DRR documentation

The **Digital Regulatory Reporting (DRR)** programme is an open-access, cross-industry initiative to transform regulatory reporting from a painful chore into a streamlined, repeatable, and efficient process, reducing time, resources, and cost.

Built by extending the **FINOS Common Domain Model (CDM)**, complex regulatory text is converted into transparent, testable, and reusable logic through industry interpretation and collaboration. It provides the industry with an open-access, functional expression of the reporting rules that act as a common foundation and frees people from the grind of decoding rules so they can focus on actual business, not bureaucracy.

At its core, DRR brings together firms and SMEs to interpret the reporting requirements for each regime, and convert these requirements into unambiguous, machine-executable code. Using this mutualized industry interpretation of reporting rules, anyone can reuse the logic to generate regulatory reports in the required formats. By expressing these rules in a standardised, executable form, it enables firms, vendors, and regulators to align on a single, authoritative interpretation of reporting requirements.

DRR can reduce implementation cost, improve reporting accuracy, and increase transparency across the entire regulatory reporting chain. It supports global jurisdictions, provides a shared data and process model, and allows firms to test, validate, and evolve reporting logic collaboratively. With DRR, reporting becomes more predictable, more consistent, and easier to maintain as regulations change.

This documentation will guide you through the process from understanding the model, to running the services, to contributing to the future of DRR.

What's covered

This documentation offers a general introduction to DRR, including background, theory, and best practices. It also includes practical guidance related to DRR implementation and management.

What's not covered

This is not a guide to creating models for DRR. Although technical examples are provided, in-depth analysis of code or implementation techniques are not covered. It assumes an

understanding of basic coding principles although in-depth development expertise is not required. See the DRR prerequisites.

Explore the documentation

Get started

What DRR does, how it does it, and how it helps firms automate their regulatory reporting.

Using DRR

Getting technical – explore how DRR is structured, how to implement it and how to work it.

Tutorials

Step by step guides, including the full DRR data workflow.

Reference guides

Quick-access reference materials to help you use DRR.

Resources

FAQs, glossary and more to support your understanding of DRR.

Governance

The systems that keep DRR at the forefront of regulatory reporting.

Get involved

How to contribute and help shape the evolution of DRR.

Get started

Begin your DRR journey with the foundational concepts you need to understand the model, the tooling and the workflow.

Prerequisites

What you need before you can start working with DRR.

Introducing DRR

A high-level overview of the DRR initiative, its goals and structure.

How DRR works

A walkthrough of the DRR pipeline and how it executes regulatory logic.

DRR and CDM

Learn how DRR uses the Common Domain Model to build consistent logic.

Prerequisites and dependencies

What you'll need to get started with DRR:

- **CDM Data:** Transaction and event data converted to the Common Domain Model (CDM).
- **Rune DSL:** An understanding of the domain-specific language used for modelling CDM and DRR logic.
- **Access to DRR from ISDA:** Please contact CDMDRR@isda.org
- **Development environment:** e.g. Eclipse, the Rosetta platform or other environment capable of processing DRR.
- **DRR:** A working knowledge of principles and implementation of Digital Regulatory Reporting.
- **CDM:** An understanding of the principles behind the CDM open-source data standard.

Introducing DRR

Digital Regulatory Reporting (DRR) is an open-access, cross-industry initiative to transform regulatory reporting requirements into an industry-agreed implementation of unambiguous, machine-executable code. Instead of relying on lengthy rulebooks and firm-specific interpretations, industry participants collaborate to interpret and build the open-access DRR model, making transaction reporting more efficient and cost effective than ever. DRR extends the Common Domain Model (CDM) for the purpose of regulatory reporting, leveraging the open-source data standard for financial products, trades, and lifecycle events.

DRR is designed to provide **standardised and collaborative implementation of reporting logic** that moves away from siloed interpretation and implementation toward an **industry-led machine-executable, human-readable design**. This creates a single, authoritative representation of what each rule means and how it should be applied, so there's no ambiguity in the reporting process.

Consistent and transparent

Through industry collaboration, DRR reduces interpretation risk, eliminates duplicated implementation effort and provides a shared code base that can be reused across jurisdictions. Reporting rules are interpreted by the industry and implemented in a common, reusable way, limiting the need for siloed implementation projects at individual firms and across different jurisdictions. DRR also accelerates onboarding for new regulations since teams can build on common and reusable components.

For regulators and the wider market, DRR delivers a standardised implementation of reporting logic. When every participant applies the same logic, supervisory analysis becomes more reliable and systemic risks become easier to detect. The transparency of the model – every rule visible, versioned and reviewable – supports stronger governance and clearer communication between industry and regulators.

In short, DRR replaces fragmented interpretations with a **shared, executable source of truth** that raises the quality and efficiency of regulatory reporting across the entire ecosystem.

Benefits of using DRR

- **Consistency:** By following established design principles and a unified modelling guide, all users ensure their **reporting** data meets the same regulatory standards and that they're 100% aligned with best practice.
- **Transparency:** The scope and status of reporting requirements are clearly defined, reducing ambiguity for implementers. All data is traceable and all rules and reportable fields annotated with the relevant provisions, regulatory references, and peer review decisions, so that firms, auditors and regulators can understand exactly **why** a value appears in a report.
- **Efficiency:** The workflow automates the path from raw data to final regulatory projection, minimising manual errors. Because core logic is shared across jurisdictions where possible, DRR avoids duplication and applies jurisdiction-specific logic only where it's required. New regulations can leverage existing logic, meaning implementation is faster and at the same standard as any existing regulations.
- **Compliance:** Each regulation is reviewed and interpreted by industry-led working groups. This interpretation is then written into DRR, with each field annotated with regulatory references, working group decisions and ISDA best practices. Compliance has never been easier, with each rule being now being traceable to the field in the regulation and any ambiguity traceable to the working group decision or the ISDA best-practice publication. DRR is written in Rune, a human-readable coding language, meaning stakeholders from across the spectrum (compliance, executive leadership, regulators) can understand exactly how a reporting output is derived.
- **Interoperability:** The CDM ensures data is structured the same way across firms so everyone is using the same representation of the financial data.
- **Reduced cost:** Integrating open-access solutions can save significantly on IT expenditure. DRR 7 currently supports nine jurisdictions, and the burden of updating or adding jurisdictions is significantly reduced, with the cost and implementation effort spread across the firms involved.

How it works

DRR brings together stakeholders across the industry to intepret and then implement regulatory text.

Learn more in How DRR works.

Scope and jurisdictions

DRR covers a range of trade and transaction reporting regimes, including:

North America

- CFTC — Commodity Futures Trading Commission (United States)
- SEC — Securities and Exchange Commission (United States)
- CSA — Canadian Securities Administrators (Canada)

Europe and United Kingdom

- ESMA — European Securities and Markets Authority (European Union)
- FCA — Financial Conduct Authority (United Kingdom)

Asia-Pacific

- ASIC — Australian Securities and Investments Commission (Australia)
- HKMA — Hong Kong Monetary Authority (Hong Kong)
- JFSA — Japan Financial Services Agency (Japan)
- MAS — Monetary Authority of Singapore (Singapore)

How DRR works

Digital Regulatory Reporting (DRR) works by turning complex regulatory requirements into structured, machine-readable logic that can be reused, tested, and executed consistently across jurisdictions. Instead of firms implementing their own interpretation of regulatory requirements, DRR presents an industry-led interpretation and understanding in a shared model that defines what needs to be reported and how the rules should be applied.

DRR workflow

The DRR workflow aims to translate regulatory text into machine-executable logic to produce standardised reporting outputs across jurisdictions.

1: Industry-led interpretation

The DRR project brings together different firms, large and small, to convert the often ambiguous and inconsistent language of regulatory text into logical code that is traceable and human-readable.

Working groups meet to discuss the reporting requirements for each regime, determining the required output for each field. Any ambiguous or unclear requirements are determined by group consensus, and if there is still an impasse then the group can go to the regulators directly.

Regulatory text may include:

- CFTC Part 43/45
- EMIR RTS/ITS
- ASIC DTR
- MAS reporting rules
- JFSA OTC reporting
- MiFID/MiFIR

2: Collaborative implementation effort

Following working group consensus on how fields should be reported, the interpretation is translated into DRR rules to build reports for the necessary jurisdictions. Because DRR is open access, the implementation effort can be spread across different industry

participants meaning the workload and barrier to entry is significantly reduced for those seeking to adopt it.

Rune DSL (the domain-specific language designed for DRR) and DRR itself, are designed to be readable by professionals with domain knowledge (e.g. regulatory analysts), regardless of their programming competence or experience. This means that reporting logic can be written by any team member who understands the requirements and the regulation.

Working groups continue to meet regularly to discuss new or proposed changes to the model.

3: Scalable report design

DRR is designed to reduce duplication where possible while reusing components and logic across jurisdictions. Critical Data Elements set by the Committee on Payments and Market Infrastructures (CPMI) and the International Organization of Securities Commissions (IOSCO), and common reporting fields act as the foundation for every regime, each leveraging the core components on a transcation report and further extending it to support their own reporting requirements. Logic only needs to be written once and it can be reused multiple times, making implementation of new jurisdictions faster.

```
type CommonTransactionReport extends cde.CriticalDataElement:
<"Common Transaction Report represents common attributes">
```

4: Built-in testing and validation

Each report output in DRR is evaluated against a set of mandated data validation rules. These conditions and validations verify the data is correct syntactically, logically, and in accordance with allowable values set by the regulators.

This example shows a commonised validation rule that checks that `counterparty1` must not be the same as `counterparty2`.

```
    condition DTCC_ASIC_BR_1006_01: <"Counterparty 1">
        [docReference ASIC DTCC_Specs table "1" dataElement "6"
validationRule "Transaction"
```

```
        provision "when [Action Type] = NEWT, MODI, CORR, REVI, TERM, PRTO, ERROR, MARU, VALU then [Counterparty 1] must NOT = [Counterparty 2] if [Counterparty 2 identifier type] is = LEI"]
        common.party.Counterparty_Validation(
                actionType,
                counterparty2IdentifierType,
                counterparty1,
                counterparty2
            )
```

Synthetic data samples for each step’s inputs and output are captured in a 'Test Pack' that allows users to validate any implementation against those tests. Users can also test against their own proprietary data with their own anonymised samples.

DRR and the CDM

DRR and the CDM

The Common Domain Model (CDM) is the foundation of DRR. It provides a shared, unambiguous way to describe financial products, events and processes so that every firm, vendor and regulator can interpret data and rules in the same way. You can read the **full documentation for the CDM** but here are a few highlights.

What is the CDM

The CDM is a standardised data model for representing:

- Financial products
- Trades and trade states
- Lifecycle events
- Processes and workflows
- Legal agreements

The CDM defines these concepts in a machine-readable and machine-executable format. Instead of each firm codifying the data for regulatory rules differently, the CDM ensures that everyone uses the same underlying structures and definitions.

How DRR uses the CDM

DRR uses the CDM as its **single source of truth** for:

- How trades are represented.
- How lifecycle events are applied.
- How derived fields are calculated.
- How reporting logic is executed.

This gives DRR three essential qualities:

- **Consistency:** The same input produces the same output across firms.
- **Transparency:** Every field and rule is explicitly defined.
- **Interoperability:** CDM based data can flow across systems without translation.

Without the CDM, DRR would be just another set of bespoke rules. With the CDM, DRR becomes a **shared industry standard**.

Core components of the CDM

The CDM is organised into several key building blocks:

1. Product

Defines the economic terms of a trade, including payouts, notionals, prices and product identifiers.

2. Trade and TradeState

Represents the trade itself and its current state at any point in time.

3. Event

Describes lifecycle events such as new trades, amendments, terminations, compressions and allocations. Events are applied to TradeStates to produce updated states.

4. Party and roles

Identifies the parties involved in a transaction and their roles (e.g. counterparty, reporting party).

5. Processes

Standardised workflows that describe how events are applied and how states evolve.

Using DRR

More technical analysis of how DRR is structured, how to implement it and how to work with the model effectively.

Scope and structure

How jurisdictions, rulebooks and reporting regimes are represented in the model.

DRR namespace structure

An explanation of bodies, corpuses, rules, functions and projections.

Core modelling components of DRR

Understand the three crucial elements in DRR models.

DRR design principles

The modelling standards and conventions that ensure DRR is consistent and maintainable.

Implement DRR

How to bring DRR to your organization.

FpML ingest

How DRR can ingest data directly from FpML.

DRR namespace structure

DRR namespace structure

Digital Regulatory Reporting is layered and modular by design. The DRR model follows the CDM's organising principles into namespaces.

- **Base:** The *Base* namespace leverages elements defined in the CDM. It contains the core, reusable building blocks that underpin the entire DRR model.

- **Enrichment:** The *Enrichment* namespace defines types, functions and rules that enrich DRR objects with the information required for regulatory reporting. Within this namespace, a trade can be enriched from a `ReportableEvent` into a `TransactionReportInstruction`, for example, adding elements such as `ReportableInfomation` and `ReportingSide`.

- **Ingest:** The *Ingest* namespace contains the functional mappings for FpML Record Keeping trade messages. This namespace details the elements required to ingest a `FpML NonPublicExecutionReport` message and transform it directly into a `ReportableEvent`.

- **Standards:** The *Standards* namespace defines and codifies the Critical Data Elements or CDEs of regulatory reporting. Critical Data Elements are standardised, well-defined data fields that are used consistently across regulatory reporting frameworks. They ensure that different firms and regulators interoperate and report the same information the same way. CDEs are the building blocks for regulatory reports.

- **Regulation:** The *Regulation* namespace of DRR contains the reporting-specific logic, where regulatory text is digitised and translated into code. Reporting rules and their interperations include references to the relevant regulatory text and where applicable, the outcomes of the working group discussions. Within this namespace, sub-namespaces are organised by regulation name, along with a *Common* namespace. This houses shared rules and regulatory logic that apply across multiple regulatory regimes to ensure consistency and reuse throughout the DRR framework.

- **Projection:** The *Projection* namespace contains elements necessary to transform DRR report output into a report file formatted for submission to trade repositories

and regulators. This layer of the DRR structure ensures the report is structured, compliant and ready for regulatory consumption.

Workflow summary

This overview of the DRR structure explains how the workflow is organised:

- **Projections** build on **Regulations**
- Which in turn draw on **Common** rules
- Which call on **CDEs**
- Which leverage the DRR **Base** namespace.

This hierarchical approach ensures that DRR is consistent, traceable and reusable across regulatory regimes.

DRR design principles

The DRR model is built as an extension of the Common Domain Model (CDM). DRR uses the CDM to represent trades and lifecycle events as a required input for the DRR process. In practice, this means that DRR adheres to the same design principles as the CDM, while adapting them to meet the specific requirements of regulatory reporting. In this context, these principles can be summarised into four key design principles:

- Functional
- Composable and reusable
- Auditable
- Test-driven

Functional

The DRR model contains a complete set of logical instructions to take a CDM input and produce a regulator-ready output. There is no hidden or behind-the-scene logic that an implementor would need, in addition to the one expressed in the model, to build that reporting system. If that implementation uses the model's auto-generated executable code, no further code is required to express the reporting logic - although some coding is necessary to integrate what the DRR model provides.

This feature is key to ensuring the consistency and comparability of reported data across the market, i.e. the same inputs would produce the same reporting output regardless of the particularities of each firm's implementation.

Composable and reusable

The DRR model is built on defining reusable components, which helps prevent repeating logic and duplication of effort.

The DRR function `TradeForEvent` demonstrates this approach in practice. As reporting logic starts from a transaction event, many reportable fields require access to the associated details of a trade. Rather than embedding trade retrieval logic inside every reporting rule, **DRR defines this logic once** in `TradeForEvent`, and resues it where required.

```
reporting rule NotionalQuantityLeg1 from
TransactionReportInstruction: <"Notional Quantity Leg 1">
    filter IsAllowableActionForCFTC
    then if IsEquity(ProductForTrade(TradeForEvent)) = False
        then common.quantity.NotionalQuantityLeg1
```

Auditable

Every reporting rule defined in the DRR model can be directly linked back to the regulatory text that it comes from. The model allows for precise references to documents and provisions contained within those documements. This is implemented in DRR using the **document reference** feature of the Rune DSL.

```
reporting rule Cleared from TransactionReportInstructionBase:
<"Cleared">
    [regulatoryReference CPMI_IOSCO cdeV3.CDE section "2" field
"14"
        provision "Indicator of whether the transaction has been
cleared, or is intended to be cleared, by a central counterparty."]
```

Test-driven

DRR uses a test-driven approach to develop the reporting model. This means that the model is being systematically tested using transaction data inputs, and the reported outputs are validated against an expected result.

