RIDDL Quick-Start Guide for Software Architects

From zero to a validated architecture model in one session

Published: February 2026

Author: Reid Spencer, Founder & CEO, Ossum Inc.

Website: ossum.ai

Table of Contents

Welcome, Architect

This guide is designed for solutions architects and enterprise architects who want to quickly evaluate RIDDL. By the end of this document, you will understand what RIDDL is, how it fits into your architecture workflow, and how to create your first validated model.

No prior RIDDL experience is required. Familiarity with Domain-Driven Design concepts (bounded contexts, aggregates, entities, events) is helpful but not essential — this guide explains each concept as it introduces the corresponding RIDDL syntax.

Time Estimate

Reading this guide: 20 minutes

Creating your first model: 30–60 minutes

Total investment to evaluate RIDDL: Under 2 hours

What RIDDL Is (and Is Not)

RIDDL Is

- A specification language for describing distributed systems using DDD principles
- A compiler (riddlc) that validates your specification for completeness and consistency
- Technology-agnostic — the same spec can target Akka, Quarkus, Erlang, or other frameworks

RIDDL Is Not

- A programming language — you do not write application logic in RIDDL
- A diagramming tool — RIDDL models have formal semantics, not just visual layout
- An ORM or database schema language — RIDDL operates at the architecture level
- Tied to any specific cloud provider, framework, or runtime

RIDDL Has:

- A visual tool (Synapify) for graphically composing and exploring models
- An AI integration (MCP Server) that lets Claude (and other AI systems with MCP support) help build and refine models
- 187 pre-developed models covering every industry that the Bureau of Labor & Statistics covers.

Core Concepts in 5 Minutes

RIDDL has a hierarchy of constructs that map directly to DDD concepts. Here is the full hierarchy at a glance:

RIDDL Construct	DDD Equivalent	Purpose
Domain	Domain	Top-level business area (e.g., E-Commerce, Banking)
Context	Bounded Context	A cohesive set of capabilities with its own ubiquitous language
Entity	Aggregate / Entity	A stateful object with identity, modeled as a state machine
Command	Command	An imperative request to change state (can be refused)
Event	Domain Event	A fact that something happened (immutable)
Query	Query	A read-only request for information
Saga	Saga	Multi-step distributed transaction with compensating actions
Projector	Read Model	Transforms events into query-optimized views (CQRS)
Repository	Repository	Persistent storage abstraction
Adaptor	Anti-Corruption Layer	Translates between bounded context languages
Streamlet	Stream Processor	Stateless stream processing element
Epic	User Story / Journey	Captures user interactions that span contexts

Your First RIDDL Model: Step by Step

Step 1: Define the Domain

Every RIDDL model starts with a domain — the top-level business area you are modeling. Let us model a simplified order management system:

```
domain OrderManagement is {
  ??? // Placeholder for "TBD". This is where Contexts will go
}with {
  described as "Handles the complete order lifecycle
    from placement through fulfillment."
}
```

The domain declaration names the business area and provides a human-readable description. Everything inside belongs to this domain.

Step 2: Add Bounded Contexts

Within the domain, define bounded contexts — cohesive units of functionality with their own language:

```
domain OrderManagement is {

  context Ordering is {
    ??? // Entities, commands, events go here
  } with {
    described as "Handles order placement and lifecycle"
  }

  context Fulfillment is { ??? } with {
    described as "Manages picking, packing, and shipping"
  }

  context Billing is { ??? } with {
    described as "Handles invoicing and payment processing"
  }
}
```

Each context has clear ownership of specific capabilities. The Ordering context does not know how Fulfillment works internally, and vice versa.

Step 3: Define Types and Messages

Before defining entities, declare the types and messages that form the context's ubiquitous language:

```
context Ordering is {
  type OrderId is Id(Order)
  type Money is {
    amount: Decimal,
    currency: String(3)
  }
  type LineItem is {
    productId: String,
    quantity: Integer,
    unitPrice: Money
  }
  command PlaceOrder is {
    orderId: OrderId,
    customerId: String,
    items: many LineItem
  }
  event OrderPlaced is {
    orderId: OrderId,
    customerId: String,
    items: many LineItem,
    total: Money
  }
}
```

Types are strongly defined. Commands represent requests to do something (can be refused). Events represent facts that have occurred (immutable). This distinction is fundamental to reactive architecture.

