RUSTLAB 2026 • BOLOGNA • NOV 03, 12:15

Building a Declarative Computation Engine in Rust

ABOUT THE SPEAKER

Kostas Passadis

Electrical & Computer Engineer
PhD in Electromagnetics

Section Head at NCC, Department of IT & AI

National Control Center • IPTO

Independent Power Transmission Operator.

SESSION OUTLINE

Presentation Roadmap

01

The Problem

Policy vs. mechanism and enterprise release cycles.

02

Overview

The 4-stage lifecycle from text DSL to rust.

03

Components

AST, graph definition, compilation, runtime execution.

04

Use Cases

Async HTTP, 50 Hz Kalman DSP, and wind curves.

05

Road Ahead

Algorithmic implementation, Error handling, graph optimizations

FOUNDATIONAL PHILOSOPHY

Declarative + Engine = Computation

A model is not a computer program. It is a declarative representation capturing how variables interact within our world.

ORIGINS

Domain Expert

Human elicitation: physical laws, rules, and equations.

Statistical Model

Machine learning: parameter estimation from data.

Declarative Core

THE MODEL

A standalone description of reality. Pure variable relationships.

- Pure Topology
- No Control Flow
- Solver Agnostic

Rust Infrastructure

THE ENGINE

- DAG Toposort
- Shape Inference
- Memory Arena

Execution Driver

ALGORITHM

- Forward Sweep
- SPSA Optimizer
- Exact Inference

Execution Result

COMPUTATION

The Declarative Equation: Model (Reality) + Engine + Algorithm = Computation (Results).

THE CORE CHALLENGE

Can We Make Domain Knowledge Executable?

1. Whiteboard Math

Domain experts: express equations, regulatory rules and requirements.

2. IT Implementation

The IT development cycle slows the path from equations/requirements to implementation.

3. Slow Change Cycle

Tuning one coefficient triggers another full release sprint.

> *"Settlement: 'We need to implement y = x + z.' — IT: 'Gosh, that requires databases, REST, authentication, and microservices. It will take 6 months.'"*

THE ITERATION DILEMMA

Requirements Rarely Work In One Pass

TRADITIONAL FEEDBACK LOOP

Delayed Feedback

1. Expert Clarifies → 2. Dev Revises Code → 3. Release → 4. Gap Discovered

DECLARATIVE ENGINE LOOP

Immediate Feedback

1. Expert Updates Model → 2. Engine Verifies Model → 3. Execution → 4. Direct Validation

Two motivations: Improve efficiency by moving directly from definition to production, bypassing intermediate implementation steps. Build a unified engine to execute logic across different domains.

MULTI-STAKEHOLDER ECOSYSTEM

One Engine to Rule Them All

Shared execution infrastructure. Domain-specific definitions.

Operational Dispatch

National Control Center

50 Hz PMU streaming, Kalman filter state estimation & CUSUM anomaly detection.

Market & Settlement

Balancing Operations

aFRR energy balancing rules, merit order list sorting & penalty multiplier logic.

Renewable Forecasting

Analytics & RES

Fourier logistic wind power curves, Open-Meteo REST ingest & SPSA optimization.

Compliance & Audit

ENTSO-E / OPDE

Bayesian causal factor networks modeling daily IGM non-publication risk.



Shared Engine Runtime (brick-rs)

Static Topological Verification • Contiguous Float Arena (Vec) • Graph Execution

INTERFACE SEPARATION

Two Languages, One Engine

definition.dsl

Compiled Once

Definition DSL

What the computation is



The computational engine

runtime.dsl

Queried Many

Execution DSL

What to run, with what data

```
EXECUTE ... USING ...
GIVEN ... YIELD ...
```

Feeds data into pre-allocated memory slices.

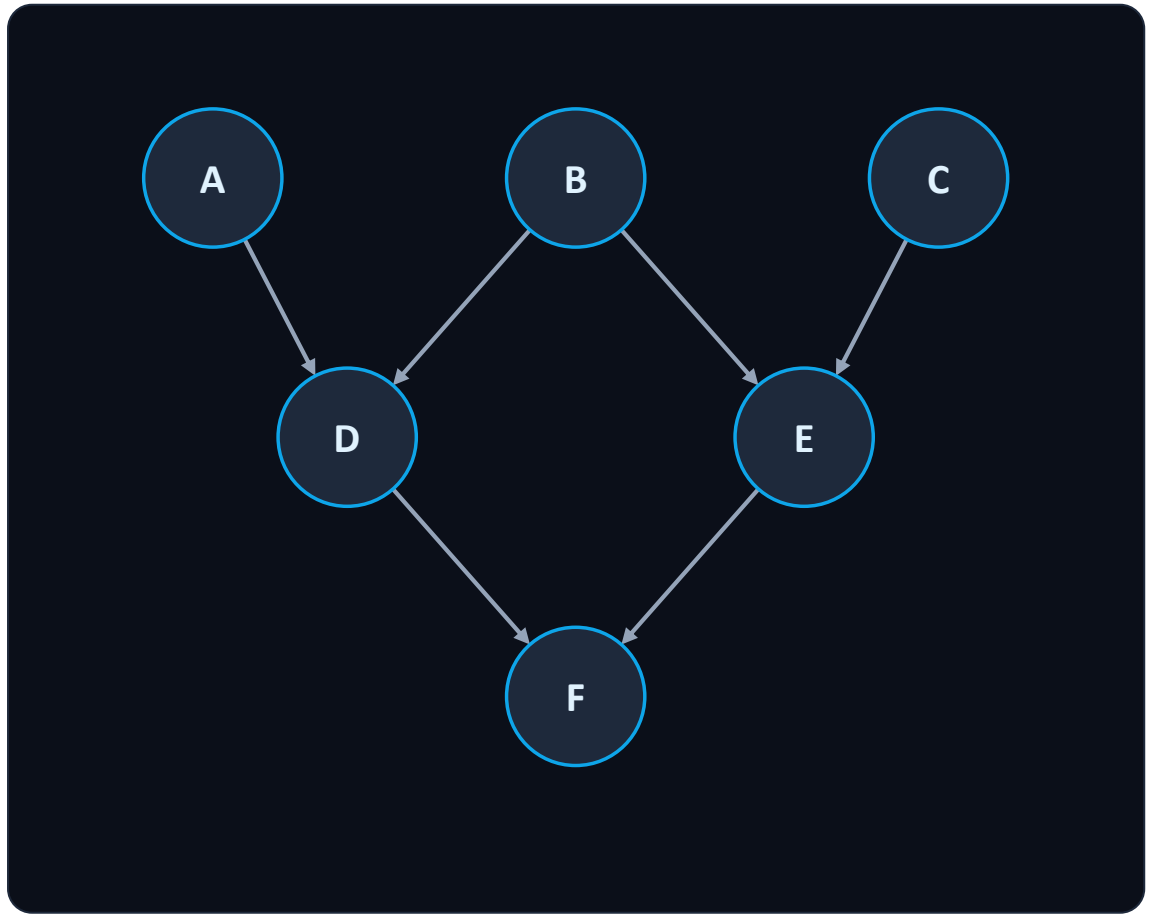
One Model File. Many Runs.

