Part 2: JSON-Driven Domain Models in Practice

Triangleness, Quantum Fields, and Beyond

Part 1: From Bits to Qubits with CMCC:Demonstrating Computational Universality through Triangles, Quantum Walks, the Ruliad and Multiway Systems

Author: EJ Alexandra
Affiliation: SSoT.me & EffortlessAPI.com
Contact: start@anabstractlevel.com
Date: March 2025

Abstract

Building upon the theoretical foundations established in Part 1, this second paper demonstrates the practical utility of CMCC through a straightforward, JSON-based rulebook approach. By encoding all conceptual knowledge in the same five declarative primitives and employing JSON as a universal, machine-readable mirror, developers can automatically generate consistent, cross-language implementations without resorting to specialized domain-specific languages or duplicated logic. We illustrate this practical approach through two distinct yet related examples: a geometric scenario defining and verifying properties of triangles, and a quantum simulation representing quantum walks and double-slit experiments.

Using template-based transformations (via Handlebars) from the JSON rulebook, we generate statically typed helper code across multiple programming environments, including Python, Golang, and Java. We demonstrate that the same declarative JSON file can robustly and simultaneously capture simple polygons as well as complex quantum phenomena, ensuring domain fidelity across diverse languages with minimal effort. This results in a robust, auditable, and easily maintainable system for modeling and evolving complex systems across domains. The paper closes by addressing performance considerations, scalability challenges, and schema evolution, reinforcing CMCC's potential as a powerful, practical, and universal declarative framework for computational modeling.

Table of Contents

1. Introduction

1.1 Context and Relationship to Part 1

In **Part 1**, we established the theoretical foundation of the Conceptual Model Completeness Conjecture (CMCC), arguing that any finite computable concept can be captured using only five primitives (Schema, Data, Lookups, Aggregations, and Lambda Calculated Fields). There, we also explored how these primitives align with Turing-completeness, Wolfram's multiway systems, and Wheeler's "It from Bit" paradigm.

This **Part 2** focuses on the *practical* implications. We move from abstract proofs of universality to concrete, JSON-driven code generation workflows. By the end of this paper, you'll see how to build real applications (in Python, Go, or Java) that share a single, authoritative "rulebook" for domain logic—eliminating domain-specific languages and repeated logic.

1.2 Objectives: Hands-On CMCC Implementation

The Conceptual Model Completeness Conjecture (CMCC) posits that any finite computable rule can be encoded without writing a custom DSL or embedding domain rules in imperative code. Instead, it relies on:

- **Schema (S):** Describes your entities, tables, or object types.
- **Data (D):** Instantiates records of those entities.
- **Lookups (L):** Establishes structural references between records.
- **Aggregations (A):** Summaries or roll-ups across a set of records (e.g., sums, counts).
- **Calculated Fields (F):** Declarative formulas that let you define logic or constraints (e.g., Pythagorean checks, unitarity tests).

When these five primitives are consistently applied in a snapshot or transactionally consistent environment, you eliminate the need for extraneous "glue code." Part 2 demonstrates how these elements map directly to JSON definitions and automatically generated source code.

Here, we aim to demonstrate how:

1. A **single JSON rulebook** captures all relevant domain definitions—fields, relationships, aggregator formulas—using minimal, standardized syntax.
2. You can **auto-generate typed stubs** in Python, Golang, or Java (or your language of choice) via a simple template engine.
3. A small, **minimal imperative script** in each language suffices to run scenarios (e.g., verifying right triangles or simulating quantum interference).
4. **Updating** the JSON definitions seamlessly re-syncs logic across all language environments, eliminating DSL duplication or specialized parser overhead.

By the end, you'll see that the same approach supports both a straightforward geometry use case (triangleness) and a more advanced quantum scenario. This further underscores CMCC's promise: once domain logic is declared structurally, it can be shared, re-targeted, or extended with minimal friction.

1.3 JSON as a Universal "Rulebook" Format

JSON is ubiquitous in modern development, supported by nearly every programming language and ecosystem. By treating **a single JSON file** (or set of JSON files) as the "rulebook" for your domain, you gain:

- **Machine-Readability:** Parsers exist for all major languages.
- **Declarative Clarity:** Every aggregator, formula, and reference is visible as structured data—no custom syntax.
- **Unified Source of Truth:** Any updates (e.g., new domain fields) happen in one place.

This eliminates the typical friction of DSL design, parser maintenance, or the risk of domain logic drifting across multiple implementations.

1.4 Objectives and Source Code

The objectives of this paper are:

1. **Show that a single JSON rulebook** can represent all domain definitions—schemas, relationships, aggregator formulas, etc.

2. **Demonstrate code generation** for multiple programming languages using templates (e.g., Handlebars).
3. **Prove practicality** through two examples: Triangleness (basic geometry) and a simplified quantum walk (physics-inspired).

All example files, templates, and generated stubs are available on GitHub:

bash

```bash
https://github.com/eejai42/conceptual-model-completeness-conjecture-toe-meta-model
```

We encourage you to fork the repo, modify the JSON definitions, and see how straightforward it is to regenerate domain-specific code across multiple languages.

