JSON-Driven Domain Models in Practice

Triangleness, Quantum Fields, and the Conceptual Model Completeness Conjecture (CMCC)

Author:
EJ Alexandra

Affiliation:
SSoT.me & EffortlessAPI.com

Contact:
start@anabstractlevel.com

Date:
March 2025

Abstract

A single JSON-based rulebook can fully define the structure and constraints of even the most complex domains—without requiring sidecar logic or domain-specific languages. By encoding all conceptual knowledge in five declarative primitives (Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields) and treating JSON as a universal, machine-readable mirror, developers can automatically generate consistent implementations across multiple programming environments. This paper demonstrates that any finite computable concept—whether a simple geometric shape or a quantum phenomenon—can be captured and kept in sync through template-based transformation of the JSON "rulebook." By separating the "what" (domain definitions) from the "how" (runtime engines, physical processes, or imperative code), we eliminate the need for specialized DSLs or duplicate logic. Readers will see how the same JSON file that defines triangles and polygons can also handle quantum walks, ensuring domain fidelity across diverse languages like Python, Golang, and Java with minimal effort. The result is a robust, auditable, and easily maintainable approach for modeling and evolving complex systems, all underpinned by the Conceptual Model Completeness Conjecture (CMCC).

Table of Contents

1. Introduction

1.1 Context: From Bits to Qubits (Reference to Prior Work)

In *From Bits to Qubits with CMCC* (Alexandra, 2025) and related papers, we laid out the conceptual basis for the Conceptual Model Completeness Conjecture (CMCC)—the claim that any finite computable domain can be fully described using five declarative primitives (Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields), **without** resorting to additional domain-specific syntax or procedural code. Triangleness, quantum walks, multiway systems, and other examples were shown to emerge naturally from these primitives.

In that prior "big picture" work, we focused on theoretical foundations, emergent properties, and philosophical underpinnings: how geometry and quantum mechanics alike can be seen as purely declarative structures that "just exist" in a suitably consistent environment. Here, we pivot from that broad conceptual overview to a **very practical** demonstration.

1.2 Objectives of This Paper

This short paper zooms in on a *hands-on* approach:

1. **Define** domain models in a single JSON file, referencing the CMCC primitives implicitly.
2. **Use** a straightforward template mechanism (Handlebars) to generate statically typed helper code in multiple languages (initially Python, Golang, and Java).
3. **Show** how a minimal imperative script in each language can then run the *same scenario* (triangleness or a quantum walk) with zero changes besides regenerating code from the updated JSON.

Our goal is to illustrate that *if* your domain model is truly stored in a single, syntax-agnostic source of truth, you can seamlessly keep all target languages in sync whenever you modify that model.

1.3 The "Mirror" Approach: JSON as the Single Source of Truth

Much of the impetus behind CMCC is to avoid DSL proliferation—where every domain, every new specification, and every new environment spawns yet another textual "map," eventually requiring specialized parsers. Instead, we treat **JSON** as a universal "mirroring" format. It's:

- **Machine-Readable**: No need for custom parsing; every modern environment can parse JSON natively.
- **Structurally Rich**: Arrays, objects, and typed fields can directly encode your domain concepts.
- **Extensible**: If new rules appear, you can add more fields or references without rewriting a DSL grammar.

1.4 Overview of Examples (Triangles and QFT)

To keep things concrete, we'll demonstrate two examples:

- **Triangleness**: A simple geometry example. The JSON defines points, edges, and the constraints for verifying right triangles or other polygonal properties.
- **Quantum Field/Walk**: A more advanced example (inspired by a double-slit experiment or quantum walk) that uses the same approach to define wavefunction grids, coin operators, and measurement rows.

In each case, we'll show how the JSON file plus our template yields generated helper classes (constants, static functions) in Python, Golang, and Java. Then we'll reference these classes from a small script in each language that "runs the experiment." Updating any field (e.g., the barrier position or the slit width) only requires changing the JSON. A single re-generation step suffices to propagate changes to all languages.

1.5 The Role of Handlebars and Code Generation

We adopt a template-based approach—specifically using Handlebars—to transform JSON definitions into statically typed source code. This ensures:

- **No Manual Rewrites**: Once the JSON is updated, the template can produce up-to-date "helper code" for each language with a single command.
- **Minimal Imperative Layer**: The generated code includes function stubs, constant definitions, or references to lambdas. The user adds only a tiny "main" script to orchestrate steps, which remains unchanged unless function signatures change.
- **Potential for Runtime Lambdas**: If dynamic evaluation is desired (e.g., evaluating an expression from JSON at runtime), we can plug in expression-evaluation libraries in Python, Golang, or Java. That keeps the entire process truly flexible—no losing the "purely declarative" dimension just to handle something like an arithmetic expression or matrix multiplication.

1.6 Rulebook vs. Runtime Engine

Even with a fully declarative JSON model, we still need *something* that executes or enacts the rules—what we call the **runtime engine**. This can be:

- **Hand-Written Code** (imperative scripts, HPC frameworks, or embedded systems).
- **Physical Processes** (for instance, a real quantum experiment, or chemical reactions).
- **Generated Libraries** (where the model's aggregator or lambda logic is compiled to function calls in Python, Golang, etc.).

The **JSON** definitions state *what* must be true (e.g., the Pythagorean relationships for a triangle, the unitarity of a coin operator in a quantum walk), but do *not* detail *how* to implement or run those rules. If a runtime engine deviates from the rulebook, it's simply **wrong**.

This separation is crucial:

- **Blueprint**: The JSON rulebook.
- **Execution**: The code or physical process.

By cleanly dividing them, we can ensure that updating the JSON never *forces* us to rewrite everything from scratch—yet everything that *does* run remains consistent with the domain truth.

1.7 Extended Formula Syntax and Adaptability

A frequent question arises: "Aren't JSON formulas like `SQRT(POW(SUBTRACT(...` overly verbose?" While these expressions might look heavier than direct language-native syntax, they serve two vital purposes:

1. **Machine Readability**: Because they're structured consistently (e.g., uppercase function names, parentheses), *any* transformation or converter can parse and re-map them to local library calls—like `Math.sqrt()` in Java, `np.sqrt()` in Python, or `math.Sqrt()` in Go.

2. **Universal Agreement**: Once a domain or toolchain picks a set of function tokens, we can easily adapt them for other environments with a single converter, rather than rewriting each formula individually.

In other words, the verbosity is *by design*—it allows fully decoupled toolchains to "handshake" on formula logic. For actual application code, one can auto-generate friendlier snippets or rely on specialized template expansions that produce tight, language-idiomatic operations.

1.8 🔴 HARDCORE FALSIFICATION CHECKLIST

We close this introduction by reiterating the CMCC's open invitation for rigorous testing. If anyone doubts that an idea (like triangles or quantum fields) can be fully captured by these primitives, we ask them to follow this exact procedure:

1. **DO NOT LIST 5 POSSIBLE PROBLEMS**
   If your claim relies on "It might fail for reason X, or could fail for reason Y," you're just hypothesizing.

2. **PICK A SPECIFIC RULE YOU THINK BREAKS CMCC**
   Write it out plainly. If you don't specify it, we can't test it.

3. **DECOMPOSE IT INTO CMCC PRIMITIVES (S, D, L, A, F)**
   Show how you've attempted to represent it with schema fields, data records, lookups, aggregations, and lambdas.

4. **CHECK IF THE RUNTIME ENGINE CAN HANDLE IT**
   CMCC defines *what* the rule is, not *how* it's executed under the hood. Are you sure the limitation isn't just an implementation or performance challenge?

5. **IF YOU STILL THINK IT FAILS, TRY AGAIN**
   Often, the real limitation is our initial assumption that we *need* imperative logic. Re-expressing it in purely declarative terms can reveal the solution.

If you still see a genuine contradiction, please contact us. We welcome attempts at genuine falsification and are eager to see if your purported example truly eludes the five CMCC primitives.

