Beyond Data Mesh

Data Mesh[1] is a major advance over past architectures. However, it's the beginning, not the end of where we need to go to enable the business. Data Mesh is about 1) finding the right boundaries around apps and data and 2) putting stable access in place at those boundaries via different types of APIs. This article discusses a next generation approach to finding the right boundaries and then enabling the business to access the domain and operate inside it. The picture below sets the context we'll cover:

- The left side shows the core of the standard data mesh architecture of nodes in a data mesh with APIs for access.[2]
- The right side looks at a single data mesh node, its APIs, and the apps and data inside the boundary. We'll build details on the right side throughtout the article.
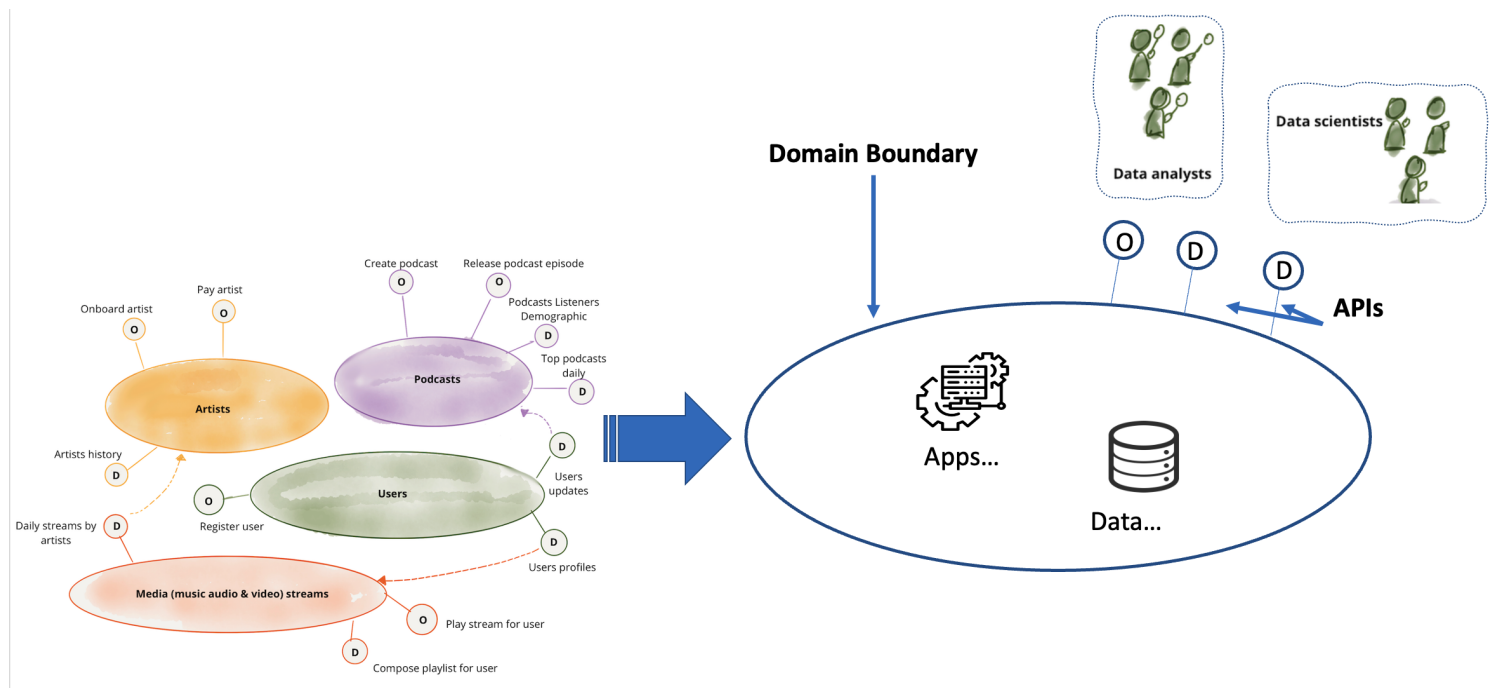
Finding the right boundaries for the nodes of the domain is critical your data mesh architecture. What I've seen of data mesh approaches is to focus on the data products to establish the boundaries. This is starting too data-centric. The secret to finding the boundaries is the ubiquitous language of the domain. The Domain Driven Design (DDD) community proposes various ways to discover, document, and visualize the ubiquitous language:

> The ubiquitous language is not just a set of terms or jargon, but a shared understanding of the domain and its problems. It reflects the domain model, which is the conceptual representation of the domain in code. The ubiquitous language and the domain model should evolve together, as the developers and the business experts learn more about the domain and refine their solutions. The ubiquitous language

> is important because it enables collaboration, alignment, and clarity among the different roles and perspectives involved in the software project.

