targeted to automation builders with some experience building scripts or applications.

Goals of this workshop:

- Introduce the fundamental technologies to store, organize and consume data (schema and database) and present the differences between them
- Present the best practices and challenges to model a network infrastructure in a Source of Truth

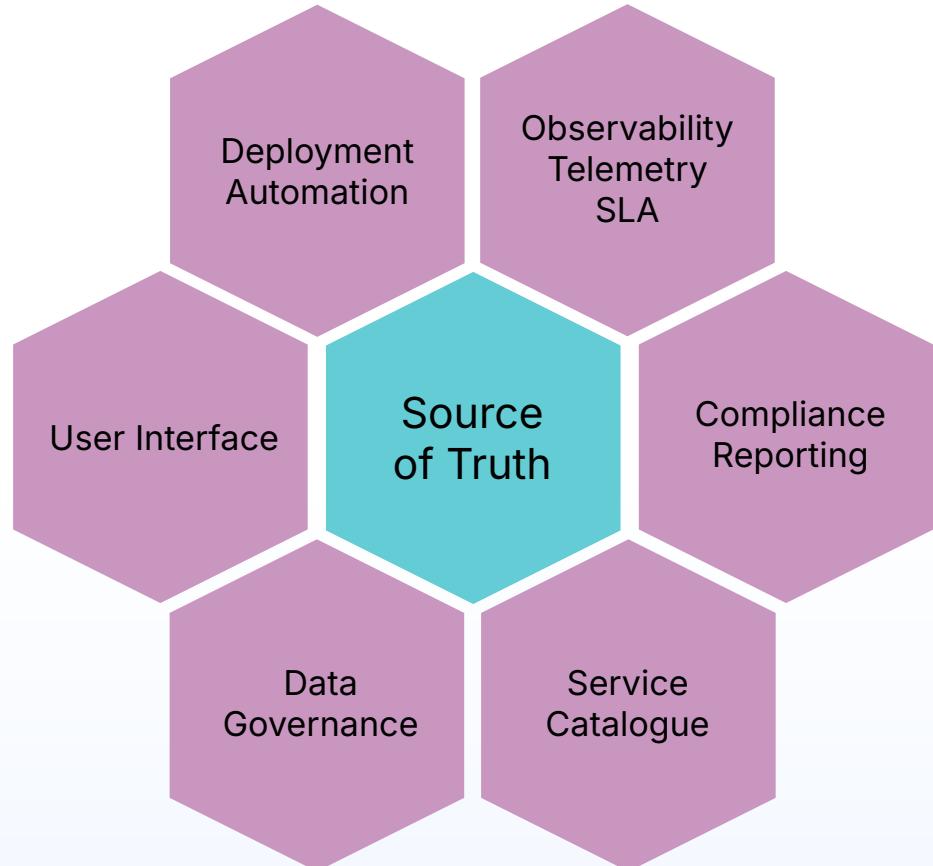
What about you?

- Familiar with SQL?
- Already used GraphQL?
- Familiar with Neo4j?
- Already tried Infrahub?
- Love XML too?

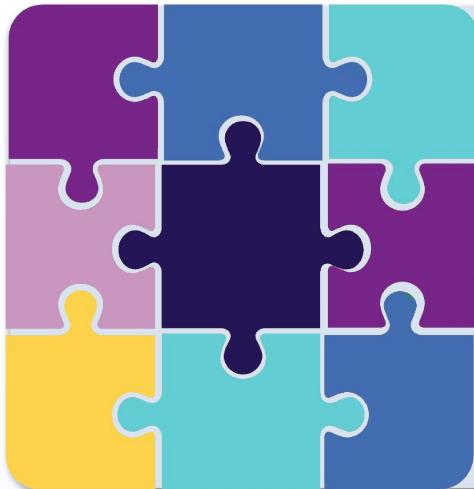


Introduction

Automation starts with data



Source of truth



Inventory

Cabling / Topology

DCIM

Circuit

IPAM

Roles/Statuses

Configuration
Templates

Routing
Information

Services

What do you need to rebuild
if the network was completely lost?

Typical network source of truth

Git

**In-house
solution**

Infrahub

**NetBox /
Nautobot**

Key pillars to successful automation



Flexible data model

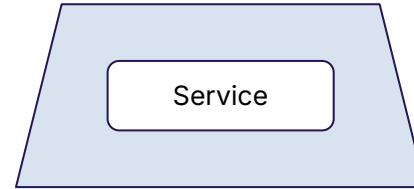


Versioning



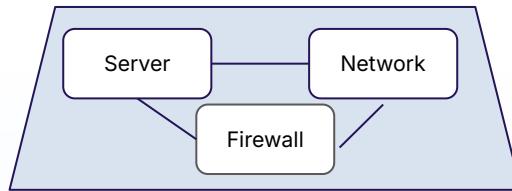
CI pipeline

Multiple layers of infrastructure data



Service / Intent Layer

Definition of what services the infrastructure needs to deliver



Technical Layer

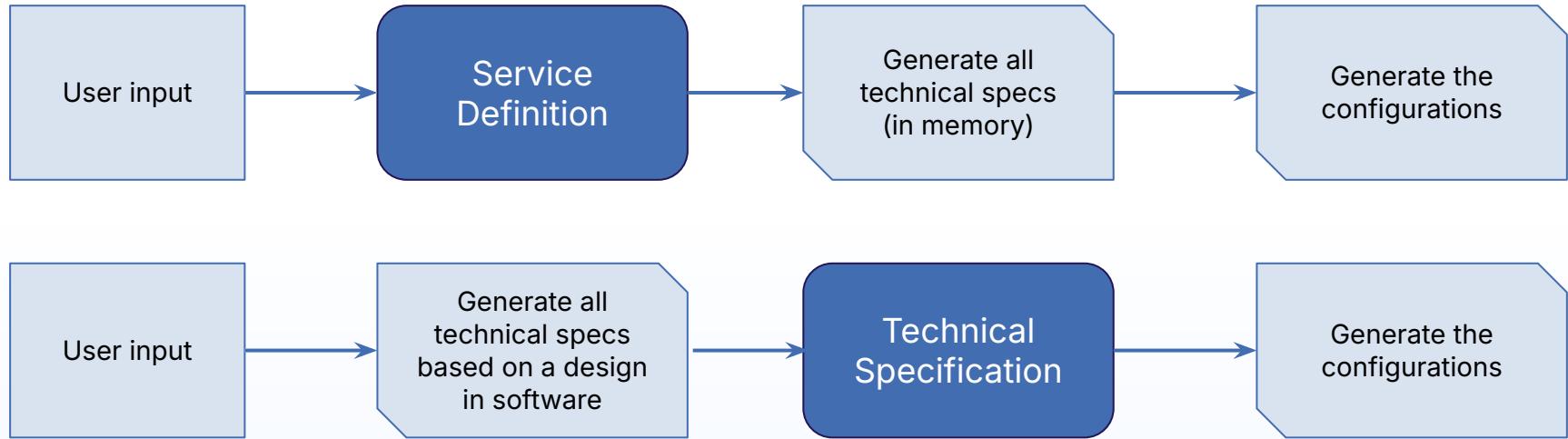
Global representation of the infrastructure elements interconnected