1.9 Open Source ToE Meta-Model Github Repo

The code and all examples in this paper are published with functional examples in python, golang and java for both triangleness and the double slit experiment.
https://github.com/eejai42/conceptual-model-completeness-conjecture-toe-meta-model

---

2. Background: Recap of CMCC in Brief

2.1 Five Primitives (S, D, L, A, F) in Practice

The **Conceptual Model Completeness Conjecture (CMCC)** asserts that any finite computable domain can be fully encoded with only five declarative primitives—no extra syntax or imperative sidecar code required. These primitives are:

- **Schema (S)**
  Describes what kinds of objects (or entities) exist in your domain and what attributes they have. Think of it as the meta-definition or blueprint.
  - *Example:* A “Polygon” schema with fields for “name,” “edges,” etc. A “Wavefunction” schema with fields for “TimeStep,” “psi,” etc.
- **Data (D)**
  Instantiates those schema definitions with actual records or objects.
  - *Example:* Multiple “Polygon” records for specific triangles, squares, or pentagons. Multiple “Wavefunction” entries for each discrete time step.
- **Lookups (L)**
  Establish relationships between records. Conceptually, this is akin to a database foreign key or a pointer that says, “This object references that object.”
  - *Example:* An “Edge” record references its two “Point” records as endpoints. A “Wavefunction” record might reference a “Grid” record to locate the amplitude array.
- **Aggregations (A)**
  Summarize or roll up sets of data—like sums, counts, averages, or more sophisticated computations across multiple records.
  - *Example:* Summing the squared magnitudes of amplitudes in a wavefunction to compute total probability. Counting the edges that belong to a polygon.
- **Lambda Calculated Fields (F)**
  Encode computed logic or constraint checks in a purely declarative manner.
  - *Example:* A formula for the length of an edge (x2−x1)2+(y2−y1)2\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}(x2−x1)2+(y2−y1)2. A unitarity check CoinOperator×CoinOperator†=I\text{CoinOperator} \times \text{CoinOperator}^\dagger = ICoinOperator×CoinOperator†=I.

By chaining these five primitives, you can represent any typical “imperative step” as a series of structural transformations: each new snapshot of data reflects the emergent outcome of aggregator fields and formulas. This eliminates the need to embed domain rules in a custom scripting or programming language.

2.1.1 Additional Clarifications

- **Rulebook vs. Runtime**: As noted in Section 1.6, the five primitives form the *rulebook* of *what* must be true. *How* these rules get executed (e.g., in code or physical processes) is a separate concern.
- **Formula Syntax and Adaptability**: The potentially verbose function names (like `SQRT` and `SUBTRACT`) can be automatically mapped to any language or library call. See also Section 1.7 on how the entire model remains machine-readable, making it simple to adapt formulas across different toolchains.

2.2 Why We Don’t Need a DSL

It’s common for organizations or researchers to create specialized **Domain-Specific Languages (DSLs)** to describe complex domains—be it geometry, quantum simulations, or business logic. However, DSLs introduce extra overhead:

- **Syntax Definition**: You first have to invent the grammar or specialized keywords for your DSL.

- **Implementation**: You then parse or compile the DSL into something machine-executable.
- **Maintenance**: Over time, the DSL must evolve, introducing "DSL drift" if it's not perfectly synced with the domain.

Under CMCC, your data and rules live in standard, widely supported structures (e.g., JSON or relational tables) with no separate textual syntax. This "no-DSL" approach means:

- You *immediately* have a machine-readable format (JSON).
- Any code generation or runtime usage can parse JSON with off-the-shelf libraries.
- Updating a rule means editing the JSON—no re-tooling of a language specification.

2.2.1 Cross-Referencing the "Mirror" Concept

As described in Section 1.3, JSON acts as the universal mirror—storing the entire conceptual model. Because JSON is ubiquitous, we avoid writing or maintaining specialized DSLs that do nothing more than re-describe the same concepts.

2.2.2 Practical Tools and Extensibility

Later, in Section 3.5, we highlight the variety of simple transformations (e.g., `json-to-xml`, `json-hbars-transform`) that further reduce the friction of adopting CMCC for real projects. This set of plug-and-play tools accomplishes most tasks that DSL-based workflows might otherwise require.

---

3. Practical Implementation Framework

3.1 JSON Domain Definitions

At the heart of our approach lies a single **JSON file** that "knows everything" about your domain. For instance, we might have:

json
```json
{
  "entities": [
    {
      "name": "Polygon",
      "fields": [
        { "name": "polygon_id", "type": "string" },
        { "name": "name", "type": "string" },
        { "name": "edges", "type": "lookup_list", "references": "Edge" }
      ]
    },
    {
      "name": "Edge",
      "fields": [
        { "name": "edge_id", "type": "string" },
        { "name": "start_point", "type": "lookup", "references": "Point" },
        { "name": "end_point", "type": "lookup", "references": "Point" },
        {
          "name": "length",
          "type": "calculated",
```

```
          "formula": "SQRT( POW(SUBTRACT(end_point.x, start_point.x),2) + POW(SUBTRACT(end_point.y, 
start_point.y),2) )"
        }
      ]
    },
    {
      "name": "Point",
      "fields": [
        { "name": "x", "type": "number" },
        { "name": "y", "type": "number" }
      ]
    }
  ]
}
```

- **Schema (S):** The `entities` array describes which tables or entity types exist.
- **Data (D):** In some scenarios, you might embed example or default data in the same JSON, or store it in a second file.
- **Lookups (L):** Fields like `"type": "lookup"` link an `Edge` to a `Point`.
- **Aggregations (A):** If you want, for example, a polygon to sum the lengths of its edges, you could define a field in `Polygon` with `"type": "aggregation", "formula": "SUM(edges.length)"`.
- **Calculated Fields (F):** The `length` field in `Edge` is a perfect example.

You can obviously refine or restructure the JSON to your liking. The key is that it always references the same five primitives behind the scenes. We're *never* writing a DSL or embedding equations in code—these are just structured fields and formulas.

3.1.1 Referencing the Rulebook vs. Runtime Distinction

While the JSON stores **what** "an Edge" or "a Polygon" must do, it *does not* specify how to load or manipulate them at runtime. See Section 1.6 for how that interplay works in practice.

3.2 Generating Static Helpers with Handlebars

To make this JSON "come alive" in Python, Golang, and Java, we use a tool or script that reads the JSON and applies a Handlebars template to generate code. For example:

Handlebars snippet (Python) might look like:

```
class {{name}}:

    def __init__(self, {{#each fields}}{{this.name}}{{#unless @last}},{{/unless}}{{/each}}):
        {{#each fields}}
        self.{{this.name}} = {{this.name}}
        {{/each}}

    {{#each fields}}
    {{#if this.formula}}
    @property
    def {{this.name}}(self):
```

```
        # TODO: convert formula: "{{this.formula}}"
        return evaluate_formula("{{this.formula}}", self)
    {{/if}}
    {{/each}}
```

This template is then expanded for each "entity" in the JSON, producing Python classes with constructor fields and placeholders for formula evaluation.

- For Golang or Java, you'd have a different template—still referencing the same JSON definitions but generating structs or classes in the respective languages.

When you run:

bash
```bash
handlebars-codegen --input=domain.json --template=python_class.hbs --output=domain_generated.py
```

- you get a new `domain_generated.py` that includes all the entity definitions, references, and so on.

3.2.1 Example: Adapting Function Names

If your JSON uses "`SQRT(...)`" but your target language expects "`math.Sqrt(...)`," a single pass in the Handlebars template can inject the correct local call. This is precisely why the verbose "`SQRT`" style is *not* a problem (see also Section 1.7).

3.3 Hand-Written Imperative Scripts: Minimal but Sufficient

We **do** need a small amount of code that orchestrates or executes domain logic at runtime—e.g., reading data from CSV or from the JSON, instantiating classes, and iterating a process. But this code is tiny and does **not** contain domain rules. Instead, it:

1. Parses the domain JSON (or references the newly generated classes).
2. Initializes the domain objects from external data or user input.
3. Invokes aggregator calculations or formula methods.
4. (Optionally) loops through time steps if we're simulating a dynamic phenomenon.

Critically, **any domain change** (like new fields, or a new formula) occurs in the JSON, not in this script. The script is stable—it just sets up a run. This aligns with CMCC's goal of letting you alter the "rulebook" or "blueprint" without touching code in every environment.