The boundaries are at the points where the language of the domain's data models and processing changes. We use the language and the boundary it defines to find the data products and APIs. Using the language also enables us to clearly understand what's going on behind those boundaries and to better enable that work.[3]

Domain Languages

Combining data mesh and DDD thinking is a good start. The DDD approach stops at things like, dictionaries, context maps, and living documentation as a dynamic form of documentation generated from source code and tests. Powerful capabilities are found by going beyond this and formalizing our understanding and definition of the ubiquitous language. That may sound cryptic or scary but it's not because there is a well established discipline and community for building Domain Specific Languages (DSLs) to help us and we don't need to go all the way to implementing a DSL to get a lot of the power we're after.[4]

Formalizing the language means we work with the Subject Matter Experts (SMEs) of the domain to define the structure and syntax of the ubiquitous language. The syntax is built on existing notations and conventions used in the domain, e.g., text, tables, symbols, and diagrams, not just lots of keywords and curly braces. It requires becoming very clear – formal! – about the concepts that go into the language. In fact, building the language, because of the need for formalization, helps you become clear about the concepts of the domain in the first place. The benefits of the approach happen right away because language definition acts as a catalyst for understanding the domain![5]

Defining and potentially implementing a true DSL is the way to get to the ultimate in power and differentiation of a data mesh solution. Thankfully, we don't need to start doing full language engineering to get benefits of a language based approach. Lets look at options for a first step short-cut to our desired result: dbt.

Whether taking a language-centric approach or not, dbt is one of the best technologies to implement the core architecture of a data mesh. Even better, is it enables us to smoothly move along the path to a full DSL for the ubiquitous language approach proposed in this article. I'm not affiliated with the company behind dbt. I'm using it as a concrete example of the features needed to take this approach. That said, I am advocating for it's use because I really like it.
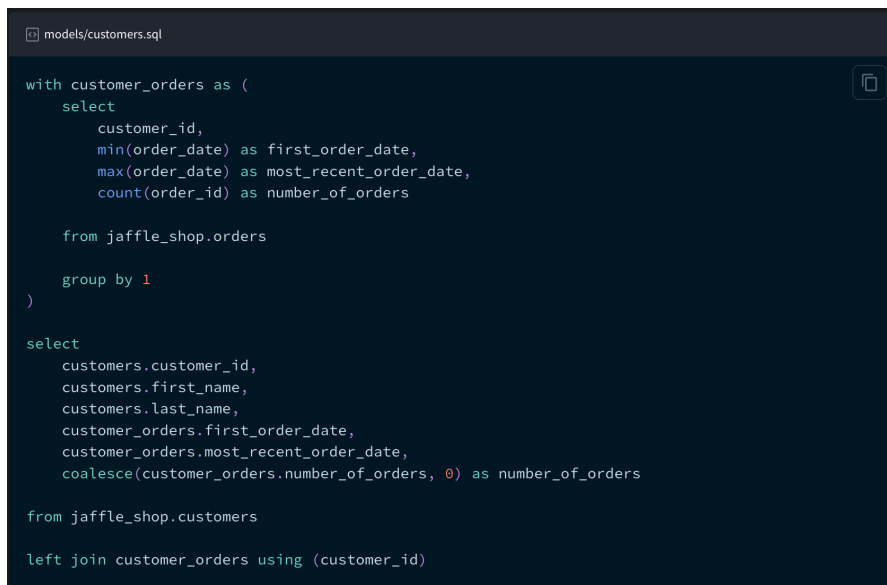
For those not familiar with dbt, the following are the important parts a dbt solution is built from:

- *Models* - Each model lives in a single file and contains logic that either transforms raw data into a dataset that is ready for analytics or, more often, is an intermediate step in such a transformation. The

essential thing in a model is some SQL.

- *Sources* - A way to name and describe the data loaded into your warehouse by your Extract and Load tools.
- *Tests* - built from SQL queries that you can write to test the models.
- *Exposures* - A way to define and describe a downstream use of your project.
- *Macros* - Blocks of code written in Jinja, a templating language that you can reuse multiple times.

