# Paradigm: Fault-Tolerant Empirical Megamodeling

R. Riley Holmes

November 16, 2025

**Abstract**

There is central tension in the problem of model management, between formal correct-by-construction approaches and real-world integration challenges involving imperfect legacy tooling. We present Paradigm, a prototype megamodeling framework that embraces fault-tolerant architecture to organize and assess structural relationships across heterogeneous resources.

Paradigm's protocol-based design provides an abstraction layer for data sources (XMI documents, databases, APIs) and transforms (structured bidirectional operations, command-line code generation tools). This allows diverse artifacts and tools to be treated uniformly within a single plane of validation, where we can observe instance propagation and perform runtime conformance checks at transformation boundaries.

We demonstrate the framework's capabilities with multi-protocol (Protobuf, Thrift, Avro) schema and message translation, detecting transformation failures and subtle semantic inconsistencies that would otherwise manifest as downstream bugs, while allowing integration of existing tools without modification. The framework's concurrent architecture also supports real-time collaborative workflows, enabling interactive probing of transformation pipelines and collaborative model evolution.

## 1 Introduction

Every organization has some collection of resources: data models, data, and code for moving data around. Often the same data exists in multiple representations. When there's some unexplained discrepancy between representations, that's a bug: the data must be corrected, and (if possible) the root cause addressed. This is historically a costly process. Data modeling frameworks have the potential to reduce complexity and speed up integration, whether for low-level data interchange (Thrift, Protobuf, Avro) or high-level system models (SysML, FMI, AADL).

As complexity grows at the model level through versioning of metamodels and models, ad-hoc solutions to model management will quickly show signs of stress. After all, models are typically more complex than the objects they describe. Subtle consistency errors live 'outside the system', introducing difficult-to-trace integration problems. An organization may quickly rocket past the breakeven point where it would have been more efficient to simply throw man-hours directly at data integration rather than brittle model manipulation infrastructure that requires specialized knowledge to maintain and debug.

While there are robust tools available for metamodel-aware model management, we envisioned a higher level of freedom and flexibility, and recognized a niche for an open-source megamodeling framework. Our goal is a megamodeling framework that can work in two roles: 1) a model management tool for checking the self-consistency of a system of data sources and transform pipelines, and 2) a command-line witness of integration requirements that can be smoothly incorporated into build processes to derive end products. Paradigm achieves this through two key mechanisms: protocol-based abstraction of resources, and instance propagation within a content-addressed megamodel that makes transformation behavior observable.

## 2 Background and Related Work

### 2.1 Property Graphs

Property graphs have emerged as a central abstraction for managing complex, interconnected data across diverse domains. The rich ecosystem of graph database implementations and query languages has led to integration challenges [1], but recent standardization efforts suggest convergence toward common abstractions: ISO 39075's establishment of Graph Query Language (GQL) in April 2024 represents the first new database query language standardized by ISO in over 35 years [2]. Surveys of graph processing systems note [3] that interoperability with cross-domain workflows and tooling remains one of the vital challenges that is unsupported by existing graph frameworks.

### 2.2 Data Pipeline Orchestration

Systems like Apache Airflow [4] and dbt manage general-purpose data transformation workflows with emphasis on observability, error recovery, and incremental materialization. They prioritize operational concerns: monitoring, retry logic, dependency management. These systems excel at heterogeneous tool integration, but lack built-in mechanisms for validating transformation semantics and analyzing model relationships.

### 2.3 Mathematical Approaches to Semantic Validation

Functorial data migration [5] provides a category-theoretic characterization of data integration that guarantees the preservation of semantic relationships. Related work formalized [6] Uber's proprietary Dragon platform into algebraic property graphs. Their case study demonstrated the efficacy of a universal ('dragon') meta-model for cross-protocol data validation. This was succeeded by the open-source Hydra language [7] which embeds executable code within algebraic graph structures for comprehensive language translation, along with handling of schemas and data across data exchange languages. PARADIGM pursues similar cross-protocol integration structures, but with a focus on rapid integration of existing transformation tools without formal specification of their semantic properties.

### 2.4 Model Management and Megamodels

Model management concerns the definition of operations (`match`, `merge`, `diff`, etc) on models to solve integration problems [8]. System-level models capturing the relationships between models have been designated 'megamodels' [9], or 'macromodels' [10]. PARADIGM defines a megamodel that is relatively rudimentary, but represents a reproducible artifact connecting disparate data sources and transformation tools.

PARADIGM treats all objects under consideration (megamodels, metamodels, models, and data) in a unified plane with the same graph abstractions - even software platforms and generated packages, which are normally treated as conceptually separate from models [11]. This provides a useful level of introspection, and we demonstrate dynamical evolution in a "mega-megamodel" setting.