Step 4: Model Entities as State Machines

Entities are the core of your model. Each entity has identity, state, and behavior defined through handlers:

```
entity Order is {
  state Active is {
    fields {
      orderId: OrderId,
      customerId: String,
      items: many LineItem,
      total: Money,
      status: String
    }
    handler OrderHandler is {
      on command PlaceOrder {
        prompt "validate items, calculate total"
        set field status to "Placed"
        send event OrderPlaced to outlet Events
      }
      on command CancelOrder {
        set field status to "Cancelled"
        send event OrderCancelled to outlet Events
      }
      on command ShipOrder {
        set field status to "Shipped"
        send event OrderShipped to outlet Events
      }
    }
  }
}
```

Every command handler explicitly declares what events it emits and what state changes it makes. The compiler verifies that all commands have handlers and all emitted events are defined.

Step 5: Add Cross-Context Communication

When Ordering needs to trigger Fulfillment, use an adaptor:

```
context Fulfillment is {
  adaptor OrderingAdapter from context Ordering {
    handler OrderEvents {
      on event OrderPlaced {
        send command CreateShipment to entity Shipment
      }
    }
  }
}
```

The adaptor makes the integration explicit: Fulfillment subscribes to Ordering's events and translates them into its own commands. Each context maintains its own language.

Step 6: Validate

Run the RIDDL compiler to validate your model:

```
reid@Ossum-MacBook-Pro riddl-models % riddlc validate government/public-safety/emergency-dispatch/emergency-dispatch.riddl
[style] government/public-safety/emergency-dispatch/types.riddl(75:3->5):
Enumerator identifier 'K9' is too short. The minimum length is 3:
  K9,
[style] government/public-safety/emergency-dispatch/types.riddl(75:3->5):
Enumerator identifier 'K9' is too short. The minimum length is 3:
  K9,
[missing] government/public-safety/emergency-dispatch/IncidentContext.riddl(9:3->32):
Schema in Repository 'IncidentRepository' at government/public-safety/emergency-dispatch/IncidentContext.riddl(142->171) should not be empty:
  repository IncidentRepository is {
[missing] government/public-safety/emergency-dispatch/IncidentContext.riddl(118:7->121:7):
OnMessageClause 'query FindIncidentById' should have statements:
      on query FindIncidentById {
        ???
      }
      on query SearchIncidents {
[warning] government/public-safety/emergency-dispatch/IncidentContext.riddl(118:7->32):
Processing for queries should result in sending a result:
      on query FindIncidentById {
[missing] government/public-safety/emergency-dispatch/IncidentContext.riddl(121:7->124:7):
OnMessageClause 'query SearchIncidents' should have statements:
      on query SearchIncidents {
        ???
      }
      on query FindActiveIncidents {
[warning] government/public-safety/emergency-dispatch/IncidentContext.riddl(121:7->31):
Processing for queries should result in sending a result:
      on query SearchIncidents {
[missing] government/public-safety/emergency-dispatch/IncidentContext.riddl(124:7->127:7):
OnMessageClause 'query FindActiveIncidents' should have statements:
      on query FindActiveIncidents {
        ???
      }
      on query FindIncidentsByUnit {
```

The warnings tell you exactly what needs attention. No errors means your model is structurally sound. Warnings indicate areas for further specification or diminished competency in simulation or execution. Errors (none shown) are semantic problems you need to fix.

Using Synapify Instead of Text

Everything shown above in text syntax can be done visually in Synapify. The workflow is:

- **Create:** Open Synapify at ossum.ai and create a new model. Drag-and-drop domains, contexts, entities, and other elements onto the canvas.
- **Connect:** Draw relationships between contexts (adaptors), define message flows, and specify entity state machines using the visual editor.
- **Validate:** Synapify runs the RIDDL compiler continuously. Errors and warnings appear in real time as you model, highlighted directly on the affected elements.
- **Export:** Export your model as RIDDL source, documentation, or diagrams at any time. The visual model and text source are always in sync.

For architects who prefer visual composition, Synapify eliminates the need to learn RIDDL syntax. For those who prefer text, Synapify provides a visualization layer over the RIDDL source you write directly.