The following pictures show examples of some dbt configuration language files. The first is a dbt model which, in this example, is nothing more than SQL placed in a file in the proper place in the dbt configuration structure.

```sql
models/customers.sql

with customer_orders as (
    select
        customer_id,
        min(order_date) as first_order_date,
        max(order_date) as most_recent_order_date,
        count(order_id) as number_of_orders

    from jaffle_shop.orders

    group by 1
)

select
    customers.customer_id,
    customers.first_name,
    customers.last_name,
    customer_orders.first_order_date,
    customer_orders.most_recent_order_date,
    coalesce(customer_orders.number_of_orders, 0) as number_of_orders

from jaffle_shop.customers

left join customer_orders using (customer_id)
```

The next picture shows the template for the additional configuration of a model. It's just more *configuration language* in a text file.

```
version: 2

models:
  - name: <model name>
    description: <markdown_string>
    docs:
      show: true | false
    latest_version: <version_identifier>
    deprecation_date: <YAML_DateTime>
    access: private | protected | public
    config:
      <model_config>: <config_value>
    constraints:
      - <constraint>
    tests:
      - <test>
      - ... # declare additional tests
    columns:
      - name: <column_name> # required
        description: <markdown_string>
        meta: {<dictionary>}
        quote: true | false
        constraints:
          - <constraint>
        tests:
          - <test>
          - ... # declare additional tests
        tags: [<string>]

      - name: ... # declare properties of additional columns

    versions:
      - v: <version_identifier> # required
```

Models are built by accessing the data exposed by other models or sources. A dbt solution built using this kind of configurations can be the core of a data API for a data mesh domain. You use SQL or additional dbt models to access the models defined as the data products of the domains in you data mesh. You formally create your data products by *exposing* them. Dbt has some basic features to control access, e.g., the Exposures described above, and they advancing those features rapidly.

All of the parts of a dbt solution are specified using the same kind of file-based configuration language. This language is the first iteration of automation of our domain language. You could just use these basic out-of-the-box dbt features and implement a reasonable data mesh. All of the configuration files taken together forms a domain language for your data transformation and access workflows. The model specifications are the only domain specific parts created using the language. Dbt's language is a low level and business domain independent language rather than the domain specific language we aspire to.

Dbt Macros as the Beginning of a DSL

We move to being more of a DSL through the use of dbt macros. Macros, written using dbt's Jinja features, are pieces of code that can be reused multiple times. Using macros we can build higher-level abstractions that are specific to the business domain. We do this to avoid having SMEs creating new data products need to rewrite common complex logic. Instead, we can write it once as a macro and simplify and standardize that part of the logic. Programmers look at this as simply not repeating ourselves (DRY). More important than just avoiding repetition we need to design the macros so they align with the ubiquitous language of the domain. There are significant limits to what we can do with macros and there is still a lot of dbt complexity and detail exposed. However, for the right audience, domain specific macros can still be a major step forward.

Using this approach the architecture of a data mesh node (the right side of the context picture) looks like the following.
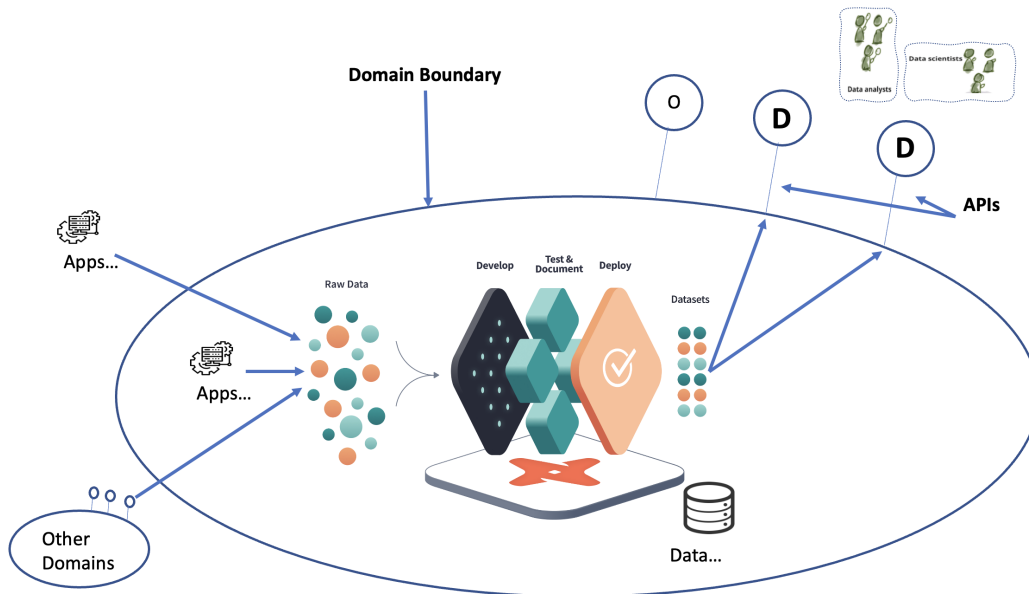
The dbt configurations are executed as the logic of the domain to produce models. The exposed dbt models serve as the **D**ata API[6]. That access can be via raw SQL or by creating new dbt models outside the domain boundary that use a data product.