1.5 🔴 CMCC Falsification Checklist

As in Part 1, we emphasize that CMCC remains open to rigorous challenges. If you suspect a particular rule cannot be captured in (S, D, L, A, F), **we welcome concrete counterexamples**. **Convinced you have a real counterexample?** Please reach out. We want to test any claim that a finite computable rule can't be expressed in CMCC.

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.

2. Practical Framework

This section presents a hands-on workflow for building fully declarative domain models in JSON and then mapping them to any runtime environment—Python, Go, Java, etc. By treating JSON as the canonical "rulebook," we avoid domain-specific languages (DSLs) and keep domain logic entirely separate from any imperative scripts. The end result is an auditable, maintainable, and extensible system where conceptual definitions remain stable even as the underlying code or platforms evolve.

2.1 Why We Don't Need a DSL

Traditional approaches to modeling complex domains often involve:

1. **Scripting or imperative logic** scattered throughout the codebase, or
2. **Domain-Specific Languages (DSLs)** that introduce specialized syntax and parsers.

Both strategies introduce long-term friction:

- **Imperative logic** duplicates domain knowledge. Each new language or environment (Python, Java, etc.) rewrites the same rules, increasing maintenance overhead.

- **DSLs** can be elegant initially but often require custom parsers or compilers, leading to "DSL drift" (the DSL evolves in parallel with the domain) and reliance on niche toolchains.

CMCC's JSON-based approach sidesteps these pitfalls:

- **No specialized syntax** beyond simple JSON structures.
- **No custom parser** needed; every environment can parse JSON out of the box.
- **No logic duplication** because all "rules" (aggregations, references, formulas) live in the single, authoritative JSON file.

Whenever you modify the domain—adding new fields, updating formulas, or extending relationships—you do so declaratively in JSON. Code in each target language is then generated or adapted automatically, ensuring perfect fidelity across your ecosystem.

2.2 Defining the Domain Model in JSON

At the core, you capture your domain in JSON using five declarative primitives (S, D, L, A, F):

1. **Schema (S)** – Defines the entity types (e.g., *Polygon*, *Wavefunction*, *Order*) and their fields.
2. **Data (D)** – Holds actual instances. You might store these records in separate JSON files or load them from a database.
3. **Lookups (L)** – Indicate references between entities (e.g., "An *Edge* looks up two *Points*").
4. **Aggregations (A)** – Summaries across collections (e.g., "SUM of edges.length to get the total perimeter").
5. **Calculated Fields (F)** – Formulas or constraints (e.g., "`is_right_triangle` = ...some Pythagorean check...").

A typical JSON snippet might look like this:

json

Copy

```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": "is_triangle",
          "type": "calculated",
          "formula": "EQUAL(edge_count, 3)"
        }
      ]
    },
    {
      "name": "Edge",
      "fields": [
        { "name": "edge_id",  "type": "string" },
        { "name": "start_pt", "type": "lookup", "references": "Point" },
        { "name": "end_pt",   "type": "lookup", "references": "Point" },
        {
          "name": "length",
          "type": "calculated",
          "formula": "SQRT( POW(SUBTRACT(end_pt.x,start_pt.x),2) +
POW(SUBTRACT(end_pt.y,start_pt.y),2) )"
        }
      ]
    },
    {
```

```
      "name": "Point",
      "fields": [
        { "name": "x", "type": "number" },
        { "name": "y", "type": "number" }
      ]
    }
  ]
}
```

Key Observations:

- The “logic” that *makes* a polygon a triangle appears in `is_triangle`. No additional scripts or DSL grammar is needed.
- Lookups (`start_pt`, `end_pt`) capture how edges refer to points.
- Calculated fields (`length`) capture geometry without imperative loops.
- Aggregations (`edge_count`) or more advanced ones (like `SUM(...)`) handle second-order logic automatically.

2.3 Template-Based Code Generation

Once you have this JSON “rulebook,” you can generate typed classes, structs, or modules in any language. One popular mechanism is using **Handlebars** (or a similar templating system):

1. **Template** – A parametric file describing how an entity, field, or formula maps to code in your target language.
2. **Code-Generation Step** – A CLI tool or script reads both the JSON definition and the template, then produces the language-specific source code.

For example, a **Python** template might look like:

handlebars

Copy

```
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):
    # Implement your formula: {{this.formula}}
    return evaluate_formula("{{this.formula}}", context=self)
{{/if}}
{{/each}}
```

And a **Golang** template might define a struct plus methods for aggregator formulas. Either way, the user never manually re-implements domain logic. You make changes in JSON, re-run your template generator, and instantly get updated code for all target environments.

Benefits

- **Maintenance** – Eliminates the risk of logic drifting across multiple languages.
- **Simplicity** – You keep a single JSON file under version control, while each generated file can be re-created at will.
- **Extensibility** – You can add a second or third code template (e.g., for Java or TypeScript) and produce those bindings with the same one-click process.

2.4 Minimal Imperative Scripts (Rulebook vs. Runtime)

Where does "execution" happen?

