Beyond Micro-Services: CORD's Model-Driven Design

Introduction

Building scalable cloud applications as a collection of micro-services is a promising approach to NFV Orchestration. This is because micro-services are well established as the cornerstone of DevOps, which network operators see as a path to improving agility. At the same time, CORD adopts *Everything-as-a-Service (XaaS)* as an organizing principle, which naturally leads to the question: How does CORD (and XaaS) differ from a micro-services architecture?

This document answers that question, and in doing so, describes CORD's model-driven design. XOS is the component of CORD that codifies its models, where XOS can be viewed as a service control plane layered on top of a collection of micro-services. What is innovative about this service control plane is that it is defined by a set of declarative *models* and a set of *operations* on those models, where XOS enforces these models and operations to extract the desired behavior from the system as a whole.

This approach effectively defines CORD's architecture programmatically, with the models and invariants serving as a specification of the architecture, and XOS "executing" this specification using a *generative toolchain* that translates the declarative specification into executable code. Moreover, as operators and developers gain experience with CORD, they are able to codify that experience in the models and invariants, making it possible to evolve the architecture over time. The paper concludes with a discussion of this architectural approach.

Micro-Service Architecture

The case for building cloud applications as a collection of micro-services is compelling. This section gives a brief overview.

Developers start with a rich collection of open source container images; everything from databases to event buses to web servers are available as building block components. Applications are then constructed by deploying these images in a set of containers, and interconnecting these containers with virtual networks. It is easy to see the appeal of this approach. Building a cloud app from a set of containers is reminiscent of writing a Unix application by piping data through a sequence of commands, each running in its own process.

But micro-services go beyond the Unix paradigm by targeting a scalable cloud rather than one machine. This means that instead of a single container at each stage in the pipeline, there is an

opportunity to run a scalable set of containers, with containers spun up and down as workload dictates. Moreover, instead of directly connecting individual containers, groups of containers running a common image (a micro-service) are interconnected by one or more virtual networks, with requests load-balanced across the set of containers implementing each micro-service.

A container management system like Kubernetes or Docker Swarm takes responsibility for scaling such an application. It both load-balances requests across the containers within a micro-service and load-balances the entire set of containers across the physical servers in the underlying cloud. One key advantage of breaking the application into micro-services is that they can each be scaled independently of each other.

The final piece to the puzzle is the configuration and management of the individual micro-services. This is typically done according to the DevOps discipline, for example, using a remote management tool like Ansible. Figure 1 depicts an example cloud application built from a collection of four micro-services.

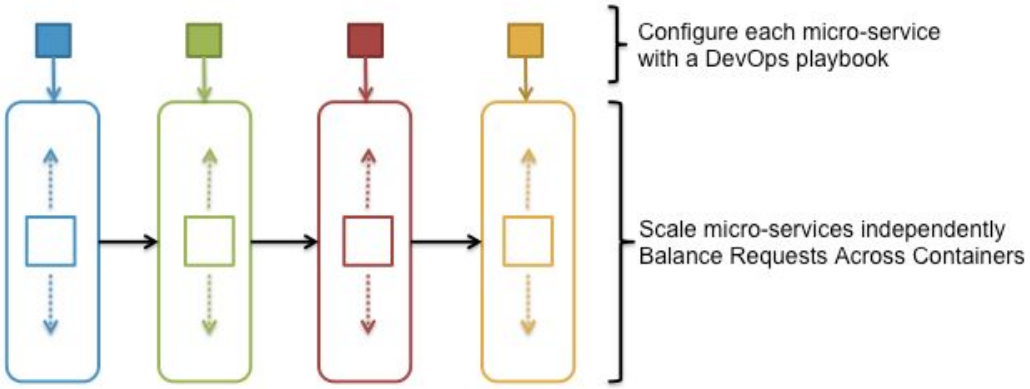

Figure 1. A collection of micro-services managed from playbooks.

Micro-services have been used to build a wide range of cloud applications, from Yelp to Netflix, but there are two key assumptions built into the approach. We discuss them here.