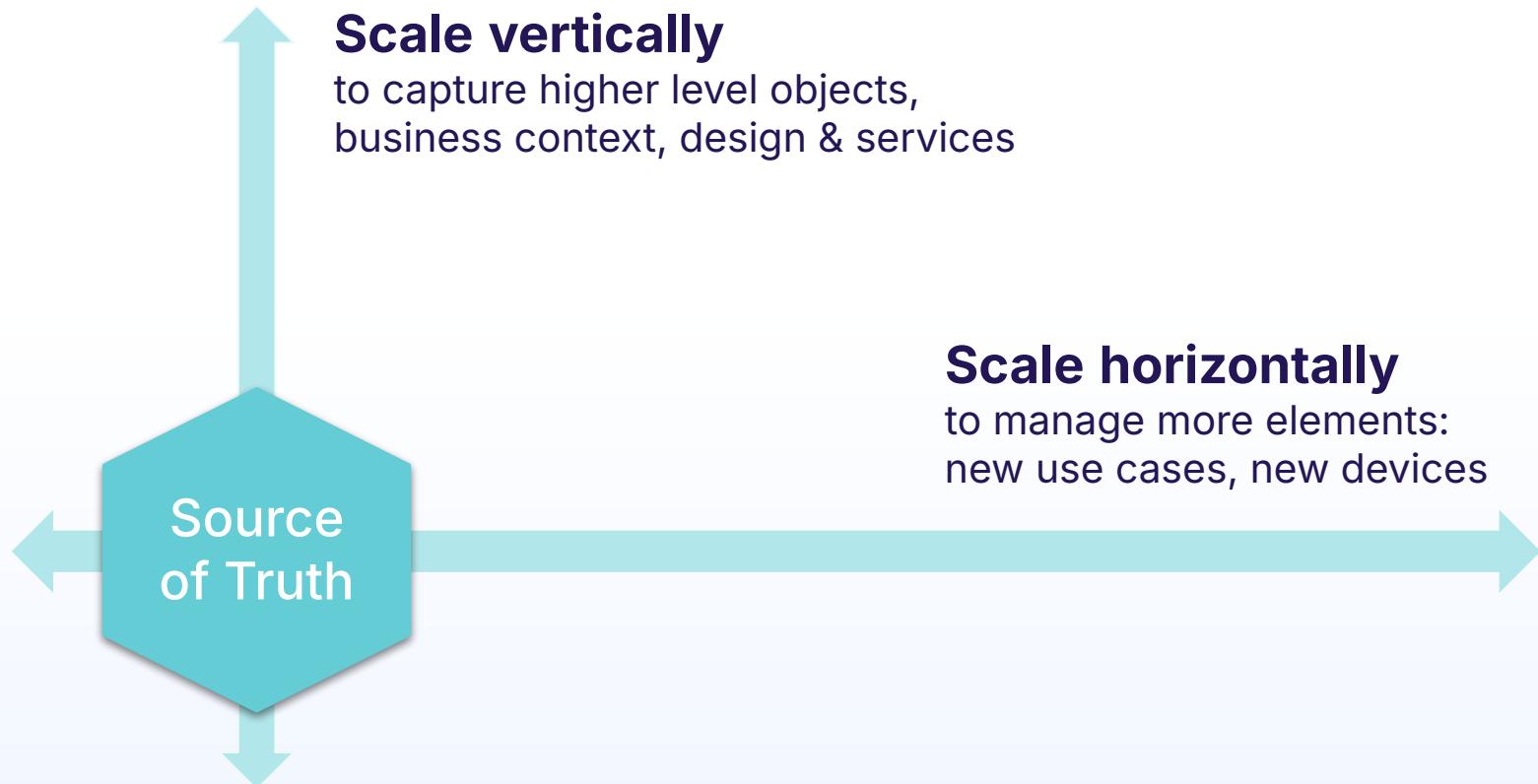
Component Layer

Each element is managed individually

Capture both service and technical information



Flexible data models



Part 1: Data Management

Schema definition

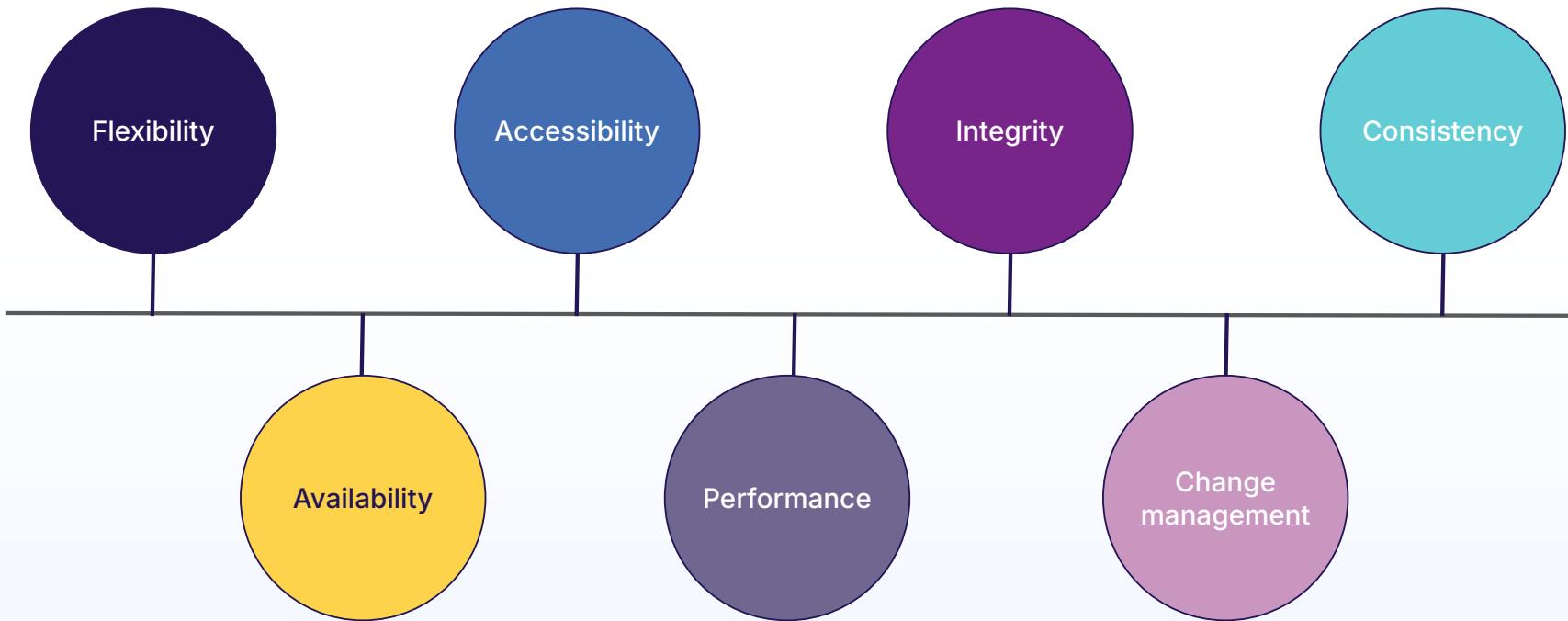
A schema defines the **structure, format, and constraints** of data, specifying **how data is organized and interpreted** in databases or data models.

It is important because it **ensures data consistency and integrity**, and **facilitates communication between different systems** by providing a shared understanding of the data's structure.

It's not about which one is the best,
it's about
**which one provides
the best trade-off for your use case.**



Data management trade-offs

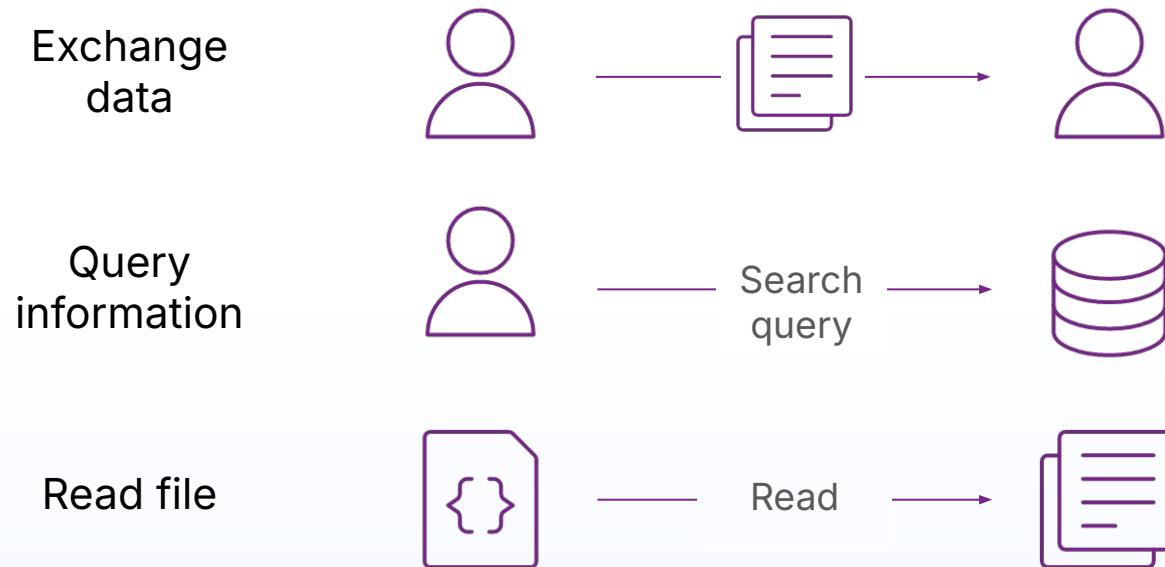


Schema definition

```
----  
- location: USA  
  site_name: TechHub  
  partial_address: 123 Main St  
- country: Germany  
  site: BlueOcean  
  address: 456 Ocean Dr  
- country: Japan  
  address: 789 Sunset Blvd
```

Does it have a schema?

Schema definition



Whether it's intentional or not, there is always a schema

Schema purpose



Data storage



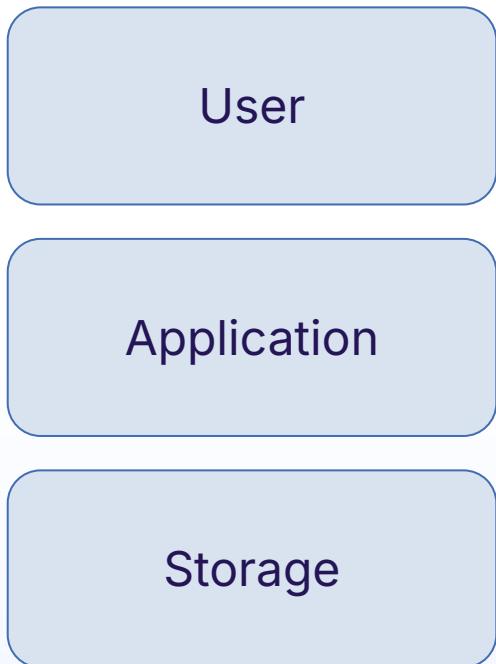
Documentation



Data integrity validation



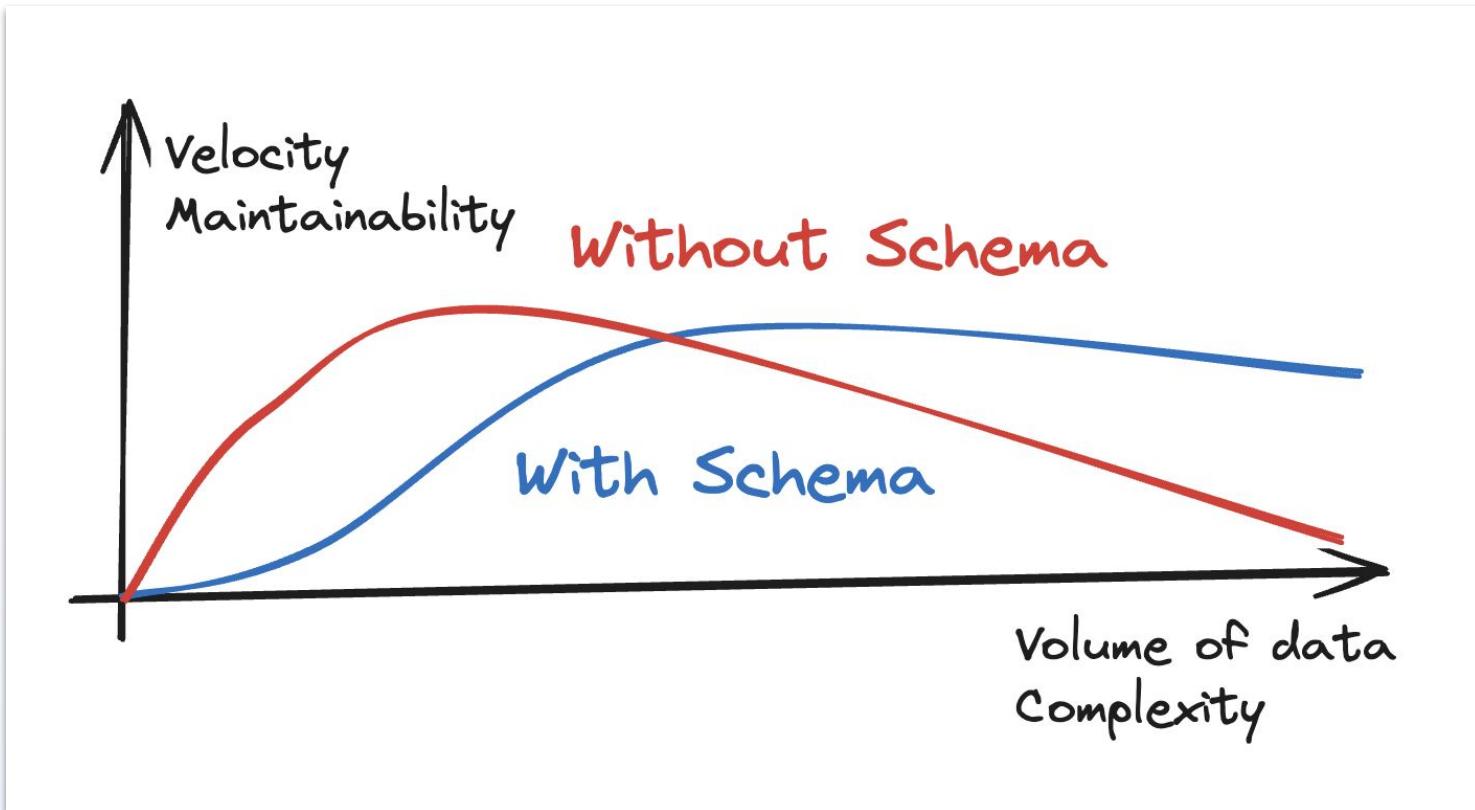
Different implementations



A schema can be defined or implemented at various levels in an application stack.

Each level has its own set of advantages and trade offs.

Pros and cons of having a schema



Schema, data format, or query?