Same model: different streaming data, different algorithms and different YIELD targets

INTERNAL REPRESENTATION

The Unifying Internal Representation

Synthesizing Computer Science and ML/Statistics



1. Computer Science & Compilers

Directed Acyclic Graphs (DAGs), topological sorting, static memory layout allocation

2. Statistics & Machine Learning

Bayesian networks [Child | Parents], bipartite factor graphs, joint probability distributions

The Internal Representation: Graphs provide the single mathematical abstraction where compiler optimization, probabilistic reasoning, and physical modeling coexist seamlessly.

EXECUTION LIFECYCLE

The Workflow: From definition to execution

01 • INGEST

definition.dsl

parse
nom — text → GraphAst

build graph
petgraph — nodes, ports, state transitions

→

02 • VERIFY

validate

- **DAG:** toposort
- **connectivity:** disconnected components?
- **arity:** producer types & port contracts
- **state-transition rules:** recurrence bounds

→

03 • LOWER

shape inference
every dimension proven

arena layout
one Vec, static offsets

CompiledGraph + ExecutionContext

→

04 • RUN

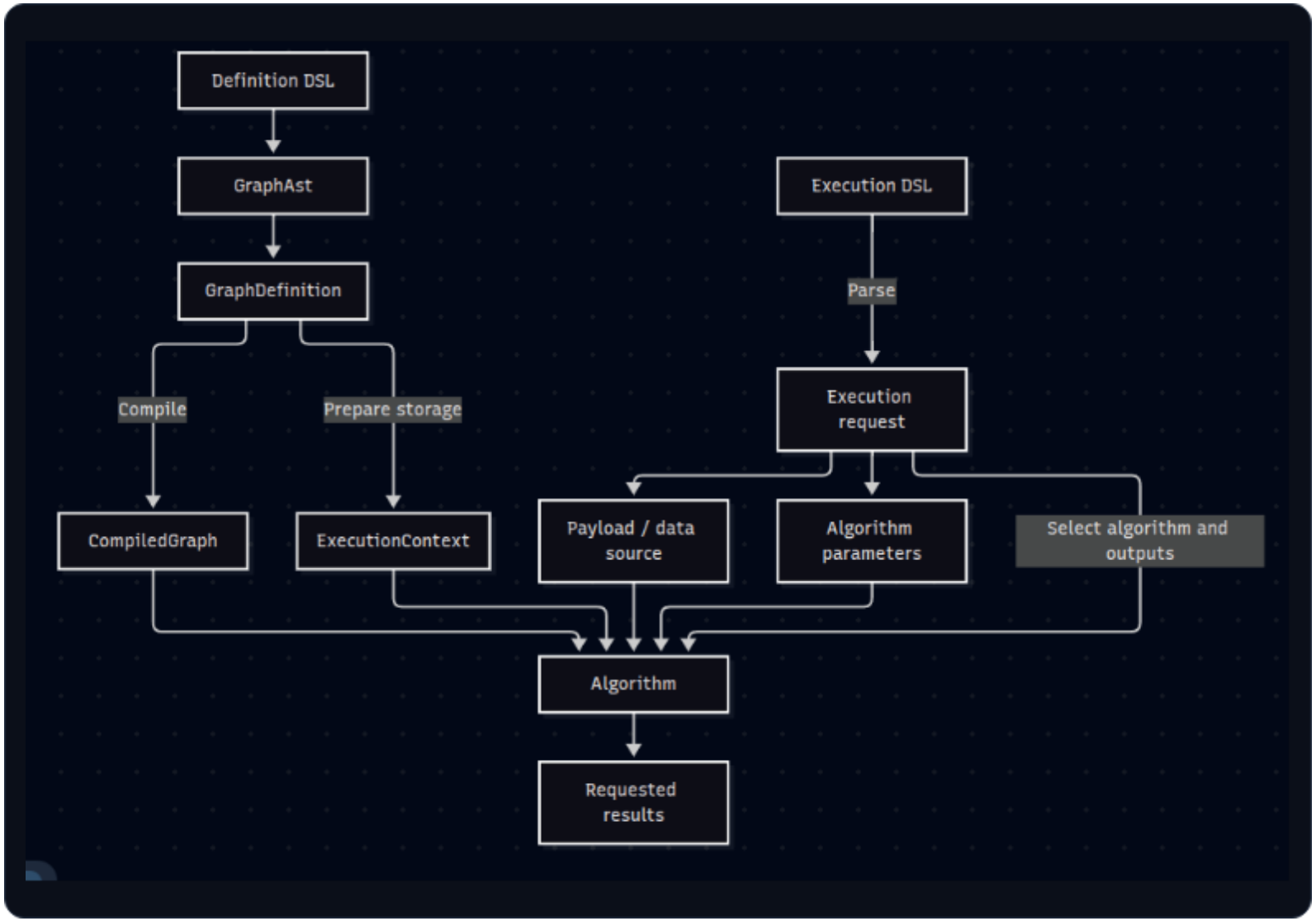
runtime.dsl

—▶ **Algorithm**
—▶ **kernels**
—▶ **arena**

FULL PIPELINE TRACK: definition.dsl —▶ nom —▶ petgraph —▶ validate —▶ shapes —▶ arena —▶ runtime.dsl —▶ kernels

SYSTEM ARCHITECTURE & LIFECYCLE

Component Interaction: Definition to Execution



01 • GRAPH DEFINITION — Cold Phase

Specification Pipeline

Compiled into immutable graph and an execution context.