- The JSON describes *what* the domain is: schemas, formulas, references, aggregator rules.
- The *runtime script* in each language determines *how* and *when* to execute these rules—often just by instantiating the classes, loading data, and calling aggregator or formula properties.

In many cases, the imperative script might simply:

1. Parse or load the JSON definitions (or use the generated code).

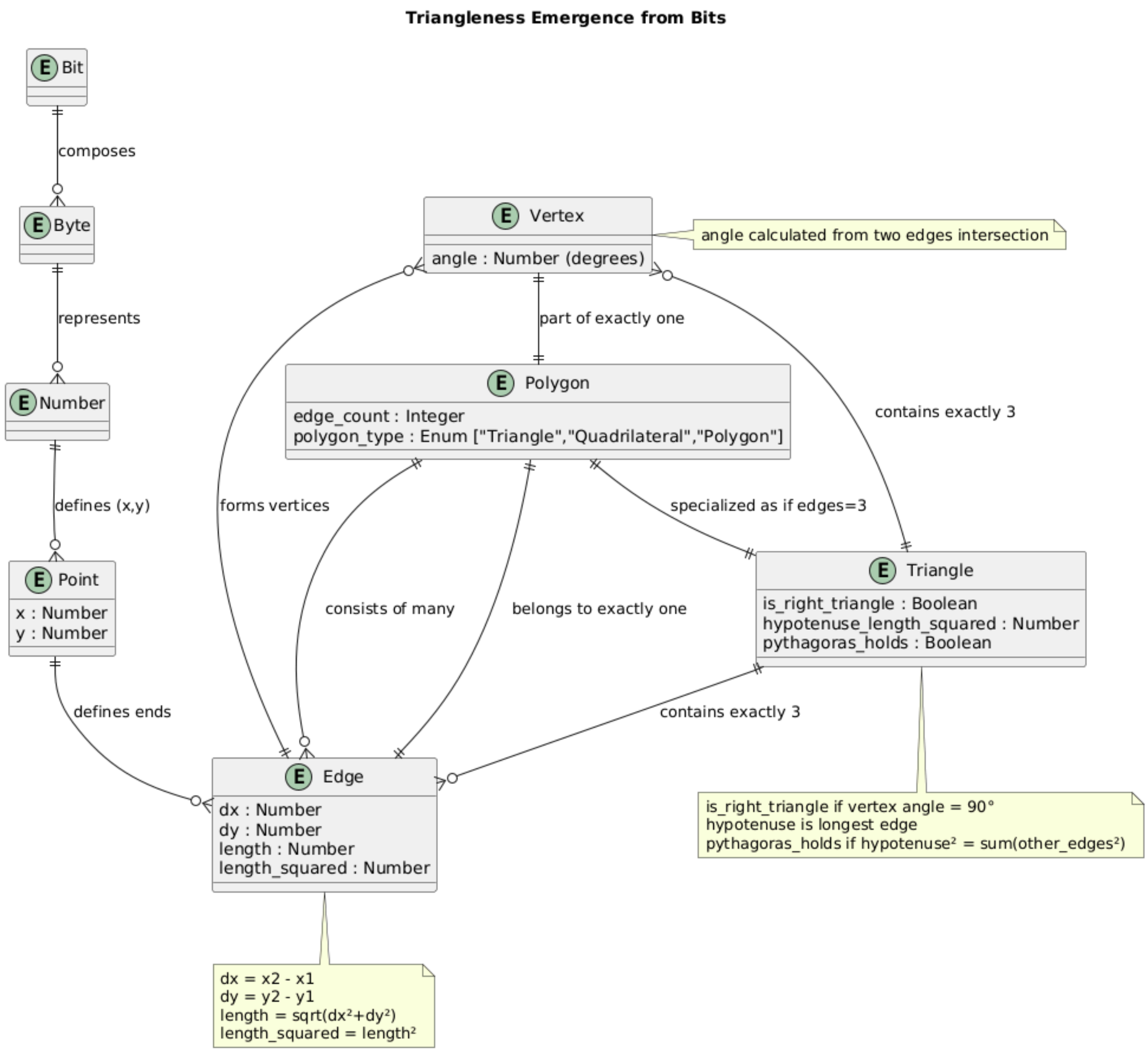
2. Build actual objects from user or database data.
3. Access aggregator or calculated properties to trigger logic automatically.

As your system grows, you can maintain concurrency or transaction boundaries at a higher architectural level (e.g., using a database with snapshot isolation). But none of that changes the domain definitions in JSON—once again separating *how you run it* from *what your logic is*.

---

3. Case Study: Triangleness

This short example demonstrates how an everyday geometric concept—“triangleness”—emerges purely from the CMCC approach. We’ll define Polygons, Edges, and Points in JSON, let aggregator fields count edges, and let a calculated field decide whether that count equals three.



3.1 JSON Fields for Polygons, Edges, Points

Consider the snippet from Section 2.2 for *Polygon*, *Edge*, and *Point*. The important bits are:

- **`edge_count`** is an aggregator: `COUNT(edges)`.
- **`is_triangle`** is a calculated field: `EQUAL(edge_count,3)`.