First, micro-services are used to build a single application. Yelp is always Yelp; it is never reconfigured to be Netflix or a Netflix+Yelp hybrid. This assumption has two implications. One is that all the micro-services run in a single trust domain. This means, for example, that if a micro-service that implements a NoSQL database is used by two other micro-services, X and Y, then there is nothing (other than the implicit trust of being part of the same app) keeping X from reading or writing Y's data. Another is that the set of micro-services that implement a cloud application are relatively static. They each evolve and scale independently of each other, but it would be uncommon for the overall composition to change. This naturally leads to both hard coding of inter-service dependencies, and ad hoc sharing of global state across micro-service boundaries. The bottom line is that our goal is to support a *multi-tenant platform* (i.e., services are isolated from each other), with each tenant of the platform running a *multi-tenant service*

(i.e., end-users/subscribers are isolated from each other), where per-subscriber context is preserved end-to-end across a set of services.

Second, container management systems that enable micro-services typically support a fixed infrastructure. This also has two implications. One is that container management systems lock in a particular virtualization technology, such as Docker containers. There is no flexibility to include either heavier-weight technologies (e.g., VMs) or lighter-weight technologies (e.g., namespaces). Another is that only servers—and not the network switches that interconnect those servers—are programmable. This means the virtual networks that interconnect micro-services have fixed behavior, and the micro-services themselves are limited to "server-based" implementations. There is no allowance for "switch-based" implementations. Consider this last point in the context of CORD, where both the vOLT and vRouter services are implemented as control apps running on ONOS. Neither qualifies as a micro-service in the traditional sense, but both provide functionality that can be composed with other functionality to build a solution.

Service Control Plane

This section describes how XOS, by implementing a service control plane on top of a collection of micro-services, addresses the shortcomings outlined in the previous section. Figure 2 sketches the high-level idea, which is inspired by (and borrows terminology from) SDN: XOS defines the *service control plane,* and it is layered on top of a collection of micro-services that collectively implement the *service data plane*. While we borrow the idea of explicitly separating the service control and data planes from SDN, the service data plane is not equivalent to the network data plane (although the former may be embedded in the latter).

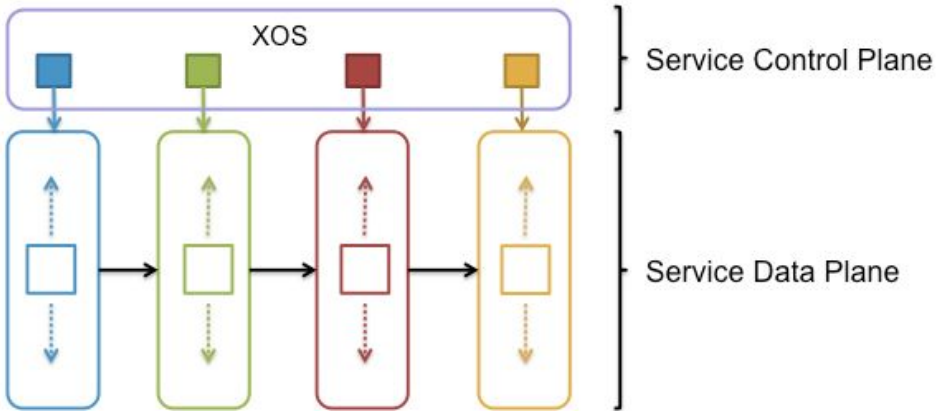

Figure 2. Service Control Plane (implemented by XOS) manages a Service Data Plane (implemented by a set of micro-services).

Figure 2 shows the playbooks as part of XOS (the solid colored squares), but that is just part of the story. Figure 3 shows a more complete picture, which includes a set of containers organized into three layers: the bottom layer implements a collection of *Synchronizers* (each container in

this set includes a micro-service playbook); the middle layer implements a *Data Model* (it includes an event bus, a persistent store, and the XOS Core); and the top layer implements various *Views* (each of which defines a different user interface for accessing the data model). Although shown as individual containers in Figure 3, each scales as an independent micro-service.