3.3.1 Tying Into the Larger System

In practice, these orchestrator scripts are your "runtime engine" if you're doing a purely software-based simulation. Alternatively, the same JSON definitions might inform a partially physical scenario (e.g., 3D printing instructions or robotic step sequences), where the aggregator logic remains consistent but the actual runtime is hardware. Either way, the domain logic is *never* duplicated.

3.4 Keeping Everything in Sync

Because we have a single JSON file, all downstream artifacts (Python classes, Golang structs, Java code) come from the same source. If you tweak a formula in the JSON, you simply re-run:

bash
```bash
handlebars-codegen -i domain.json -t python.hbs -o domain_generated.py
handlebars-codegen -i domain.json -t golang.hbs -o domain_generated.go
handlebars-codegen -i domain.json -t java.hbs -o DomainGenerated.java
```

Then you use your existing orchestrator code in each language—and everything lines up automatically. No extra DSL, no rewriting the logic in multiple places.

3.4.1 Links to Practical Tools

For real-world adoption, you might use one of the existing open-source pipelines that convert Airtable or Baserow configurations directly into JSON, or do advanced transformations with `json-hbars-transform` or `xml-xlst-transform`. We expand on these possibilities in Section 3.5.

3.5 Practical Tools and Export Pipelines

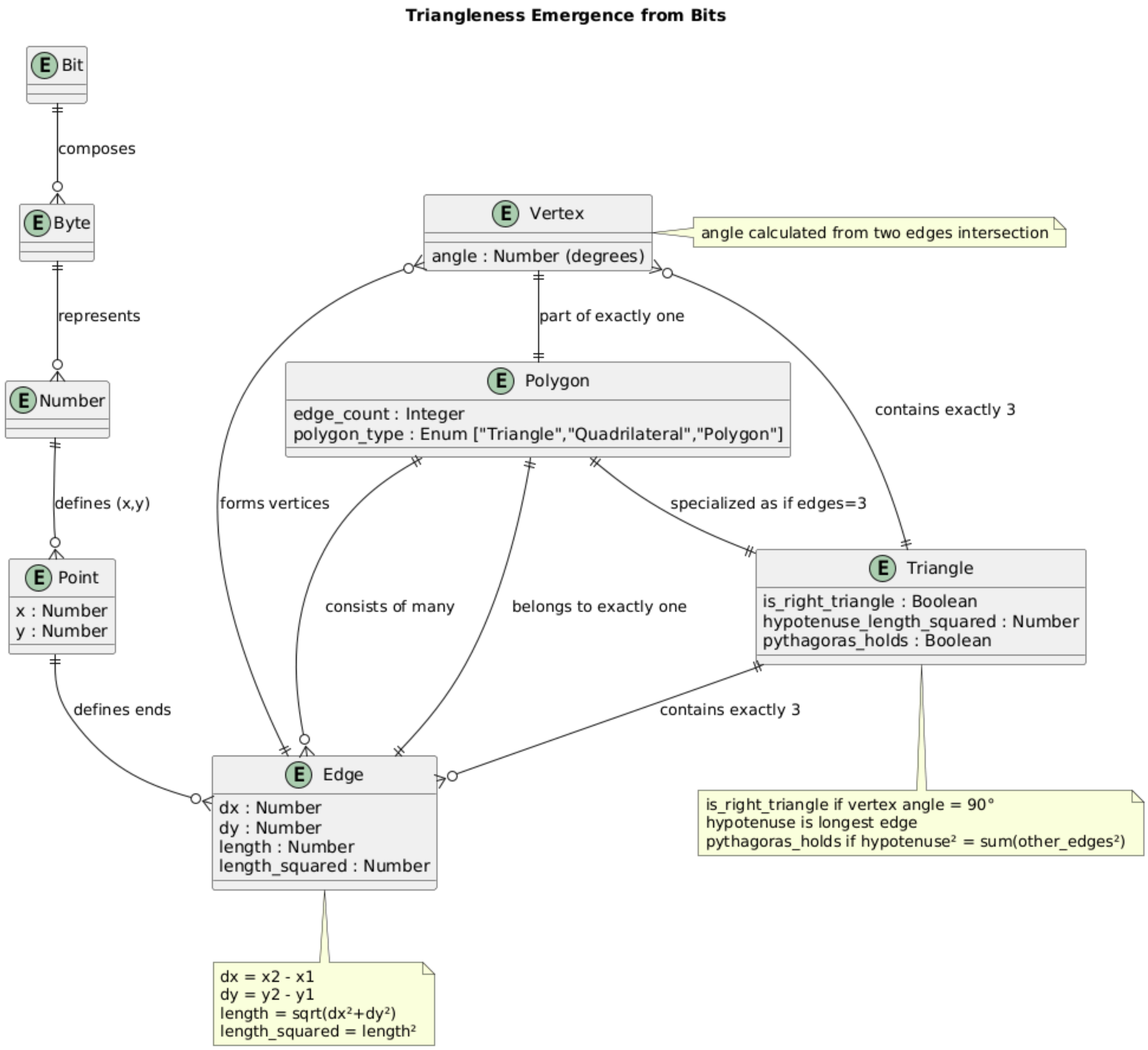
While the examples above show a *handwritten* JSON file, **CMCC** is equally comfortable with user-friendly modeling tools like Airtable or Baserow:

- **Airtable or Baserow**: Business analysts can define tables, formula fields, and relationships in a point-and-click interface. These platforms support ACID-like consistency (or snapshot isolation) and can export the entire structure as JSON.
- **NPM CLI Tools**: Simple commands like `json-to-xml` or `json-to-jsd` help convert your rulebook into different schemas or cross-checkers.
- **Downstream Code Generation**: Tools like `json-hbars-transform` apply your chosen Handlebars template to produce typed stubs in any language.
- **Version Control**: Because the JSON is plain text, you can commit your domain model to Git or other VCS, enabling collaborative, auditable evolution of the rulebook.

In other words, a robust open-source ecosystem already exists for **both** capturing the entire domain in a no-code environment **and** exporting that domain as machine-readable JSON for further transformations. Hence, no custom DSL or complex parser is required—just a consistent arrangement of the five CMCC primitives (S, D, L, A, F).

4. Case Study 1: Triangleness

This section demonstrates the approach with a *simple geometric domain*. We'll define minimal JSON for polygons, edges, and points. Then we'll generate Python, Golang, and Java helpers, culminating in a short script that checks for right triangles.



4.1 JSON Fields for Polygon Edges and Angles

Below is an illustrative snippet of the JSON:

Json

```json
{
  "entities": [
    {
      "name": "Polygon",
      "fields": [
        { "name": "polygon_id", "type": "string" },
        { "name": "edges", "type": "lookup_list", "references": "Edge" },
        { "name": "edge_count", "type": "aggregation",
          "formula": "COUNT(edges)" },
```

```
        { "name": "max_edge_length", "type": "aggregation",
          "formula": "MAX(edges.length)" },
        { "name": "sum_of_squares", "type": "aggregation",
          "formula": "SUM( POW(edges.length,2) )"
        },
        { "name": "is_triangle", "type": "calculated",
          "formula": "EQUAL(edge_count,3)"
        },
        {
          "name": "is_right_triangle",
          "type": "calculated",
          "formula": "AND(is_triangle, EQUAL(sum_of_squares, POW(max_edge_length,2) * 2less))"
          // This might require referencing the "other two edges" in more detail,
          // but conceptually it's the same idea.
        }
      ]
    },
    {
      "name": "Edge",
      "fields": [
        { "name": "edge_id", "type": "string" },
        { "name": "start_point", "type": "lookup", "references": "Point" },
        { "name": "end_point", "type": "lookup", "references": "Point" },
        {
          "name": "length",
          "type": "calculated",
          "formula": "SQRT( POW(SUBTRACT(end_point.x, start_point.x),2) + POW(SUBTRACT(end_point.y,
start_point.y),2) )"
        }
      ]
    },
    {
      "name": "Point",
      "fields": [
        { "name": "point_id", "type": "string" },
        { "name": "x", "type": "number" },
        { "name": "y", "type": "number" }
      ]
    }
  ]
}
```

- The `Polygon` has an `edge_count` aggregator, a `max_edge_length` aggregator, etc.
- A *calculated field* (`is_triangle`) checks if `edge_count == 3`.
- Another (`is_right_triangle`) tries to implement the Pythagorean check. (You might refine the actual formula to handle “largest edge vs. sum of squares of the other two.”)