02 • EXECUTION REQUEST — Runtime

Work Order Ingestion

Parsed into payloads, algorithm, and parameters.

03 • ALGORITHM ENGINE — Hot Loop

Iterative Algorithm Support

Input data streams evaluate to requested results.

AST OPERATOR TAXONOMY

Operations Are a Sum Type

Operators

```rust
pub enum Op {
    Math(MathOp),
    Http(HttpOp),
    Sink(SinkOp),
    Probabilistic(ProbabilisticOp),
    Decision(DecisionOp),
    Input(InputOp),
}
```

```rust
pub enum InputOp {
    Placeholder(Shape),
    Text,
    Constant(Shape),
    Variable(Shape),
}
```

Operation & Configuration

Each variant identifies a supported mathematical operation and encapsulates its static parameters.

Shape Declarations & Inference

Placeholders declare explicit input dimensions; downstream transforms enforce static shape-inference rules.

Exhaustive Matching

Compiler passes and execution engines match exhaustively, guaranteeing every supported variant is handled.

COMPOSITION OVER INHERITANCE

An Operation Becomes a Named Node

The Rust AST Wrapper

```
pub struct AstNode { pub label: String, pub op: Op, }
```

THE SPECIFICATION DSL STATEMENT

Subtract[label = difference]

Operation vs. Identity

Op describes the computational kernel;

AstNode assigns that occurrence a unique symbolic identity.

Multi-Instance Reusability

Multiple nodes instantiate the exact same operation with distinct labels, operands, and downstream targets.

THE DSL SPECIFICATION EVOLUTION

Initially, There Was JSON

Why We Started With JSON

Trivial serde serialization/deserialization. Ingested into Rust in five lines of code without building custom lexers or parsers.

Where JSON Failed Us

Verbose: 50 lines of braces and quotes to express a 3-step calculation.

Counter-Intuitive Flow: Edges detached in a distant list; impossible to visually scan dependencies.

The Fatal Trap: Mixes definition with runtime data. Users embed initial weights and streaming numbers directly in the blueprint.

PIPELINE_V1.JSON (ANTI-PATTERN) — Data Mixed With Definition

```
{
 "name": "wind_estimator",
 "nodes": [
  { "label": "v", "op": "Placeholder", "shape": [1, 1] },
  {
   "label": "M_0", "op": "Variable", "shape": [1, 1],
   "initial_value": [[8.2]]
  },
  { "label": "delta_M", "op": "MatMul" },
  { "label": "M_theta", "op": "Add" }
 ],
 "edges": [
  { "from": "w_M", "to": "delta_M" },
  { "from": "M_0", "to": "M_theta" },
  { "from": "delta_M", "to": "M_theta" }
 ]
}
```

A specification format must declare the factory layout, not store the goods passing through it.

THE SPECIFICATION LANGUAGE

A DSL Inspired by Conditional Probabilities

Pearl's Bayesian Notation: P(Child | Parents)

In Bayesian networks, dependencies are written as child | parent1, parent2. We adopted this intuition for DAG topology.

The Two Structural Blocks

1. Node Declarations: Operators, labels, and static shapes. Pure identity without connection clutter.

2. Topology Block: A single scannable bracketed block at the bottom defining all dependencies and port orderings.

Pure Blueprint: Contains zero runtime data. Streaming values enter strictly at query time via GIVEN JSON.

Clean & Scannable

```
DAG[label = wind_power_estimator]
### 1. NODE DECLARATIONS (Types & Shapes)
Variable[label = P_0, rows = 1, columns = 1]
Variable[label = w_M, rows = 1, columns = 4]
Variable[label = M_0, rows = 1, columns = 1]
Placeholder[label = v, rows = 1, columns = 1]
Placeholder[label = theta_basis, rows = 4, columns = 1]
MatMul[label = delta_M]
Add[label = M_theta]
### 2. TOPOLOGY (Child | Parents Notation)
[ P_0; w_M; M_0; v; theta_basis; delta_M | w_M, theta_basis; # Port 0: w_M, Port 1: theta
M_theta | M_0, delta_M ] # Port 0: M_0, Port 1: delta
```

DIRECTED DATAFLOW SEMANTICS

Dependencies Have Ordered Input Ports

DSL Dependency Declaration

```
[difference | a, b]
```

Dependency Edge	Edge Weight	Positional Meaning
a → difference	0	Left operand (minuend)
b → difference	1	Right operand (subtrahend)

Non-Commutative Algebraic Calculations

Edge weights preserve parameter ordering during AST construction, guaranteeing unambiguous matrix subtractions and multiplications without runtime lookup overhead.

RUNTIME QUERY SPECIFICATION

The Execution DSL: Selecting Graph, Algorithm & Outputs

```
EXECUTE paper
USING Forward
GIVEN JSON
{
  "x": [[50.0], [0.0], [0.0]],
  "P": [[1, 0, 0], [0, 1, 0], [0, 0, 1]],
  ...
}
YIELD ()
```

Clause	Meaning
EXECUTE paper	Select defined computation graph
USING Forward	Select execution algorithm
GIVEN JSON	Bind supplied values to named placeholders
YIELD (...)	Select result node labels to return

Training DSL: Algorithms & Sources

Clause	Responsibility
USING SPSA	Select the algorithm
WITH SPSA[…]	Configure its parameters
GIVEN SOURCE …	Read input JSON data from a local folder
YIELD loss	Return the result identified by loss

```
EXECUTE wind_power_trainer
USING SPSA
WITH SPSA[
  iterations=1000,
  a=0.5, c=0.1, A=10,
  alpha=0.602, gamma=0.101
]
GIVEN SOURCE ./data/train
YIELD loss
```

PARSER COMBINATORS IN NOM

Parsers as Little Monsters



> *"Think of parsers as little monsters. Each recognizes and consumes a piece of text, then spits out a parsed value and the remaining input. Parser combinators let us arrange these monsters into a parser for the whole language."*

A PARSER RETURNS

IResult<&str, T>

ON SUCCESS

(remaining_input, parsed_value)

