hmd-docs-mdd-mda

Release 0.3

HMD Labs

Jan 24, 2024

CONTENTS

CHAPTER

ONE

ABOUT THIS BOOK

1.1 Who should read this, and why?

- Software, Data, and Analytics Engineers
- Software and Cloud Development Managers of all flavors
- System, Data, and Information Architects
- QA and Automation enthusiasts

1.2 About the Author(s)

Brian Greene is a guy with a beard who has spent a lot of time over the last 20+ years thinking about the productivity and quality of software development in Systems of connected systems.

- Brian's Linked In
- YouTube
- NeuronSphere

Alex Burgoon is Software/data/everything Engineer that has spent a lot of time over the last 15 years thinking about ecosystems of systems, writing a broad variety of software, and telling Brian which ideas are simply not rational, or feasible. Of those that are rational and feasible, Alex has created much of the proof.

- Alex's Linked In

1.3 Reviews and Status Tracking

For now each point release will trigger a release to review, with the intent that the sections marked as in review for that point release will not change until the reviews are largely in.

Thus, the NeuronSphere MindMap which also largely represents the outline of this book should be an up-to-date map related to the maturity of different sections.

1.4 Tracked Versions

Version	Rel Date	Description
0.2	2023-06-22	Primary structure in place and sending the first draft to first-round reviewers. The intent is to verify structure, tone, and delivery tooling.
0.3	2023-07-21	Vast updates to content, with additions in every section. Many updates on code generation and data modeling, as well as new microservice drawings. Lots of drawings, new version of plant-uml, and a new section on external use Updated the main mind map and introduction.

1.5 Software Behind This Book

This book is being managed as a NeuronSphere Compliant Repository with a large but non-customized /docs folder.

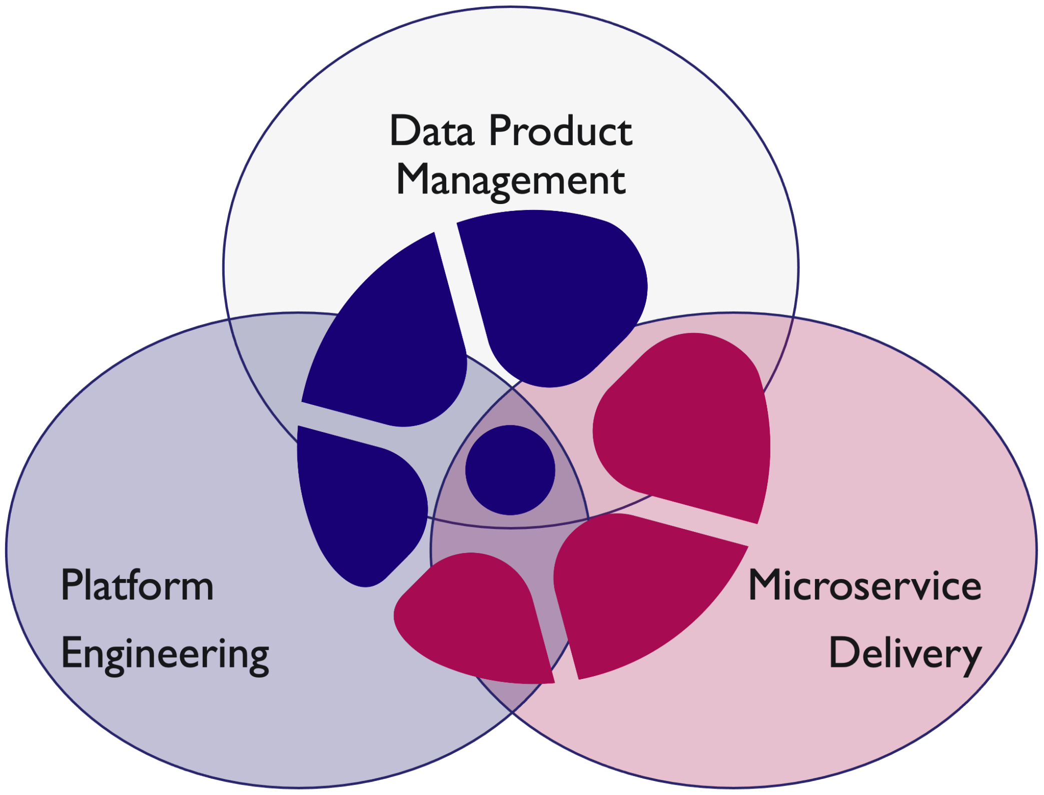
The intent is to show that from a simple document to even a book that will include references to document sections from other compliant repositories, this is an efficient toolchain for sophisticated delivery with little resource investment.

CHAPTER

TWO

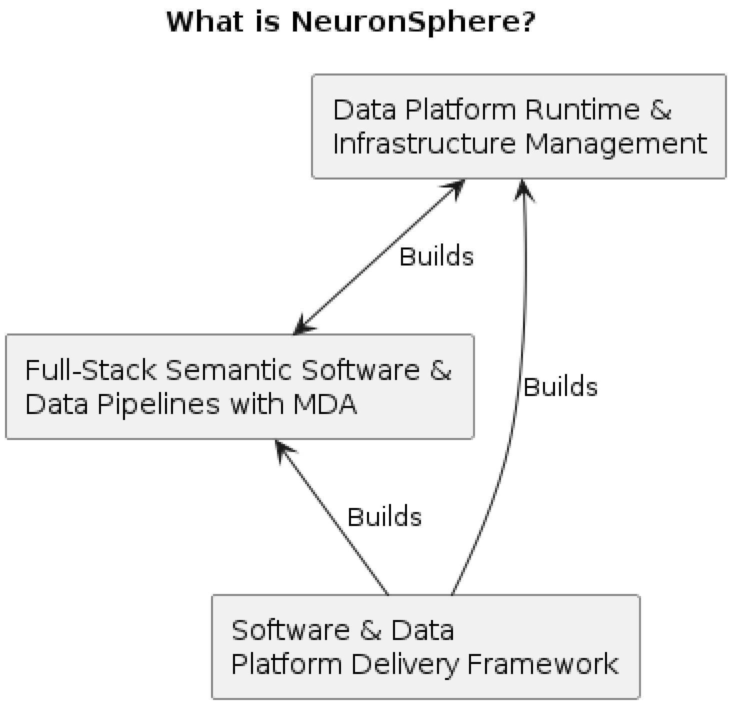
WHAT IS NEURONSPHERE?

Describing NeuronSphere is like the tale of 3 blindfolded men, each touching an elephant and giving their own different descriptions of what they've found.



Platform Engineering for [data]

2.1 The 3 Facets of NeuronSphere


What is NeuronSphere?
Data Platform Runtime & Infrastructure Management
Builds
Full-Stack Semantic Software & Data Pipelines with MDA
Builds
Builds
Software & Data Platform Delivery Framework

1. **Software & Data Platform Delivery Tools** - NeuronSphere is a collection of software and data development tools designed to make any team produce better software systems faster. It is highly opinionated in a number of areas, being designed to remove friction from the delivery of data engineering and analytical systems.
2. **Full-Stack Semantic Software & Data Pipelines with MDA** - Using a 2 tier graph & structure-based multi-layered metamodel as its basis, this key idea powers the services and code-generation frameworks that make NeuronSphere possible. This new way of managing the data estate approaches a PACELC consistency for highly-distributed systems.
3. **Data Platform Runtime & Infrastructure Management** - An end-to-end unified data infrastructure and data engineering platform. Broad capabilities across infrastructure, ingestion, cataloging, orchestration, data warehousing, information exploration, and data visualization. Extensible in profound ways.

NeuronSphere is the rare toolkit + platform that fully uses its own development tools and processes to build itself, and provides the tools with the product to allow complete extensibility and creativity.

What does this mean? You could start with the freely available (binary) NeuronSphere CLI tools, data modeling tools, and delivery philosophy, and completely rebuild the NeuronSphere Data Platform from scratch. Realistically many teams will use 50-90% of the supplied NeuronSphere capability and provide several plugins, *transforms*, and configurations to achieve their goals.

That 50-90% saves years.

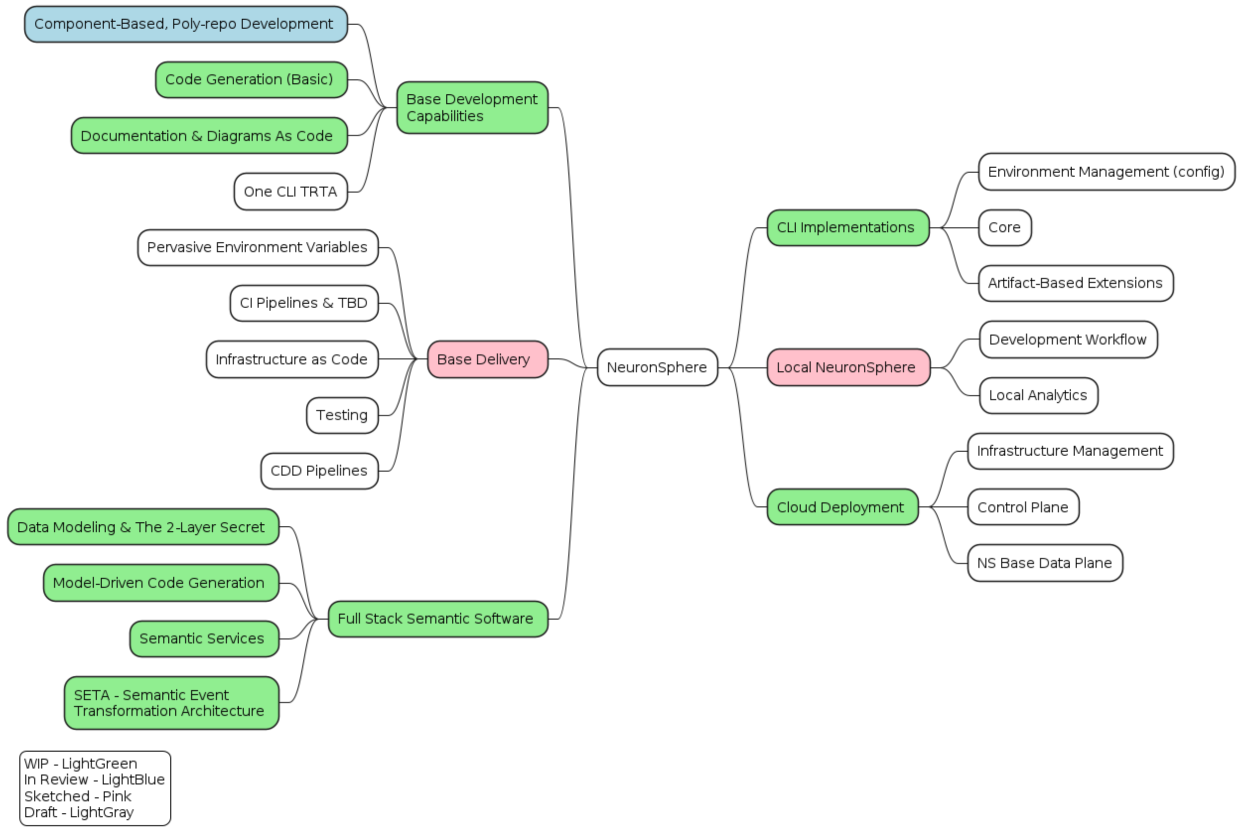
CHAPTER

THREE

A MAP OF NEURONSPHERE (AND THIS BOOK)

This work is divided into 6 main sections, further subdividied roughly by *idea* versus *implementation*.

1. *Ideas* for more effective software & platform creation & delivery
    - Base(d) Development - language & tech agnostic techniques for modern development
    - Base(d) Delivery - cloud-platform delivery MVP
    - Semantic Software -
2. *Implementations* from basic local development through enterprise-scaled multi-national deployments
    - CLI toolset & extension
    - Local Development & Analysis
    - Cloud Components & Deployment


Component-Based, Poly-repo Development
Code Generation (Basic)
Documentation & Diagrams As Code
One CLI TRTA
Base Development Capabilities
Pervasive Environment Variables
CI Pipelines & TBD
Infrastructure as Code
Testing
CDD Pipelines
Base Delivery
Data Modeling & The 2-Layer Secret
Model-Driven Code Generation
Semantic Services
SETA - Semantic Event Transformation Architecture
Full Stack Semantic Software
NeuronSphere
CLI Implementations
Environment Management (config)
Core
Artifact-Based Extensions
Local NeuronSphere
Development Workflow
Local Analytics
Cloud Deployment
Infrastructure Management
Control Plane
NS Base Data Plane
WIP - LightGreen
In Review - LightBlue
Sketched - Pink
Draft - LightGray

In the first half we'll try to keep the ideas above and outside of specific technical implementation, though we will allude to NeuronSphere's reference implementation as an example of some ideas.

Into implementation - a fair bit of the NeuronSphere CLI and local development tooling is freely available, and this will serve as a development and extension guide. Whether its documentation as code or packaging or testing, many existing teams and projects can take advantage of NeuronSphere development tools and techniques.

The last section will talk about NeuronSphere cloud services & data platform - discussion about NeuronSphere the software product that is sold and serviced by HMD Labs Inc.

CHAPTER

FOUR

ABOUT SOFTWARE - PERSPECTIVES & EXAMPLES

Software can be quite amorphous in its willingness to be described.

For all of the advancements in software and technology, it's still quite difficult from a variety of perspectives software and systems of systems are more complex than they've ever been.

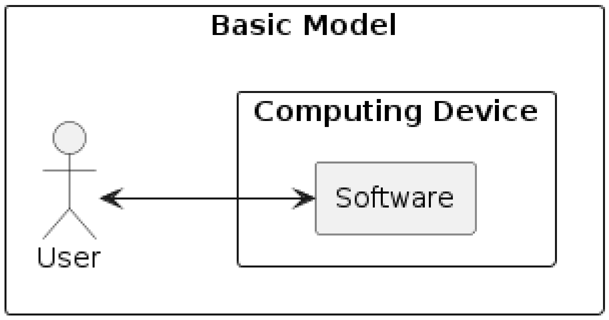
In order to talk about large-scale models for distributed software and data delivery, we first need to agree on a few fundamental models and *points in evolution* in system architecture. This is intended to be something of a whirlwind, and high-level.

TODO: About CLIs and APIs and Client(SDKs)

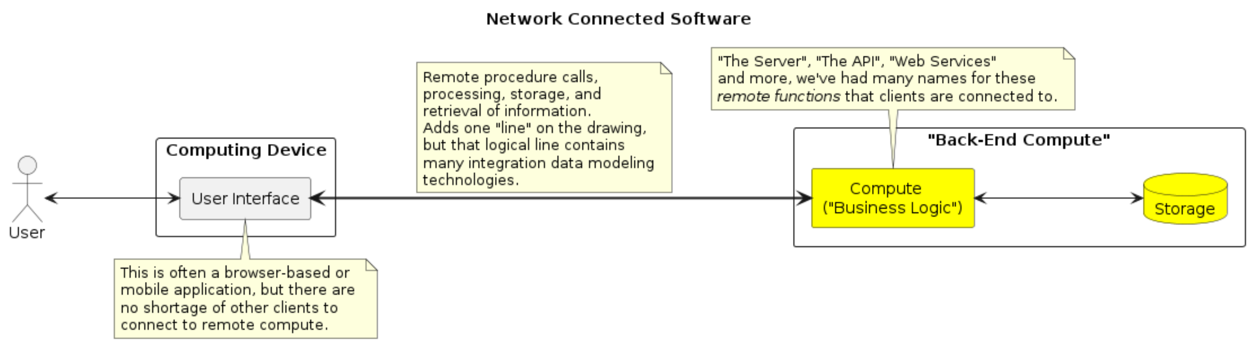
4.1 Simple Software

In the beginning, there was simple software. . .

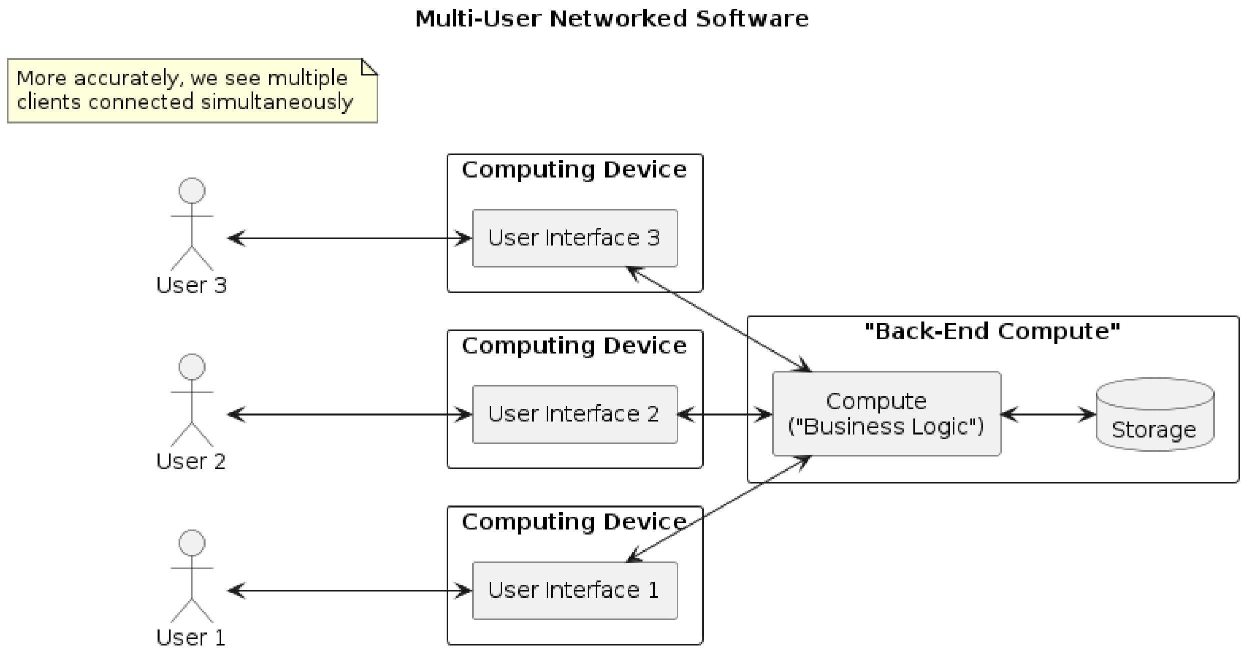
This is an overly simplistic view of software, but it's true that much of the software developed is as simple as this - running on some computing device, making some user marginally satisfied with the UX, not relying on any external resources. Within this view we will can and will decompose a bit, but our emphasis is not on this simple software.



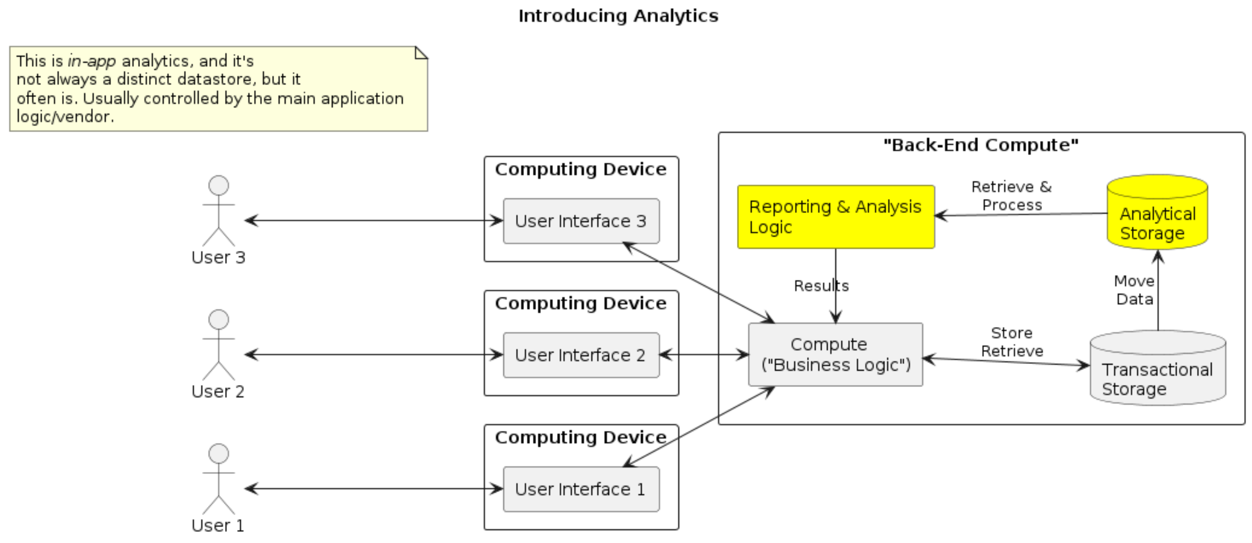
4.2 Networked Software


Network Connected Software
User
Computing Device
User Interface
Remote procedure calls, processing, storage, and retrieval of information. Adds one "line" on the drawing, but that logical line contains many integration data modeling technologies.
"The Server", "The API", "Web Services" and more, we've had many names for these *remote functions* that clients are connected to.
"Back-End Compute"
Compute ("Business Logic")
Storage
This is often a browser-based or mobile application, but there are no shortage of other clients to connect to remote compute.

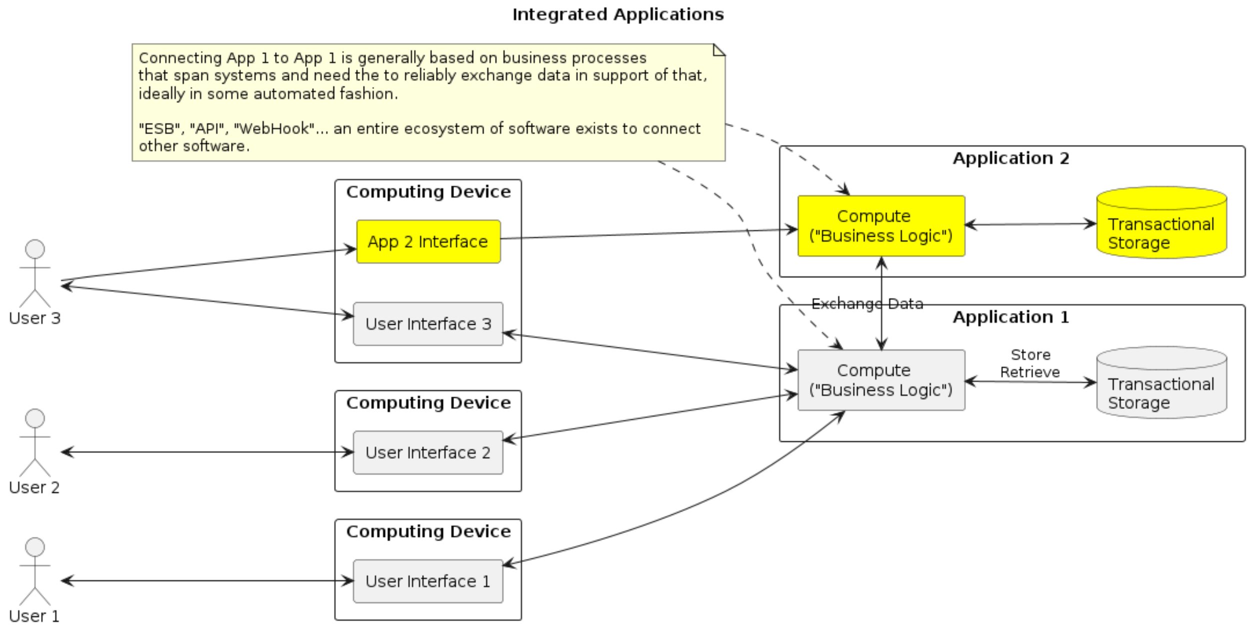
4.3 Multi-User Software



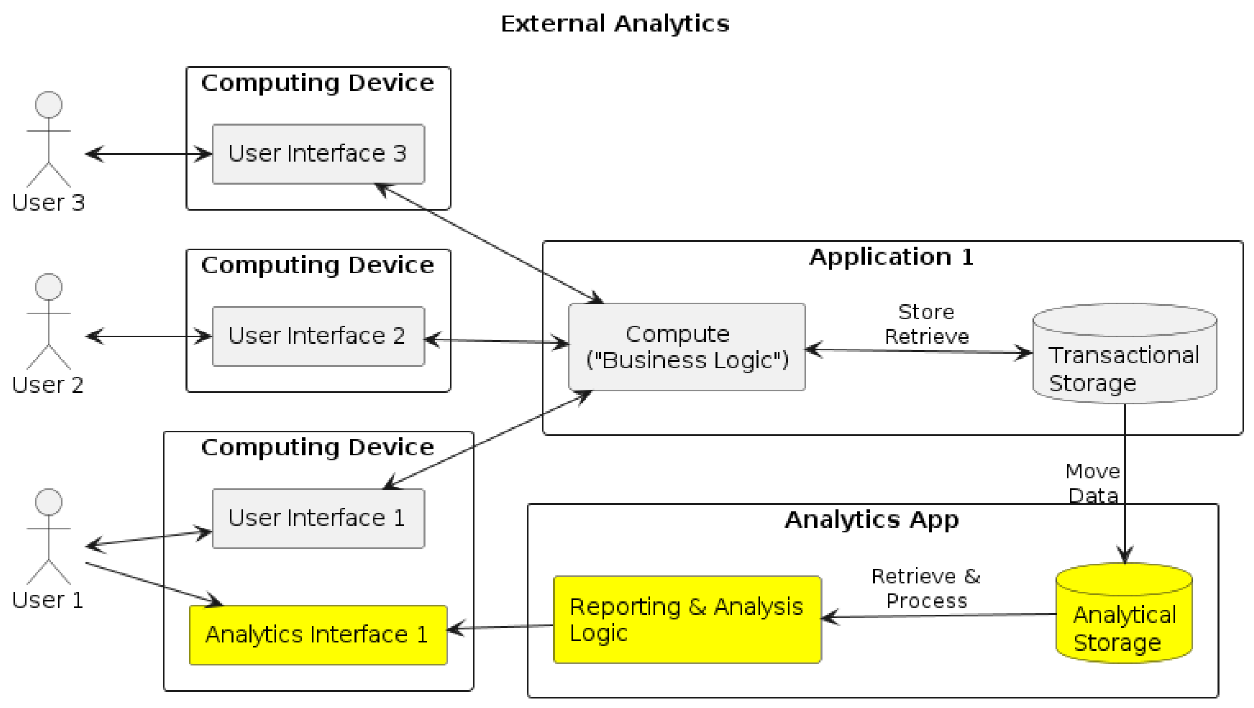
4.4 Introducing Analytics



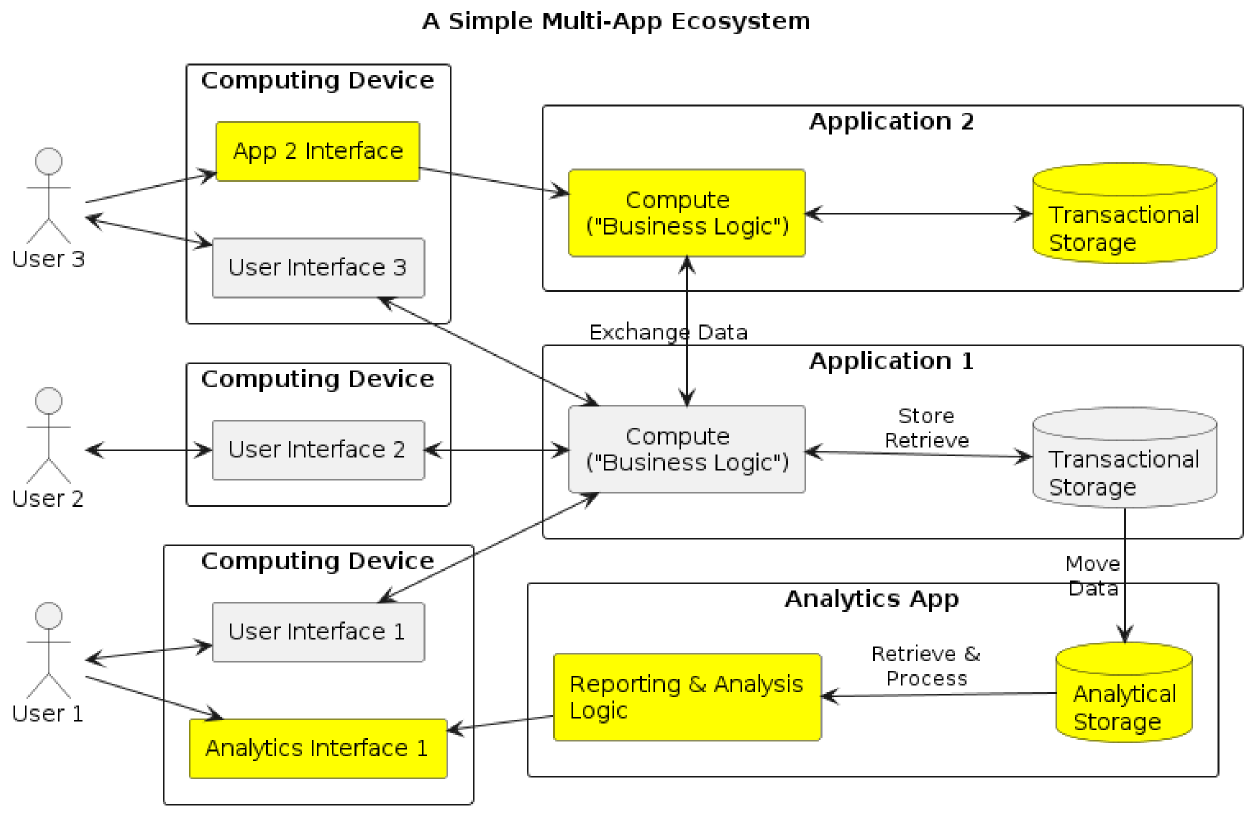
4.5 Integrating Another Application


Integrated Applications
Connecting App 1 to App 1 is generally based on business processes that span systems and need the to reliably exchange data in support of that, ideally in some automated fashion.
"ESB", "API", "WebHook"... an entire ecosystem of software exists to connect other software.
Computing Device
App 2 Interface
User Interface 3
User 3
Computing Device
User Interface 2
User 2
Computing Device
User Interface 1
User 1
Application 2
Compute ("Business Logic")
Transactional Storage
Exchange Data
Application 1
Compute ("Business Logic")
Store Retrieve
Transactional Storage

4.6 Using External Analytics



4.7 A Small App Ecosystem



4.8 Realistic Ecosystems

A picture and some statistics will be added here related to a *realistic* scenario for a medium-sized product mfg company.

CHAPTER

FIVE

BASE(D) SOFTWARE DEVELOPMENT

Regardless of language, team size, scope, or industry, this section defines basic capabilities that all software developers and delivery platforms can use as a baseline.

About Data Engineering and Data Modeling

This section does not appear to directly address these disciplines, however the techniques and ideas are not only applicable, but will be referenced throughout the rest of the book.

That said, I suppose it might be considered a bit boring and theoretical. Feel free to skip to more practical sections and see what you pick up - this section will be here later if you want the detail.

If you want to ship more software and deliver better data platforms more reliably, read on.

5.1 Component-Based Poly-repo Development

NeuronSphere SDLC puts forth the simple idea that by effecting a few simple constraints and conventions we can produce an exponential increase in software delivery velocity, quality, and developer happiness. These ideas are not simply optional but can form the bedrock for the delivery of a number of advanced technical models.

Foundational Development Practices & Characteristics for building software and data ecosystems

1. A collection of “small” and standardized set of source repositories
2. Managed by an extensible and inverted behavioral CLI framework
3. Using a consistent programming and architectural behavioral model

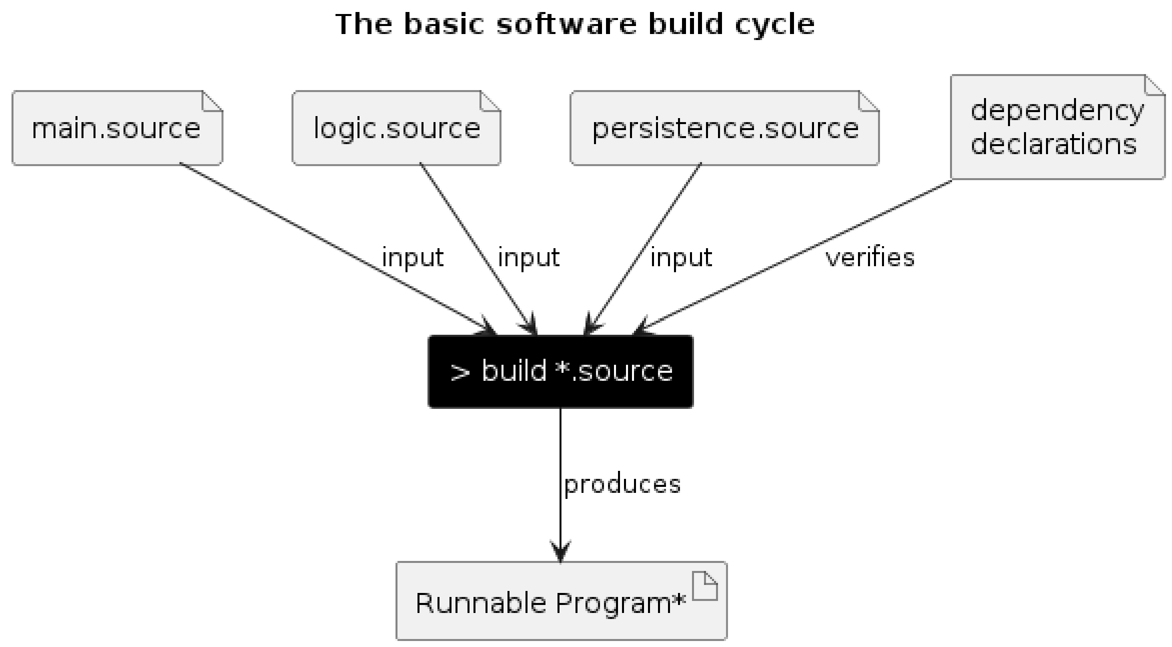
This is Occam’s Razor in software delivery - the simplest possible foundation that solves for the broadest number of outcomes.

5.1.1 Versioning, Packaging, Dependency & Artifact Management (VPDA)

All software is composed of a series of pieces, generally starting as a collection of text files and potentially other resources like icons and images.

A programmer enters syntactically correct commands as carefully formatted text, and then runs through the well-known cycle of build, run, package, publish (test[2]).

[2] We are intentionally leaving out the myriad testing steps in this process for now, but we will return to it in a dedicated section.



At this point, the developer will often exclaim “it works on my machine!”, and in fact it often does. Now we have to ensure that it will work in other environments, and that requires several underlying capabilities and ideas.

All but the simplest software will utilize external libraries to gain additional functionality[1]. From calling a database to rendering a template to reading a pdf to performing statistical analysis… there are libraries for everything, and the more software you build, the more you’ll realize that much of the job is learning to properly stitch together these amazing pre-existing capabilities into whole new things.

In fact, even “internal software” not meant for open source or even public distribution is often built of libraries and layers of components and technologies - we can’t escape dependency management and it’s related ideas.