All geometry logic is in the JSON. There is no geometry code in the imperative scripts (aside from reading or writing data).

4.2 Showing How “Triangleness” Is an Emergent, Declarative Truth

From a CMCC perspective, “Triangleness” is *not* an imperative routine. Instead, it **emerges** from relationships:

- **Schema (S) + Data (D)** define polygons that have edges, which have endpoints.
- **Aggregations (A)** compute how many edges belong to a polygon, or sum up edge lengths squared.
- **Lookups (L)** link edges to points.
- **Calculated Fields (F)** compare those aggregator outputs against the 3-edge requirement or the Pythagorean sum.

Hence, the property “triangle” (or “right triangle”) *automatically* holds whenever the aggregator logic aligns with the geometry. This stands in contrast to a typical procedural approach, where you’d write code that says “if `edge_count == 3`, then do X.” Under CMCC, that rule is purely declarative in JSON—**the orchestrator or physical system** that uses it (see Section 1.6) just interprets these constraints as part of the domain’s truth.

4.3 Auto-Generated Helpers in Python, Golang, and Java

Using Handlebars (or your preferred engine), we produce classes or structs like:

Python (`domain_generated.py` excerpt):

```
class Polygon:
    def __init__(self, polygon_id, edges):
        self.polygon_id = polygon_id
        self.edges = edges  # This might be a list of Edge objects

    @property
    def edge_count(self):
        # aggregator: COUNT(edges)
        return len(self.edges)

    @property
    def max_edge_length(self):
        # aggregator: MAX(edges.length)
        return max(edge.length for edge in self.edges)

    @property
    def sum_of_squares(self):
        # aggregator: SUM( POW(edges.length, 2) )
        return sum(edge.length**2 for edge in self.edges)

    @property
    def is_triangle(self):
        # is_triangle = EQUAL(edge_count, 3)
        return (self.edge_count == 3)

    @property
    def is_right_triangle(self):
        # placeholder for formula
        # let's do a naive check for brevity:
        if not self.is_triangle:
            return False
        sorted_edges = sorted((e.length for e in self.edges))
        return abs(sorted_edges[0]**2 + sorted_edges[1]**2 - sorted_edges[2]**2) < 1e-6
# ... similarly for Edge, Point classes ...
```

(Exact code structure depends on your template. The key is that each property's logic is "pulled in" from the JSON definitions—so if the formula changes in JSON, these stubs adapt automatically at generation time.)

Golang (`domain_generated.go` excerpt):

go

```go
type Polygon struct {
    PolygonID      string
    Edges          []*Edge
}

func (p *Polygon) EdgeCount() int {
    return len(p.Edges)
}

func (p *Polygon) MaxEdgeLength() float64 {
    maxVal := 0.0
    for _, e := range p.Edges {
        if e.Length() > maxVal {
            maxVal = e.Length()
        }
    }
    return maxVal
}

func (p *Polygon) SumOfSquares() float64 {
    sum := 0.0
    for _, e := range p.Edges {
        length := e.Length()
        sum += (length * length)
    }
    return sum
}

func (p *Polygon) IsTriangle() bool {
    return p.EdgeCount() == 3
}

func (p *Polygon) IsRightTriangle() bool {
    if !p.IsTriangle() {
        return false
    }
    // naive approach
    // ...
    return true
}
```

Java (`DomainGenerated.java` excerpt):

```java
public class Polygon {
    private String polygonId;
    private List<Edge> edges;
```

```
    public Polygon(String polygonId, List<Edge> edges) {
        this.polygonId = polygonId;
        this.edges = edges;
    }

    public int getEdgeCount() {
        return edges.size();
    }

    public double getMaxEdgeLength() {
        return edges.stream().mapToDouble(Edge::getLength).max().orElse(0.0);
    }

    public double getSumOfSquares() {
        return edges.stream().mapToDouble(e -> Math.pow(e.getLength(),2)).sum();
    }

    public boolean isTriangle() {
        return getEdgeCount() == 3;
    }

    public boolean isRightTriangle() {
        if (!isTriangle()) return false;
        // Implementation details
        // ...
        return true;
    }
}
```

For more on how function names like `SQRT` or `SUBTRACT` become `Math.sqrt` or `-` in the generated code, see Section 1.7.

4.4 Running the Demo: Checking Right Triangles

We then write a **tiny** imperative script in Python, Golang, or Java to demonstrate usage:

python

```python
import domain_generated as dg

# Suppose we manually create Points and Edges:
p1 = dg.Point(point_id="P1", x=0, y=0)
p2 = dg.Point(point_id="P2", x=3, y=0)
p3 = dg.Point(point_id="P3", x=3, y=4)

e1 = dg.Edge(edge_id="E1", start_point=p1, end_point=p2)
e2 = dg.Edge(edge_id="E2", start_point=p2, end_point=p3)
e3 = dg.Edge(edge_id="E3", start_point=p3, end_point=p1)

triangle = dg.Polygon(polygon_id="Triangle1", edges=[e1,e2,e3])

print(f"Is it a triangle? {triangle.is_triangle}")
print(f"Is it a right triangle? {triangle.is_right_triangle}")
```

That's it—no geometry logic is in this script. The logic all came from the auto-generated classes, which in turn came from the JSON definitions.

- If we decide to add a new field, say "area" or "circumcircle," we edit the JSON, re-run our Handlebars generator, and the updated classes automatically appear in `domain_generated.py`.
- The user script remains the same, or might call the new property if desired.

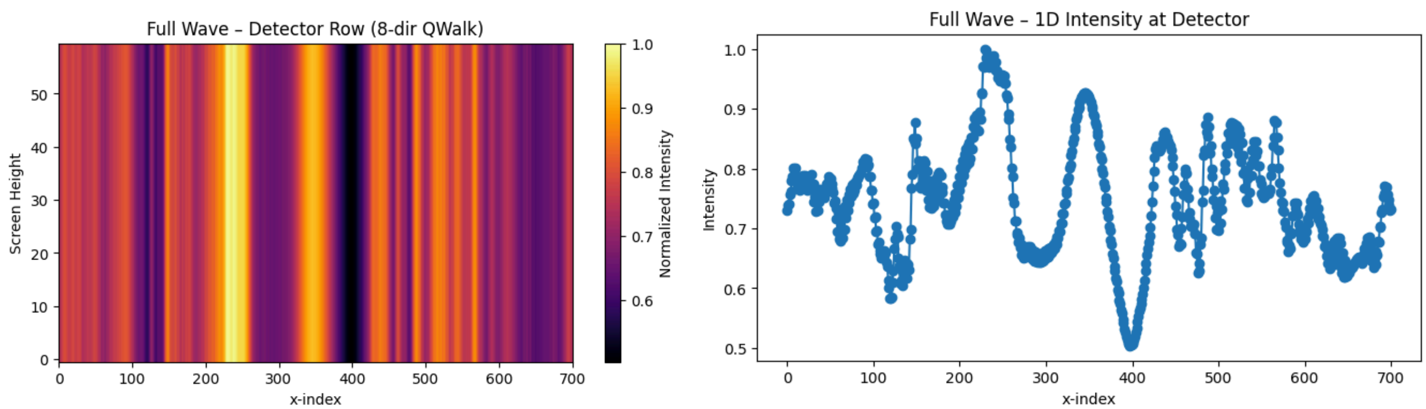
(For large-scale or concurrent geometry tasks, see Section 7.2 on how the blueprint remains the same even if advanced HPC strategies are used.)