AI-Assisted Modeling with MCP Server

The RIDDL MCP Server gives Claude deep knowledge of the RIDDL language and your specific model. This enables a conversational modeling workflow:

```
You: "I need a saga to coordinate the payment and
      fulfillment workflow. If payment fails, cancel
      the shipment. If shipment fails after payment,
      issue a refund."

Claude: [Generates a complete RIDDL saga definition
         with steps, compensating actions, and proper
         cross-context references — all validated]
```

The MCP Server is not generating generic advice. It is generating a valid RIDDL model that integrates with your existing model(s) and can be compiled and validated immediately.

Architecture Decision Checklist

As you build your first real RIDDL model, use this checklist to guide your design decisions:

Decision	Question to Answer	RIDDL Construct
Domain boundaries	What are the major business areas?	domain
Context boundaries	Where should language change?	context
Entity identification	What has a unique identity and lifecycle?	entity
Command design	What actions can be requested?	command
Event design	What facts need to be recorded?	event
State machine	What states can an entity be in?	state + handler
Integration points	Which contexts need to communicate?	adaptor
Read models	What views do users need?	projector
Distributed transactions	What multi-step processes span contexts?	saga
External systems	What third-party services are involved?	external context
Persistence	How is data stored?	repository
User journeys	What end-to-end flows exist?	epic

Common Patterns for Architects

Pattern: Event-Driven Integration

Contexts communicate through events, not direct calls. This ensures loose coupling and independent deployability. In RIDDL, every context defines its own “ubiquitous language” (a DDD concept). Potentially, then, an impedance mismatch may occur in cross-context interactions. That’s what adaptors are for, making integration points explicit and traceable.

Pattern: CQRS Separation

Use entities for the write/command side and projectors for the read/query side. The RIDDL model makes this separation visible: commands flow to entities, events flow to projectors. There is no ambiguity about which components handle writes versus reads.

Pattern: Saga Coordination

For distributed transactions spanning multiple contexts, define sagas with explicit steps and compensating actions. The compiler verifies that each saga step references existing entities and that compensating actions are defined for each forward step.

Pattern: External System Boundaries

Model third-party services as external contexts with defined interfaces. This documents the dependency explicitly, specifies the expected messages, and ensures that internal contexts do not leak their language across the boundary. Use the “external” option in the metadata, so you might have something like this:

```
context Fulfillment is {
  ??? // external
} with {
  option external // Let RIDDL processors know this is an external context.
  briefly "External fulfillment service."
  described as {
    | ## External Fulfillment Service
    | We plan to use Acme Fulfillment Services to fulfill our orders.
    | This context serves to capture the interaction with that Fulfillment service.
  }
}
```

Quick Reference: RIDDL Syntax Cheat Sheet

Element	Syntax	Notes
Domain	domain Name is { ... }	Top-level container
Context	context Name is { ... }	Bounded context with ubiquitous language
Entity	entity Name is { state ... handler ... }	Stateful aggregate with identity
Command	command Name is { field: Type, ... }	Request to change state
Event	event Name is { field: Type, ... }	Immutable fact
Query	query Name is { field: Type, ... }	Read-only request
Saga	saga Name is { step ... step ... }	Distributed transaction
Projector	projector Name is { repository ... handler ... }	Read model builder
Repository	repository Name is { ... }	Persistence schema
Adaptor	adaptor Name from context X { handler ... }	Cross-context translator
Type	type Name is { field: Type, ... }	Data structure
Epic	epic Name is { case ... }	User journey

Next Steps

You now have enough knowledge to create your first RIDDL model. Here is the recommended path forward:

- **1. Try Synapify:** Go to ossum.ai, sign up for the free tier, and create a model for a domain you know well. Start small — one domain, two or three contexts, a few entities.
- **2. Walk through the Reactive BBQ tutorial:** A comprehensive example that demonstrates every concept covered in this guide using the familiar domain of a BBQ restaurant.
- **3. Read the concepts documentation:** This section provides detailed explanations of each RIDDL construct with examples.
- **4. Join the community:** Visit the RIDDL GitHub repository to browse the source, open issues, comment on discussions, and connect with other users.

Resources

Synapify (visual modeling): https://ossum.ai

RIDDL documentation: https://ossum.tech/riddl

RIDDL compiler (open source): https://github.com/ossuminc/riddl

RIDDL models repository: https://github.com/ossuminc/riddl-models

Language reference: https://ossum.tech/riddl/references/language-reference.html

Cheat sheet: https://ossum.tech/riddl/references/cheat-sheet.html