ON FAILURE

a parsing error

Parsing Key-Value Pairs

RUST IMPLEMENTATION (NOM)

```rust
pub fn parse_kv_pair(
    input: &str,
) -> IResult<&str, (&str, &str)> {
    separated_pair(
        parse_any,
        ws(tag("=")),
        parse_any,
    )
    .parse(input)
}
```

INPUT SLICE

`seed=42`

PARSED OUTPUT

`("seed", "42")`

REMAINING SLICE

`""` (entire slice consumed)

COMBINATOR ORCHESTRATION

The Runtime Parser: Assembling nom

PARSE_RUNTIME IMPLEMENTATION

```rust
pub fn parse_runtime(
    input: &str,
) -> IResult<&str, (&str, &str, Payload, &str, Option<Vec<Param>>)> {
    let (input, dag)    = parse_dag_label(input)?;
    let (input, algo)   = parse_algorithm(input)?;
    // 3. Algorithm parameters
    let (input, params) = parse_algorithm_parameters(input)?;
    let params: Option<Vec<Param>> = params.map(|params| {
        params
            .into_iter()
            .flat_map(|tuple| {
                let param: Result<Param, _> = tuple.try_into();
                if let Ok(param) = param {
                    Some(param)
                } else {
                    None
                }
            })
            .collect::<Vec<Param>>()
    });

    let (input, given) = alt((parse_given_json, parse_given_source)).parse(input)?;
    let (input, out)   = parse_yield_label(input)?;

    Ok((input, (dag, algo, given, out, params)))
}
```

1. EXECUTE & USING

Target blueprint and algorithm driver resolution.

2. WITH Parameters

Typed hyperparameter vector conversion.

3. GIVEN Ingestion (alt)

Dual branching: inline JSON vs folder source.

4. YIELD Sink

Output node label extraction.

PARSER OUTPUT AST

(dag, algo, given, out, params)

GRAPH DEFINITION ANATOMY

Nodes & Connections Form the Definition

GRAPH AST CONTAINER

```
pub struct GraphDefinition {
pub label: String,
pub symbol_table: HashMap<String, usize>,
pub nodes: Vec<AstNode>,
pub graph: PGraph<(), usize, Directed>,
pub state_transitions: Vec<StateTransition>,
}
```

Mirrors the DSL's two sections: node declarations and dependency topology.

Field	Responsibility
label	Identifies the computation
symbol_table	Resolves string labels to node indices
nodes	Stores named operations (AstNode)
graph	Stores dependencies and input ports
state_transitions	Stores state updates between execution steps

> **Acyclic Preservation:** State transitions live separately from dependency edges, preserving strict DAG properties within each evaluation step.

```
pub graph: PGraph<(), usize, Directed>
```

$$\text{Graph} < \underbrace{N}_{\text{Node Weight / Payload}} , \underbrace{E}_{\text{Edge Weight / Payload}} , \underbrace{Ty}_{\text{Directed/Undirected}} >$$

LOWERED EXECUTION IR

CompiledGraph: Execution Blueprint

compiled_graph.rs — Rust IR

```rust
pub struct CompiledGraph {
    pub label: String,
    pub symbol_table: HashMap<String, usize>,
    pub nodes: Vec<CompiledNode>,
    pub graph: PGraph<(), usize, petgraph::Directed>,
    pub execution_order: Vec<usize>,
    pub state_transition: Vec<(usize, usize)>,
}
```

Field & Type	Engine Responsibility
label: String	Model identifier for telemetry and logging
symbol_table	O(1) resolution from DSL labels to slots
nodes: Vec	Flat array of lowered ops and arena layouts
graph: PGraph<(), usize, …>	Topology with edge weights as port indices
execution_order	Topological schedule for linear evaluation
state_transition	Recurrence modelling across time steps (unfold recurrent connections)

EXECUTION INTERMEDIATE REPRESENTATION

CompiledNode: Ready for Execution

compiled_node.rs — Rust IR

```rust
pub struct CompiledNode {
    pub label: String,
    pub op: Op,
    pub node_slot: usize,
    pub arena_layout: Option<BufferLayout>,
    pub parents: Vec<usize>,
}
```

parents: Vec — Zero-Hop Inputs

Precomputed parent indices in input-port order, avoiding topology traversal during execution.

node_slot: usize — Dynamic Storage

An index into the context's dynamic storage (ctx.data and ctx.tables).

arena_layout: Option — Tensor Slice

Numerical storage metadata, when applicable. Maps directly to contiguous arena coordinates.

MEMORY ARCHITECTURE

BufferLayout: Where a Tensor Lives

buffer_layout.rs

Copy IR

```rust
pub struct BufferLayout {
    pub offset: usize,
    pub rows: usize,
    pub columns: usize,
}
```

1. Arena Offset (offset)

Points to the starting element in the pre-allocated contiguous Vec memory arena.

2. Matrix Geometry (rows, columns)

Defines the 2D tensor dimensions for zero-copy column-major views (faer::MatRef / MatMut).

3. Zero-Allocation Slicing

Slices &arena[offset..offset + rows * columns] directly, eliminating heap allocations.

MEMORY SLICING EXAMPLE

BufferLayout in Practice

```
BufferLayout { offset: 6, rows: 3, columns: 3, }
```

&arena[6..15]

9 consecutive f64 values in the contiguous arena

Arena index	6	7	8	9	10	11	12	13	14
Value	**1.0**	4.0	7.0	2.0	**5.0**	8.0	3.0	6.0	**9.0**

Those nine values represent:

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

3×3 Matrix (Column-Major)

RUNTIME STATE CONTAINER

ExecutionContext: Memory for the Compiled Computation

context.rs — Mutable State

```rust
pub struct ExecutionContext {
    pub arena: Vec<f64>,
    pub tables: Vec<Option<FactorTable>>,
    pub data: Vec<Option<Value>>,
}
```

Storage	Purpose
Numerical arena	Preallocated contiguous Vec for matrices and intermediate numerical results
Factor tables	Storage for Bayesian factors
JSON values	Structured data used by non-numerical operations

The compiled graph describes the operations and storage layouts. The execution context owns the working data and state.