These alone suffice to categorize a polygon as a triangle, without any “if/else” code. Once you run code generation, your classes (Python, Go, Java, etc.) automatically have `is_triangle` properties. Whether it’s an actual triangle is determined at *runtime* purely by the data.

3.2 Emergent Geometry & Checking Right Triangles

You can take this further with a *right-triangle check*. For example:

json

Copy

```
{
  "name": "is_right_triangle",
  "type": "calculated",
  "formula": "AND(is_triangle, EQUAL(PYTHAG_SUM, ???))"
}
```

Even more advanced geometry—like checking angles or verifying circumscribed circles—can all be expressed as aggregator or formula fields. No specialized geometry DSL or library is strictly necessary. The “triangle vs. not triangle” question is answered automatically as soon as you have data for the edges and points.

Result: *Triangleness* is not a separate procedure. It’s simply an emergent property derived from relationships (lookups) plus aggregator logic.

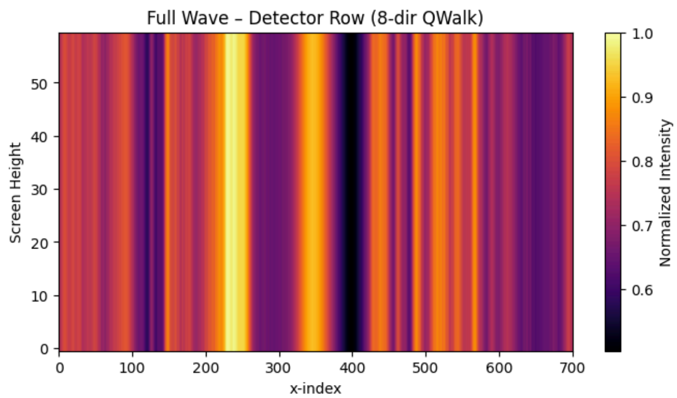
4. Case Study: Quantum Walk / Double-Slit

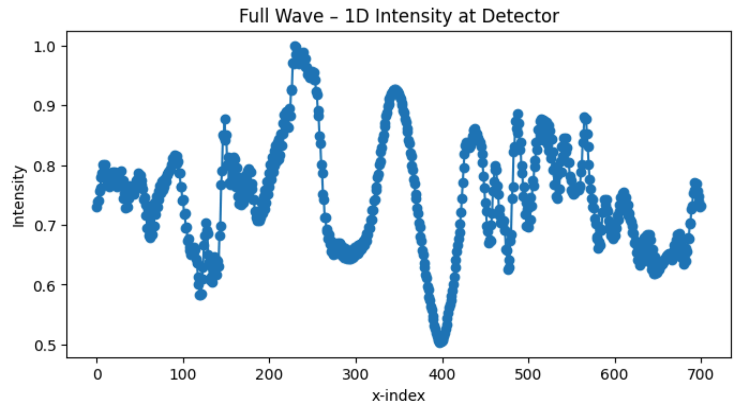
This section illustrates how our JSON-based CMCC approach extends beyond simple geometric domains (triangleness) to more complex physical phenomena. We use the example of a **quantum walk** (akin to a discrete double-slit experiment). Despite seemingly large conceptual distance from triangles, the structural methodology remains the same: we define everything—entities, references, aggregator formulas—in JSON, and let the system’s “runtime” interpret those rules.

Full Wave vs. Collapsed Wave Visualizations

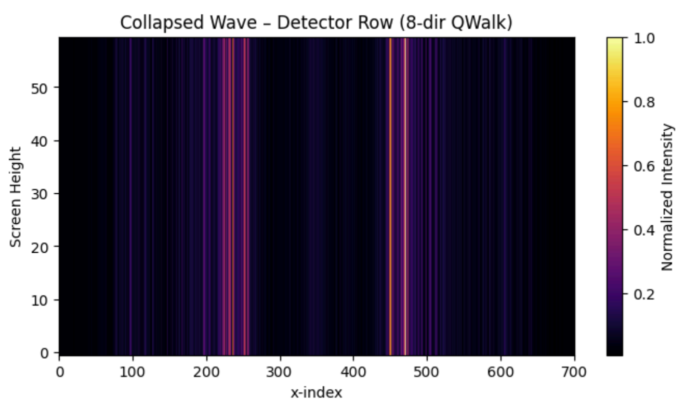
By simulating a double-slit interference scenario, we can produce two distinctive images of the wavefunction at a detector row:

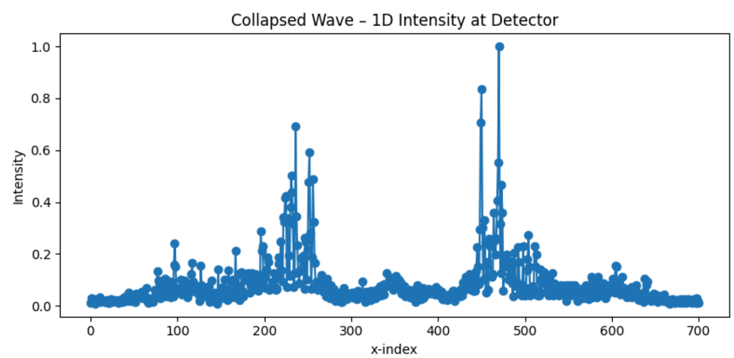
1. **Full Wave Visualization** – Shows interference patterns that emerge when the wavefunction is allowed to propagate unimpeded.