Key Elements of Software Delivery Process & Ecosystem

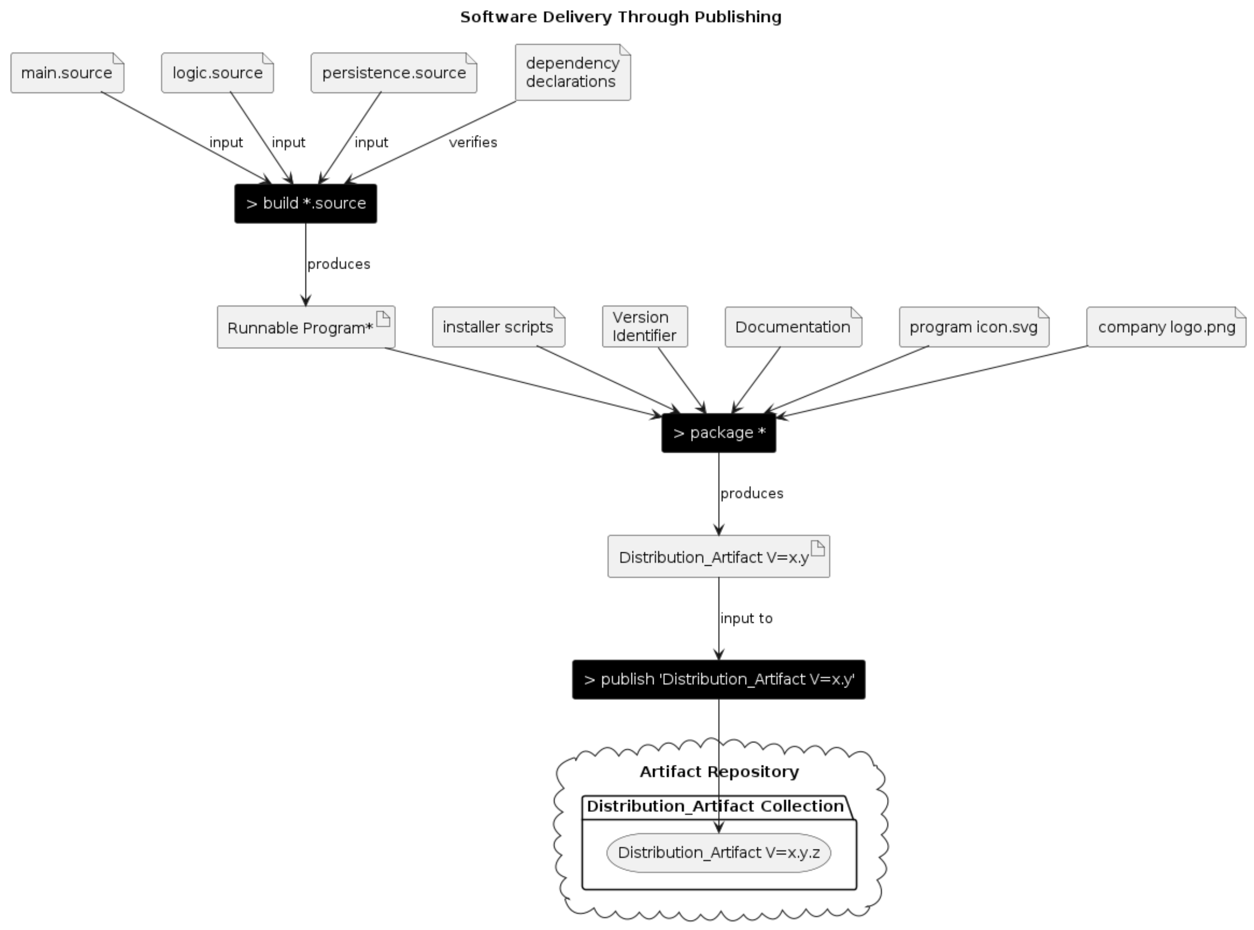
1. Version Identification - software modules should be identified by a unique version number. For something so seemingly simple, version numbering is a subject worthy of much study and debate. At a high level for discussion, we will often use a ‘simple semantic version + build number’ scheme, with a subtle hint of cal-ver later. This means that logically as we build and release iterations of our module, we will see versions like 0.1.12, 0.1.13, 0.1.14 for sequential builds, and 1.0.1, 1.0.2, 1.0.3 once we release a “1.0” release.
2. Packaging - Once software is compiled, it is often combined with other resources and uniquely labeled with a Version Identifier. Ideally this creates a single file, broadly ‘an artifact’ (or binary, image, package, wheel, gem, etc). Sometimes this packaging step will include special cryptographically significant steps to provide for various software attestation(s).[3]
3. Dependency Management - Configuration options and tooling to support declarative expression of other software or infrastructure modules at specific versions that must be ‘present’ in order to build or run a software module. Dependency or “package” managers control software installation and upgrade at various scopes, maintaining state as well as package caches.
4. Artifact Repositories - storage, management, serving, and security for outputs from packaging operations. Management of relationships and package metadata are key capabilities of artifact repositories.

[1] While there are highly successful pieces of software that are a single file, they are exceedingly rare, to the extent we can consider them non-existent.

[3] Coverage of ‘securing the supply chain’ and software toolchain attestation are outside the scope of this overview.

Public, private, and replicated repository infrastructure are also important features to ensure global adoption and usability.

Visually, this process generalizes as the following image:



This cycle and set of capabilities is the lowest common denominator for a programming language or technology ecosystem, and where a language doesn't provide them natively, communities and companies have built them.

Linux separates entire distributions based on implementations of these ideas, python has several competing implementations, and both Java and .Net have ecosystems they've stabilized around. Docker follows this workflow too, with millions of chained-together repositories and hierarchies of images providing a revolution in software of it's own.

Why do we go through all this? Regardless the size or language of a given package, software will express it's dependencies and the installers & packaging tools should prevent installation of software without the proper prerequisites. Often times this means that installing one package will 'drag along' a number of others that are required, or will force you to provide references/instantiations of your own prior to installation.

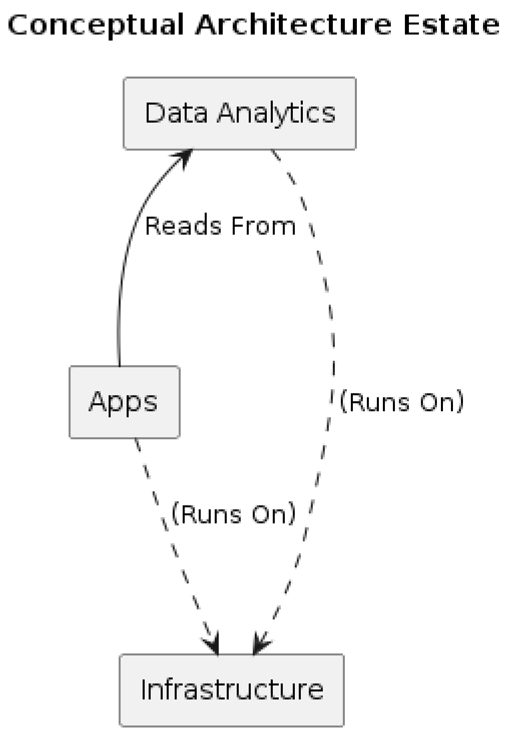
When done well these basic capabilities power massive ecosystems, and countless technology professionals are familiar with this flow and these processes. Unfortunately, these capabilities are often bound to a singular technology or language stack, with the occasional cross-over.

For all the power in these capabilities, they largely exist 'within the scope of the operating environment'. This is to say, python will do dependency management and resolution for python within the scope of a virtual environment or a machine. Java, .Net, Docker - each will do dependency management and resolution in their own space. This dependency management capability manifests well at the "software on an operating system" level, but is critically missing when considering delivery of a basic information ecosystem running on a combination of public cloud and SaaS/PaaS/IaaS providers.

5.1.2 Everything is Code - an incomplete movement?

Looking at a 30,000 foot view of a "normal" enterprise architecture, we often discuss the (extremely) simplified buckets of:

- Infrastructure
- Applications
- Data Analytics



For each of these broad categories we will look at (mis)alignment to managing full-stack-delivery like software.

Infrastructure as Code

Infrastructure as Code (IaC) is a common enough idea in computing today to be considered mainstream, but looking at the state of the art begins to show some profound differences and challenges. For all the value and pervasive use in 'declarative state-seeking' deployment technologies (e.g.: k8s/helm, terraform, cloudformation, etc.), little attention is paid to versioning, packaging, and distribution.

This seems to come from the idea that many organizations model "infrastructure" as a semi-static idea, a collection of "fixed and elastic" resources, managed often as a monolithic repository that mimics the speed of change in physical data center.

Many will exclaim "but we have GitOps to solve these problems", and GitOps as implemented in many places is part of the problem. It's really effective in concept, and you can get going quickly, and even operate at scale. Unfortunately, this often resolves to complex branching and folder structures in an attempt to use naming, folder, and branch conventions to make git sort of operate like a database for configuration files in an ever-growing footprint.

Application of modularity in design as well as consistent versioning, packaging, and dependency management provides a clean solution to this conundrum. Within this new delivery pattern we can also see how to more easily integrate multiple IaC tools into a single solution with clear release planning and dependency control.

Apps as Code

For the sake of this discussion, we'll say "Well of course apps are code!" Realistically "apps" is a vast category, spanning from N variations of JavaScript-on-browser technologies to cloud "Functions as Services" packaged as Docker containers, from Electron fat clients to collections of micro-services. All of these things have common traits in versioning, packaging, and distribution via repositories.

As well-established and powerful as these capabilities are in each of these technologies, there are a raft of teams suffering greatly because they skip them, opting instead for some attempt at a "mono repo", or a variation of semi-random-repos. In the latter case, we often get to see a great example of Conway's law in action, as allowing each team or function great creativity in source-control repository allocation and design is a sure spiral toward development hell.

Truthfully, it's rare that even small teams truly use a mono repo. Saying "we're building as a modular monolith" is fancy technical talk for "we don't want to solve a handful of basic problems related to versioning and library management, so we'll opt to stuff everything into one repository, and hope for the best!"

The truth is, mono repos allow you to take ugly shortcuts far too often, and lead to all manner of ever-increasing problems. Can we point to large software systems with lots of source code in the same place? Yes, there are few (all with fairly extreme characteristics). Can we point to a lot more small, medium, and large sized teams that are using a combination of "gitflow" and "monorepo" and having delayed delivery, lack of quality, inability to do CI/CD, claims that automated testing is hard, and low developer engagement? This is far more common.

Without having this turn into an exhausting discussion on the subject, let's dispatch the 3 most common objections to a component-based poly-repo approach:

1. Claim: We don't need to package and publish libraries if our internal apps are the only consumers. Response: Good software and system architecture will create natural boundaries around logically organized components. It is far easier to reason about a collection of version-numbered components in a configuration to deploy & test versus fiddling with multiple branches and merge conflicts. The truth is, the longer you delay the process of creating well-delineated components in your software, the more likely you are to 'cheat' with imports, create circular dependencies, and produce brittle systems.
2. Claim: We're making a 'modular monolith' and keeping things cleanly separated, so we *can* break it into modules later if/when we want or need. Response: If there are 5 developers (data - app - infra) all working, is it easier to reason about work coordination between myriad files in a single non-homogenous directory structure, or a set of modules with version-enforced dependencies between them? If we want to roll-back a change in a subsystem due to a bug, is it as easy as changing the version number in a dependency file, or do we have to do source-code surgery with branches and hope to back out the right set of changes? If it's so easy to break apart later, why not break it apart now? Is it because you want to cheat a bit (see point 1), or is it because the system has little intentional design?
3. Claim: We often need to make changes that have to ripple through multiple layers and changing this in multiple repositories with their own pipelines is annoying/time-consuming. The normal example for this is things like adding attributes to an existing data message being passed between the user/app/persistence layers. Response: a) This might be a code smell. At the least it's an opportunity to talk about using model-driven code generation to manage multiple versions of the same domain objects. b) if this is about changing a library and a consumer in relatively tight synchronization, there is nothing preventing you from building the library locally and using that version from your local consumer. Yes, you still need to manage changes between the 2, and publish them 'in order', but if you're making non-breaking, forward-compatible changes between modules this is practically not a challenge.

Analytics and Integration as Code

This exists in a state of complete naivete for many/most tools in the space, and it's one of the largest headwinds to reliable data platform delivery in the industry.

Q: "Why is DevOps for data tools so hard?" A: "Because the vendors don't care about you"

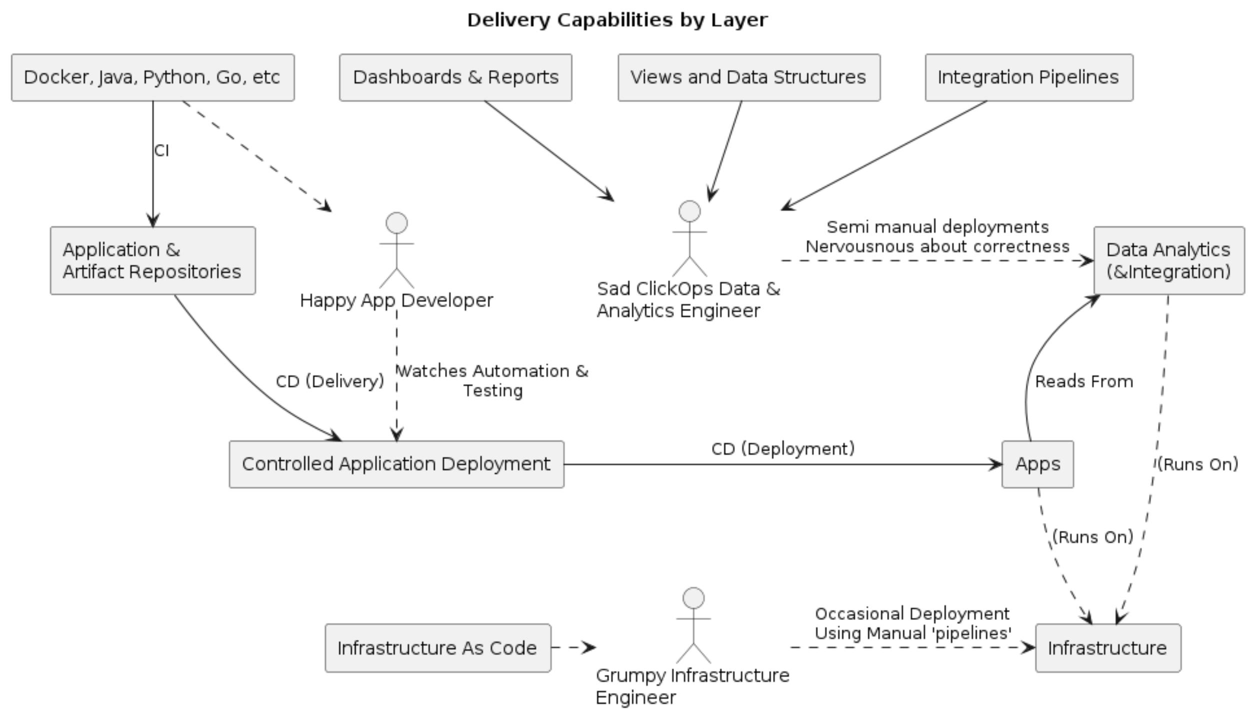
So perhaps it's not that extreme, but the data integration and analytics tools ecosystem has grown dramatically over the last 20 years, and yet often these tools leave "deployment" as a complex exercise for the reader. Forget "Analytics as Code", in many cases it is often a challenge to get 'code' out of point-and-click driven analytics tools.

Defining a repeatable and consistent process to 'on-board' new data tooling such that it can properly participate in versioning, dependency management, and repository-based distribution is a key process for a platform team focused on data, and pays huge dividends in the ability to automate and manage the data estate the same way we manage the software estate.

Data teams often push the envelope with a desire to experiment - new schemas, new databases, new transformation jobs and engines - experience has shown that collections of templated modules in componentized repositories can accelerate all of these activities.

The sad state of non-unified deployment and delivery

Looking at the 3 very basic layers of "Data, Applications, Infrastructure" from above, and adding in a few actors, and we start to see why releasing predictably is so hard.


Delivery Capabilities by Layer
Docker, Java, Python, Go, etc
Dashboards & Reports
Views and Data Structures
Integration Pipelines
CI
Application & Artifact Repositories
Happy App Developer
Sad ClickOps Data & Analytics Engineer
Semi manual deployments Nervousnous about correctness
Data Analytics (&Integration)
CD (Delivery)
Watches Automation & Testing
Reads From
Controlled Application Deployment
CD (Deployment)
Apps
(Runs On)
(Runs On)
Infrastructure As Code
Grumpy Infrastructure Engineer
Occasional Deployment Using Manual 'pipelines'
Infrastructure

The misalignment between these major categories is obvious and many readers will find it familiar. We have "infrastructure as code", but in many ways we don't manage and deploy it with the same concepts, tooling, and lifecycle. We have numerous data integration and analytics tools, but constructs around versioning their artifacts and rationally controlling their dependency graph is generally a manual exercise.

Logically, we'd like to say "Dashboard collection X depends on BI schema Y, which depends on Extraction Job 1, both of which depend on database instance foo" in a nice, clean graph of modules. Usually, we have a lot of meetings, code reviews, and sometimes prayers.

5.1.3 Minimum Viable Repository Model - 1.0

Having established that multiple software languages and technology ecosystems thrive because of versioning etc, what is the minimum basic repository structure we could use to create a more pervasive framework?

```
.
├── basic.txt
├── meta-data
│   ├── VERSION
│   └── manifest.json
├── src
└── test

4 directories, 3 files
```

In this MVP repository structure, we will set a few basic standards that we'll use as a baseline for all future repositories. in the meta-data directory, we find a VERSION file, a simple text file that expresses the logical version of the repository.

The immediate implication is that any script that interacts with this repository to build artifacts can (and should) use the 'content' of this version file in their activities.

Generally this follows the pattern "for a given technology x, where we have script(s) that perform build/packaging/publishing, and in each of these scripts we will consistently 'use' this version file and whatever other conventions for build/labeling we determine as an enterprise". We will elaborate on these ideas shortly.

Note: An enterprise in this scenario can be 1 or 5 developers, or 100. One of the keys to having things work as a scaled and integrated ecosystem is to have version numbering be as broadly consistent as possible (and it's fairly easy to be quite effective here).

You always have an enterprise architecture, the only difference is some teams are intentional and some feel like it's happening to them.

Thus if you have a script that builds Python wheels, it should use this version file (plus whatever build/labeling strategy you have) to create consistently versioned packages. If there is a docker image in the same repository, scripts that build/tag/publish docker images can (and should) use this version file (plus whatever build/labeling strategy you have).

We will also include the manifest.json file, which will be used to express dependencies on other repositories that follow this convention. (for the Maven savvy reader, the manifest.json is similar to the pom)

Technology Facets and Domains

Earlier we talked about software being built of a number of text files and other ingredients, but we've also paid more attention to versions and dependencies and looked at Artifacts that are the result of the build and packaging processes.

Commonly, these artifacts (aka libraries, binaries, images, etc) have an 'intended use' that transcends their existence as a file. For example, a python library may contain a set of utility functions for mathematics and provide no command line interface. Another .Net binary may be executable from the console and call a c++ library to perform it's work. Yet another java library may be code that's designed to work with configuration files from a particular persistence framework.

As you consider the development work before you, it's worth thinking about the 'kinds of things you'll make' with software.

TODO: what makes a good repo? *Deployable unit of common change*.

5.1.4 Standard Repository Behaviors & Inversion of Control

The idea that “all repositories should have reasonable responses to a consistent set of lifecycle commands” is not new, but like data modeling it’s an idea that bears resurrection and application to modern technologies. For example Apache Maven powers the build, test, packaging, and publishing lifecycle for millions of builds a day globally. With a 1.0 release in 2004 and going strong today, evidence of the power of these ideas is compelling.

Similarly Make has seen a great resurgence in the last decade (as teams had nothing else), yet like any ad-hoc build scripting system.

What we need to do is apply these ideas more universally - specifically by applying these ideas to the Data and Infrastructure layers in a way that’s harmonious with the existing application ecosystems and designed to power the future of data-centric ecosystems.

A [data|software|infrastructure] Engineer’s daily work

While the tasks of a developer are myriad, the lifecycle of most work is simplified as such:

1. Get task to change system (bug fix, change feature, new feature/module)
2. Find code and subsystem(s) in question
3. Figure out how to build it, test it, and execute it in various levels of isolation
4. Make changes
5. Build, test, and repeat until believed complete
6. Publish Changes
7. Test Deployment of changes

Small things are easier to reason about than large things. This is especially true in software systems, where there are numerous layers of complex abstractions working together to produce running solutions.

A source repository with 500 source files in it spanning a complete application stack will certainly work, but it is far more complex to reason about than 10 repositories with 50 files in each with clear dependencies and interfaces between them. Modules with well-defined interfaces are easier to test, and downstream processes related to risk assessment and impact analysis are greatly simplified.

As the team grows, org structures change, new technologies are introduced and others fall out of favor, it is useful to consider the long-term structure of your logical solution architecture and it’s relationship to your source code and multiple downstream delivery processes and capabilities in the CI/CDD delivery chain. As a key asset, it should stand the test of time, growing and evolving predictably and controllably.

Small things are easier to test and document.

Standard Repository Behaviors - (Repos as Cattle not Pets)

Considering the daily work of an engineer in a ‘software-defined-everything’ ecosystem, we see a general need for the lifecycle commands build[test], package, publish, deploy, int-test, release.

What does it look like for a “repository to have a behavior” - to a developer, it starts at the command line, and the basic idea that for any given repository I should be able to execute these commands from the cli and expect generally consistent behavior.

This is “poly-morphic” behavior in that “build” as a command in a given repo may produce several different outcomes based on the combination of *facets* and *technologies* in the repository, but they should all be ‘default and expected’.

One of the primary goals & outcomes of the NeuronSphere approach to repository structure is that we ‘normalize’ these behaviours across *Facets* and *Technologies*.

1. build - compile/link/transpose the source code into an executable format. For any number of technologies, this is a 'no-op' in that it doesn't effectively do much (and that's ok).
   1. Code Generation - code generation based on metadata/models is a prerequisite to compilation, and for many technologies is the only thing that happens during 'build'. See Also [CodeGeneration in lifecycle]
   2. UTest per lang/tech, local int-test(s)
   3. Build Docs - by default, documentation should be built after the code generation and successful compilation. Documentation building may happen at other phases depending on what gets added to the bundle. See Also [DocsAsCode]
2. package - adds images, installers, object files, etc into a 'single' deliverable & deployable artifact.
   1. Considerations for local packaging
3. publish - take packaged artifacts in the current filesystem/repository, the result of "package" command, and move them to a 'remote' artifact repository. This is the point where a build is available to other builds.
   1. Doc Publishing - See the section on Modern Documentation as Code
4. deploy <artifact==v> <env> <config==v> - Takes a packaged artifact and attempts to apply it as a change to a given environment with a given configuration. Can use a 'locally' packaged artifact to do 'in-line continuous deployment', or often uses a remote artifact reference.
5. int-test <artifact==v> <env> <test_config==v> - Looks in a given artifact for integration tests, and if available executes them against a given environment with a given test configuration. Integration tests should be optimized to run against the same locally-deployed and remotely-deployed artifact version without change.
   1. Test Artifacts
   2. Test Analytics (advanced)
6. release <artifact==v>

Writing scripts that perform these actions for common technologies used by a team, when combined with the MVP repository structure begins to produce exponential results.

Note: About Implementation - the intent is that these ideas, as presented, could be implemented in any number of languages or techniques.

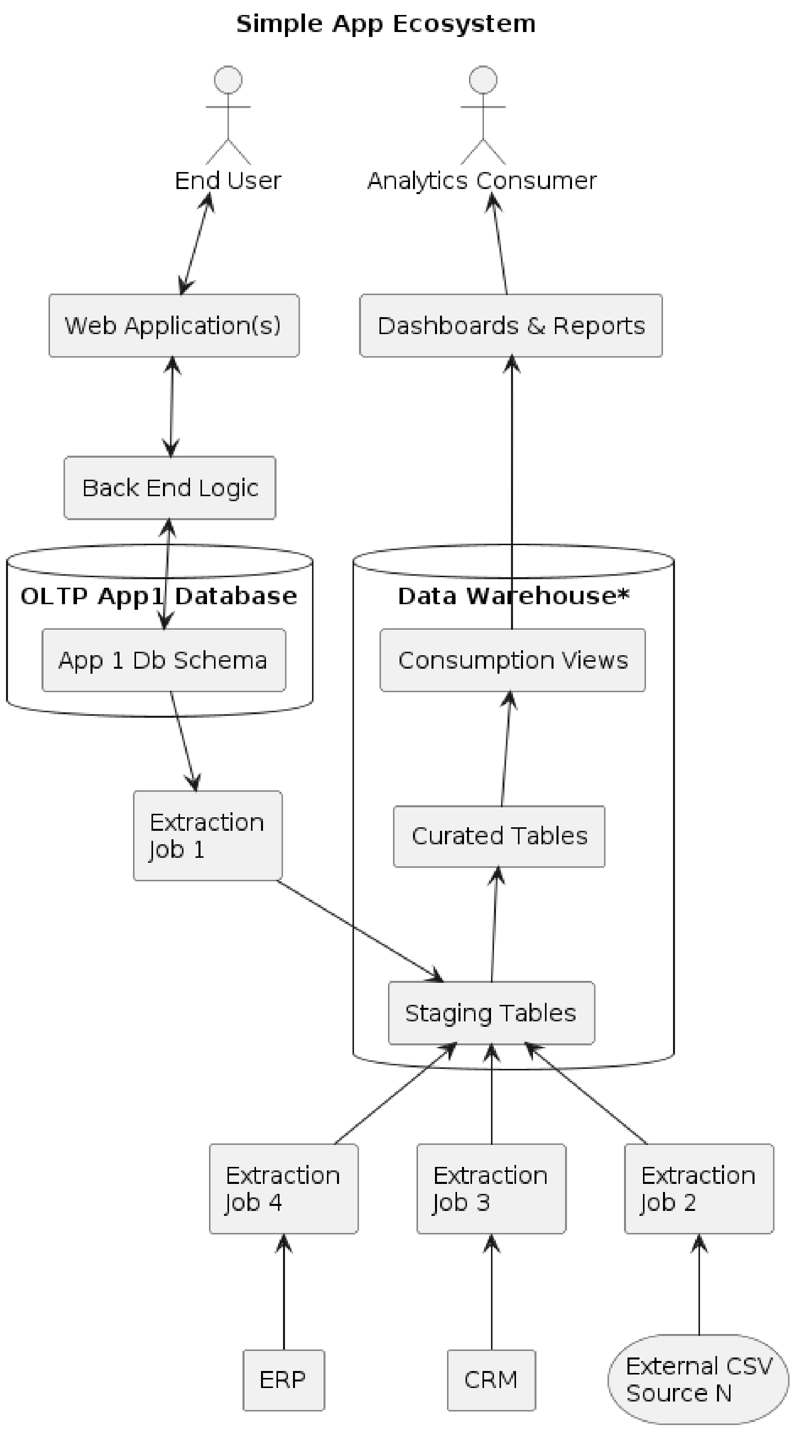
For instance, these 'multi-repository' commands could easily be implemented in bash or powershell.

NeuronSphere provides one implementation (python centric), but several of these capabilities and ideas are excellent exercises for platform engineers and can be easily piecemeal re-implemented or extended.

If these ideas and the initial implementation are sound, they should be the most effective way to create a new language and ecosystem-centric extension of NeuronSphere. Said another way, porting sections of the NeuronSphere architecture or toolchain to .Net, Java, Go, etc will be most effectively done using the (current) base Python implementation.

5.1.5 An Example

Being convinced that a poly-repo approach using a unified versioning and dependency management system is appropriate, we will look at an example implementation. Taking our simplified 3 layers above, we will design a more detailed yet simplified "typical" application architecture.

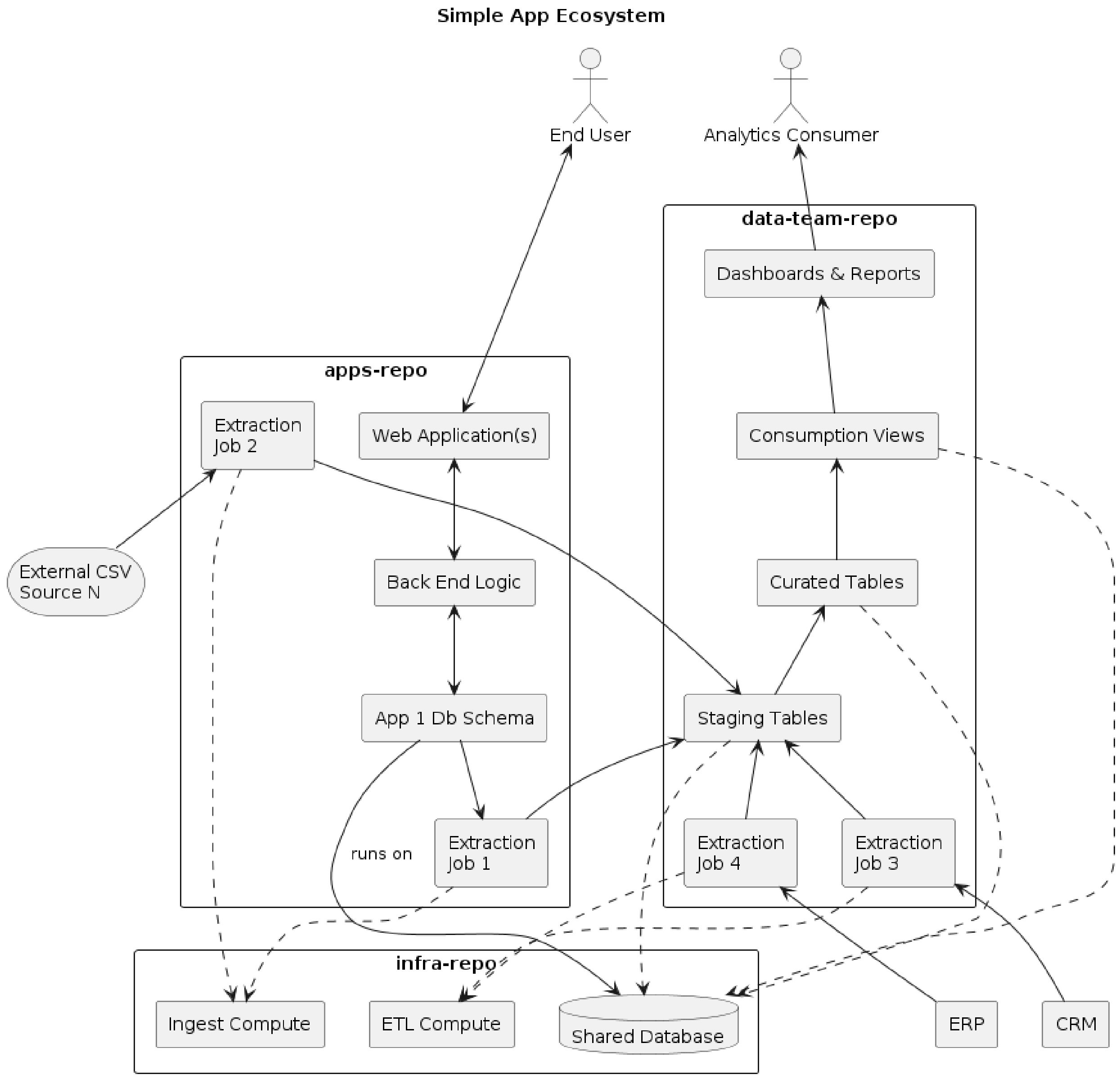


This drawing is intentionally at a level "just above" implementation details, and could be materialized and implemented in any number of ways. Note this is a logical architecture, and we should start here rather than jumping immediately to a drawing that's a collection of specific technical icons and technologies, as this hides the architecture by focusing on the detailed technology end-state.

When presented with this simple application architecture, we often move to create a source code repository or more, but let's explore how the architecture maps logically onto a set of modules, focused on 'change radius', dependency, and deployability. Since this isn't a text on system architecture design choices, we'll make *very* plain and straightforward implementation decisions, focusing instead on building a firm conceptual foundation and example.

A pseudo-mono-ish repo strategy - typical

Variations on this theme will exist, but often we'll see repos 'as large as possible' and each one quite unique in how it works. This is often great evidence of Conway's law, wherein <team> == <technical capability> == <repository> is the "design". This has some benefits but overall is substandard.



This is kind of gross, but it's easy to accidentally get here and then have a hard time seeing "what's next".

Explicitly - this is the kind of 'least-efficient' code organization we are trying to avoid.

1. The repos don't have any consistent behaviors. As a developer, qa person, ops staff, etc... there is no consistent expectation I can place on a collection of code - each one is a 'choose-your-own-adventure

where the clock is ticking and you just hope the 'self-organizing' group that's been nursing this pile has left decent make files?

2. Deterministic enumeration of our components and their interactions is difficult or impossible to achieve across the asset landscape. (All quality and security policies start with the ability to deterministically enumerate your software architecture)
3. As a manager, it is difficult to keep a pulse on which portions of the system are working, in-flight, suffering, etc.
4. Code/Feature release will be slow, with lots of meetings, reviews, and "double checks" in code merges and dependencies that people know about, but that systems don't well enforce.

Note: Where is the documentation? If one were to attempt to do *documentation as code* in this environment, how would you track it, build it, etc? This is left as a difficult exercise for the reader later in this 'layout strategy'

To show a *very* simplified example of this after checking out the repositories, you might find this:

```
.
├── data-apps
│   ├── README.md
│   ├── dashboards
│   │   ├── main_car_dashboard_new.xml
│   │   ├── main_car_dashboard_old.xml
│   │   ├── wheel_data_prod.xml
│   │   └── wheel_experiments_kate.xml
│   ├── dbt
│   │   ├── consumption_views
│   │   │   ├── README.md
│   │   │   ├── analysis
│   │   │   ├── data
│   │   │   ├── dbt_project.yml
│   │   │   ├── macros
│   │   │   ├── models
│   │   │   │   └── example
│   │   │   │       ├── my_first_dbt_model.sql
│   │   │   │       ├── my_second_dbt_model.sql
│   │   │   │       └── schema.yml
│   │   │   ├── snapshots
│   │   │   └── tests
│   │   └── new_wheel_models
│   │       ├── README.md
│   │       ├── analysis
│   │       ├── data
│   │       ├── dbt_project.yml
│   │       ├── macros
│   │       ├── models
│   │       │   └── example
│   │       │       ├── my_first_dbt_model.sql
│   │       │       ├── my_second_dbt_model.sql
│   │       │       └── schema.yml
│   │       ├── snapshots
│   │       └── tests
│   ├── scripts
│   │   └── load_old_data.py
```

(continues on next page)

(continued from previous page)

```
│   └── sql
│       ├── car_dashboard_exploration.sql
│       ├── wheel_DQ_WIP.sql
│       └── wheel_creates.sql
├── infrastructure-cloud-new
│   ├── Docker_buildimage.docker
│   ├── check_drift.sh
│   ├── deploy.sh
│   ├── ec2_module.tf
│   ├── main.tf
│   ├── modules
│   │   └── s3_module_old.tf
│   ├── network
│   │   └── networks.tf
│   ├── scripts
│   │   └── new_branch.sh
│   ├── terraform
│   └── variables
│       ├── dev
│       ├── prod
│       └── staging
└── transport-applications
    ├── Docker_Local.docker
    ├── MAKE
    ├── api_local.sh
    ├── docs
    │   ├── Readme.md
    │   └── installation.txt
    ├── libs
    ├── localdebug.sh
    ├── main.go
    ├── persistence_util.go
    ├── tests
    └── web
        ├── index.ts
        └── modules

37 directories, 36 files
```

On one hand, this *incredibly* simplified group of directories might have enough decent naming convention that you can, for any small piece of it, understand it. You can even look at all of it and see how it fits together, sort of, if you know what you're looking at.

Here's what we don't know:

- What version of anything is any of this? That's hidden on the build server?

5.1.6 Component-based intentional poly-repo decomposition

Combining *VPDA* and standardized repositories, we get a new decomposition of the solution, a better decomposition. What makes a "good" repository in this methodology?

Each repository is at least one (and often only one) logical component, which is roughly a *facet* created from a *technology*. This allows you to create rules and guidelines for how components made of different kinds of facets will interact and be packaged together.