ZERO-COPY TENSOR ACCESS

An Arena Slice Becomes a Matrix View

get_matrix(&self, layout: &BufferLayout) faer::MatRef

```
let slice = &self.arena[
    layout.offset..layout.offset + layout.len()
];

MatRef::from_column_major_slice(
    slice,
    layout.rows,
    layout.columns,
)
```

1. Slice the Contiguous Slab

Bounds resolved in O(1) via precomputed offset and length.

2. Project Column-Major Strides

Wraps raw float slice in a 2D matrix view header.

A matrix view borrows existing arena storage without copying the values.

DISJOINT ARENA ACCESS

Borrowing Inputs and Output Together

BINARY MATRIX OPERATION

$$C = AB$$

Simultaneous Borrow Requirement:

Read-only views: Immutable access to A and B.

Writable view: Mutable access to destination C.

Single slab: All backed by the same numerical arena.

get_matrix_triple — Disjoint View Construction — Zero-Allocation

```rust
pub fn get_matrix_triple<'a>(
    &'a mut self, src1: &BufferLayout, src2: &BufferLayout, dst: &BufferLayout,
) -> (MatRef<'a, f64>, MatRef<'a, f64>, MatMut<'a, f64>) {
    let s1 = (src1.offset, src1.offset + src1.len());
    let s2 = (src2.offset, src2.offset + src2.len());
    let d = (dst.offset, dst.offset + dst.len());
    let overlaps = |a: (usize, usize), b: (usize, usize)| a.0 < b.1 && b.0 < a.1;
    debug_assert!(!overlaps(s1, s2) && !overlaps(s1, d) && !overlaps(s2, d));
    let arena_ptr = self.arena.as_mut_ptr();
    unsafe {
        let src1_slice = std::slice::from_raw_parts(arena_ptr.add(src1.offset), src1.len());

        let src2_slice = std::slice::from_raw_parts(arena_ptr.add(src2.offset), src2.len());

        let dst_slice = std::slice::from_raw_parts_mut(arena_ptr.add(dst.offset), dst.len());

        let a = MatRef::from_column_major_slice(src1_slice, src1.rows, src1.columns);
        let b = MatRef::from_column_major_slice(src2_slice, src2.rows, src2.columns);
        let c = MatMut::from_column_major_slice_mut(dst_slice, dst.rows, dst.columns);
        (a, b, c)
    }
}
```

COMPILER PIPELINE • PHASE 1

Compilation: Checking the Computation

compiler_verification.rs — Static Passes

```rust
// Step 1: Topological / Cycle verification
let topo_order = Self::validate_dag(&dag)?;

// Step 2: Component integrity check
if !dag.is_single_component() {
    let n = dag.get_components().len();
    errors.push(CompilationError::MultipleComponents { .. });
}

// Step 3: Domain semantic validation
let _ = dag.semantic_validation()?;

// Step 4: State transition validation
Compiler::validate_state_transition(&dag, &mut errors, &shapes);
```

Stage	Compile-Time Invariant
1. Cycle Verification	DFS toposort aborts immediately if circular dependencies exist.
2. Component Check	Verifies graph connectivity, catching orphaned nodes or isolated partitions.
3. Semantic Validation	Enforces operator arity, valid producer types, and port connection rules.
4. State Transitions	Guarantees temporal recurrences carry matching tensor shapes across steps.

COMPILER PIPELINE • PHASE 2

Compilation: Preparing Execution

TOPOLOGICAL LOWERING LOOP

```
for idx in &topo_order {
// 1. Infer and check output shape
// 2. Assign numerical arena region
// 3. Record ordered parent indices
// 4. Construct the CompiledNode }
```

1. Infer and Check Output Shape

Propagates dimensions forward; verifies inner matrix algebra compatibility.

2. Assign Numerical Arena Region

Allocates contiguous BufferLayout; advances memory offset by rows × columns.

3. Record Ordered Parent Indices

Sorts inputs by edge port weight to ensure canonical LHS/RHS ordering.

4. Construct CompiledNode

Packs lowered operator, dynamic slot, and memory layout into immutable node.

MEMORY ALLOCATION • CONCRETE EXAMPLE

Layout Assignment: Matrix Multiplication

ALGEBRAIC OPERATION $A_{2 \times 3} \; B_{3 \times 4} = C_{2 \times 4}$

1. DIMENSION CHECK

Inferred Output

Eight f64 elements (2 × 4 matrix).

2. BUFFER LAYOUT

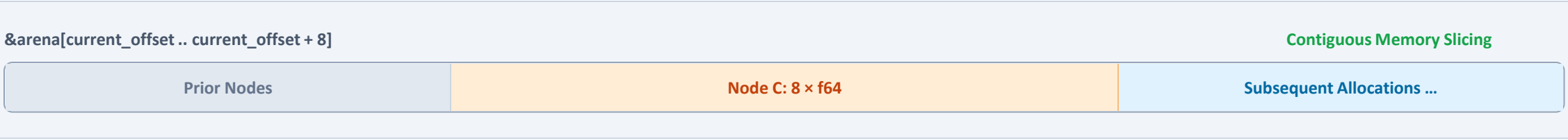
Assigned Layout

Current offset, 2 rows, 4 columns.

3. OFFSET ADVANCE

Next Free Offset

Current offset + 8 elements.



EXECUTION ABSTRACTION

Algorithm: A Common Execution Contract

```rust
#[async_trait]
pub trait Algorithm {
    type Input;
    type Output;
    type Error;
    type Graph;
    type Config;

    async fn execute(
        &self,
        graph: Self::Graph,
        ctx: &mut ExecutionContext,
        input: Self::Input,
        config: Self::Config,
    ) -> Result<Self::Output, Self::Error>;
}
```

1. Dialect Specialization

Associated types bind graph formats without vtables.

2. Direct Context Borrow

Mutation is localized to &mut ExecutionContext.

3. Reentrant & Thread-Safe

Graph can be shared via Arc across concurrent workers.