Adding Metrics to the Language

The next step along the path to a DSL is already part of dbt: the dbt Semantic Layer. "The dbt Semantic Layer allows data teams to centrally define essential business metrics like revenue, customer, and churn in the modeling layer (your dbt project) for consistent self-service within downstream data tools like BI and metadata management solutions. The dbt Semantic Layer provides the flexibility to define metrics on top of your existing models and then query those metrics and models in your analysis tools of choice."[7]. This layer is a language for defining metrics. Dbt talks about its value from the technical perspective. We're looking at it as another part of our domain specific language. The business surely includes a lot in their ubiquitous language about the metrics, e.g., how are they named, how are they calculated, how do they evolve over time and where are they used. The following shows an example of a metric defined in the dbt language.

```yaml
version: 2
metrics:
  - name: total_revenue
    label: The revenue of our business
    model: ref('order_events')
    description: "The revenue for our business, as defined by Jerry in Finance"

    calculation_method: sum
    expression: amount

    timestamp: event_date
    time_grains: [day, week, month, all_time]

    dimensions:
      - customer_status
      - order_country
```
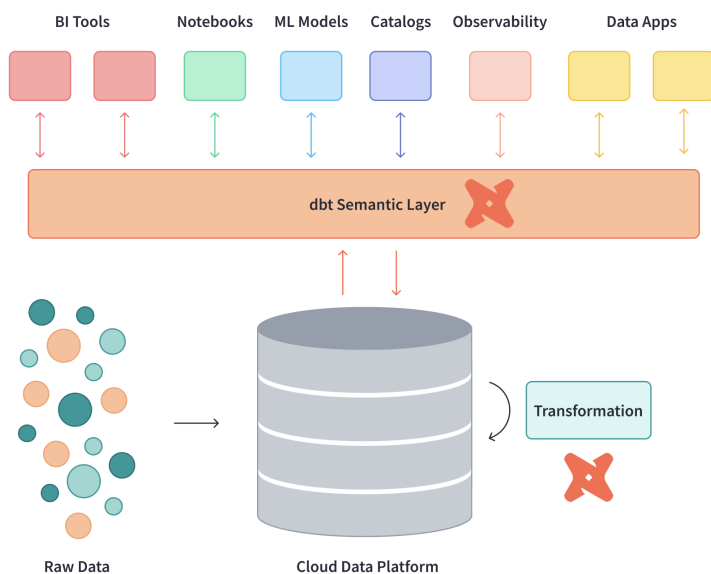
The following shows how the semantic layer fits into business use.

Examples of the kinds of metrics that can be expressed in the language:

- Expressions, e.g., `transactions — cancellations`
- Ratios, e.g., revenue per customer
- Cumulative Metrics, e.g., weekly active users
- Aggregation types, e.g., sum_boolean and percentile TODO: get better example of aggregation types

I see the value of a central definition of metrics in a semantic layer as transformative for a business. It will have dramatic effects on standardizing everything from basic BI reporting to the most advanced AI. The fact that the business can now see and configure the definition is a big part of this transformation.

Similar to the previously introduced parts of dbt, even the metrics language is low level and generic rather when compared to the what SMEs use the ubiquotous langage do describe metrics. However, once the metrics are defined, using them in combination with the domain specific dbt macros is a significant step forward.

A Full DSL

There are limits to how well we can model the ubiquitous language of the business using dbt or similar generic tools. Our ability to really model the language becomes possible when we formalize our understanding of the language of the domain as a DSL. With the infrastructure of something like dbt we can have the DSL generate dbt configurations that do what the semantics of the DSL specify. The DSL isn't limited to just generating dbt. It would generate whatever is needed to perform the DSL statements. The following figure introduces the what the architecture would look like when we introduce a full DSL.
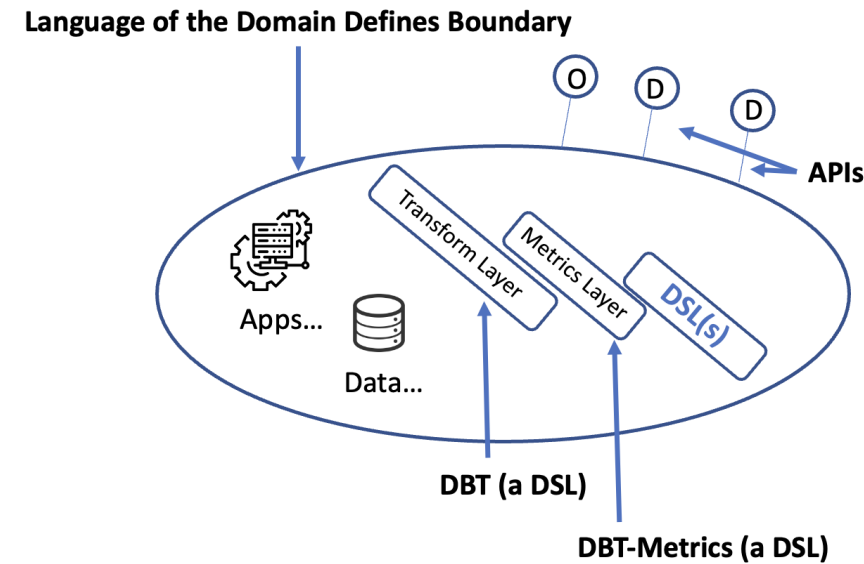
The following are concrete examples of formalization of the language of the domain from my recent projects. The examples go from simple and more technical language structures to more complex and domain specific:

Data Vault Creation and Evolution

A data vault[8] is a great data structure for use inside domains of the data mesh. Covering data vaults requires a separate article. For this example, just consider the vault to be a complex relational database structure that is used to organize the data into a kind of graph build from Hubs, Satellites, and Links. Creating and modifying data vaults was a common task on multiple of my projects. This example can be considered a *technical* DSL used by specialists responsible for loading the data. Initially, setting up the vaults was the domain of the modeling team, over time the SMEs started to propose the structures and talk in terms of the DSL when describing the data they wanted to access. We implemented a DSL for creation or change of a vault via a series of one line statements, e.g., the following is a simple examples that set up a hub and then does a data quality check to verify it worked:

	A	B	C	D	E	F
1	Source Table	Source Variable	Instruction	Target Table	Target Variable	Comments
2	VISIT_SCHEDULE		Hub(BK:BK_STUDY_ID,From:STUDY_NAME)(BK:BK_VISIT_ID,From: VISIT_NUMBER)	HUB_VISIT		
3						

	A	B	C	D	
1	**Run Time Check**	**Source/Target**	**Table**	**Variables**	
2	Exists(HUB_STUDY)	Target	HUB_VISIT	BK_STUDY_ID	
3					

In this example we didn't generate raw dbt configurations. Instead we used a the dbt vault extension AutomateDV[9] to simplify the implementation. Such a tool can be considered yet another generic technical DSL layer on top of dbt. A DSL was needed here because even using this extension was too technical. Using the extension took too much effort to use and test even for a dbt expert.Our modelers barely understood dbt but their language matched the new DSL.

Clinical Trial Data Mapping and Transformation

There are SMEs who's job is to load and convert clinical trial data from arbitrary input formats to an industry standard format. They need to do custom versions of this for every clinical trial project and then deal with a series of changes. We implemented a language to express the mappings and transformations where SQL was only used in very special cases. An example of the kinds of high-level domain specific instructions are those for processing data from laboratory tests which converts multiple laboratory values in a horizontal data layout, pivots it to be vertical as required by the standard and automatically deals with standard conversion tables and normal range checking. While you may not understand the details of this, describing how to do this is a central part of the ubiquitous language of clinical trail data. It typically requires detailed specifications that are then implemented as custom ETL or complex SQL. We implemented a single instruction, e.g.,

```
Lab Stack("WBC","WHITE BLOOD CELL COUNT","HEMATOLOGY","","BLOOD",LBHLAB,GEND,WBCRES,WBCU_)
```

A clinical data conversion SME would read this like a sentence because the parameters are in an expected order of the domain when working with this type of data. This is a simple single instruction. We also have sequences of instructions that work as a unit, e.g., *Nesting* is the ability to use instructions within each other to provide seamless transformations while eliminating the need for temporary variables. We implemented a large set of interlocking statements like this that allowed a SME to perform their entire job. Each statement generated dbt configurations and additional code and then our runtime system executed it.

Full Clinical Trial Specification

Both of the above DSL examples were *relatively* simple languages because their scope was relatively small and they were relatively technical. They were substantially easier to implemented because they were a layer on top of dbt. A tool like dbt is ideal for DSL creation because it is text based (a.k.a. configuration-as-code). The DSL then generates the dbt configuration files[10]. The focus can be on creating the language of the domain rather than that plus deep technical challenges related to making it possible to execute the DSL

instructions. Next, I'll briefly describe another example that doesn't use dbt but supports a much richer domain at a much more domain specific level.

Clinical trails are done to evaluate new medicines. They always start with writing a scientific specification of the evaluation called a *Clinical Protocol*. We built a DSL that enables the SMEs that specify data collection, calculations, workflows, and reports to be run on specialized clinical trial software systems. The following show examples of the IDE for defining these configurations.