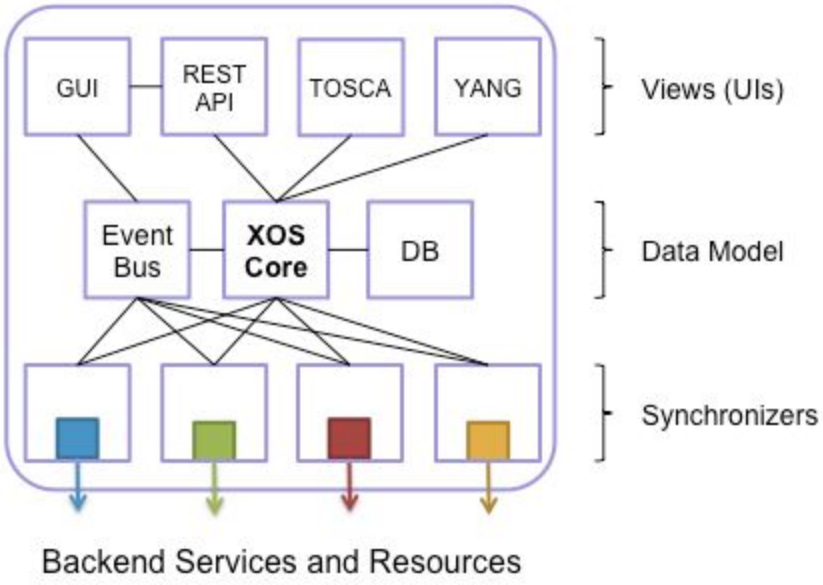

Figure 3. XOS internal organization: A collection of containers (micro-services) organized into three layers: Views, Data Model, and Synchronizers.

The interaction between XOS and the backend micro-services is similar to that of an SDN controller and a set of switches in the data plane: XOS *configures* and *controls* the micro-services, but it does not interject itself into inter-service communication, with one important exception: XOS mediates one micro-service *acquiring tenancy* in another micro-service. But once tenancy is acquired, communication is direct between micro-services (in the service data plane) without XOS involvement.

By mediating trust, XOS relaxes the micro-service assumption that all micro-services operate in a single trust domain. Different domains interact with XOS serving as a trusted intermediary. By defining a logically centralized data model, XOS is able to explicitly model all inter-service dependencies and apply policies uniformly across them, thereby creating a configurable platform rather than a fixed cloud application. By separating the service control plane (which models a service) from the service data plane (which implements a service) XOS is able support a range of backend service implementations, including both server-based (running in containers or using some other virtualization mechanism) and switch-based (running as flow rules installed in the underlying switches).

Being able to model all inter-service dependencies is also important because it provides a logical representation of the system that is easy to reason about. This makes it possible to

define a set of policies and invariants about how the system should behave as a whole, where Synchronizers then localize the complexity of dealing with the operational components.

Another useful perspective about the role of XOS is that it roughly corresponds to Ansible Tower combined with Model-Driven Engineering (MDE). Ansible Tower curates a library of Ansible playbooks, and can trigger them at particular times, in response to particular events, or as part of larger workflows. This approach is effective for scaling IT automation, but is largely mechanistic and lacks higher-level abstractions that can help with reasoning about a complex domain. In contrast, the MDE methodology builds a software system around an *abstract model* of a domain. Benefits of MDE include accelerating and simplifying the design process via reuse of standardized models and recurring design patterns. In essence, XOS layers a lightweight MDE discipline over something like Ansible Tower: models in XOS abstractly represent the domain to be orchestrated, where changes in the models trigger Ansible playbooks to run using information stored in the model.

The next section discusses the role of the Data Model at the heart of XOS in more detail, but we conclude this section by showing how XOS is applied in the specific case of CORD. As depicted in Figure 4, a common configuration of a CORD POD consists of XOS serving as the CORD Controller, a network operating system (e.g., ONOS) controlling the virtual and physical switches, a set of SDN control applications running on ONOS, one or more Compute-as-a-Services (e.g., OpenStack and/or Docker Swarm) managing a set of VMs and/or containers, and a set of VNFs running in those VMs/containers. Because *everything* is a service, even platform components like ONOS and the selected CaaS(es) are treated as services.

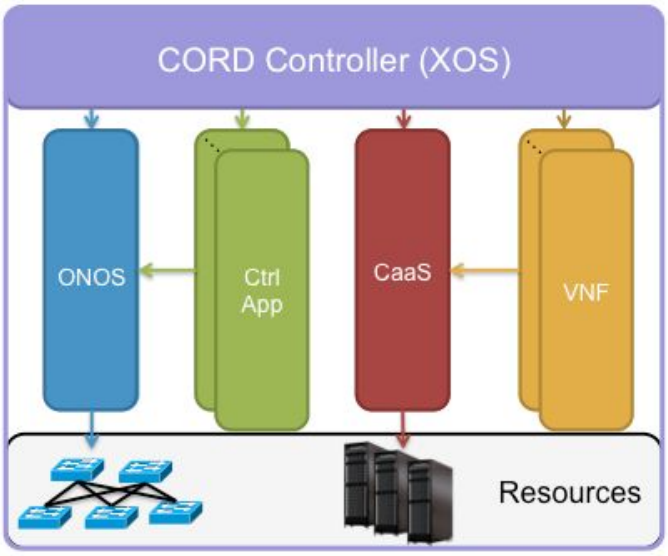

Figure 4. A typical CORD POD, consisting of XOS (serving as the CORD Controller) managing a set of backend micro-services.