2. **Collapsed Wave Visualization** – Depicts the wavefunction after a measurement (collapse) at the barrier row, highlighting how measurement “zeroes out” amplitude outside the slit channels.





Critically, these outcomes arise **entirely from the JSON-based structural definitions**, *not* from explicitly coding Schrödinger’s equation. We never directly solve partial differential equations; interference and collapse appear naturally from aggregator and calculated-field logic.

---

5. QFT Primer for Non-Physicists

A **quantum walk** is a discrete analog of how quantum particles spread and interfere:

1. A **“coin operator”** (unitary matrix) rotates or mixes spin/directional states of the wavefunction.
2. Amplitudes **shift** to neighboring cells on a grid, causing interference where paths overlap.
3. **Measurement (collapse)** “zeros out” amplitude in certain locations or states, reflecting physical detection or barrier interactions.

In a traditional approach, one might encode these steps in specialized quantum-simulation libraries or PDE solvers. **Under CMCC**, the same logic emerges as a set of declarative references, aggregations, and formulas—written in JSON, with no specialized quantum DSL.

5.1 JSON Layout for Grid, CoinOperator, and Wavefunction

Below is a simplified excerpt (inspired by *From Bits to Qubits with CMCC*) defining three key entities: **Grid**, **CoinOperator**, and **Wavefunction**. Each encapsulates part of the quantum system's state or transformations:

5.2 Illustrative Excerpt and Reference to Appendix

The JSON above comprehensively models a quantum walk/double-slit scenario—defining **Grid**, **CoinOperator**, **Wavefunction**, and associated "step" processes (BarrierStep, CollapseBarrierStep, etc.). For discussion in the main text, we often focus on just a few key entities, like `Grid` and `CoinOperator`, as shown below:

```
[
  {
    "name": "Grid",
    "fields": [
      { "name": "nx", "type": "number" },
      { "name": "ny", "type": "number" },
      { "name": "Lx", "type": "number" },
      { "name": "Ly", "type": "number" }
      // ...
    ]
  },
  {
    "name": "CoinOperator",
    "fields": [
      { "name": "Matrix", "type": "tensor", "tensor_shape": "(8,8)" },
      { "name": "seed", "type": "number" }
      // ...
    ]
  }
  // ...
]
```

In practice, **all** of the fields—calculated or otherwise—play a role in orchestrating quantum-walk behavior. Readers interested in the full schema (including aggregator formulas for barrier steps, measurement collapse, and final wavefunction evolution) can refer to **Appendix A** (above) or consult the open-source repository listed in Section XX for the complete JSON file.

- **Grid**: Defines lattice dimensions, physical geometry, barrier/detector positions, etc.
- **CoinOperator**: Holds an 8×8 matrix mixing spin or directional states each step; a `unitarity_check` field confirms MM†=I\mathbf{M}\mathbf{M}^\dagger = \mathbf{I}MM†=I.
- **Wavefunction**: Stores ψ\psiψ as a 3D tensor (ny,nx,8)(ny, nx, 8)(ny,nx,8) capturing amplitude distribution. An aggregator `total_norm` sums ∣ψ∣2|\psi|^2∣ψ∣2 over the grid, confirming probability conservation.

Even advanced quantum details (e.g., multi-slit barriers, spin dimension) reside in these JSON definitions. No specialized quantum language or PDE code is used; we simply specify the "facts" and "formulas" in a standard JSON structure.

5.3 Evolving the System in a Minimal Imperative Script

While CMCC covers **what** each entity is and how values relate (aggregators, references, formulas), it says little about **how** you step through time or apply updates. That task falls to a small "orchestrator" script in Python, Go, Java, etc., which references the auto-generated classes.