TRAIT SPECIALIZATION

Different Algorithms, Different Requirements

Associated Type	Init	DSeparation
Graph	`Arc<CompiledGraph>`	`Arc<FactorizedGraph>`
Input	`serde_json::Value`	`Payload`
Output	`()`	`bool`
Config	`()`	`()`

ALGORITHM IMPLEMENTATIONS

Init: Loading Values into Context

```rust
pub struct Init;

#[async_trait]
impl Algorithm for Init {
    type Error = RuntimeError;
    type Input = Value;
    type Output = ();
    type Config = ();
    type Graph = Arc<CompiledGraph>;

    async fn execute(
        &self,
        graph: Self::Graph,
        ctx: &mut ExecutionContext,
        input: Self::Input,
        _config: Self::Config,
    ) -> Result<Self::Output, Self::Error> {
        ctx.write_json_to_arena(input, &graph);
        Ok(())
    }
}
```

The Single Operation

`ctx.write_json_to_arena` binds raw JSON payloads to preallocated arena slices (shapes must match).

Symbol Resolution

Matches JSON keys against symbol_table without scanning nodes.

Zero Subsequent Allocations

Once seeded by Init, execution runs with zero heap churn.

USE CASE 1 • PARALLEL HTTP FETCH

A Graph Can Describe an HTTP Request

> **The Core Challenge: The computational engine is designed to do far more than numerical linear algebra.**

```
DAG[label = openmeteo]
RequestUrl[label = openmeteo]
RequestParam[label = longitude, name = longitude]
RequestParam[label = latitude, name = latitude]
RequestParam[label = hourly, name = hourly]
Text[label = longitude_value]
Text[label = latitude_value]
Text[label = hourly_value]
Text[label = url]
Get[label = get]
[longitude|longitude_value; latitude|latitude_value
 hourly|hourly_value; openmeteo|url;
 get|openmeteo,longitude,latitude,hourly]
```

1. Heterogeneous I/O in the Graph

Network calls are expressed as standard nodes in the topological graph without altering compiler invariants.

2. Dynamic String Storage

Text and URL parameters live in dynamic context slots, bypassing the contiguous numerical float arena.

3. Deterministic Dependency Wiring

The dependency syntax maps parent port order directly into integer indices for zero-overhead execution.

The Execution DSL: Parallel Batch Semantics

```
EXECUTE openmeteo
USING ParallelFetch
GIVEN JSON
{
  "url": "https://api.open-meteo.com/v1/forecast",
  "longitude_value": [52.52, 52.33],
  "latitude_value": [13.41, 42.0],
  "hourly_value": "temperature_2m"
}
YIELD get
```

Clause / Token	Semantic Role
openmeteo	Label of the compiled DAG blueprint to execute
ParallelFetch	Algorithm coordinating concurrency and batching
GIVEN JSON	Supplies broadcast scalars and vectorized coordinate arrays
get	Label of the sink node whose payload is extracted and yielded

> **Batch Axis Detection:** Array inputs define the parallel width (N = 2), while scalar strings broadcast invariant across all instances.

USE CASE 1 • PARALLEL HTTP FETCH

The Parallel Operation Challenge

> “The graph stays fixed while the algorithm controls how many executions occur and how they are coordinated.”

PARALLEL TASK SPAWNING IN RUST

```rust
for (i, mut fork_ctx) in contexts.into_iter().enumerate() {
    let graph_ref = graph.clone();

    join_set.spawn(async move {
        let output = HttpFetch
            .execute(graph_ref, &mut fork_ctx, (), ())
            .await;

        (i, output)
    });
}
```

Zero Graph Duplication

Arc is immutable; cloning incurs only an O(1) atomic pointer bump.

Context Isolation

Each branch mutates its own forked ExecutionContext without locks or race conditions.

Order-Preserving Aggregation

Returning index i maps asynchronous completions back to the caller's array ordering.

USE CASE 2 • GRID DISPATCH & CURTAILMENT

Wind Power: Estimating Available Production

Over-Generation

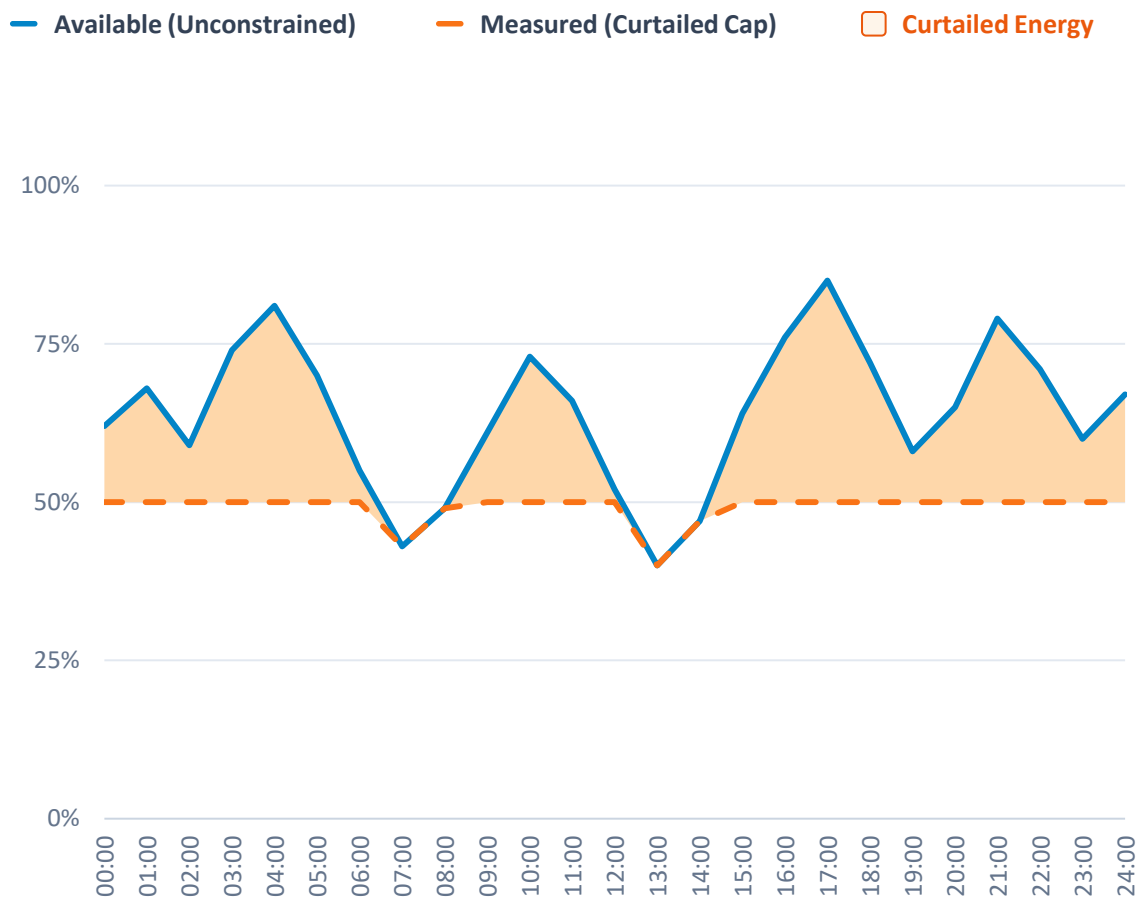
During periods of excess generation, the NCC may need to curtail wind production.