1. Major Domain & Technical Capability Dependency
2. Likelihood or desirability of a synchronized release cycle
3. Deployability

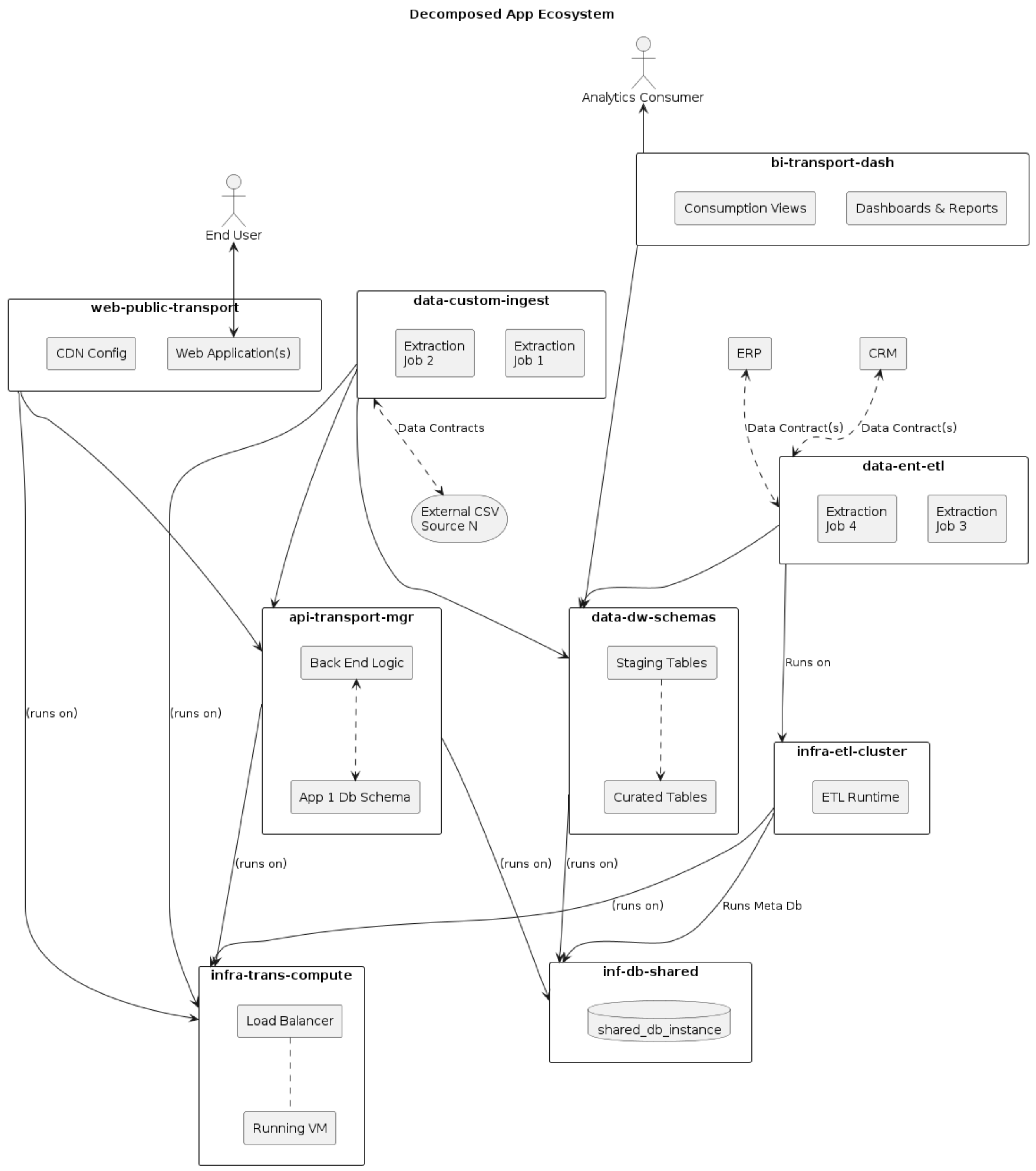
For instance, a repository with a 'sql_transforms' facet will likely have a dependency on at least a database that it runs in, and perhaps other packages that declare DDL. A collection of dashboards may depend on the sql_transforms component.

Another repository might be a java CLI (cli facet in java) with a dependency on a library (library facet in java) defined in another repository and a docker image that must be deployed to specific docker repo for use.

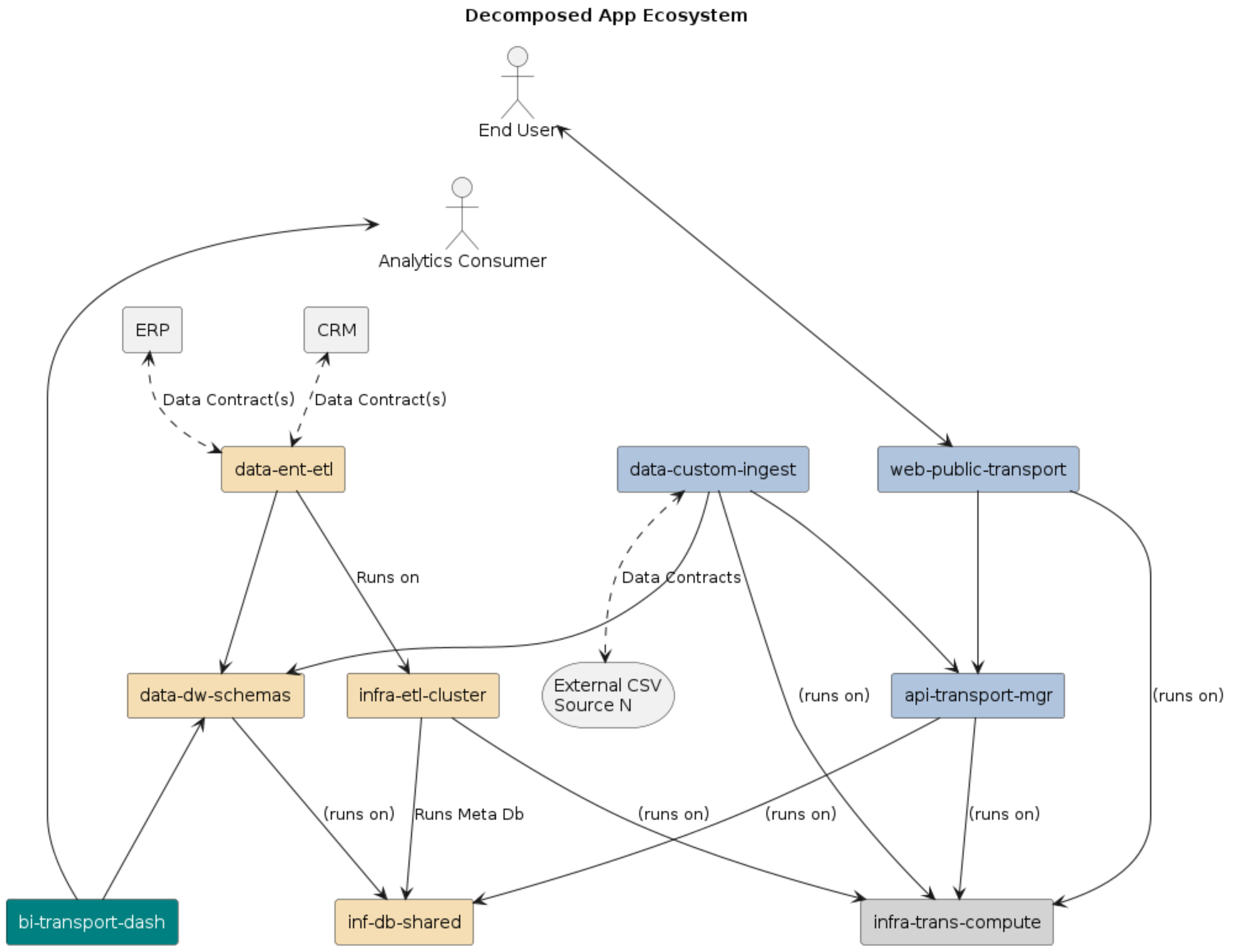
Perhaps we have a series of docker images designed to run in Argo workflows - these would be 'argo workflow images' facet of the docker technology type.

We have an ETL server that needs to run somewhere, and on it a few import jobs are defined and we need to manage those dependencies on data contracts with external teams.

This creates a set of repositories as such:



Simplified, we see:



Looking at this as a series of standard directories, we see a much easier to reason about picture:

```
.
├── dmo-api-trans-mgr
│   ├── README.md
│   ├── docs
│   │   ├── api_template.rst
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   ├── openapi
│   │   └── python
│   │       └── dmo_api.py
│   └── test
│       ├── api_test_template.robot
│       └── example_config.json
├── dmo-bi-transport-dash
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
```

(continues on next page)

(continued from previous page)

```
│   │   └── manifest.json
│   ├── src
│   │   └── python
│   │       └── dmo_api.py
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-data-custom-ingest
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   └── argo_ingest
│   │       ├── extract_1.docker
│   │       └── extract_2.docker
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-data-dw-schema
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   └── python
│   │       └── dmo_api.py
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-data-ent-etl
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-inf-db-shared
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   ├── argo_ingest
```

(continues on next page)

(continued from previous page)

```
│   │   ├── python
│   │   │   └── dmo_api.py
│   │   └── terraform
│   │       ├── main.tf
│   │       └── rds.tf
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-inf-etl-cluster
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   └── terraform
│   │       └── main.tf
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
├── dmo-inf-trans-compute
│   ├── README.md
│   ├── docs
│   │   └── index.rst
│   ├── meta-data
│   │   ├── VERSION
│   │   └── manifest.json
│   ├── src
│   │   └── terraform
│   │       ├── ec2.tf
│   │       └── main.tf
│   └── test
│       ├── base_test_template.robot
│       └── example_config.json
└── dmo-web-pub-trans
    ├── README.md
    ├── docs
    │   └── index.rst
    ├── meta-data
    │   ├── VERSION
    │   └── manifest.json
    ├── src
    │   └── typescript_web
    │       └── index.ts
    └── test
        ├── base_test_template.robot
        └── example_config.json

57 directories, 67 files
```

5.1.7 Standard Layouts, Facet proto-types & Composable Templates

What's a critical mass worth?

5.1.8 Component Metadata

Given that we're starting to really 'architect with the repositories', and we have a common manifest file that we're using in all repositories, perhaps that's where we should track other component metadata?

Yes, and we'll use it to not only track repository compliance but also transparently extract and track this metadata in standardized CI/CDD pipelines.

TODO: examples of tracking metadata and use cases?

Inversion of control for tooling

Each repository having a .config file for each of the N tools we wish to use in the CI chain is an antipattern.

Having each repository share a common configuration that applies the appropriate CI tooling based on repository facets automatically is the way to victory.

TODO: there needs to be an example here.

5.1.9 Will this scale?

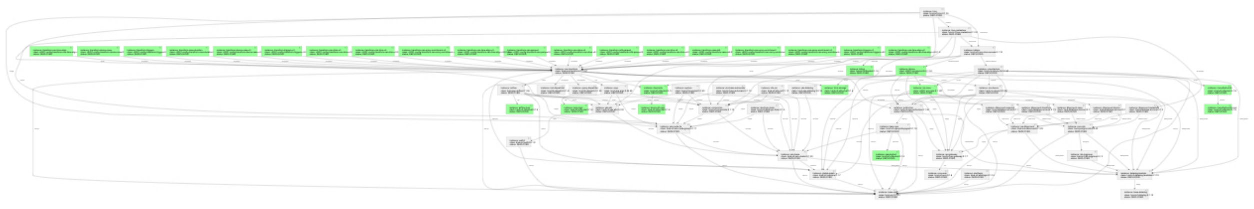
"Scale" is a pervasive idea in software delivery, and it is routinely measured in multiple dimensions.

For the sake of this discussion, we are not talking about the scalability of the software itself (transactions per/period, data volumes per day, users per hour), rather a potentially more important part of scaling software delivery - that of scaling a team in various dimensions.

1. Can this proposal help make smaller teams deliver more complex software?
2. Does this technique accelerate onboarding new developers throughout the stack?
3. Will we be able to experiment and iterate on new technologies more quickly and safely?
4. Does this make it easier to develop 'true platform technology'?

The answer to all of these questions is 'yes', and the results are predicated on the simple ideas presented here.

This image is a "pipeline deployment" of the NeuronSphere data platform. This is from a previous customer deployment, and shows the component dependency graph for their deployed production instance. As an indicator of the "platform versus end-usage", the modules highlighted in green are "customer-specific modules" built on top of the NeuronSphere platform, showing broad reuse from infrastructure up through multiple layers of application and data platform.



Note: This is the entire deployment, representing everything from object ingestion and VPC configuration through airflow dag generation and management, coordination of the Argo workflow engine, orchestration and management of dynamic k8s clusters and their underlying infrastructure, and then coordination of dynamic workloads for trino, spark, and other engines that can be plugged into the cluster manager. This is not an exhaustive list of “what’s in the box” with NeuronSphere, but the “final state” drawing of what can be built with this simple framework is worth examining.

5.1.10 See Also

Another way to discuss and work within this methodology is to consider hexoganal architecture (TODO insert ref and more definition). Each facet that’s available in a given component/repository has *known and allowed* interaction mechanisms and models for other facets. This allows exponentially faster composition of new solutions and highly flexible development teams.

5.2 Release Planning and Management with VDPA

5.2.1 Releasing Public NeuronSphere Tools

Imagine that you had a collection of libraries, CLIs, images, and various collections of json configs, and you wanted to consider publishing binary releases for some of those items.

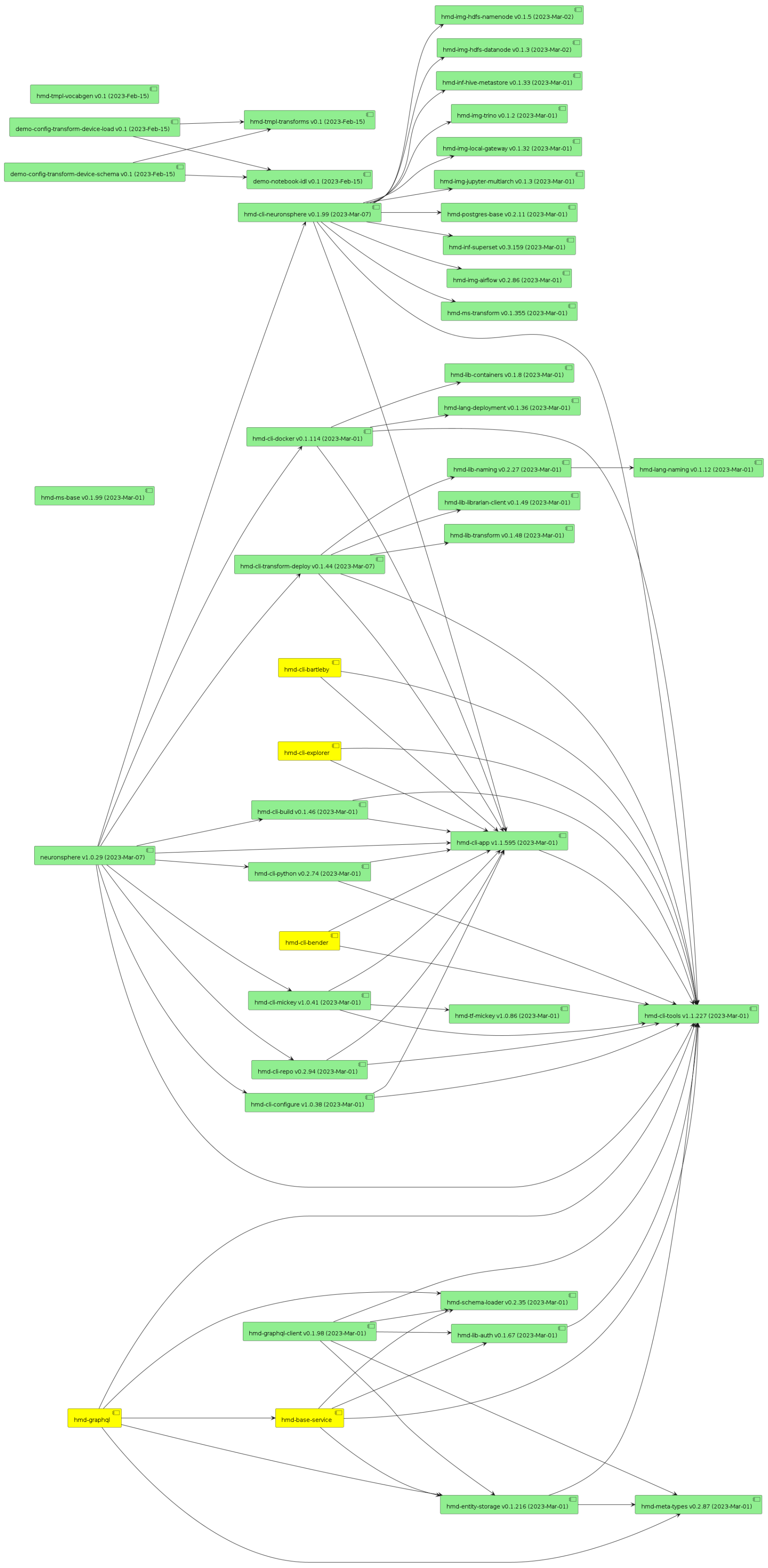
What does that mean? “Internally” as a company, we’ve been using these ideas extensively for several years, and we wanted to share some of our core tooling as freely available binary releases. This allows anyone to use and look at the base development tooling we use to build… whatever they want.

Our challenge was, we didn’t want to release all of it, rather pieces of it in waves.

The dialog went like this:

Brian: “I want to release our CLI <x> to public pypi”

Alex: “Ok, let’s look at what that entails”


hmd-img-hdfs-namenode v0.1.5 (2023-Mar-02)
hmd-img-hdfs-datanode v0.1.3 (2023-Mar-02)
hmd-inf-hive-metastore v0.1.33 (2023-Mar-01)
hmd-tmpl-vocabgen v0.1 (2023-Feb-15)
hmd-img-trino v0.1.2 (2023-Mar-01)
demo-config-transform-device-load v0.1 (2023-Feb-15)
hmd-tmpl-transforms v0.1 (2023-Feb-15)
hmd-img-local-gateway v0.1.32 (2023-Mar-01)
demo-config-transform-device-schema v0.1 (2023-Feb-15)
demo-notebook-idl v0.1 (2023-Feb-15)
hmd-img-jupyter-multiarch v0.1.3 (2023-Mar-01)
hmd-cli-neuronsphere v0.1.99 (2023-Mar-07)
hmd-postgres-base v0.2.11 (2023-Mar-01)
hmd-inf-superset v0.3.159 (2023-Mar-01)
hmd-img-airflow v0.2.86 (2023-Mar-01)
hmd-ms-transform v0.1.355 (2023-Mar-01)
hmd-lib-containers v0.1.8 (2023-Mar-01)
hmd-lang-deployment v0.1.36 (2023-Mar-01)
hmd-cli-docker v0.1.114 (2023-Mar-01)
hmd-lib-naming v0.2.27 (2023-Mar-01)
hmd-lang-naming v0.1.12 (2023-Mar-01)
hmd-ms-base v0.1.99 (2023-Mar-01)
hmd-lib-librarian-client v0.1.49 (2023-Mar-01)
hmd-lib-transform v0.1.48 (2023-Mar-01)
hmd-cli-transform-deploy v0.1.44 (2023-Mar-07)
hmd-cli-bartleby
hmd-cli-explorer
hmd-cli-build v0.1.46 (2023-Mar-01)
hmd-cli-app v1.1.595 (2023-Mar-01)
neuronsphere v1.0.29 (2023-Mar-07)
hmd-cli-python v0.2.74 (2023-Mar-01)
hmd-cli-bender
hmd-cli-mickey v1.0.41 (2023-Mar-01)
hmd-tf-mickey v1.0.86 (2023-Mar-01)
hmd-cli-tools v1.1.227 (2023-Mar-01)
hmd-cli-repo v0.2.94 (2023-Mar-01)
hmd-cli-configure v1.0.38 (2023-Mar-01)
hmd-schema-loader v0.2.35 (2023-Mar-01)
hmd-graphql-client v0.1.98 (2023-Mar-01)
hmd-lib-auth v0.1.67 (2023-Mar-01)
hmd-graphql
hmd-base-service
hmd-entity-storage v0.1.216 (2023-Mar-01)
hmd-meta-types v0.2.87 (2023-Mar-01)

Because we use dependency management and versioning that transcends technologies and uses the repositories as a unit of architectural quanta, we were able to start with the components we want, examine the graph, make some changes to alter some dependencies, and start working our process to release.

Since we use the idea of consistent repository behaviors and we have the capability to "release" Python packages and Docker images, it's a single command from any of the required repositories to move a binary from a private artifact store to public.

Rather than attempting to sift through a mono-repo or haphazard repo build process, likely trying to figure out which files to copy where in different parts of various scripts, this was a nearly painless process. We can reliably determine which versions of what were released when, and we can easily, reliably, and consistently plan for modules and features that we wish to release in the future.

5.3 Code-Generation

Code Generation is a key component of the software creation and execution process.

What is code generation? Roughly speaking, it's a process to take a small amount of data, (usually text), and create more text that happens to be some kind of computer-readable file(s), often *code*.

Where do we do it? Everywhere in computing. Everywhere. Many of the most ubiquitous and powerful pieces of software created have been yet another code generator in a specific sub-domain. There are many products in the software and data space that are little more than a graphical user interface over a simple code generator.

If you want to make a leap forward as a software or data engineer, there are few less powerful techniques than code generation. Nearly every system I've been a part of creating that really provided outsized developer power, efficiency, and longevity used code-generation as a critical part of its delivery.

Why would you write code, when you can write code that writes code for you? That's what code generators do, and learning to make your own and extend others as-needed is a distinctly powerful skill for software and data engineers.

Generally speaking, nearly all data & information modeling in software is a good candidate for code generation. Code generation is particularly useful when we want to interact with or manifest the same data models across multiple technologies or interfaces.

In any project of decent size you'll likely be using multiple code generators, either in concert or cacophony.

Note: For most of this book we will constrain our discussion of code-generators to those than accept *structured data as text* as opposed to those that take programs in a given and specific programming language as an input.

Said another way, we won't cover code-generation that requires the specification of a grammer or generation of a parser, though that is a great deal of fun and an additive power.

See also Mickey Transform Docs See also Data Models

5.3.1 Categories of Code Generation

This technique is so common that it is worth breaking down it's use cases into major categories. Some align with the project lifecycle while others align routinely with distinct parts of the architecture.

1. Project & Feature Scaffolding
2. IDLs & Data Exchange
3. Model-Driven Development/Engineering/Architecture
4. Runtime - makes expressions, makes sql/commands, makes whole new functions & programs.
5. Text to Visualization renderers - Web Browsers, Web Templating Frameworks, Document/Report rendering tools (e.g.: Jasper Reports, Sphinx-Doc/RST), Wikis

Note: Coverage of these categories is disproportionately focused on 1, 2, and 3, which will be given deeper coveregage in different parts of the book as they are most relevant.

Project and Feature Scaffolding

No one likes a blank slate to work from. On one hand the freedom to create is unparalleled, on the other you're required to make every single decision to go from nothing to done. This can be not only overwhelming, but offers you ample opportunity to waste your precious time figuring out how things work as well as making beginner mistakes or simple ommisions in layout and design.

This is where project and feature templates or scaffolding come into play. The idea is simple - take a few pieces of information from the user, and produce a *well-structured starting point* for a given kind of technology or feature. In platform engineering parlance, we would say this is the beginning of a *happy path* for a developer that needs to ship a new thing.

Maven(Java) & CookieCutter(Python) are both large ecosystems and powerful tools that primarily perform project scaffolding and many other languages have created variations on these themes. In Ruby and many web frameworks, you can use *subtemplates* to scaffold new features into existing projects. This *scaffolding* approach is commonly provided by the tools we use, and we can and should take advantage of it for ourselves.

Normally a *one and done* approach, project templates are a powerful way for the designers of a technology to communicate their ideas for a default unit of deployable work within a space. If you're making a product that has things *deployed onto or into* it, it behooves you to create an easy and effective project template or two. A recent and effective example of this is *dbt init*, which will create a default dbt project with all manner of nice defaults set up for the user.

A common complaint against setting up an internal template repository is that *old* projects will get out of date with the templates as they evolve over time. This is true, however there is a simple enough work-around that we've seen work quite well (and it's one of the only places you'll hear me advocate for the use of a source control branch). If you have a directory structure that came from a template and you want to check for drift or applicability of the new template, simply execute the new version of the template with the same input data into a distinct local branch <from_new_template>. Now compare <main> to <from_new_template>. You'll immediately see major file or structural differences, and can make an easy and informed decision about merging things from the new templates' execution into your <main> branch. Then of course you delete the local branch <from_new_template>. The *hmd repo* tool has a convenience feature to streamline this process.

A challenge with using public or tool-supplied project templates is that they're rarely aligned in structure and behavior across the multiple technologies, standards, or conventions that you have in place. There are several reasonable approaches.

If you're using One CLI to Rule Them All and the public or default template is good enough, simply create an extension to your cli that delgates to the tool, providing defaults and options that meet your standards.

Alternatively, take the template from the project's author and then merge it into your own ideas of a standard project for that type of technology. This allows you to make *N* of a type of project or feature, starting from the point that your team or company prefers. The naive belief that you'll only need to do a thing once keeps a lot of people from doing it, but it's an essential capability for scaling delivery.

If you're in a mono-repo, this looks more like a *feature scaffold* in that you're likely adding a templated subdirectory of some larger structure.

In a *rando-repository* scheme, you might see some repositories that look like *pure* versions of a project mixed arbitrarily with manually defined structures.

Using a standardized poly-repo strategy you will create a new sub-template of the standard base (or some other) for any new technology or feature type you introduce into your development platform. This aligns with the idea of *repository facets*, in that you should be able to overlay multiple project templates into a single repository target and always come up with an integratable and standardized repository structure. Another way to think about this is to contemplate the templates for each repository/technology facet as a *mixin* or *interface implementation*, in that executing a template into a standardized repository should produce not only code and file structures in the target, but also known behaviours for that facet related to build, deployment, and documentation.

Enterprise Architecture introduces the idea of an *executable reference architectures*, and effective project templating can be seen as a low-level implementaiton of that idea. If a developer uses a template to create a repository the code and configuration produced will ideally execute successfuly to a known simple result with no changes. This gives the developer something that *works as intended* and is designed according to standards as a starting point. Creating multiple repository templates that are designed to produce inter-operable components can again further increase the speed to get a development team moving on a new initiative.

IDLs & Data Exchanges

As long as a piece of software is only communicating with itself in a given runtime, communicating with memory and the processor and going about it's merry way executing instructions and storing results, software can be considered *relatively* simple. That said, much of the software written today is intended to operate in a far more complex environment - one where it is connected via network to other running pieces of software, sharing data back and forth.

This is a distinct field of data modeling patterns that also *nearly always* uses code generation, so we'll see this list discussed again from that perspective.

- CORBA
- WSDL/POX
- JSON*
- OpenAPI
- Protobuf
- GraphQL

TODO: drawing on the details of where generated clients and sdks fit into the ecosystem

In nearly every case I'm aware of, this is already a *lossy* conversion process, in that there will be (in some cases) multiple language-specfic implementation and type-coertion details baked into all generated implementations.

Model Driven Development

See :ref:Model-Driven Code Generation

Runtime

TODO add flesh

- ORMs & SQL (ish) tools (Hibernate/Linq/SqlAlchemy/Orms & Record Tools)
- ETL/ELT Tools (from airflow & Dbt to "source to warehouse" ETL tools)
- 'Semantic Layers' and BI Tools writ large
- IaC tools

Visualization & Document Renderers

From Tex to Tableau, Jasper Reports to Sphinx-Doc. Oh, and the internet cuz HTML is one. Wikis, web-templating frameworks that MAKE html. SVG is a structured text to Visualization abstraction that's pervasive in its interoperability.

5.3.2 A code generator of your own

With all these amazing tools that generate nearly infinite amounts of code, why would I want another one?

Polyglot input models. Ability to use advanced model-driven-engineering and architecture techniques pervasively.

TODO refine/remove/add to

5.4 Documents and Diagrams as Code

What does it mean to do something "as Code"? It's an interesting expression surely created by software engineers because while it's accurate I'm not sure it really conveys where we are as an industry with this technique.

Realistically, it was more likely created by savvy marketers in an attempt to capture the attention of the industry as a parallel to the far better-known "Infrastructure as Code", and that's not a bad comparison.

What makes it "as code"? Documentation and Drawings that are:

1. authored in *plain text* usually a *markup language*.
2. stored in a *source control system*

It certainly alienates a lot of people that could be benefiting from the technology, calling it *code*, when in fact it's one of the most well-known and used techniques in academic and technical writing.

If you want to be far more productive at creating and managing high-quality technical documentation, these ideas have proven to be helpful in removing the more annoying parts of the process.

5.4.1 Separating content from style - again

In software, we often see the same themes presented over and over in different guises, and the idea of separating content from style is pervasive.

The basics of HTML include the foundational idea that the content is presented in markup, and the style is "applied" to it for presentation to the consumer. The basics of this are often deeply hidden beneath layers of javascript web framework at this point, but all of them further enforce and often make more powerful this useful separation.

> "Software is hard" - Donald Knuth

The case for doing things "as code"

1. Challenges with software and data documentation

   1.1 Layout is hard. 1.2 Repetition (particularly with data) is inherent and problematic

5.4.2 Why isn't Software Documentation Easier/Better?

When discussing software documentation it's often noted that "It's not a tool problem, documentation is just hard to write and time-consuming and devs don't like to do it and it's always out of date with the code anyway". There are truths in these assertions, but little hope of a solution unless we examine the problem more carefully.

To deliver a *business capability* we need to consider the 4 aspects of People, Process, Information, and Tools.

As such for the broad capability of "Produces Appropriate Software Documentation" for a development team, we consider all 4 aspects.

1. People
2. Process
3. Information
4. Tools

The code moves faster than the docs. Then you have to make the docs move really fast.

Documents as Code

TODO

5.4.3 Updating the standard repository for DDaC

Given the standard repository from the previous section and the idea that all repositories should follow similar conventions and behaviors, we will add a "/docs" folder to our standard repository layout.

5.4.4 What about comments in the code?

5.4.5 Automating Data Documentation [as Code]

5.4.6 What about Readme?

5.4.7 Implications of Code Generation and the Build Lifecycle

5.4.8 Consumer Focused Publishing

Building for local use Publishing for the current release Publishing for environment-specific collections Publishing to document control and qa

5.4.9 Output Format Considerations

When considering various consumers for software documentation we also need to look at the specific technical capabilities of our DDaC toolchain related to output format.

Common target formats include: * Single-Page HTML (with interlinks etc) * Multi-Page HTML * PDF

5.4.10 What goes in the software wiki?

(blank section)

5.4.11 Why is the wiki section blank?

- Creating "internal" and "external" documentation is an anti-pattern, sure to make things worse not better. wikis are often the "internal" document source.

If you do TBD+simple CI/CD for the 'wiki' section of your documentation, you have everything you get with a wiki and all the benefits of DDaC

If you do 'gitflow' for your wiki, it'll never work.

Diagrams as Code

"A picture is worth a thousand words" - a lovely adage, and one that is often true in software and data architecture. I have created architecture drawings that have lasted as references for years and acted as guideposts for multi-million dollar, multi-year, multi-national IT programs. It cannot be overstated the value of a good drawing in unifying people around a complex set of ideas.

While there's a lot written on the subject of drawings in software architecture it's not a commonly taught skill. I'll also note that I'm not an authority on this subject, but the ability to use these techniques to illustrate the vast majority of this book quickly should be evidence that one does not need to be a "software drawing expert" in order to more effectively use drawings to communicate your ideas.

- It's actually kind of hard to draw decent pictures, especially without guidance or examples. This is a place where 'constraints make you go faster'. The infinite pallet of drawing tools and combinations in Visio or LucidChart are great to control the whole design, but they require you to control the whole design in many cases.
- Similar to documentation, there is an issue of styling and consistency
-
- Version control of drawings is crazy hard. Given up layout control presents an exponential reduction in the problem space, once that's often worth giving up layout consistency for.

5.4.12 Letting go of layout control (mostly)

5.5 Branching

We have a lot of things to cover on the way to building scalable data ecosystems, and since a key premise of this work is *everything is code* and we store it all religiously in source control - we have to talk about branching.

In short - don't do it.

What do you do instead? Trunk Based Development (TBD)

Their Summary:

A source-control branching model, where developers collaborate on code in a single branch called 'trunk', resist any pressure to create other long-lived development branches by employing documented techniques. They therefore avoid merge hell, do not break the build, and live happily ever after.

- main for the Git community since 2020

Note: It is challenging to think about TBD if you're only used to gitflow or don't have multiple environments set up. In the sections on CI/CDD we'll spend more time discussing how TBD and good CI/CDD work hand-in-glove for a rapid software delivery experience.

CHAPTER

SIX

BASE(D) SOFTWARE DELIVERY

This section adds to the ideas for basic software delivery and extends them to providing ‘application, data, and platform services’

This includes things like Continuous Integration, Continuous Delivery, and importantly, Continuous Deployment and Verification.

About “pipelines” - a software package should have no control over the deployment pipeline that delivers it.

CHAPTER

SEVEN

FULL-STACK SEMANTIC SOFTWARE

7.1 Entities From Data

All models are wrong, some are useful - George Box

7.1.1 Storing Things With Code - An Ever-Present Opportunity

Regardless of programming language, business domain, or architecture, there is a consistent challenge pertaining to the storage and transmission of data *outside of* a program's internal runtime & memory representations.

Why a property graph model? A "pure" ER model has a number of benefits.

- Projection onto multiple 'relational' modeling targets (3nf, vault, anchor, etc)
- IDLs are often modeled for storage OR transit, but not both. This creates an inherent and deeper *impedance mismatch* than that of a simple ORM or language <-> persistence mechanism.
- The code manifestations are "simple" objects, using the OO behaviors of languages as appropriate, but avoiding the challenges of a *deep* OO mapping attempt. Attempting to fully map to the array of OO is part of why ORMS are so tightly tied.

At a logical level, we need to talk about *the 2 layers* problem.

We model a lot of data, we've discussed countless ways to represent it, yet each contains substantial mismatches.

Using the global graph, new applications can add new relationships and capabilities in an additive yet namespaced way.

You can add ontologies on the global state data and look for data that is an is not included to drive knowledge governance.

How to use this within existing repositories and source models to add lineage/tracking by creating the pointers from the NS information objects into the manifestations (openAPI, protobuf extensions, etc)

7.2 The Power and Pain of Polyglot Storage

Software that interacts with data over time that is somehow persistent introduces several new dimensions of challenges in software design.

1. **Software Representation and Separation of Data & Function** -
2. **Data Model Evolution** -

3. **Ease & Performance of Data Storage & Retrieval** - This covers a matrix of challenges - ease for programmers to interact with the data store, ease to perform various kinds of retrieval and bulk operations. When 2 or more applications share a storage system, discussion about read versus write performance and ease of change can become complex.

These are vast subjects, and many of us in technology will spend years chasing subtle changes and differences in these factors. To be clear, trade-offs in these ideas can have long-lasting implications for your software, and will be where you will spend a lot of time in discussion, change, and optimization.

7.2.1 Chasing the Holy Grail of "One Database to Rule Them All"

This is a good pattern, because databases are complex, expensive, and tend to last a long time. As such from an architectural constraint standpoint, we should strive to have as few *kinds* of databases as possible, and ideally as few *instances* to manage and reason about.

Of course, like all architectural advice this exists on a gradient, and we have to accept that it is highly likely that we will end up with not only multiple instances of databases, but also multiple different kinds of databases.

We spend a lot of time as an industry discussing things like ACID capabilities and the CAP theorem, or PACELC, and these foundational ideas in conjunction with fully software-defined ecosystems have allowed an explosion in new data persistence and computation engines.

Given a modular and ever-growing solution landscape, how to we consider PACELC in consideration of integration between the various systems and their persistence stores?