### 2.5 Type-Safe Model Transformation

The Eclipse Modeling Framework (EMF) represents a mature implementation of traditional model-driven development, and enforces structural invariants through metamodel-driven code generation. PARADIGM's bootstrap metamodel draws inspiration from the Essential Meta Object Facility (EMOF) [12], sharing foundational abstractions of Class, Property, and Package. EMF prioritizes GUI-based metamodel design requiring extensive point-and-click workflows [13], which creates barriers to integration with command-line systems and composable transformation workflows.

ATL (ATLAS Transformation Language) [14] and QVT (Query/View/Transformation) [15] similarly prioritize correctness by construction through declarative transformation rules. While these approaches offer strong theoretical foundations, they require significant upfront investment and often necessitate rewriting existing transformation logic to fit their type systems. PARADIGM prioritizes flexibility, and ATL or QVT-based transforms could potentially be integrated as first-class transformation primitives in future work.

## 2.6 Build System Reproducibility

Modern build systems like Bazel [16] and Buck2 [17] achieve reproducibility through hermetic execution and content-addressed caching, along with explicit handling of dependencies. Nix [18] extends this approach to entire software environments, treating all dependencies as immutable, content-addressed objects. Empirical evidence demonstrates the power of the functional package management model: `nixpkgs` is the largest existing cross-ecosystem open source package repository, with excellent expectations of bit-for-bit reproducibility [19].

Model management would seem to benefit greatly from reproducibility, cross-platform provenance, and reliable composition of tools. We'll therefore embrace Nix via use of Flakes [20] in: 1) the execution environment of Paradigm itself, 2) the addition of runtime dependencies for transforms, and 3) specification of generated code packages. The fact that the entire computational environment can be efficiently and exactly specified doesn't seem to have been widely leveraged by model-based engineering toolchains.

## 2.7 Elixir, OTP, and Protocols

Elixir is a high-level language that runs on Erlang's BEAM virtual machine. The VM was designed for highly scalable fault-tolerant applications using the Open Telecoms Platform (OTP) library. It leverages lightweight isolated processes [21] arranged in supervision trees [22]. The Elixir language introduced ergonomic syntax and metaprogramming primitives [23] that have led to an enthusiastic dev ecosystem, high-profile adoption [24], and the highest performance upper-bounds in LLM code generation benchmarks [25].

*Protocols* are a language feature, first introduced in Clojure in 2010 [26] as an alternative to interfaces. They enable a more powerful decoupling mechanism for polymorphism, allowing type definitions and their protocol implementations to originate from different compilation units and be composed post-hoc. Their implementation in Elixir [27] brings a powerful abstraction to the BEAM ecosystem, and Paradigm leans heavily on their use to achieve generic model manipulation algorithms with little syntactic overhead.

## 2.8 Positioning Paradigm

Paradigm borrows inspiration from all these systems, and fills the gap between theoretical rigor and practical adoption. It provides a lightweight meta-integration layer that allows observation of graph transformations and rigorous structural checks at each transformation boundary.

This approach enables three key capabilities that existing frameworks cannot easily provide: First, arbitrary command-line tools can be integrated as transforms without modification—a shell script that converts CSV to JSON becomes a first-class transformation with full validation. Second, transformation failures are immediately observable through traceable instance data. Third, organizations can incrementally formalize existing workflows without abandoning working systems.

We demonstrate this philosophy through integration scenarios that can be solved using an iterative megamodeling approach. In each case, the solution encapsulates a reproducible belief about the self-consistency of the system - including all of its models, transforms, generated resources, and test cases.

## 3 Conceptual Framework

Paradigm's design centers on the idea of instance propagation - the flow of actual data values through transformation pipelines. This enables validation of complex model relationships across multiple abstraction levels. In this section we will show how this idea applies to basic model management tasks, and to system-level megamodeling.

### 3.1 Instance Propagation

By *instance*, we mean the actual realized data values that populate a schema, as distinct from the schema definition itself. A filesystem schema defines abstract concepts like folders, files, and hierarchical containment relationships, while a filesystem instance contains concrete data that could be materialized onto physical storage. We adopt a diagrammatic convention of showing these realization relationships vertically:

$$\begin{tikzcd}
\mathit{Schema} & \mathit{Model} & \mathit{Metamodel} \\
\mathit{Instance} \arrow[u, dashed] & \mathit{Data} \arrow[u, dashed] & \mathit{Model} \arrow[u, dashed]
\end{tikzcd}$$

Figure 1: This diagram asserts some relationships that should be mechanically checkable and falsifiable.

We'll depict transforms as horizontal mappings between schemas.

$$\begin{tikzcd}
\mathit{Schema1} \arrow[r, squiggly, "\mathit{transform}"] & \mathit{Schema2} & \Longrightarrow & \mathit{Schema1} \arrow[r, squiggly, "\mathit{transform}"] & \mathit{Schema2} \\
\mathit{Instance1} \arrow[u, dashed] & & \Longrightarrow & \mathit{Instance1} \arrow[u, dashed] \arrow[r] & \mathit{Instance2} \arrow[u, dashed]
\end{tikzcd}$$

Figure 2: The presence of a transform indicates the ability to move instances along its arrow. By applying `transform` to `Instance1` we produce `Instance2`.