The transaction data inputs are synthetic data - i.e. not actual production data but data that are representative of real-life transaction scenarios and that can be used to test the validity of the reporting logic. They are typically provided by firms participating in DRR, after those firms apply suitable anonymisation and data scrambling to preserve privacy. A reporting rule is considered fully developed only once its logic has been verified against relevant test data.

Test data is organised around themes and grouped into *Test Packs* in the DRR model repository - for instance, by asset class. Each test pack contains both the transaction data inputs and their expected output. Test packs are an integral part of the model and are readily available to allow firms to benchmark their own implementations.

This approach supports the ongoing governance of the DRR model. Any mismatch in the pre-loaded test packs would generate expectation differences which need to either be resolved, or explained if the change is justified.

Core modelling components of DRR

A DRR model is structured around three main modelling components:

- **Types** – define how data is represented in the model.
- **Reporting rules** – define regulatory logic attached to those data structures.
- **Functions** – define reusable logic used by rules and calculations.

Together, these components define how regulatory reports are produced from the CDM.

Types

A **type** defines the structure of a data object. Types represent domain concepts used in DRR, such as `PartyInformation`. A type describes the fields that belong to the object and the relationships between them.

This example shows the complex type `PartyInformation`.

```
type PartyInformation: <"Specifies party specific information
required for reporting of the transaction.">
    partyReference Party (1..1) <"Specifies the party that is
associated with the enriched information.">
        [metadata reference]
    relatedPerson NaturalPersonRole (0..*) <"The role(s) that
natural person(s) may have in relation to the transaction.">
    relatedParty RelatedParty (0..*) <"Specifies one or more
parties that perform a role within the transaction. The related
party performs the role with respect to the party identified by the
'partyReference'.">
```

In this example:

- `PartyInformation` is the type.
- The type contains the attributes `partyReference`, `relatedPerson`, and `relatedParty`.
- Each field specifies a data type and a cardinality.

Types form the foundation of the DRR model. Reporting rules and functions operate on the data defined within these structures.

Reporting rules

Reporting rules define regulatory logic used to populate individual reporting fields.

A reporting rule is attached to a specific input type. The rule evaluates functional logic that extracts or derives a value from that type.

This example shows the reporting rule `Counterparty2`.

```
reporting rule Counterparty2 from TransactionReportInstruction:
<"Counterparty 2">
    if cde.party.Counterparty2IdentifierTypeIndicator = True
        then cde.party.Counterparty2
    else reportingSide -> reportingCounterparty -> partyId ->
identifier first
```

In this example:

- `Counterparty2` is the rule name.
- `TransactionReportInstruction` is the input type.
- The rule evaluates logic to determine which value should be returned by calling reporting rules `cde.party.Counterparty2IdentifierTypeIndicator` and `cde.party.Counterparty2` or extracting the identifier directly from `reportingSide -> reportingCounterparty -> partyId`.

Functions

Functions define reusable logic that can be used across the model.

A function accepts one or more inputs, performs a set of operations, and produces an output. Functions are typically used to encapsulate logic that may be required by multiple reporting rules.

The following example shows the function `CounterpartyRoleFromLEI`.

```
func CounterpartyRoleFromLEI:
    inputs:
        counterparties Counterparty (2..2)
        partyLei LEIIdentifier (0..1)
```

```
    output:
        reportingParty CounterpartyRoleEnum (0..1)
    alias party1: ExtractCounterpartyByRole(counterparties, Party1)
    alias party2: ExtractCounterpartyByRole(counterparties, Party2)
    set reportingParty:
        if partyLei = PartyLei(party1 -> partyReference -> partyId)
        then Party1
        else if partyLei = PartyLei(party2 -> partyReference ->
partyId)
        then Party2
```

In this example:

- `CounterpartyRoleFromLEI` is the function name.
- `Counterparty` and `LEIIdentifier` are the input types.
- `CounterpartyRoleEnum` is the output type.
- `party1` and `party2` are temporary variables defined using `alias`.
- The functional logic that takes the inputs and returns the output variable `reportingParty` is defined after the keyword `set`.

Modelling components using Rune DSL

The Rune DSL provides additional modelling components that support more advanced modelling patterns, such as enumerations, conditions, metadata, mappings and other reusable constructs. You can find out more about these modelling and their syntax in Rune DSL modelling components.

Scope and structure

Reporting regimes

DRR currently supports these trade and transaction reporting regimes:

North America

- CFTC — Commodity Futures Trading Commission (United States)
- SEC — Securities and Exchange Commission (United States)
- CSA — Canadian Securities Administrators (Canada)

Europe and United Kingdom

- ESMA — European Securities and Markets Authority (European Union)
- FCA — Financial Conduct Authority (United Kingdom)

Asia-Pacific

- ASIC — Australian Securities and Investments Commission (Australia)
- HKMA — Hong Kong Monetary Authority (Hong Kong)
- JFSA — Japan Financial Services Agency (Japan)
- MAS — Monetary Authority of Singapore (Singapore)

For each regulation, the model defines the set of regultory references together with their issuing authority. They are represented as:

- `Body` – the entity that is the author, publisher or owner of the referenced document
- `Corpus` – the document set that contains the referenced information

```
body Authority CFTC <"Commodity Futures Trading Commission (CFTC):
The Federal regulatory agency established by the Commodity Futures
Trading Act of 1974 to administer the Commodity Exchange Act.">
corpus Specifications "DTCC Specs" DTCC_Specs <"Document providing
the message specifications required for inbound message for DTCC
for CFTC.">
corpus Regulation "CFTC 17 CFR Parts 45 Version 3.0" Part45 <"Part
45 of the CFTCs regulations specifies the Commissions swap data
recordkeeping and reporting requirements, pursuant to section 2(a)
(13)(G) of the Commodity Exchange Act (CEA), which states that all
```

```
swaps, whether cleared or uncleared, must be reported to a Swap
Data Repository (SDR)">
```

The same structure is used for **standard-setting bodies**, such as CPMI–IOSCO's Critical Data Elements (CDE):

```
body Authority CPMI_IOSCO <"IOSCO and the Committee on Payments and
Market Infrastructures (CPMI) work together to enhance coordination
of standard and policy development and implementation, regarding
clearing, settlement and reporting arrangements including financial
market infrastructures (FMIs) worldwide...">
corpus TechnicalGuidance "Harmonisation of Critical Data Elements
(other than UTI and UPI)" CDE <"The G20 Leaders agreed in 2009 that
all over-the-counter (OTC) derivative transactions should be
reported to trade repositories (TRs) to further the goals of
improving transparency, mitigating systemic risk and preventing
market abuse...">
```

Namespace

The DRR model adopts the **CDM's namespace structure**. DRR reporting logic sits in a dedicated **regulation layer**, using the below convention:

```
drr.regulation.<body>.<regulation>
```

Reporting regulations can themselves be subdivided, for instance when regulatory specifications are revised:

```
drr.regulation.cftc.rewrite
```

Reporting regulations are then further subdivided by report type:

```
drr.regulation.cftc.rewrite.trade
drr.regulation.cftc.rewrite.margin
drr.regulation.cfrc.rewrite.valuation
```

By design, the model is composable. Therefore, there are some components which are not specific to any regulation and are built to be resuable across several of them. These components can be found in the `drr.regulation.common` namespace.

Elements defined in other namespaces can also be reused throughout the model by **importing** that namespace. For example, CDE reporting rules defined in `drr.standards.iosco.cde` are leveraged and reused within the CFTC regulation namespace:

```
namespace drr.regulation.cftc.rewrite
import drr.standards.iosco.cde.*
```

This keeps the model modular, reusable, and consistent across jurisdictions. Read more about **DRR namespace structure**.

Reportable event

All reporting regimes assume that reporting starts from a **transaction event**. In DRR, this is represented by the `ReportableEvent` data type. The `ReportableEvent` type **extends** `ReportableEventBase`. This means `ReportableEvent` (known as a *sub-type*) inherits all of its behaviour and attributes from `ReportableEventBase` (known as *super-type*) and adds its own behaviour and attributes on top.

```
type ReportableEvent extends ReportableEventBase: <"Specifies a
workflowstep with enriched information required for reporting.">
    [rootType]
    override reportableInformation ReportableInformation (1..1)
<"Additional information required for a reportable transaction,
including the reporting regime.">
type ReportableEventBase:
    originatingWorkflowStep WorkflowStep (1..1) <"The workflowstep
that originated the reportable event.">
    reportableTrade TradeState (0..1) <"The reportable trade
decomposed from the originating workflow step when required.">
    reportablePosition CounterpartyPositionState (0..1) <"The
reportable position decomposed from the originating workflow step
when required.">
    reportableInformation ReportableInformationBase (0..1)
```

```
<"Additional information required for a reportable transaction,
including the reporting regime.">
```

In the CDM, the `WorkflowStep` data type, an attribute of `ReportableEvent`, is used to represent the state of a business event. While a `WorkflowStep` provides some reportable information, a transaction will typically need to be enriched with further information that is only relevant in a reporting context.

The dedicated `ReportableEvent` data type in the DRR model allows us to capture this additional, enriched information, ready for DRR consumption.

Report definition

Each regulatory report is defined using three components:

- *What* to report, i.e. the reportable fields
- *Whether* to report – eligibility criteria.
- *When* to report – timing (this is informational only, not executable).

```
report CFTC Part43 in T+1
    from TransactionReportInstruction
    when IsReportableEvent
        with type CFTCPart43TransactionReport
```

The reportable fields are defined by specifying:

- A `body` and `corpus` as the source of the reporting mandate - `CFTC Part43`
- A timing constraint as a syntactic indication. Note this does not generate any executable code and therefore has no impact on the reporting process – `T+1`
- Eligibity criteria which references a functional rule that returns a Boolean – `IsReportableEvent`
- The report's output as a data type whose attributes represent the required reportable fields – `CFTCPart43TransactionReport`

In the report's output definition, each attribute (or reportable field) may be linked to a reporting rule through a `ruleReference`. A `ruleReference` contains the logic used to extract or compute the value to be reported for the associated attribute.

```
type CFTCPart43TransactionReport:
  [rootType]
  cleared string (1..1)
    [ruleReference Cleared]
  counterparty1 string (1..1)
    [ruleReference Counterparty1]
```

A report's output is validated using a `condition`. In the model, conditions are used to define the validation rules for reportable fields. Each `condition` evaluates to a Boolean value, and if it evaluates to False, the validation fails.

```
condition ReportingTimestampCondition:
[regulatoryReference CFTC Part43 appendix "1" dataElement "97"
validationRule "Transaction"
    provision "M, the value shall be equal to or later than the
value in [Execution timestamp]"]
        reportingTimestamp = executionTimestamp or
reportingTimestamp > executionTimestamp
```

Implement DRR

Reporting firms can implement DRR using three approaches that can be combined: **Build, Benchmark, Buy.**

Build

A firm uses the open-acess DRR model to develop its own internal implementation. This involves these implementation steps:

Execution enviroment

The firm develops or deploys a **run-time execution engine** capable of executing the DRR reporting rules. This engine runs the DRR logic against the firm's trade data within the firm's own infrastructure.

Software integration

The DRR code artefacts are integrated into the firm's software development lifecycle, allowing them to be versioned, tested, and deployed alongside the firm's internal systems.

Data translation

The firm implements the necessary data transformation logic to translate its internal data formats into the CDM objects required as input to the DRR process.

Testing and validation

The firm uses the DRR Test Packs for quality assurance. Input data is processed through the firm's implementation and the resulting output is compared with the expected outputs provided in the test pack to verify that the implementation behaves correctly.

The Build approach provides firms with maximum control over deployment and integration, while leveraging the shared DRR model and regulatory logic maintained by the community.

Benchmark

A firm uses the testing capabilities that are freely available under the Community Edition of the Rosetta platform to validate its own DRR reporting implementation.

This allows firms to compare the outputs of their internal implementation with the outputs generated by the reference DRR model.

Buy

A firm buys a reporting solution from a third-party vendor where the vendor itself may have followed the Build approach to develop their commercial product.

The following sections expand on the Build approach. They detail the Ingest, Enrich, Report and Project steps of the DRR process and how reporting firms can use them to develop their implementation.

The DRR Pipeline

The DRR pipeline steps **Ingest, Enrich, Report and Project** are necessary for reporting firms to develop their implementation using any of the methods above.

1. Ingest

Why?

To transform a firm’s internal message format into a CDM object, which is the required input for the DRR process.

How?

Option A - FpML Ingestion

DRR includes built-in capabilities to ingest an FpML record-keeping message using functional mappings. These mappings are available in the `drr.ingest.fpml.recordkeeping` namespace. They provide the standard fucntions to translate an FpML record-keeping structure into a DRR object - `ReportableEvent`.

Option B - Manual Ingestion

Alternatively, a firm can manually define it's internal message structure into equivalent Rune data types. These data types can then be used in mapping functions to transform a firm's internal data representation into the corresponding CDM objects required for DRR processing.

The below ingest function converts the FpML message into a **CDM workflow representation**.

```
func Ingest_FpmlConfirmationToWorkflowStep:
    [ingest XML]
    inputs:
        fpmlDocument fpml.Document (0..1)
    output:
        workflowStep WorkflowStep (0..1)
    set workflowStep:
        fpmlDocument switch
            fpml.ClearingConfirmed then
MapClearingConfirmedToWorkflowStep,
            fpml.ExecutionAdvice then
MapExecutionAdviceToWorkflowStep,
            fpml.ExecutionAdviceRetracted then
MapExecutionAdviceRetractedToWorkflowStep,
            fpml.ExecutionNotification then
MapExecutionNotificationToWorkflowStep,
            fpml.RequestClearing then
MapRequestClearingToWorkflowStep,
            fpml.TradeChangeAdvice then
MapTradeChangeAdviceToWorkflowStep,
            default empty
```

2. Enrich

Why?

To enrich a CDM object with additional information required for regulatory reporting. This can be obtained from internal or external reference data. This step is necessary because transaction data orginating from front-office systems do not usually include the static data required for reporting.

How?

Enrichment functions extract relevant attributes from the input object and call external API(s) based on those attributes to retrieve additional information from external sources.

- The DRR Java code containing the enrichment and API call functions can be added as a code dependency (e.g. using Maven and Gradle) or downloaded from the

Rosetta application.

- The enrichment and external API functions are distributed in the DRR Java code as interfaces, which implementors are meant to develop according to their own business requirements and implementation choices.
- The Rosetta platform provides some built-in implementation of the enrichment and external API functions, allowing users to automatically enrich data when developing and testing their regulatory logic. Examples of reference data for which a built-in API call is provided include:
  - Legal Entity Identifier (from GLEIF)
  - Market Identifier Code (from ISO)

3. Report

Why?

To generate a report output which is a DRR object in JSON format based on a CDM object input.

Once enrichment has been completed within the DRR process, the **regulatory reporting logic** is applied. At this stage, minimal additional implementation is required from a DRR firm. The model itself defines the regulatory rules and reporting logic for each supported regime.

How?

The report and rule definitions are avaliable in the `drr.regulation.*` namepsaces and prefixed accordingly e.g. `drr.regulation.cftc.rewrite.trade`.

The DRR Java code containing the reporting rules can either be downloaded from **Rosetta** or added as a code dependency (e.g. using Maven or Gradle).