SQL

JSON
Schema

YANG

Pydantic

Excel

XML

PromQL

YAML

GraphQL

JSON

Protobuf

TOML

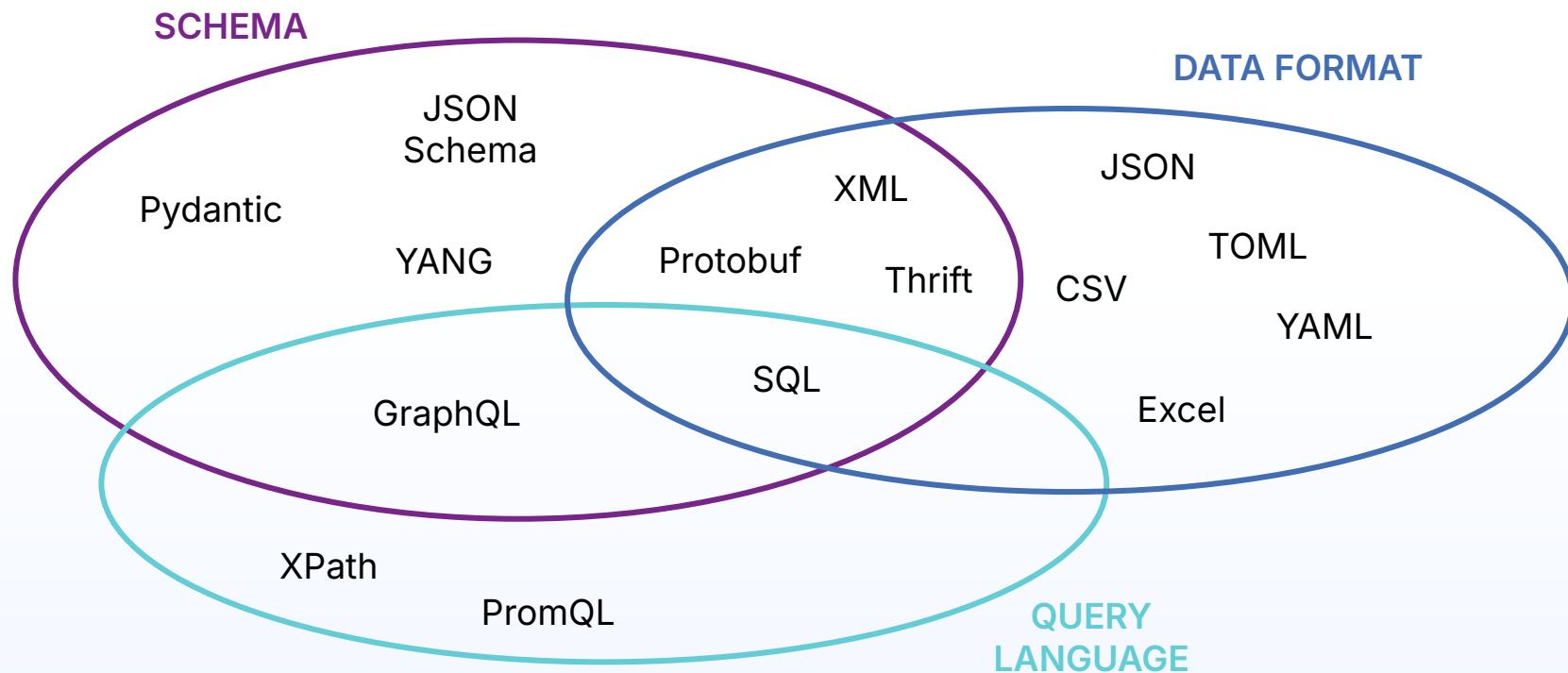
XPath

Thrift

Avro

JMESPATH

Schema, data format, or query?



Schema: Key Concepts

Schema principles

- **Structure:** A schema defines how data is organized. It specifies what kind of data can go where.
- **Relationships:** Schemas describe how different pieces of data are connected, such as linking customers to their orders.
- **Constraints:** It sets rules for the data, such as what values are allowed or required. This helps ensure data accuracy.

Schema principles

A schema is composed of entities or nodes.

Each node is usually composed of some attributes.

Attributes can have various types depending on what's supported:

- Integer or number
- Text or string
- Date
- JSON blob

DEVICE	
str	name
str	manufacturer
enum	status

PRODUCT	
int	id
str	name
date	created_at
date	updated_ad

Schema principles

Relationships define how nodes are connected together.

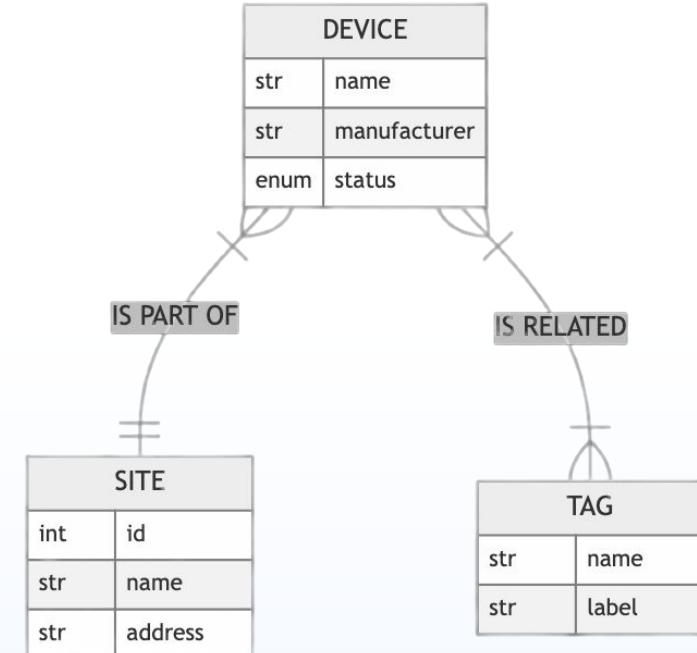
DEVICE is connected to SITE
DEVICE is connected to TAG

Relationship types aka cardinality

One to One +—+

One to Many +—< (DEVICE - SITE)

Many to Many <—< (DEVICE - TAG)



Relationships are often called edges.

Constraints and validation rules

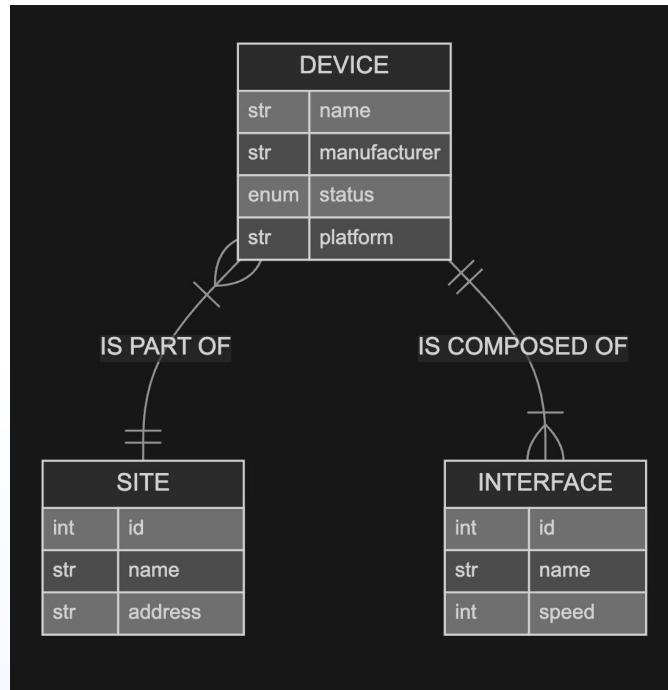
Constraints sets rules for acceptable values for each attribute or relationship.

Constraints can include:

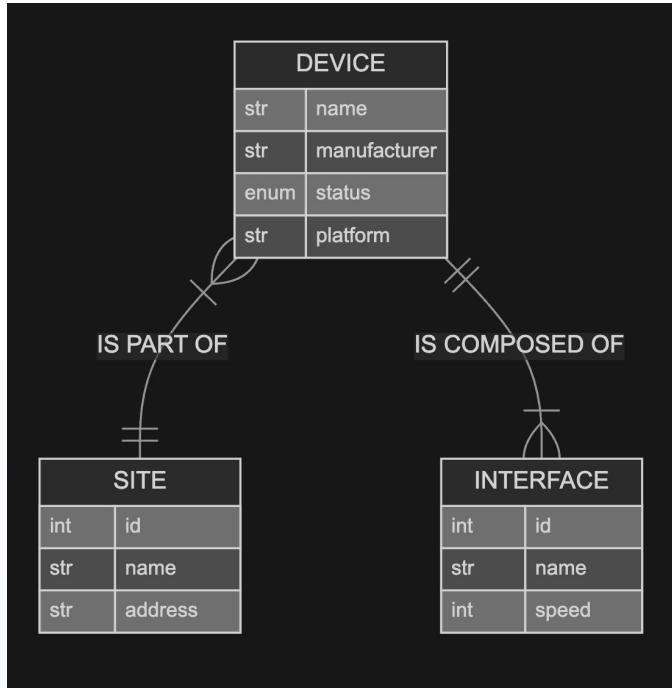
- Required fields
- Unique values
- Default values
- Format
- Maximum and minimum values
- Length restrictions
- Maximum and minimum number of related nodes

Example constraints

- An interface must be related to a device
- The speed of an interface must be an integer
- The address of a site must include a zip code
- The status of a device must be one of [active, maintenance, offline]
- 2 interfaces with the same name can't be associated with the same device
- 2 devices can't have the same name

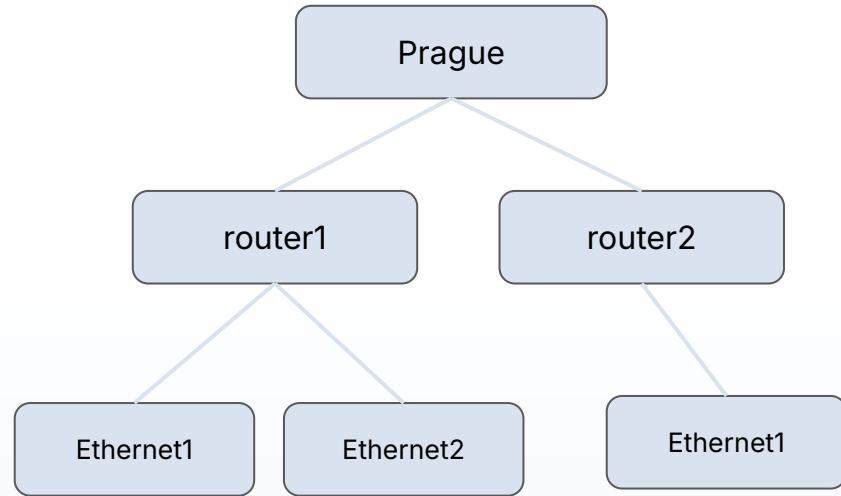


Schema vs instance



SCHEMA

Site
Device
Interface



INSTANCE

Different types of databases



**THE PERFECT
DATABASE**

"Let's see who you really are!"



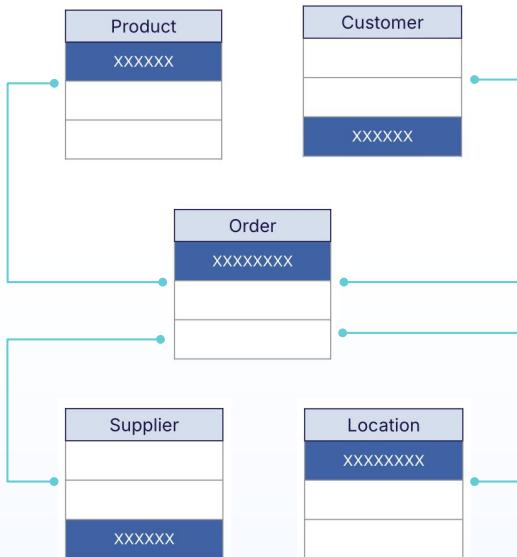
**JUST A BUNCH
OF TRADE OFF**

Different types of databases

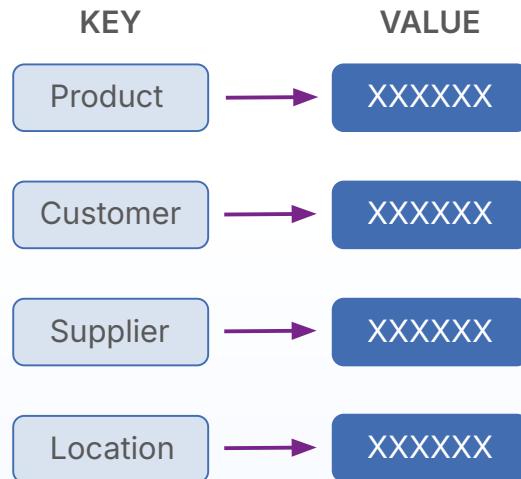
	Schema	Query	Other			
Relational DBMS	Mandatory SQL	Powerful SQL	-	 PostgreSQL	 ORACLE	 MySQL
Key-Value Store	No	-	Optimized for speed	 redis	 Memcached	
Documents Store	Optional JSON schema	Simple	Optimized for scale	 mongoDB.	 elasticsearch	
Graph DBMS	Optional	Powerful (GQL)	-	 neo4j	 ArangoDB	
Time Series DBMS	No	Powerful	Domain-specific	 influxdb	 Prometheus	