The following shows[11] an example of how a DSL can look like a form but still contain complex domain specific instructions. For example there are multiple expressions in the fields that reference data in other parts of the DSL, e.g., "First Scheduled" is defined as "Activation Completion + 6 days". These expressions can be arbitrary complex and the user is guided so that they only create valid expressions while still just typing.

The following shows a more complex DSL structure for defining when patients move through the phases of the clinical trial, e.g., they move when specific expressions about the eligibility critera evaluate to true. The implementation of this is essentially a state transition diagram. While we have the capability to build a pure state transition DSL, we instead used the language of the SMEs in the domain. Notice that the word 'event' has a red squiggle under it. This is an example of a rich IDE that detects inconsistencies in the language as the user types and shows them as errors.

A DSL with this level of complexity requires more infrastructure than just a way to generate dbt files. In this case the runtime that needed to be configured is made up of many separate systems each different and each evolving what they support at different rates. Luckily, there are powerful tools available for building DSLs of this complexity.[12]

Language Spectrum

How far to go on the spectrum of domain languages depends on the domain. A full DSL is appropriate when the work of specifying a solution:

- Has complex rules, data, and processes and if the work of specification is dominated by business considerations rather than technical details
- Is repeated frequently by different SMEs in the domain

This level of work justifies the extra implementation effort for a full DSL. That work could be inside a single domain or across a closely related set of domains. When considering the boundary of the language don't get trapped into thinking that a domain is a simple flat structure. Domains are almost always a hierarchy containing sub-domains. Modeling the data mesh nodes and the languages they use needs to consider what level in the hierarchy of domains is the right place to establish the boundary to best serve the business needs.

Examples of applying this rule for deciding if a full DSL is justified:

- The above example of of specifying and automating the execution of the data collection and processing of a Clinical Trial justifies a full DSL because every trial is unique, the specification is dominated by the combination of how the science drives the technical details, and many trials are run in a year.

- Specifying the tax rules for a country. The rules are complex, they change across each year so must be re-specified, and the specification is dominated by a mix of business and human complexity.[13]
- Specifying the data products, analytics, metrics, and BI reports for a financial product. If the specification changes for every customer in complex ways and lots of new customers are setup regularly.

A situation that might justify a full DSL even if the above criteria aren't met is if there is a lot of experimentation needed to find the right version of the configuration, e.g., as part of a rapid selling process the spec needs to be evolved and simulated.

A domain language effort can start with the generic out-of-the-box domain language features of a tool like dbt and over time evolve to a full DSL.

How does buying a vendor solution fit into the DSL building decision. Vendor products need to be generic rather than domain specific so their target market size justifies their business. There are surely domain specific products, e.g., a product targeting clinical trials or financial investment products. Products always struggle with being domain specific enough to exactly match the organization and function of your business domains. Even the *configurable* or *programmable* products I've worked with over the years are rarely customizable enough to become truly domain specific without contorting them to the point where there is a major struggle to maintain them. Those that do offer configuration should be on the path to the ideal configuration via a DSL. When considering products which are on the configurability path, favor those that enable you to add your business specific specification via something like a DSL, e.g., products that are based on configuration-as-code like dbt.

Data Mesh APIs

The previous sections focused on how dbt or an DSL that extends dbt would serve the Data APIs. We haven't talked about how to implement the regular API, e.g., http REST calls to retrieve data or do other processing.[14] These are the APIs labeled with 'O' in the following figure. It is my believe that there is a deep problem with the current state of APIs and how clients use them, especially when we are trying for strong domain boundaries. APIs typically do one, rather restricted thing, e.g., retrieve some data possibly filtered, store some data, launch some processing. Ideally the APIs match the part of the language of the domain that we want to expose to clients. Current technology doesn't allow an API to do the kind of rich semantic operations that the ubiquitous language supports. The client needs to string together API calls to do something like select some data, transform it, calculate something, format it, and bring back the right subset of the results. I'm not talking about just SQL statements. I'm talking about doing interesting things in the ubiquitous language. DSLs offer a novel way to define APIs that solve this problem.

Language of the Domain Defines Boundary

DBT (a DSL)