The transform is fallible: it may fail hard with a runtime error. It may produce an output that is not actually a valid instance. It may (more subtly) produce an output that is valid but incorrect. For example, an empty output is trivially valid but generally not a desirable result.

### 3.2 Roundtrips and Other Equality Checks

Consider a common scenario, with 1) some code generation routine that can take `Protobuf` models and produce `Filesystem` artifacts like header files, and 2) a parsing utility that can read in header files and recover the corresponding model.

While these are intended to be opposite transforms, short of type-theoretic correct-by-construction routines, this assumption should be checked. Given either a `Protobuf` model or `Filesystem` headers, we insert the instance into the system, propagate it 2 steps, and see what happens.

$$\begin{tikzcd}
Protobuf \arrow[r, squiggly, shift left, "codegen"] & Filesystem \arrow[l, squiggly, shift left, "parse\_schema"]
\end{tikzcd}$$

Figure 3: A pair of inverse transforms.

$$\begin{tikzcd}
Protobuf \arrow[r, squiggly, shift left, "codegen"] & Filesystem \arrow[l, squiggly, shift left, "parse\_schema"] \\
Model \arrow[u, dashed] \arrow[r, bend left=10] & Artifacts \arrow[u, dashed] \arrow[l, bend left=10]
\end{tikzcd}$$

Figure 4: The expected behavior is to cycle back and forth between the same instances.

If the system is in a stable state, then further propagation doesn't produce any new instances. The transform has been validated *with respect to this instance.* With different data, we may discover a discrepancy.

$$\begin{tikzcd}
 & Protobuf \arrow[r, squiggly, shift left, "codegen"] & Filesystem \arrow[l, squiggly, shift left, "parse\_schema"] \\
Model^* & Model \arrow[u, dashed] \arrow[r, bend left=10] & Artifacts \arrow[u, dashed] \arrow[ll, bend left=5]
\end{tikzcd}$$

Figure 5: A machine-detectable roundtrip failure.

When an instance unexpectedly fails to roundtrip, the megamodel offers a handful of leads: The two non-congruent instances, the `diff` between them, the intermediate representation, and the ability to reproduce both of the transforms.

## 3.3 The Bootstrap Metamodel

We have not specified what things are "allowed" to be instantiated. To that end we bootstrap the system with a `Metamodel` describing the basic structure of classes and properties. Its key characteristic is that it can host itself as instance data. Now every valid `Metamodel` instance describes a schema, and can therefore be instantiated.

$$\begin{tikzcd}
Metamodel \arrow[loop above, dashed] \\
Model \arrow[u, dashed] \\
Data \arrow[u, dashed]
\end{tikzcd}$$

Figure 6: We can validate `Metamodel`, `Model`, and `Data` - in each case because it is an instance of a `Metamodel` instance.

With a 'second-layer' metamodel such as `Protobuf`, its models can be validated, but they are too far removed from `Metamodel` to be instantiated. A new transform serves the purpose of hoisting the model up abstraction levels. Then we can work with the model data in a canonical form and (with a little more work later) serialize it to and from `Protobuf` binary.

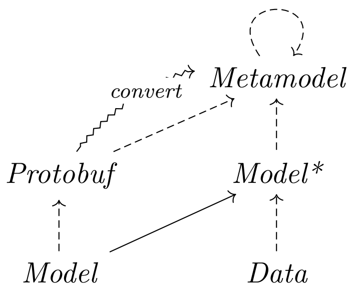


Figure 7: Moving the `Protobuf` model up a level where it can serve as a schema for something else.

This places a lot of pressure on the representational power of `Metamodel`. Whether this approach will prove sufficiently expressive for complex real-world scenarios remains an open question. We draw some confidence from EMOF (Essential Meta Object Facility), a similar meta-metamodel that underpins UML, but acknowledge that more specialized forms of validation (like OCL constraints) may require handling at the transform level—though exploring these extensions is beyond the scope of this work.

## 3.4 System-level Megamodels

The `Universe` megamodel introduces some computational meaning to the above diagrams. It contains graph objects and their instantiation relationships, transform definitions, and the traces of graphs that have been propagated. Everything - metamodels, models, data - are treated as graphs, including the megamodel itself: `Universe` is a `Metamodel` instance. If we permit ourselves to move to the 'mega-megamodel' level, we get a simple view of instance propagation as a transform from `Universe` to itself:

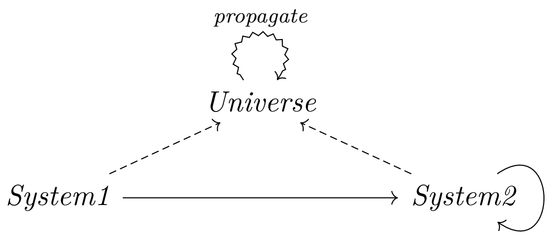


Figure 8: This `Universe` propagation depicts the progression of Figure 2. The stability of `System2` indicates that further propagation does not alter the diagram.