Relational vs KV vs graph

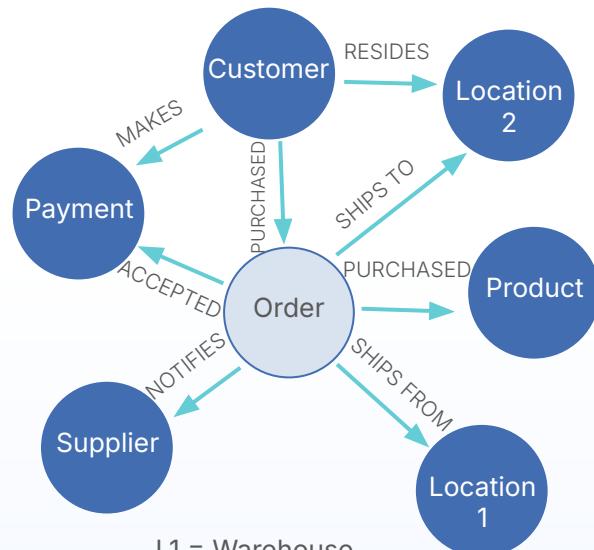
Relational Database



Key-Value Database



Graph Database



L1 = Warehouse
L2 = Delivery location

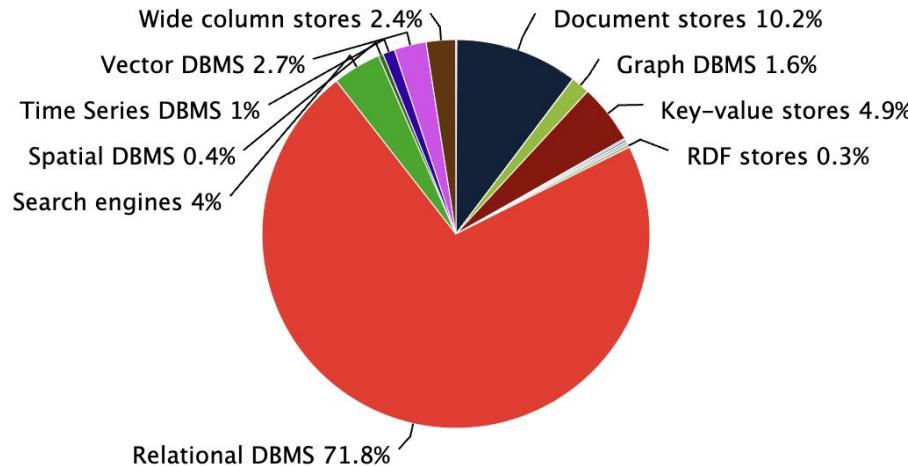
- ✗ Rigid schema
- ✓ High performance for transactions
- ✗ Poor performance for deep analytics

- ✓ Highly fluid schema / no schema
- ✓ High performance for simple transactions
- ✗ Poor performance for deep analytics

- ✓ Flexible schema
- ✓ High performance for complex transactions
- ✓ High performance for deep analytics

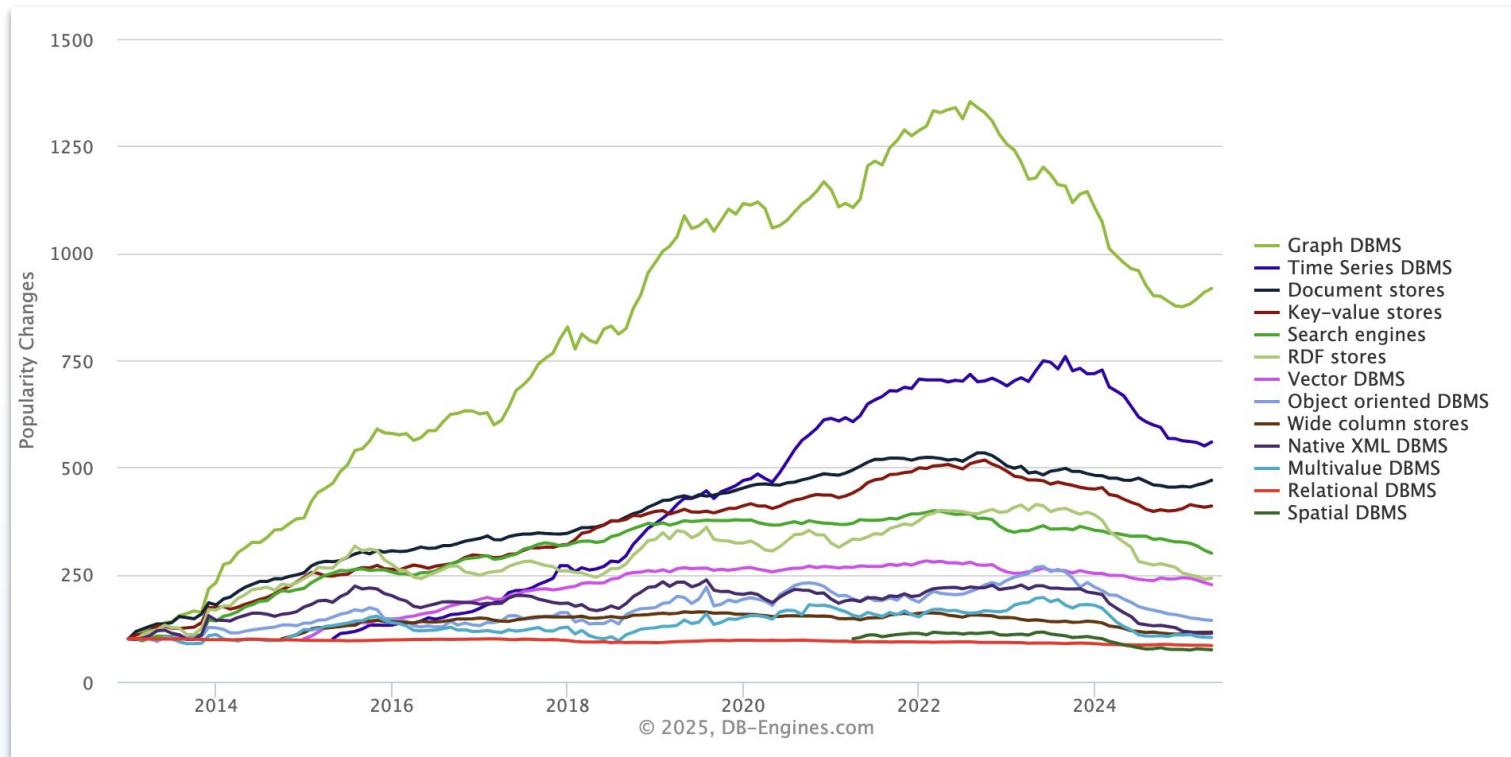
Database popularity

Ranking scores per category in percent, May 2025



© 2025, DB-Engines.com

Evolution of popularity



Query execution time

Jonas Partner and Aleksa Vukotic performed an experiment using social networks. They built a query in both MySQL and Neo4j (a popular type of graph database) with a database of 1,000,000 users. The tables comprised of the user's friends and friends of those friends.

Depth	Execution Time – MySQL	Execution Time – Neo4j
2	0.016	0.010
3	30.267	0.168
4	1,543.505	1.359
5	Not Finished in 1 Hour	2.132

Source — Results from the experiment conducted by Jonas and Aleksa

Cypher and GQL

Cypher is the query language for Neo4j, the leading graph database, and its specifications are public under the OpenCypher spec.

GQL is a standard language to query graph databases. It's been recently standardized by ISO. GQL aims to be the SQL of graph databases.

GQL is heavily inspired by Cypher, and Neo4j has contributed to its development, meaning GQL shares many similarities with Cypher.

Lab 1: Goals and Agenda

Part 1

Explore and compare different schema languages

- JSON Schema, GraphQL, Pydantic

Part 2

Explore and compare different databases (relational and graph)

- SQLite & Neo4j

Lab 1



[/opsmill/workshop-data-modeling](https://github.com/opsmill/workshop-data-modeling)



<https://bit.ly/ac4-ws-c2>



#ac4-ws-c2-nsot



Lab tips and tricks

Labs are running on the Instruqt platform. Everything is already installed and ready to go!

Between each step or challenge, you'll find helpful slides with more info and context about the task.

You can open these notes anytime using

[View notes](#)

We know laptop screens can feel a bit tight!

You can show or hide the assignment panel using

[Hide Instructions](#)



Getting started with the Infrahub Lab

BREAK

Schema: Advanced Concepts

Advanced schema concepts

Migrations update the data to match the new version of the schema.

Inheritance allows object to inherit structure or attributes from a parent or base object / entity.

Polymorphism allows systems to handle different types of related entities through a shared interface or structure.

Migrations

Updating a schema is easy.

Updating the existing data to match the new schema is the hard part.

Migrations

What is a migration? When do I need one?

Anything that changes the structure of the existing data will require some migrations.

If there's no data associated with the schema, no migration is required.

Migration examples

- Add an attribute
- Change the type of an attribute: String > Boolean
- Change the name of an object
- Change the relationships between objects

Migration strategy per platform

Git

Application developer needs to update the data manually or with a script.

In-house Application

Application developer needs to provide the migrations for each update. Some libraries are available to help.

NetBox / Nautobot

Migrations are built into the core products and the plugins.

Infrahub

Some migrations are automatically handled by the platform. For the other, the platform is running some validation to ensure the data has been updated prior to the migration.

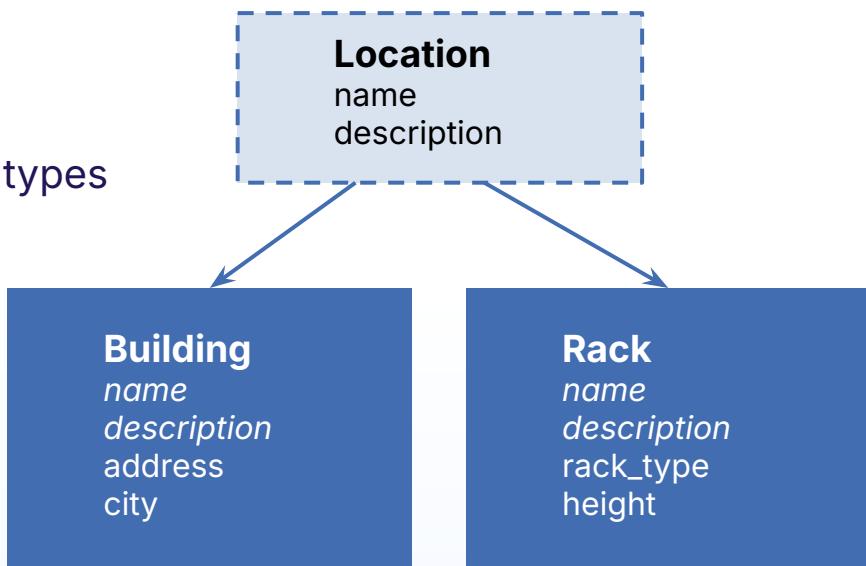
Inheritance / polymorphism

Inheritance allows object to inherit structure or attributes from a parent or base object / entity.

Polymorphism allows systems to handle different types of related entities through a shared interface or structure.