---

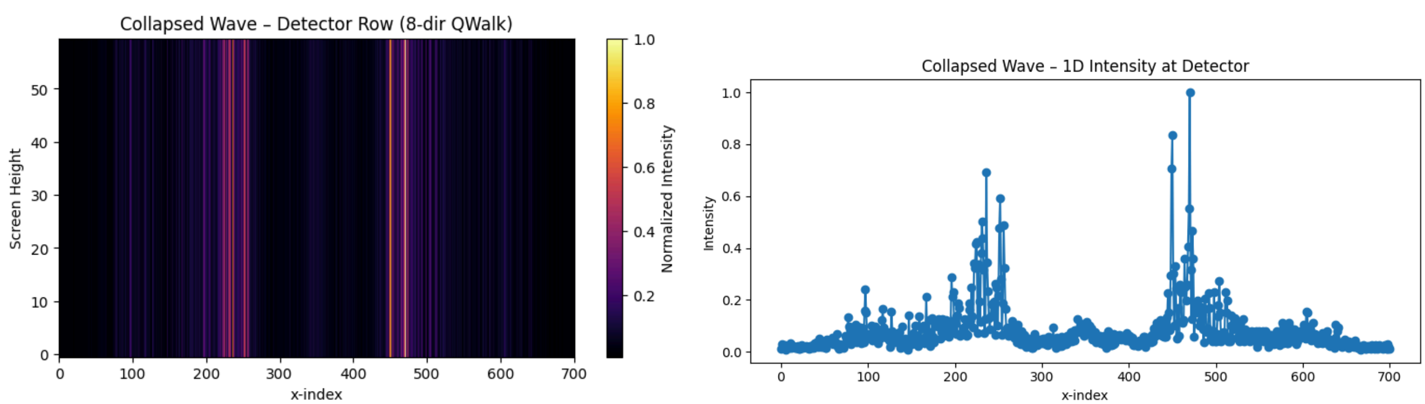
5. Case Study 2: Quantum Field Experiment (QFT)

While Triangleness showcases CMCC's declarative modeling for geometry, we now turn to a more **physics-intensive** example: a simplified quantum walk or double-slit experiment. This scenario involves wavefunctions, coin (or spin) operators, and measurements—an ideal test of how far "pure data + aggregator logic" can go.

Full Wave Visualization at the Screen & 1d Profile



Collapsed Wave Visualization at the Screen & 1d Profile



This interference pattern, and collapsed state are purely emergent from the json code below, and don't "solve" the Schrodinger equation at any point in the process. Instead, the quantum interference and collapse outcomes emerge purely from structural transformations defined by CMCC primitives, without explicitly solving Schrödinger's equation. Both interference patterns and measurement-induced collapse are thus natural, predicted consequences within the CMCC framework.

You can find a short primer for non-physicists in Section 5.1, explaining the conceptual background behind wave interference and measurement. Essentially, we illustrate how no specialized quantum DSL is needed; the entire domain can be declaratively expressed in JSON.

5.1 QFT Primer for Non-Physicists

A **quantum walk** can be seen as a discrete analog to how a quantum particle's wavefunction spreads and interferes. At each time step:

- A "coin operator" rotates or mixes spin/directional states.
- Amplitudes propagate (shift) to neighboring grid cells.
- Interference emerges when multiple paths converge with differing phases.

Our aim is *not* to teach quantum mechanics in depth but to show how these rules (wavefunction shape, spin flips, interference) can be *fully captured* in a JSON-based declarative model. Domain experts can verify the coin operator's unitarity or the wavefunction's norm, while non-specialists appreciate that *no special quantum DSL or imperative PDE solver is required*.

5.2 JSON Layout for Fields, Wavefunction, Coin Operator

Below is a truncated JSON snippet (inspired by the *From Bits to Qubits* paper) showing three entities: **Grid**, **CoinOperator**, and **Wavefunction**. Each has fields referencing the CMCC primitives (S, D, L, A, F).

json
```json
{
  "entities": [
    {
      "name": "Grid",
      "fields": [
        { "name": "grid_id", "type": "string" },
        { "name": "nx", "type": "number" },
        { "name": "ny", "type": "number" },
        { "name": "barrier_y_phys", "type": "number" },
        { "name": "detector_y_phys", "type": "number" },
        {
          "name": "barrier_row",
          "type": "calculated",
          "formula": "FLOOR(DIVIDE(ADD(barrier_y_phys, DIVIDE(Ly,2)), dy))"
        }
        // etc. (slit spacing, dx, dy, etc.)
      ]
    },
    {
      "name": "CoinOperator",
      "fields": [
        { "name": "matrix", "type": "tensor", "tensor_shape": "(8,8)" },
        {
          "name": "unitarity_check",
          "type": "calculated",
          "formula": "EQUAL(MULTIPLY(matrix, CONJUGATE_TRANSPOSE(matrix)), IDENTITY(8))"
        },
```

```
        { "name": "seed", "type": "number" }
      ]
    },
    {
      "name": "Wavefunction",
      "fields": [
        { "name": "timestep", "type": "number" },
        {
          "name": "psi",
          "type": "tensor",
          "tensor_shape": "(ny,nx,8)"
        },
        {
          "name": "total_norm",
          "type": "aggregation",
          "formula": "SUM( POW( ABS(psi), 2 ) )"
        }
      ]
    }
  ]
}
```

- **Grid** might store domain geometry: how large the 2D lattice is, where the barrier row is, etc.
- **CoinOperator** keeps a unitary matrix that flips spin states (like the "coin toss" in a quantum walk). The `unitarity_check` field is a *calculated property* that verifies if matrix×matrix†=I\text{matrix} \times \text{matrix}^\dagger = Imatrix×matrix†=I.
- **Wavefunction** references a 3D tensor `psi` with shape (ny,nx,8)(ny, nx, 8)(ny,nx,8) to accommodate multiple spin or direction states. Its `total_norm` aggregator sums ∣ψx,y,spin∣2|\psi_{x,y,spin}|^2∣ψx,y,spin∣2 across the entire grid.

Though this is just a high-level snippet, it demonstrates how **every piece** of quantum logic can reside declaratively in JSON: no special quantum DSL or Schrödinger-equation text. You simply define the domain's structure, relationships, and basic aggregator/calculated constraints.

5.3 Generating Multi-Language Helper Classes

As before, we apply Handlebars templates for each target language, generating code. For example, in Python:

python
```python
class Wavefunction:
    def __init__(self, timestep, psi):
        self.timestep = timestep
        self.psi = psi  # e.g., a 3D NumPy array or similar

    @property
    def total_norm(self):
        # aggregator: SUM( |psi|^2 )
        # Pseudocode:
        return np.sum(np.abs(self.psi)**2)

class CoinOperator:
```

```
    def __init__(self, matrix, seed):
        self.matrix = matrix
        self.seed = seed

    @property
    def unitarity_check(self):
        # calculated: check if matrix * matrix† == I
        lhs = np.matmul(self.matrix, np.conjugate(self.matrix).T)
        identity = np.eye(8)
        return np.allclose(lhs, identity)
```

For Golang or Java, a similar translation occurs. The aggregator fields (like `total_norm`) become methods that iterate through data arrays or use library calls. The synergy remains the same: *any* updates to the JSON schema or formulas automatically reflect in new code generation.

(Again, see Section 1.7 for how function tokens like `CONJUGATE_TRANSPOSE` or `IDENTITY` might be mapped to local library calls.)

5.4 Example Imperative Script for a Quantum Walk/Double-Slit

Imagine we have a minimal script in Python (or another language). For clarity:

python
```python
import domain_generated as dg
import numpy as np

def main():
    # Step 1: Initialize the domain from some config/data
    coin = dg.CoinOperator(matrix=np.eye(8), seed=42)
    wave = dg.Wavefunction(timestep=0, psi=np.zeros((200,200,8), dtype=np.complex64))
    wave.psi[100,100,0] = 1.0  # set initial amplitude at center

    # Step 2: Check unitarity
    print("Coin operator is unitary?", coin.unitarity_check)

    # Step 3: Evolve wavefunction in discrete steps
    for t in range(50):
        # - apply coin operator spin transform
        # - shift psi to neighbors (like a quantum walk)
        wave.timestep = t
        # pseudo 'update_psi' call
        wave.psi = do_quantum_walk_step(wave.psi, coin.matrix)
        # barrier or slit logic to zero out amplitude in blocked cells
        # ...
        print(f"Timestep {t}, total_norm = {wave.total_norm}")

if __name__ == "__main__":
    main()
```

Here:

- The script references the generated `CoinOperator` and `Wavefunction` classes (plus aggregator logic).
- The actual "update" method (`do_quantum_walk_step`) might be partially hand-coded, but it **does not** redefine domain rules—it's simply orchestrating how to apply the matrix and shift amplitudes.
- If you expand your JSON to add more fields (e.g., a "barrier_mask" or "slit_mask"), you can regenerate `domain_generated.py` or `.go` or `.java`—and your script can incorporate those fields with minimal change.