The `propagate` algorithm might apply 1 change at a time, or search for a steady-state. It might apply transforms one time or many. Rather than prescribing a universal propagation algorithm, we've focused on 2 other routes: manual user-driven evolution, or purpose-built operations detecting specific preconditions to further processing, such as schema stability prior to code generation.

## 4 Architecture and Implementation

With the overarching goal of flexible empirical validation, transformation using a diverse set of data sources and transforms, and a reproducible megamodel structure, we designed the Paradigm library. An open-source prototype is available on the Hex package manager [28]. The core implementation consists of approximately 6,000 lines of Elixir code organized into several key modules which we'll go over briefly for high-level architectural discussion.

### 4.1 The `Graph` Protocol

The `Graph` protocol is central to Paradigm's empirical validation approach. It defines a handful of basic methods for traversing graph data, enabling uniform treatment of disparate data sources. Implementers agree to produce `Node` structs that include an identifier, a class, ownership/containment information, and a data (property) map. References to other nodes are handled as `Node.Ref` structs.

The `Conformance` module validates instantiation relationships between two `Graph` objects by checking class and property relationships against the metadata and properties of the nodes. There are 15 distinct conformance issues defined that come back as highly structured exact feedback.

Listing 1: Validating the self-hosting of the bootstrap metamodel

```
metamodel_graph = Builtin.Metamodel.definition()
Conformance.assert_conforms(metamodel_graph, metamodel_graph)
```

The `Graph` protocol's design enables transparent integration of diverse data sources, from in-memory structures to live filesystem directories to Git repositories. This polymorphic approach allows the same conformance validation and transformation logic to operate uniformly across systems.

### 4.2 The `Transform` Protocol

The `Transform` protocol defines transforms as structures that consume a source graph and produce results into a target graph. It may require some particular graph `Graph` implementations (such as `Filesystem` inputs or outputs of CLI tools). The protocol defines a single function:

Listing 2: The single callback of the `Transform` protocol

```
def transform(transform, source, target, options \\ [])
```

Different implementations enable complementary integration patterns:

1. *Sandboxing and introspection*: AST analysis enables validation of transformation logic before execution
2. *Element-wise traceability*: Class-based transforms handle common patterns and allow fine-grained element-wise tracing
3. *Legacy integration*: Existing command-line tools are wrapped into first-class transforms

4. *Compositional validation and transform reuse*: Pipeline definitions glue together any number of other transforms

These implementation details remain abstracted at the megamodel level, allowing uniform treatment of diverse transformation approaches within the same validation framework.

## 5 Empirical Validation in Practice

We demonstrate Paradigm's capabilities through several integration scenarios. To support interactive exploration of these concepts, we've built a prototype web platform for real-time collaborative megamodeling, using OTP primitives for concurrent propagation operations. Interactive demos are available at paradigmpro.live, and we'll use those diagrams in the following examples.

### 5.1 Data Exchange Schema Translation

To demonstrate low-level data interoperability, we integrated 3 popular data exchange languages: Thrift, Protobuf, and Avro. Each format had an open-source library available on Hex [29], [30], [31]. The integration required similar library-specific connective pieces: schema handling, a Paradigm metamodel, and conversion routines to and from `Metamodel`.

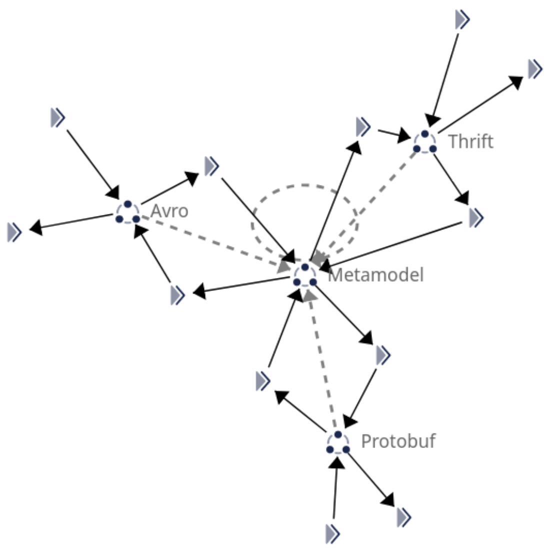


Figure 9: Cross-protocol integration structure, including serialization and deserialization functions treated as transforms lacking source or target models

We focused on representing simple data types: structs and fields, including arrays and nested structs. We ignored maps, unions, and language-specific constructs in favor of commonality. Altogether, each library took only about 300 lines of adapter code. The megamodel setting proved useful to define schema roundtripping and pass/fail conditions for validation, with useful feedback on failures.