Example using Maven

This dependency gives access to the generated Java class representing a specific report object e.g. `CFTCPart45TransactionReport`

```
<dependency>
  <groupId>com.regnosys.drr</groupId>
    <artifactId>rosetta-source</artifactId>
```

```
    <version>LATEST</version>
</dependency>
```

The DRR artefacts can be found in the DRR repository: `https://europe-west1-maven.pkg.dev/production-208613/isda-maven`. Add the following repository block to your pom file or settings file:

```
<repository>
    <id>digital-regulatory-reporting</id>
    <url>https://github.com/rosetta-models/digital-regulatory-reporting</url>
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</repository>
```

- To execute a report for a particular DRR version:
  - **Namespace**: the namespace for the report, e.g. `drr.regulation.cftc.rewrite`
  - **Body**: the body of the report that this class will generate, e.g. `CFTC`
  - **Corpus** list: a list of corpus for the report that this class generates, e.g. `Part45`
- Create the Guice module used to initialise the report functions. The convention for DRR is `DrrRuntimeModuleExternalApi.class`
- Create the report function for the report
  - The class will have been automatically generated by the DSL

```
CFTCPart45ReportFunction function =
injector.getInstance(CFTCPart45ReportFunction.class)
```

- Create an input CDM object representing the transaction (always a RosettaModelObject sub-type, e.g. ReportableEvent) by either:

- Converting JSON to Java by using a Jackson Object Mapper (the RosettaObjectMapper utility helps set things up),
  - Creating a `ReportableEvent` by Ingesting Record Keeping FpML or other external models into the CDM,
  - Creating a `ReportableEvent` using Java code, or
  - Extracting a `ReportableEvent` from a CDM native implementation
- Run the report based on that input CDM object (`inputData`), to return a structured object, according to the report's specified data type.

```
CFTCPart45TransactionReport report =
function.evaluate(reportableEvent);
```

A full example using the "CFTC Part 45" reporting regime:

```
package com.regnosys.drr.examples;
import cdm.base.staticdata.party.CounterpartyRoleEnum;
import cdm.base.staticdata.party.metafields.ReferenceWithMetaParty;
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.regnosys.drr.DrrRuntimeModuleExternalApi;
import com.regnosys.drr.examples.util.ResourcesUtils;
import
com.regnosys.rosetta.common.serialisation.RosettaObjectMapper;
import drr.regulation.cftc.rewrite.CFTCPart45TransactionReport;
import drr.regulation.cftc.rewrite.reports.CFTCPart45ReportFunction;
import drr.regulation.common.ReportableEvent;
import drr.regulation.common.ReportingSide;
import drr.regulation.common.TransactionReportInstruction;
import
drr.regulation.common.functions.Create_TransactionReportInstruction;
import drr.regulation.common.functions.ExtractTradeCounterparty;
import java.io.IOException;
import java.util.List;
public class CFTCPart45ExampleReport {
  public static void main(String[] args) throws IOException {
      // 1. Deserialise a `ReportableEvent` JSON from the test pack
      ReportableEvent reportableEvent =
ResourcesUtils.getObjectAndResolveReferences(ReportableEvent.class,
"regulatory-reporting/input/events/New-Trade-01.json");
      // Run report
```

```
        CFTCPart45ExampleReport cftcPart45ExampleReport = new
CFTCPart45ExampleReport();
        cftcPart45ExampleReport.runReport(reportableEvent);
    }
    private final Injector injector;
    CFTCPart45ExampleReport() {
        this.injector = Guice.createInjector(new
DrrRuntimeModuleExternalApi());
    }
    void runReport(ReportableEvent reportableEvent) throws IOException
{
        // TransactionReportInstruction from ReportableEvent and
ReportingSide
        // For this example, arbitrarily PARTY_1 as the reporting
party and PARTY_2 as the reporting counterparty
        final ReportingSide reportingSide = ReportingSide.builder()
                .setReportingParty(getCounterparty(reportableEvent,
CounterpartyRoleEnum.PARTY_1))

.setReportingCounterparty(getCounterparty(reportableEvent,
CounterpartyRoleEnum.PARTY_2))
                .build();
        final Create_TransactionReportInstruction
createInstructionFunc =
injector.getInstance(Create_TransactionReportInstruction.class);
        final TransactionReportInstruction reportInstruction =
createInstructionFunc.evaluate(reportableEvent, reportingSide);
        // Run the API to produce a CFTCPart45TransactionReport
        final CFTCPart45ReportFunction reportFunc =
injector.getInstance(CFTCPart45ReportFunction.class);
        final CFTCPart45TransactionReport report =
reportFunc.evaluate(reportInstruction);
        // Print

System.out.println(RosettaObjectMapper.getNewRosettaObjectMapper()
                .writerWithDefaultPrettyPrinter()
                .writeValueAsString(report));
    private ReferenceWithMetaParty getCounterparty(ReportableEvent
reportableEvent, CounterpartyRoleEnum party) {
        ExtractTradeCounterparty func =
injector.getInstance(ExtractTradeCounterparty.class);
        return func.evaluate(reportableEvent,
party).getPartyReference();
```

```
  }
}
```

4. Project

Why?

To convert a DRR reports output (JSON) into the format required for Trade Repository submission (e.g. ISO 20022)

How?

In the previous step, we converted the DRR report object to JSON using this code:

```
RosettaObjectMapper.getNewRosettaObjectMapper()
    .writerWithDefaultPrettyPrinter()
    .writeValueAsString(report);
```

However, the ISO 20022 standard requires the output to be serialised as XML. As an example, this code can be used to serialise an `iso20022.auth108.esma.Document` object to XML.

```
URL xmlConfig = Resources.getResource("xml-config/auth108-esma-rosetta-xml-config.json");
RosettaObjectMapperCreator
    .forXML(xmlConfig.openStream())
    .create()
    .writerWithDefaultPrettyPrinter()
    .writeValueAsString(document);
```

The JSON file `auth108-esma-rosetta-xml-config.json` defines the necessary metadata to ensure that the output conforms exactly to the ISO auth.108.001.01.xsd XML schema file. It's available as a resource in this Maven artefact:

```
<dependency>
    <groupId>org.iso20022</groupId>
    <artifactId>rosetta-source</artifactId>
```

```
    <version>LATEST</version>
</dependency>
```

The ISO artefacts are in the ISDA repository in Maven: `https://europe-west1-maven.pkg.dev/production-208613/isda-maven`. This repository block can be added to a POM file or settings file:

```
<repository>
    <id>isda-maven</id>
    <url>https://europe-west1-maven.pkg.dev/production-208613/isda-maven</url>
    <releases>
        <enabled>true</enabled>
    </releases>
    <snapshots>
        <enabled>false</enabled>
    </snapshots>
</repository>
```

FpML ingest

DRR ingests data that has been transformed into the CDM. However, it can also directly ingest raw data from FpML, the shared language used by many banks, trading platforms and other financial institutions to describe complex trades.

FpML messages supported for ingestion to CDM and DRR:

- `FpML Confirmation To TradeState` – ingests a FpML messgae to produce a `TradeSate` (CDM object)
- `FpML Confirmation To WorkflowStep` – ingests a FpML message to produce a `WorkflowStep` (CDM object)
- `FpML RecordKeeping To ReportableEvent` – ingests a FpML message to produce a `ReportableEvent` (DRR object)

`TradeState`

A `TradeState` is a pure representation of the trade itself. It only contains the details of the trade, without any information about how the trade came into existence or changed. Think of `TradeState` as a snapshot of the trade at a specific point in time.

It contains:

- Economic terms (e.g. notional, rate, dates, counterparties)
- No business event context
- No regulatory information

`WorkflowStep`

A `workflowStep` contains information about the business event for that trade, including:

- Trade information or `tradeState`
- Business event information (what is happening to the trade)

Proposed event

At ingestion, the `WorkflowStep` details:

- The `ProposedEvent` (the proposed Buisness Event)

- The **before** `TradeState`
- The `PrimitiveInstruction` (the instruction to be applied to the **before** `TradeState`, a `QuantityChange` for example)

Example: Ingest output (proposed event)

- Before: trade notional = 1,000,000
- Instruction: `quantityChange` decrease of 500,000
- After: not yet calculated

Proposed event = before `TradeState` + instruction

Business event

When data has been ingested to a `WorkflowStep`, functions in the CDM use the `PrimitiveInstuction` and the **before** `TradeState` to generate an **after** `TradeState`. Therefore what was once a proposed event, is transformed to a business event. The resulting, fully realised business event is reflected in DRR reports input within the `WorkflowStep`.

Post ingestion, the `WorkflowStep` therefore details:

- The `BusinessEvent`
- The **before** `TradeState`
- The `PrimitiveInstruction`
- The **after** `TradeState`

Example: Report input (business event)

- Before: trade notional = 1,000,000
- Instruction: `quantityChange` decrease of 500,000
- After: trade notional = 500,000

Business event = before `TradeState` + instruction + after `TradeState`

`ReportableEvent`

Ingestion of an FpML `NonPublicExecutionReport` message generates a `ReportableEvent` - a DRR object. This represents the trade **as required for**

regulation. It contains:

- Trade information or `TradeState`
- Buisness event information
- Regulatory reporting information

Tutorials

Step by step guides, including the full DRR data workflow.

DRR data journey using Rosetta

A walkthrough of the DRR data flow using the Rosetta platform.

DRR data journey using an IDE

A walkthrough of the DRR data flow using an integrated development environment.

DRR data journey via an API

How to use DRR's API services for your regulatory data flow.

Create a new regime

How to add a new regulatory reporting regime to DRR.

DRR data journey using Rosetta

This page explains the full end-to-end workflow using the **Rosetta** platform for converting an **FpML Recordkeeping XML** into a:

1. **DRR ReportableEvent**
2. **TransactionReportInstruction**
3. **Regulatory transaction report**, and finally
4. **ISO 20022 XML projection**

1. Ingest

1.1. Create a **DRR workspace** in Rosetta.

1.2. Open the **Ingest** panel.

1.3. Upload your own FpML sample or select a preloaded test file (e.g. Equity Option Price Return Basket ex01 New).

Rosetta uses these functions to ingest different FpML message types to produce CDM or DRR object outputs:

- `FPML Confirmation To TradeState` – Ingests the message to create a CDM `TradeState` output. This represents only the trade itself, providing a snapshot at a specific point in time without any business event details or regulatory information.
- `FPML Confirmation To WorkflowStep` – Ingests the message to create a CDM `WorkflowStep` output. This adds context about the business event being applied (such as contract formation). The output includes both `TradeState` data and the associated `BusinessEvent details`.
- `FPML Confirmation To ReportableEvent` – Ingests the message to create a DRR `ReportableEvent` output. The output contains `TradeState` and `WorkflowStep` information, along with regulatory data related to the trade, such as the financial nature of the counterparty.

Of these three ingest functions, only `FPML Confirmation To ReportableEvent` produces a DRR object (`ReportableEvent`) as its output. This is because the input

message includes regulatory information that allows it to be ingested directly into a `ReportableEvent`.

1.4. When the `FPML 5 Confirmation To ReportableEvent` function is called, it directly produces a **DRR ReportableEvent** as an output. The diagnostics on the right of the screen reveal any issues in the testing.

2. Enrich

2.1. Open the **Enrich** panel.

2.2. Select the **Enrich** function from the dropdown.

2.3. Upload `ReportableEvent` from the previous step or select a preloaded test file from the table of samples. (e.g. Equity).

2.4. The Enrich service runs and produces a DRR `TransactionReportInstruction` enriched with additional required data from internal or external sources.

3. Reports

3.1. Open the **Reports** panel.

3.2. Select a report type (e.g. **ESMA / EMIR Trade**) and a test pack (e.g. Equity).

3.3. Upload the `TransactionReportInstruction` from the previous step (e.g. Equity_Option_Example) or select a preloaded test file from the table of samples.

3.4. The Report service runs and produces a single regime-specific `TransactionReport` with that regime's logic applied.

4. Projection

4.1. Open the **Projection** panel.

4.2. Select a projection (e.g. **EsmaEmirTradeReportToIso20022**) and a test pack (e.g. Equity).

4.3. Upload the `TransactionReport` JSON from the previous step (e.g. Equity_Option_Example_Upload) or select a preloaded test file from the table of samples.

4.4. The Projection service runs and produces the correctly formatted projection for the chosen regime (e.g. ISO 20022 XML, DTCC Harmonized XML).

DRR data journey using an IDE

The development environment may be different for each firm – DRR is not tied to a specific development environment. Below is a simplified, generic example of how to use DRR to produce a report using an IDE (Integrated Development Environment) such as Visual Studio Code, IntelliJ IDEA, Eclipse or PyCharm.

Whichever environment you use, the process is essentially the same, as DRR processes a trade event (which has been converted into the CDM), enriches it with regime-specific logic, and ultimately produces the correct regulatory report for a given jurisdiction (e.g. DTCC RDS Harmonized).

1. Start with the CDM event

Everything begins with a CDM trade event. This is the raw representation of what happened: a new trade, an amendment, a termination, etc.

```
type ReportableEvent { event: CdmEvent }
```

This stage captures the trade lifecycle event in a standard CDM format so DRR has a clean, consistent starting point.

2. Attach the regulatory context

Next, DRR takes the CDM event and wraps it in a structure that prepares it for a specific regime:

```
func Create_RegimeReportableEvent(event: ReportableEvent, regime:
Regime)
```

This stage prepares the event for a particular regulator (e.g. CFTC, ESMA, MAS) by applying the correct reporting rules.

3. Apply regime-specific logic

This is where DRR interprets the CDM event through the lens of the chosen regime.

```
type RegimeReportableEvent extends ReportableEvent { regimeContext:
Regime }
```

This stage applies the required regime specific rules, handles overrides and normalises the data according to that regulator's expectations.

4. Convert the event into a report instruction

Next, DRR transforms the regime interpreted event into a transaction report instruction.

```
func Create_TransactionReportInstructionForRegime(rre:
RegimeReportableEvent)
```

This stage turns the regulatory interpretation into a specific instruction that outlines the type of report to be produced and the data required to build it.

5. Per regime reporting logic

Each regulator has its own reporting structure – this data type captures the structure required.

```
type TransactionReportInstructionForRegime { reportFields:
RegimeSpecificFields }
```

This stage defines the exact fields, formats, timestamps, and logic required by the supervisory body (e.g. DTCC, ASIC, MAS). This is the point where the output diverges depending on the jurisdiction.

6. Final projection to the jurisdiction's report format

From the instruction, DRR produces the final regulatory report. For example, under the Canadian Securities Administrators (CSA) workflow, the instruction may be projected into one of two outputs:

Option A – CSA Trade Report

`CSA_Trade Project_CSATradeReportToDtccRdsHarmonized`

Option B – CSA PPD Report

`CSA_PPD Project_CSAPpdReportToDtccRdsHarmonized`

This stage converts DRR's internal representation into the regulator's required schema. It then applies final formatting and harmonization rules to produce the exact structure expected by DTCC RDS Harmonized (or another TR).

DRR data journey via API

This page explains the full end to end workflow using the DRR API services for converting an **FpML Recordkeeping XML** into:

1. a **CDM ReportableEvent**
2. a **TransactionReportInstruction**
3. a **regulatory transaction report**, and finally
4. an **ISO 20022 XML projection**

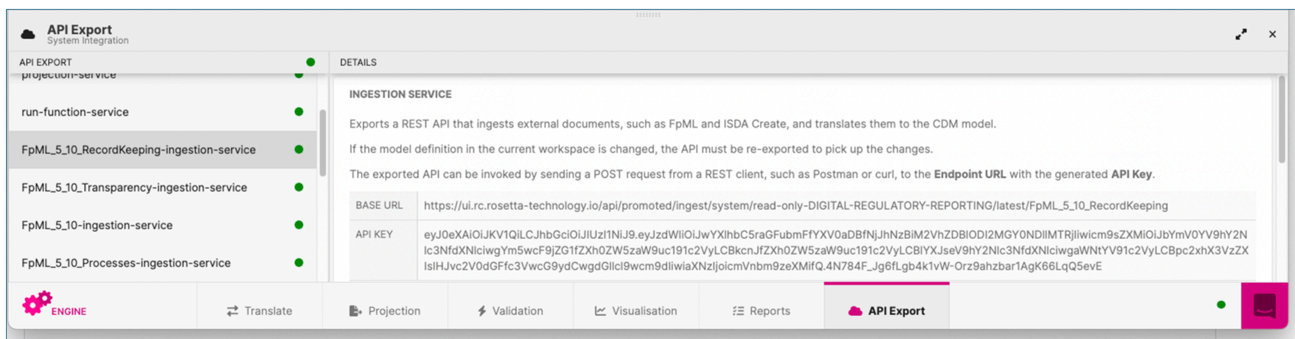
The following example uses Postman to send API requests, but virtually any API tool could be used in this way.