Example (Python)

```
import domain_generated as dg
import numpy as np

def main():
    # 1) Initialize domain objects
    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  # initial amplitude at center

    # 2) Check unitarity from the JSON-based formula
    print("Coin operator is unitary?", coin.unitarity_check)

    # 3) Evolve wavefunction step by step
    for t in range(50):
        wave.timestep = t
        wave.psi = do_quantum_walk_step(wave.psi, coin.matrix)
        # Possibly apply barrier logic or partial measurement
        print(f"Timestep {t}, total_norm = {wave.total_norm}")

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

Here, the domain logic (e.g., unitarity checks, shape of ψ\psiψ) is fully declared in JSON. The script's `do_quantum_walk_step` remains minimal "glue," orchestrating step-by-step transformations. If new fields (like a `slit_mask`) appear in the JSON, we just regenerate `domain_generated.py` and incorporate them—no rewriting domain logic in code.

5.3 Observing Interference and Measurement

Over time, the wavefunction spreads and overlaps, forming an interference pattern at the detector row. If we include a measurement step that collapses amplitude outside the slits, the wavefunction changes accordingly. **No imperative "quantum code"** is needed to produce these phenomena. The aggregator formulas in JSON yield emergent properties (like probability norm, overlap sums, or partial collapses) each time the data updates.

Performance scaling tips—such as storing large arrays in HDF5 or leveraging HPC libraries—are discussed in **Section 7.2**. Security concerns around dynamic formula evaluation are addressed in **Section 7.5**.

6. “Lambda-to-X” Runtime Libraries

Our examples so far have used either (a) compiled aggregator logic in the generated classes or (b) small formula snippets. However, certain use cases may require changing formulas at runtime, without regenerating code.

1. **Python** has `eval()` or libraries like *numexpr*, *asteval* for dynamic expressions.
2. **Golang** can leverage *govaluate* or *expr* to interpret string-based formulas on the fly.
3. **Java** has *MVEL* or *Janino*, enabling in-process compilation or interpretation.

This approach makes the system *truly dynamic*, letting operators edit JSON formulas mid-run. The trade-offs include performance overhead and security concerns (see **Section 7.5**). For production, design-time generation often remains safer, more stable, and easier to optimize.

7. Discussion

7.1 Advantages and Caveats

Advantages

- **No Drift Across Languages**
 A single JSON rulebook updates all generated code in Python, Go, Java, etc.

- **Declarative Clarity**
 Domain logic resides in aggregator formulas and references that are human-readable (and machine-parseable), avoiding hidden assumptions.

- **Handles Large & Varied Domains**
 Extending the domain means adding or adjusting JSON definitions—no new DSL syntax.

- **Bridges Theory and Practice**
 The same CMCC primitives proven in “paper-level” arguments (triangles, quantum systems) now integrate seamlessly into real code.

Possible Caveats

- **Performance Overheads**
 Especially with many aggregator fields or dynamic expression parsing.

- **Complexity in Large Models**
 JSON might become unwieldy for thousands of entities or relationships. Splitting into multiple files or using a model editor can help.

- **Runtime vs. Design-Time Confusion**
 Emphasize that JSON is a rule definition; the actual “stepping” or dynamic changes happen in whichever runtime orchestrator you choose.

7.2 Scaling, Performance, and HPC Considerations

Relational databases or distributed systems can handle large concurrency and aggregator queries. For HPC-level tasks (e.g., simulating a 1000×1000 grid wavefunction over many time steps), link aggregator fields to numeric libraries (NumPy, BLAS) or store big arrays in specialized data formats. The JSON remains the "blueprint," ensuring consistency of domain definitions, even if the actual numeric kernels run on GPUs or large clusters.

7.3 Future Directions for Real-Time or Multi-Node Setups

- **Real-Time Streaming:** Reapply aggregator calculations on incoming data. Could modify or add aggregator fields on the fly with dynamic expression engines.
- **Distributed Environments:** Sync JSON-based domain definitions across multiple nodes, with each node generating local code. Common concurrency protocols (Raft, Paxos) maintain global consistency in snapshot updates.

7.4 Security and Deployment

Permitting untrusted users to alter JSON formulas at runtime can introduce vulnerabilities. Attackers might embed malicious expressions or resource-intensive computations:

- **Recommended**:
    - Use design-time generation (static code) for stable production deployments.
    - If truly dynamic editing is necessary, implement strict sandboxing with memory/time limits.

8. Conclusion and Future Work

8.1 Key Takeaways

1. **JSON as a Single Source of Truth**
   All domain rules—whether geometry or quantum amplitude—live in the same universal format.

2. **Minimal Glue Code**
   Just orchestrate or step through the domain logic; the logic itself remains in aggregator fields and formula definitions.

3. **End-to-End CMCC Demonstration**
   Triangles and quantum phenomena both align seamlessly with CMCC. This underscores the practicality behind the theoretical claims of *From Bits to Qubits with CMCC*.

8.2 Next Steps

- **Broader Domain Applications**: Finance, biology, knowledge graph alignment, or AI introspection.
- **Schema Evolution & Tooling**: Larger models might benefit from integrated visual editors or specialized JSON-based schema versioning.
- **HPC Integrations**: Mapping aggregator logic onto GPU kernels or MPI-based clusters while preserving the JSON blueprint.

CMCC invites continued experimentation and potential falsification: any finite computable domain that *cannot* be captured in (S, D, L, A, F) would be a major finding. None have been discovered thus far, reinforcing the conjecture's standing as a universal declarative framework.

---

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.

Additional Background and Research on the CMCC Framework

9. Alexandra, E.J. *BRCC-Proof: The Business Rule Completeness Conjecture and Its Proof Sketch.* 2025. Zenodo: 14759299.
10. Alexandra, E.J. *The Conceptual Model Completeness Conjecture (CMCC) as a Universal Declarative Framework.* 2025. Zenodo: 14760293.
11. Alexandra, E.J. *Formalizing Gödel's Incompleteness Theorem within CMCC and BRCC.* 2025. Zenodo: 14767367.
12. Alexandra, E.J. *Computational Paradoxes: A Database-Theoretic Approach to Self-Reference, Causality, and Gödel's Incompleteness.* 2025. Zenodo: 14776024.
13. Alexandra, E.J. *Quantum CMCC: A High-Fidelity Declarative Framework for Modeling Quantum Concepts.* 2025. Zenodo: 14776430.
14. Alexandra, E.J. *CMCC-Driven Graph Isomorphism: A Declarative and Semantically-Rich Framework.* 2025. Zenodo: 14776619.
15. Alexandra, E.J. *Applying CMCC to Model Theory: Zilber's Pseudo-Exponential Fields and the Real Exponential Field.* 2025. Zenodo: 14777134.

Appendix A: Full JSON for Quantum Walk & Double-Slit System