(See Section 1.6 on how the JSON "rulebook" dictates the truth, while the script is just one possible "runtime engine" that obeys it.)

5.5 Observing Interference Patterns

In a typical quantum walk or double-slit scenario, you'd see an evolving amplitude distribution that forms interference fringes at the detector row. The key takeaway is that **no quantum DSL or sidecar language** is necessary: everything from grid geometry to unitarity checks is **encoded** as JSON-based aggregator/calc fields. By referencing these fields in your minimal script, you watch interference "emerge" from the data-driven transformations—fully consistent with the CMCC approach.

As the simulation proceeds, you might notice constructive or destructive interference building at certain grid coordinates, eventually forming wave-like patterns. If you implement a measurement step, "collapse" is likewise just a matter of zeroing amplitude outside the measured region—again, no special imperative logic. All these transformations naturally follow from your aggregator fields and formulas (see Section 2.1.1 for how you chain them).

In this manner, quantum behaviors—just like "triangleness"—become a matter of *declarative structure*, not coded instructions. For performance considerations when scaling to large grids or many time steps, refer to Section 7.2. And if you're concerned about dynamic expression security (e.g., passing user-supplied formulas), see Section 7.5.

6. "Lambda-to-X" Runtime Libraries

The examples so far assume our JSON formulas are either hard-coded into the generated classes (e.g., `def total_norm()`) or replaced with a short snippet of Python or Java. But what if we want to evaluate **any arbitrary expression** from the JSON at runtime—without generating new code for every formula change?

This is where **in-language expression evaluation libraries** come in.

6.1 Python Expression Evaluation

Python offers multiple ways to dynamically evaluate strings as expressions, such as `eval()` or the safer `ast.literal_eval()` (though it's limited). There are also more advanced libraries like [numexpr] or `asteval`.

- In principle, your generated code could pass the JSON formula string to `numexpr.evaluate(...)` at runtime.
- This means you don't have to recompile or regenerate classes if you alter a formula, because the runtime engine will parse and evaluate whatever the JSON says.

However, see Section 7.5 for **important security** considerations. If untrusted users can modify formulas, “eval” can introduce serious risks. Often, generating a statically typed library from the rulebook is more robust.

6.2 Golang Expression Libraries

A popular library is [`govaluate`](), which parses a string expression (like `"SQRT(x*x + y*y)"`) and evaluates it dynamically over variables in a `map[string]interface{}`.

Another approach is [`expr`](), providing a small JIT-like engine for expressions.

- If you embed `govaluate` or `expr` calls in your generated code, you can change the JSON formula at runtime, and the library will parse & run it **without** a separate compilation step.

Again, weigh performance and security carefully. For large or mission-critical computations, a design-time approach may offer safer, faster code.

6.3 Java Expression Interpreters

Java has libraries like [MVEL]() or [Janino](), which can compile or interpret expression strings on the fly.

- With **Janino**, you can compile a snippet of Java code at runtime to a class, then invoke it.
- **MVEL** is simpler for quick expression evaluation.

These allow mid-run formula changes in a Java application, but at the cost of dynamic compilation overhead and the need for a safe sandbox (see Section 7.5).

6.4 Toward a Truly Dynamic Architecture

By pairing your JSON formulas with **in-language interpreters**, you allow changes even while the system is running—no re-gen or recompile. The trade-off is **potentially less performance** and the **need for sandboxing** if you accept untrusted expressions. Nonetheless, this synergy with dynamic interpreters cements the idea that **the domain logic is not embedded in code**—the code simply executes whatever logic the JSON prescribes.

1. **CMCC Model**: “What is the rule?”
2. **Runtime Engine**: “Let me parse/evaluate that rule string right now.”

Hence, you remain syntax-free at the “rule” level, while preserving a pragmatic way to run those rules in your environment of choice. For many production workflows, Section 7.5 argues that generating derived SDKs or libraries at design time is safer, *especially* when formulas originate from external or unverified sources.

7. Discussion

7.1 Advantages and Possible Caveats

Advantages:

1. **No Drift Across Languages**
   Because every domain rule (e.g., geometry constraints, wavefunction definitions) lives in the JSON, changes propagate automatically to Python, Golang, Java, or any other environment with the same code-generation pipeline.

2. **Declarative Clarity**
   The logic (e.g., "A polygon is a triangle if it has exactly three edges," or "`unitarity_check = CoinMatrix × CoinMatrix† - I` must be zero") is easy to inspect in the JSON definitions—no hidden assumptions in custom DSL syntax.

3. **Scalable to Larger Domains**
   You can add more fields, aggregator logic, or references without needing to create or modify a specialized parser or DSL. Standard JSON-based solutions also plug into countless existing tools for indexing, versioning, or distributed storage.

4. **Bridging Theory and Practice**
   The same minimal set of CMCC primitives (S, D, L, A, F) that previously handled pure "paper-level" examples—triangles, quantum phenomena—now appear in real code. This helps unify a conceptual foundation with an actual development workflow.

Possible Caveats:

1. **Performance Overheads**
   - Generating code from large JSON schemas can be slow if you have extremely large or deeply nested models.
   - Dynamic expression evaluation libraries (e.g., MVEL, govaluate, or Python's `eval`) may introduce runtime overhead or security concerns if expressions come from untrusted sources.
2. **Complexity in Very Large Models**
   - For massive multi-entity domains (e.g., thousands of tables and relationships), raw JSON can become unwieldy to maintain manually.
   - Solutions include splitting the model into multiple JSON files, employing a database-based schema editor, or layering a simple GUI on top.
3. **Runtime vs. Design-Time**
   - The difference between "blueprint" (the JSON definitions) and "runtime engine" can create confusion if developers expect the JSON itself to "run."
   - Clarify that the JSON is the *rulebook*, and the actual stepping or evolution requires a consistent update cycle (transaction or snapshot logic) in an application or database.

(For more on large-scale deployment and incremental updates, see Sections 7.2 and 7.3.)

7.2 Performance and Scalability Notes

Several database and big-data approaches can optimize the aggregator logic or handle concurrency seamlessly—e.g., ACID-compliant relational systems (PostgreSQL, MySQL, Oracle) or distributed DBs with snapshot isolation. If you store these JSON definitions in a table and rely on standard SQL queries for aggregator expressions, you leverage decades of database optimizations for scaling.

Additionally, for HPC (High-Performance Computing) contexts (like large grid-based simulations of wavefunctions), you might store the large data arrays separately (in HDF5, for example) while still referencing them in your JSON-based "schema" definitions. That way, your aggregator fields or formula logic can call into optimized native libraries (NumPy, BLAS, etc.) for the heavy lifting.

(Recall, Section 5.5 illustrates how quantum-level models can be quite large but remain valid under a purely declarative definition.)

7.3 Future Directions for Real-Time or Large-Scale Systems

- **Real-Time Updates**: With incremental or streaming data, you can re-apply aggregator calculations at fixed intervals. Using a dynamic expression-evaluation library, you can tweak formulas while the system runs.
- **Distributed Systems**: If multiple nodes share a "master" JSON schema, code generation or formula evaluation can be done per node. Ensuring consistent snapshots across distributed nodes calls for well-known concurrency protocols (e.g., Raft or Paxos).

(See also Section 7.4 on how schemas can evolve in large or long-lived deployments, and Section 7.5 on securing dynamic evaluations.)

7.4 Schema Evolution and Long-Lived Systems

Real-world systems often require updating domain definitions over time—adding new fields, changing formulas, or splitting out large aggregates. Because CMCC treats the **model** as self-describing data (the JSON), these changes can be versioned in Git or a comparable VCS:

1. **Add or Remove Fields**: You can insert or remove schema fields, letting aggregator or calculated properties adjust accordingly.
2. **Generate Code**: Re-run the generation step for all target languages, ensuring each environment "follows" the new rules.
3. **Agent or Human Check**: Any hand-written references to old fields can be updated by developers or AI-based updaters, ensuring no orphaned references linger.