We introduce a model by deserializing it from Thrift IDL:

Listing 3: The restaurant model

```
struct Restaurant {
    1: required string name,
    2: required list<MenuItem> menu_items
}

struct MenuItem {
    1: required string name,
    2: required string description,
    3: required double price
}
```

then propagate it through all the conversion functions. At the `Metamodel` representation, the propagation system recognizes (using the content hash) that it's arrived at a known model that's already in the database, so it's correctly labeled as the `Restaurant_Model`.

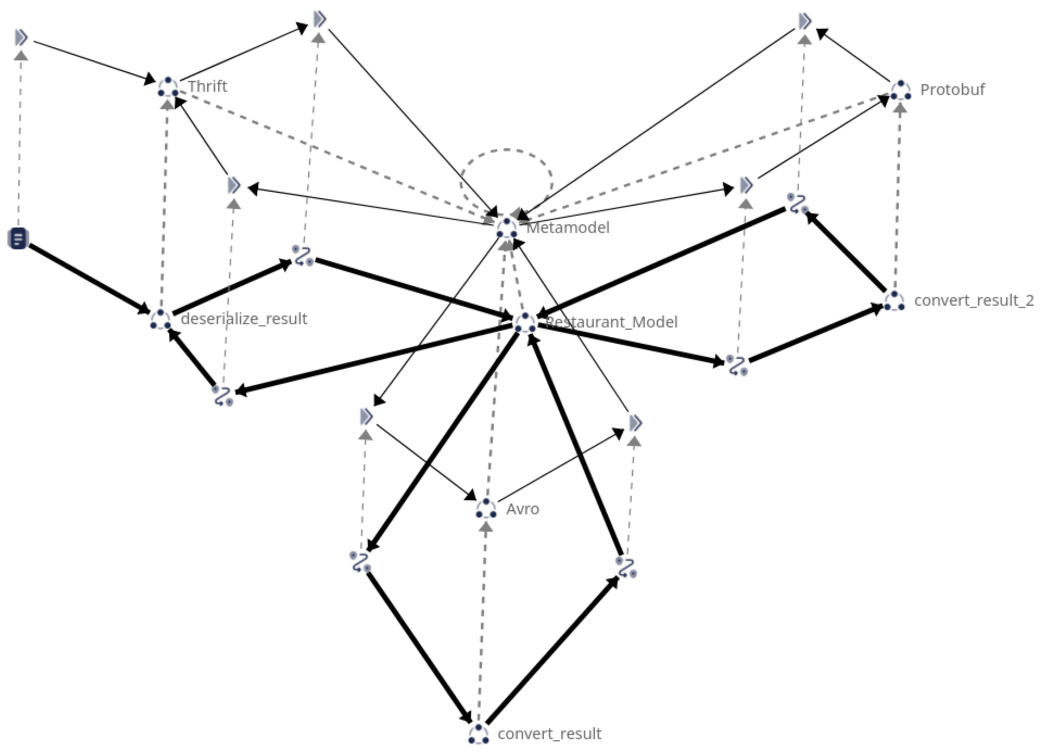


Figure 10: Schema circulation demonstrates successful roundtrips to different protocols

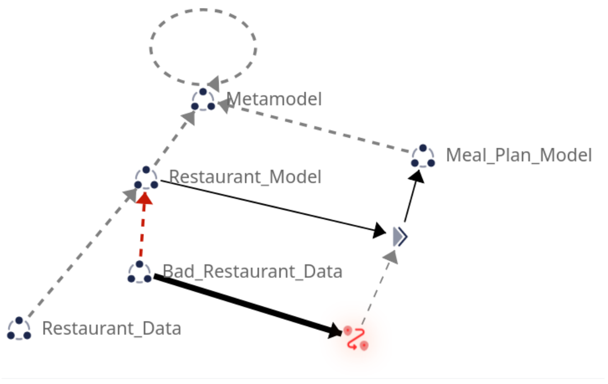
The roundtrip schema demonstrates stability, roundtripping to different protocol representations without data loss. This indicates we can generate equivalent schemas in each language, and use `Restaurant_Model` as a schema for validating canonical graph data. The foundation for full data-level interoperability is established, with only some code generation required to define serialization and deserialization functions.

## 5.2 Fault Detection and Recovery Mechanisms

The previous example demonstrates successful operation, but PARADIGM's fault-tolerant aspect becomes most valuable when things go wrong. We demonstrate several common failure modes and discuss how the framework supports diagnosis and correction.