Use cases

- Reusability
- Precise schema per object
- Hierarchical data
- Simplify relationships between objects
- Easier to extend schema over time

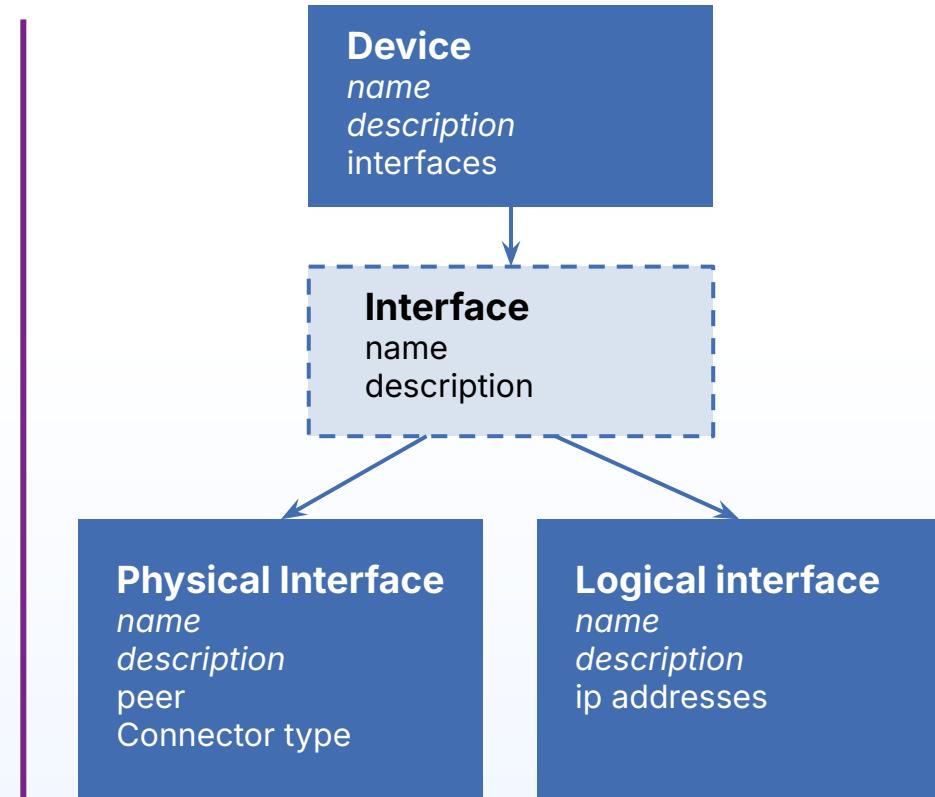
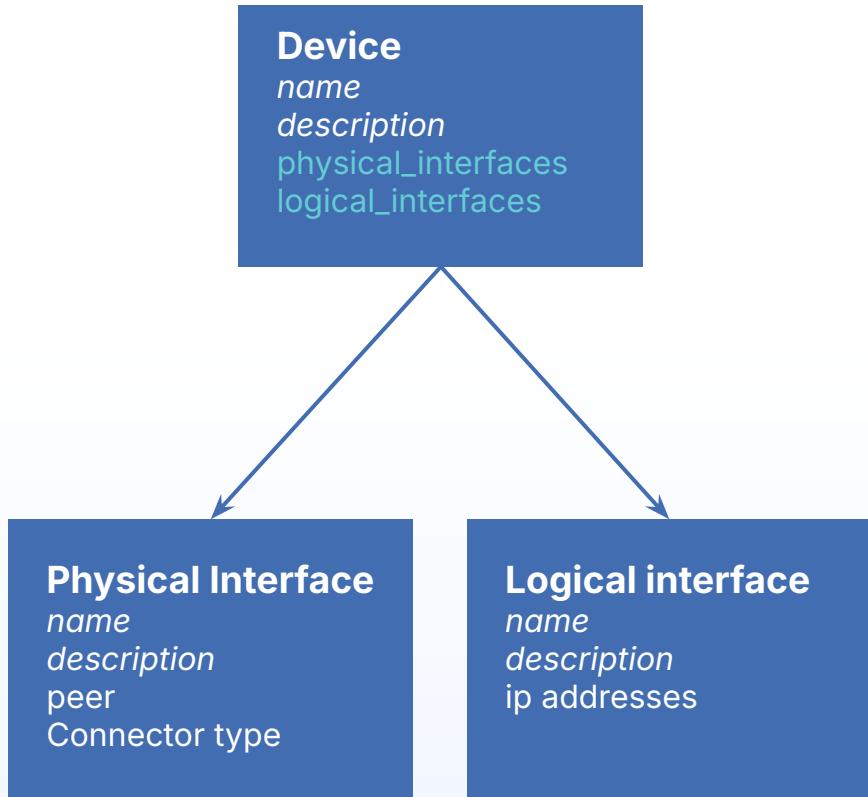


Inheritance / Polymorphism

Inheritance is about the data structure and reusing/sharing attributes and relationship between objects. It ensures consistency.

Polymorphism is about supporting multiple types of objects behind the same relationships.

Polymorphism



GraphQL interface and fragments

GraphQL natively supports inheritance via interface and fragments.

```
query {
  network_interface {
    id
    name
    parent {
      name
    }
    ... on PhysicalInterface {
      connector_type
    }
    ... on LogicalInterface {
      ip_addresses {
        address
      }
    }
  }
}
```

Query

```
interface NetworkInterface {
  id: ID!
  name: String!
  description: String!
  parent: Device!
}

type PhysicalInterface implements NetworkInterface {
  peer: Device!
  connector_type: String!
}

type LogicalInterface implements NetworkInterface {
  ip_addresses: [IPAddress]!
}
```

Schema

Inheritance / polymorphism support

Schema Language	Inheritance	Polymorphism	Comments
SQL	Partial	Partial	Not part of SQL , supported by Postgres
JSON Schema	Yes	Yes	through oneOF
GraphQL	Yes	Yes	through interface
Yang	Yes	Partial	through grouping and augment
Infrahub	Yes	Yes	through generic

Beyond the schema

Model vs schema

A model is a superset of a schema that includes application-specific logic and validation.

A **general purpose schema** describes only the structure, constraints, and rules of the data.

A **model** encompasses not only the structure but also behavior and logic around the data.

Presentation

Display

Model (Software)

Data Integrity

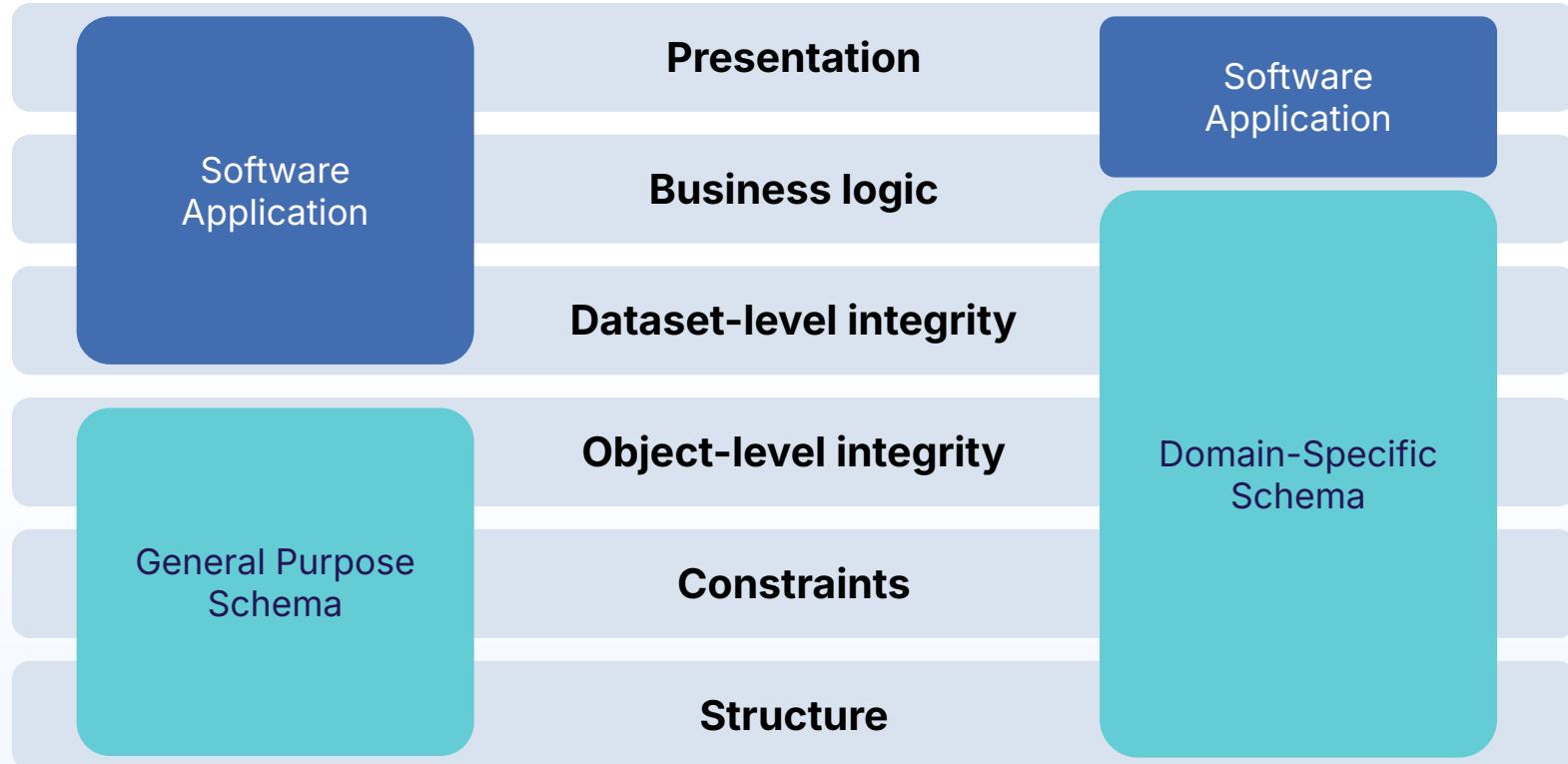
Domain Specific

Structure

Constraints

Schema

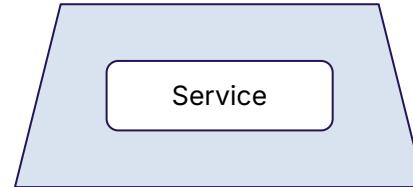
General purpose vs domain-specific schema