7.2.2 Starting at ORMs

We start with ORMs because it's the 'oldest' technique most readers will encounter at the time of this writing. ORMs are considered 'old hat' by many technologists, but their success and continual usage means they present several patterns worth examining.

- Influence of the ORM on App and DB design - it works that well

7.3 Data Modeling - Where Does It Fit?

Having explored a common framework for discussion, we want to delve briefly into data modeling and how it fits into a larger mental model of what we're building.

Speaking of models being wrong:

Storage models vs Interchange models (relational storage models vs hierarchical interchange models) OLTP vs OLAP Transactional vs Master vs Reference

OLTP - 3(x)nf, OLT, Activity, Entity Graph OLAP - Dimensional, Vault, Anchor, 3nf+,

There's a difference between a database technology and its capabilities versus how you choose to model a piece of information within it. You can do amazing and horrible things with every database available.

3nf, anchor, vault, dimensional, activity, and property graph all map cleanly to relational databases, but each has substantial pros and cons.

There is a correlation between the time and complexity put into the data model versus its long-term usability, and much time and energy will likely be spent evolving these data structures over the years to come in support of your systems.

By modeling and unifying our models in a language that is itself designed to be a knowledge graph means the act of delivering neuronsphere modeled and powered data products and using the build service with introspection and extraction means the act of delivery IS the act of updating the lineage in many cases.

More exactly - a 'true' entity relationship model can be projected onto N storage mechanisms and models deterministically, and we take advantage of that simple idea to build as complete a data model as we want.

7.4 Data Modeling - Poorly Captured Reality

Accepting that we have multiple places in our architecture to model data, we should put together some notes around these different ideas - a model for our data models.

Note: It can be challenging to discuss this at the right level of detail without delving into some technology-specific ideas around different vendors and implementation details.

In the RDMS/"SQL supporting" categories, there can be vast performance differences with different models that appear identical.

This next level of detail - determining the model's validity *implemented on* a given technology on given infrastructure, with a given set of readers and writers - is a routine part of the detailed architecture & delivery process, and it's anticipated that experimentation and blending will happen here as you evolve.

To be clear - it is likely that there is no *one perfect data model* for what you want to do, in either the transactional or analytical domains. In fact, data model evolution is a large and never-ending part of the profession.

7.4.1 State, history, and the challenges of time

TODO

7.4.2 Transactional Storage Models

This is the one many developers encounter first - the challenge of needing to store transactional, detailed information. Things like accounting ledgers, inventory movements, sales deal tracking, and crm interactions are all easy examples of transactional systems.

Broadly speaking, storage engines intended for transactional workload are often broken into SQL/noSQL engines - a broad but reasonable categorization for the sake of this discussion.

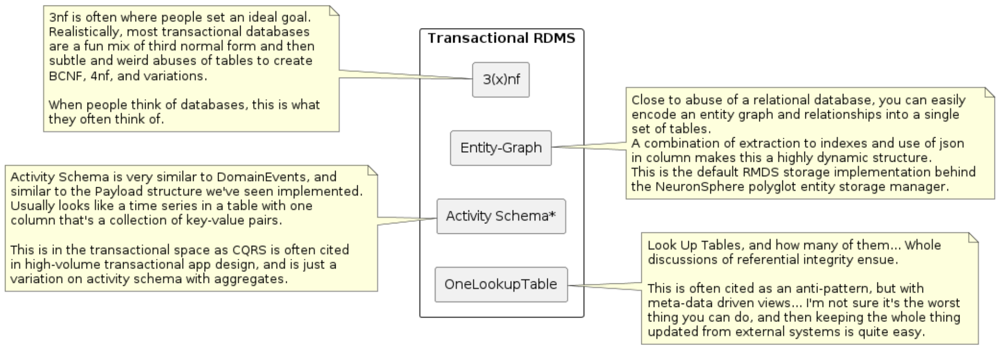
Transactional RDMS (SQL Databases)

Solving this problem - *store the data in the database* is ubiquitous and the Relational Database Management System (RDMS) is still the workhorse in many applications.

Note that there are multiple data modeling patterns that can all be implemented in an RDMS, and often we don't see *pure* implementations of any of them, rather most SQL databases will be a combination of patterns based on subtle application demands and design decisions.

A lot of what drives RDMS design away from clean 3nf/4nf implementations and into interesting patterns is the demand for change and the knowledge of how hard physical change in a database is over time. This inherent knowledge, combined with things like views, procedures, and effective complex-structure handling, and most non-trivial transactional RDMS will exhibit all kinds of other anti-patterns (sparse table etc)

- 3/x+ nf
- One Lookup Table
- Entity-Graph
- Activity Schema/Domain Events



3nf is often where people set an ideal goal. Realistically, most transactional databases are a fun mix of third normal form and then subtle and weird abuses of tables to create BCNF, 4nf, and variations.

When people think of databases, this is what they often think of.

Close to abuse of a relational database, you can easily encode an entity graph and relationships into a single set of tables.
A combination of extraction to indexes and use of json in column makes this a highly dynamic structure.
This is the default RMDS storage implementation behind the NeuronSphere polyglot entity storage manager.

Activity Schema is very similar to DomainEvents, and similar to the Payload structure we've seen implemented. Usually looks like a time series in a table with one column that's a collection of key-value pairs.

This is in the transactional space as CQRS is often cited in high-volume transactional app design, and is just a variation on activity schema with aggregates.

Look Up Tables, and how many of them... Whole discussions of referential integrity ensue.

This is often cited as an anti-pattern, but with meta-data driven views... I'm not sure it's the worst thing you can do, and then keeping the whole thing updated from external systems is quite easy.

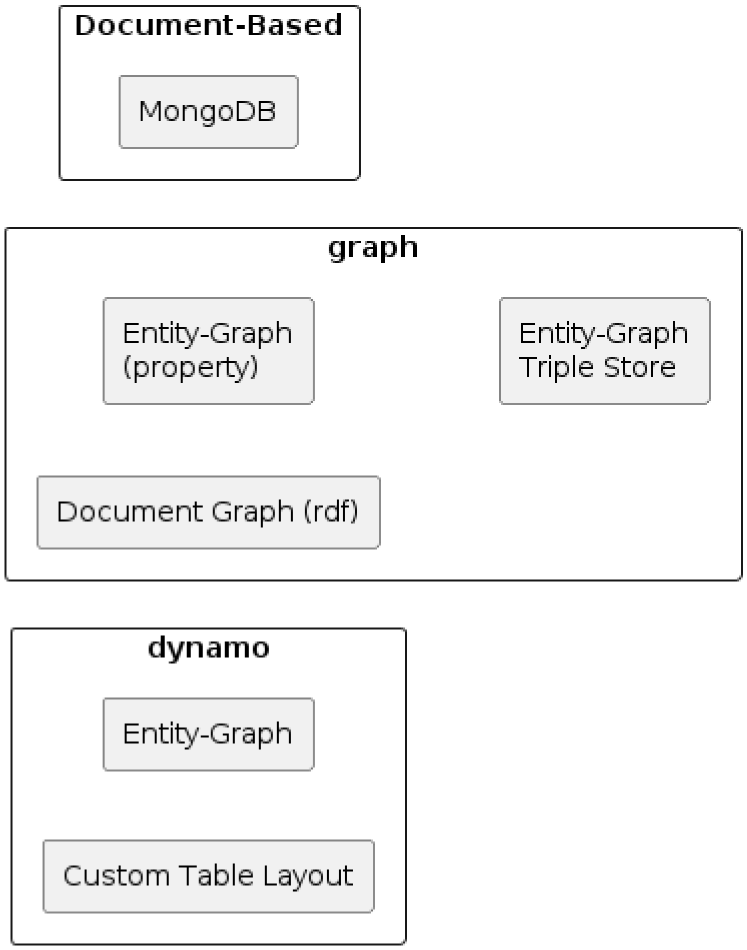
NoSQL Models

For all the power and flexibility in RDMS-based storage engines (aka *SQL Databases*), they have their challenges. Data model evolution is one area where SQL databases can find challenge, and performance for particular use-cases is the other.

In response an array of new storage engines have come to market, and in broad strokes they're called *noSQL* databases because the method of data modeling and interaction is not SQL based.

Without going into a long discussion on the subject, the decision to use a NoSQL engine has broad consequence, and in many cases will see greater issue with even basic *analytical* style queries that the application may want to serve. Said another way, there is enough commonality between transactional RDMS systems that often switching is expensive but reasonable. Switching from a SQL-based engine to a NoSQL engine, or back, can be substantially more difficult.

Note: Many/most don't or can't reasonably support SQL (even on the read side), and there's often tight binding in the code to the specificities of the storage engine. This often has a deep impact on portability of code and logic, as abstraction layers between code and storage engine are less common in the NoSQL ecosystem. This often presenting scaling challenges in the analytical dimention that will lead to data replication into storage systems more suited for analytics.



7.4.3 Interchange Models

The demand to interchange data between software systems has existed since we began to connect software to communication networks. Understanding and appreciating the various dimensions at play in this space is more an exercise in field theory than a simple *(x,y)* space, and we will only lightly touch on it in this text. (see :ref: Enterprise Integration Patterss)

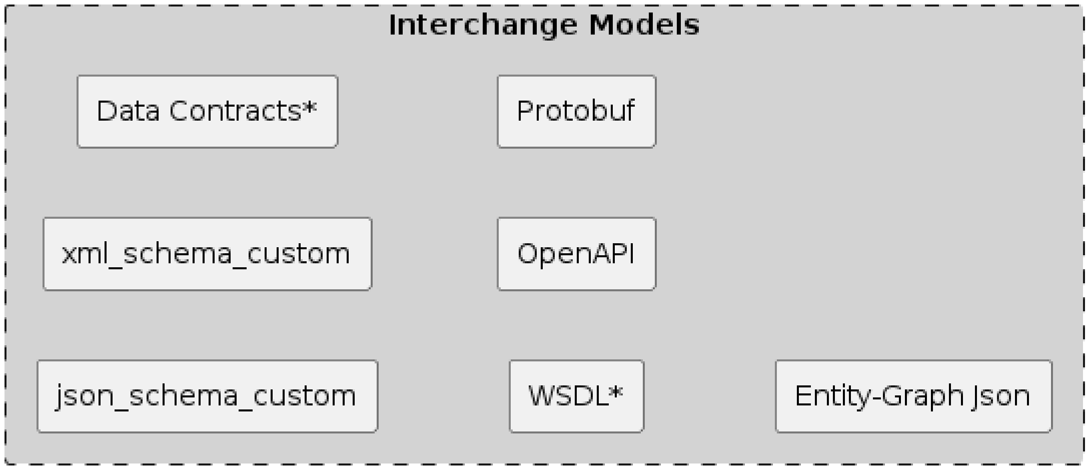
What problems are we trying to solve?

1. Demands for RPC/Distributed processing/data interoperability
2. Multiple programming languages

TODO: Marshalling/Unmarshalling, a-priori knowledge, readability, performance, implementatbility, data shape and flow shape

Portable (complex) Object Structures & Messages

- *proprietary record formats* we'll lump everything pre-corba into this category. Data was exchanged, but not in an RPC fashion, more likely tape-based record structures.
- **CORBA** - 1991, the primogenitor of interchange technologies, introduces the IDL and a number of key ideas that are still with us today
- **XML-Schema/WS*/POX** - came to kill CORBA, ended up hurting us all with it's pointy bits and schema ideas so complex only an engineer can love it
- **JSON** - came to kill xml, ended up with competing and incomplete schema ideas and downstream pain galore, but hey, braces instead of <*pointy_things*> has benefits
- **OpenAPI** - adds the *RPC* parts of an IDL to *JSON exchanges* and solidifies an object-in-json metamodel and like WSDL and XML schema, lots of people don't really do this completely (because you don't NEED to to *make things work*)
- **Protobuf** - full IDL including RPC, good object modeling metamodel with vast capability, opinion, and output options, trades opinion and performance for difficulty in extending the generator model and unreadability of marshalled objects without a-priori message knowledge. Possible to abuse, but hard. Particularly with recent advances, this is a very safe choice for many interchange and large-scale data modeling demands.
- **GraphQL** - TODO
- **OWL & RDF** - TODO



Note: Each of these technologies also has implications for code-generation and the overall required capabilities of the build and artifact management capabilities of a codebase & team. see :ref: *code-generation* ` and :ref: *CICDD* `

TODO: Wire Protocols and Transport and relationship to HTTP/REST and Messaging

Sync and Async, Flostam and Jetsam

Bulk Data aka Table Formats and Tabular Structures

The data storage community finally gets serious about interoperable database-capable storage.

TODO: Describe why we'll never have one format to rule them all. TODO: picture of where table formats "fit" in the standard architecture TODO: picture of where data contracts "fit" in the standard architecture

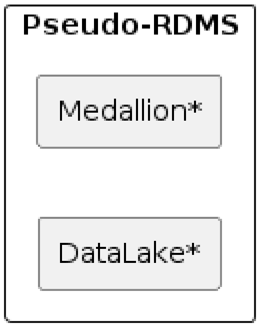
- **File & Table formats** - (avro, parquet/iceberg/delta, orc)
- **Data Contracts (etc)** - (other schema/rule management approaches)

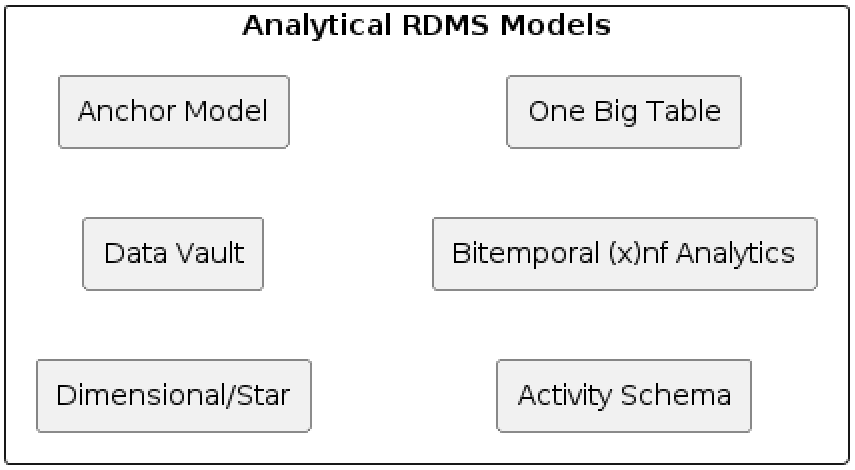
7.4.4 Analytical Models

TODO: also "big data" "cloud data warehouse" and discussions of SMP v MPP, ACID/CAP/etc.

The Data Warehouse according to Bill Inmon: "subject-oriented, nonvolatile, integrated, time-variant collection of data in support of management's decisions."

The new (since hashtag#wwdvc 2019) definition updated by Mr Inmon with respect to hashtag#Datavault is "A data warehouse is a subject-oriented, integrated (by business key), time-variant and non-volatile collection of data in support of management's decision-making process, and/or in support of auditability as a system-of-record."





7.4.5 NeuronSphere Models

NeuronSphere introduces a 2-layer information and data modeling system, designed to address the underlying challenges inherent in today's polyglot data model landscape.

At the Semantic Model layer, information is modeled as a graph of entities and relationships, allowing infinite flexibility and seamless evolution and rule enforcement of the data model.

At the data layer, data is modeled more specifically as tables or pointers into object-based structures in more specific modeling specifications (e.g.: OpenAPI, Protobuf). This *data metamodel* allows interlinking across various specific models used in a modern software and data ecosystem, providing logical and runtime lineage capabilities at a field level, regardless of the storage engine, abstraction, or format.

Why would we add another modeling technique to an already crowded field?

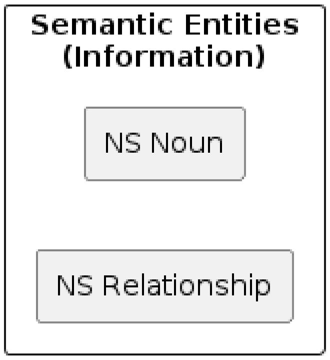
An Entity-Relationship(Property Graph) flavored model allows fluent data, entity, and rule modeling without consideration of storage, transit, or language, while allowing for a *clean default projection* onto polyglot storage, multiple transit options, and *n* supportable target languages.

NeuronSphere Semantic Metamodel - The Entity Graph

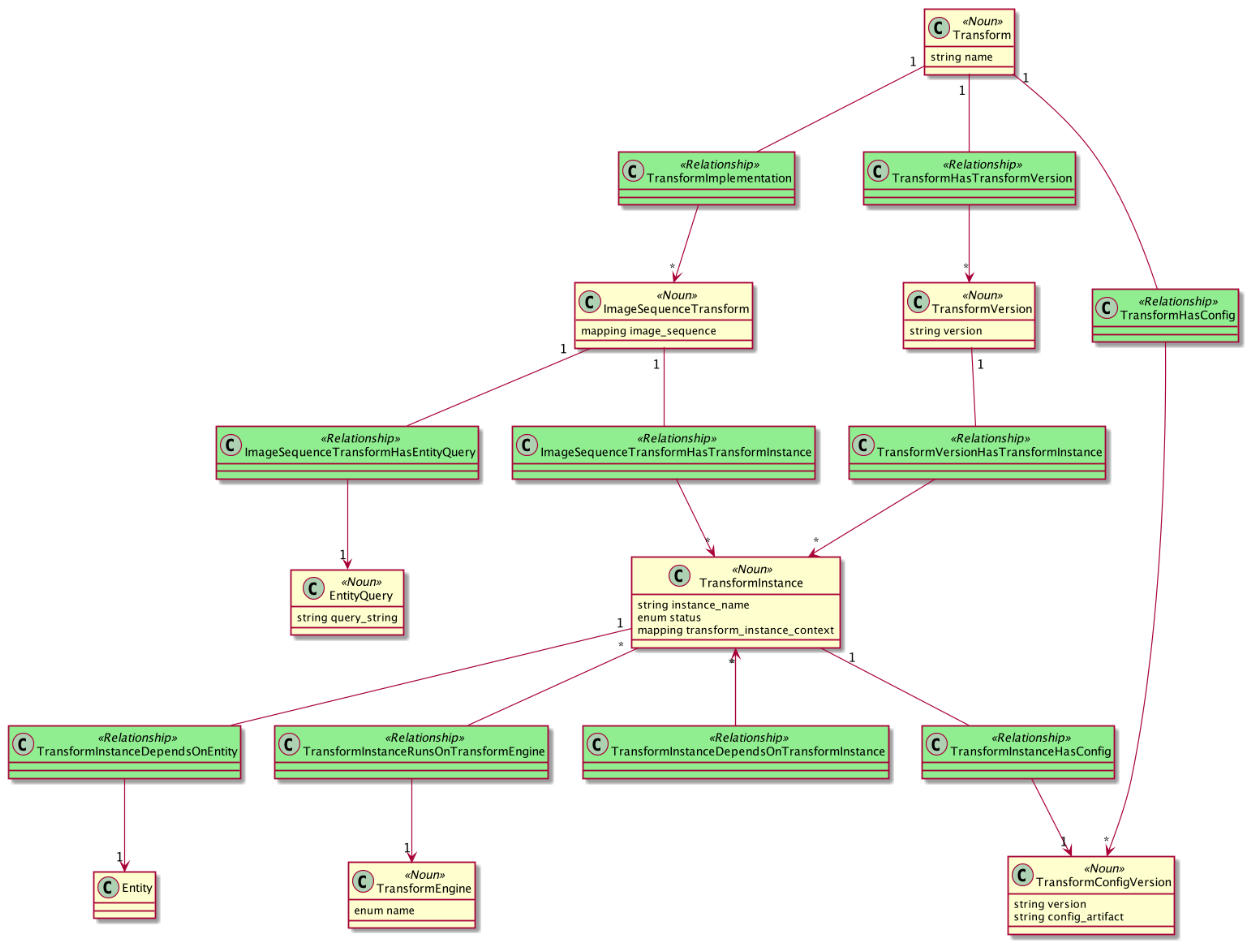
'HMD Entities' - Nouns and Relationships, each with attributes.

This is strongly modeled after a property graph, and the attribute type-system is intentionally high-level. Entities are intended to be at the 'Information' level of the logical DIKW pyramid and form the default objects interchanged by the NeuronSphere Microservice & Entity Persistence frameworks. A 'Language Pack' is a namespaced collection of HMD Nouns and Relationships.

Additionally, these namespaces form namespaces, packages, and modules as appropriate in target languages.



An example language pack is shown here, this is a slightly aged version of the Transform language pack, the primary semantic namespace for the Transform Service.


«Noun» Transform
string name
«Relationship» TransformImplementation
«Relationship» TransformHasTransformVersion
«Noun» ImageSequenceTransform
mapping image_sequence
«Noun» TransformVersion
string version
«Relationship» TransformHasConfig
«Relationship» ImageSequenceTransformHasEntityQuery
«Relationship» ImageSequenceTransformHasTransformInstance
«Relationship» TransformVersionHasTransformInstance
«Noun» EntityQuery
string query_string
«Noun» TransformInstance
string instance_name
enum status
mapping transform_instance_context
«Relationship» TransformInstanceDependsOnEntity
«Relationship» TransformInstanceRunsOnTransformEngine
«Relationship» TransformInstanceDependsOnTransformInstance
«Relationship» TransformInstanceHasConfig
Entity
«Noun» TransformEngine
enum name
«Noun» TransformConfigVersion
string version
string config_artifact

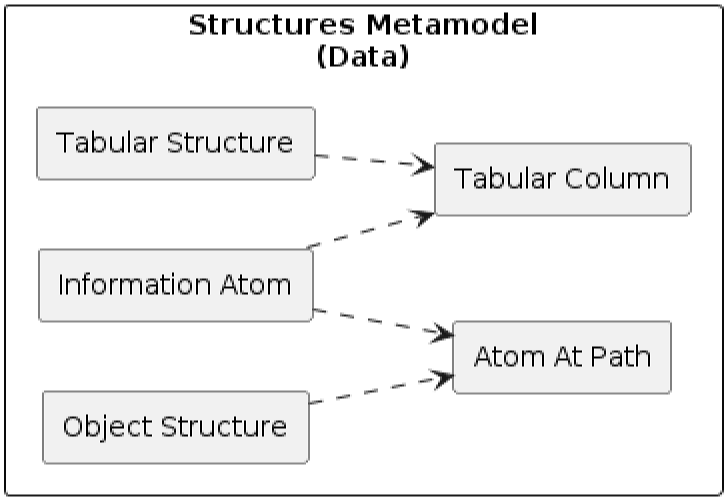
Note: During the initial implementation of NeuronSphere's core modeling constructs and capabilities, we went to some lengths to try to simply use a subset of an existing schema technology, with the goal to allow modeling in a subset of an existing schema framework. It did not work well and was abandoned quickly.

NeuronSphere Data Model

At the *data* layer where we model detailed data layouts and specifications into things like tables and object graphs, we see abstractions for those 2 branches respectively, with abstractions to unify across both into a single information model for polyglot data governance.

***Tabular_Structure*, *Tabular_Column* et al - Entities in Language Pack hmd-lang-structure, *Tabular_Structure* and**

its related *Tabular_Column* and *Manifestation* are used to model largely *rectangular* collections of data at the 'Data' layer of the DIKW pyramid. Tabular Structures and their related Columns are logical, providing a layer of abstraction over the entity *Manifestation*, which is a *Tabular Structure* and its *Tabular Columns*, combined with properties required by the *Manifestation*, Entities, and templates to create a physical entity (table/view/file) in a database. Of note is that referential relationships between tables are not currently modeled. Cross-column references are possible between *Tabular Columns* but are not used to derive or produce referential integrity constructs at this time.



If we're putting a rule for a data field on a table, it's in the wrong place. This model extends into default places for rules associated with data elements, regardless if used in a protobuf message or a MySql database table.

Data contracts that are putting rules on table columns without reference to a higher-order information store will produce redundant and potentially conflicting implementations.

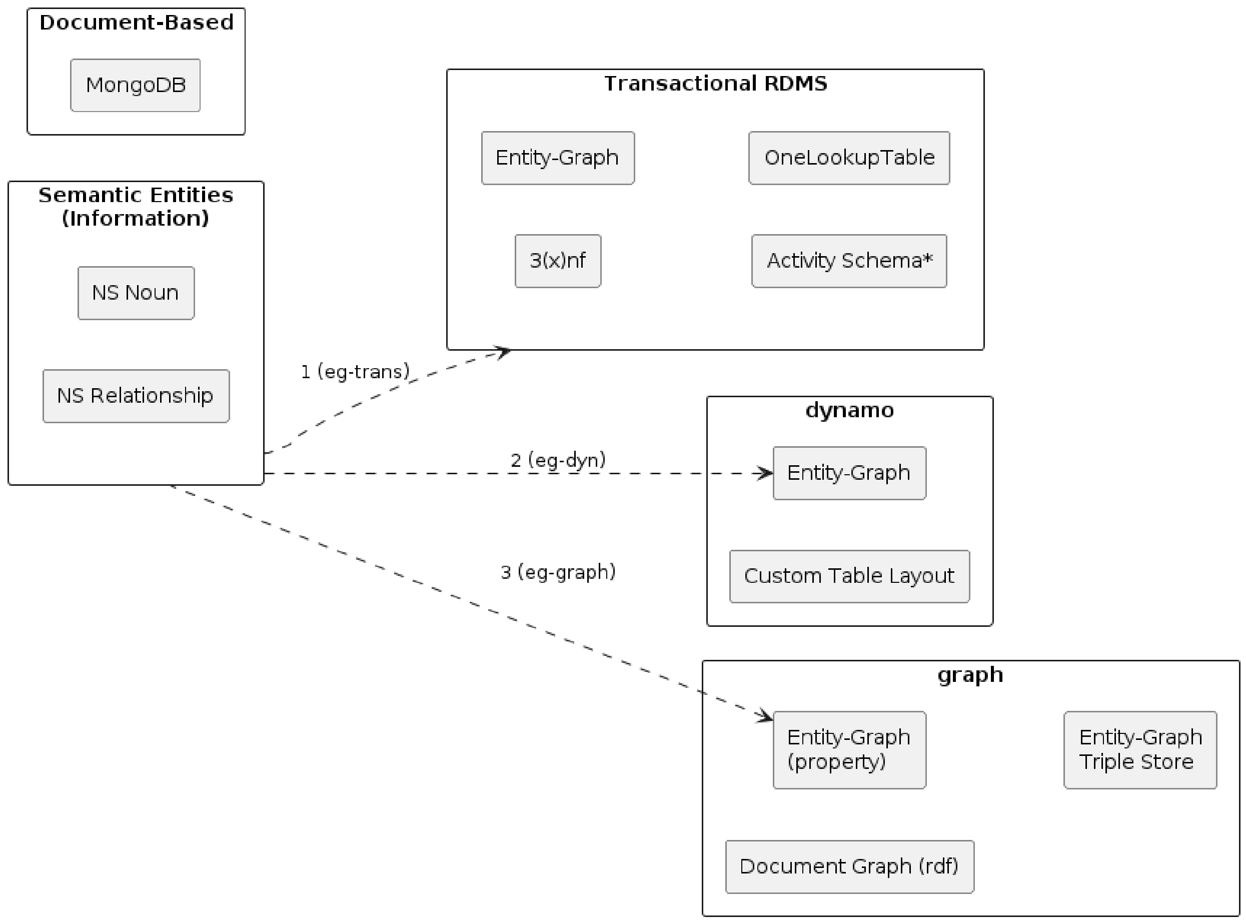
7.4.6 Model Projection, Transformation, Lineage

Given a desire to *build software and data systems* and an understanding of the multiple kinds of data models needed to make that work, combined with ideas around code generation and model-driven architecture, we arrive at a powerful and visual demonstration of how the NeuronSphere 2-layer model allows projection into *n* implementing models.

What does it mean to *project* from one model to another in this context? Moving from something general - the semantic model - to more specific (polyglot) implementing models, usually via code-generation.

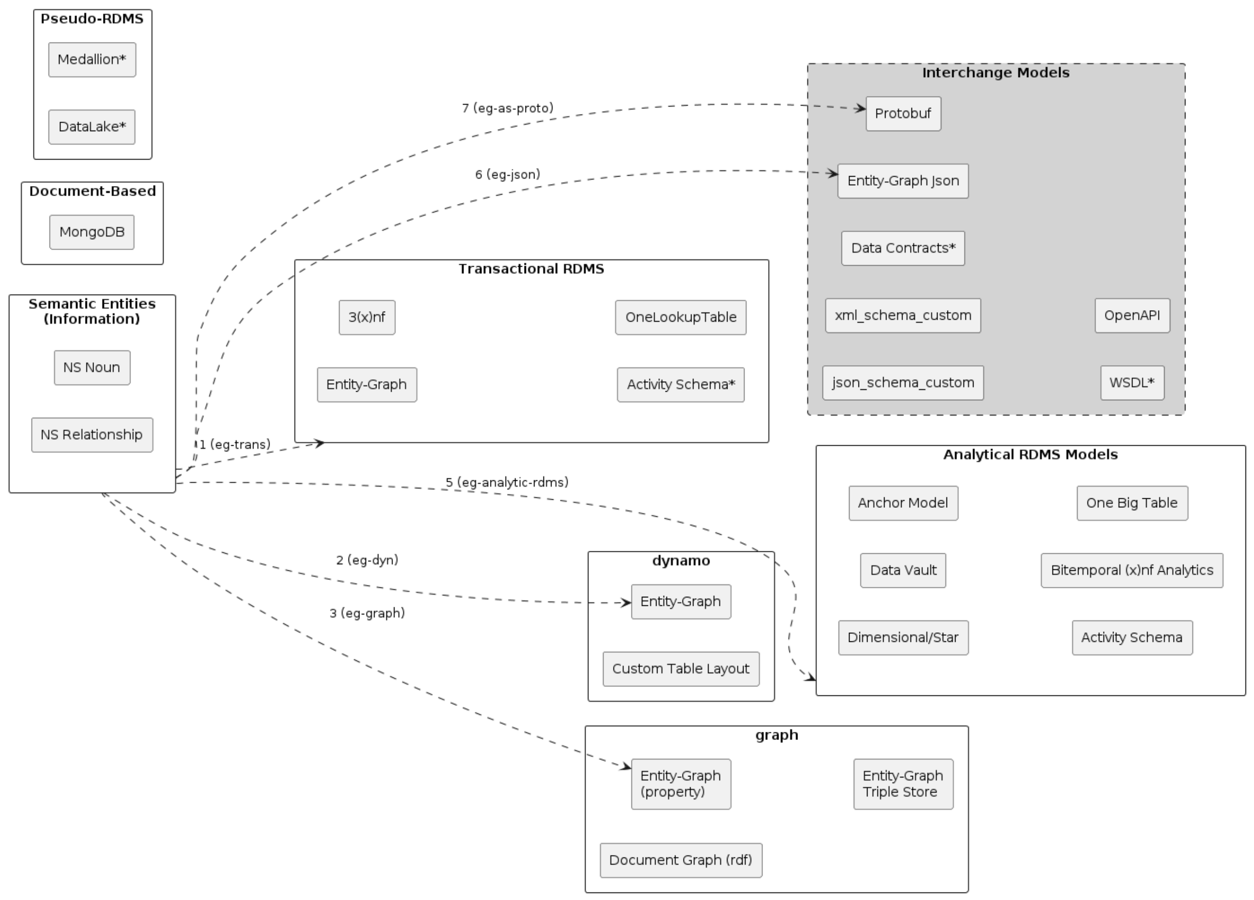
Semantic Model Projections

1. eg-trans - Entity Graph to Transactional Storage - NeuronSphere Nouns and Relationships, generally an ACID compliant RDMS.
    - Entity Graph (Denormalized RDS)
    - 3-(x) Normal Form
    - One Lookup Table
2. eg-dyn
3. eg-graph



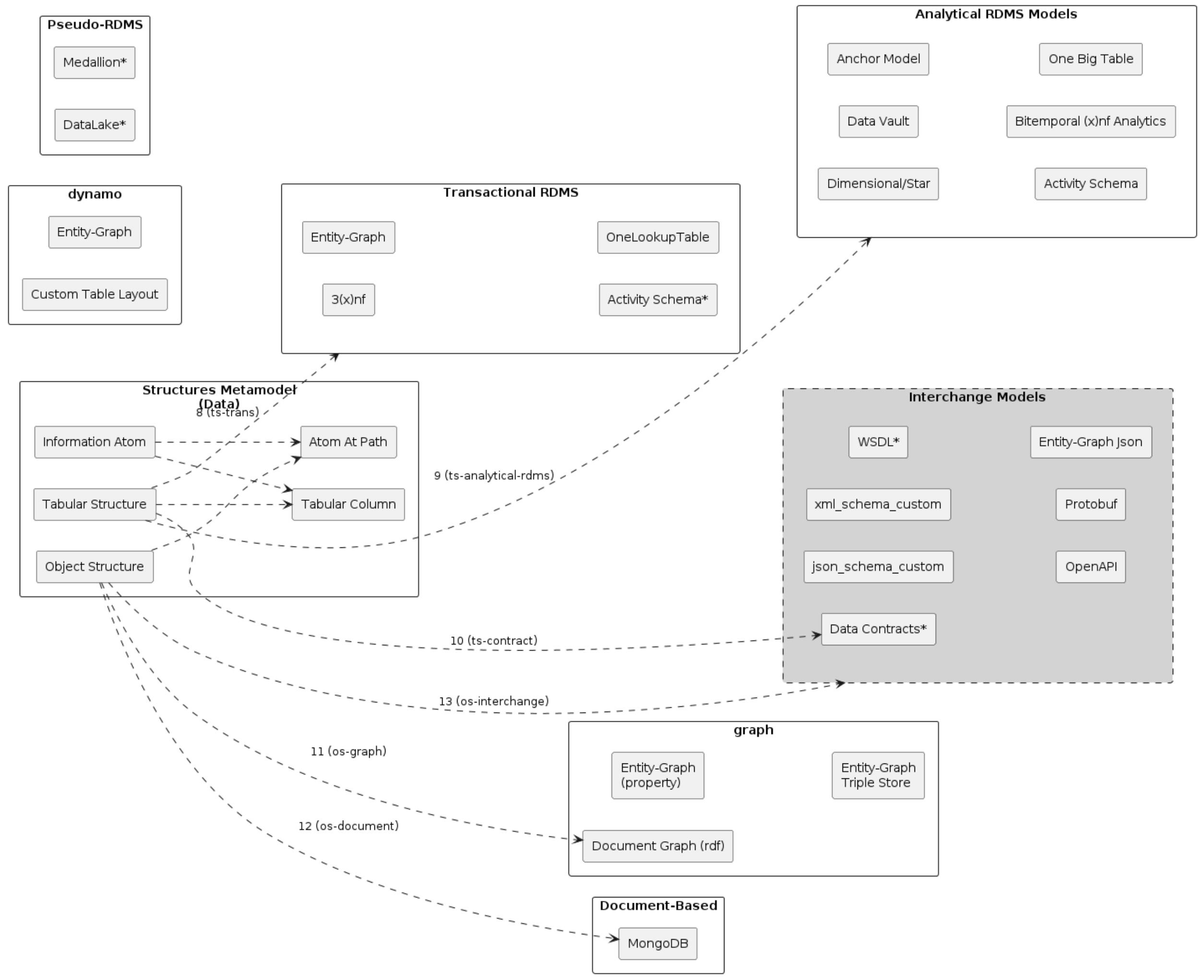
4. eg-analytical-rdms - Semantic Entities, as an ER model, can be projected to multiple analytical materializations.