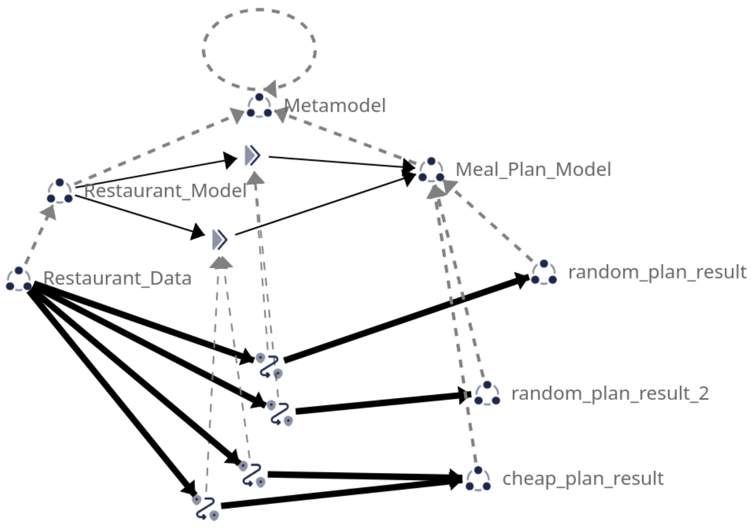
### 5.2.1 Hard Failure

Here transforms convert `Restaurant_Model` instances to `Meal_Plan_Model` instances. The transform's job is to pick out breakfast, lunch, and dinner from the available menu items. A conformance check reveals that `Bad_Restaurant_Data` contains numerous validation errors. But that doesn't stop us from trying to push it through a transform. The runtime error is handled by the megamodel execution environment. If the failure is unexpected, then the megamodel contains enough information to generate a (failing) unit test to hand off to developers.



### 5.2.2 Non-determinism

Now compare 2 transforms: one that always picks the cheapest item, and another that picks at random. Each can be re-run with the same input to probe for determinism. We get the expected result that the cheap meal plan gives the same result every time, while the random plan is usually different.



### 5.2.3 Roundtrip Failure

We'll demonstrate a realistic metamodeling failure encountered during development. We start by parsing the following Thrift schema into graph data.

Listing 4: A product model with non-consecutive field numbers

```
struct Product {
  1: required string name,
  3: required double price,
  5: required i32 inventory
}
```

After propagating the instance data we see two different `Thrift` instances: the return conversion doesn't coincide with the original parsed data. We can serialize the 2nd one and see it's now got field numbers 1, 2, 3.

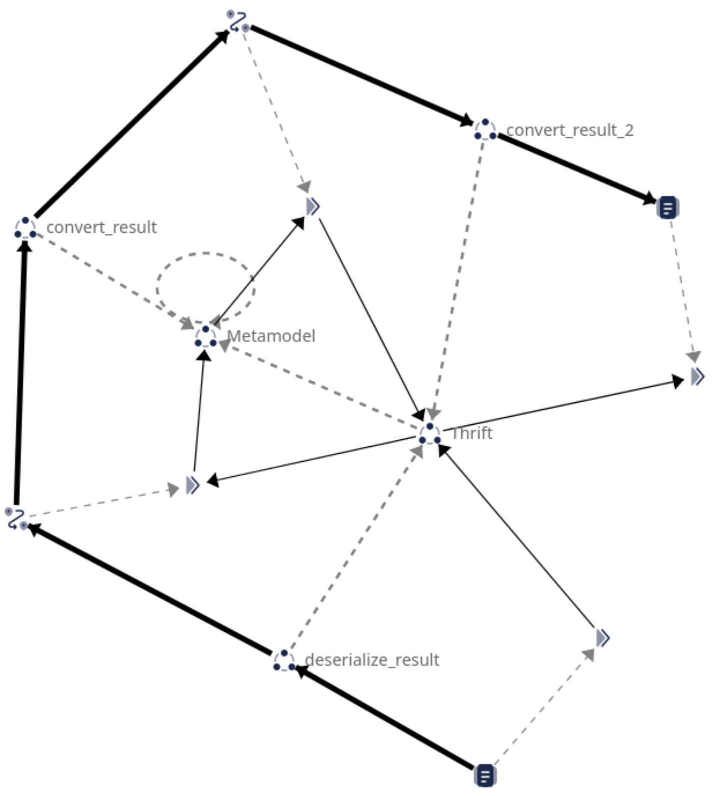


Figure 11: This model does not roundtrip to the `Metamodel` representation

The web UI supports exploring the megamodel's internal models and node properties to hunt down the problem: We see that the `Metamodel` instance has the correct 1, 3, 5 numbering, but on the conversion back to `Thrift` they're turned into consecutive numbers. This manual inspection allows us to isolate the data loss issue to a logic error in a specific transform.

For automated analysis, Paradigm supports automatic detection of failed roundtrips by analyzing transform endpoints, and providing a structured diff:

```
Property values changed:
```

```
Product_price.id      3 -> 2
Product_inventory.id  5 -> 3
```

This eliminates the need for node-by-node comparison.

## 5.3 Message Translation

Paradigm can be ‘configured’ at runtime as a cross-protocol message translation service. The first insight is that the evolution from Figure 9 to Figure 10 can be automated when a model is introduced. The resulting megamodel is assessed for roundtrip stability. For `Protobuf` we leverage `Protox`’s macro syntax, and Elixir’s willingness to compile and load new modules at runtime:

Listing 5: Runtime code generation of Protobuf module.

```
def create_protox_module(schema) do
  """
  defmodule GeneratedProtox do
    use Protox, schema: ~s\"\"\"
      #{schema}
    \"\"\"
  end
  """
  |> Code.eval_string()
end
```

This provides a protocol-specific encoding and decoding method for our schema. A thin layer over Phoenix PubSub iterates over protocols, allowing participants to receive messages in their native format of choice. The `Paradigm.Canonical` module produces standardized Elixir structs from graph data, which should be equivalent to library-generated structs (and thus serializable).