Restricted Output Telemetry

Measured output reflects the restriction rather than true aerodynamic capability.

SCADA Reconstruction

Historical SCADA data helps estimate production from wind speed and direction.



Illustrative profile (% of rated power), not measured data

CENTRAL QUESTION

How much power could the wind park have produced without curtailment?

PARAMETRIC AERODYNAMIC MODEL

A Direction-Dependent Power Curve

$$\hat{P}(v,\theta) = \frac{P_{max}(\theta)}{1 + \exp\left(-B(v - M(\theta))\right)}$$

Wind Speed (v)

Wind speed determines position along the curve.

Wind Direction (θ)

Wind direction adjusts its maximum and midpoint.

MSE Optimization

Trainable parameters fitted by minimizing mean squared error.

Fourier Harmonics: Directional wake deficits and terrain sheltering modulate parameters with zero angle discontinuities.

LIFECYCLE ORCHESTRATION

Initialize, Train, Predict

Phase 01

Initialize

```
USING Init GIVEN JSON { ... } YIELD ()
```

Parameter priors and structural scalars into context.

Phase 02

Train

```
USING SPSA WITH SPSA[iterations=1000, ...] GIVEN SOURCE ./data/train YIELD loss
```

Optimizes variable slices in-place

Phase 03

Predict (Forward)

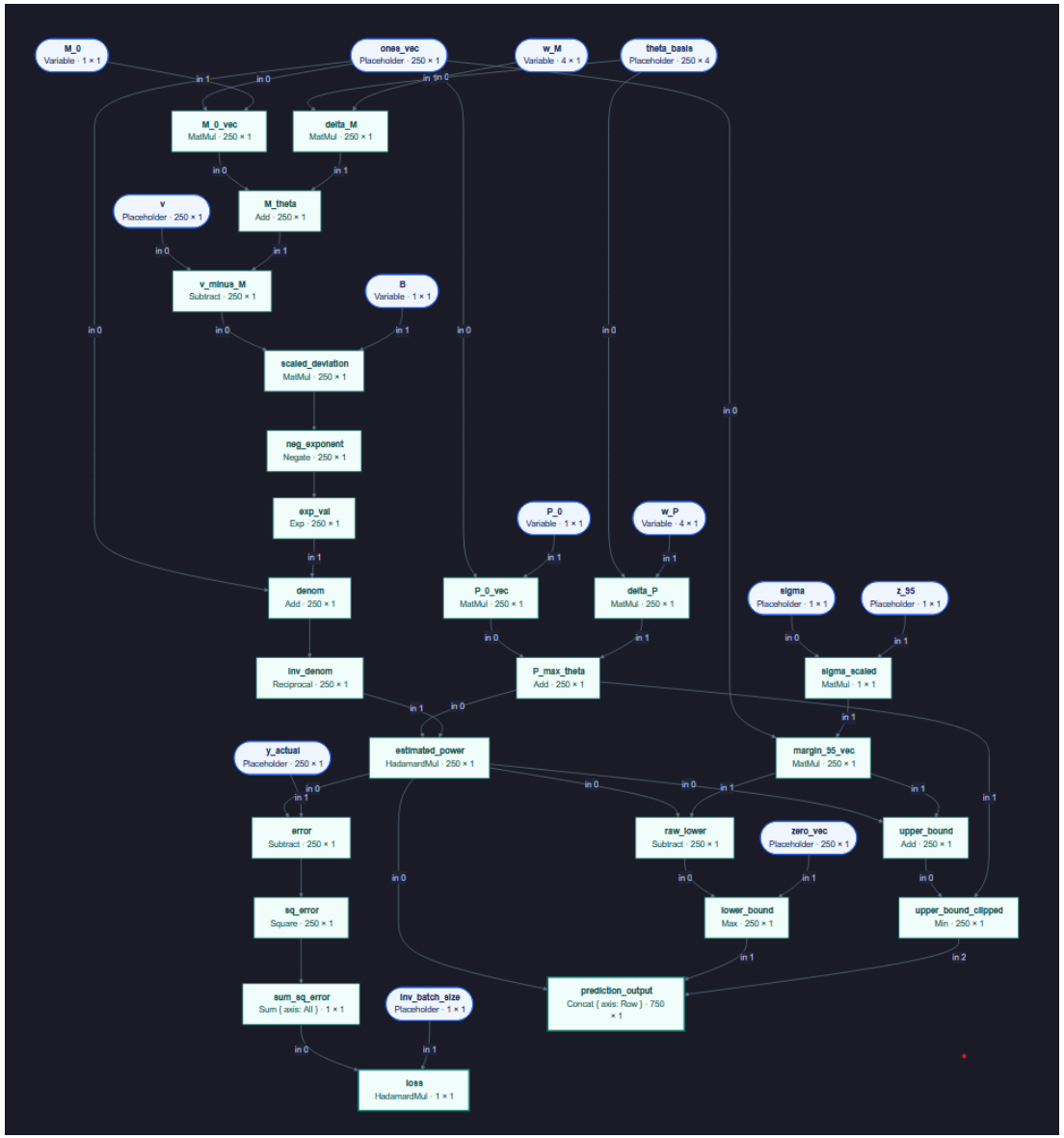
```
USING Forward GIVEN JSON { ... } YIELD estimated_power
```

Evaluates streaming telemetry with pruned downstream loss branches.