Part 2: **Network infrastructure modeling** **in a source of truth**

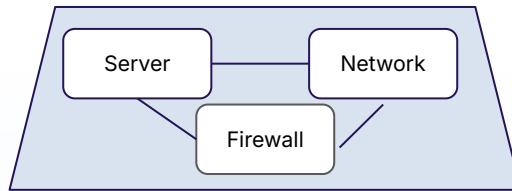
Data in layers

Multiple layers of infrastructure data



Service / Intent Layer

Definition of what services the infrastructure needs to deliver



Technical Layer

Global representation of the infrastructure elements interconnected



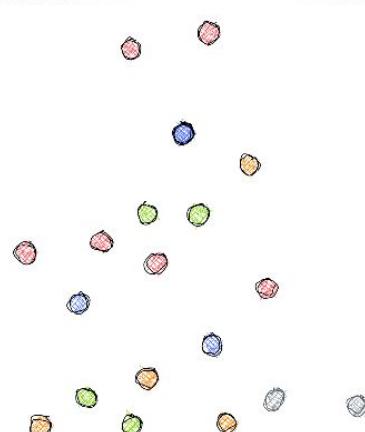
Component Layer

Each element is managed individually

Information vs intent

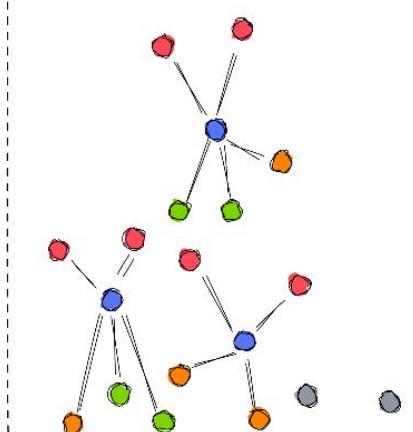
Component

Information



Technical

KNowledge



Service / Intent

Intent / Wisdom

Instance A

Instance B

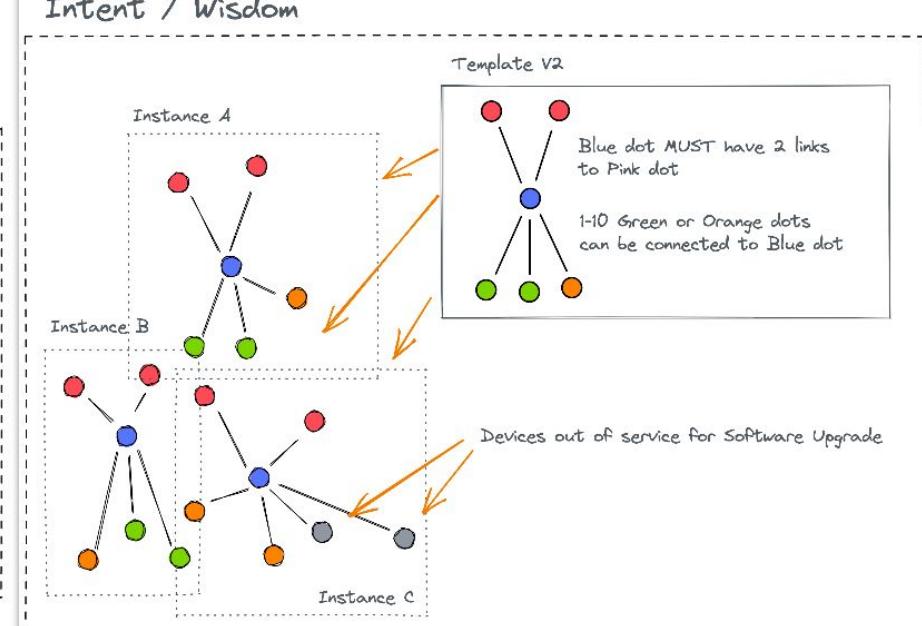
Instance C

Template V2

Blue dot MUST have 2 links to Pink dot

1-10 Green or Orange dots can be connected to Blue dot

Devices out of service for Software Upgrade



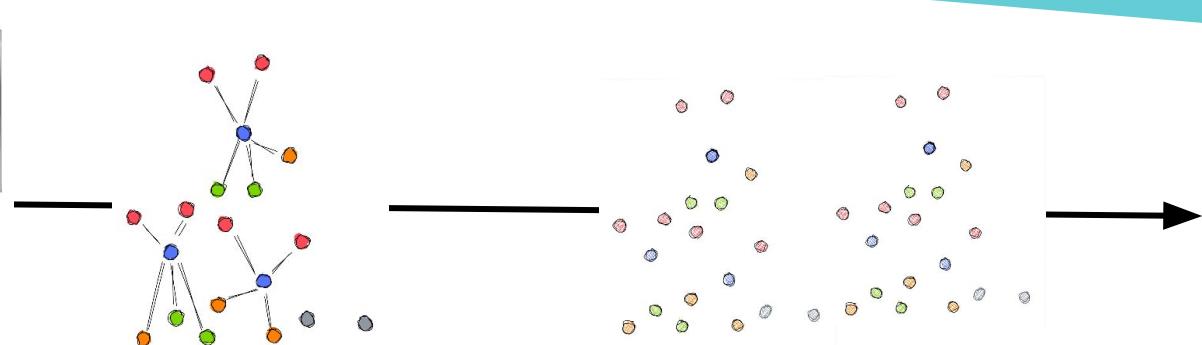
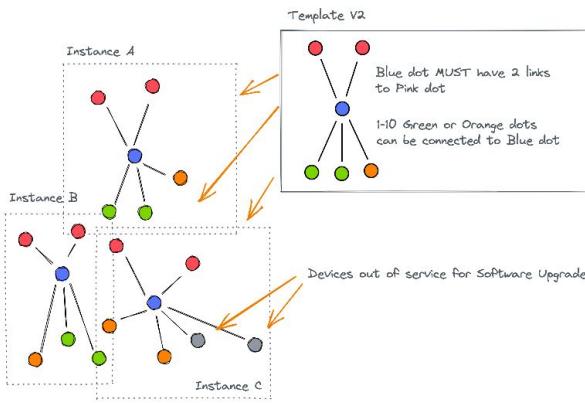
The data funnel



Intended State

Production System

CMDB
Documentation
ITAM / ITOM



Multiple layers of infrastructure data

I want LDAP from server hosting Application YY to communicate with All Domain Controllers

Firewall rule, ALLOW, port 389 from IP of server 1 & server 2 to IP of server 8 & server 10

Firewall rule, ALLOW, port 389 from IP1 & IP2 to IP5 & IP8

Service Layer

Technical Layer

Component Layer

Source of Truth



Configuration Artifacts

Business & operational context

Business contact / classification

Interface	Link status	Role	Status
Ethernet 1	Up	Uplink	Active
Ethernet 2	Down	Uplink	Maintenance
Ethernet 3	Up	Uplink	Active
Ethernet 4	Down	Server	Provisioning
Ethernet 5	Up	Server	Active

The 3 primary attributes

ROLE

Captures the primary function of an object

STATUS

Captures all the stages of the lifecycle of an object

KIND

Captures the nature of an object

Role

Role defines the main function of an element.

- **For a server:** is it a database or web server?
- **For a network device:** is it a core router or an access switch?
- **For a site:** is it a manufacturing site or an office?

In some cases a given object may have multiple roles, if it's delivering multiple functions. Example: a server hosting both a web portal and a file server.

Status

The status is meant to capture all the stages of the lifecycle of each object.

- Active
- Provisioning
- Maintenance
- Software-Upgrade
- Closed-for-Business

The list of possible statuses will vary greatly between a site and a server, but the idea remains the same.

Kind

The kind (or type) captures implementation differences.

- **For a server:** is it running Linux or Windows?
- **For a network device:** is it running Cisco or Arista?
- **For a site:** Is it a large office or a small one?

The kind is very important because usually it defines the implementation and helps manage vendor specific requirements.

Mapping workflows to role, status, and kind

```
- name: Reboot network devices
  hosts: status_maintenance
  gather_facts: false
  tasks:
    - name: Reboot EOS device
      arista.eos.eos_command:
        commands: [ "reload now" ]
      when: platform == "eos"

    - name: Reboot Junos device
      juniper.junos.command:
        commands: [ "request system reboot"
      ]
      when: platform == "junos"
```

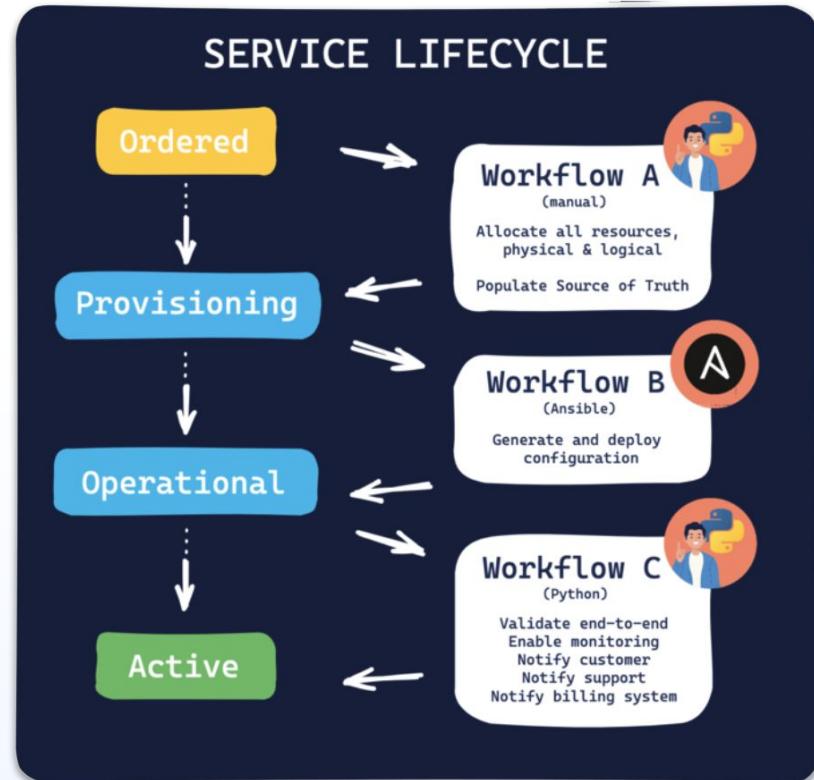
Every playbook should map to a group of hosts defined by their role and status.

Specific actions should be controlled by the kind.

Mapping workflows to role, status, and kind

Typically a workflow (manual or automated) is required to change the status from one value to another.

This approach helps to map a declarative approach and a workflow-based automation.



Design for idempotency

Infrastructure as Code principles

Idempotent: Always the same results

Version-control friendly: Input as text file, peer review

Safe & predictable: Plan everything before, know what changes will be made before you run it.

Design for idempotency

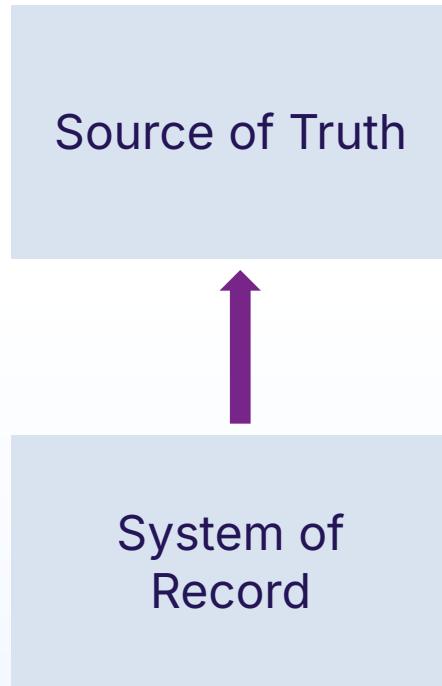
- Ensure all objects have a unique identifier that's independent of any systems
 - Unique names
 - Unique combination of names / relationships
- Support declarative API

Infrahub's schema integrates idempotency natively with the **Human Friendly Identifier (HFID)**.

Data synchronization

Data synchronization presents its own set of challenges

- How can we map objects from system A to system B?
- What's the state of the destination system before the sync?



A
B

Lab 2



[/opsmill/workshop-data-modeling](https://github.com/opsmill/workshop-data-modeling)