1. Running the ingestion service via (e.g.) Postman API tool

You can ingest an FpML example directly via API.

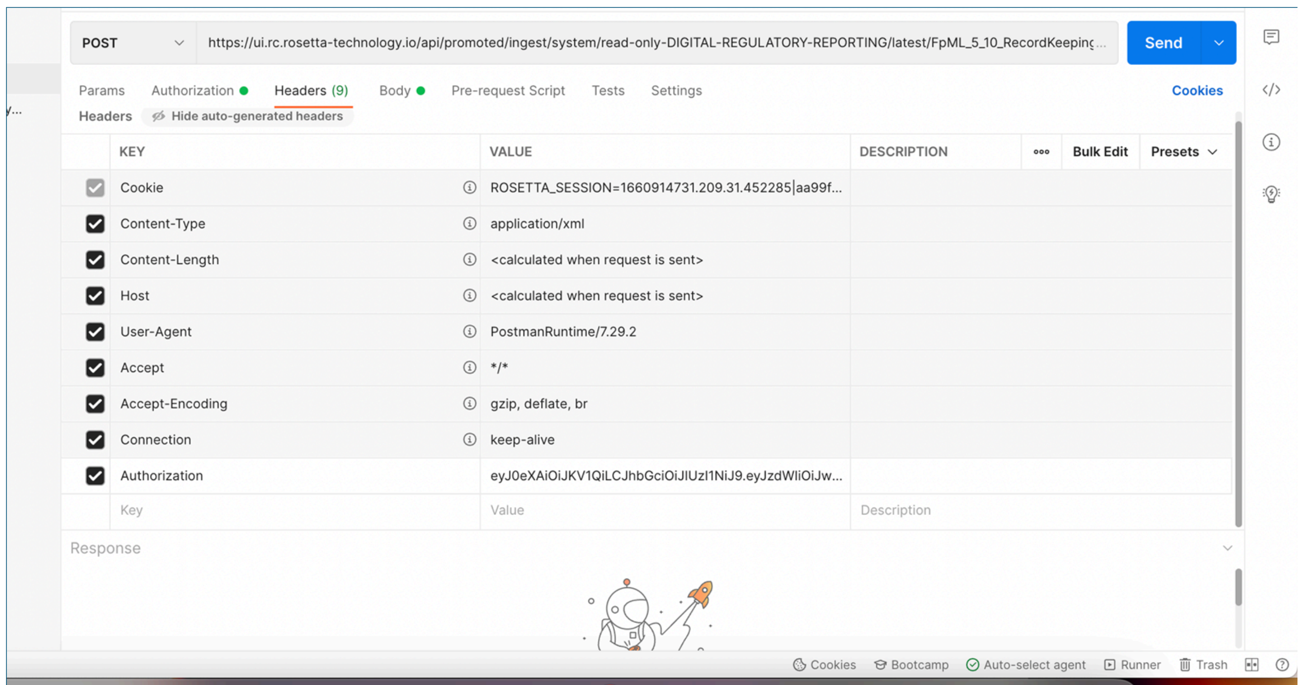
1.1 In Rosetta, open **DRR Model → API Export**.

1.2 Select `FpML_5_10_RecordKeeping ingestion service`.



1.3 Copy the **Base URL** and **API Key**.

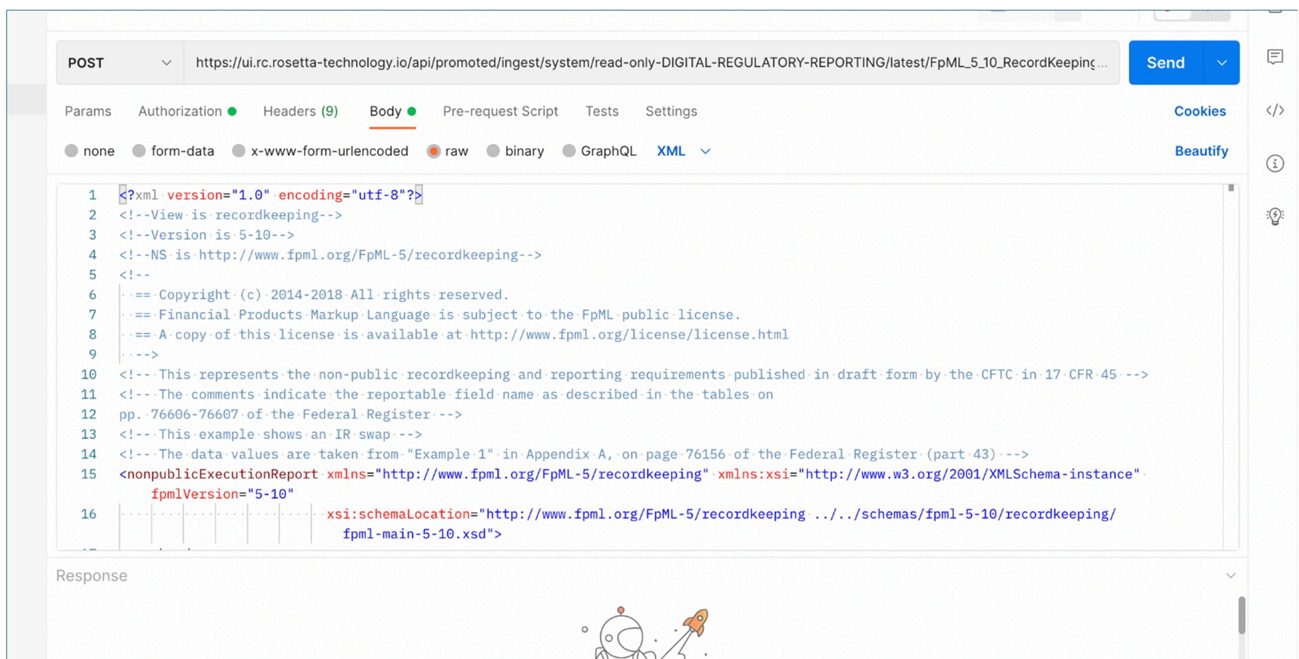
1.4. In Postman, create a **POST** request using the copied URL.



1.5. In **Headers**, set:

- Key: Authorization
- Value: API Key from Rosetta

1.6. In **Body**, choose **raw → XML**.



1.7. Paste your FpML XML and click **Send**.

1.8. Copy the `originatingWorkflowStep` and `reportableInformation` sections:

```
{
“originatingWorkflowStep”: { * }, “reportableInformation”: { * }
}
```

2. Running the custom function service

Use the ingestion output to generate a `ReportableEvent`.

2.1 In Rosetta → API Export, select `run-function-service`.

2.2. Copy the URL and create a new POST request in Postman. Then add:
`/drr.regulation.common.functions.Create_ReportableEventFromInstruction`

2.3. Add the **Authorization** header again.

2.4. In **Body**, choose **raw → JSON**.

2.5. Paste the JSON copied from the ingestion response and click **Send**.

Copy the resulting JSON for the next step.

3. Running the regulation report service

Use the `ReportableEvent` to generate the final regulatory report.

3.1. In Rosetta → API Export, select **regulation-report-service**.

3.2. Copy the URL and create a new **POST** request. Append the regulation path, e.g. `/CFTC/Part45`

3.3. Then add the **Authorization** header.

3.4. In **Body**, choose **raw → JSON**.

3.5. Paste the output from the custom function step and click **Send**. This returns the final **regulatory transaction report**:

Create a new regime

Adding a new regime to DRR, whether for your private model or as a contribution back to DRR can be done in a few simple steps.

This example walks through the creation of a new regime, demonstrating how to take advantage of the common structure and shared logic across existing regimes. It highlights the core benefits of DRR and shows how financial institutions can develop reporting solutions quicker, more reliably, and with greater confidence.

1. Defining your data

Every regime begins by anchoring the model to the relevant regulatory text. To establish this link, you define:

- **Body** – the organisation responsible for the regulation.
- **Corpus** – the documents that contain the rules, such as technical standards or reporting instructions.

This provides DRR with a clear reference point, identifying who authored the rules and which documents govern them. As a result, every reporting decision can be traced back to an authoritative regulatory source.

1.1 Imports and using the exisiting implementation

At the top of a namespace, the import statement allows you to reference and reuse types, functions, and rules defined elsewhere in DRR.

For example, the `CommonTransactionReport` type is one of the fundamental shared types. It contains attributes that are reused across multiple regimes and is defined in `drr.regulation.common.trade.type`.

This import statement makes this type available within your regime namespace:

```
import drr.regulation.common.trade.*
```

Imports effectively “bring in shared building blocks,” allowing a regime to leverage existing implementations rather than redefining common concepts. This avoids duplication, reduces errors, and ensures consistency across regimes.

2. Extending the exisiting implementation

To create a new regime, you define a regime-specific TransactionReport type that extends `CommonTransactionReport`

For example:

```
import drr.regulation.common.trade.* as common
type ACMETransactionReport extends common.CommonTransactionReport:
    [rootType]
```

`CommonTransactionReport` contains attributes that are shared across multiple regimes. It itself extends `CriticalDataElementV3`, which contains the IOSCO core OTC derivatives data elements used throughout DRR.

All DRR regimes follow this pattern. This hierarchy keeps DRR scalable and avoids duplication: when shared concepts change, they only need to be updated once, while each regime adds only its specific logic and requirements.

3. Overriding exisiting logic and adding new logic

Different regulators may apply different interpretations, labels, or business logic to the same reportable field. DRR handles this through the use of the **override** keyword.

```
import drr.regulation.common.trade.* as common
type ACMETransactionReport extends common.CommonTransactionReport:
    [rootType]
    override eventType EventTypeEnum (0..1)
      [label "1 Event type"]
      [regulatoryReference ACME Trade dataElement "1" field "Event
Type"]
      [ruleReference EventType]
```

In this example we've used `eventType` a reportable field shared across our new regime and multiple existing regimes. The override allows us to:

- Apply a regime-specific label
- Add a `regulatoryReference` to maintain traceability to the regulatory text
- Replace the inherited logic with a regime specific `ruleReference`

The **override** keyword can also be used to remove fields that are not applicable to a particular regime.

```
import drr.regulation.common.trade.* as common
type ACMETransactionReport extends common.CommonTransactionReport:
    [rootType]
    override eventType EventTypeEnum (0..1)
      [label "1 Event type"]
      [regulatoryReference ACME Trade dataElement "1" field "Event
Type"]
      [ruleReference EventType]
    override effectiveDate ISODate (0..1)
      [ruleReference empty]
```

Here we have choosen `effectiveDate` as a reportable field that is not required for this regime. Setting `[ruleReference empty]` explicitly removes all reporting logic for our regime.

When defining a new regime, it's common to introduce reportable fields that are unique to that regime. These fields are added directly to the regime specific type, after all required overrides have been defined. For example:

```
import drr.regulation.common.trade.* as common
type ACMETransactionReport extends common.CommonTransactionReport:
    [rootType]
    override eventType EventTypeEnum (0..1)
      [label "1 Event type"]
      [regulatoryReference ACME Trade dataElement "1" field "Event
Type"]
      [ruleReference EventType]
```

```
    override effectiveDate ISODate (0..1)
      [ruleReference empty]
    regimeSpecificField string (0..1)
```

Here, `regimeSpecificField` represents a data element required only for the ACME regime. Because it does not exist in the shared `CommonTransactionReport`, it's defined directly within the regime type.

4. Creating a DRR report

Once the `ACMETransactionReport` type has been defined, the next step is to declare the report that will use it. example:

```
report ACME Trade in T+1
        from TransactionReportInstruction
        when ReportableProduct
        with type ACMETransactionReport
```

The **report Logic** defines how DRR assembles and produces a report.

Here for the ACME regime, a T+1 report is generated using the ACMETransactionReport type.

The report becomes immediately usable, and by supplying sample or test data you can test the generated output.

5. Creating a Test Pack

To test the regime and view the report output, you need to add sample data:

1. Open the **Test Pack** dropdown.
2. Select **Add Test Pack**.
3. Add sample data to the new Test Pack.

A Test Pack is like a “sandbox” where you can try out the rules and see what the report would look like with real data. This ensures the regime behaves correctly before it’s used

in production and also helps non-technical users understand how the rules will play out with real data.

6. Reviewing the output

Once the Test Pack is populated, DRR generates the reported fields according to the logic defined for `regimeExample`.

You can now review and validate the output, ensuring the regime behaves as expected.

Reference guides

Quick-access reference materials to help you use DRR.

CDM data model reference guide

Detailed documentation of the CDM types, structures and semantics used throughout DRR.

CDE specs for all jurisdictions

The Common Data Elements (CDE) definitions and mappings for each regulatory regime.

Rune DSL documentation

Essential information on the domain-specific language developed specifically to enable DRR logic.

Rosetta platform

The bespoke development environment created to manage DRR logic and practices.

ISDA and DRR

ISDA launched the DRR initiative, and there's a wealth of information on their website.

Resources

Additional materials to support your understanding of DRR.

FAQs

Answers to common questions about DRR, from financial to technical.

Glossary

Definitions of key terms, acronyms and modelling concepts used across the documentation.

FAQs

General

What is DRR?

Digital Regulatory Reporting (DRR) is an industry initiative that converts regulatory reporting rules into machine-executable code, ensuring consistent and cost-effective implementation across firms. It provides a shared, open-access expression of reporting logic that firms can adopt directly.

Who created DRR?

DRR is led by **ISDA** and developed collaboratively across the financial industry. It's built using the **Common Domain Model (CDM)**, developed by FINOS, which provides the shared data and process model underpinning the reporting logic.

What problem does DRR solve?

Regulatory reporting is traditionally expensive, inconsistent and prone to interpretation differences. DRR standardises the rules so all firms implement the same logic, reducing cost and improving data quality.

How does DRR ensure consistent reporting across firms?

DRR expresses regulatory rules as **machine executable logic**, removing ambiguity in how firms interpret requirements. As more firms adopt the same CDM-based transformations, reporting outputs become far more consistent across the industry.

Does DRR support multiple jurisdictions?

Yes, very much so. DRR is designed to be **multi jurisdictional**. It includes rule models for major regimes such as CFTC, EMIR, ASIC, MAS and others. Each jurisdiction has its own transformation logic, but all share the same CDM foundation.

How often is DRR updated?

DRR is updated whenever regulatory rules change or clarifications are issued. Because it is an open, collaborative model, updates are published centrally so all firms can adopt the latest logic at the same time, reducing divergence.

Can I extend or customise DRR for my own use case?

Yes. DRR is open-access and can be extended to support firm-specific workflows or additional reporting requirements. However, core DRR logic should remain unchanged if you want to stay aligned with the industry standard interpretation.

Technical

How does DRR relate to the CDM?

DRR is built as an extension of the CDM, using CDM data structures to represent transaction inputs and CDM functions to express reporting logic. This alignment reduces duplication, improves interoperability, and ensures that reporting outputs are consistent across jurisdictions and firms.

What data do I need to run DRR?

You need transaction data structured according to the **Common Domain Model (CDM)**. DRR then applies enrichment, transformation, and projection steps to produce the required regulatory fields. The DRR documentation includes examples of the expected input payloads.

Can I test DRR logic before integrating it into my systems?

Yes. The DRR documentation includes examples for running transformations directly in the **Rosetta UI** or via **Postman**. This allows you to validate your data, inspect intermediate steps and confirm outputs before embedding DRR into production workflows.

Regulatory

How does DRR ensure that regulatory rules are interpreted consistently across firms?

DRR expresses regulatory requirements as machine executable logic built on the **Common Domain Model (CDM)**. Because the logic is deterministic, the same input always produces the same output, no matter who runs it. This removes firm specific interpretation and ensures that all participants apply the rules in a consistent, regulator aligned way.