While Figure 4 shows the control apps dependency on ONOS and the VNFs dependency on some CaaS, it does not show the relationships and dependencies among the VNFs and control

apps. These dependencies are managed by the CORD Controller (i.e., XOS), as discussed in the next two sections.

XOS: Modeling Framework

XOS both defines the authoritative state for the system and enforces a set of policies and invariants that govern the system's behavior. The latter includes mediating trust among the micro-services and assembling the data path through the underlying servers and switches. At the core of XOS (implemented by the "XOS Core" container shown in Figure 3) is a modeling framework. This framework consists of two parts.

First, XOS defines a language for specifying data models. This language is based on Google's protocol buffers (protobufs), borrowing their syntax, but extending their semantics to express additional behavior (see below). Although these extensions can be written in syntactically valid protobufs (using the protobuf option feature), the resulting model definitions are cumbersome and their semantics are under-specified. The XOS Core therefore introduces an alternative syntax, called xproto, that is a syntactic wrapper around protobufs. xproto is syntactically cleaner than protobufs, but also highlights the fact that the language's semantics are not fully captured by protobufs. Users are free to define models using standard protobufs instead of the xproto syntax, but doing so obscures the fact that packing new behavior into the options field renders protobuf's semantics under-specified. With this understanding, we refer to the modeling language as xproto throughout the rest of this document.

Whereas protobufs facilitate one operation on models—namely, data serialization—xproto goes beyond protobufs to provide a framework for implementing custom operators. This includes explicit support for the following:

- Relationship Operators → define relationships among models
- Policy Operators → logic formulae that produce "yes" or "no" verdicts
- Map/Reduce Operators → split/merge single values into/from multiple buckets

Second, XOS provides a tool chain for generating code based on a set of models loaded into the XOS Core. This generative tool chain, as illustrated in Figure 5, first translates xproto model specifications into an internal representation (currently implemented as a Python dict), and then applies a per-target Jinja2 template to this internal representation to produce a given target. Example targets that can be generated from the xproto-based data model include:

- Object Relation Mapping (ORM) – maps the data model onto the underlying database.
- gRPC Interface – how all the other containers communicate with XOS Core.
- TOSCA API – one of the UI/Views shown in Figure 3.
- Security Policies – governs which principals can read/write which models.
- Synchronizer Framework – execution environment in which Ansible playbooks run.
- Hierarchical CORD – distribute data model over a set of CORD domains.

- Auto-virtualizer – enforce multi-tenancy on micro-services.
- Unit Tests – auto-generate API unit tests.

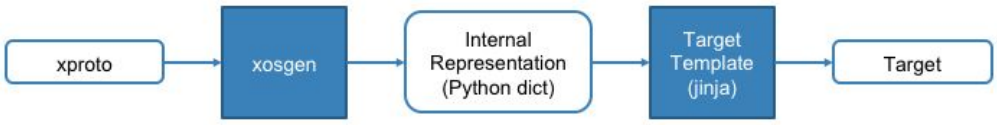

Figure 5. XOS generative tool chain.

As an example, consider the following xproto definition for the ***Privilege*** model, which represents *read*, *write*, and *grant* rights that have been granted to users. This model is policed by the ***grant_policy*** policy, which governs when the user has the right to grant a privilege to another user.

```
policy grant_policy < ctx.user.is_admin
                    | exists Privilege:Privilege.object_type = obj.object_type
                     & Privilege.object_id = obj.object_id
                     & Privilege.accessor_type = "User"
                     & Privilege.accessor_id = ctx.user.id
                     & Privilege.permission = "role:admin" >

message Privilege::grant_policy (XOSBase) {
    required int32 accessor_id = 1 [null = False];
    required string accessor_type = 2 [null = False, max_length=1024];
    required int32 controller_id = 3 [null = True];
    required int32 object_id = 4 [null = False];
    required string object_type = 5 [null = False, max_length=1024];
    required string permission = 6 [null = False, default = "all", max_length=1024];
    required string granted = 7 [content_type = "date", auto_now_add = True, max_length=1024];
    required string expires = 8 [content_type = "date", null = True, max_length=1024];
}
```

In this example, the policy is executed relative to three implied inputs: (1) the object and model on which the policy is invoked (e.g., ***obj.object_type***), (2) the context in which the policy is invoked (e.g., ***cxt.user***), and (3) the data model as a whole (e.g., ***"role:admin"***).