   - Dimensional/Star Elements
   - Bitemporal (x)nf
   - Anchor - decorate the semantic entities to get to the input graph for anchor modeling
   - Data Vault - decorate the semantic entities to get to the input graph for anchor modeling

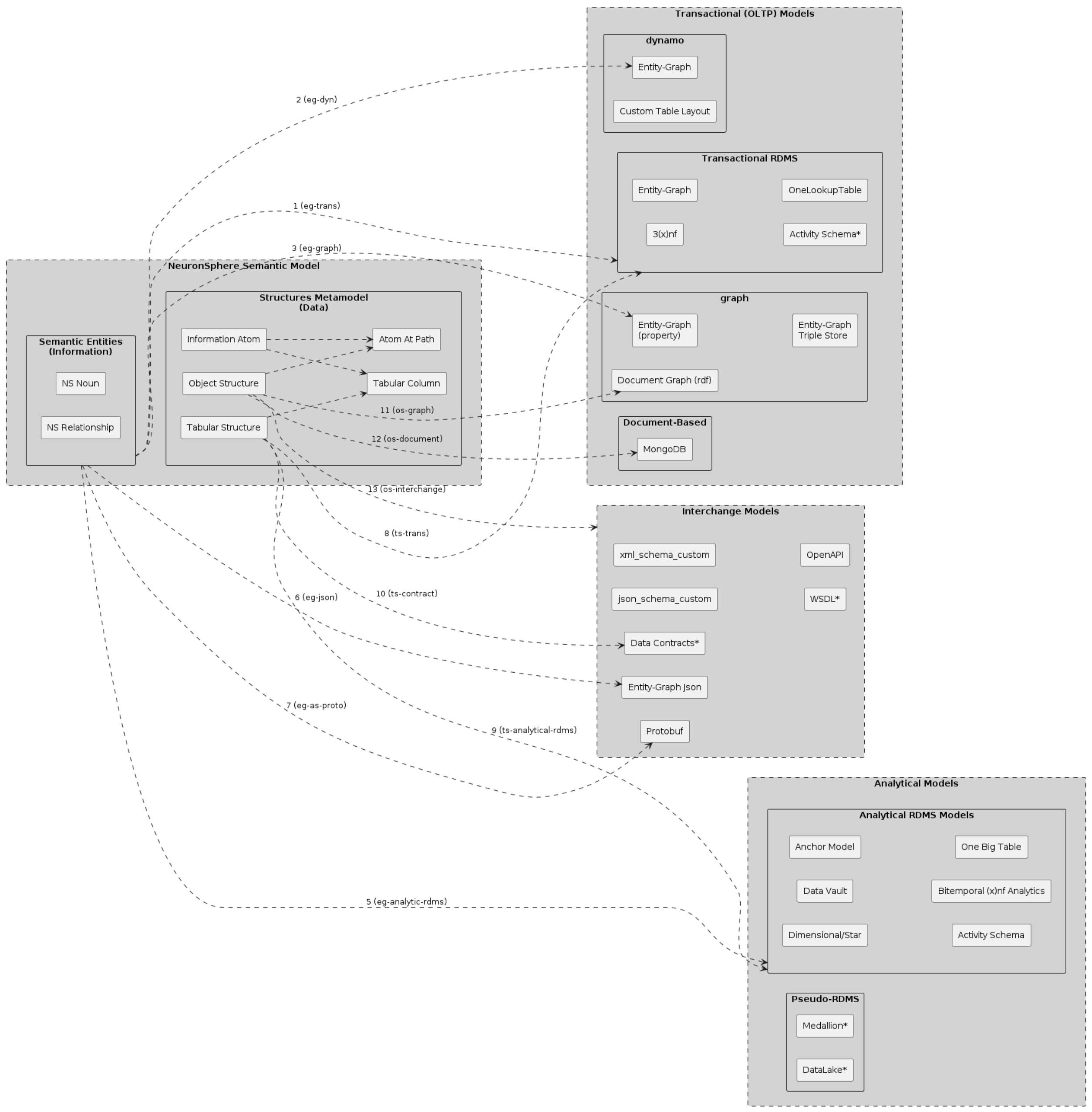


This means that we can create a single set of entities and relationships, decorate them as needed to override defaults, and generate transactional storage models for various state-managing applications as well as the analytical structures required to store and process historical states.

Information Model Projections



7.4.7 Full Model of Models



Relationships to Other Ideas

Domain Driven Design is a popular and powerful technique, and the ability to combine collections of entities and relationships into logical domains is useful, but lacks implementation specificity. Further, it is often primarily discussed in the context of [stateful]application design, not analytical systems.

By using an Entity Relational model as a baseline, rather than directly tabular, we can construct various models across the stack?

A domain could be new Nouns and Relationships, or instances of metamodel objects like *TabularStructure* and *Information Field*, and is often a combination of both.

- The major entities in a given domain, and their *major* attributes & identifiers, will inform and be refelcted in all subordinate and implementing structures. Thus governance needs to start at this *infor-*

mation/semantic layer, and then can be projected as appropriate.

7.5 Full Stack Semantic Software

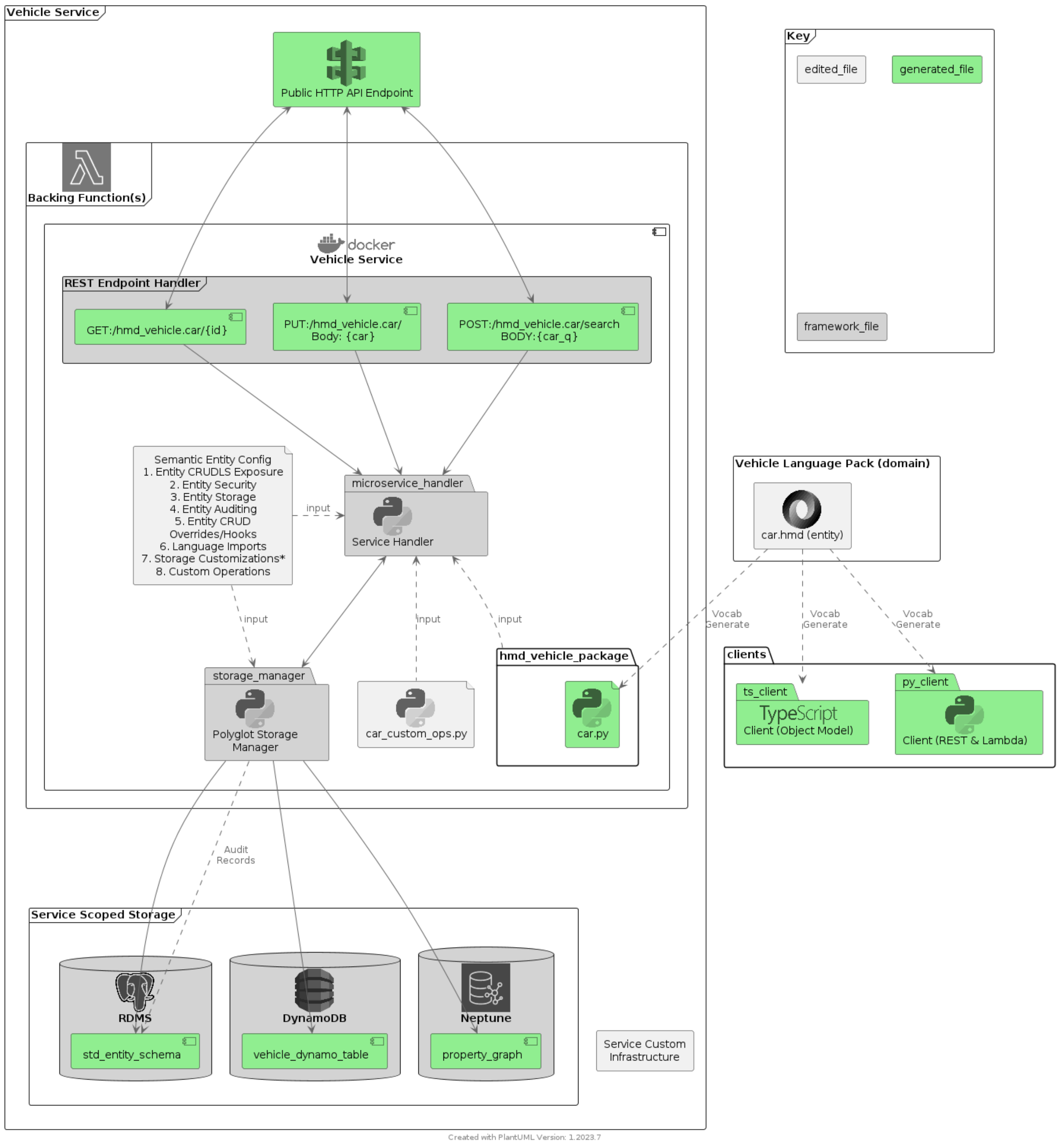
AKA Model Driven Architecture aka model-driven engineering.

What is a semantic service?

Given that the representation of data for transit, storage, query, and the constituent layers programmatic behavior are orthogonal, we define a semantic service as such:

- Data is defined outside of a programming language in an [data]IDL
- All configuration is done 'via the semantic', which is to say orthogonal aspects of the service behavior should be configured based on expressions related to the objects, relationships, and attributes' defined in the semantic language pack collection for that service.
  - API CRUDS & Custom Operation Exposure
  - Object & Custom Operation Security (RBAC)
  - Object Attribute-based security (ABAC)
  - Object Audit Trails
  - Object Storage configuration (local and global)
  - CRUDS Overrides
  - Custom Storage Implementation
  - Field masking
- The semantic objects in scope for a given service will affect their configuration on the service in 2 ways:
  - Generated code and configuration based on the semantic+concern config(s) and templates that are included in the service/passed to other frameworks
  - Use at runtime for code generation (generated CRUDLS queries based on the storage engine)
- The semantic service engine should offer a polymorphic storage framework that uses the property graph object model 'all the way through' to storage, with default CRUD implementation per storage engine.

An Example semantic service:



7.6 Semantic Event Transformation Architecture

Software is made up of *instructions* and *data*, *actions we want to happen*, and *things to be saved and analyzed*.

As often is the case, this simple bifurcation ignores or hides *time* and the ideas of parallel or asynchronous computation.

7.6.1 Concurrency and Asycnronicity

The processor in a computer executes instructions one after the other, in sequential order. (shush about PBE) Programmers create these instructions in the form of programs which are said to *run*, usually in the context of other software - an Operating System (OS).

The operating system manages all sorts of complexities: resource allocation, work coordination, and memory management, and re-abstracting these ideas in different variations has netted all manner of innovation in computing.

Work schedulers are common software at the enterprise level, and it's such a common problem that many pieces of software include some form of schedule management on their own, leading to multiple schedule managers. It usually doesn't take long to get to scheduling coordination issues.

At the lowest level, our simple mental model of a computer program is slightly inaccurate, in that computers are inherently multiple processes running in the context of the operating system, each with diffent thread models and coordination capabilities.

In the context of data transformation, each engine is a program, with multiple programs within, and there is zero interest in coordination with other programs. This leaves a fundamentally simple problem unsolved for the most common use case around - asynchronous data processing at scale in the cloud.

Enter the DAG.

7.6.2 Inversion of the Dag

Before we can talk about inverting a DAG, we should understand why they're so popular.

A DAG is a Directed Acyclic Graph, and a simple one looks like the following:

TODO: insert simple graph drawing

In more recent times and particularly in the data engineering space, DAGs have become synonymous with Apache Airflow's data orchestration abstraction.

Time as a series of objects in the graph.

7.7 Model-Driven Code Generation

Software development is filled with code generation, from compilers to web frameworks to sql optimizers, code generators are pervasive.

This is not intended to be an exhaustive review of code generators or techniques, but will present the basic ideas through more advanced features, demonstrated with the NeuronSphere code generation tool *hmd mickey*.

At its basic level, model-driven code generators are a class of software that take a structured *data context* (often structured text-based files), combines it with some string/text *templates*, and *renders* strings or text files as output.

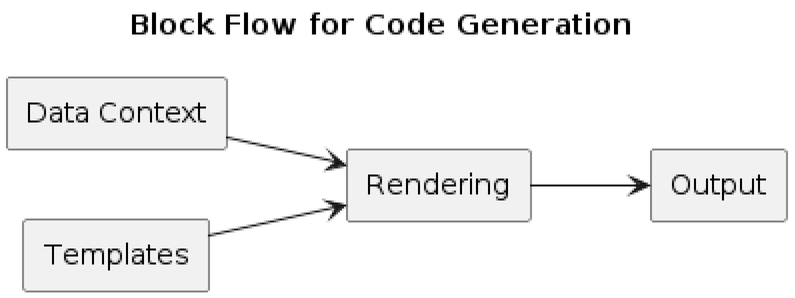


Fig. 1: Data Context + Templates = Output

Note: This document will limit the discussion of "code generators" to those taking "markup based text data" as an input class, specifically using json, yml, xml, csv, and other structured formats to model data and configuration. This specifically excludes code generation techniques that require advanced input parsing (e.g.: protobuf, programming languages)

If you want to build a parser and take the true DSL route, get ANTLR and get busy - it is incredibly rewarding in recent incarnations.

7.8 Customers and Orders - A Larger Example

Many of us will build small scripts & code generators, and this is a more illustrative albeit simple example. Note this example is simply added templates to the default example generated by *hmd mickey init*.

In this case we're going to use some simple *.json* metadata files and a few jinja templates to create simple table creation scripts as well as example plantUML drawings (.puml files)

Note: There is a growing set of tools that allow one to use simple markup to produce drawings and diagrams. A great tool in this space is plantUML, and the authors of mickey often output 'puml drawings' as part of code-generating documentation.

The examples will start with a straightforward set of files such as:

Listing 1: customer.json

```json
{
  "name": "customers",
  "columns": [
     { "name": "id", "datatype": "int" },
     { "name": "cust_name", "datatype": "varchar" },
     { "name": "cust_address", "datatype": "varchar" },
     { "name": "cust_city", "datatype": "varchar" },
     { "name": "cust_state", "datatype": "varchar" }
  ]
}
```

This simple representation obviously lines up with many basic tabular data structures, and the complete flow of input files to outputs looks like this:

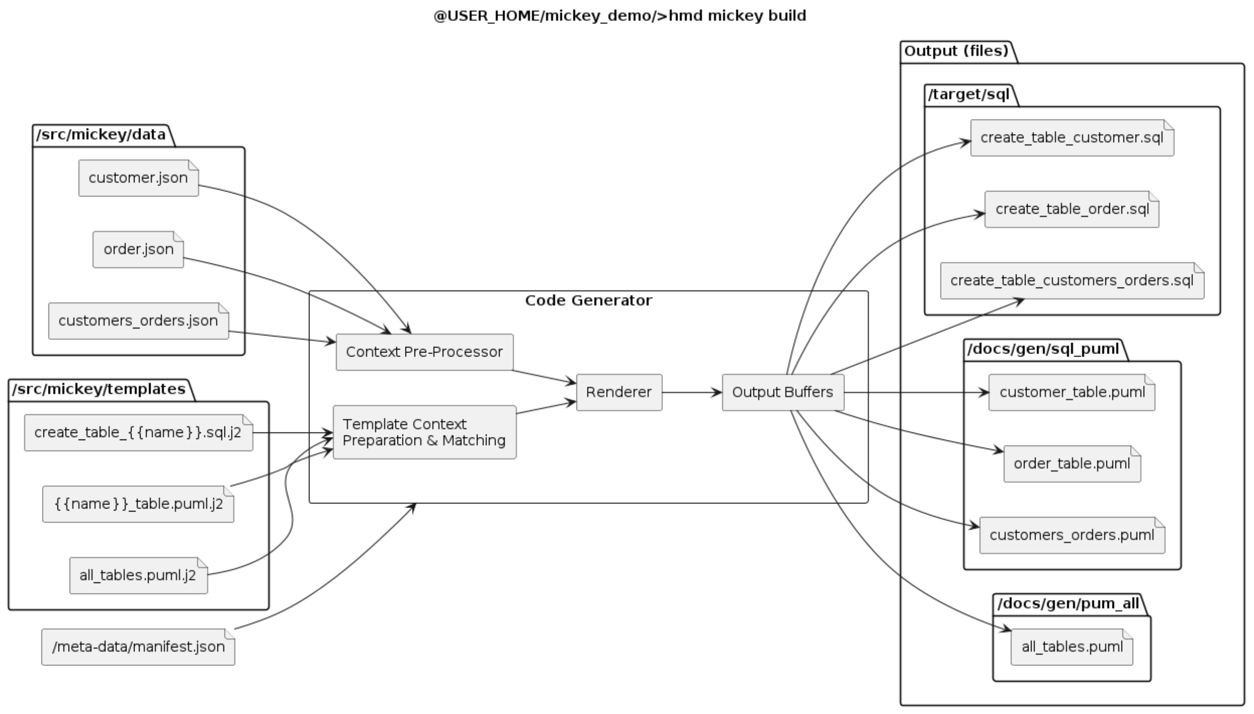


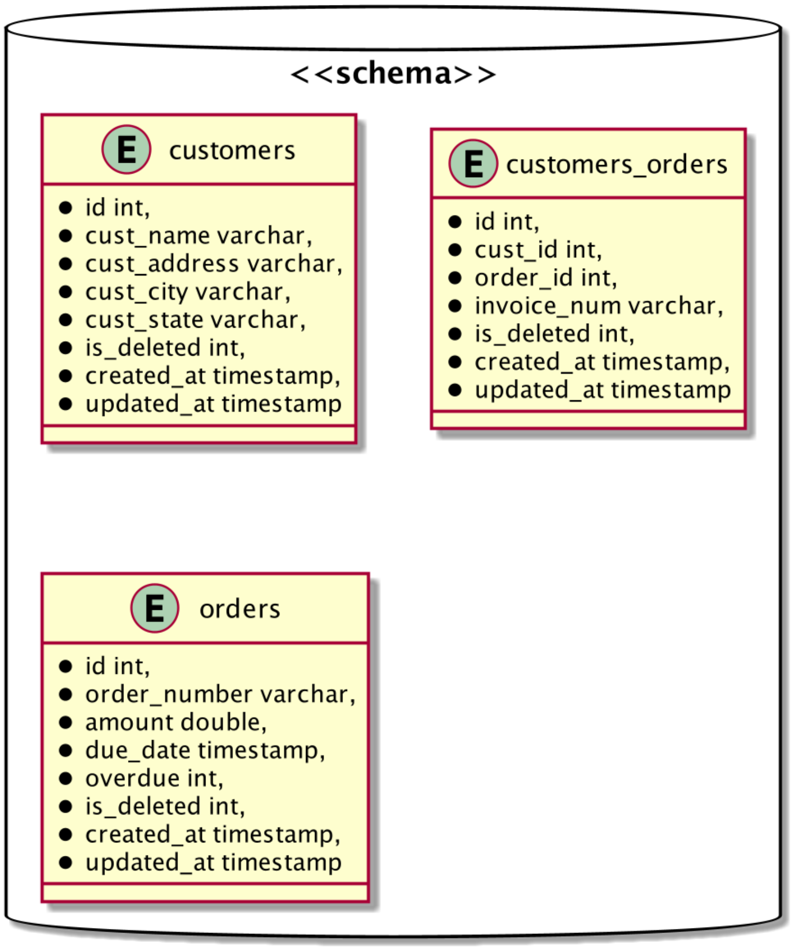
Fig. 2: File Based Code Generation Example

Assuming our sql template is making a simple table for persisting rows based on our model, we have table creation scripts that look like:

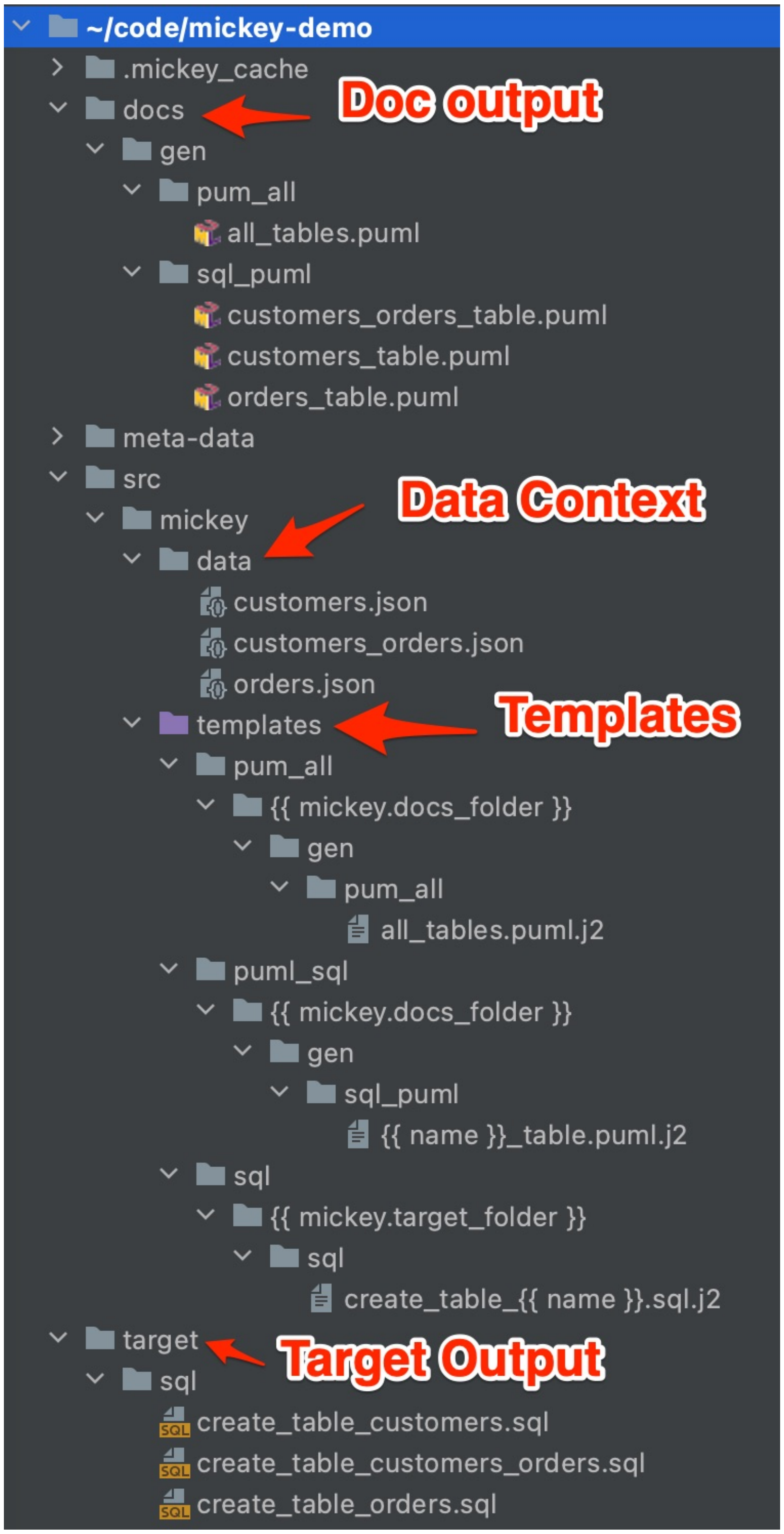
Listing 2: Basic create script

```
CREATE TABLE customers (
  id int,
  cust_name varchar,
  cust_address varchar,
  cust_city varchar,
  cust_state varchar
  );
```

The combination of input files into the all_tables.puml.j2 template produces:



In an IDE we can inspect the entire project directory structure and see the input data files, the templates, and the corresponding outputs.



```
~/code/mickey-demo
  .mickey_cache
  docs
    gen
      pum_all
        all_tables.puml
      sql_puml
        customers_orders_table.puml
        customers_table.puml
        orders_table.puml
  meta-data
  src
    mickey
      data
        customers.json
        customers_orders.json
        orders.json
      templates
        pum_all
          {{ mickey.docs_folder }}
            gen
              pum_all
                all_tables.puml.j2
        puml_sql
          {{ mickey.docs_folder }}
            gen
              sql_puml
                {{ name }}_table.puml.j2
        sql
          {{ mickey.target_folder }}
            sql
              create_table_{{ name }}.sql.j2
  target
    sql
      create_table_customers.sql
      create_table_customers_orders.sql
      create_table_orders.sql
```

7.9 Template and Data Reuse

The utility of a code generator is quite limited without a few other features - template and data context reuse being high on the list.

Why? Given our above example, let's assume we'd like to generate the same SQL for any number of data contexts and similarly generate drawings for other teams. Remote templates allow us to have a versioned single-source set of code generation logic that is reused across multiple projects.

Conversely, reuse of data contexts allows new and unforeseen combinations of our data models into new combinations for outputs. A good example of this is the *NeuronSphere Semantic Microservice framework* takes multiple external *Language Pack* references and produces a set of unified outputs for a service.

hmd mickey supports both template and data context reuse, and you may reference *NeuronSphere Artifacts* from a *NeuronSphere Librarian* or github repositories as remote contexts.

This example shows remote references for templates and data. Note that the remote data context reference excludes **orders**, allowing the user in this scenario to have their own copy of *customers* based on a shared master, but adds a 'local' data object to the context, creating a target set of entities related to customers & complaints rather than orders.

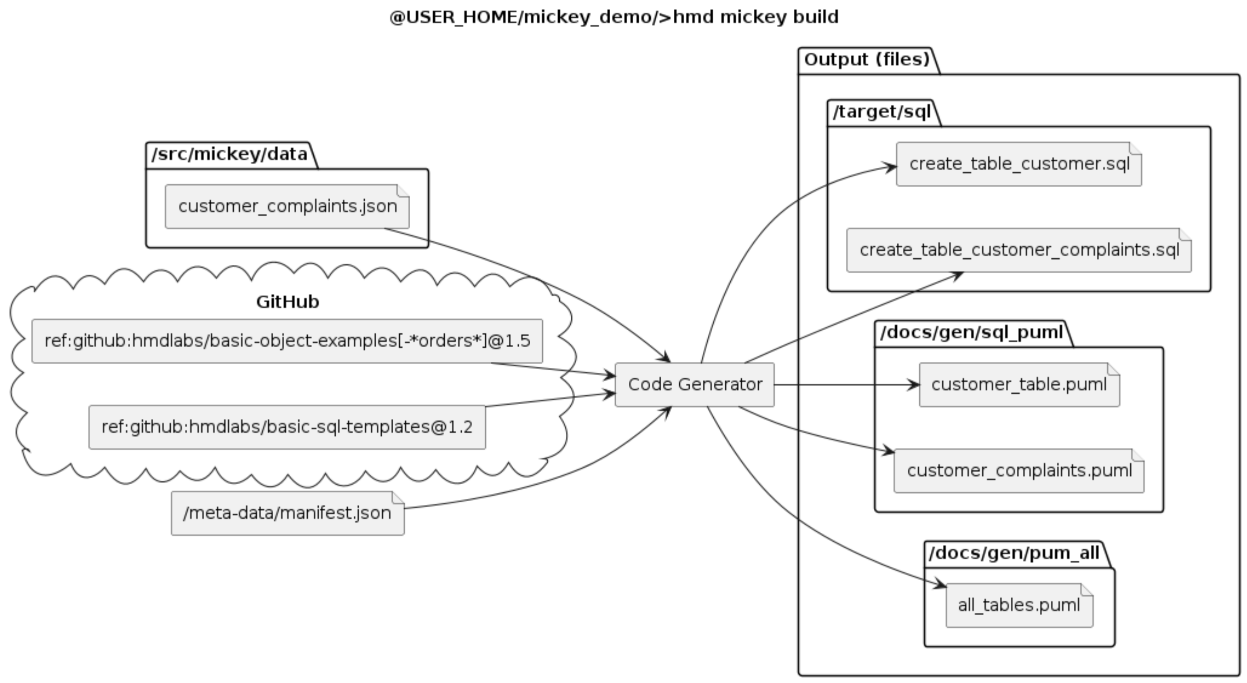


Fig. 3: Data and Template Reuse

7.10 Output Considerations

7.10.1 Output Locations for Generated Code

There exist strong opinions regarding never outputting generated code into the same directory structures as manually edited code, but for any number of technologies this presents more annoyance than the purity is worth.

The output locations of mickey templates are determined by the templates' author.

Template output will be in one of the following locations. Note that the relative folder that each of these is mapped to may be overridden by changing environment variables (convenient to re-target template output to an alternate repository layout).

- `{{mickey.docs_folder}}`: `/docs`
- `{{mickey.src_folder}}`: `/src`
- `{{mickey.target_folder}}`: `/target`
- `{{mickey.metadata_folder}}`: `/meta-data`

7.10.2 Source Code Control (SCM) Considerations

There is a simple rule - don't check generated files into source control. This implies that all code generation prior to a build must happen in the build environment (see CI/CD considerations).

Template authors should add .gitignore files to their template projects such that certain generated files are ignored, but this is difficult to perform universally.

In the event that you are not using CI/CD, or code generation is 'occasional' and based on a manual process trigger, (such as awareness of a version change of an external data model) you may find yourself checking in generated code. It's not the end of the world, but is technical debt to be tracked and eliminated.

7.10.3 CI/CD considerations

Ideally, you're using modern CI/CD practices and trunk-based development, such that a build happens when you push code (to trunk/main). In this case, it is preferable to have the CI system automatically run the code generator against the repository to be built, after checkout and before any build activities.

This is up to your build tooling & process, and different tools take different approaches to solving these problems.

7.11 Advanced - Overrides, Defaults, Mix-Ins

One of the great benefits of code generation is the automation of updating multiple outputs with the change to a simple & single input, in our example a simple tabular entity model.

The power of code generators quickly surfaces a key drawback - like a WYSIWYG user interface, what you see is not just what you get, but often all you get. For code generators, this often translates to a direct binding and limitation between input models and the outputs that are possible.

Suppose we have centered around that simple model, and we'd like to experiment with different storage and data analytics modelling techniques. Specifically we'd like to use our existing tabular models for Customer and Order, but rather than a simple table designed to store only the current state, we'd like to entertain the following use cases:

1. When manifested in a relational database, we'd like to add standard columns to all entities in the data model(s)
2. Build simple python classes to represent the tables
3. Manifest all entities in the model using Anchor Modelling
4. Manifest customer entity as a SCD using Dimensional Modelling
5. Manifest all entities in the model using Data Vault Modelling

All of these use cases require an understanding of the target artifacts, their conventions, and new templates to populate. Critically, each potentially(likely?) requires additional metadata be added to our models.

Practically this often means the specifics of the intended output(s) 'leak into' the abstraction that is the input model. In most cases the input models are specifically designed and constrained for a known set of outputs and their requisite metadata. While this is expedient, it can also limit the re-use of the data model and in many cases the *entire* code-generator.

Said another way, most code generating tools focus on a specific input and output, hiding the details of the generator and critically hampering extensibility. With the right code generating capabilities and a few advanced features & ideas, we can instead create an extensible and effective meta-model based Model Driven Development chain, and potentially Model Driven Architecture.

mickey provides a 'data context preprocessor' that allows orthogonal and vertical augmentation of the input data models, supporting all of the above use cases using several strategies. These will be elaborated with examples in a future version of this document.

7.12 Advanced - Using Intermediate Models for Leverage

The power of a code generator is partially in the technology choices and a few identifiable features of the sub-components, but largely is defined by the quality or diversity of the input model(s) and the outputs from available templates.

7.12.1 IDLs and code generation

In many cases, broad agreement on a well-designed model is the major focus, with a bevy of tools and code generators all built to take that input and produce a known set of outputs. A canonical use case where code generation excels is IDLs such as OPEN API. As IDLs are designed to facilitate interprocess communication (RPC or otherwise), they often combine a few key ideas:

1. A type system for primitives and collections
2. A system for declaring structures of primitives (and/or structures)
3. A system for declaring 'interface operations' with their parameter, return, and error behaviors

Using OpenAPI as example, we see entire toolchains in multiple languages designed to produce any number of different source code languages and styles. It is incredibly effective, yet many IDL systems suffer a similar concern - their type system is "too rich", often owing to their primary purpose of designing transactional data objects to transit across a network.

On one hand the richness of the type and structure system solves the RPC declaration problem well, however it leaves a data model that is not tied to the rest of the application architecture or data model(s), and often isn't useful for describing storage or physical data requirements. Protobuf suffers similarly, making the creation of 'data objects and services' very easy while leaving storage and analytics use cases far out of scope.

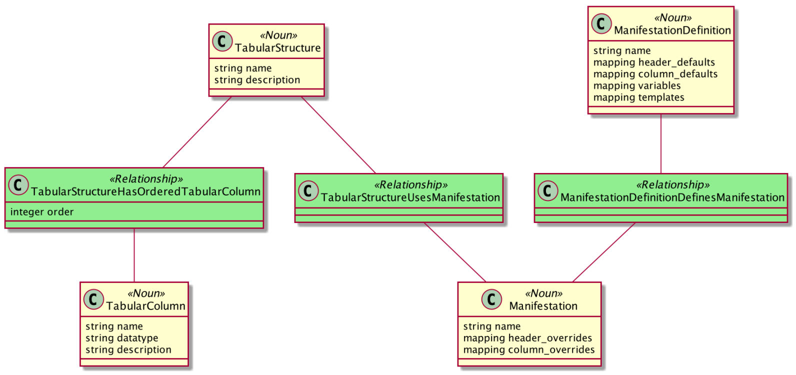
Inadvertently, this often surfaces an old challenge - the "object to relational impedance mismatch (ORIM)". At the same time that a new generation of IDLs was replacing WSDL and CORBA, "noSQL" technologies also saw an uptick in capability and usage. Whether a key-value store or a document database, a new generation of developers and application architectures became pervasive, somewhat solving the ORIM challenge but just as often creating analytics challenges and "kicking data governance" far down the line.

This document will not provide exhaustive coverage of various approaches for interchange with different IDLs, but will focus on a few key examples with assumptions, demonstrating larger patterns and ideas for integration of mickey with other IDLs.

7.12.2 Relevant NeuronSphere Ideas

From the section on NeuronSphere Semantic Metamodel - The Entity Graph, recall the following diagram that shows an overview of the 2 layer NeuronSphere semantic model.

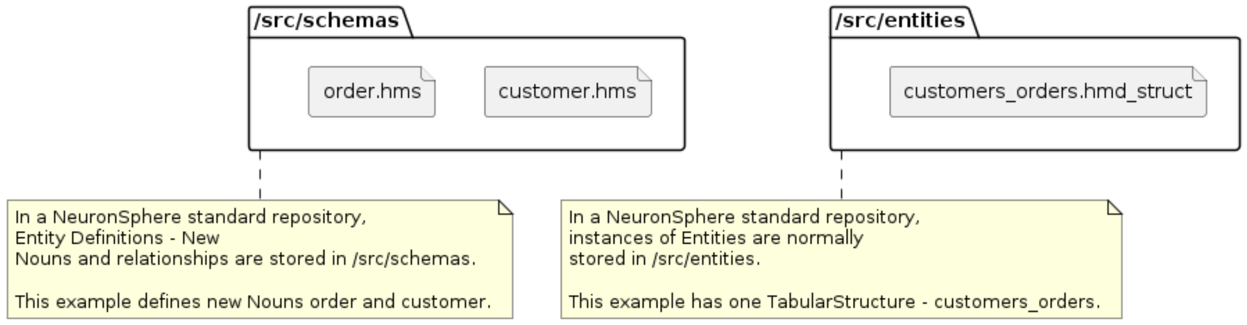
A simple set of Entities from the hmd-lang-structure language pack, rendered with the NeuronSphere "vocab generate" command:



Note: The entities from the hmd-lang-structure language pack will primarily be used as 'data contexts' for code generation in this document, ignoring the larger usages around data lineage and governance that *TabularStructure* and it's related structures fulfill in the larger NeuronSphere ecosystem.

7.12.3 A "More Complete" NeuronSphere Example

Continuing with the logical example of Customer, Order, and Customers_Orders, we will model 2 of them as NeuronSphere entities (nouns in this case), and one as a *TabularStructure struct* (containing both table and column metadata in a single file).



In a NeuronSphere standard repository,
Entity Definitions - New
Nouns and relationships are stored in /src/schemas.

This example defines new Nouns order and customer.

In a NeuronSphere standard repository,
instances of Entities are normally
stored in /src/entities.