Does DRR replace a firm's responsibility to understand the regulation?

No. DRR helps firms implement the rules accurately, but it does not replace the need to understand the underlying regulation. Firms remain responsible for compliance, governance and oversight. DRR simply provides a transparent, industry agreed interpretation that reduces ambiguity and implementation risk.

How does DRR handle differences between regulatory jurisdictions?

DRR uses a shared CDM foundation but applies jurisdiction specific logic in the Report and Project stages. This means common concepts (like events, products and parties) are reused across regimes, while each jurisdiction's unique reporting rules are implemented separately. The result is consistency where possible and specificity where required.

Can regulators rely on DRR outputs for supervisory analysis?

Yes. One of DRR's core goals is to improve the quality and comparability of regulatory data. Because DRR logic is transparent, version controlled and applied consistently across firms, regulators receive cleaner, more standardised data. This supports better supervision, reduces reconciliation issues and strengthens trust between industry and regulators.

How does DRR support auditability and regulatory assurance?

Every value produced by DRR can be traced back through the full reporting pipeline – Ingest, Enrich, Report and Project. An execution engine such as Rosetta exposes each intermediate step, making it easy to see exactly how a rule was applied and where a value came from. This end to end lineage provides strong evidence for audits, internal controls and regulatory reviews.

Troubleshooting

Why is my DRR run failing before it reaches the Report stage?

This usually happens when the CDM input is incomplete or doesn't match the expected structure. DRR validates the data during the Ingest and Enrich stages and any missing fields, incorrect types or invalid event structures will stop the run early. Check the

validation messages in your execution engine (e.g. Rosetta) – they'll point to the exact field or object that needs fixing.

My output doesn't match the example in the documentation. What should I check first?

Start by confirming that you're using the same DRR version as the example. DRR is versioned, and even small updates can change outputs. Next, compare your CDM input with the example input to ensure the same fields, values and event types are present. Differences in enrichment logic often come from differences in the underlying data.

Why am I seeing 'no mapping found' or 'unmapped attribute' errors?

These errors appear when DRR logic expects a value that isn't present in the CDM input or hasn't been enriched earlier in the pipeline. Check the relevant enrichment rules to confirm that the attribute is being populated. If the attribute is optional in the regulation but required by the logic, review the jurisdiction's mapping rules to see whether a default or fallback value should be applied.

The DRR output schema doesn't match what my reporting system expects. What should I do?

First, verify that you're using the correct jurisdiction and version of the DRR Project logic. Each jurisdiction has its own output schema, and older versions may differ from current regulatory requirements. If the schema is correct but your system still rejects it, compare the field names and types against the regulator's published schema to identify mismatches.

How do I troubleshoot unexpected values in the final reporting output?

Use your execution engine's step by step view to trace the value back through Project, Transform, Enrich and Translate. Each stage shows exactly how the value was derived. Look for:

- Incorrect or missing enrichments
- Conditional logic that didn't behave as expected
- Upstream CDM fields that were populated incorrectly

Once you identify the stage where the value changed unexpectedly, you can adjust the input or logic accordingly.

Governance

Learn about our Working Groups, roadmap and guidelines.

Working Group Governance

Details on the Working Group process and responsibilities

Development roadmap governance and process

Our Working Groups and their responsibilities.

Working Group Terms of Reference

The rationale behind our Working Groups approach.

DRR community roadmap

Our latest development roadmap for DRR.

Governance guidelines

How DRR governance is managed.

Development governance

Further detail on developer governance.

Maintenance and release

How contributions are reviewed and approved.

Versioning

How DRR release versions are annotated.

DRR development design principles

The principles we follow when developing the model.

Contribution guidelines

How to propose changes and submit pull requests.

Change control guidelines

How changes to the model are managed and controlled.

Editing the model

Instructions on how to edit the DRR model files.

Release governance

Information on upcoming work and how it's released

Release schedule

DRR releases past, present and future.

Major release scheduling guidelines

How we define major release updates.

Governance guidelines

A change proposal can be defined at a conceptual level in an issue before being defined at a logical level (i.e. in code). In each case, the proposal must be developed in line with the DRR development design principles.

Once approved on the Working Group, the proposal will be scheduled to be merged with the DRR's main code branch by a reviewer.

This document provides the governance policy for specifications and other documents developed using the Community Specification process in a repository (each a “Working Group”). Each Working Group must adhere to the requirements.

1. Roles

Each Working Group may include the following roles. Additional roles may be adopted and documented by the Working Group.

- 1.1. Participants. “Participants” are those that have made Contributions to the Working Group subject to the Community Specification License. Participants are automatically abiding by the IP policy of the standard by just participating in a meeting or by actively "enrolling" in the standard.

- 1.2. Discussion Groups. The Working Group may form one or more "Discussion Groups" to organize collaboration around a particular aspect of a specification. Discussion Groups are for discussion only - approval of all portions of a specification is subject to the consensus-based decision-making process.

- 1.3. Working Group Host. The Working Group will have a host that will run the call. They should follow the Working Group guidelines laid out below

2. Working Group meeting flow

The host of the DRR Working Group calls should follow the below process when going through the Kanban project board to allow efficient use of time:

1. Run through Agenda items which will be found on this week's discussion call (Host's role to set up this Discussions page)
2. Approved column:

i. Ask the group if anyone has any blockers on their items and who needs to action to unblock
ii. Check if a reviewer has been assigned & who is the reviewer
    a. Release Management Team to facilitate assignment of a reviewer

3. Follow-up column:
    i. Ask the group if anyone has any updates on their item
    ii. It will be the task of the issue owner to chase whoever wants to follow up. Provided another contributor/reviewer has given an input it should not be blocked
    iii. When an issue is approved, request on the call one of the reviewer does the review.
4. Current column:
    i. Go through each item, and move issues to either of the following:
        a. Approved - Request on the call one of the reviewer does the review, and leave a comment and assign an approver
        b. Follow-Up - Leave a comment with the action whoever wants to leave it in Follow-up.

3. Decision making

- 3.1. Consensus-based decision making. Working Groups make decisions through a consensus process (“Approval” or “Approved”). While the agreement of all Participants is preferred, it is not required for consensus. An individual's role will determine the extent of their decision making abilities. For example, the reviewer will determine consensus based on their good faith consideration of a number of factors, including the dominant view of the Working Group Participants and nature of support and objections. The reviewer will document evidence of consensus in accordance with these requirements.

- 3.2. Appeal process. Decisions may be appealed via an issue, and that appeal will be considered by the reviewer in good faith, who will respond in writing within a reasonable time.

4. Major release scheduling guidelines

The Steering Working Group has the role of defining major releases of DRR and shaping their content. The major release scheduling guidelines page discusses the objectives for

defining major releases and guidelines that the Steering Working Group (SWG) must follow in scheduling major releases.

5. Change control guidelines

The change control guidelines discusses how changes to DRR are controlled within and between releases, in particular:

- Principles
  - What we are trying to achieve with the change control guidelines;
  - What constraints/objectives we have for putting these guidelines in place
- Rules
  - The specific rules we want to define and enforce to meet the principles
- Evaluation methods
  - How we want to ensure that the rules are evaluated and enforced during development
  - This includes development processes (e.g. review and approval) as well as automated tooling (e.g. regression test cases)

6. Release build approval guidelines

For scheduling of minor, development, patch releases and approvals for all builds and releases, see Maintenance and release.

Major release scheduling guidelines

The Steering Working Group has the role of defining major releases of DRR and shaping their content. This section discusses the objectives for defining major releases and guidelines that the Steering Working Group (SWG) must follow in scheduling major releases.

Objectives of defining major releases

- To identify and communicate to users of DRR when changes will happen that could affect them in a profound way, e.g.
    - Changes to existing functionality that may create challenges for upgrading
    - Changes to technology architecture that may create challenges for upgrading
    - Introduction of major new functionality that may affect how users use DRR going forward
- To help developers of DRR understand the roadmap for the most critical changes to the DRR, so they can better plan their work
- To promote planned and new DRR capabilities to encourage adoption

Objectives of defining guidelines for scheduling and approving major releases

- To ensure that major releases are planned, scheduled, and approved in a predictable, consistent, and transparent way
    - Ensure smoother development
    - Reduce conflict
- To ensure that we follow industry best-practices for evolving software.

Overall principles for scheduling major releases

- Major releases shall be planned ahead of time and these plans reviewed and approved by the DRR SWG so that consumers of DRR are aware of the planned changes and can plan for those changes. There is a balance between moving too quickly (and creating many changes, potentially discouraging adoption) and moving too slowly (and not addressing major issues in a timely fashion). The DRR SWG will be tasked with assessing and maintaining that balance and communicating its decisions. That balance is likely to change over time as the DRR software matures; likely major release frequency will slow down in the future.

- Part of the role of the guidelines will be to help the DRR SWG to resist pressure to create too many major releases. However, the guidelines need to provide the DRR SWG with enough flexibility to address major challenges relatively quickly and flexibly when required.
  - Defining the guidelines is important to implement the above objective

Detailed guidelines – scheduling major releases

- No major release will be planned/scheduled (decision and content) without formal approval at a meeting of the DRR SWG
  - *Rationale:* Designation of a major release is an important decision that requires transparency and control
- The intention is that major releases shall be planned and reviewed at the DRR SWG at least 3 months ahead of the anticipated release date.
  - *Rationale:* Giving the community advance warning of major changes will help DRR users plan for how they will use DRR and avoid major surprises. It will also help DRR developers plan their own changes
- It is anticipated that for at least the next several years (say 4-5) at least one major release will be planned each year.
  - *Rationale:* we anticipate that there will be an accumulation of desired changes that cannot be accommodated within a minor release and we wish to ensure that these can be addressed without undue delay
- Any addition to the scope/contents of (or technical change to) a planned major release requires DRR SWG approval
  - *Rationale:* similar to the above guideline on scheduling major releases
- If planned scope items for a major release are not available in time for the planned release date, the DRR SWG will need to decide whether to slip the release date or drop the item, based on industry priorities
- These guidelines can be overridden in exceptional circumstances by a formal vote of the DRR SWG.
  - *Rationale:* Sometimes unanticipated issues will come up and we need the ability to move quickly in these cases. However, there should be an explicit decision process when breaking a guideline.

- These guidelines can be amended by the DRR SWG following a formal review process

Detailed guidelines – long-term planning and outreach

- Ideally the DRR SWG will establish plans for upcoming major releases for at least the following 9-12 months
    - *Rationale:* this provides transparency for the users and potential users of DRR (supporting adoption)
- Major release schedules shall be published on the DRR GitHub repository once approved by the DRR SWG (in the roadmap).
    - *Rationale:* as above

Detailed guidelines – changes vs major versions

- Breaking changes (as defined in the change control guidelines) can only be implemented in a major version
    - *Rationale:* this is required to ensure that within a single major version there is stability across minor versions.
- Changes (PRs) will be categorized into those that can only be done in major releases (because they contain breaking changes) and others. PRs requiring a major release shall only be approved for major releases.
    - *Rationale:* this is necessary to ensure that the meaning of major releases is enforced
- Even in a new major version, changes that are contrary to the change control guidelines will not be approved unless the DRR SWG executes an exception process.
    - *Rationale:* this is required to ensure that DRR provides stability across major versions, in terms of functionality that is supported
- When a major version includes breaking changes, the DRR SWG will endeavour to ensure that appropriate migration guides and transition plans are in place
    - *Rationale:* this is to support DRR users in migrating to new versions of DRR

release-schedule

title: Release schedule sidebar_label: Release schedule

The Release Schedule outlines the states that we move between versions of CDM and DRR moving from Development, to Production, to Maintenance. More information on dates can be found here

Change control guidelines

This section discusses how changes to the DRR are controlled between releases, in particular:

- Principles
  - What we are trying to achieve with the change control guidelines;
  - What constraints/objectives we have for putting these guidelines in place
- Rules
  - The specific rules we want to define and enforce to meet the principles
- Evaluation methods
  - How we want to ensure that the rules are evaluated and enforced during development
  - This includes development processes (e.g. review and approval) as well as automated tooling (e.g. regression test cases)

Change control principles

- We're trying to ensure rapid, smooth, and predictable evolution of the model by controlling when and how breaking changes are introduced.
  - We want to allow changes where needed, with defined process to make those changes, to meet evolved and improved understanding of the business and technical requirements.
  - We want to give ourselves some freedom to make changes more easily when there are newly introduced components/structures that may not be fully mature, but we don't want to spend a lot of effort on planning for that. We will do this using the pull request approval guidelines for bug fixes, giving some scope for correcting recently introduced changes.
- Prohibiting breaking changes within a major version should allow users to upgrade to minor versions more quickly and easily, and plan for when to implement larger changes.
  - By limiting and control the amount of change to key business models and technology structures, DRR users can have confidence that functionality they develop using DRR will continue to work with new versions of DRR with minimal effort, at least for a defined period of time.

Change control rules

- Unless explicitly indicated otherwise, components of DRR (such as data types and functions) will be under change control once released into production.
- Within multiple minor releases of a single major release, the following must be true:
    - Within business objects, any object that is valid in version M.N should be representable and valid in version M.N+1 .
        - For example, existing data fields may not be changed in type, reduced in cardinality, or removed, and new mandatory data fields may not be added.
        - Specific rules are described below in "Backward Compatibility"
        - All validations that pass in version M.N should also pass in version M.N+1
    - Function signatures may not be changed in such a way as to invalidate previous callers (e.g addition of new mandatory parameters, or removal/change of existing parameters.)
    - Test cases that passed in a prior version shall continue to work.
    - We allow some minor exceptions to these rules for newly introduced functionality that may not be fully formed, as part of the PR process for defect corrections.
    - Functionality shall not be removed between major versions without advance notice. (This can be done as part of the advance planning of a major version with SWG approval.

Please note that full, bidirectional interoperability between minor versions is not required. If an application uses functionality in version M.N, it does not need to fully interoperate with version M.N-1, assuming that the older version does not include that functionality. However, if an application uses functionality found in version M.N, it should be able to interoperate with version M.N+1.

Backward compatibility

Like other types of software, backward compatibility in the context of a domain model means that an implementor of that model would not have to make any change to update to such version.

- Prohibited changes:

    - Change to the structure (e.g. the attributes of a data type or the inputs of a function) or removal of any model element
    - Change to the name of any model element (e.g. types, attributes, enums, functions or reporting rules)

    - Change to any condition or cardinality constraint that makes validation more restrictive
- Allowed changes:
    - Change that relaxes any condition or cardinality constraint
    - Change to any mappings that improves, or at least does not degrade, the mapping coverage
    - Addition of new examples or test packs
    - Change to the user documentation or model descriptions
    - Addition of new data types, optional attributes, enumerations, rules or functions that do not impact current functionality

Exceptions to backward compatibility may be granted for emergency bug fixes following decision from the relevant governance body.

Change control evaluation and enforcement

- Designers and contributors to DRR are responsible for being aware of and following the change control guidelines. This includes flagging pull requests when they involve breaking changes to controlled objects.
    - Backward incompatible changes shall be documented and include a migration guide (remap from old structures and functions to the new)
- Reviewers will be responsible for assessing ("double checking") whether any changes may violate the change control guidelines, and flag questionable changes for further review. This process is described in more detail here
    - Part of the role of the DRR Working Group (DWG) and of the reviewers is to enforce these guidelines for any change.
- There will be a set of regression test cases developed for each supported major version. Subsequent DRR minor and major versions will be tested against these test cases and a report prepared indicating which cases succeed and fail, and this will be compared against the guidelines. For example:
    - DRR version 6.2 will be tested against the 6.1 test cases; all should succeed, unless included in the exception/noncontrolled list.
    - DRR version 6.0 will be tested against the latest 5.x test cases; the list of failures should be compared against the approved scope of change for 6.0. (NB: performing this test might involve making some technical changes to the 5.0 test cases to work with the 6.0 technical architecture if that has changed, but the functionality should not otherwise be changed.)

Contribution guidelines

These are the guidelines relevant to contributing to DRR with regards to issues and pull requests.

All contributions must be handled through a GitHub issue and approved on the Working Group.