Listing 6: A cross-protocol PubSub service

```
def broadcast(struct) do
  for format <- [:elixir, :protobuf] do
    Phoenix.PubSub.broadcast(
        ParadigmInterop.PubSub,
        "paradigm:#{struct.__struct__}:#{format}",
        from_struct(struct, format))
  end
end

defp from_struct(struct, :elixir), do: struct
defp from_struct(struct, :protobuf), do: Protobuf.encode(struct)
```

This is an unusual example, as the model management framework is turned into a network participant. A more standard approach to code generation involves production of `Filesystem` instances with associated Nix artifacts. We have implemented proof-of-concept demonstrations from Protobuf schemas to compiled C++ packages, but comprehensive workflows for package generation and system integration merit dedicated treatment in future work.

### 5.4 Stateful Graph Sources: XML Document Store

While previous examples used in-memory graphs or filesystem-based sources, Paradigm's protocol design also supports integration with external systems. We demonstrate this by integrating `BaseX` [32], maintaining a persistent connection to query large XML documents in-place.

The `BaseXGraph` module uses Elixir's `Port` module to open a persistent connection to the `BaseX` CLI. Then the `Graph` protocol is implemented by generating and evaluating the appropriate `XQuery` code. `Paradigm`-aware context is added to upgrade node references and do explicit type resolution.

We validated this approach using an open systems architecture standard for avionics, with its relatively complex metamodel expressed in EMOF XMI. The stateful connection enabled direct validation of large models without requiring data migration — an essential capability for model-to-code generation processes that must work with industry-standard model repositories. This demonstrates that Paradigm can integrate with domain-specific standards and external document stores while providing the same conformance validation and transformation capabilities as other graph sources.

## 6 Conclusion

We presented Paradigm, a framework for model management with a focus on practical integration flexibility. We demonstrated that through instance propagation and protocol-based abstraction, organizations can incrementally formalize existing transformation pipelines into explicit and transparent model relationships.

The framework's ability to detect subtle consistency errors, integrate legacy tools without modification, and provide real-time collaborative workflows addresses key gaps in existing model management approaches. Future priorities, depending on organizational adoption patterns and funding, include some promising directions:

1. UML and SysML support to achieve interoperability with industry standards
2. Policy-driven development; Use of consistency requirements on megamodels to automatically generate unit tests and acceptance criteria
3. Organizational governance: Multi-party consensus protocols leveraging the Merkle tree structure as efficient witness to incremental introduction and acceptance of modeling resources in a blockchain structure

The open-source prototype is available for evaluation, and we welcome collaboration from anyone interested in advancing a Paradigm shift in model management.

## References

[1] M. E. Coimbra, L. Svitáková, A. P. Francisco, and L. Veiga, *Survey: Graph databases*, 2025. arXiv: 2505.24758 [cs.DB]. [Online]. Available: https://arxiv.org/abs/2505.24758.

[2] K. W. Hare, "ISO/IEC 39075 Database Language GQL: What is the database language GQL?" ISO/IEC JTC 1/SC 32/WG 3, Tech. Rep., Apr. 2024, Accessed: November 16, 2025. [Online]. Available: https://jtc1info.org/wp-content/uploads/2024/04/2024-Article-39075-Database-Language-GQL.docx.pdf.

[3] S. Sakr et al., "The future is big graphs! A community view on graph processing systems," *CoRR*, vol. abs/2012.06171, 2020. arXiv: 2012.06171. [Online]. Available: https://arxiv.org/abs/2012.06171.

[4] J. Yasmin, J. A. Wang, Y. Tian, and B. Adams, "An empirical study of developers' challenges in implementing workflows as code: A case study on apache airflow," *Journal of Systems and Software*, vol. 219, p. 112 248, Jan. 2025, ISSN: 0164-1212. DOI: 10.1016/j.jss.2024.112248. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2024.112248.

[5] D. I. Spivak, *Functorial data migration*, 2013. arXiv: 1009.1166 [cs.DB]. [Online]. Available: https://arxiv.org/abs/1009.1166.

[6] J. Shinavier, R. Wisnesky, and J. G. Meyers, *Algebraic property graphs*, 2022. arXiv: 1909.04881 [cs.DB]. [Online]. Available: https://arxiv.org/abs/1909.04881.

[7] Categorical Data, *Hydra: Graph programming language*, https://github.com/CategoricalData/hydra, Accessed: 2025-11-02, 2024.