This example has one TabularStructure - customers_orders.

Fig. 4: Example Entities and TabularStructure

Microservice and Persistence Artifacts

With the entities, we want to use a collection of templates that will create the various libraries used by the NeuronSphere MicroService & Persistence frameworks.

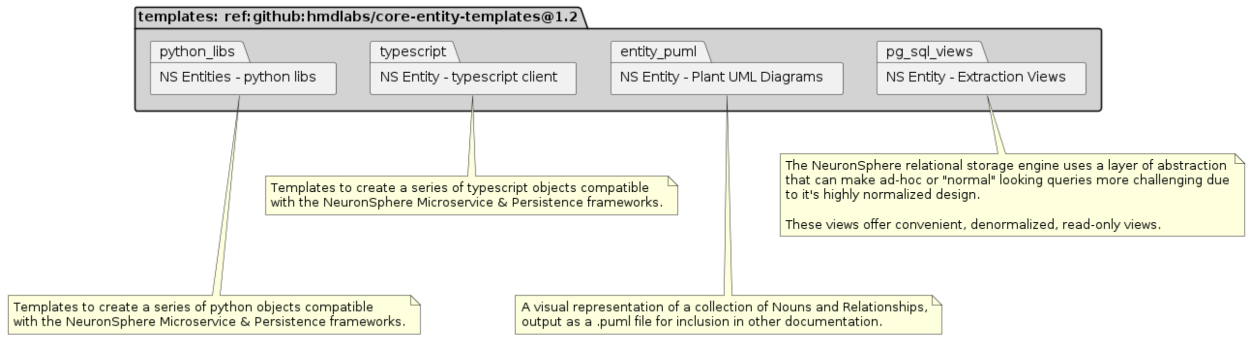

templates: ref:github:hmdlabs/core-entity-templates@1.2
python_libs
NS Entities - python libs
typescript
NS Entity - typescript client
entity_puml
NS Entity - Plant UML Diagrams
pg_sql_views
NS Entity - Extraction Views
Templates to create a series of typescript objects compatible with the NeuronSphere Microservice & Persistence frameworks.
The NeuronSphere relational storage engine uses a layer of abstraction that can make ad-hoc or "normal" looking queries more challenging due to it's highly normalized design.
These views offer convenient, denormalized, read-only views.
Templates to create a series of python objects compatible with the NeuronSphere Microservice & Persistence frameworks.
A visual representation of a collection of Nouns and Relationships, output as a .puml file for inclusion in other documentation.

Fig. 5: Entity Templates

Looking at the end to end flow for generating the MS&P artifacts as well as the Typescript clients and the documentation pieces, we have this flow:

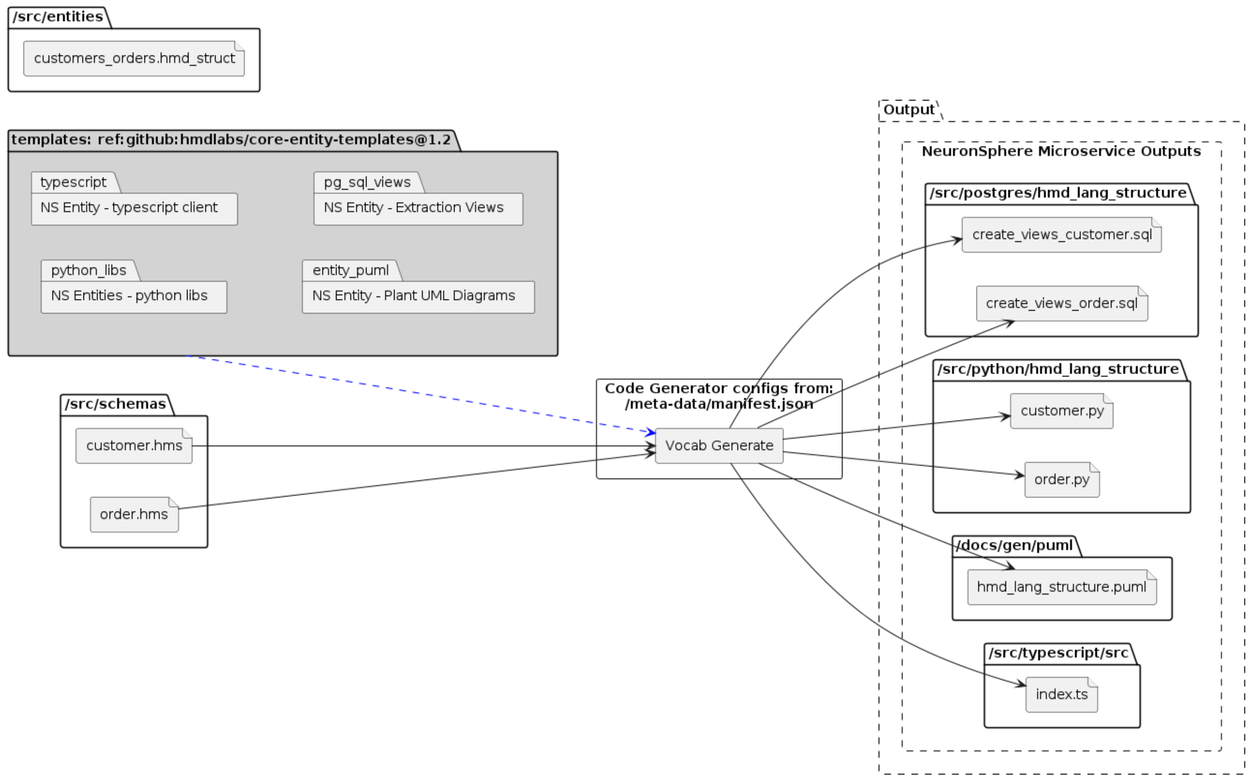

/src/entities
customers_orders.hmd_struct
templates: ref:github:hmdlabs/core-entity-templates@1.2
typescript
NS Entity - typescript client
pg_sql_views
NS Entity - Extraction Views
python_libs
NS Entities - python libs
entity_puml
NS Entity - Plant UML Diagrams
/src/schemas
customer.hms
order.hms
Code Generator configs from: /meta-data/manifest.json
Vocab Generate
Output
NeuronSphere Microservice Outputs
/src/postgres/hmd_lang_structure
create_views_customer.sql
create_views_order.sql
/src/python/hmd_lang_structure
customer.py
order.py
/docs/gen/puml
hmd_lang_structure.puml
/src/typescript/src
index.ts

Fig. 6: Full Entity Codegen

TabularStructure Load from external partitions

With the Customer and Order Entities available at the microservice/application layer, we need to generate a set of concrete structures to process the 'big data' pipeline for customer_orders in our example. For this we'll use a package of templates designed to perform a basic data ingestion from raw text files to compressed parquet files for a given *TabularStructure* using the popular Trino compute engine.

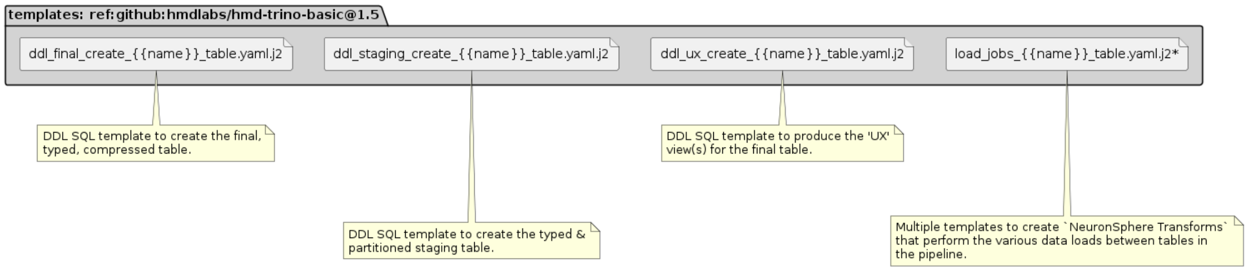

templates: ref:github:hmdlabs/hmd-trino-basic@1.5
ddl_final_create_{{name}}_table.yaml.j2
ddl_staging_create_{{name}}_table.yaml.j2
ddl_ux_create_{{name}}_table.yaml.j2
load_jobs_{{name}}_table.yaml.j2*
DDL SQL template to create the final, typed, compressed table.
DDL SQL template to produce the 'UX' view(s) for the final table.
DDL SQL template to create the typed & partitioned staging table.
Multiple templates to create `NeuronSphere Transforms` that perform the various data loads between tables in the pipeline.

Fig. 7: Basic Trino Loading Templates

Applying these templates to the customer_order structure produces the following flow:

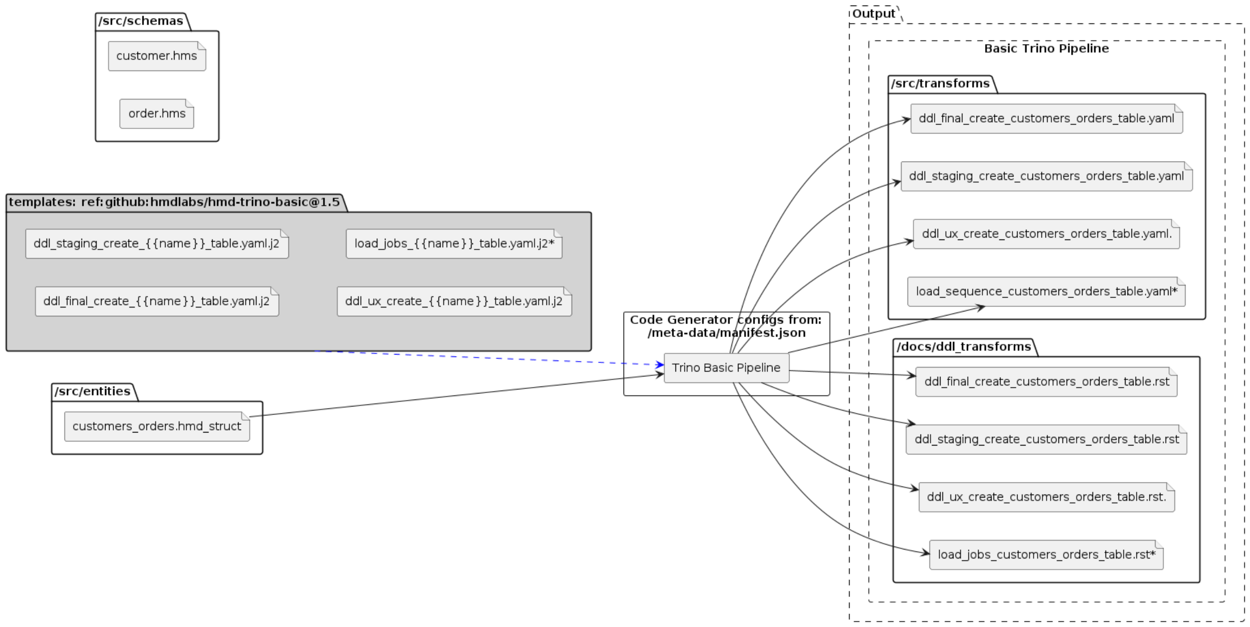

/src/schemas
customer.hms
order.hms
templates: ref:github:hmdlabs/hmd-trino-basic@1.5
ddl_staging_create_{{name}}_table.yaml.j2
load_jobs_{{name}}_table.yaml.j2*
ddl_final_create_{{name}}_table.yaml.j2
ddl_ux_create_{{name}}_table.yaml.j2
/src/entities
customers_orders.hmd_struct
Code Generator configs from: /meta-data/manifest.json
Trino Basic Pipeline
Output
Basic Trino Pipeline
/src/transforms
ddl_final_create_customers_orders_table.yaml
ddl_staging_create_customers_orders_table.yaml
ddl_ux_create_customers_orders_table.yaml.
load_sequence_customers_orders_table.yaml*
/docs/ddl_transforms
ddl_final_create_customers_orders_table.rst
ddl_staging_create_customers_orders_table.rst
ddl_ux_create_customers_orders_table.rst.
load_jobs_customers_orders_table.rst*

Fig. 8: Full Tabular Structure Codegen

Using an Entity as an SCD

After some consideration, it is determined that we need the Customer entity to also be materialized as table in our data warehouse, specifically we'd like to model it as a Slowly Changing Dimension (SCD).

This is a common use case - the demand to maintain an entity via an API in front of a 'domain microservice' such as surfaced via the NeuronSphere MicroService framework, yet maintain it's history in an SCD for analytics use cases in both a relational and 'big data' systems.

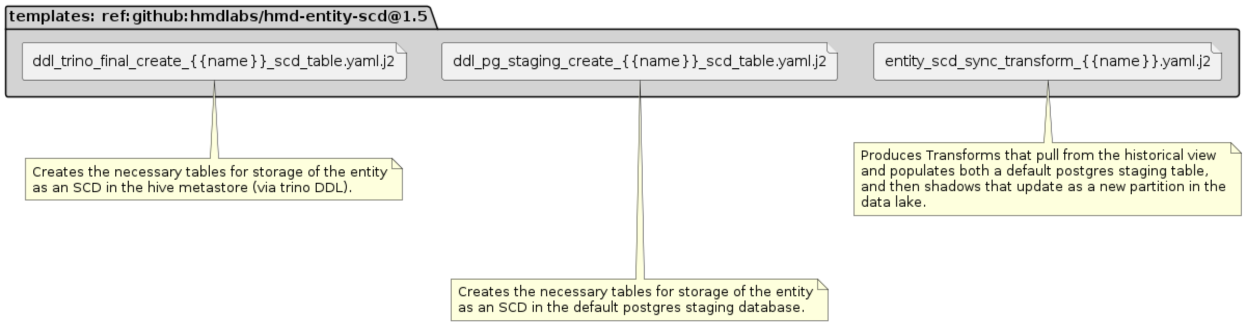

templates: ref:github:hmdlabs/hmd-entity-scd@1.5
ddl_trino_final_create_{{name}}_scd_table.yaml.j2
ddl_pg_staging_create_{{name}}_scd_table.yaml.j2
entity_scd_sync_transform_{{name}}.yaml.j2
Creates the necessary tables for storage of the entity as an SCD in the hive metastore (via trino DDL).
Produces Transforms that pull from the historical view and populates both a default postgres staging table, and then shadows that update as a new partition in the data lake.
Creates the necessary tables for storage of the entity as an SCD in the default postgres staging database.

Fig. 9: SCD Templates

The SCD templates also use the Customer entity in this use case, producing this data flow:

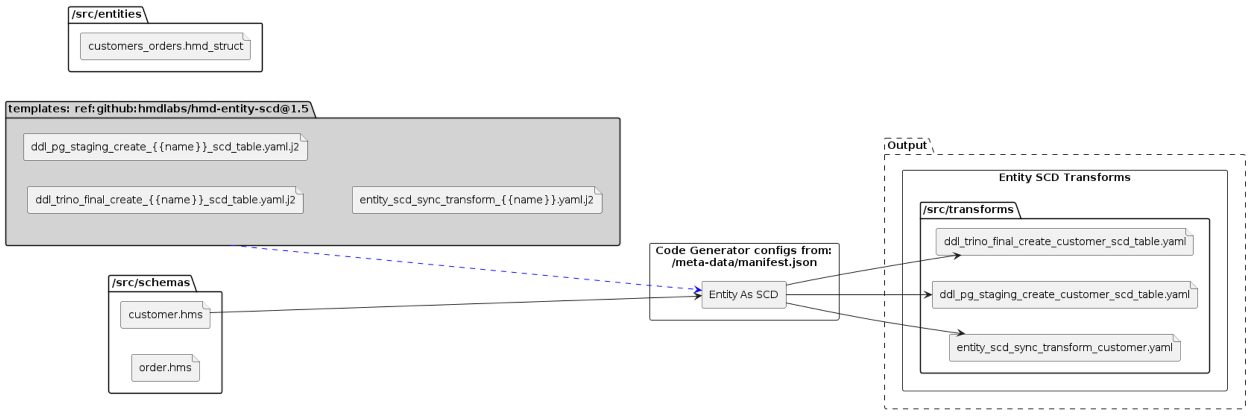

/src/entities
customers_orders.hmd_struct
templates: ref:github:hmdlabs/hmd-entity-scd@1.5
ddl_pg_staging_create_{{name}}_scd_table.yaml.j2
ddl_trino_final_create_{{name}}_scd_table.yaml.j2
entity_scd_sync_transform_{{name}}.yaml.j2
/src/schemas
customer.hms
order.hms
Code Generator configs from: /meta-data/manifest.json
Entity As SCD
Output
Entity SCD Transforms
/src/transforms
ddl_trino_final_create_customer_scd_table.yaml
ddl_pg_staging_create_customer_scd_table.yaml
entity_scd_sync_transform_customer.yaml

Fig. 10: SCD from Customer Entity CodeGen

7.12.4 Complete Example

Combining the previous sections, we start to see the scale of possibilities enabled by these structures, reusable templates, and a good code generating framework.

In this example, authoring 3 small files - 2 HMD entities and 1 tabular structure, results in:

- python and typescipt libraries to use in the NeuronSphere MicroService & Persistence framework. Author Entities -> Configure Microservice -> Push code and see full pipeline deployment of a CRUDLS service for those entities.
- Visual documentation about the entity model for use in documentation, and documentation about the sql views used to access the entity persistence for export.
- sql to run against trino (via the NeuronSphere Transform service) to ingest from external files to internal parquet-backed tables.
- sql for both postgres and trino to create SCD dimension tables & structures, as well as NeuronSphere Transforms to routinely update the SCDs from the entity service.

Less than 200 lines of data models as input turns into thousands of lines of cross-technology platform & pipeline code, with substantial sections of generated documentation and lineage kept up to date along with the code & build pipelines.

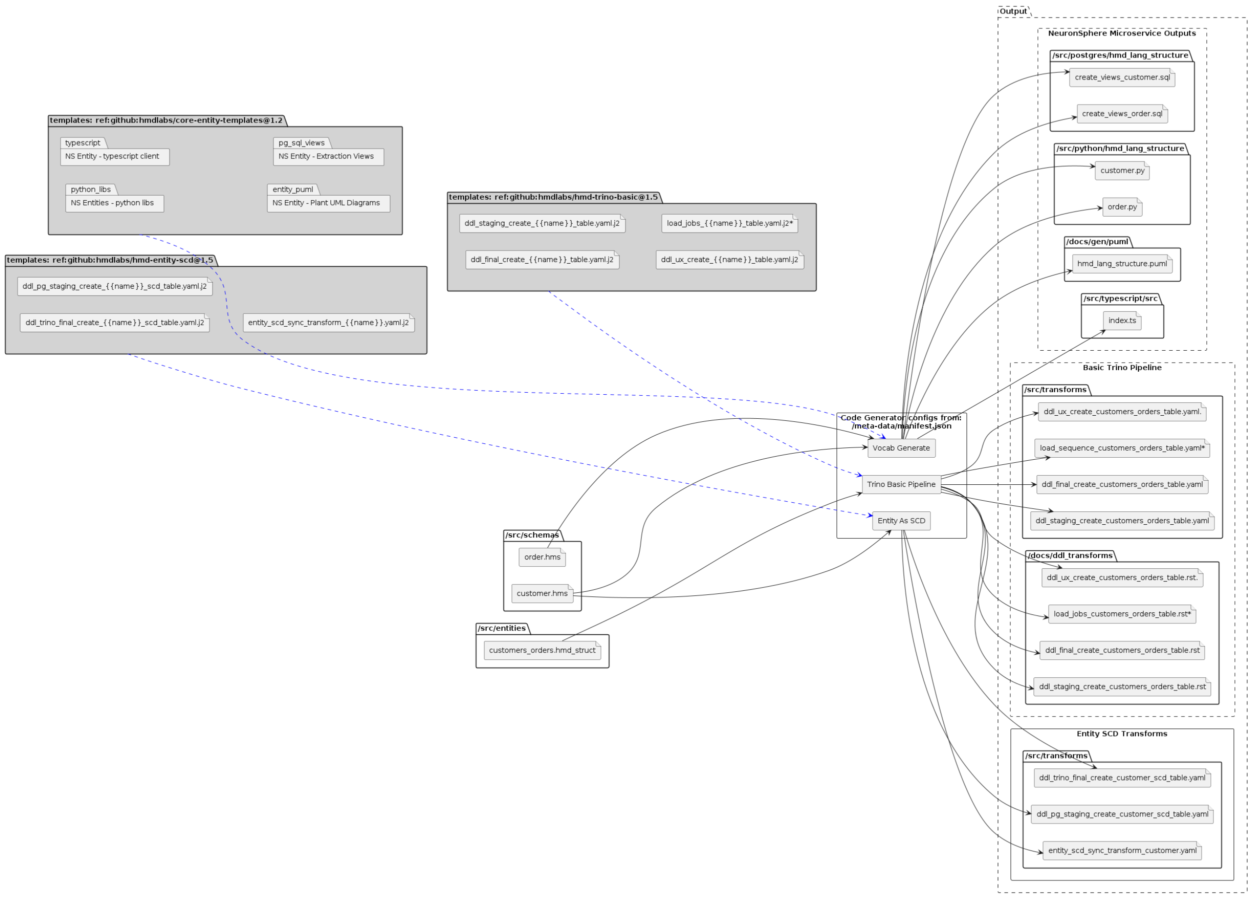

templates: ref:github:hmdlabs/core-entity-templates@1.2
typescript
NS Entity - typescript client
pg_sql_views
NS Entity - Extraction Views
python_libs
NS Entities - python libs
entity_puml
NS Entity - Plant UML Diagrams
templates: ref:github:hmdlabs/hmd-entity-scd@1.5
ddl_pg_staging_create_{{name}}_scd_table.yaml.j2
ddl_trino_final_create_{{name}}_scd_table.yaml.j2
entity_scd_sync_transform_{{name}}.yaml.j2
templates: ref:github:hmdlabs/hmd-trino-basic@1.5
ddl_staging_create_{{name}}_table.yaml.j2
load_jobs_{{name}}_table.yaml.j2*
ddl_final_create_{{name}}_table.yaml.j2
ddl_ux_create_{{name}}_table.yaml.j2
/src/schemas
order.hms
customer.hms
/src/entities
customers_orders.hmd_struct
Code Generator: configs from: /meta-data/manifest.json
Vocab Generate
Trino Basic Pipeline
Entity As SCD
Output
NeuronSphere Microservice Outputs
/src/postgres/hmd_lang_structure
create_views_customer.sql
create_views_order.sql
/src/python/hmd_lang_structure
customer.py
order.py
/docs/gen/puml
hmd_lang_structure.puml
/src/typescript/src
index.ts
Basic Trino Pipeline
/src/transforms
ddl_ux_create_customers_orders_table.yaml
load_sequence_customers_orders_table.yaml*
ddl_final_create_customers_orders_table.yaml
ddl_staging_create_customers_orders_table.yaml
/docs/ddl_transforms
ddl_ux_create_customers_orders_table.rst
load_jobs_customers_orders_table.rst*
ddl_final_create_customers_orders_table.rst
ddl_staging_create_customers_orders_table.rst
Entity SCD Transforms
/src/transforms
ddl_trino_final_create_customer_scd_table.yaml
ddl_pg_staging_create_customer_scd_table.yaml
entity_scd_sync_transform_customer.yaml

Fig. 11: Complete Entity and Tabular Structure CodeGen

Now the question is - what do we do with all that generated code? We not only use these techniques to build NeuronSphere services, but then use those services as a framework to deliver our array of metadata-driven data products.

7.13 Graphical Modeling

Graphical code generators are *really* neat.

WYSIWYG though, and you only get what the UI author allows you to see.

Because of this, they tend to be tightly bound to the input model assumptions, limiting their use.

Because of the limited usability, they will often lack the more advanced features found in a generic multi-layered code-generator like *hmd-cli-mickey*.

Because of Mickey's functionality, it is the ideal code generation engine to put behind N user interfaces with their constituent constrained and more complex models.

If you want to build multiple user interfaces that ease generating complex code outputs - create a UI that outputs the structured text you want and then delegate the complexities of code generation and template management to *mickey*.

CHAPTER

EIGHT

CLI IMPLEMENTATIONS

The previous sections have all been about ideas and proposals, frameworks & theories about how to create a more unified yet loosely-coupled development, data modeling, testing, and deployment experience for developers of all kinds of specialties.

This section largely *verbatim includes* the documentation from the individual NeuronSphere CLI tools and current extensions.

Note: Some of these extensions are released as public binaries to pypi and dockerhub, others are still internal tools. TODO: see also public releases repo for more tracking information on detailed status.

8.1 Installing Default NeuronSphere Tools

Requirements:

- python 3.9 (not 3.10, not yet)
- docker 21+

In the section on release management, if you squint, you can see in the dependency diagram that there is a large collections of dependencies on the python package *neuronsphere*. This package exists soley as a *distribution collector*, as the prerequisites that it installs will comprise all of the required *local NeuronSphere* packages and utilities.

```
pip install neuronsphere
```

To give you a sense of the minimal packages that make up the local NeuronSphere, these are the declared dependencies as of 2023-07-13:

```
hmd-cli-app~=1.1.595
hmd-cli-tools~=1.1.228

hmd-cli-configure==1.0.38
hmd-cli-mickey==1.0.41
hmd-cli-repo==0.2.118
hmd-cli-neuronsphere==0.2.114

hmd-cli-build==0.1.53
hmd-cli-python==0.2.75
hmd-cli-docker==0.1.117
hmd-cli-transform-deploy==0.1.56
hmd-cli-explorer==0.1.28
```

(continues on next page)

(continued from previous page)

```
hmd-cli-bartleby==0.1.35
hmd-cli-bender==0.1.24
```

8.2 Local Environment Setup

All NeuronSphere CLI tools and the Local NeuronSphere operate in the context of several *environment variables*, HMD_HOME and HMD_REPO_HOME.

When you run the *hmd configure* command, you will be promted for a temporary HMD_HOME to use for that session, and will be reminded to set it in your session and in your shell for future ease of use.

8.2.1 HMD_HOME

HMD_HOME is an environment variable that points to the root directory used to store NeuronSphere configuration and data.

If you are working with a single NeuronSphere cloud root environment, you will likely have one HMD_HOME set in your bash profile or the like.

As a consultant or someone using NeuronSphere for professional work and local learning and exploration, you can have multiple HMD_HOMEs set up on your machine, simply change the environment variable for the shell to use the alternate configuration.

8.2.2 HMD_REPO_HOME

HMD_REPO_HOME points to the directory where a collection of NeuronSphere compliant source code repositories are housed as a colleciton of peers. Note that HMD_REPO_HOME is often set in hmd.config as a subdirectory of HMD_HOME, so you may not need to explicitly set it.

8.2.3 HMD Config file

What's it called? What's in it?

Commands to dump/edit, config command interactions

Show a current default example

8.2.4 Setting up a venv

It is recommended to set up a python virtual environment per HMD_HOME, and activate it in your shell profile for ease of use.

TODO: add venv options and details.

8.3 NeuronSphere Docker-Based CLI?

Docker is awesome at putting a bunch of dependencies into a nice package that's (largely) portable.

It started (and still) spends *much* of it's time as a technology providing *port-based services*, which is to say that many containers are designed to be stateless, relatively long-lived, and provide their value via interactions over some network protocol/port like https:443 or http:80 or ssh:23.

Many pieces of software however, are also distributing containers that are designed to act upon a mounted file system and produce some other actions or file-based outputs.

Github actions follow this model, wherein an *action* is a container that runs having mounted a checked-out version of a source-code repository, often *leaving artifacts* in the mounted directory that are the results of the container's work. Many containers will often *call home*, making external API calls, storing the information of their work in some remote system.

This alternative model for using containers is powerful, but comes with a substantial catch.

Launching a docker container with mounted filesystems, mapped environment variables, and all the proper parameters can be very annoying. You end up with long docker commands, often saved into all manner of different shell scripts, and it's messy.

The NeuronSphere CLI framework is 2 key pieces - the extensible python CLI framework that allows you to write plugins, and the docker-running abstraction layer and default containers that allow you to package much of your work into a container, greatly reducing the installation requirements on CLI users.

An example of this is the cli *hmd bartleby*, which we use to render documents and diagrams as code. This cli uses Sphinx-doc behind the scenes, with a number of extensions and customizations defined. Rather than forcing the users of the CLI to have java and a number of dependencies installed, all of those are installed and configured in the bartleby container. The bartleby CLI provides an easy to use user-interface, and handles mapping the user commands and current working environment and directories into the complex docker command that the container requires to execute.

There are a number of other CLIs in the stack that work this way, and it is a powerful abstraction in the ability to deliver governed and controlled tools at scale.

8.4 CLI - SDLC, CI/CDD, DevOps Considerations

TODO * How the CLI tools are used in the local development workflow * Tooling for 'local' deployments/releases * Containerizing the CLIs for use in CI/CD pipelines

8.5 hmd-cli-configure

This package is a command line utility for configuring your local environment to use other HMD CLI tools, it will prepare a new HMD_HOME, and should be used to initialize any new HMD_HOME.

All configuration for HMD CLI tools is done via a dotenv file located at *$HMD_HOME/.config/hmd.env*.

The path *$HMD_HOME* must be set by you to a directory that will be used with HMD tools only, and be present in your session. It is a good idea to set this in a *.bash_profile* or similar file.

Usage

In order to set an environment variable in the *hmd.env* file, you would use the *set-env* command and provide *key* and *value* arguments.

```
` hmd configure set-env <key> <value> `
```

This will update or append the following line in *hmd.env*: *key='value'*.

## Common Variables

There are some common variables that you might want to set upon first installation.

- *HMD_GH_USERNAME*: the GitHub username to use when connecting; primarily used by *hmd-cli-mickey* to fetch remote templates
- *HMD_GH_PASSWORD*: either a GitHub PAT or absolute path to file containing only the PAT. The entire file contents will be read and used.

8.6 hmd-cli-repo

8.6.1 hmd-cli-repo README

A CLI application for creating NeuronSphere-compliant project repositories from templates. This tool is also the base for multi-repository operations.

- Additional Requirements
- Git

## Basic Usage

hmd repo create will provide prompts to choose which Project Type Category and then Project Type you want to create. After having selected a Project Type, you will be prompted for any additional information need to create the project, i.e. a name for the project.

After all prompts have been answered, it will render a series of templates into a folder with the Project Name. This will be a standard Git repository and all generated files will be committed for you. Depending on configuration, this may create a remote repository and push the initial code as well.

Commands

- **hmd repo create**: create a NeuronSphere-compliant repository based on chosen Project Type
- **hmd repo add**: overlay a new Project Type onto an existing repository in a new branch
- **hmd repo configure**: configures hmd-cli-repo specific environment variables in *$HMD_HOME/.config/hmd.env*
- **hmd repo pull-all**: pull all changes and new repositories from a GitHub organization locally into *$HMD_REPO_HOME*
- **hmd repo create-remote**: create a remote repository for an existing local repo, will set *origin* remote in Git to new remote.

TODO: more descriptions and examples of these commands

Environment Variables

This command must be run with a valid HMD_HOME set and configured.

Optional Variables

- HMD_REPO_ORG_NAME: a common org code prefix to prepend to all create repo names.
- HMD_REPO_REMOTE_GITHUB_ENABLED: if set to "true", a remote repository will be created in GitHub upon running *hmd repo create*
- HMD_GH_USERNAME: the GitHub username to use
- HMD_GH_PASSWORD: the GitHub PAT to use

- HMD_GH_ORG_NAME: the GitHub Organization to create the repository in

8.6.2 Project Types and Templates

The `hmd repo` CLI uses two different constructs for generating NeuronSphere compliant projects, `ProjectType` and `ProjectTemplateSet`. A `ProjectType` is a named collection of `ProjectTemplateSet` objects. A `ProjectTemplateSet` is a directory containing a valid Cookiecutter config file and templates. The `cookiecuttter.json` context files in each `ProjectTemplateSet` listed in a `ProjectType` will be merged together in the order listed. The final context object will be used for all calls Cookiecutter when generating the project.

`ProjectType` configuration object:

```
{
  "name": "<Name of ProjectType>",
  "templates": [
    {
      "name": "<cookiecutter folder name>"
    }
  ]
}
```

Each `ProjectTemplateSet` must have a unique `name`. In addition to the templates listed, a default Cookiecutter will be run first to ensure proper folders exist. This default Cookiecutter is included in `hmd repo` and does not need to be explicitly listed. The `hmd repo` package also contains a number of standard `ProjectTemplates`.

Create Project Templates

A NeuronSphere `ProjectTemplateSet` is nothing more than a Cookiecutter directory that conforms to standards of a NeuronSphere compliant repository. For more information about Cookiecutter, visit their website. A generica Cookiecutter contains one or more Jinja templates to render, and a `cookiecutter.json` file defining variables to use in the templates. A NeuronSphere `ProjectTemplateSet` must place the templates and `cookiecutter.json` file in a folder with the `ProjectTemplateSet` name. Furthermore, Cookiecutter allows Jinja style templating in file paths, and all templates that must be rendered into the project must live in a folder called `{{ cookiecutter.repo_name }}`, and it must be a sibling to `cookiecutter.json`. By standardizing on the Cookiecutter folder name, we can support layering different `ProjectTemplateSets` onto the same project without causing files to be rendered outside the project.

The `cookiecutter.json` file contains any variables and their default values that can be used in the Jinja templates. You can access the variables in templates like so, `{{ cookiecutter.<var name> }}`, where `<var name>` is the key in the JSON object in `cookiecutter.json`. Upon running `hmd repo create`, after choosing a `ProjectType`, all the `cookiecutter.json` files will be merged together and you will prompted to enter values for each key. Therefore, it is the responsibilty of both the `ProjectTemplateSet` author and `ProjectType` creator to ensure unique keys where necessary. Each `ProjectTemplateSet` listed in the `ProjectType` will be rendered with the same merged variables. In addition, `hmd repo create` and the `default_repo` provides some standard variables that can always be referenced without explicitly listing in the `cookiecutter.json` directly.

```
{
  "_org": "<HMD_REPO_ORG environment variable>",
  "_repo_type": "<defaulted in some other template sets>",
  "short_name": "",
  "project_name": "{{ cookiecutter._org|lower|replace(' ', '-') }}{{ cookiecutter._repo_
↪type|lower|replace(' ', '-') }}{{ cookiecutter.short_name|lower|replace(' ', '-') }}",
  "repo_name": "{{ cookiecutter.project_name|lower }}",
```

(continues on next page)

(continued from previous page)

```
  "description": "",
  "_author": "",
  "_author_email": ""
}
```

Notice that certain variables default to a Jinja template string referencing other varibles. The first two variables `_org` and `_repo_type` are special. You will not be prompted for them, because the `_` prefix. The `_org` variable defaults to the environment variable `HMD_REPO_ORG`. It is intended to allow organizations to enforce a global prefix on all repository names. The `_repo_type` does not get defaulted. However, many of the included `ProjectTemplateSets` override it to provide a short code designating the repository type. For example, the `Microservice` type runs the `microservice_repo` template set last which sets it to `ms`. Therefore, if `HMD_REPO_ORG=hmd-` and you run `hmd repo create Microservice` the resulting repository name will be `hmd-ms-<short name>`, with `<short name>` being replace with what you entered when prompted.

8.7 hmd-cli-neuronsphere

A CLI tool for controlling the Local NeuronSphere.

8.7.1 Additional Requirements

- Docker Compose
- Docker

8.7.2 Configuration

Required Environment Variables:

- *HMD_HOME*: folder to store data and configuration for Local NeuronSphere
- *HMD_REPO_HOME*: folder containing NeuronSphere compliant projects, it is mounted into some services

8.7.3 Commands

- *hmd neuronsphere up*: starts all enabled services in the Local NeuronSphere
- *hmd neuronsphere down*: stops all enabled services in the Local NeuronSphere
- *hmd neuronsphere run*: run within a NeuronSphere Microservice project to run it locally for testing
- *hmd neuronsphere update-images*: pull down updated images to run

8.7.4 Included Services

All services are enabled by default.