In database-backed environments (Airtable, Baserow, etc.), these schema changes are typically atomic or transactional, making it trivial to expand your model without breaking old data. The synergy between a no-code platform's consistent snapshots and CMCC's purely declarative definitions simplifies long-lived model evolution.

7.5 Security and Deployment (eval vs. Derived SDK)

Dynamic evaluation of JSON formulas (as per Section 6) can be powerful but poses potential risks:

- **Malicious Input**: An attacker could inject OS commands or memory exploits if “eval” is not properly sandboxed.
- **Performance Surprises**: If a user-provided formula tries to do nested loops or large allocations, you may see unexpected CPU or memory use.

Recommended Approach:

- **Generate a Domain-Specific Library** from the JSON at design time, rather than evaluating expressions on the fly.
- Only allow live formula editing when you trust the users or have a robust sandbox (and still be mindful of performance overhead).
- For extremely large or specialized domains, you might compile the rulebook into a custom HPC or GPU pipeline—still referencing the same aggregator logic from the JSON, but ensuring safe, optimized code.

In short, “eval” is a **last-resort** technique. CMCC’s machine-readability and structural approach make it easy to produce statically typed “SDKs” that handle all the aggregator or calculated fields at high speed, with minimal risk.

---

8. Conclusions and Future Work

8.1 Key Takeaways for Cross-Language Code Generation

1. **JSON as Single Source of Truth**
   By modeling your entire domain (S, D, L, A, F) in a single JSON, you eliminate the need to re-implement domain logic in each target language.

2. **Minimal “Glue Code”**
   The handful of imperative lines—loading data, instantiating objects, stepping through timesteps—remains nearly identical across Python, Golang, or Java, with **no embedded domain logic**.

3. **CMCC in Action**
   Triangleness (geometry) and quantum walks (wavefunction evolution) demonstrate that even *diverse* domains can be handled under the same method. This approach systematically shows the *practical* dimension of earlier theoretical claims from *From Bits to Qubits with CMCC*.

8.2 Next Steps: Expanding to More Domains

- **Biology and Systems Modeling**: Use aggregator fields to represent gene regulatory networks, with references to enzymes, promoters, and dynamic concentration fields.
- **Real-Time Finance**: Capture trading rules or compliance constraints in JSON aggregator logic, automatically generate code for multiple high-level languages.
- **AI/ML Model Introspection**: Store model hyperparameters and transformations in JSON as aggregator or lambda fields. Generate code for data preprocessing in Python, Java, and Golang.

(And for advanced concurrency or schema evolution in these domains, see Sections 7.3 and 7.4.)

8.3 Final Remarks

This paper illustrates one possible route for turning CMCC's structural declarations into actual multi-language runtime code using widely available tools (JSON, Handlebars, expression-evaluation libraries). The approach is far from the only option, but it clearly highlights the power of storing the "what" (domain logic) as data, removing the friction of DSL design or ad hoc duplication across languages.

CMCC remains open for extension, testing, and real-world adoption. We look forward to collaborations that push these ideas into even broader, more complex arenas—and, of course, any sincere falsification attempts that might sharpen or challenge the conjecture's boundaries (see the "Hardcore Falsification Checklist" in Section 2.3).

---

9. References

1. Wheeler, J. A. (1989). "Information, Physics, Quantum: The Search for Links," in *Proceedings of the 3rd International Symposium on Foundations of Quantum Mechanics*, Tokyo: Physical Society of Japan, pp. 354–368.
2. Wolfram, S. (2002). *A New Kind of Science.* Champaign, IL: Wolfram Media, ISBN 978-1579550080.
3. Turing, A. M. (1936). "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society, Series 2*, 42(1), pp. 230–265.
4. Raft consensus algorithm documentation: https://raft.github.io/.
5. Paxos consensus overview: Paxos (computer science).
6. MVEL: A powerful expression language for Java: http://mvel.documentnode.com/.
7. Janino expression compiler/interpreter: https://janino-compiler.github.io/janino/.
8. Alexandra, E. J. (2025). *From Bits to Qubits with CMCC: Demonstrating Computational Universality through Triangles, Quantum Walks, the Ruliad, and Multiway Systems with the Conceptual Model Completeness Conjecture (CMCC)*. Zenodo, DOI: 10.5281/zenodo.14761025.