Issues are used as the primary method for tracking any changes to the DRR models. Under the ‘Issues’ tab in GitHub, one of the three types of issues can be selected and populated (each with their own corresponding label):

1. Feature. Used to suggest functional improvements to DRR.

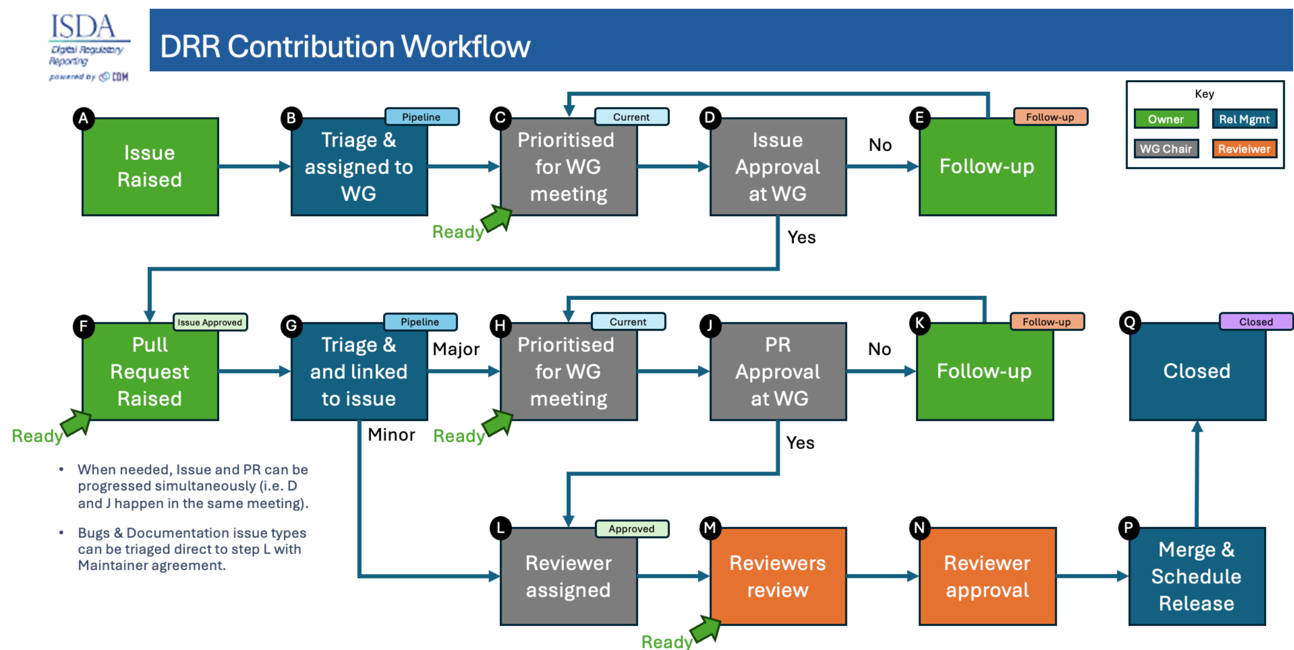
2. Task. Used to break down larger features into actionable units of work.

3. Bug. Used to report a bug or unexpected behaviour in the system.

Following approval, contributions are submitted via pull requests.

1. Issue lifecycle

All issue types follow the same general lifecycle.



2.1. Triage

The DRR Release Management team should pick up the new Issue and perform the following activities:

First, review the **Issue quality**; contact the creator if necessary (or make minor updates directly).

Next, **tag the Issue**:

1. Add appropriate labels:
    i. "backward-incompatible":
    ii. "complex"
    iii. "documentation"
    iv. "technical"
    v. Note that other Label values are available, such as "dependencies" but these are auto-assigned, and should only be used by bots and other automation including Rosetta.
2. **Assign the Issue** to a Working Group by adding as a Project
3. If known, the following additional information can be assigned on the Issue
    i. Assignee: To the user working on the issue, which can be different to the issue owner
    ii. Milestone: If a target Release is specified or clear, select it as the Milestone. For issues with releases on multiple versions, the milestone set is the earliest version milestone, while the PRs of the later versions are linked to their related versions
    iii. Relationships to other issues

Once all the above activities are complete, the "Triage" label is removed.

2.2. Discussion

- Issues are discussed on the working group call where they are either rejected, approved, or moved to follow-up pending further discussion.

3. Pull request workflow

The next section contains more information on the workflow followed for Pull Requests.

3.1. Pull request creation

- We welcome contributions from any member of the DRR repository.
- Pull request must be linked to a specific issue.
- Once the issue has been approved on the working group, the Release Management team will link the PR to the issue

3.2. Reviewing/discussion

- All reviews will be completed using the review tool.
- Reviewers must validate the quality of the PR and the quality of the release note comment.
- A "Comment" review should be used when there are questions about the spec that should be answered, but that don't involve spec changes. This type of review does not count as approval.
- A "Changes Requested" review indicates that changes to the spec need to be made before they will be merged.
- Final approval is required by a reviewer. Merging is blocked without this final approval. Reviewers will factor reviews from all other reviewers into their approval process.

3.3. Responsive

Pull request owners should try to be responsive to comments by answering questions or changing text. Once all comments have been addressed, the pull request is ready to be merged.

3.4. Merge or close

- A pull request should stay open until a reviewer has marked the pull request as approved.
- Pull requests can be closed by the author without merging.
- Pull requests may be closed by a reviewer if the decision is made that it is not going to be merged.
- Pull requests should not be merged unless linked to an issue.

4. Adoption of contributions

Contributions merged into the master branch of the DRR repository will form part of the next relevant DRR release based on the release the issue is tagged against. This must be

approved by the Working Group voting participants before it is accepted as a draft and subsequently released.

DRR development design principles

The purpose of this section is to detail DRR design principles that any contribution to DRR development must adhere to. DRR supports the market objectives of standardisation via a set of design principles that include the following concepts:

- **Normalisation** through abstraction of common components, e.g. *price* or *quantity*
- **Composability** where objects are composed and qualified from the bottom up
- **Mapping** to existing industry messaging formats, e.g. *FpML*
- **Embedded logic** to represent industry processes, e.g. data validation or state-transition logic
- **Modularisation** into logical layers, using *namespace* organisation

Normalisation

To achieve standardisation across products and asset classes, DRR identifies logical components that fulfil the same function and normalises them, even when those components may be named and treated differently in the context of their respective markets. By helping to remove inefficiencies that siloed IT environments can create (e.g. different systems dealing with cash, listed, financing and derivative trades make it harder to manage aggregated positions), such design reaffirms the goal of creating an interoperable ecosystem for the processing of transactions across asset classes.

An example of this approach is the normalisation of the concepts of *quantity*, *price* and *party* in the representation of financial transactions. DRR identifies that, regardless of the asset class or product type, a financial transaction always involves two counterparties *trading* (i.e. buying or selling) a certain financial product in a specific quantity and at a specific price. Both quantity and price are themselves a type of *measure*, i.e. an amount expressed in a specific unit which could be a currency, a number of shares or barrels, etc. An exchange rate between currencies, or an interest rate, also fit that description and are represented as prices.

This approach means that a single logical concept such as *quantity* represents concepts that may be named and captured differently across markets: e.g. *notional* or *principal* amount etc. This in turn allows us to normalise processes that depend on this concept: for instance, how to perform an allocation (essentially a split of the quantity of a

transaction into several sub-transactions) or an unwind, instead of specialised IT systems handling it differently for each asset class.

It is imperative that any request to add new model components or extend existing ones is analysed against existing components to find patterns that should be factored into common components and avoid specialising the model according to each use case. For instance, in the model for *averaging* options (often used for commodity products, whereby multiple price observations are averaged through time to be compared to the option's strike price), the components are built and named such that they can be re-used across asset classes.

Composability

To ensure re-usability across different markets, DRR is designed as a composable model whereby financial objects can be constructed bottom-up based on building-block components. A composable and modular approach allows for a streamlined model to address a broad scope of operational processes consistently across firms' front-to-back flows and across asset classes. The main groups of composable components are:

- **Financial products**: e.g. the same *option* component is re-used to describe option payouts across any asset class, rather than having specialised *Swaption*, *Equity Option* or *FX option* etc. components.
- **Business events** that occur throughout the transaction lifecycle are described by composing more fundamental building blocks called *primitive events*: e.g. a *partial novation* is described by combining a *quantity change* primitive event (describing the partial unwind of the transaction being novated away) and a *contract formation* primitive event (describing the new contract with the novation party).
- **Legal agreements** that document the legal obligations that parties enter into when transacting in financial products are constructed using *election* components associated to functional logic that is re-usable across different types of agreement: e.g. the same logic defining the calculation of margin requirements can be re-used across both initial and variation margin agreements.

In this paradigm, the type of object defined by DRR, whether a financial product, business event or legal agreement, is not declared upfront: instead, the type is inferred through some business logic applied onto its constituents, which may be context-specific based on a given taxonomy (e.g. a product classification).

The benefit of this approach is that consistency of object classification is achieved through how those objects are populated, rather than depending on each market participant's implementation to use the same naming convention. This approach also avoids the model relying on specific taxonomies, labels or identifiers to function and provides the flexibility to maintain multiple values from different taxonomies and identifier sets as data in the model related to the same transaction. This has a number of useful application, not least for regulatory purposes.

Mapping

To facilitate adoption by market participants, DRR is made compatible with existing industry messaging formats. This means that DRR does not need to be implemented "wholesale" as a replacement to existing messaging systems or databases but can coexist alongside existing systems, with a translation layer. In fact, DRR is designed to provide only a logical model but does not prescribe any physical data format, neither for storage nor transport. This means that translation to those physical data formats is built-in, and DRR is best thought of as a logical layer supporting inter-operability between them.

Note: Although DRR features a *serialisation* mechanism (currently in JSON), this format is only provided for the convenience of representing physical DRR objects and is not designed as a storage mechanism.

The need for such inter-operability is illustrated by a typical trade flow, as it exists in derivatives: a trade may be executed using the pre-trade FIX protocol (with an FpML payload representing the product), confirmed electronically using FpML as the contract representation, and reported to a Trade Repository under the ISO 20022 format. What DRR provides is a consistent logical layer that allows to articulate the different components of that front-to-back flow.

In practice, mapping to existing formats is supported by *synonym* mappings, which are a compact description in DRR of how data attributes in one format map to model components. In turn, those synonym mappings can support an *ingestion* process that consumes physical data messages and converts them into DRR objects.

DRR recognises certain formats as de-facto standards that are widely used to exchange information between market participants. Their synonym mappings are included and rigorously tested in each DRR release, allowing firms that already use such standards to bootstrap their DRR implementation. Besides, because most standard messaging formats are typically extended and customised by each market participants (e.g. FpML or FIX), DRR allows the synonym representation for those standards to be similarly inherited and extended to cover each firm's specific customisation.

Embedded logic

DRR is designed to lay the foundation for the standardisation, automation and inter-operability of industry processes. Industry processes represent events and actions that occur through the transaction's lifecycle, from negotiating a legal agreement to allocating a block-trade, calculating settlement amounts or exchanging margin requirements.

While FINOS defines the protocols for industry processes in its documentation library, differences in the implementation minutia may cause operational friction between market participants. Even the protocols that have a native digital representation have written specifications which require further manual coding in order to result in a complete executable solution: e.g. the validation rules in FpML, the Recommended Practices/Guidelines in FIX or CRIF for SIMM and FRTB, which are only available in the form of PDF documents.

Traditional implementation of a technical standard distributed in prose comes with the risk of misinterpretation and error. The process is duplicated across each firm adopting the standard, ultimately adding up to high implementation costs across the industry.

By contrast, DRR provides a fully specified processing model that translates the technical standards supporting industry processes into a machine-readable and machine-executable format. Systematically providing the domain model as executable code vastly reduces implementation effort and virtually eliminates the risk of inconsistency. For instance, DRR is designed to provide a fully functional event model, where the state-transition logic for all potential transaction lifecycle events is being specified and distributed as executable code. Another DRR feature is that each model component is associated with data validation constraints to ensure that data is being validated at the point of creation, and this validation logic is distributed alongside the model itself.

Modularisation

The set of files that define DRR data structures and functions are organised into a hierarchy of namespaces. The first level in the namespace hierarchy corresponds to the layer of DRR that the components belong to, and those DRR layers are organised from inner to outer-most.

DRR roadmap governance and process

DRR is constantly evolving, with regular updates and improvements contributing to its development roadmap. There are currently two Working Groups which oversee developments across DRR: the ISDA DRR Working Group and the DRR Steering Group. The Steering WG oversees the process and the integration across Working Groups.

2026 DRR Community Roadmap

DRR Working Groups

- **ISDA DRR Working Group** (Chair: Tabish Ahmed. Fortnightly)
- **Steering Working Group** (Chair: David Shone. Monthly)

ISDA DRR Working Group

The DRR WG is a call with representatives from various organisations across the industry with experience or interest in DRR that meets fortnightly to discuss issues that have been raised around DRR e.g. bugs, feature changes etc, as well as industry changes, deadlines and updates.

Responsibilities

- Reviews, approves, and implements contribution proposals to DRR.
- Arbitrates disputes arising from DRR WG contributions.
- Guards and enforces design principles and guidelines.
- Triage and facilitates long-dated PRs and Issues.
- Meets fortnightly, with sub-groups as required.

Meetings

Join us on the Second and Fourth Tuesday at 9 AM EST (2 PM GMT).

Membership: You can join this working group if you're a member of a digital-regulatory-reporting repo or if you're on the mailing list for the teams invite from ISDADataReporting@isda.org.

You can also find details of upcoming calls on the DRR Discussions board.

To view meeting notes and agendas, view our current and past GitHub Meeting Issues.

DRR Steering Working Group

Responsibilities

This group focuses on:

- Overall governance
- Release schedules for work completed
- DRR production release schedule
- Agreements of roadmaps, strategy and direction.

Membership: Invite only. However, you can contact the group by email: cdmdrr@isda.org.

To view meeting notes and agendas, view our current and past GitHub Meeting Issues.

DRR Working Group Terms of Reference

1 January 2026

1. Establishment and mandate

The Digital Regulatory Reporting (DRR) Working Group (the **"Working Group"**) is reconstituted as a single, consolidated DRR Working Group operating under the strategic oversight of the DRR Steering Group (the **"Steering Group"**). This restructure aligns DRR governance with the Common Domain Model (CDM) Working Group operating model and establishes GitHub as the primary mechanism for agenda and issue management.

1.1 Rationale for transition

ISDA is transitioning to this consolidated Working Group model to strengthen the DRR governance framework in preparation for expanded joint industry work, including increased coordination with industry partners such as ISLA and ICMA, and to support upcoming and evolving reporting jurisdictions that include a wider range of in-scope products, including OTC derivatives and other product classes. A single, globally consistent Working Group with transparent GitHub-based workflows will better enable multi-association collaboration, shared design discussions, and scalable contribution practices as DRR adoption and scope evolve.

The DRR Working Group is established to provide practical, multijurisdictional delivery support to the DRR initiative by translating Steering Group priorities into executable workstreams, technical outcomes, and implementable artefacts across the global derivatives market. The Working Group serves as an implementation-focused forum for adoption and non-adoption entities, reporting service providers, and other stakeholders to identify, discuss, evaluate, prioritise and progress DRR projects, enhancements, maintenance and adoption activities in a transparent and globally consistent manner.