```json
[
  {
    "name": "Grid",
    "fields": [
      { "name": "nx", "type": "number", "description": "Number of grid points in the x-direction." },
      { "name": "ny", "type": "number", "description": "Number of grid points in the y-direction." },
      { "name": "Lx", "type": "number", "description": "Physical domain size in x." },
      { "name": "Ly", "type": "number", "description": "Physical domain size in y." },
      { "name": "dx", "type": "calculated", "formula": "DIVIDE(Lx,nx)", "description": "Spatial step in x (Lx / nx)." },
      { "name": "dy", "type": "calculated", "formula": "DIVIDE(Ly,ny)", "description": "Spatial step in y (Ly / ny)." },
      { "name": "barrier_y_phys", "type": "number", "description": "Physical y-coordinate where barrier is placed." },
      { "name": "detector_y_phys", "type": "number", "description": "Physical y-coordinate of the detector row." },
      {
        "name": "barrier_row",
        "type": "calculated",
        "formula": "FLOOR(DIVIDE(ADD(barrier_y_phys,DIVIDE(Ly,2)),dy))",
        "description": "Barrier row index, computed from physical coordinate."
      },
      {
        "name": "detector_row",
        "type": "calculated",
        "formula": "FLOOR(DIVIDE(ADD(detector_y_phys,DIVIDE(Ly,2)),dy))",
        "description": "Detector row index, computed from physical coordinate."
      },
      { "name": "slit_width", "type": "number", "description": "Number of grid columns spanned by each slit." },
      { "name": "slit_spacing", "type": "number", "description": "Distance (in columns) between the two slits." },
      {
        "name": "center_x",
        "type": "calculated",
        "formula": "FLOOR(DIVIDE(nx,2))",
        "description": "The x-center column index (middle of the domain)."
      },
      {
        "name": "slit1_xstart",
        "type": "calculated",
        "formula": "SUBTRACT(center_x,FLOOR(DIVIDE(slit_spacing,2)))",
        "description": "Left edge of slit #1."
      },
      {
        "name": "slit1_xend",
        "type": "calculated",
        "formula": "ADD(slit1_xstart,slit_width)",
        "description": "Right edge of slit #1."
      },
      {
        "name": "slit2_xstart",
        "type": "calculated",
        "formula": "ADD(center_x,FLOOR(DIVIDE(slit_spacing,2)))",
        "description": "Left edge of slit #2."
      },
      {
        "name": "slit2_xend",
        "type": "calculated",
        "formula": "ADD(slit2_xstart,slit_width)",
        "description": "Right edge of slit #2."
      }
    ]
  },
  {
    "name": "CoinOperator",
    "fields": [
      {
        "name": "Matrix",
```

```
        "type": "tensor",
        "tensor_shape": "(8,8)",
        "description": "8x8 unitary coin operator matrix."
      },
      { "name": "seed", "type": "number", "description": "Random seed for reproducibility." },
      {
        "name": "UnitarityCheck",
        "type": "calculated",
        "formula": "EQUAL(MULTIPLY(Matrix,CONJUGATE_TRANSPOSE(Matrix)),IDENTITY(8))",
        "description": "Checks if Matrix * Matrix^† = I (tests unitarity)."
      }
    ]
  },
  {
    "name": "WavefunctionInitial",
    "fields": [
      { "name": "src_y",   "type": "number", "description": "Y-center of the initial Gaussian wave packet." },
      { "name": "sigma_y", "type": "number", "description": "Std. dev. of the Gaussian in y." },
      {
        "name": "psi_init",
        "type": "calculated",
        "tensor_shape": "(ny,nx,8)",
        "formula": "GAUSSIAN_IN_Y_AND_UNIFORM_IN_X_AND_DIRECTION(src_y, sigma_y, Grid.ny, Grid.nx, 8)",
        "description": "Initial wavefunction: Gaussian in y, uniform across x and spin directions."
      }
    ]
  },
  {
    "name": "CoinStep",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Input wavefunction for the coin step."
      },
      {
        "name": "coin_matrix",
        "type": "tensor",
        "tensor_shape": "(8,8)",
        "description": "Coin operator to be applied."
      },
      {
        "name": "psi_out",
        "type": "calculated",
        "tensor_shape": "(ny,nx,8)",
        "formula": "MATMUL(psi_in, TRANSPOSE(coin_matrix))",
        "description": "Applies the coin operator to each spin component."
      }
    ]
  },
  {
    "name": "ShiftStep",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Input wavefunction for the spatial shift."
      },
      {
        "name": "offsets",
        "type": "array",
        "items": "tuple(int,int)",
        "description": "List of (dy,dx) offsets for each direction index (0..7)."
      },
```