DBT-Metrics (a DSL)
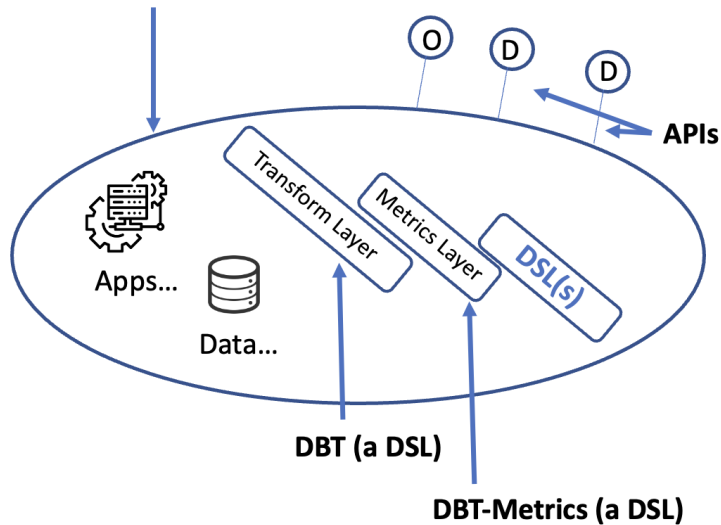
In a DSL-based architecture, the API accepts a group of statements in the domain language, executes them and returns the results. This has benefits including:

- The client gets to fully express the full set of semantic actions they want to perform in the language instead of a series of separate API calls
- The language is part of the domain boundary because clients can't do anything that the language doesn't support. Making traditional API calls allows more extensive data extraction and manipulation without these limits.
- Only one API that accepts the language need be implemented[15]

As discussed in earlier sections, there are multiple levels of language when using dbt. The out-of-the-box, the addition of macros, and the addition of the semantic layer. Each can be directly exposed as a data API, e.g., expose an API that is just the dbt models serving as data products. The APIs can get richer as the implementation of the data mesh evolves eventually having an API that accepted the full DSL.

DSL Inside vs. Outside the Domain

It may be necessary to formalized two kinds of ubiquitous languages:

- the language used to do the work inside the domain boundary
- the language used by clients to interact with the domain

The language inside the domain can express operating on all the internal capabilities and data. External clients may be much more restricted in what they can access or do. When focused on the data mesh you are most likely to start with the client language, e.g., how to they interact with the data products.

Everyone wants Self-Service

Few have credibly attained *self-service* data processing and there is little agreement on how attain it: low-code/no-code, drag-and-drop UIs, AI/ML, Citizen Data Scientists, etc. I define self-service as the ability of the users to create *executable solutions* in or from the domain without the IT team doing a software development cycle. The *solution* can be as simple as getting access to existing data and using it to create new data or as elaborate as building a new application. With any of the dbt intermediate architectures described above in place, self-service is enabled for technically capable SMEs. With a full DSL in place we attain elaborate self-service for a much wider audience of SMEs. For example, a data analyst could:

- Define new data models inside the domain
- Use those domains to create a new data product to expose to other analysts
- Use the internal or data product models to define a new metric and expose that

If we don't attain these levels of self service we will never break out of the cycle of always being behind the business demands. Even more important, the right *solutions* will be built because the SMEs won't make mistakes on what to build they way it that so frequently happens in a standard IT software dev cycle. The Subject Matter First manifesto covers this in detail.

If we allow business users to build their own *solutions*, it needs to be done at level equivalent to an IT solution. This means real support for:

- Testing - before it can be used in production the analyst built solution needs to be tested. Dbt includes test automation and data quality checking as part of its language. Part of building a real DSL must include either including and integrating the dbt testing features or, ideally, having domain specific ways to test.
- Governance - before it can be moved to production impacts must be understood and managed, versioning must be supported, updates to metadata documentation must be done. Dbt includes a promotion process the supports moving new solution elements from dev to production, it supports versioning (and major extensions to versioning are coming soon), documentation is automatically produced.

A full DSL typically includes an integrated editing and testing tool, e.g., an IDE style tool that is specific to the DSL. This level of DSL support dramatically enhances the self service.

Summary

As we model the language of the domains of our data mesh we can move along the following path:

- Use an out-of-the-box generic DSL style tool like the basics of dbt.

- Expand to use more features of the tool, potentially in combination with other tools, e.g., use of the semantic layer language of dbt potentially in combination with another tool to do more advanced data quality checking, e.g, [Great-Expectations](TODO: get url)
- Introduce a DSL. Frequently the first DSL tends to be more technical
- Expand the DSL to be more targeted at the SMEs of the domain. (Ideally you'd skip the more technical and start your DSL work here.)
- Expand to more comprehensive DSLs covering different domains.