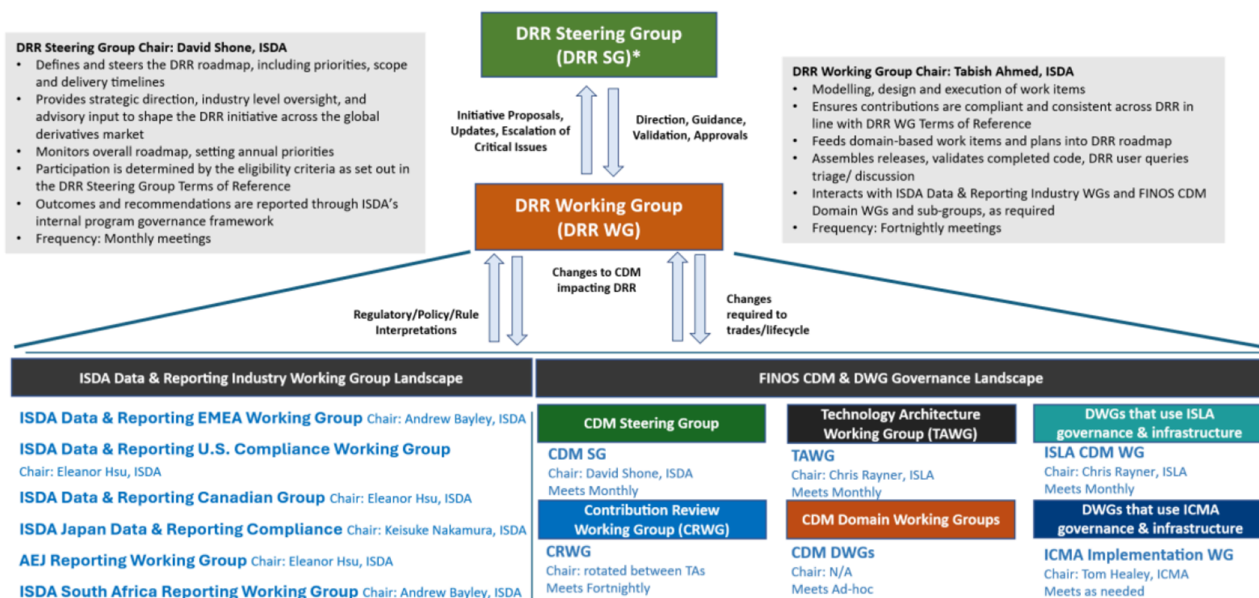
The Working Group's overarching objective is to promote safe, transparent, accurate and efficient regulatory reporting practices through delivery and continuous improvement of the DRR model. To achieve this, the Working Group will:

- Operate in the same format as CDM Working Groups, chaired by ISDA

- Manage the agenda and priorities through the DRR GitHub Projects page, with priorities set by the Chair in consultation with members and aligned to DRR Steering Group direction;
- Provide a forum for all contributors to raise and review pull requests and issues directly in GitHub, ensuring traceability, transparency, and consistency of the DRR contribution process
- Allocate dedicated agenda time for multi-lateral design discussions and community-wide questions from participating firms
- Draft, validate and propose enhancements to DRR logic, code and associated artefacts for DRR Steering Group review and escalation where needed
- Identify implementation issues, interpretive questions, and adoption blockers, and propose solutions for industry consideration
- Contribute to DRR education through playbooks, examples, implementation notes and knowledge-sharing sessions.

The **DRR Working Group**, as shown in the diagram, is intended to function as equivalent to other Domain Working Groups within the broader FINOS Working Group structure, enabling items impacting CDM design or contribution to be raised ultimately through the FINOS CDM Contribution Review Working Group (CRWG). Initial interaction with CDM WGs will likely occur at the Domain WG level e.g. product enhancements may go to the Derivatives WG or ISLA/ICMA WGs or collateral enhancements may go to the Collateral WG.

DRR Governance Landscape – updated and effective January 1st, 2026



* Outcomes/ recommendations from the DRR SG are reported up through ISDA's internal program governance framework

1 ISDA Safe, Efficient Markets

1.2 Key benefits of this approach

- Alignment of the DRR contribution process with CDM, enabling standardisation and greater process efficiency
- Familiarisation of DRR member firms with CDM working practices, ensuring global consistency and minimising the learning curve as they progress to CDM adoption
- Simplified governance through a single global DRR Working Group forum under the Steering Group
- Enhanced transparency and multi-regional coordination through GitHub-based backlog and agenda management.

2. Participation and consensus

2.1 Open participation

Participation in the Working Group is open to all interested stakeholders across adoption entities, service providers, infrastructure providers, relevant industry association, and other parties with a legitimate interest in DRR adoption, implementation and maintenance. ISDA will maintain a participant list and may invite subject-matter experts to relevant discussions as required.

2.2 Consensus and recommendations

The Working Group operates in a consultative and delivery capacity and does not hold final decision-making authority. The Working Group will seek to achieve majority-based consensus to form recommendations, technical outputs, and proposed deliverables for ISDA consideration and escalation to the Steering Group where needed.

Where majority consensus cannot be achieved, the Chair may:

- Defer the matter for further analysis and re-discussion; and/or
- Escalate the matter to the DRR Steering Group or relevant DRR governing body for review or determination.

3. Meetings

3.1 Frequency

Meetings shall be held fortnightly on an alternate-week cadence to the FINOS CDM CRWG, enabling ongoing alignment and collaboration. Additional ad-hoc sessions may

be convened as required by program milestones, UAT cycles or emerging issues.

3.2 Recording and minutes

Meetings may be recorded by ISDA and AI-generated transcripts may be used to assist minute-taking. Minutes shall:

- Summarise discussions, key points, and actions
- Be circulated within a reasonable timeframe
- Be archived in a designated repository by ISDA

3.3 Agenda, inputs and outputs

A standing agenda and materials, together with relevant inputs and outputs for each meeting, shall be prepared in advance of each meeting and published via the DRR GitHub Projects page. Working Group participants will be invited to propose agenda items or topics for upcoming meetings at least two days prior to each scheduled meeting, either through GitHub issues or directly to the Chair. Issues/topics raised after that may still make it into the meeting time, but this is not guaranteed.

Key outputs include, as applicable:

- GitHub issues capturing questions, defects and enhancement requests
- Pull requests for DRR logic, code, models, tests, and documentation
- Design notes and interpretive guidance arising from multilateral discussions
- Recommendations for DRR Steering Group consideration

4. Confidentiality

Participants shall maintain the confidentiality of non-public information disclosed during meetings, except where disclosure is required by law or expressly authorised by ISDA or the DRR Steering Group. Materials shared through the DRR Working Group shall not be used for commercial or competitive purposes.

5. Competition law compliance

All participants shall comply with applicable competition and antitrust laws. Participants agree not to exchange competitively sensitive information, discuss individual pricing or strategic commercial decisions, use the Working Group to reach any anticompetitive

understanding, or otherwise coordinate market conduct. Each participant acts independently in all business decisions.

6. Record keeping and transparency

ISDA will maintain records of:

- Participation and attendance
- Agendas, minutes, and actions
- GitHub Projects backlogs, issues, pull requests and associated artefacts
- Written recommendations or outputs escalated to the DRR Steering Group

7. Amendments and review

These Terms of Reference may be reviewed and amended periodically by ISDA to ensure continued relevance, alignment with DRR Steering Group direction, CDM/FINOS governance and industry needs.

8. Effective date

These Terms of Reference are effective as of 1 January, 2026, and supersede any prior DRR Working Group terms.

Maintenance and release

Reviewing model changes

Contributions are reviewed by the DRR Working Group

Review checklist

Before starting to review a contribution, the reviewer should go through the following review checklist:

- Review Pull Request to assert that:
    - Model changes fulfil the proposed design and use-case requirements
    - Mapping have been updated and output (JSON) looks correct
    - Contributed model version is not stale and does not conflict with any recent changes
    - Changes are in accordance with DRR governance guidelines, including the change control guidelines in the change control guidelines page

---

Note: It is not yet possible to verify that mapping, validation and qualification expectations have been maintained by looking at the output of the Pull Request and DRR build only.

- DRR build process completed with no errors or test failures
- Review additional samples provided (if use-case is not covered by existing samples)
- All model components positioned in the correct namespace
- All model components have descriptions
- Additional documentation provided, if necessary.
- Release note provided

Any review feedback should be sent to the Contributor as required via Slack, email or in direct meetings.

Model maintenance and release

After learning about how to edit the model, please refer to this section to learn more about its maintenance.

Introduction

Before the Pull Request can be merged into DRR's main branch, some work is usually required by the reviewer to preserve the integrity of the model source code and of its downstream dependencies.

Post-review technical tasks

A number of technical tasks may need to be performed on the Pull Request once it is approved:

- **Stale DRR version**: Contribution is based on an old DRR version and model changes conflict with more recent changes. If the conflicting change is available in Rosetta, the contributor should be asked to update their contribution to the latest version and resubmit. If the conflicting change is not yet available in Rosetta, this merge will need to be handled by DRR reviewer.
- **Failed unit tests**: Java unit tests in DRR project may fail due to problems in the contributed changes. Alternatively it may be that the test expectations need to be updated. The reviewer should determine the cause of the test failure and notify either the Contributor or work on adjusting the test expectations.
- **Code generation**: Model changes may cause code generator failures (e.g., Java, C#, Scala, Kotlin etc.). In the unlikely event of code generation failures, these will need to be addressed by the reviewer.

The change can be merged into the main DRR code base only upon:

- approval by DRR reviewers and/or DRR Architecture and Review Committee,
- successful completion of all the above technical tasks, and
- successful builds of DRR and all its downstream dependencies.

Releasing model changes

Once the contributed model change has been merged, a new release can be built, tested and deployed. The reviewer will work with DRR Owners and the Contributor on a deployment timeline.

The following release checklist should be verified before deploying a new model:

- Update DRR version number, using the semantic versioning format
- Build release candidate, and test
- Build documentation website release candidate, and test
- Deploy release candidate and notify channels if need be
- (Currently done at a later stage) Update the latest DRR version available in Rosetta

---

Note: When the release process is handled through Rosetta Deploy, the reviewer should contact the Rosetta support team to request that deployment and discuss a timeline for the release.

Release build approval guidelines

This section covers scheduling of minor, development, and patch releases, and approvals for all builds and releases.

Development release scheduling and approvals

- Development releases may be scheduled by the reviewers to optimize development resources, based on the queue of approved PRs
    - There is no particular desired/expected release frequency; releases may be cut as soon as there is an approved PR, or several PRs may be consolidated into a single release at the convenience of the reviewers and dev staff
    - *Rationale:* Development releases are expected to change in functionality, and getting changes out as quickly as practical is usually desirable.
    - Each development release shall require the approval of one reviewer once all the PRs are approved, and the test cases all pass successfully.
- Development releases shall be reported in brief to the DRR WG and the DRR SWG

Major production release build and release approvals

- Major production releases will be scheduled by the DRR SWG as described above
- Each major production release shall require the approval of two reviewers after the following are complete:
    - The scope of the major production release is finalized and ratified by the DRR SWG

    - All approved PRs for the major production release are complete
    - The DRR SWG reviews the final list of enhancements in the release and signs off on releasing the development version into production

Minor production release scheduling and approvals

- Minor production releases may be scheduled by the reviewers based on the queue of approved PRs
- Minor production releases to introduce enhancements should be combined to minimize the number of production releases, targeting minor production releases to be issued around four weeks or so as long as there is a queue of approved PRs. (This frequency can be increased in times of urgent need for new functionality).
    - *Rationale:* Minimizing the number of production releases will help with supportability, by reducing the number of releases that end users wishing to remain current need to consider, and reducing communications overhead.
- Each minor production release shall require the approval of two reviewers.
- Minor production releases shall be reported in brief to the CRWG and the SWG,
- A roadmap of anticipated minor production releases shall be reported by the reviewers to the DRR WG based on PRs that are in process.

Production patch release scheduling and approvals

- Production patch releases to correct defects without releasing new functionality may be scheduled by the reviewers based on the presence of approved defect correction PRs, or other non-functional PRs (e.g. security remediations).
- Production patch releases require the approval of one reviewer
- Production patch releases shall be reported to the CRWG.

Summary of release approval requirements

Type of release	Approval requirement	Notes
Major Release (6.0.0)	2 reviewers	Scheduling via SWG; Include analysis of the changes from last major release as part of the approval
Minor Release (6.1.0)	2 reviewers	Scheduling is up to the reviewers, but aim to keep to around every 4 weeks and no more than fortnightly

Type of release	Approval requirement	Notes
Patch Release (6.1.1)	1 reviewer	Scheduling is up to the reviewer
Development Release (6.0.0-dev.13)	1 reviewer	Scheduling is up to the reviewer

Versioning

DRR is released using the semantic versioning 2.0 system - See SemVer 2.0.0. At high-level, the format of a version number is MAJOR.MINOR.PATCH (e.g. `1.23.456`), where:

- A MAJOR (`1`) version may introduce backward-incompatible changes and will be used as high level release name (e.g. "DRR Version 1"). See our major release scheduling guidelines for guidelines on how major releases are scheduled.
- A MINOR (`23`) version may introduce new features but in a backward-compatible way, for example supporting a new type of event or function.
- A PATCH (`456`) version is for backward-compatible bug fixes, for example fixing the logic of a condition.

In addition, pre-release versions of a major release will be denoted with a DEV tag as follows:

- MAJOR.0.0-DEV.x (e.g. `1.0.0-DEV.789`), where x gets incremented with each new pre-release version until it becomes the MAJOR.0.0 release.

The minor, patch and pre-release numbers may sometimes increment by more than one unit. This is because release candidates may be created but not immediately released. Subsequently, a version associated with the next incremental unit may be released that includes the changes from the earlier release candidate.

Unless under exceptional circumstances, the major number will be incremented by one unit only.

Version availability

Several versions of DRR will be made available concurrently, with a dual objective.

- The latest *development* version (i.e. with a pre-release tag) fosters continued, rapid change development and involves model contributions made by the industry community. Changes that break backward compatibility are allowed. This development version is available in read-only and read-write access on DRR's modelling-platforms.
- The latest *production* version (i.e. without any pre-release tag) offers a stable, well-supported production environment for consumers of the model. Unless under

exceptional circumstances, no new disruptive feature shall be introduced, mostly bug fixes only. Any change shall adhere to a strict governance process as it must be backward-compatible. Generally, it can only be developed by a DRR Maintainer. This production version is available in read-only access through DRR's modelling-platforms.
- Earlier production versions, when still supported, are also available in read-only access for industry members who are still implementing older versions of the model. Over time, those earlier production versions enter *long-term support* in which supportability will be degraded, until they eventually become unsupported.

Example. Assume that the latest major release of the model is 5. The various versions available would be as follows:

- 5.0.0 and any subsequent 5.x would be the latest production version. Backward-compatibility to the initial 5.0.0 version would be maintained for any 5.x successor version.
- The latest 4.x and 3.x may also be supported, but 2.x could be under long-term support and 1.x unsupported altogether.
- 6.0.0-DEV.x would be the latest development version. It can, and will generally, contain changes that are not backward-compatible with version 5. Backward-compatibility between successive 6.0.0-DEV.x versions is also not assured. Once fully developed, version 6.0.0 can be tagged as a major release and becomes the new latest production version.

Please note, all contributions must follow the change control guidelines. Guidelines for approval of new versions are described in the Maintenance and Release Guidelines.

View the current planned release timeline.

Note: The above example is for illustration only and not indicative of actually supported DRR versions.

Editing the model

When editing DRR, it's essential to go through the following modelling checklist:

- DRR version: use the latest available development version
- Syntax: no syntax warnings or errors
- Compilation: model compiles ok with no *static compilation* errors
- Testing: all translate regression tests expectations for mapping, validation and qualification maintained or improved. Additional test samples may be needed if use-case is not covered by existing samples.
- Namespace: all model components positioned in the correct namespace
- Descriptions: all model components have descriptions

The following sections detail that checklist. When using the Rosetta Design web application to edit the model, the Contributor should also refer to the Rosetta Design Guide.

DRR version

To the extent possible it is recommended that the Contributor keeps working with a version of DRR that is as close as possible to the latest to minimise the risk of backward compatibility.

Please refer to the Source Control Integration Guide for more information.

Syntax

The model is represented in the Rune DSL syntax. All syntax warnings and errors must be resolved to have a valid model before contributing any changes. For further guidance about features of the syntax, please refer to the Rune DSL Documentation.

In Rosetta Design, that syntax is automatically checked live as the user edits the model, as described in the Rosetta Design Content Assist Guide.

Compilation

Normally, once the model is syntactically correctly edited, valid code is being auto-generated and compiled. However, certain model changes can cause compilation errors when changes conflict with static code (e.g. certain mapper implementations).