Appendix A: Concrete example in Python

```
#!/usr/bin/env python3
import numpy as np
import matplotlib.pyplot as plt

##############################################################################
# CLASSES
##############################################################################

class Grid:
    """
    Holds geometry info: nx, ny, domain size, barrier row, slit geometry, etc.
    """
    def __init__(self, nx, ny, Lx, Ly, barrier_y_phys, detector_y_phys,
                 slit_width, slit_spacing):
        self.nx = nx
        self.ny = ny
        self.Lx = Lx
        self.Ly = Ly
        self.dx = Lx / nx
        self.dy = Ly / ny

        # Convert physical coords to grid indices
        self.barrier_row = int((barrier_y_phys + Ly/2) / self.dy)
        self.detector_row = int((detector_y_phys + Ly/2) / self.dy)

        # Slit geometry
        center_x = nx // 2
        self.slit_width = slit_width
        self.slit_spacing = slit_spacing
        self.slit1_xstart = center_x - slit_spacing // 2
        self.slit1_xend   = self.slit1_xstart + slit_width
        self.slit2_xstart = center_x + slit_spacing // 2
        self.slit2_xend   = self.slit2_xstart + slit_width

        # For logging/demonstration
        print(f"Barrier row={self.barrier_row}, Detector row={self.detector_row}")
        print(f"Slit1=({self.slit1_xstart}:{self.slit1_xend}),
Slit2=({self.slit2_xstart}:{self.slit2_xend})")


class CoinOperator:
    """
    Stores the NxN (in this case 8x8) matrix for the local coin operation.
    """
    def __init__(self, seed=42):
        self.matrix = self._make_coin_8(seed)

    @staticmethod
    def _make_coin_8(seed):
        rng = np.random.default_rng(seed=seed)
```

```
        mat = np.ones((8,8), dtype=np.complex128)
        alpha = 2.0
        for i in range(8):
            mat[i,i] -= alpha
        rnd = 0.05*(rng.random((8,8)) + 1j*rng.random((8,8)))
        mat += rnd
        # Force unitarity via SVD
        U, s, Vh = np.linalg.svd(mat, full_matrices=True)
        return U @ Vh

    def apply(self, spin_in):
        """
        spin_in shape=(8,), returns spin_out shape=(8,)
        """
        return self.matrix @ spin_in


class Wavefunction:
    """
    An immutable snapshot of the wavefunction at a given time:
      psi.shape = (ny, nx, 8)

    We'll have a method evolve_one_step(...) that returns a NEW Wavefunction.
    """
    DIRECTION_OFFSETS = [
        (-1,  0),  # up
        (+1,  0),  # down
        ( 0, -1),  # left
        ( 0, +1),  # right
        (-1, -1),  # up-left
        (-1, +1),  # up-right
        (+1, -1),  # down-left
        (+1, +1),  # down-right
    ]

    def __init__(self, grid: Grid, array_psi: np.ndarray):
        """
        array_psi is shape=(ny,nx,8), complex
        """
        self.grid = grid
        self.psi = array_psi  # store the array as immutable
        # no direct assignment to self.psi[...] from outside

    @classmethod
    def initial_condition(cls, grid: Grid):
        """
        Build the wavefunction at t=0, as a Gaussian in y near the bottom,
        wide in x, same amplitude in all directions.
        """
        psi0 = np.zeros((grid.ny, grid.nx, 8), dtype=np.complex128)
        # let's pick a src_y ~ 15% from bottom:
        src_y = int(grid.ny * 0.15)
```

```
        sigma_y = 5.0

        for y in range(grid.ny):
            dy = y - src_y
            amp = np.exp(-0.5*(dy/sigma_y)**2)
            for d in range(8):
                psi0[y,:,d] = amp

        return cls(grid, psi0)

    def evolve_one_step(self, coin: CoinOperator, measure_barrier=False):
        """
        Return a NEW Wavefunction at time t+1, applying:
          1) coin step
          2) shift step
          3) barrier or measure (if measure_barrier=True)
        """
        ny, nx, ndir = self.psi.shape
        # 1) Coin step
        psi_coin = np.zeros_like(self.psi, dtype=np.complex128)
        for y in range(ny):
            for x in range(nx):
                spin_in = self.psi[y,x,:]  # shape=(8,)
                spin_out = coin.apply(spin_in)
                psi_coin[y,x,:] = spin_out

        # 2) Shift step
        psi_shift = np.zeros_like(psi_coin, dtype=np.complex128)
        for d, (ofy, ofx) in enumerate(self.DIRECTION_OFFSETS):
            shifted_dir = np.roll(psi_coin[:,:,d], shift=ofy, axis=0)
            shifted_dir = np.roll(shifted_dir, shift=ofx, axis=1)
            psi_shift[:,:,d] = shifted_dir

        # 3) Barrier or measurement
        if measure_barrier:
            # measure_collapse_barrier logic
            psi_out = self._collapse_barrier(psi_shift)
        else:
            # normal barrier
            psi_out = self._apply_barrier(psi_shift)

        return Wavefunction(self.grid, psi_out)

    def _apply_barrier(self, psi_in):
        """
        Normal barrier => zero out barrier row except slit columns.
        """
        psi_out = psi_in.copy()
        br = self.grid.barrier_row
        psi_out[br,:,:] = 0
        s1s, s1e = self.grid.slit1_xstart, self.grid.slit1_xend
        s2s, s2e = self.grid.slit2_xstart, self.grid.slit2_xend
```

```python
        psi_out[br, s1s:s1e, :] = psi_in[br, s1s:s1e, :]
        psi_out[br, s2s:s2e, :] = psi_in[br, s2s:s2e, :]
        return psi_out

    def _collapse_barrier(self, psi_in):
        """
        Collapsing amplitude in barrier row => sum intensities across directions,
        keep only slit columns, sqrt(keep / max), put in direction=0
        """
        psi_out = psi_in.copy()
        ny, nx, ndir = psi_in.shape
        br = self.grid.barrier_row
        # sum intensities across directions
        row_intens = np.sum(np.abs(psi_in[br,:,:])**2, axis=-1)  # shape=(nx,)

        keep = np.zeros_like(row_intens)
        s1s, s1e = self.grid.slit1_xstart, self.grid.slit1_xend
        s2s, s2e = self.grid.slit2_xstart, self.grid.slit2_xend
        keep[s1s:s1e] = row_intens[s1s:s1e]
        keep[s2s:s2e] = row_intens[s2s:s2e]

        m = np.max(keep)
        if m > 1e-30:
            keep /= m
        amps = np.sqrt(keep)
        psi_out[br,:,:] = 0
        psi_out[br,:,0] = amps  # put amplitude in direction=0
        return psi_out

    def total_norm(self):
        """
        Returns the sum of |psi|^2 over all y,x,d.
        """
        return np.sum(np.abs(self.psi)**2)

    def detector_row_intensity(self):
        """
        Summation over directions at 'detector_row', returns shape=(nx,).
        """
        dr = self.grid.detector_row
        row_amp = self.psi[dr, :, :]  # shape=(nx,8)
        row_intens = np.sum(np.abs(row_amp)**2, axis=-1)  # shape=(nx,)
        return row_intens


class QWalkRunner:
    """
    Orchestrates the layer-by-layer evolution in an immutable, functional style.
    - We keep a list of Wavefunction objects, wave[t].
    - wave[t+1] = wave[t].evolve_one_step(...)
```

```
    We can optionally do a measurement collapse at t=steps_to_barrier.
    """
    def __init__(self, grid: Grid, coin: CoinOperator, steps_to_barrier, steps_after_barrier):
        self.grid = grid
        self.coin = coin
        self.steps_to_barrier = steps_to_barrier
        self.steps_after_barrier = steps_after_barrier

    def run_experiment(self, collapse=False):
        """
        Return the final Wavefunction after steps_to_barrier + steps_after_barrier.
        """
        t_final = self.steps_to_barrier + self.steps_after_barrier
        # We'll keep each wavefunction in a list for demonstration
        wave = [None]*(t_final+1)
        wave[0] = Wavefunction.initial_condition(self.grid)

        # Evolve up to barrier
        for t in range(self.steps_to_barrier):
            wave[t+1] = wave[t].evolve_one_step(self.coin, measure_barrier=False)

        # If collapse => measure at t=steps_to_barrier
        if collapse:
            wave[self.steps_to_barrier] = wave[self.steps_to_barrier-1].evolve_one_step(self.coin,
measure_barrier=True)
        else:
            # else wave[self.steps_to_barrier] was already created with measure=False above
            pass

        # Evolve remainder
        for t in range(self.steps_to_barrier, t_final):
            wave[t+1] = wave[t].evolve_one_step(self.coin, measure_barrier=False)

        return wave[t_final]  # final wavefunction

##############################################################################
# MAIN
##############################################################################
def main():
    # 1) Build the Grid
    nx = 201
    ny = 201
    Lx, Ly = 16.0, 16.0
    steps_to_barrier = 80
    steps_after_barrier = 200

    barrier_y_phys = -2.0
    detector_y_phys = 5.0
    slit_width = 3
    slit_spacing = 12

    grid = Grid(nx, ny, Lx, Ly, barrier_y_phys, detector_y_phys,
```

```
            slit_width, slit_spacing)

# 2) Create the Coin
coin = CoinOperator(seed=42)

# 3) Create a QWalkRunner
runner = QWalkRunner(grid, coin, steps_to_barrier, steps_after_barrier)

# 4) Run the "FULL" wave (no measurement)
print("Running FULL wave (no barrier measurement)...")
wave_full = runner.run_experiment(collapse=False)
norm_full = wave_full.total_norm()
print(f"Final norm (full)={norm_full:.3g}")

# 5) Run the "COLLAPSED" wave (with measurement)
print("Running COLLAPSED wave (with barrier measurement)...")
wave_coll = runner.run_experiment(collapse=True)
norm_coll = wave_coll.total_norm()
print(f"Final norm (collapsed)={norm_coll:.3g}")

# 6) Measure intensity at the detector row => sum over directions => shape=(nx,)
int_full = wave_full.detector_row_intensity()
int_coll = wave_coll.detector_row_intensity()

# 7) Normalize each
mf = np.max(int_full)
if mf > 1e-30:
    int_full /= mf
mc = np.max(int_coll)
if mc > 1e-30:
    int_coll /= mc

# 8) Build 2D "screen" => tile 1D intensity
screen_height = 60
screen_full = np.tile(int_full, (screen_height,1))
screen_coll = np.tile(int_coll, (screen_height,1))

# 9) Plot
plt.figure(figsize=(8,4))
plt.imshow(screen_full, origin="lower", aspect="auto", cmap="inferno")
plt.title("Full Wave - Detector Row (OO, Functional layering)")
plt.xlabel("x-index")
plt.ylabel("Screen Height")
plt.colorbar(label="Normalized Intensity")

plt.figure(figsize=(8,4))
plt.plot(int_full, 'o-')
plt.title("Full Wave - 1D Intensity at Detector")
plt.xlabel("x-index")
plt.ylabel("Intensity")

plt.figure(figsize=(8,4))
```

```python
    plt.imshow(screen_coll, origin="lower", aspect="auto", cmap="inferno")
    plt.title("Collapsed Wave – Detector Row (OO, Functional layering)")
    plt.xlabel("x-index")
    plt.ylabel("Screen Height")
    plt.colorbar(label="Normalized Intensity")

    plt.figure(figsize=(8,4))
    plt.plot(int_coll, 'o-')
    plt.title("Collapsed Wave – 1D Intensity at Detector")
    plt.xlabel("x-index")
    plt.ylabel("Intensity")

    plt.tight_layout()
    plt.show()


if __name__=="__main__":
```