- Gateway Proxy Server: used to forward calls to running NeuronSphere Microservices
- Postgres Database: relational storage backend for running NeuronSphere Microservices
- DynamoDB: NoSQL storage backend for running NeuronSphere Microservices
- Jupyter Lab Server: used to run Jupyter Notebooks, mounts HMD_REPO_HOME for project access, located at *http://localhost:8888/*
- NeuronSphere Transform Service: a local instance of the NeuronSphere Transform service to test NeuronSphere Transforms locally
- Trino: a Trino database instance to use with local Transforms *http://localhost:8081*
- Airflow: an Airflow instance used by the Transform Service, located at *http://localhost:175/*
- Explorer Portal: an instance of the NeuronSphere Explorer portal for local dashboard development, located at *http://localhost:8088/*
- Minio Object Storage: *http://localhost:9001*
- SQS Queues: *http://localhost:9235*

The following environment variables disable certain services by setting them to *'false'*. For example, *hmd configure set-env HMD_LOCAL_NEURONSPHERE_ENABLE_TRINO false* will disable Trino.

- *HMD_LOCAL_NEURONSPHERE_ENABLE_TRINO*: disables Trino services
- *HMD_LOCAL_NEURONSPHERE_ENABLE_AIRFLOW*: disables Airflow
- *HMD_LOCAL_NEURONSPHERE_ENABLE_TRANSFORM*: disables Transform Service
- *HMD_LOCAL_NEURONSPHERE_ENABLE_DYNAMODB*: disables DynamoDb
- *HMD_LOCAL_NEURONSPHERE_ENABLE_APACHE_SUPERSET*: disables Explorer portal

Disabled services are not started and removed the next time you run *hmd neuronsphere up*.

8.8 hmd-cli-mickey

See also hmd-tf-mickey for specific configuration, also hmd-lib-mickey. See also NeuronSphere writing on model-driven-code generation, as this is the tool used to provide that capability across the suite.

The *hmd-cli-mickey* package, or just Mickey, is a CLI general-purpose code generator. It takes in one or more data files, or contexts, and uses them to render one or more templates. The contexts can be structured text in the form of JSON, YAML, or XML, and run through a simple preprocessor to allow modifications if necessary. The templates use standard Jinja syntax. They are rendered with the values from each context found, plus additional variables that can be supplied via configuration or CLI and some default variables. The rendered output will be saved relative to the repository root in which you run the command. Template paths can also include Jinja syntax to allow for dynamic paths based on contexts.

All of the actual code generation occurs within a Docker container. The CLI is merely a wrapper around pulling and running the *hmd-tf-mickey* container. It will pass along any necessary configuration and environment variables, as well as mounting an necessary volumes. This does mean that Docker must be installed in order use Mickey.

## Installation

The hmd-cli-mickey package can be installed PyPI via Pip:

` pip install hmd-cli-mickey `

The CLI will pull and run a Docker image from *ghcr.io/neuronsphere/hmd-tf-mickey:stable*.

## Configuration

Mickey relies on certain environment variables and a JSON configuration file located at *./meta-data/manifest.json* in your repository.

### Environment Variables

- **HMD_CONTAINER_REGISTRY**: the Docker container from which to pull the *hmd-tf-mickey* image, defaults to *ghcr.io/neuronsphere*
- **HMD_TF_MICKEY_VERSION**: the version tag of *hmd-tf-mickey* to use, defaults to *stable*

### manifest.json

Mickey expects a file located at *./meta-data/manifest.json* relaive to your repository root with the following structure:

```` {

    "name": "<name of repository>, "generate": {

        **"example": {**
            "contexts": ["ref:examples"], "templates": ["<name of repository>/examples/"]

        }

    }, "contexts": {

        "examples": ["src/mickey/data/*.json"]

    }, "templates": ["src/mickey/templates/"]

### 8.8.1 }

The *init* command will generate the above *manifest.json* file for you if you don't already have one. The *generate* property is an object containing different run configurations, or combinations of contexts and templates to render. The *contexts* property is where you define how to collect the different data contexts used to render the templates. The *templates* property is where you reference the location of the Jinja templates. Every template path, or remote package spec (defined later), will be moved into a *.mickey_cache* folder in the repository root. The templates can be referenced from there in *generate* or other templates in a similar manner to Python packages. Any local paths will be copied to *.mickey_cache/<repository name>/path/to/templates*, and can be referenced via *<repository name>/<path>*. For example, a set of templates at *./src/mickey/templates/examples*, and listed in *templates* as above, will be copied to *.mickey_cache/<repository name>/examples* and can be referenced via *<repository name>/examples*.

In the example above, the *examples* definition means that every JSON file in *./src/mickey/data* will be read and used to render all the templates. The run configuration *example* in *generate* then specifies to use the *examples* contexts, and for each file read render every template found in *src/mickey/templates/examples/*. The templates are rendered once for each context found, i.e. if there are 5 JSON files in *./src/mickey/data* for the configuration above, the templates will be rendered 5 times.

## Commands

- *init*: initialize a new Mickey repository, adds manifest.json if doesn't exist.
- *build*: gather contexts and render templates per "generate" configs
````

8.9 hmd-cli-bartleby

8.9.1 Bartleby Install and Run

Installation

The HMD CLI Bartleby tool can be installed using `pip` and specifying the HMD pypi server (via command line or using a pip config file).

```
pip install hmd-cli-bartleby
```

Running the Bartleby Transform

The bartleby CLI uses the `--repo-name` and `--repo-version` arguments inherited from the base cli app to help build the rendered documents. However, the CLI is also built with the assumption that the command is being run from the desired repository root in order to avoid dependencies upon the HMD_REPO_HOME environment variable:

```
hmd bartleby <command>
```

For the `<command>`, any combination of the configured options listed under the bartleby transform (see the "transforms" document under the `hmd-tf-bartleby` repo) can be entered as input. If rendered documents in multiple formats is desired, enter the options as a comma-separated list with *no spaces*:

```
hmd bartleby --shell <option1>,<option2>
```

Configuring Multiple Root Documents

Bartleby can render multiple different root documents with different builders available to each. For example, you might want to render one toctree for PDF outputs and another for HTML. The below config enables that. It should be put in the `meta-data/manifest.json` file of the project.

```
{
    ...
    "bartleby": {
        "roots": {
            "html_doc": {
                "root_doc": "index",
                "builders": ["html"],
            },
            "pdf_doc": {
                "root_doc": "pdf_index",
                "builders": ["pdf"]
            }
        }
    }
}
```

Each entry should contain a `root_doc` property equal to the name of the root RST file to use, without the `.rst` extension. The paths are relative to the `docs/` directory. The entry should also have an array of `builders` that can render this document. The values in the array should be valid options sent to the `--shell` flag, i.e. html, pdf, revealjs, confluence.

When a specific shell is specified on the command line, only documents with that value in their `builders` array will be rendered. For example, running `hmd bartleby --shell html` or the shortcut `hmd bartleby html` will only render the `html_doc` document.

Additional Setup

Ensure the `hmd-tf-bartleby` image is built locally using the hmd docker build tool (`hmd docker build` from the repository root) prior to running the bartleby CLI. The bartleby CLI will look for a local image under the registry name in the HMD_CONTAINER_REGISTRY environment variable (defaults to the HMD registry) in order to run the transform.

Development Setup

After building a new `hmd-tf-bartleby` image locally, you need to set the environment variable `HMD_TF_BARTLEBY_VERSION` to the new tag created. By default, the tag will be the contents of `./meta-data/VERSION` and `-linux-<amd64|arm64>` based on the architecture you are running. For example on Intel machines with VERSION as 0.1, the tag will be `0.1-linux-amd64`. So, you can run and test your newly built local image by setting `export HMD_TF_BARTLEBY_VERSION=0.1-linux-amd64`.

8.9.2 NERD Proposals

NERD001 Specify Root Document

Requirement: **Specify Root Document** *HMD_CLI_BARTLEBY_NERD001*

links incoming: *HMD_CLI_BARTLEBY_NERD001_SPEC001*, *HMD_CLI_BARTLEBY_NERD001_SPEC002*

The Bartleby CLI should read from a NeuronSphere project manifest configuration for which root documents to render, and what shell builders can be used with them. It should default to the root document `'index'` and all shell builders.

The Bartleby CLI already reads some configuration from the project's manifest file. With this proposal, it will read an additional optional property called `roots`. It will be a dictionary with the keys being the name of the root document to render, and the value being a list of valid `shell` values to use. When some runs `hmd bartleby`, it will run for each combination of root document and valid `shell` value.

Specification: **Allow rendering subset of roots** *HMD_CLI_BARTLEBY_NERD001_SPEC001*

status: implemented

links outgoing: *HMD_CLI_BARTLEBY_NERD001*

An additional command line option will be added to allow only rendering a specific root document. Also if a shell builder is passed via command line, e.g. `hmd bartelby pdf`, then only roots that specify `pdf` in the configuration will be rendered.

Specification: **Limit default 'index' root to certain builders** *HMD_CLI_BARTLEBY_NERD001_SPEC002*

status: implemented

links outgoing: *HMD_CLI_BARTLEBY_NERD001*

A special root document shall be reserved for the default 'index'. Specifying this in the manifest is optional but will allow limiting which shell builders run for the default root.

8.10 hmd-cli-bender

8.10.1 Bender Installation and Development

Installation

The HMD CLI Bender tool can be installed using `pip` and specifying the HMD pypi server (via command line or using a pip config file).

```
pip install hmd-cli-bender
```

Running the Bender Transform

The bender CLI uses the `--repo-name` and `--repo-version` arguments inherited from the base cli app to identify the repository that the integration test output should be generated for and tags the test cases accordingly. However, the CLI is also built with the assumption that the command is being run from the desired repository root in order to avoid dependencies upon the HMD_REPO_HOME environment variable:

```
hmd --repo-name <repo> bender <arguments>
```

Optional arguments include `--test-suite` or `-ts` so that a test suite name can be specified in the event test cases are nested under a directory within the test folder (default is "*.robot"):

```
hmd --repo-name <repo> bender -ts <test_suite>
```

The second optional argument `--include` or `-i` can be used if only a subset of the test suite should be run. For example, if the test suite includes two test cases tagged as "Transform build" and "Transform run" and only the second test needs to be run, the command would be entered as follows:

```
hmd --repo-name <repo> bender -ts <test_suite> -i Transform_run
```

Additional Setup

Ensure the `hmd-tf-bender` image is built locally using the hmd docker build tool (`hmd docker build` from the repository root) prior to running the bender CLI. The bender CLI will look for a local image under the registry name in the HMD_CONTAINER_REGISTRY environment variable (defaults to the HMD registry) in order to run the transform.

8.11 hmd-cli-version

The `hmd-cli-version` repo contains a cli command that manages version numbers in a standard manner. There are 2 usages:

```
usage: hmd version [-h] [-v] {tag,update} ...

Manage version numbers

optional arguments:
  -h, --help     show this help message and exit
  -v, --version  Display the version of the version command.

sub-commands:
  {tag,update}
    tag          tag the repo with the version number
    update       update the version number with the build number
```

The `update` subcommand appends the specified build number onto the end of the contents of the repo `meta-data/VERSION` file and saves it back into the same file on the local disk.

Subsequent commands read the version from this file and label artifacts with the appropriate version.

The `tag` subcommand adds a tag to the git repo. The tag is the version contained in the `meta-data/VERSION` file.

Note: This command is rarely run manually or locally, rather is more often used in an automated fashion as a part of the build process in the CI system.

8.12 hmd-cli-docker

The `docker` command is responsible building and publishing docker images.

There are six subcommands, `build`, `publish`, `deploy`, `login`, `destroy`, `release`.

```
usage: hmd docker [-h] [-v] {build,deploy,destroy,login,publish,release} ...

Build and deploy docker images

optional arguments:
-h, --help            show this help message and exit
-v, --version         Display the version of the docker command.

sub-commands:
{build,deploy,destroy,login,publish,release}
    build               build a docker image
    deploy              migrate a docker image to aws
    destroy             Destroy is a noop
    login               logins into multiple Docker registries
    publish             publish a docker image to the HMD image repo
    release             release a Docker image to the public
```

Additional Requirements

- Docker >=20

8.12.1 Commands

build

The `build` command assumes that the repository is a valid HMD docker repo and contains a `src/docker` directory that contains the resources required to build a docker image.

It will by default build two Docker images, one for the `linux/amd64` instruction set and one for `linux/arm64`. The command effectively does the following:

```
docker buildx build -t <HMD_CONTAINER_REGISTRY>/<repo-name>:<version>-amd64
    \ -f ./src/docker/Dockerfile
    \ --no-cache
    \ --platform linux/amd64
    \ --secrets id=pipconf,src=<filepath to local pip.conf>
    \ --progress plain
    \ --output type=docker,name=<HMD_CONTAINER_REGISTRY>/<repo-name>:<version>-amd64
    \ ./src/docker/

docker buildx build -t <HMD_CONTAINER_REGISTRY>/<repo-name>:<version>-arm64
    \ -f ./src/docker/Dockerfile
    \ --no-cache
    \ --platform linux/arm64
    \ --secrets id=pipconf,src=<filepath to local pip.conf>
    \ --progress plain
    \ --output type=docker,name=<HMD_CONTAINER_REGISTRY>/<repo-name>:<version>-arm64
    \ ./src/docker/
```

The multiple architecture build can be disabled via setting the `HMD_HMD_DOCKER_SINGLE_ARCH` environment varaible in `$HMD_HOME/.config/hmd.env` to `'true'`. It will then only build the architecture of the current host machine.

The Docker build process can be further configured in the containing Project's `manifest.json`. A common configuration is shown below. The `docker` property should be inserted into the Project's `manifest.json`.

```
{
    "docker": {
        "build": {
            "context_dir": "./",
            "install_local": true
        }
    }
}
```

The above configuration tells `hmd docker build` to use the repository root as the Docker context directory, instead of `./src/docker`, and to install the Python package in this Project via the `./src/python/` folder, instead of publishing to a registry first and pulling from there. You are responsible for adding the appropriate `COPY` commands to the Dockerfile.

```
COPY meta-data/ /meta-data
COPY src/python/ /src/python/

# you must specify COPY source paths relative to the Project root with "context_dir" set to
↪"./"
COPY src/docker/requirements.txt requirements.txt
```

The `hmd docker build` command will append `/src/python` to the generated `requirements.txt` file in `./src/docker/` automatically, if `install_local` is true.

publish

The publish command pushes a docker image(s), created with the build command, to the coinfigured image repository. The configured target image repository is determined by the environment variable `HMD_CONTAINER_REGISTRY`. If multiple images were built, i.e. multiple architectures, `hmd docker publish` will push each image and then push an image manifest that points to the images.

deploy

The deploy command migrates an image in the `HMD_CONTAINER_REGISTRY` to an AWS ECR in a specific AWS account. This is used for images that contain the code for an HMD service to be deployed as an AWS Lambda Function. (AWS Lambda requires images to exist in ECR.)

login

The login command iterates over the dictionary in environment variable `DOCKER_REGISTRIES` and performs a `docker login` command. Each registry in the dictionary must contain at least a `url` property, pointing to the location of the registry. If the registry is AWS ECR, the property `type` must equal `ecr` and there must be a property `account` equal to the AWS Account number. All other registries only require, `url`, `username` and `password`.

destroy

The destroy command is a noop and only present because it might be called during a destroy workflow from the Deployment Service.

release

The release command migrates a Docker image from our private registry to the public one. This command is automatically called by `hmd release`, if `manifest.json` lists it as a release command. It requires the following environment variables:

- `DOCKER_RELEASE_USERNAME`: the username to login into the public registry
- `DOCKER_RELEASE_PASSWORD`: the password to use logging into the public registry
- `HMD_RELEASE_CONTAINER_REGISTRY`: the url of the public registry, e.g. `ghcr.io/neuronsphere`

Repository Credentials

Docker credentials are looked for as follows:

1. Environment variable `DOCKER_REGISTRIES` configured with multiple Docker registries to authenticate against
2. Environment variables: `DOCKER_USERNAME` and `DOCKER_PASSWORD`; used as defaults for any registries above

All environment variables should be set via `hmd configure set-env`. Docker credentials are only required if pushing/pulling private images.

8.12.2 Python Dependency Management

Note: See the `hmd python` command documentation for details on dependency management for Python packages.

The `build` command uses the `pip-compile-multi` tool to manage python dependencies.

If there is a file named `requirements.in` in the `src/docker` folder, the `build` command uses `pip-compile-multi` to compile it into the file named `requirements.txt`. The Dockerfile file is responsible for consuming `requirements.txt` to build the image.

The `requirements.in` file contains the python packages (with versions specified) that the image requires to execute. Once compiled with `pip-compile-multi`, the `requirements.txt` file contains all dependencies required, including both direct and transitive dependencies.

If the `requirements.in` file references a Python package with the same name as the current image being built, then it is assumed that the version number of the Python package being included in the docker image should be the current version number and the build command replaces whatever is specified with the current version number.

This is primarily to support micro-service and transform repositories that contain both Python and Docker sources.

In many cases, especially for micro-service images, the `requirements.in` file can be as simple as specifying the python package of the same name. For example, for the `hmd-ms-deployment` repo, the `src/docker/requirements.in` file can simply contain:

```
hmd-ms-deployment==<current>
```

The build command will replace `<current>` with the current version number. This works because the image is just used to run the service, which is specified by the Python package, and all necessary dependencies are specified by the Python package. If `install_local` was set to `true`, then the above line will be replaced with `/src/python/`. It is assumed that the Dockerfile has the appropriate COPY commands to install from `/src/python/`, as described in the Build section above.

Only the `requirements.in` file is required to be checked into the repository. Unless there are special circumstances for a specific image, the `requirements.txt` file should not be added to the repository.

8.12.3 Local Development

For local development, the `requirements.txt` will need to be generated manually using the `pip-compile-multi` command (see `hmd python` command docs for details) in order to ensure the build runs successfully.

In order to successfully generate the `requirements.txt` file, the `requirements.in` file will first need to be updated to replace the `<current>` version tag with an actual version number. This happens in a temporary build directory, and the resulting `requirements.txt` is copied back to the project.

Note: For Windows users building Linux images, if `docker` is included as a direct or transient dependency in the `requirements.txt` file, ensure any additional dependencies specific to Windows hosts (e.g. `pywin32`) are removed from the `requirements.txt` file prior to running the docker build command.

8.13 Indexes and tables

- genindex
- modindex
- search

8.14 hmd-cli-python

The `python` command is responsible buiding, testing and packaging Python libraries.

There are five subcommands, `build`, `publish`, `install-local`, `login`, and `release`.

```
usage: hmd python [-h] [-v] {build,install-local,login,publish,release} ...

Build and deploy python packages

optional arguments:
-h, --help            show this help message and exit
-v, --version         Display the version of the python command.

sub-commands:
{build,install-local,login,publish,release}
    build               Execute unit tests and build package.
    install-local       Install Python package into local environment
    login               login into multiple registries and edit pip config
```

(continues on next page)

(continued from previous page)

```
    publish             deploy a python package
    release             release a project to PyPI
```

8.14.1 build

The `build` command assumes that the repository is a valid HMD python repo and has a `src/python` directory that contains the resources required to build a python package.

If the directory `src/python/test` exists, the build `command` installs the python package and executes the tests contained in the `test` directory with the following command:

```
cd src/python
python -m pytest test
```

If the tests are successful, a distributable python wheel is built.

The `build` command also manages the python dependencies. See the section below for a discussion.

8.14.2 publish

The `publish` command uploads a distributable wheel file to the HMD pypi index using `twine`. (The wheel file must exist and is created with the `build` command.)

If present, `twine` uses the environment variables, `TWINE_REPOSITORY_URL`, `TWINE_USERNAME` and `TWINE_PASSORD` environment variables to authenticate to the artifacts repository.

Otherwise, credentials for the artifacts repository must be configured in the file, `.pypirc`, with the name `neuronsphere` like so:

```
[distutils]
index-servers =
    neuronsphere
    pypi

[pypi]
repository: https://pypi.org/pypi

[neuronsphere]
repository: <repository url>
username: <email>
password: <password>
```

The `login` command will create this file if it does not exist. However, you must mark one of the configured registries with `{"publish": true}`. See the `login` command section for more details.

8.14.3 install-local

The `install-local` command will build the wheel and install it into your local environment. This is different than `pip install -e` as it does not create a symlink in your `site-packages` directory. The `install-local` command will allow you to mount your `site-packages` folder into a Docker container, and still use your local changes. Symlinks cannot be resolved to folders outside the mounted volume. However, you will need to re-run the command to reflect new changes.

8.14.4 login

The `login` command handles authenticating to multiple Python package registries or repositories. It looks for an environment variable named `PYTHON_REGISTRIES` that is a JSON strting containing an object with different options for authentication.

```
{
    "jfrog": {
        "url": "https://hmdlabs.jfrog.io/hmdpypi/pypi/simple",
        "username": "<email>",
        "password": "<password>",
        "publish": true
    },
    "codeartifact": {
        "url": "https://codeartifact",
        "account": "AWS_ACCOUNT",
        "type": "codeartifact",
        "domain": "<domain>"
    }
}
```

The above configuration will update the `pip.conf` to have two extra index urls, one for jfrog and one for codeartifact. The `type` parameter is only required for AWS CodeArtifact as we use that to determine if we need to make a boto3 call for a temporary authorization token. Any registry with `publish` equal to `true` will also be used to configure a `.pypirc` file for Twine uploads, if one doesn't exist already.

8.14.5 release

The `release` command moves a Python package from our private repository, usually JFrog, to public PyPI. The destination url is determined by standard Twine environment variables. The source url must be configured in `pip.conf`, just as if you were to install the package.

8.14.6 Python Dependency Management

Note: See the `hmd docker` command documentation for details on dependency management for docker images.

The `build` command uses the `pip-compile-multi` tool to manage python dependencies.

If there is a file named `requirements.in` in the `src/python` folder, the `build` command uses `pip-compile-multi` to compile it into the file named `requirements.txt`. The contents of `requirements.txt` are then inserted into the `setup.py` file in the `install_requires` argument of the `setup` method call.

The `requirements.in` file contains the python packages (with versions specified) that the project depends on directly. Once compiled with `pip-compile-multi`, the `requirements.txt` file contains all dependencies required, including both direct and transitive dependencies.

Both the `requirements.in` and `requirements.txt` should be added to the git repository for a python project. The reason for this is that the `pip-compile-multi` command is executed with the `--no-update` option. This means that once a transitive dependency is specified in the `requirements.txt` file, it will not be updated as long as it still satisfies all dependency requirements.

Developer Responsibilities

This section outlines the responsibilities of developers when dealing with Python dependencies.

Initial Project Creation

When a Python project is initially created, the following steps should be taken.

1. Create a `requirements.in` file in the `src/python` folder that contains the direct dependencies.
2. Use `pip-compile-multi` to generate the `requirements.txt` file (see the command below).
3. Update `requirements.in` with specific version numbers from `requirements.txt` (if necessary).
4. Add both files to the git repository.

Dependency Version Change

If the version of an existing required version changes or a new dependency is required, the following steps should be taken.

1. Update the `requirements.in` file with the new dependecy (version).
2. Use `pip-compile-multi` to generate the `requirements.txt` file (see the command below).
3. Commit both files.

Special Treatment for HMD Libraries

For all HMD Python libraries, i.e., those specified by the glob, '`hmd-*`', the pip version specified will be changed to the PIP compatible version operator of ~=. For example, if the following is specified in `requirements.in`:

```
hmd-cli-tools==0.2.273
```

it will be translated to the following in `requirements.txt`:

```
hmd-cli-tools~=0.2.273
```

The compatible operator, ~=, is roughly the same as the following specifier:

```
>= V.N, == V.*
```

This will make it easier to deal with changes to HMD libraries that are consumed in many other HMD projects.

pip-compile-multi Command

The pip-compile-multi command is executed as part of the build command. The `--pip-compile-only` option can be used to only run `pip-compile-only` and exit. For example:

```
hmd python build -pco
```

To execute `pip-compile-multi` directly, cd to the python folder and use the following command:

```
pip-compile-multi -d $PWD -t $PWD/requirements.in --no-upgrade --no-annotate-index -c 'hmd-*'
```

For Windows users, the following command should be used in order to correctly parse the filepath:

```
pip-compile-multi -d $PWD -t $PWD\requirements.in --no-upgrade --no-annotate-index -c 'hmd-*'
```

8.15 hmd-cli-explorer

The hmd-cli-explorer tool is a CLI to manage NeuronSphere compliant Apache SuperSet projects, including extraction of subsets of assets to projects, packaging, and deployment to other environments.

TODO: usage and examples

8.16 hmd-cli-build

Generic build tool for NeuronSphere compliant projects.

8.16.1 Declaring Build Steps

A NeuronSphere Project declares which HMD CLI commands are required to build it in the *./meta-data/manifest.json* file. The package *hmd-cli-build* installs a general build command, *hmd build* that handles calling each required command in order. The *hmd build* command reads the *./meta-data/manifest.json*, and attempts to run the HMD tools listed in the *build.commands* section. The tools listed must be installed separately (most are included in the *neuronsphere* package).

Example:

```
{
  "name": "example-project",
  "build": {
    "commands": [["mickey"], ["python"]]
  }
}
```

In the above example, *hmd build* will first run *hmd mickey build* and then *hmd python build*. If a build command errors or fails, the entire build will exit and fail. All subsequent commands listed will not be called. Build commands are run sequentially in the order listed. It is not possible to run commands in parallel.

8.16.2 Strict Mode

By default, *hmd build* runs with *HMD_BUILD_STRICT_MODE=true*, or in **strict mode**. While in **strict mode**, the build will error and fail if one of the listed build commands is not installed. This behavior is desirable on a CI or build server, but not for local development. You can turn off **strict mode** by setting *HMD_BUILD_STRICT_MODE=false*. This will be set for you when you run *hmd configure* after installing the *neuronsphere* package.

When **strict mode** is disabled, *hmd build* will only warn about a missing command and continue on to the next command.

8.17 hmd-cli-account

The `account` command is responsible for creating and configuring AWS accounts for use within the NeuronSphere ecosystem.

NOTE: In both commands, for ENG (engineering) account types, the `environment` parameter **must** be set. Likewise, the value used for the environment parameter will need to be used when deploying repo classes to your ENG account For non ENG account types, the environment parameter can be omitted.

8.17.1 Account Config File

In order to configure provider related actions, the config file will need to be setup.

Notes:

- **Only provide Datadog config if the intent is to create a new account for the customer.**
  - There is no way to check via the API if the account is already present. Else a duplicate account will be created.
- Alertlogic info will not be updated (CURRENTLY) if the secrets already exist in the organization.

Config file structure is as follows:

```
{
  "datadog_api_key": "",
  "datadog_app_key": "",
  "alertlogic_account_id": "",
  "alertlogic_secret_key": "",
  "alertlogic_access_key_id": "",
  "okta_org_url": "",
  "okta_api_key": ""
}
```

8.17.2 Account CLI Help

```
usage: hmd account [-h] [-v] {bootstrap,bootstrap-organization,create,create-state-bucket} ..
↪.

Setup new account

optional arguments:
  -h, --help            show this help message and exit
  -v, --version         Display the version of the account command.
```

(continues on next page)

(continued from previous page)

```
sub-commands:
  {bootstrap,bootstrap-organization,create,create-state-bucket}
    bootstrap           add critical resources to a HMD account
    bootstrap-organization
                        add critical resources to the HMD AWS organization
    create              create an HMD account for
    create-state-bucket
                        adds state bucket to an AWS account
```

8.17.3 Create Command

Note: This command must be executed by the customer's root account.

```
usage: hmd account create [-h] -at {eng,dev,test,prod,grc,security} -e ENVIRONMENT -bhr␣
↪BACKUP_HMD_REGION -aa ADMIN_ACCOUNT_NUMBER -cf STDIN < <PATH_TO_CONFIG_FILE>

optional arguments:
  -h, --help            show this help message and exit
  -bhr BACKUP_HMD_REGION, --backup-hmd-region BACKUP_HMD_REGION
  -at {eng,dev,test,prod,grc,security}, --account-type {eng,dev,test,prod,grc,security}
  -e ENVIRONMENT, --environment ENVIRONMENT (defaults to neuronsphere)
  -aa ADMIN_ACCOUNT_NUMBER, --admin-account ADMIN_ACCOUNT_NUMBER
```

Table 1: create options

Option	Description
`--backup-hmd-region`	The backup HMD region
`--account-type`	The type of account being created.
`--environment`	A name indicating the environment of the account.
`--admin-account`	The account number of the customer's admin account. Note: This is not required if the account is the admin account.

Discussion

The `create` command creates a new AWS account in the HMD organization/customer structure. The account has the following characteristics:

Account Name	`<customer_code>.<environment>.<account_type>`
Root email address	`<customer_code>.<environment>.<account_type>@neuronsphere.io`

The create command prints account name and id. Example output is given below.

Account Type ``admin``

For accounts of type `admin`, a special user named `hmd.deployer` is created.

```
Account Name: hmd.neuronsphere.dev
AccountId:    <new account id>

<Remaining secret and bootstrap output>
```

8.17.4 Bootstrap Command

See also hmd-cli-ns-bootstrap

```
usage: hmd account bootstrap [-h] -at {eng,dev,test,prod,grc,security} -e ENVIRONMENT -bhr␣
↪BACKUP_HMD_REGION -a ACCOUNT_NUMBER  -aa ADMIN_ACCOUNT_NUMBER -cf STDIN < <PATH_TO_CONFIG_
↪FILE>

optional arguments:
  -h, --help            show this help message and exit
  -bhr BACKUP_HMD_REGION, --backup-hmd-region BACKUP_HMD_REGION
  -at {eng,dev,test,prod,grc,security}, --account-type {eng,dev,test,prod,grc,security}
  -e ENVIRONMENT, --environment ENVIRONMENT (defaults to neuronsphere)
  -a ACCOUNT_NUMBER, --account ACCOUNT_NUMBER
  -aa ADMIN_ACCOUNT_NUMBER, --admin-account ADMIN_ACCOUNT_NUMBER
```

Table 2: create options

Option	Description
`--backup-hmd-region`	The backup HMD region
`--account-type`	The type of account being created.
`--environment`	A name indicating the environment of the account.
`--account`	The account number of the account to bootstrap
`--admin-account`	The account number of the customer's admin account

Discussion

The `bootstrap` command will bootstrap an existing account with the necessary resources. This command is a non destructive command and is safe to run as many times as necessary. There are two ways to run this command:

1) Using an/the admin account profile and specifying which account to bootstrap (this assumes the account to bootstrap has the deploy role present)
2) Using a profile from the account to bootstrap and specifying the admin account

```
Option #1

    hmd -p hmd.neuronsphere.admin -hr reg1 account bootstrap -a 090132820619 -at dev

Option #2

    hmd -p hmd.michael.misshore.eng -hr reg1 account bootstrap -aa 830985159802 -e mam -at␣
↪eng
```

Both the admin account and account flags cannot both be set. Only one can be presented or an error will be thrown.

8.17.5 Bootstrap Organization Command

Note: This command must be executed by the customer's root account.

```
usage: hmd account bootstrap-organization [-h] -bhr BACKUP_HMD_REGION -sa SECURITY_ACCOUNT_
↪NUMBER -ga GRC_ACCOUNT_NUMBER -aa ADMIN_ACCOUNT_NUMBER -cf STDIN < <PATH_TO_CONFIG_FILE>

optional arguments:
  -h, --help            show this help message and exit
  -bhr BACKUP_HMD_REGION, --backup-hmd-region BACKUP_HMD_REGION
  -sa SECURITY_ACCOUNT_NUMBER, --security-account SECURITY_ACCOUNT_NUMBER
  -ga GRC_ACCOUNT_NUMBER, --grc-account GRC_ACCOUNT_NUMBER
  -aa ADMIN_ACCOUNT_NUMBER, --admin-account ADMIN_ACCOUNT_NUMBER
```

Discussion

- Cloudtrail, Security Hub, and Guard Duty are setup in the root account.
- GRC (only primary region) and Security accounts are re-bootstrapped
- **Missing root secrets are created and all member account secrets are created if missing**
  - A flag will be introduced in the future that will allow a force override.

8.18 hmd-cli-app

8.18.1 HMD Command-line Tool Application Installation

The HMD command-line tool can be installed using `pip` by specifying HMD's pypi server either on the command line or in the `~/.pip/pip.conf` file. (See the `start_here` document in the Core Arch Docs repo.)

Development

For development, it is helpful to install the app and commands in `develop` mode. To do that, follow these instructions.

1. clone `hmd-cli-tools`
2. clone `hmd-cli-app`
3. clone `hmd-cli-docker`
4. clone any other command repos

Assuming that each of these repos are in `~/hmd-cli`, execute the following commands. All commands that are installed in the same environment will be detected by the application.

```
# virtualenv is optional
cd ~/hmd-cli
virtualenv hmdenv
source env/bin/activate

# common tool library
cd ~/hmd-cli/hmd-cli-tools/src/python
python setup.py develop

# docker command
```

(continues on next page)

(continued from previous page)

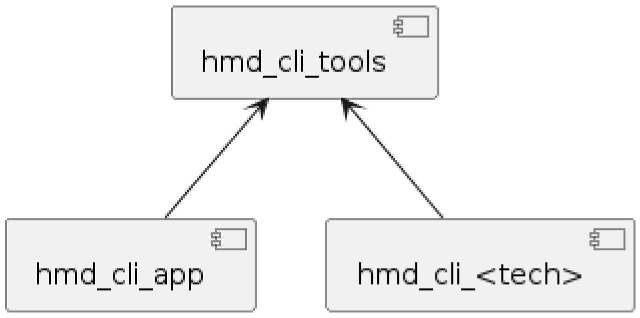
```
cd ~/hmd-cli/hmd-cli-docker/src/python
python setup.py develop

# app
cd ~/cmd-cli/hmd-cli-app/src/python
python setup.py develop
```

8.18.2 HMD CLI Design

The HMD CLI is designed in a way to accommodate growth to an arbitrarily large number of commands. There are multiple repositories that support the CLI:

- `hmd-cli-tools` A shared python library that contains tools common across the tool infrastructure.
- `hmd-cli-app` The CLI application, itself, implemented using the `Cement` API framework.
- `hmd-cli-<tech>` One repository for each technology implementation. This repository contains the actual implementation of the lifecycle support.

The following diagram indicates the dependency structure between the repos.



All common code resources will reside in the `hmd-cli-tools` library. The `hmd-cli-app` and all technology repos will depend on `hmd-cli-tools` as necessary.

The CLI application will auto-discover, at runtime, all technology tools that have been installed in the environment in which it is executed. The auto-detection works as follows:

- For each module, the name of which starts with "`hmd_cli`", the CLI application looks for a package named "`controller`" that contains a class named "`LocalController`".
- For each one it finds, it adds the `LocalController` class to the list of controllers configured for the application.

Implementing a Technology

Each technology command is plugged into the CLI application by means of a `Controller` class. The controller class declares all of the arguments to the command as well as to each of its sub-commands. It also provides the code to execute when the command or one of its sub-commands is executed.