[8] P. A. Bernstein, "Applying model management to classical meta data problems," in *Conference on Innovative Data Systems Research*, 2003. [Online]. Available: https://api.semanticscholar.org/CorpusID:5859503.

[9] J.-M. Favre, "Megamodelling and etymology," Jan. 2005.

[10] R. Salay, J. Mylopoulos, and S. Easterbrook, "Using macromodels to manage collections of related models," in *CAiSE*, Springer, 2009, pp. 141–155.

[11] A. Rodrigues da Silva, "Model-driven engineering: A survey supported by the unified conceptual model," *Computer Languages, Systems & Structures*, vol. 43, pp. 139–155, 2015, ISSN: 1477-8424. DOI: https://doi.org/10.1016/j.cl.2015.06.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1477842415000408.

[12] Object Management Group, "Meta object facility (mof) core specification," Object Management Group, OMG document formal/2013-06-01, version 2.4.1, Jun. 2013.

[13] H. Banerjee, *Design and implementation of a domain-specific language for modelling evacuation scenarios using eclipse emg/gmf tool*, 2025. arXiv: 2509.06688 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2509.06688.

[14] F. Jouault and I. Kurtev, "Transforming models with atl," vol. 3844, Jan. 2005, ISBN: 978-3-540-31780-7. DOI: 10.1007/11663430_14.

[15] I. Kurtev, "State of the art of qvt: A model transformation language standard," in *Applications of Graph Transformations with Industrial Relevance*, A. Schürr, M. Nagl, and A. Zündorf, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 377–393, ISBN: 978-3-540-89020-1.

[16] Bazel Authors, *Bazel: A fast, scalable, multi-language and extensible build system*, A fast and scalable build system for software of any size, 2015. [Online]. Available: https://github.com/bazelbuild/bazel.

[17] Meta Platforms, Inc., *Buck2: Build system, successor to buck*, A fast, hermetic, multi-language build system, 2024. [Online]. Available: https://github.com/facebook/buck2.

[18] E. Dolstra, "The purely functional software deployment model," PhD Thesis, Utrecht University, Utrecht, The Netherlands, 2006.

[19] J. Malka, S. Zacchiroli, and T. Zimmermann, *Does functional package management enable reproducible builds at scale? yes*, 2025. arXiv: 2501.15919 [cs.SE]. [Online]. Available: https://arxiv.org/abs/2501.15919.

[20] NixOS Community, *Flakes*, https://nixos.wiki/wiki/Flakes, NixOS Wiki. Accessed: 2025-11-02, 2024.

[21] Ericsson AB. "Processes." Erlang/OTP Efficiency Guide, Release 23, Accessed: Nov. 9, 2025. [Online]. Available: https://www.erlang.org/docs/23/efficiency_guide/processes.

[22] Ericsson AB. "Supervision principles." Erlang/OTP Design Principles User's Guide, Accessed: Nov. 9, 2025. [Online]. Available: https://erlang.org/documentation/doc-4.9.1/doc/design_principles/sup_princ.html.

[23] C. McCord, *Metaprogramming Elixir: Write Less Code, Get More Done (and Have Fun!)* Raleigh, NC: The Pragmatic Bookshelf, 2015, ISBN: 978-1-68050-041-7. [Online]. Available: https://pragprog.com/titles/cmelixir/metaprogramming-elixir/.

[24] Monterail. "Famous companies using elixir," Monterail, Accessed: Nov. 9, 2025. [Online]. Available: https://www.monterail.com/blog/famous-companies-using-elixir.

[25] J. Chou et al., *Autocodebench: Large language models are automatic code benchmark generators*, 2025. arXiv: 2508.09101 [cs.CL]. [Online]. Available: https://arxiv.org/abs/2508.09101.

[26] Clojure Core Team, *Protocols*, https://clojure.org/reference/protocols, Clojure Documentation. Accessed: 2025-11-02, 2024.

[27] Elixir Core Team. "Protocols." Elixir Documentation, Elixir, Accessed: Nov. 9, 2025. [Online]. Available: https://hexdocs.pm/elixir/protocols.html.

[28] R. R. Holmes, *Paradigm: An experimental megamodeling framework*, https://hexdocs.pm/paradigm/readme.html, Accessed: 2025-11-06, 2025.

[29] Pinterest, *Elixir-thrift: A pure elixir thrift implementation*, Pinterest, 2024. [Online]. Available: https://github.com/pinterest/elixir-thrift.

[30] BEAM Community, *Avro_ex: An avro library for elixir*, Version 2.2.0, 2024. [Online]. Available: https://github.com/beam-community/avro_ex.

[31] A. Hamez, *Protox: A fast, easy to use and 100% conformant elixir library for google protocol buffers*, Version 2.0.4, 2025. [Online]. Available: https://github.com/ahamez/protox.

[32] C. Grün, *Storing and Querying Large XML Instances* (Dissertationen der Universität Konstanz). Konstanz, Germany: Südwestdeutscher Verlag für Hochschulschriften, 2010, ISBN: 978-3838124596.