You need not create a unique DSL for every domain, especially in the early part of the journey. There are almost certainly common data structures and operations shared by domains and basic things like how you express your data-products that could be supported by a common DSL. For example every data product likely needs common ways to express:
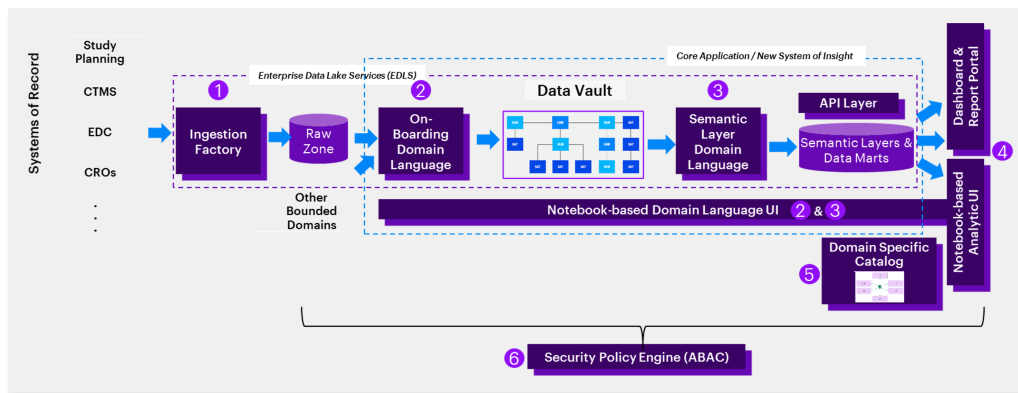
- Data quality constraints
- Retention
- Tests
- Access policies
- API characteristics

Appendix

#TODO: decide if this material is needed…

Full Architecture Example

#TODO: consider adding this or a variation on it as an example of a full architecture for a domain.

Key Elements to Deliver Value

1. Source data ingested into the Raw Zone

2. Data in the Raw Zone is on-boarded to the Clin-Ops data mesh node by mapping it into the Data Vault using a Domain Language. As part of the mapping, tests and data quality checks are also defined in the Domain Language.*

3. Data in the Vault is transformed and mapped, tested, and quality checked into analytic ready forms using a Domain Language.*

4. Data Visualizations, Dashboards and Data Science use the data

5. Data is found and understood via metadata available in the Catalog

6. Access at every point is controlled and tracked via centrally defined security policies.

* Steps 2 & 3 are performed via a Notebook-based Domain Language environment

Bounded Domains is Data Mesh++

Write about how Data Mesh is too narrow a name vs. Bounded Domains. It's not just about the data. It needs to cover all the systems not just the new ones on a cloud platform dedicated to data access and analytics. It needs to be an Enterprise Architecture pattern.

TODO: This is where we revisit the question about apps in the domain vs. just data mentioned earlier in the article. In the data mesh implementations I've seen, it's mostly been the apps as external data sources to a cloud-based data mesh. I feel we need to take a more whole-enterprise view of the bounded domains that makeup the data mesh so the apps should sometimes be inside.

1. See Data Mesh: Delivering Data-Driven Value at Scale or Starburst's Description of a Data Mesh and Accelerating Your Cloud Migration Journey with a Data Mesh Architecture ↵

2. This picture is is copied from the original article on data mesh: Data Mesh Principles and Logical Architecture. ↵

3. For more about the intersection of DDD and the language of domains see On the Relationship between Domain-Driven Design and Domain-Specific Languages ↵

4. Great places to start with DSLs are this community: Subject Matter First and the writtings of this master practitioner: the further reading list after this article or just google for anything written by Markus Voelter. ↵

5. This paragraph is taken from articles written by Markus Voelter. ↵

6. In the data mesh pictures, APIs with a 'D' are Data APIs. Those with an 'O' are operational or other types of APIs. ↵

7. See: https://docs.getdbt.com/docs/use-dbt-semantic-layer/dbt-semantic-layer ↵

8. See https://www.data-vault.co.uk/what-is-data-vault/ or google 'data vault' to see the massive amount of information available about it. ↵

9. We used an earlier version of it called dbt-vault. It may have become easier to configure since then. What's important is the example of hiding tecnical details from the SMEs by wrapping it with a DSL. An important benefit of a tool evolution like this, discussed later in this article, is that the DSL isolates you from the tech changes, see: TODO: write section on tech evolution. TODO: investigate whether it's more DSL in the new version. ↵

10. It's never a simple as just generating dbt config files but that is the majority of what is generated. Other things like database DDL, blocks of shell scripts or python code are also generated to fill gaps between tools like dbt. ↵

11. These examples are taken from a presentation by Clario at a DSL conference. ↵

12. Getting into DSL technology requires a separate article. Some great places to start are looking at [MPS](TODO: get URL) and [LionWeb](TODO: get URL) ↵

13. TODO: get a reference to the tax dsl ↵

14. TODO: Investigate why data-mesh calls these 'operational APIs. The APIs labeled as 'O' in the diagrams. Operational sounds like they are limited to just managing the domain vs. accessing the data via them. Do they consider the 'D' APIs to be both the database access to data-products via SQL and the http style access? ↵

15. Yes there will potentially need to be different APIs for different aspects or sub-sets of the language. I'm exaggerating for impact. ↵