<https://bit.ly/ac4-ws-c2>

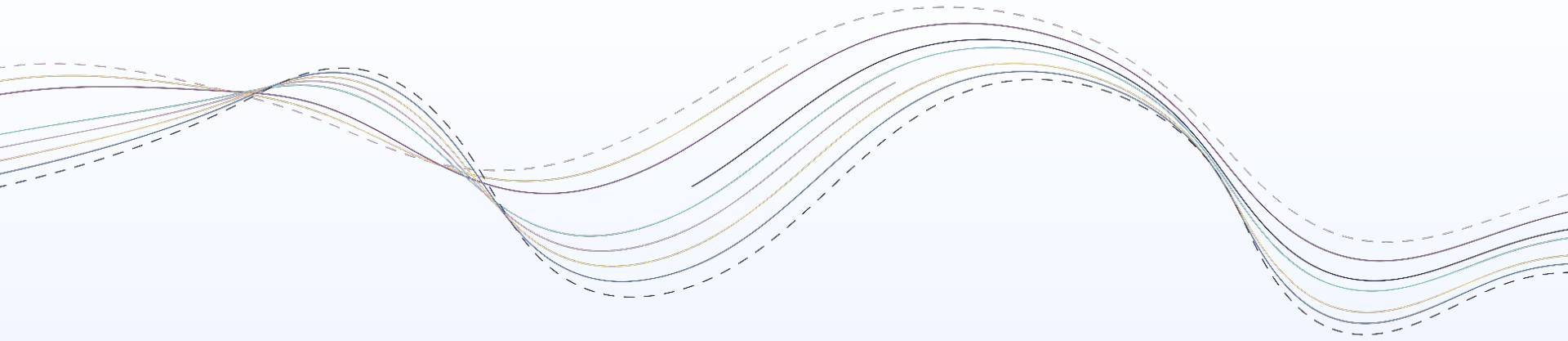


#ac4-ws-c2-nsot





Thank you



Schema: A closer look

SQL

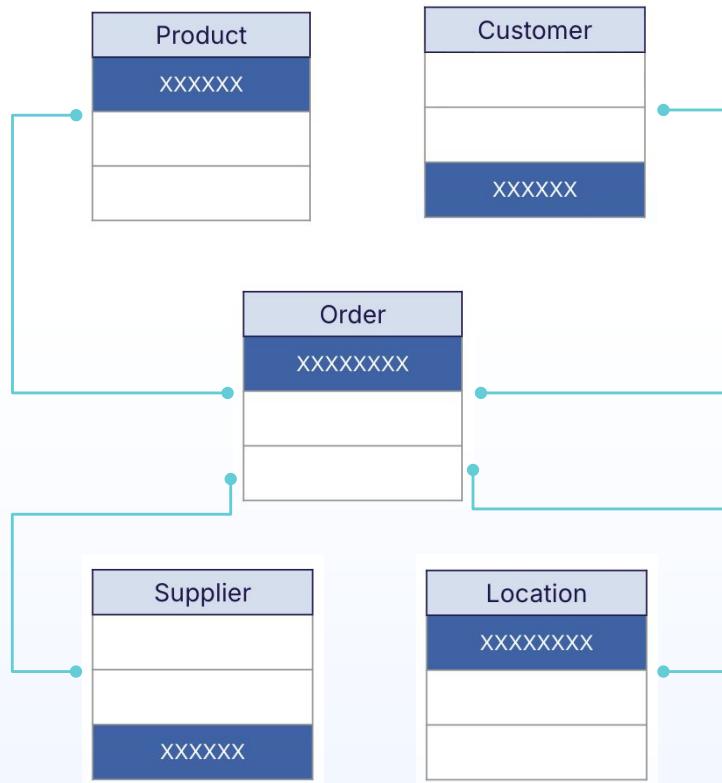
Introduced in 1970. The language of ALL relational databases.

Both a schema and a query language.

Key concepts:

- Schema is mandatory
- Data is organized in tables
- Additional features
 - Permissions
 - Transactions

SQL tables



- ✗ Rigid schema
- ✓ High performance for transactions
- ✗ Poor performance for deep analytics

SQL

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(50),
    Description TEXT,
    Price DECIMAL(10, 2),
    Quantity INT
);
```

Attribute → **Description** → Constraints

Schema

```
SELECT ProductName, Price
FROM Products
WHERE Price < 50;
```

Query

SQL key differentiators

- Standardization and wide adoption
- Used for schema, query, and storage
- Support for ACID

JSON Schema

Introduced in 2010. First draft in 2013.

The de-facto standard for JSON validation and structure definition.
Supported by many libraries, frameworks, and tools across
programming ecosystems.

Key concepts:

- Data structure definition
- Extensibility and modularity
- Leverage the concept of “REF”

JSON Schema

```
{  
  "title": "New Blog Post",  
  "content": "content of the blog...",  
  "publishedDate": "2023-08-25T15:00:00Z",  
  "author": {  
    "username": "authoruser",  
    "email": "author@example.com"  
  },  
  "tags": ["Technology", "Programming"]  
}
```

Data

External Ref

```
{  
  "$id": "https://example.com/blog-post.schema.json",  
  "$schema": "https://json-schema.org/draft/2020-12/schema",  
  "description": "A representation of a blog post",  
  "type": "object",  
  "required": ["title", "content", "author"],  
  "properties": {  
    "title": {  
      "type": "string"  
    },  
    "content": {  
      "type": "string"  
    },  
    "publishedDate": {  
      "type": "string",  
      "format": "date-time"  
    },  
    "author": {  
      "$ref": "https://example.com/user-profile.schema.json"  
    },  
    "tags": {  
      "type": "array",  
      "items": {  
        "type": "string"  
      }  
    }  
  }  
}
```

JSON Schema key differentiators

- Object definition can be imported from a remote location
- Works with most data structure, not just JSON
- Supports complex structure (heterogeneous / oneOf, allOf)

GraphQL

Developed at Facebook, released in 2015.

Schema, data query, and manipulation language for APIs.

Designed to make APIs fast, flexible, and developer-friendly.

Complementary / alternative to REST API.

Key concepts:

- Strongly typed schema
- Supports query & mutation
- Designed to be integrated with a storage engine

GraphQL schema / query

```
query {
  posts {
    id
    title
    author {
      id
      name
    }
    comments {
      id
      content
    }
  }
}
```

```
type Query {
  posts: [Post]          # Get a list of all posts
  post(id: ID!): Post   # Get a single post by its ID
  users: [User]           # Get a list of all users
  user(id: ID!): User   # Get a single user by their ID
}

# Types representing the data structures in the system.
type Post {
  id: ID!
  title: String!
  content: String!
  author: User!          # Relationship to the User type
  comments: [Comment]    # List of related Comment types
}

type User {
  id: ID!
  name: String!
  email: String!
  posts: [Post]           # List of posts authored by this user
}

type Comment {
  id: ID!
  content: String!
}
```

GraphQL schema / mutation

```
mutation {
  createPost(
    input: {
      title: "Introduction to GraphQL",
      content: "GraphQL is a query language for
[...] executing those queries.",
      authorId: "1"
    }
  ) {
    id
    title
    content
    author {
      id
      name
    }
  }
}
```

Input

Response

Mutation

```
type Mutation {
  createPost(input: CreatePostInput!): Post
  createUser(input: CreateUserInput!): User
  addComment(input: AddCommentInput!): Comment
}

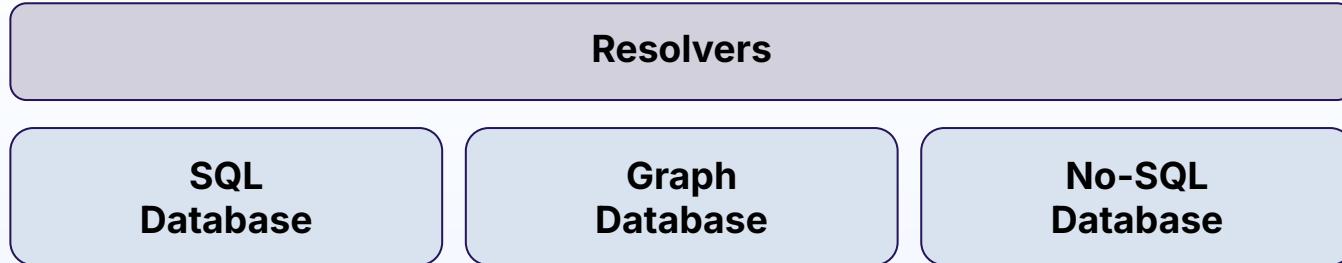
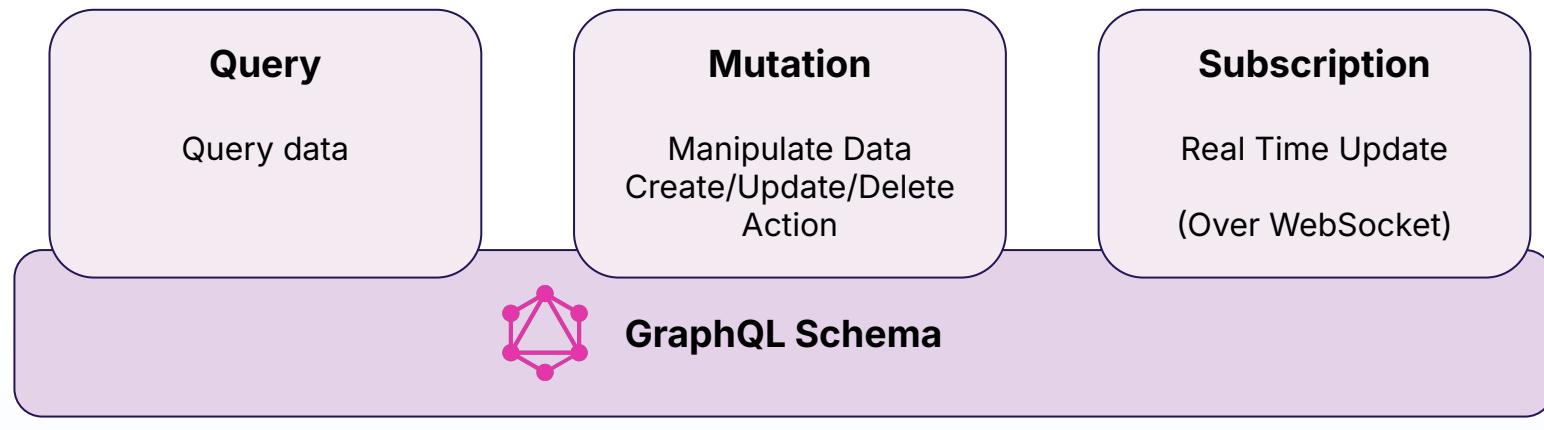
input CreatePostInput {
  title: String!
  content: String!
  authorId: ID!
}

input CreateUserInput {
  name: String!
  email: String!
}

input AddCommentInput {
  postId: ID!
  content: String!
  authorId: ID!
}
```

Schema

GraphQL



GraphQL key differentiators

- You get (only) what you query
- Supports inheritance (interface)
- Native support for subscriptions
- Designed to be integrated with a backend by default

YANG

YANG (Yet Another Next Generation) is a data modeling language designed for defining network configurations, state data, and operational behavior.

Standardized by the IETF and widely used in network automation and management.

Key concepts:

- Hierarchical, tree-based structure for defining data
- Easy to extend
- Supports reusability and modularity via groupings and augments.
- Integrates with protocols like NETCONF, RESTCONF, and gNMI for configuration and state management

YANG schema

Groups
related nodes

Attributes

```
module example-network {
    namespace "http://example.com/network";
    prefix "ex";

    container device {
        leaf hostname {
            type string;
        }
        leaf ip-address {
            type inet:ipv4-address;
        }
        leaf model {
            type string;
        }
        list interfaces {
            key "name";
            leaf name {
                type string;
            }
            leaf enabled {
                type boolean;
            }
        }
    }
}
```

YANG key differentiators

- Extensibility
- Integration with network protocols
- Hierarchical and modular data modeling

Infrahub schema

Domain specific schema which goes beyond a generic schema language.

Designed with infrastructure modeling in mind.

Provides schema, query, and storage out of the box, similar to SQL.

Key concepts:

- Domain-specific schema
- Captures how to store, query, and represent data
- Natively supports inheritance / polymorphism
- Supports hierarchical nodes & IPAM

Infrahub schema