```
      {
        "name": "psi_out",
        "type": "calculated",
        "tensor_shape": "(ny,nx,8)",
        "formula": "SHIFT(psi_in, offsets)",
        "description": "Rolls each direction's amplitude by the specified (dy,dx) offsets."
      }
    ]
  },
  {
    "name": "BarrierStep",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Input wavefunction before barrier is applied."
      },
      { "name": "barrier_row",   "type": "number", "description": "Row index of the barrier." },
      { "name": "slit1_xstart",  "type": "number", "description": "Slit #1 start column." },
      { "name": "slit1_xend",    "type": "number", "description": "Slit #1 end column." },
      { "name": "slit2_xstart",  "type": "number", "description": "Slit #2 start column." },
      { "name": "slit2_xend",    "type": "number", "description": "Slit #2 end column." },
      {
        "name": "psi_out",
        "type": "calculated",
        "tensor_shape": "(ny,nx,8)",
        "formula": "APPLY_BARRIER(psi_in, barrier_row, slit1_xstart, slit1_xend, slit2_xstart, slit2_xend)",
        "description": "Zero out barrier row except in the slit columns."
      }
    ]
  },
  {
    "name": "CollapseBarrierStep",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Input wavefunction before measurement collapse at barrier."
      },
      { "name": "barrier_row",   "type": "number", "description": "Barrier row index (where measurement occurs)." },
      { "name": "slit1_xstart",  "type": "number", "description": "Slit #1 start column." },
      { "name": "slit1_xend",    "type": "number", "description": "Slit #1 end column." },
      { "name": "slit2_xstart",  "type": "number", "description": "Slit #2 start column." },
      { "name": "slit2_xend",    "type": "number", "description": "Slit #2 end column." },
      {
        "name": "psi_out",
        "type": "calculated",
        "tensor_shape": "(ny,nx,8)",
        "formula": "COLLAPSE_BARRIER(psi_in, barrier_row, slit1_xstart, slit1_xend, slit2_xstart, slit2_xend)",
        "description": "Implements a barrier measurement collapse: amplitude outside slits is lost."
      }
    ]
  },
  {
    "name": "WavefunctionNorm",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Wavefunction whose norm we want to compute."
      },
      {
        "name": "total_norm",
```

```json
        "type": "calculated",
        "formula": "SUM(ABS(psi_in)^2)",
        "description": "Computes the total probability norm: sum(|psi|^2)."
      }
    ]
  },
  {
    "name": "DetectorAmplitude",
    "fields": [
      {
        "name": "psi_in",
        "type": "tensor",
        "tensor_shape": "(ny,nx,8)",
        "description": "Wavefunction to extract the detector row from."
      },
      {
        "name": "detector_row",
        "type": "number",
        "description": "Row index where the detector is located."
      },
      {
        "name": "row_amp",
        "type": "calculated",
        "tensor_shape": "(nx,8)",
        "formula": "SLICE(psi_in, axis=0, index=detector_row)",
        "description": "Extracts the wavefunction's amplitude at the detector row."
      }
    ]
  },
  {
    "name": "DetectorIntensity",
    "fields": [
      {
        "name": "row_amp",
        "type": "tensor",
        "tensor_shape": "(nx,8)",
        "description": "Detector row amplitude over x, with 8 spin directions."
      },
      {
        "name": "intensity_1d",
        "type": "calculated",
        "formula": "SUM(ABS(row_amp)^2, axis=-1)",
        "description": "Sums |amplitude|^2 over spin directions, yielding intensity profile vs. x."
      }
    ]
  },
  {
    "name": "QWalkRunner",
    "fields": [
      {
        "name": "steps_to_barrier",
        "type": "number",
        "description": "Number of steps taken before potentially measuring at the barrier."
      },
      {
        "name": "steps_after_barrier",
        "type": "number",
        "description": "Number of steps taken after the barrier event."
      },
      {
        "name": "collapse_barrier",
        "type": "boolean",
        "description": "If true, measure/collapse at the barrier; otherwise let the wave pass."
      },
      {
        "name": "final_wavefunction",
```

```
        "type": "calculated",
        "formula": "EVOLVE(WavefunctionInitial.psi_init, steps_to_barrier, steps_after_barrier, collapse_barrier)",
        "description": "Resulting wavefunction after the prescribed sequence of steps and optional barrier collapse."
      }
    ]
  }
]
```

Appendix B: 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.
```

```python
    """
    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):
        """
```

```python
        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

        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
```

```python
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")
```

```python
    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))
    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__":
```