This example model includes a set of primitive field types (e.g., ***int32***, ***string***), but does not include any foreign keys (relationships to other models). An example that does is the following definition of a ***ServiceInstance***, which includes a many-to-many relationship to other service instances.

```
message ServiceInstance (XOSBase, AttributeMixin) {
    optional string name = 1 [db_index = False, max_length = 200, null = True,  blank = True];
    required manytoone owner->Service:service_instances = 2 [db_index = True, null = False, blank = False];
    optional string service_specific_id = 3 [db_index = False, max_length = 30, null = True, blank = True];
    optional string service_specific_attribute = 10 [db_index = False, null = True, blank = True,  varchar = True];
}
```

These examples are intended as simple illustrations of how xproto is used. For a complete specification see https://guide.opencord.org/xos/dev/xproto.html, and for the full set of core models see https://github.com/opencord/xos/tree/master/xos/core/models.

The bottom line is that XOS generates much of the software that “binds” the elements of CORD together, approximately 120k lines-of-code in the latest version. Figure 6 depicts where this code is embedded in the system.

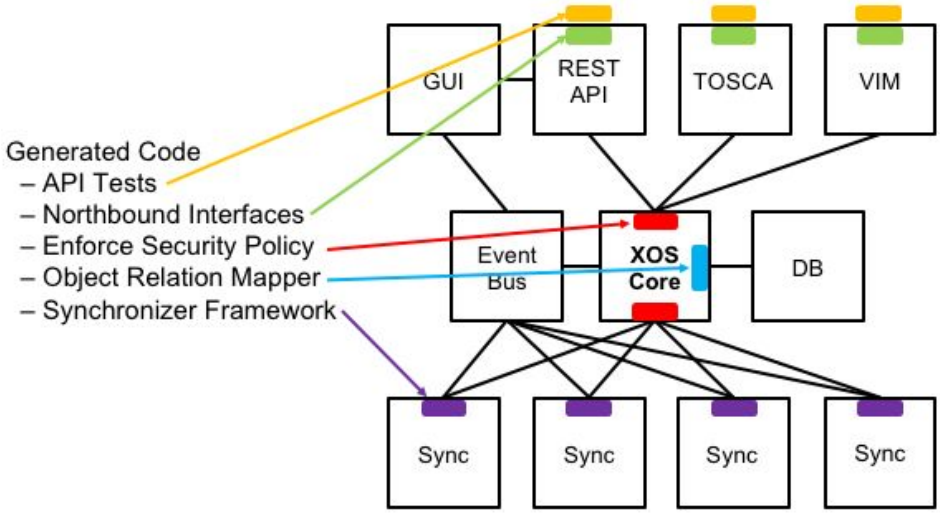

Figure 6. Generated Software in CORD.

XOS: Core Models

The XOS modeling framework is a software tool for building a system like CORD, but it’s also necessary to define a set of core models. These core models—which evolved out of years of experience with CORD and CORD-like systems—effectively specify the system’s architecture. For CORD, this starts with the ***Service*** model, which generalizes the micro-services architecture outlined in an earlier section. The CORD data model is defined elsewhere, so we focus here on three key ideas.

First, a Service is bound to a set of ***Slices***, each of which represents a combination of virtualized compute resources (both containers and VMs) and virtualized network resources (virtual networks). Whether a given service is implemented by a VNF image that runs in a virtual compute instance (a so-called *server-based service*) or an SDN control application that installs flow rules in a white-box switch (a so-called *switch-based service*) is an implementation detail in the sense that the service developer decides the right approach.

Second, a ***ServiceDependency*** model represents the binding of a *subscriber service* to a *provider service*. This dependency is parameterized by a ***ConnectMethod*** that defines how the

two services are interconnected in the underlying network data plane. The approach is general enough to interconnect two server-based services, two switch-based services, or a server-based and a switch-based service pair. This makes it possible to construct a *service graph* without regard to how the underlying services are implemented in the underlying servers and switches.