The Rosetta support team can help resolve these errors before the changes are contributed. In most cases you will be able to contact the team via the In-App chat. If the support team identifies that significant work may be required to resolve these errors, they will notify the Contributor who should then contact DRR reviewer originally appointed for the proposed change and/or DRR Owners. The latter will be able to assist in the resolution of the issues.

For more information about auto-compilation using the Rosetta DSL, please refer to the Rosetta Auto Compilation Guide.

Testing

DRR has adopted a test-driven development approach that maps model components to existing sample data (e.g., FpML documents or other existing standards). Mappings are specified in DRR and the sample data are collected into a Test Pack. Each new model version is regression-tested using those mappings to translate the sample data in the Test Pack and then comparing against the expected number of mapped data points, validation and qualification results.

When using Rosetta to edit the model, contributors are invited to test their model changes live against the Test Pack using the Rosetta Translate application, referring to the Rosetta Translate Guide. When editing existing model components, the corresponding synonyms should be updated to maintain or improve existing mapping levels. When adding new model components, new sample data and corresponding synonym mappings should also be provided so the new use-case can be added to the set of regression tests.

Please refer to the Mapping Guide for details about the synonym mapping syntax.

Namespace

All model components should be positioned appropriately in the existing namespace hierarchy. If the proposed contribution includes changes to the namespace hierarchy, those changes should be justified and documented. Any new namespace should have an associated description, and be imported where required.

Please refer to the namespace documentation section for more details.

Descriptions

All model components (e.g. types, attributes, conditions, functions etc.) should be specified with descriptions in accordance with the guidance in Descriptions.

Contributing model changes

Contribution checklist

Before you start contributing your model changes, please go through the following contribution checklist:

- Specify a meaningful title and description for the contribution
- Notify DRR reviewers (via email or Slack) of the submitted contribution
- Include:
  - Any notes on expected mapping, validation or qualification changes (success numbers should not decrease)
  - Additional data samples, if necessary
  - Documentation adjustment, if necessary
  - Release notes
  - Any other additional materials or documentation that may help with the review and approval process

---

Note: A contribution should be a whole releasable unit and its size calibrated in accordance with DRR's development roadmap process.

Contributing

Changes are contributed by submitting a Pull Request for review into the DRR source-control repository. This pull request will invoke a build process to compile and run all DRR unit tests and regression tests.

Given the alignment:

> 1 pull request = 1 contribution = 1 releasable unit = 1 user story,

we recommend labelling the pull request with the user story label, i.e. "STORY-XYZ: ..." to facilitate its tracking.

Note: All contributions are submitted as candidate changes to be incorporated under DRR licence.

When using Rosetta to contribute model changes, the contribution interface allows to specify a title and description for the contribution. Those inputs are used to create a Pull Request on a one-off branch in the source-control repository. Please refer to the Rosetta Workspace Contribution Guide for more information.

Documentation

DRR documentation must be kept up-to-date with the model in production. Where applicable, the Contributor should provide accompanying documentation (in text format) that can be added to DRR documentation for their proposed changes.

The documentation includes code snippets that directly illustrate explanations about certain model components, and those snippets are validated against the actual model definitions. When a model change impacts those snippets, or if new relevant snippets should be added to support the documentation, those snippets should be provided together with the documentation update.

Release note

A release note should be provided with the proposed model change that concisely describes the high-level conceptual design, model changes and how to review.

Reviewing model changes

Review checklist

Before starting to review a contribution, DRR reviewer should go through the review checklist.

Note: It is not yet possible to verify that mapping, validation and qualification expectations have been maintained by looking at the output of the Pull Request and DRR build only. Please refer to the downstream dependencies section for more details.

- DRR build process completed with no errors or test failures
- Review additional samples provided (if use-case is not covered by existing samples)
- All model components positioned in the correct namespace
- All model components have descriptions
- Additional documentation provided, if necessary.
- Release note provided

Any review feedback should be sent to the Contributor as required via Slack, email or in direct meetings.

Note: Depending on the size, complexity or impact of a contribution, DRR reviewer can recommend for the contribution to be presented with an appropriate level of details with DRR Architecture and Review Committee for further feedback. DRR reviewer will work with the Contributor to orchestrate that additional step. The additional feedback may recommend revisions to the proposed changes. When it is the case the review process will iterate on the revised proposal.

Get involved

DRR is a collaborative initiative. Learn how to contribute and help shape its evolution.

Contribute to DRR documentation

Tell us how you'd like to improve our tutorials, reference materials and examples.

Contribute to DRR

How to propose changes, submit improvements and participate in model development.

Contribute to DRR documentation

Good documentation helps everyone – from new developers to regulators – understand how DRR works.

You can contribute by:

- Clarifying confusing sections
- Adding missing explanations
- Updating pages when the model changes
- Improving examples or diagrams

Even small improvements can make a big difference.

To contribute to the documentation, send your suggestion to: CDMDRR@isda.org

Contribute to DRR

DRR is an open-access project built by the industry, for the industry. Anyone can potentially contribute – whether you're fixing a bug, improving a rule, updating documentation, or helping with release notes. This guide explains, in simple terms, how you can get involved and why your contributions matter.

Benefits of contributing

1. Improve the industry standard: DRR is used across banks, vendors and regulators. Your contribution helps improve the shared logic everyone relies on.

2. Build your expertise: Working on DRR gives you hands on experience with:

- The CDM
- Regulatory reporting logic
- Open-access governance
- Real-world financial data models

This is valuable experience for any developer or modeller working in finance or regtech.

3. Help others understand the rules: Clear release notes, examples and documentation make DRR easier for everyone – especially new contributors.

4. Shape the future of regulatory reporting: DRR is becoming the industry's shared source of truth. Contributing means you're helping to define how reporting works across jurisdictions.

1. How to contribute (step by step)

If you're comfortable reading DRR logic, you can help improve:

- Reporting rules
- Validation rules
- Mapping logic
- Example datasets

Your contribution might include:

- Fixing an incorrect rule

- Updating logic to match new regulatory guidance
- Improving examples so they're easier to understand

When describing rule changes, be explicit about:

- What changed
- Which examples were affected
- What the new expected output is

1.1 Make your change

Before making your change, please see additional details for editing DRR Further details on design-principles for changes made can be found here You can request changes using the pull request system in the GitHub repository. Changes might include:

- Editing a Markdown file
- Updating a rule
- Fixing an example
- Writing release notes

1.2 Explain your change

Every contribution should include:

- What changed
- Why it changed
- Where the change appears (e.g. file, rule, example)

Further information around change-control-guidelines

1.3 Submit your contribution

You can contribute through the normal workflow in the GitHub repository:

- Fork the repository
- Make your changes
- Submit a pull request

A reviewer will look at your update, ask questions if needed, and help you get it merged.

Details on the how contributions are discussed and approved can be found here

2. Pull request (PR) standards

All contributions to DRR must be submitted via a pull request (PR) that meets the required standards.

2.1 Title

PR titles must be clear, specific, and descriptive. A reviewer should be able to understand the nature and scope of the change without opening the PR.

Do

- Clearly state **what** is changing and **where**.
- Reference the affected regime, common type, or architectural layer where applicable.

Do not

- Use vague titles such as “Updates”, “Fixes”, or “Changes”.

Example

- `Add EMIR refit fields to CommonTransactionReport`
- `Refactor ASIC margin mapping to use DRR base abstractions`

2.2 Background and context

Each PR must include sufficient background to explain **why** the change is required.

Do

- Describe the regulatory, architectural, or functional motivation.
- Reference external specifications, tickets or regulatory texts where applicable.
- Explain why the chosen approach is appropriate within the DRR architecture

2.3 Explanation and justification of changes

All non-trivial changes must be explicitly explained and justified in the PR description.

Do

- Where possible, include references to supporting documentation or primary regulatory sources **directly within the model** using appropriate `doc Reference` or `regulatory Reference` annotations.

- Clearly describe the behavioural or structural changes introduced.
- Call out any deviations from existing patterns or common implementations.
- Justify any relaxation or tightening of type constraints.

Where changes affect existing behaviour, the PR must clearly state:

- **What** behaviour has changed.
- **Why** the change is necessary.
- **Which** jurisdictions or reports are impacted.

2.4 Review standards

PRs must demonstrate compliance with DRR 7 architectural boundaries. Further information on review standards can be found here

Do

- Confirm that all CDM interactions occur exclusively via the DRR `base` layer.
- Confirm that all changes have valid doc references or regulatory references.

Do not

- Use duplicate or poor quality code.
- Introduce direct CDM usage outside of `base`.
- Add expectation diffs (differences) without explanations

2.5 Reviewer expectations

PRs that lack sufficient explanation, justification or architectural alignment may be returned for rework, even if the implementation is technically correct.

3. Architectural overview

All interactions between DRR and CDM **must** be implemented on the **DRR base** layer. Direct references to CDM types or semantics outside of `base` are not permitted.

This means that all regimes report against shared common report types:

- `CommonTransactionReport`
- `CommonMarginReport`
- `CommonValuationReport`

Each common report type extends a corresponding `CriticalDataElement` (CDE), enabling consistent modelling of shared regulatory concepts and significantly reducing duplication across regimes.

3.1 Namespace dependency rules

This structure enforces a strict hierarchy governing allowable dependencies:

- **Regime namespaces** may import from `common` and `cde`
- **Common namespaces** may import from `cde`
- **CDE namespaces** may only import from `base`

4. Contribution guidelines

4.1 Type structuring and metadata

Rune DSL allows regulatory metadata to be attached directly to both simple and complex types.

Do

- Attach `label`, `regulatoryReference`, and `ruleReference` metadata directly to types.
- Follow the standard annotation order:
  1. `label`
  2. `regulatoryReference`
  3. `ruleReference`

Do not

- Use empty `ruleReference` declarations to capture regulatory metadata.
- Rely on `rule source` (deprecated).

Examples

```
override attribute boolean (0..1)
    [label "Test Label"]
    [regulatoryReference Jurisdiction table "1" dataElement "13"
field "Attribute"
        provision "Example Provision"]
    [ruleReference AttributeRule]
```

```
override attribute Type (0..1)
    [ruleReference empty]
```

```
override attribute ComplexType (0..1)
    [label for nestedAttribute "Test Label"]
    [regulatoryReference for nestedAttribute Jurisdiction table "2"
dataElement "55" field "Nested Attribute"
        provision "Example Provision"]
    [ruleReference for nestedAttribute NestedAttributeRule]
```

4.2 Jurisdiction-specific mappings

All new jurisdiction-specific mappings must be implemented using `CommonTransactionReport` and the appropriate `CriticalDataElement` types.

Do

- Add attributes to `CommonTransactionReport` when required by more than one jurisdiction
- Relax type constraints at the Common or CDE level if required, then reapply stricter constraints at the regime level
- Reuse existing `ruleReference`'s defined in Common or CDE namespaces

Do not

- Duplicate logic that already exists in `CommonTransactionReport` or `CriticalDataElement`.

When adding jurisdiction-specific rules for existing attributes:

Do

- Prefer existing common rule references.
- Make jurisdictional divergences explicit and easy to identify.

Do not

- Create new rule logic where a suitable common rule already exists.

Example

```
reporting rule JurisdictionSpecificRule from
TransactionReportInstruction: <"Cleared">
    filter IsAllowableAction
    then extract
        if common.CommonRule = value
        then extract jurisdiction
        else common.CommonRule
```

4.3 Validations

New validations must be assessed against existing validations across jurisdictions before introducing new logic.

Do

- Reuse existing common validations where applicable
- Promote validations to the common namespace when shared across jurisdictions

Do not

- Implement new validations from scratch before confirming there is no existing equivalent.

5. Adding a new jurisdiction

The introduction of a new jurisdiction must follow a consistent and repeatable process:

5.1 Analyse reportable attributes

Identify all reportable fields required for the jurisdiction-specific report.

5.2 Classify attributes

Determine whether each attribute is:

- Already present in `CommonTransactionReport`
- Present in Common but not reportable (override with `ruleReference empty`)
- Unique to the jurisdiction
- Shared with one or more existing jurisdictions (promote to Common)

5.3 Document common attributes

Record new or reused common attributes in DRR tracking artefacts (e.g. Airtable) to ensure consistency and visibility.

5.4 Compare mappings and rule references

Assess differences between jurisdiction-specific mappings and existing common rules, promoting logic where appropriate.

5.5 Compare validations

Reuse existing common validations where possible. If validations are shared across jurisdictions, promote them to the common layer.

6. Release notes

Release notes explain what changed in each DRR release. They help users understand updates quickly and track how the model evolves. You can contribute by:

- Writing clear summaries of changes.
- Describing updates to rules, mappings, or documentation.
- Showing before/after examples when something changes.
- Explaining why a change was made.

Release notes are written in Markdown (.md), a simple formatting language. If you know how to write # headings and - bullets, you're ready. If you're not sure, take a look at this **Markdown cheat sheet**.

If you're offering contributions via Rosetta, you can download this **DRR Release Notes Template**.

For additional background on governance and contribution processes you can also refer to the **CDM Development Guidelines**.

6.1 Tips for writing release notes

When you write release notes in Markdown, include:

- A short headline e.g. **# DRR EMIR Refit – Clearing threshold update**.
- A clear explanation of what changed.
- Before/after examples if relevant.

- A short justification (why the change was needed).
- Directions for reviewers (where to look in the development environment e.g. Rosetta).

More tips

Use:

- Correct grammar (e.g. complete sentences with full stops)
- Consistent terminology (e.g. if you define 'agreement specification details' do not refer later to 'agreement content')
- Bullets for lists
- Third-person terminology (e.g. 'the CDM model provides...', not 'we provide...')

Markdown basics you'll use:

- `#` for headings
- `-` for bullets
- `** **` for **bold** (use sparingly)
- `` ` ` `` back tick 'code quotes' for rule names, attributes, or functions

Example: Release note for EMIR refit

```
# **DRR EMIR refit - Rule model updates**
_Background_
The existing implementation for the EMIR Refit rules that are fully
or partially aligned with the CDE and CFTC jurisdictions has been
reviewed and updated to match the latest CFTC version using common
functions.
_What is being released?_
This second release fixes various issues identified in the EMIR
Refit rules listed below:
1. EMIR 2.1 - `UTI`:
  -    Replaced the filter based on the deprecated FpML's coding
scheme unique-transaction-identifier by the CDM `identifierType`.
  -    Removed the filter to the last UTI version.
2. EMIR 1.2 - `Report Submitting Entity ID`:
  - Changed the `SupervisoryBodyEnum` value to `ESMA`.
_Review directions_
In Rosetta, select the Textual View and search for the released
rules.
```

```
In Rosetta, open the reports tab, select the report `ESMA / EMIR
Refit` and the dataset `Credit` and review the expectations for the
field `2.1 - UTI` in the sample Credit-Swaption-Single-Name-ex01.
In Rosetta, open the reports tab, select the report `ESMA / EMIR
Refit` and the dataset `Rates` and review the expectations for the
field `1.2 - Report Submitting Entity ID` in the sample IR-Option-
Swaption-ex01-Bermuda.
```

Example: Code snippets

Code snippets should be preceded by the string: `.. code-block:: Language` (where the Language could be any of Haskell, Java, JSON, etc), followed by a line spacing before the snippet itself.

So a code snippet that reads:

```
.. code-block:: Haskell
 type Party:
   [metadata key]
   partyId PartyIdentifier (1..*)
   name string (0..1)
     [metadata scheme]
   businessUnit BusinessUnit (0..*)
   person NaturalPerson (0..*)
   personRole NaturalPersonRole (0..*)
   account Account (0..1)
   contactInformation ContactInformation (0..1)
```

Will be rendered as:

```
type Party:
  [metadata key]
  partyId PartyIdentifier (1..*)
  name string (0..1)
    [metadata scheme]
  businessUnit BusinessUnit (0..*)
  person NaturalPerson (0..*)
  personRole NaturalPersonRole (0..*)
```

```
  account Account (0..1)
  contactInformation ContactInformation (0..1)
```

Note: Code snippets that appear in the user documentation are being compared against actual CDM components and any mismatch will trigger an error in the build. This ensures that the user documentation is kept in sync with the model in production before any release.