```
---  
nodes:  
  - name: Device  
    namespace: Dcim  
    label: Network Device  
    icon: clarity:network-switch-solid  
    inherit_from:  
      - DcimGenericDevice  
      - DcimPhysicalDevice  
attributes:  
  - name: name  
    kind: Text  
    unique: true  
    order_weight: 1000  
  - name: height  
    label: Height (U)  
    optional: false  
    default_value: 1  
    kind: Number  
    order_weight: 1400  
relationships:  
  - name: platform  
    peer: DcimPlatform  
    cardinality: one  
    kind: Attribute  
    order_weight: 1300
```

Presentation

Structure

Relationship kind

Kind	Description
Generic	A flexible relationship with no specific functional significance. It's commonly used when an entity doesn't fit into specialized categories like Component or Parent.
Attribute	A relationship where related entities' attributes appear directly in the detailed view and list views. It's used for linking key information, like location.
Parent	This relationship defines a hierarchical link, with the parent entity often serving as a container or owner of another node. Parent relationships are mandatory and allow filtering in the UI, such as showing all components for a given parent.
Component	This relationship indicates that one entity is part of another and appears in a separate tab in the detailed view of a node in the UI. It represents a composition-like relationship where one node is a component of the current node.

Infrahub key differentiators

- Extensibility
- Migrations built in
- Natively supports inheritance & polymorphism
- Supports hierarchical nodes & IPAM

Summary

	JSON Schema	GraphQL	Yang	SQL	Infrahub
Flexibility	High Schema-less and easily adaptable	High Defined in the API Layer	High Models are easy to extend	Low Schema changes require migrations	Medium Some schema changes require migrations
Data Integrity	Limited Lacks strong constraints	Medium Client-driven, depends on backend logic	Medium	Strong Enforced with keys, constraints, and ACID	Strong Enforced keys, constraints,
Nested Data	Strong Suited for complex, nested structures	Strong Suited for complex, nested structures	Strong Suited for complex, nested structures	Weaker Requires complex table structures	Strong Suited for complex, nested structures
Use Cases	Dynamic or semi-structured data, flexible schemas	Dynamic API data with customizable queries	Network specific API Netconf, RESTConf, OpenConfig	Structured data with strict integrity needs	Infrastructure Source of Truth

Stateless and stateful

	JSON Schema	GraphQL	Yang	SQL	Infrahub
Flexibility	High	High	High	Low	Medium
Data Integrity	Limited	Medium	Medium	Strong	Strong
Nested Data	Strong	Strong	Strong	Weaker	Strong

Stateless

Schema only. Not coupled with a storage solution by design.

Stateful

Schema is coupled with a system to store the data, which means that any change in the schema may require some changes in the data as well (migration).

Cypher

Pattern matching

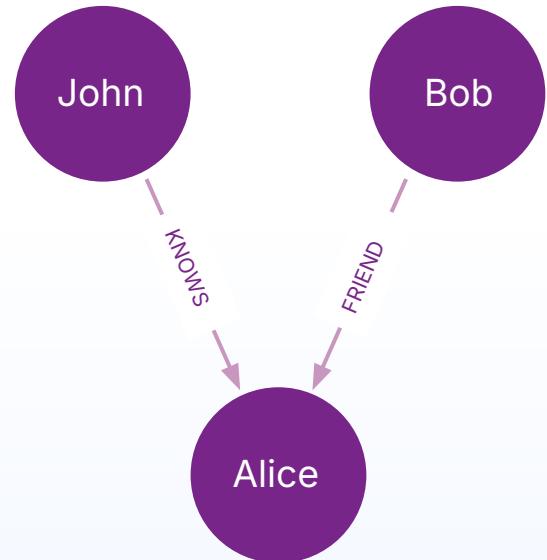
```
MATCH (a:Person) - [:KNOWS] -> (b)  
RETURN a.name, b.name
```

Filtering

```
MATCH (a:Person {name: 'Alice'}) - [:KNOWS] - (b:Person)  
RETURN b.name
```

Count

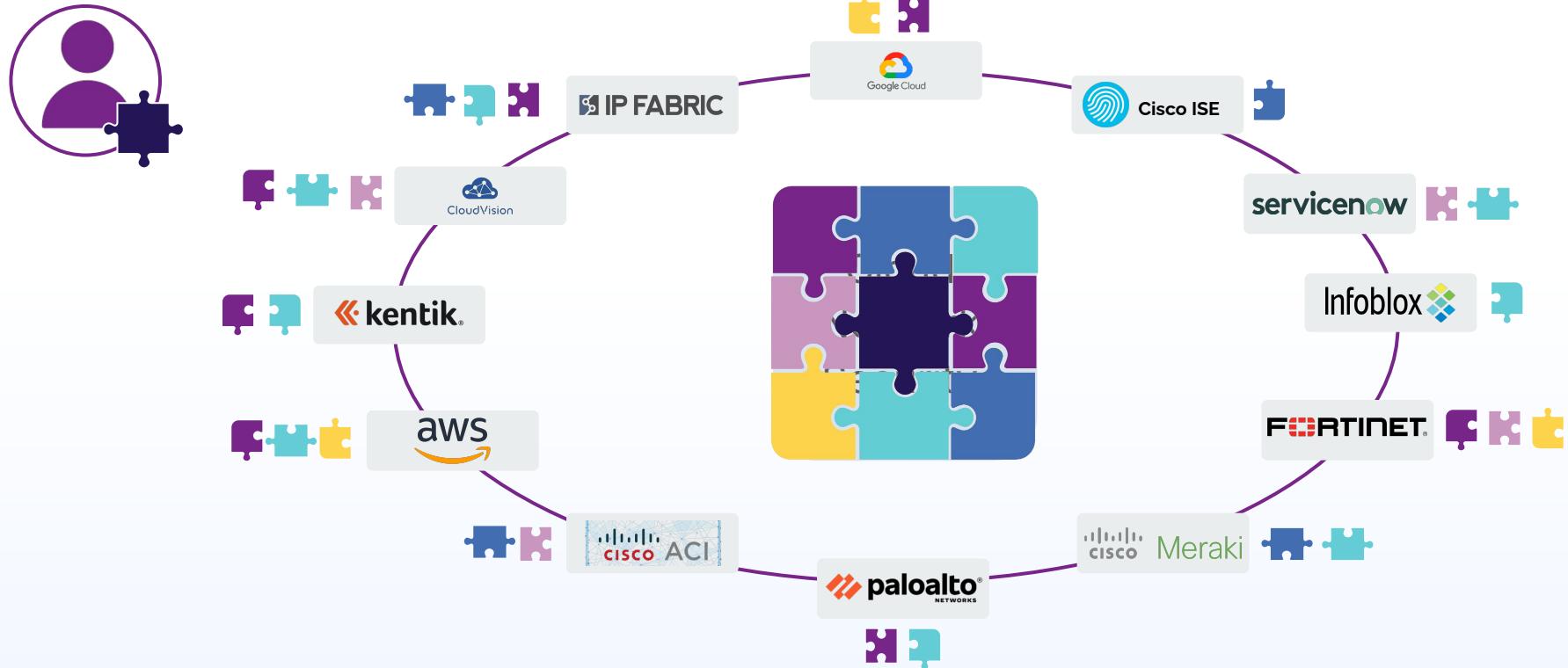
```
MATCH (p:Person) - [:KNOWS] -> (f)  
RETURN p.name, count(f) as friends_count
```



Data federation / aggregation



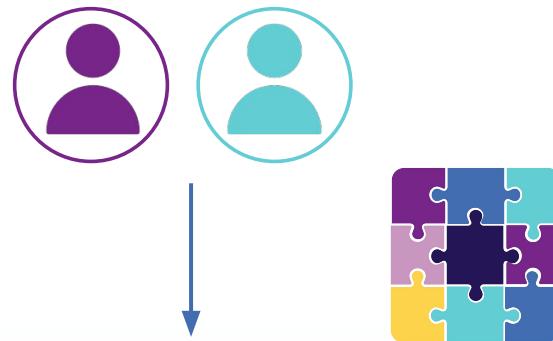
Many systems of record



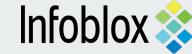
Intent federation / aggregation

Simplify intent consumption for all systems

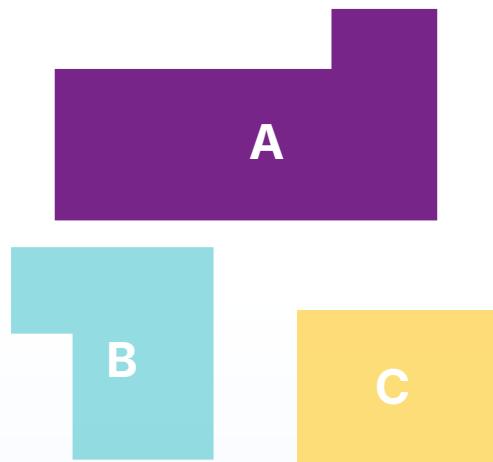
Abstract the complexity away



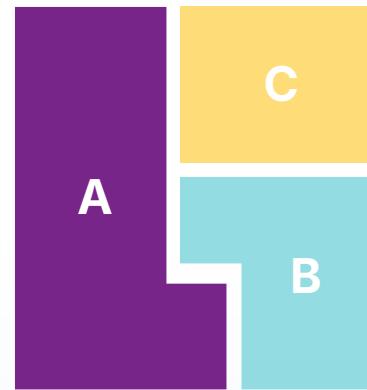
Intent federation / aggregation



Data federation / aggregation

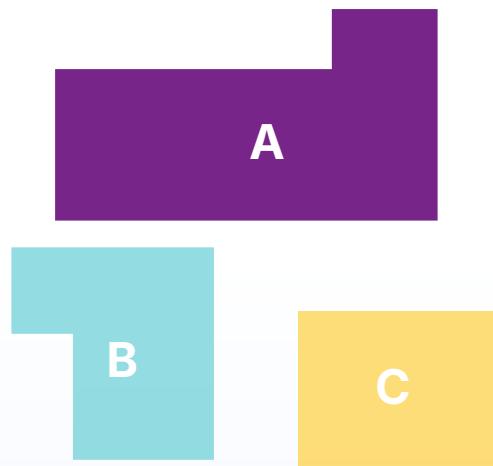


Individual Datasets

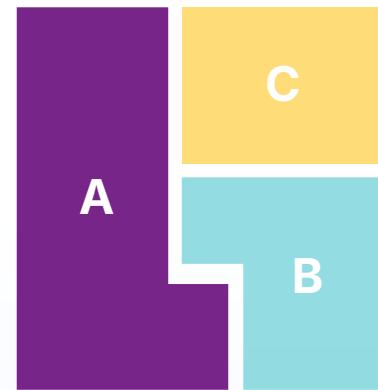


Schema

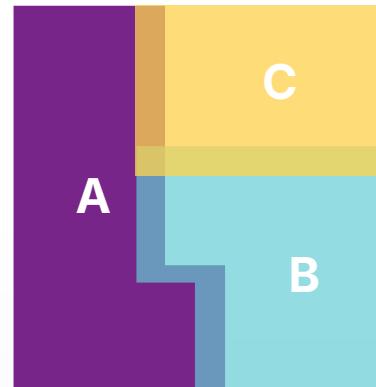
Data federation / aggregation



Individual Datasets

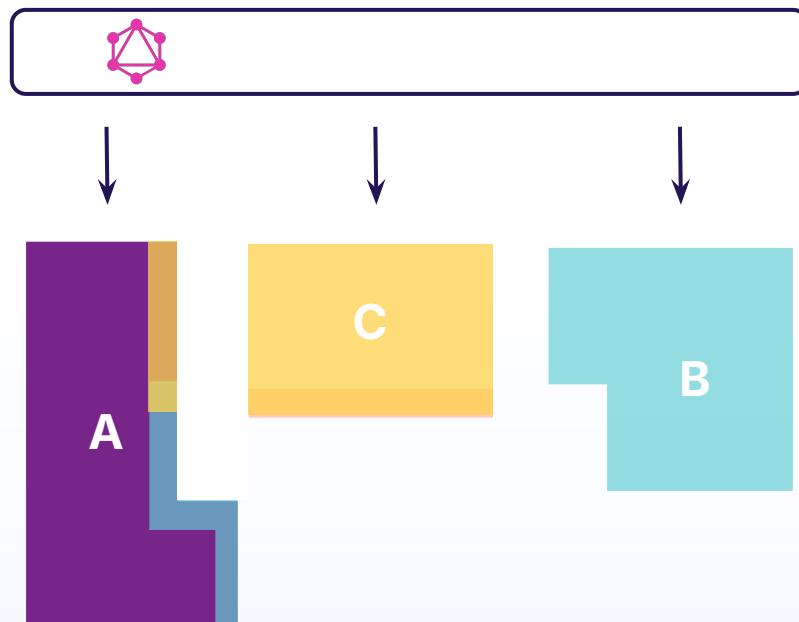


Schema



Connected Datasets
(Relationships)

Stateless federation



Data synchronization