Third, for a service graph defined by a collection of Service and ServiceDependency models, every time a subscriber requests service (e.g., connects their cell phone or home router to CORD), a ***ServiceInstance*** object is created to represent the virtualized instance of each service traversed through the service graph on behalf of that subscriber. Different subscribers may traverse different paths through the service graph, based on their customer profile, but the end result is a linked sequence of ServiceInstance objects, forming an end-to-end, per-subscriber *service chain*. Figure 7 shows a set of ServiceInstances associated with the service graph from the Residential deployment of CORD (R-CORD). Note that this example is simpler than the general case, which permits both the service graph and the per-subscriber service chains to be arbitrary directed acyclic graphs.

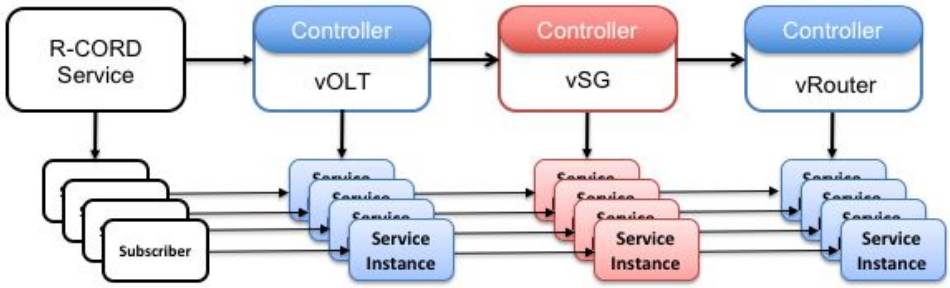

Figure 7. Example service graph and associated per-subscriber service chains.

Importantly, this service chain corresponds to a list of ServiceInstance objects, where each node in the list represents some combination of virtualized compute and network resources; the service chain is not necessarily implemented by a sequence of containers or VMs. That would be one possible incarnation in the underlying service data plane, but how each individual service instance is realized in the underlying resources depends on how the selected set of services happen to be implemented (e.g., as a legacy VNF running in a VM, as a micro-service running in containers, or as an SDN control app running on top of ONOS). Moreover, because the data model provides a way to represent this end-to-end service chain, it is possible to access and control resources on a per-subscriber basis, in addition to controlling them on a per-service basis.[1] This is an essential element supporting end-to-end isolation for subscribers across a sequence of multi-tenant services running on a multi-tenant platform.

[1] Confusingly, the set of resources bound to a Service is called a “Slice” in XOS, while the set of resources bound to a each subscriber (or class of subscribers) is often called a “Network Slice” in the emerging 5G arena. Both are modeled in XOS, with the latter (Network Slice) spanning a collection of the former (XOS Slices).

Architectural Approach

The word architecture means different things to different people. To a software engineer, it typically corresponds to the software stack—the collection of software modules, their interfaces, and the dependency among the modules. To a network operator, it often corresponds to the hardware equivalent—a wiring diagram for a collection of devices and switches.

In both cases, there are best practices that guide the evolution of the architecture over time, but these guiding principles are implicit, often in the heads of the people responsible for building the system. They also accept a set of user requirements as input, but it is the engineer's responsibility to translate those requirements into running code, and when it's necessary to adapt or extend the architecture, to do so in a way that's consistent with best practices. In the context of cloud applications, for example, implementing everything as micro-services is an example best practice, but it is effectively cloud-speak for adhering to a modular design (done in a way that allows each module to scale independently).

At a higher level, an architecture is generally taken to imply something more than the set of building blocks and how they are assembled. It also includes the set of rules or invariants that guide how the building blocks are all assembled, reconfigured, and extended onto over time. This includes "be modular" but typically goes beyond modularity in some domain-specific way. Because software is infinitely malleable, an architecture is often a statement about what engineers *cannot* do as they evolve the system to meet new requirements. One challenge is how to enforce those rules, so as to avoid the situation where the system remains modular, but is the software equivalent of the Winchester house.

On the flip slide, these architectural rules might be wrong or short-sighted, resulting in a system that cannot be cleanly extended to meet some new requirement. This might imply it's time for a fresh start (clean slate), but quite often the system continues to evolve with little or no architectural integrity, again leading to a convoluted and brittle system.

The key idea behind the model-driven approach adopted by CORD is to codify the "architectural definition of the system" in a data model. This includes the models and the relationships among them, but it also includes a set of policy statements and invariants made about those models. We see the model definitions and policy statements as "jointly authored" by the architects (who evolve the definition of the system based on their experience trying to meet user requirements) and engineers (who evolve the definition of the system based on their experience trying to implement the system). But instead of this definition being in words in a requirement document, it is executable code (expressed in *xproto*, in our case), where the generative toolchain then outputs the elements that realize and enforce those definitions.