As an example, the `LocalController` class for the hmd-cli-docker implementation is given here.

```
import os

from cement import Controller, ex
```

(continues on next page)

(continued from previous page)

```python
from importlib.metadata import version
from hmd_cli_tools import get_version

VERSION_BANNER = "hmd docker version: {}"


class LocalController(Controller):
    class Meta:
        # the name of the command...
        label = "docker"

        # Always stack on the base controller
        stacked_type = "nested"
        stacked_on = "base"

        # text displayed at the top of --help output
        description = "Build and deploy docker images"

        # Display the version of the local command (not the CLI app)
        arguments = (
            (
                ["-v", "--version"],
                {
                    "help": "Display the version of the docker command.",
                    "action": "version",
                    "version": VERSION_BANNER.format(version("hmd_cli_docker")),
                },
            ),
        )

    def _default(self):
        """Default action if no sub-command is passed."""
        self.app.args.print_help()

    @ex(
        help="build a docker image",
        arguments=[
            (
                ["-n", "--name"],
                {"action": "store", "dest": "name", "required": False},
            ),
            (
                ["--build-args"],
                {
                    "action": "store",
                    "dest": "build_args",
                    "required": False,
                    "nargs": "*",
                },
            ),
        ],
    )
    def build(self):
        args = {}
```

(continues on next page)

(continued from previous page)

```
        # Prepare the args
        #...

        from .hmd_cli_docker import build as do_build

        result = do_build(**args)
        for l in result:
            print(l)

    @ex(
        help="deploy a docker image",
        arguments=[
            (
                ["-n", "--name"],
                {"action": "store", "dest": "name", "required": False},
            )
        ],
    )
    def deploy(self):
        args = {}
        # Prepare the args
        #...

        from .hmd_cli_docker import build as do_deploy

        do_deploy(**args)
```

The `@ex` annotation declares the method as a sub-command and defines any arguments that it requires. The methods in the controller class are strictly the glue between the UI and the implementation. They are responsible for declaring and preparing arguments and then delegating to the implementation code. They will also manage the display of any results to the end-user.

Note that it is only in the method body for the sub-command that the implementing code is imported. In this manner the library for the implementation is only imported when when necessary. Because an execution of the tool always specifies one-and-only-one technology, only the library that is specifically required will be imported.

The HMD CLI is a set of command-line tools to support the CI/CD pipeline and development on the NeuronSphere platform. Documentation of each of the individual commands is located in the respective command repository.

The general design of the CLI is as follows:

`` `bash $ hmd {--global-options} <technology> {--tech-arguments} <lifecycle-phase> {--phase-arguments} ` ``

Where:

- <*technology*> represents the type of technology, e.g., docker, python
- <*lifecycle-phase*> indicates the lifecycle phase within the CICD process, e.g., build, unit-test

Examples:

`` `bash $ hmd mickey build $ hmd docker tag my-tag-name $ hmd python deploy ` ``

## Global Options

Option | Description |

–repo-name	The name of the repository on which the command is operating. Defaults to the base name of the current working directory.
–repo-version	The current version number. Defaults to the value in the file, *meta-data/VERSION*. Defatults to deploy resources.
–region	The NeuronSphere region in which to operate, e.g. reg1, reg2, etc.
–profile	The AWS profile to use for authentication to AWS.

Each lifecycle phase can have its own set of options. However, the *deploy* phase in many commands have a common set of options, including:

## Deploy Options

Option	Description
–instance-name	The name to give this specific deployment instance
–deployment-id	The deployment id.
–environment	The NeuronSphere environment into which to deploy resources, e.g. dev, prod
–customer-code	The unique assigned customer code
–account	An AWS account nubmer to which to deploy resources. Can be inferred via *–profile*
–profile	The AWS profile to use for authentication to AWS

8.19 hmd-cli-vpn

CHAPTER

NINE

LOCAL NEURONSPHERE

The Local NeuronSphere is a configuration of the NeuronSphere Services and tools, optimized to run on a laptop or standard desktop computer.

The overall development experience locally is designed to support rapid iteration in an environment that mimics as much of the cloud environment as possible, while making a number of provisions for convenient single-user local mode.

Parity and transparency between local and cloud, as it makes sense to purpose, is the goal. If it runs locally, it should run with as little or no change in the cloud, only different configurations and service implementations.

Requirements - NeuronSphere CLI and Docker

9.1 Getting Started with Local NeuronSphere

Local URLs and credentials to bookmark:

9.2 Local Directory Map

The local NeuronSphere runs as a series of docker containers, and mounts a number of local directories into the running containers for convenience of local deployment.

Generally you won't need to directly interact with these folders, rather relying on NeuronSphere CLI tooling to manipulate them, however it is useful to see the pattern and see how other tools will appear and behave in the local stack if you add them.

```
/Users/briangreene/hmd_home
├── data
│   ├── local_transforms
│   ├── raw
│   └── trino
├── datadog
│   ├── log
│   └── s6
├── dynamodb
├── hadoop
│   └── config
├── hive
│   └── config
├── hmd_pyenv
│   ├── LICENSE
│   ├── NOTICE
```

(continues on next page)

(continued from previous page)

```
│   ├── bin
│   ├── include
│   ├── lib
│   └── pyvenv.cfg
├── language_packs
├── postgresql
│   ├── data
│   └── scripts
├── superset_home
│   ├── celerybeat-schedule
│   └── sqllab
├── transform
│   └── airflow
├── trino
│   ├── config
│   ├── data
│   └── hadoop
└── warehouse

30 directories, 4 files
```

9.3 Local NeuronSphere Examples

Now that we've talked about advanced code generation ideas, see the examples on local pipeline development on youtube at

CHAPTER

TEN

NEURONSPHERE CLOUD SERVICES

Given the NeuronSphere development ideas and practical reference implementations for SETA and it's supporting capabilities, we examine the network of components that have been developed to create the NeuronSphere cloud platform.

10.1 NeuronSphere Cloud Deployment Models

The NeuronSphere platform is designed to be highly modular, supporting numerous flexible deployment models to suit the smallest teams, multi-national enterprises, or service providers.

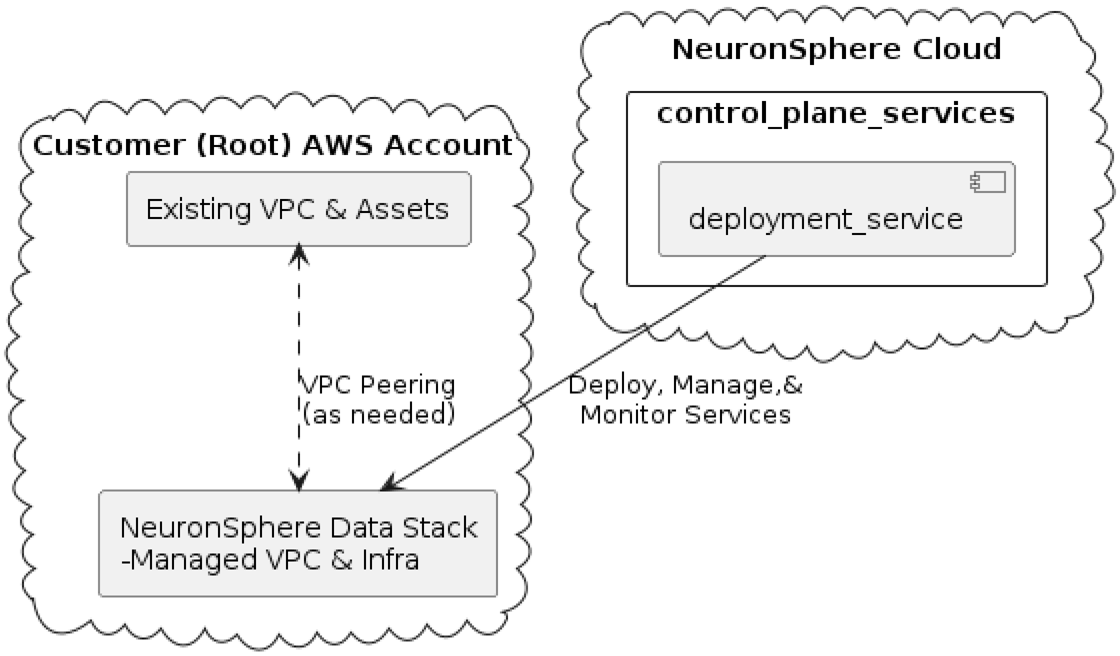
The NeuronSphere cloud components are generally divided into 2 categories:

- Control Plane Services are used to orchestrate and control deployment, infrastucture, configuration, and operations. The end users are Software and Data Developers and operations staff.
- Pipeline aka *workload* Services are used to provide *pre-production* development accounts for uses such as development and testing, as well as production & dr/bc. End users are internal employees and external customers performing development activities or routine work.

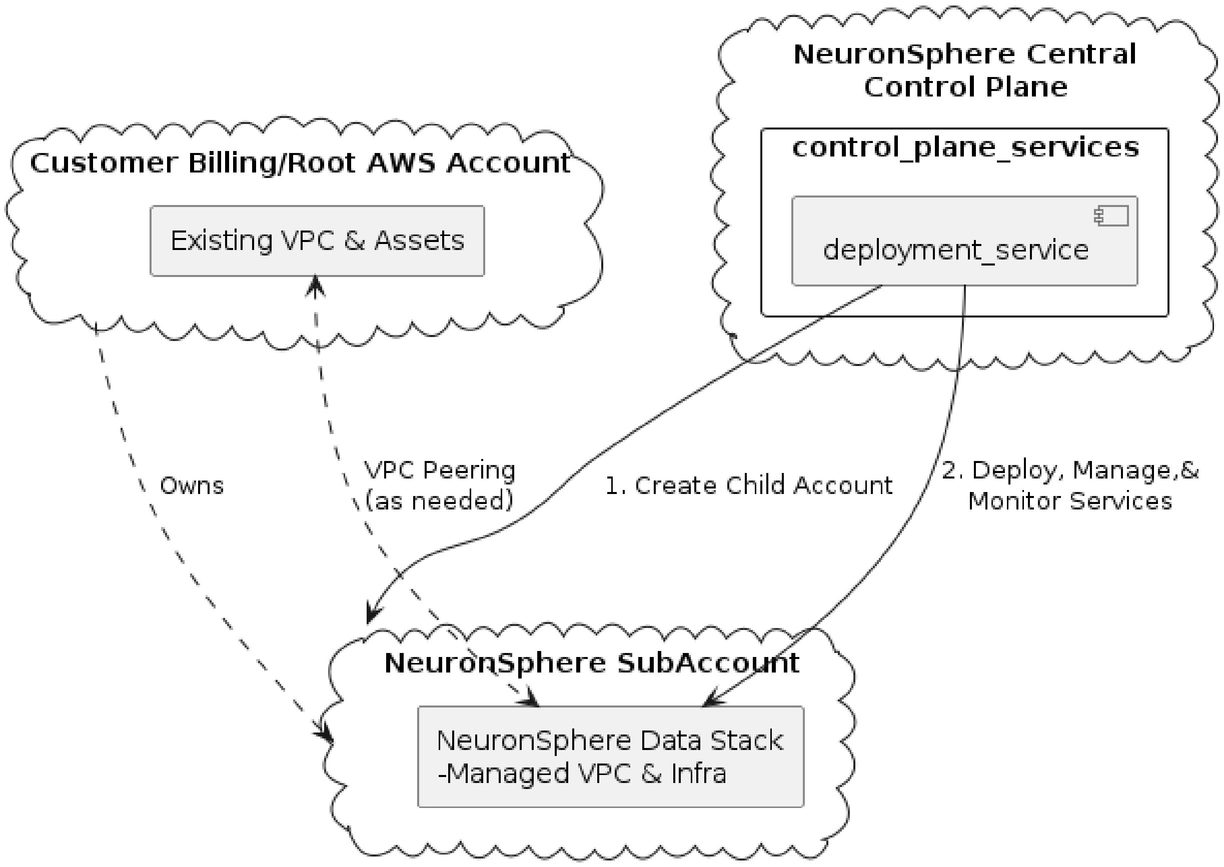
Technically, there is nothing different about the control plane and anything chosen to deploy in any other environment. In fact the control plane and pipeline services can deploy into the same target environments.

There are surely other variations of deployment one could come up with, but this series should cover the main variations and patterns.

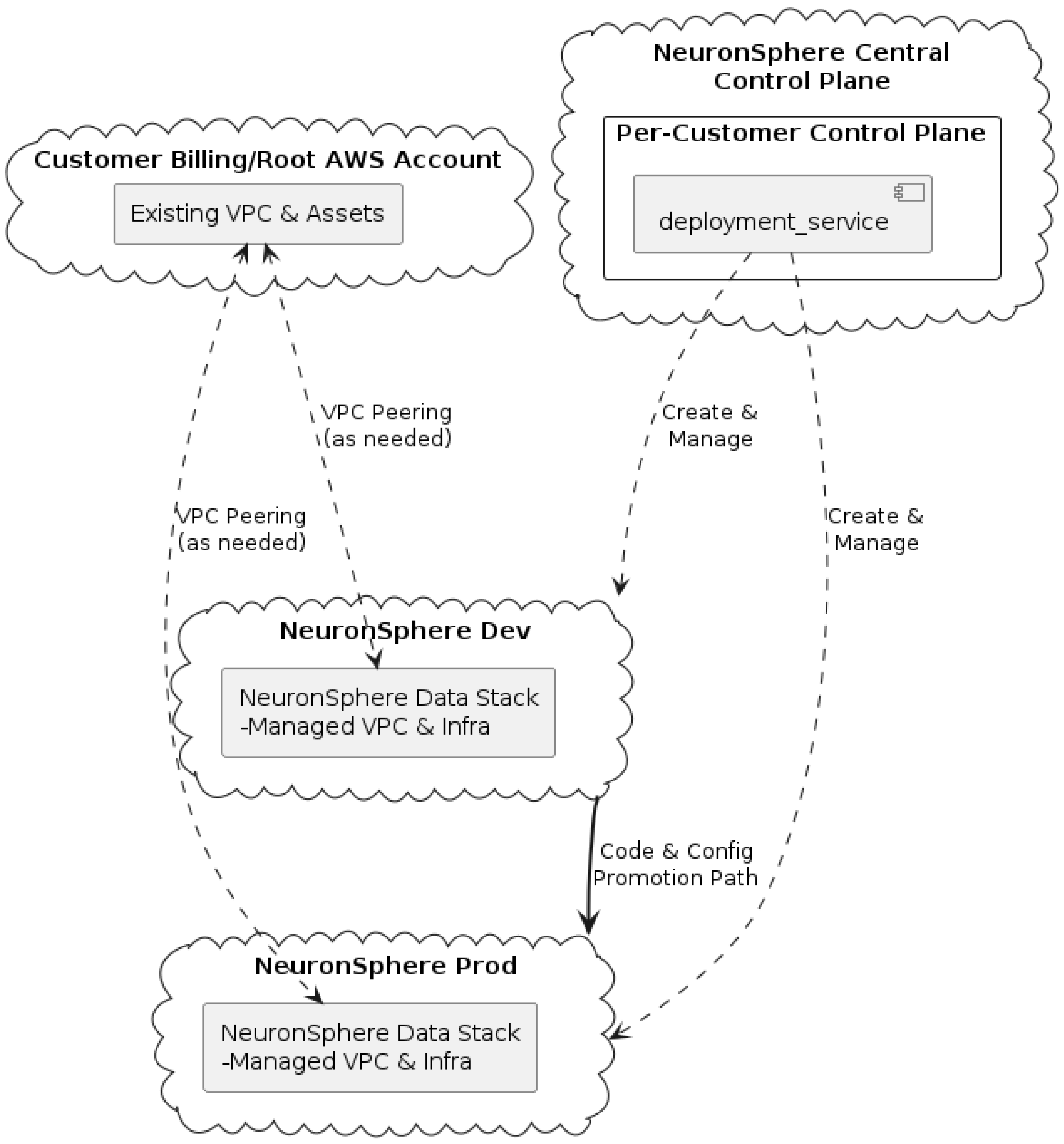
10.1.1 NeuronSphere Stack as Peer



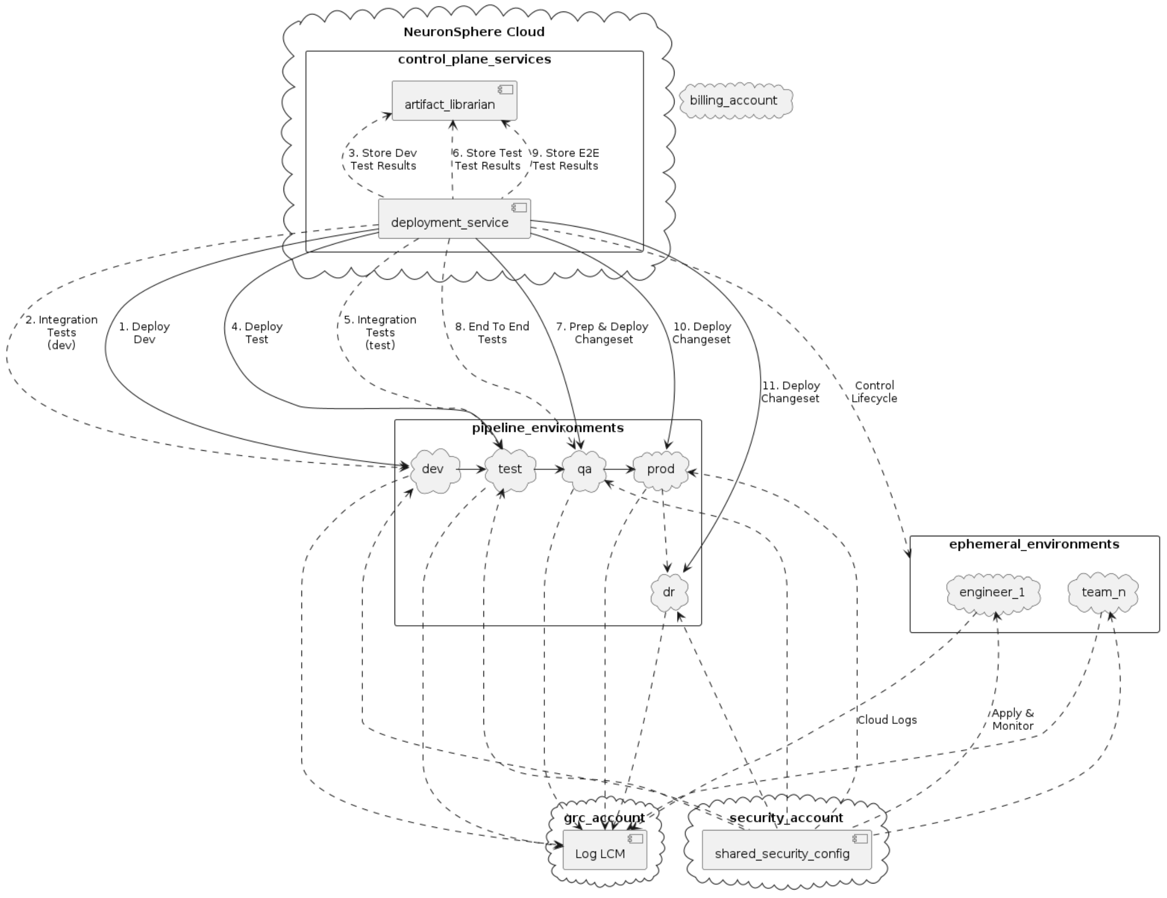
10.1.2 Single Node NeuronSphere as Child



10.1.3 Simple Dual Node Deployment


NeuronSphere Central Control Plane
Per-Customer Control Plane
deployment_service
Customer Billing/Root AWS Account
Existing VPC & Assets
VPC Peering (as needed)
VPC Peering (as needed)
Create & Manage
Create & Manage
NeuronSphere Dev
NeuronSphere Data Stack -Managed VPC & Infra
Code & Config Promotion Path
NeuronSphere Prod
NeuronSphere Data Stack -Managed VPC & Infra

10.1.4 Multi-Node Hosted Control Plane



10.1.5 Multi-Node Isolated Control Plane

TODO

10.1.6 Isolated Control Plane Multi-Target Pipelines

TODO

10.1.7 Fully-Federated Distributed Fractal Model

TODO

10.2 Default NeuronSphere Cloud Data Platform

A listing of the deployable repositories/components available in the NeuronSphere cloud. Note that many teams will not use all of these services, particularly in the dynamic compute space, but much of this stack forms a largely suitable default for many teams.

Note: The components and services available for deployment are within the context of an environment, which is a related but orthogonal construct. For much of the stack described here, we must also ask a related series of questions about deployment model(s).

What does a NeuronSphere Cloud do *out of the box*? As a toolkit and developer platform, we look at the kinds of things we're trying to optimize the building of, and then look at how the supplied tools support these capabilities.

10.2.1 What are we trying to build?

The goal of many software & data teams can be summarized:

For a given *domain*, we wish to build the following kinds of capabilities and assets as quickly, cheaply, and with the most appropriate quality and attributes around scale, reuse, and security.

- Data Visualization and Exploration surfaces
- Fluent & Semantic APIs
- Metrics, Features, and Pipelines
- Governed & Unified Data Models (with lineage)
- Useful Data Marts & Warehouse nodes as needed
- Searchable, Extensible, Integratable Content Stores
- Polyglot Ingress, Egress, Orchestration

In order to deliver these capabilities to a given domain, we rely on the following provided NeuronSphere services.

10.2.2 Data Platform Services

- Data Visualization - Apache SuperSet and tooling for analytics as code using CLIs and projects
- Exploration - Jupyter Notebooks with custom magic libraries to ease cluster interaction and automations
- Metrics & Features Service to store calculated metrics and features against arbitrary entities
- Policy Service manages the abstractions over the integrated OpenPolicy Agent capabilities for ABAC
- Semantic Collection Service contains a complete data catalog of all deployed data models for an environment
- Transform Service using SETA, coordinates the data transformations using Airflow, Argo, dbt, Jupyter, Docker, etc

- Governed Compute Service handles dynamic management of runtime-elastic compute, from k8 infra up through helm chart deployment.
  - Trino - Cluster plugins available for k8s deployment
  - Postgres - plugins for RDS or k8s Postgres instances
  - Spark - plugins for Apache Spark 3.x on k8s*
  - Kafka - plugins for Apache Kafka on k8s*
  - (hive)
  - BYOC - build plugins for any compute engine for governance
- Data Lake Librarian(s) - a content model and storage for a default shared *Librarian* to be used as a *data lake*. This is the default librarian for all managed compute clusters.

10.2.3 Infra and Ops Services (Control Plane)

- *Ops* Services
  - Build - performs build activities based on external triggers (e.g.: github webhook)
  - Deploy - performs deployment and integration test coordination across *NeuronSphere Environments*
  - Project - maintains template hierarchies and provides a proxy over SCM for repo creation and management
  - Naming - resource name resolution for NeuronSphere services and components
- Artifact Librarian - a content model and storage for build, testing, and deployment of artifacts

10.3 Infrastructure Deployable Components

These components have standard factories/modules built and integrated into the *NeuronSphere cloud runtime*. If you want to use a new cloud service, you start by creating a new plugin here.

This ensures things like naming, logging, security, encryption, and regionality, and any needed CLIs can be developed.

- Environment (Account) Management
- EKS & Node Group components - covers AWS EKS and supporting services (e.g.: node groups)
- AWS Lambda variants - AWS Lambda base components in docker and python base models
- RDS & Neptune - AWS RDS Postgress & Neptune clusters and supporting instance management
- S3 variants - S3 modules with require
- VPC & Networking

Note: One of the primary concerns at the infrastructure layer is the careful and consistent application of naming standards, environment variable utilization, and tagging. By abstracting common infrastructure building blocks into consistent implementations with suitable defaults we can much more rapidly construct full-stack services while maintaining orthogonal infrastructure governance and controls.

Note: Because *NeuronSphere is made of NeuronSphere*, all NeuronSphere plugins and components are created and managed with the NeuronSphere CLIs and control plane. As such, extending any layer of it is as easy as using the *repo create* command for the appropriate project type.

10.3.1 External Integrations

Standard modules and extensions exist to integrate the following third-party services. It is expected that this section will be expanded with publicly available plugins.

- AlertLogic - deploys AlertLogic appliance(s) and configures VPC monitoring and GuardDuty integration
- DataDog - streams logs and events from all environments to DataDog, showing a well-defined top-to-bottom view of deployed components
- Okta - AWS and all user-facing services are integrated with Okta, and there are utility libraries to accelerate adding other applications to the Okta tenant for a customer.

10.4 Tenancy - A Slippery Fractal

- One person's tenant is another persons's user.
- One person's customer is another person's control plane.
- One person's environment is another person's shard.

Its turtles all the way down

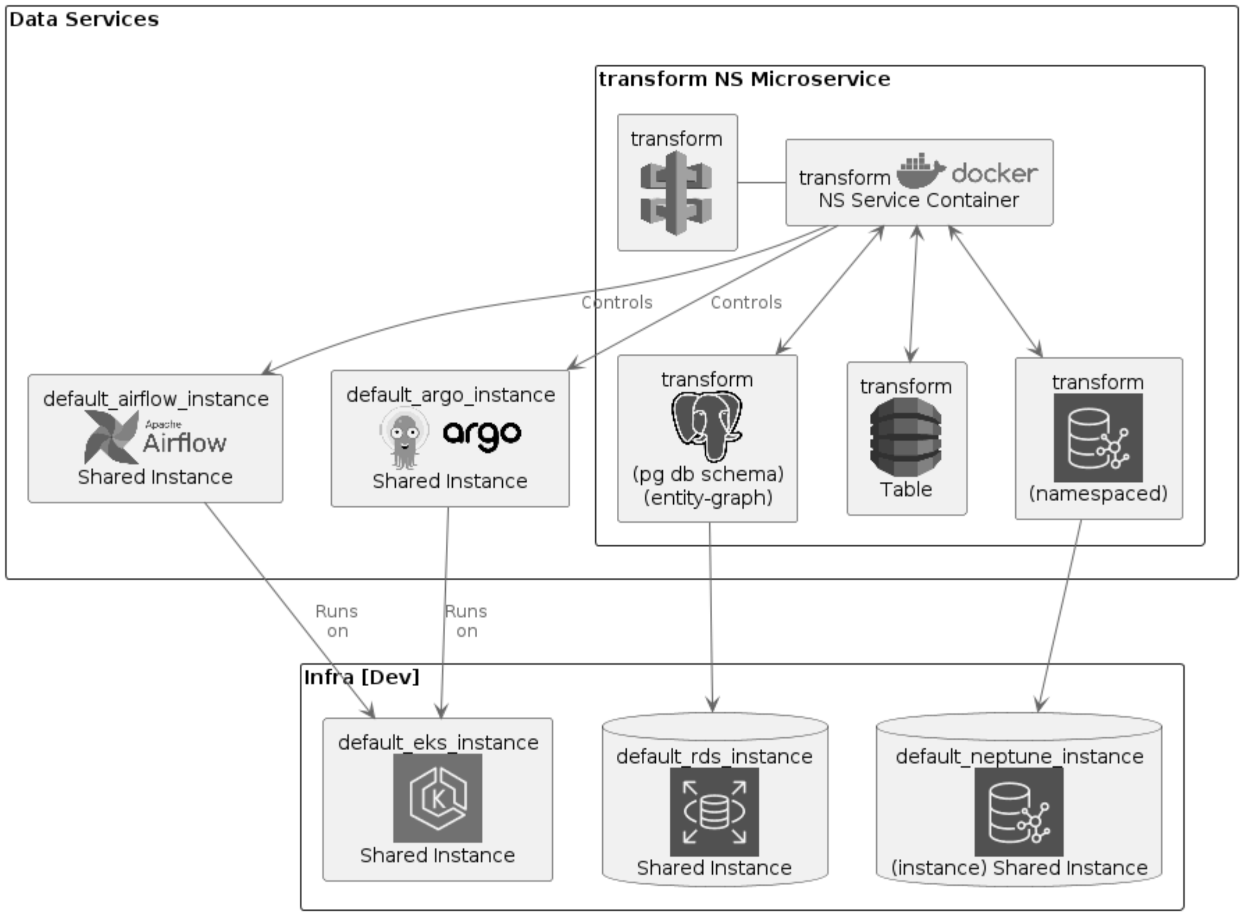
10.5 Transform Service

Finally, we're fully into the *data engineering* space with the *NeuronSphere Transform Service*.

This is the cloud implementation of the Semantic Execution Transformation Architecture. It was once described as an *uber orchestrator*, and that's not a terrible description.

The transform service monitors the global graph state for *Transforms* to execute based on the condition of the semantic graph. When suitable for execution, the transform service delegates the configured work to the appropriate *transform engine* (currently Airflow, Argo, and dbt).

To ease authoring and coordination, the transform service supports a simplified authoring framework that helps to unify dependency and state management across the various transform engines.



Transforms authored as Airflow operators (or dags), dynamic Argo workflows, and variable-driven dbt runs can all be natively executed or coordinated with the transform service for full visibility.

By using the NeuronSphere graph as the state store, we can author dags and workflows in a more *functional* way, allowing substantially better options for reversability and highly-selective backfill.

See also * Transform cli * Transform project templates

10.5.1 Implications for Airflow Users

Included in the Transform service is a complete Airflow management plane, with a number of convenience and enterprise-grade operational features available by default.

- Local Airflow with testing available with a management CLI
- Multi-stage CI/CD deployments of Dags and other Airflow artifacts/config
- Multi-tenant/instance management across stages - run multiple parallel Airflow deployments with optimized configuration by team or workload type
- Mix and match Executors, support k8s and GPU workload routing
- Custom/Private Airflow Images
- Okta integration for SSO and group management
- DataDog integration for operational visibility
- Dag versioning and Dag-generation factory tools

10.6 Canonical Cloud NeuronSphere

Having looked at various deployment models available for the NeuronSphere cloud components at a high level, this section will examine a more detailed view of the components and services that make up the NeuronSphere cloud.

NeuronSphere takes a unique *fractal-federated control plane* architecture for deployment, and it's instructive to visually build up the NeuronSphere components in the sequence and method by which they're installed.

Note: For the sake of visual clarity, networking components such as VPCs and subnets are not shown in most NeuronSphere architecture drawings unless relevant to the discussion.

A standard set of plugins related to networking and core storage are used throughout the platform.

10.6.1 Control Plane base Infrastructure

10.6.2 Control Plane with Artifact Librarian

Create the NeuronSphere environment for the control plane, install the *Artifact Librarian*, and upload the seed binaries.

TODO: pic of artifact librarian in control plane

10.6.3 Control Plane ready to Deploy

Install the Deployment Service in the control plane

10.6.4 Control Plane Complete

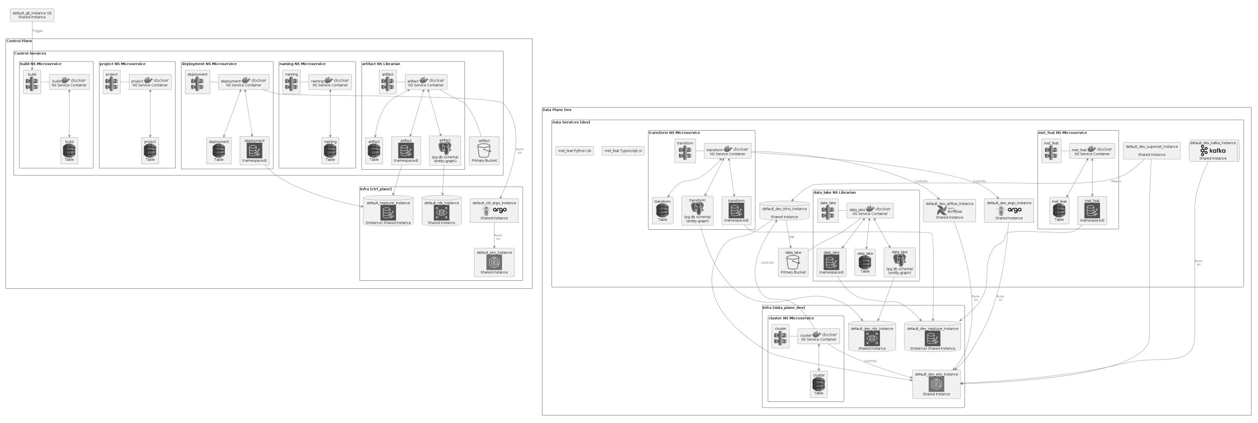
Use the deployment service and a default change set to deploy the rest of the control plane account and configure the defaults.

Create the first pipeline account

Add the minimum shared infra

Add the data layer in pieces

10.6.5 Control Plane, Single Pipeline Environment, Default Data Services


default_git_instance Git Shared Instance
Trigger
Control Plane
Control Services
build NS Microservice
build
build NS Service Container
build Table
project NS Microservice
project
project NS Service Container
project Table
deployment NS Microservice
deployment
deployment NS Service Container
deployment Table
deployment (namespaced)
naming NS Microservice
naming
naming NS Service Container
naming Table
artifact NS Librarian
artifact
artifact NS Service Container
artifact Table
artifact (namespaced)
artifact (pg db schema) (entity-graph)
artifact Primary Bucket
Runs on
Infra [ctrl_plane]
default_neptune_instance (instance) Shared Instance
default_rds_instance Shared Instance
default_ctrl_argo_instance argo Shared Instance
Runs on
default_eks_instance Shared Instance
Data Plane Dev
Data Services [dev]
met_feat Python Lib
met_feat Typescript UI
transform NS Microservice
transform
transform NS Service Container
transform Table
transform (pg db schema) (entity-graph)
transform (namespaced)
default_dev_trino_instance Shared Instance
RW
data_lake Primary Bucket
Controls
data_lake NS Librarian
data_lake
data_lake NS Service Container
data_lake (namespaced)
data_lake Table
data_lake (pg db schema) (entity-graph)
Controls
Controls
default_dev_airflow_instance Airflow Shared Instance
default_dev_argo_instance argo Shared Instance
Reads
met_feat NS Microservice
met_feat
met_feat NS Service Container
met_feat Table
met_feat (namespaced)
default_dev_superset_instance Shared Instance
default_dev_kafka_instance kafka Shared Instance
Runs on
Runs on
Runs on
Infra [data_plane_dev]
cluster NS Microservice
cluster
cluster NS Service Container
cluster Table
default_dev_rds_instance Shared Instance
default_dev_neptune_instance (instance) Shared Instance
Controls
default_dev_eks_instance Shared Instance

Add domain services

Add data products

CHAPTER

ELEVEN

BROWNFIELD CONSIDERATIONS

Assuming you find the contents of this book useful and wanted to start using NeuronSphere, where would you start?

11.1 NeuronSphere Development CLI Tools

Add docs as code and mickey to existing repositories

For existing repos, use a manifest with a build-script driver that allows your exisiting build script(s) to function using NS build pipeline

11.2 NeuronSphere Cloud Delivery Platform

Add NS in a new VPC next to your existing, and plumb into it.

State diagram of bringing NS online to existing root.

How to pull in existing resources.

11.3 NeuronSphere Data Platform

Add new services that your existing app ties to for analytics - event/activity or otherwise Add a data warehouse and libraries to easily start streaming data Use the information model to graph out phi etc across the other models Use it to generate data contract docs and workflows and generated artifacts.

11.4 By User Type

Single project, open source Small co, new Development Adding docs as code and code gen adding a ‘data layer’ adding transforms to existing s3 bucket adding transforms and llm to existing databases

CHAPTER

TWELVE

EXTERNAL REFERENCES

This is a ‘catch all’ list of external references. Full links and mild explanations will be kept here, along with explicit targets. The intent is that the rest of the documents will reference these internal targets, streamlining the cross-referencing process.

Short Name link	Description
12 Factor Apps	A useful set of principles related to distributed software and platform development.
Apache Maven	powers the build, test, packaging, and publishing lifecycle for millions of builds a day globally. With a 1.0 release in 2004 and going strong today, evidence of the power of these ideas is compelling.
Code Generators	A listing of code-generating tools.
Data Modelling	Wikipedia page on data modeling, a key component of NeuronSphere.
Fowler Microservices	A good overview on microservices, good links to related items
Model Drive Architecture	See also MDE
Model Driven Engineering	See also MDA
Software Factory	A design Concept related to making software faster
Trunk Based Development	A superior way to use source control. TBD + CI/CDD is the way.

CHAPTER

THIRTEEN

THINGS NOT IN THIS BOOK

There are many important subjects in computing not covered in this text, most of them actually, as this work is far more about how to coherently tie together lots of detailed ideas and systems into a whole.

That said, a few key ideas are related and not given much or any coverage:

- Master Data Management - a fascinating thing - once this book is done I'll have the foundation for the more detailed text on MDM.
- EA MetaModel - the People Process Information Technology quadrants and their most interesting relationships need a dedicated section written.