One graph definition, reused across the model's lifecycle

Execution order pruning

Output Determines Execution



Training: YIELD loss — Exclude prediction band

Execute prediction path and loss calculation.

Prediction: YIELD estimated_power — Exclude loss estimation

Execute only prediction dependencies. Actual production values are unnecessary.

```
let pruned = graph.pruned_execution_order(output_node_id.index());
```

The graph stays fixed. The selected execution path changes.

USE CASE 3 • HIGH-FREQUENCY STREAMING TELEMETRY

Processing PMU Measurements Every 20 ms

Continuous Ingestion Engine

Building a real-time system to collect and process synchronized Phasor Measurement Unit (PMU) streams.

High-Resolution Grid Samples

Each 50 Hz sample arrives every 20 milliseconds, delivering voltage, current, frequency, and RoCoF.

Disturbance Detection

Detect abrupt frequency shifts and forced oscillations as data arrives.

Time constraints: samples arrive every 20 ms



PRODUCTION INTEGRATION

From Research to Streaming Computation



Power System Frequency Disturbance Detection – A Data-Driven Statistical Process Control

Spiros Chountasis
ECC- Remote Monitoring
IPTO
Athens, Greece
schountasis@admie.gr

Dimitrios Sklavounos
Dept of Computer Science
UEL
London, UK
dsklavounos@uel.ac.uk

Aris Dentsoras
Electrical and Comp. Engineering
NTUA
Athens, Greece
adentsor@power.ece.ntua.gr

Orestis Dantsos
Electrical and Comp. Engineering
NTUA
Athens, Greece
dorestis@mail.ntua.gr

George Kontes
Electrical and Comp. Engineering
NTUA
Athens, Greece
gkontes@cs.ntua.gr

Abstract— Power systems frequency stability is vital for their reliable operation, especially in modern grids with high penetration of renewable energy sources and distributed generation. Robust, multi-stage hybrid frameworks for real-time frequency disturbance detection using high-resolution Phasor Measurement Unit (PMU) data is presented in this work. The recommended technique employs a Discrete Wavelet Transform (DWT) to isolate high-frequency transient singularities and a Kalman Filter (KF) for recursive state estimation to suppress stochastic measurement noise. Cumulative Sum (CUSUM) and the Subspace Identification (SID) function together providing a high-precision framework for identifying abrupt frequency changes and certain step-like irregularities in power transmission systems. This framework revealed superior capacity to perform early fault detection being evaluated using synchrophasor data from the Independent Power Transmission Operator (IPTO) network.

Keywords— power system, frequency disturbance, detection method, phasor measurement unit

I. INTRODUCTION

Maintaining a constant system frequency (50 Hz in Europe) is fundamental to the reliable and consistent operation of electrical power systems. Also, it provides a real-time balance metric between power generation and consumption [1]-[3]. The converter-based nature of wind parks and solar photovoltaics reduces natural grid inertia, leading to increased power instability and higher frequency volatility. Consequently, conventional control and monitoring methods might not be adequate for assuring system stability. Modern data-driven methods circumvent the shortcomings of conventional physical modeling, that frequently lacks to take into consideration the stochastic nature of Inverter-Based Resources (IBRs), by utilizing high-fidelity streams from PMUs. These advances are based on a number of diverse but complementary mathematical techniques that result in the formation of real-time analytical frameworks that can accommodate the intricacies of contemporary, low-inertia grids.

In contrast to traditional models, SID methods directly extract intrinsic system dynamics from observed input-output data by using singular value decomposition [4]. Coupled with the KF, it provide an optimal recursive estimate of the grid's state vector by effectively distinguish the fundamental frequency dynamics from Gaussian measurement noise and communication-induced jitter. However, in noise-laden environments, predictive modeling is not sufficient by itself. The framework could utilize DWT to improve sensitivity to non-stationary transients. Through decomposition of frequency residuals into multi-resolution coefficients, the DWT efficiently isolates singularities or discontinuities that define a fault's initiation [5]–[6].

With a focus on early detection of a frequency disturbance (forced oscillation) this study suggests a data-driven modeling framework for the forecasting and analysis of frequency dynamics in high-voltage transmission systems. Without a requirement of specific physical parameters, the SID technique generates a state-space model that accurately depicts the underlying system dynamics. This model establishes a baseline for nominal operation by predicting the system frequency during an initial stage. The CUSUM statistical process control chart is employed in the final stage of the proposed hybrid detection technique. The CUSUM is particularly suited to identify subtle, persistent changes in process behavior by accumulating the log-likelihood ratios of the residuals derived from the SID and KF phases. The technique optimizes the trade-off between fault alarm likelihood and detection time by triggering alarms when the cumulative deviation exceeding a statistically significant threshold. The proposed integrated methodology is validated using PMU data from the IPTO synchrophasor network, obtained via the FABCROSS EU Horizon 2020 project [7]. Data from PMUs are transported via a communication network using the IEEE C37.118.2 protocol. The outcomes of investigations reveal that the incorporation of CUSUM, SID, KF, and DWT can detect the initial stages of forced oscillations before they actually appear in raw frequency measurements. This research develops a robust diffusive mechanism for real-time deployment in energy control centers, indicating the potential benefits of such hybrid detection methods. The proposed methods represent a robust and versatile tool for strengthening the reliability and resilience of modern power systems in an increasingly complex, data-driven energy landscape.

II. PROPOSED METHOD AND THEORETICAL ELEMENTS

The IPTO energy control center's advancement in real-time monitoring is indicative of a broader shift in perspectives toward data-driven grid observability. The transition from inflexible physical models to flexible, high-resolution frameworks that are capable of processing synchrophasor streams to preserve stability in renewable-rich environments.

Leveraging on the voltage stability method using PMU data at transmission buses, by Vournas et al. [8], the current application employing hybrid detection methods for frequency monitoring. The techniques establish an optimal recursive estimate of frequency considering the PMU stream

XXX-X-XXXX-XXXX-X/XX/$XX.00 ©20XX IEEE

Original Prototype in MATLAB

Validated offline on historical PMU datasets from the Independent Power Transmission Operator (IPTO) network.

Target: Continuous 50 Hz Pipeline

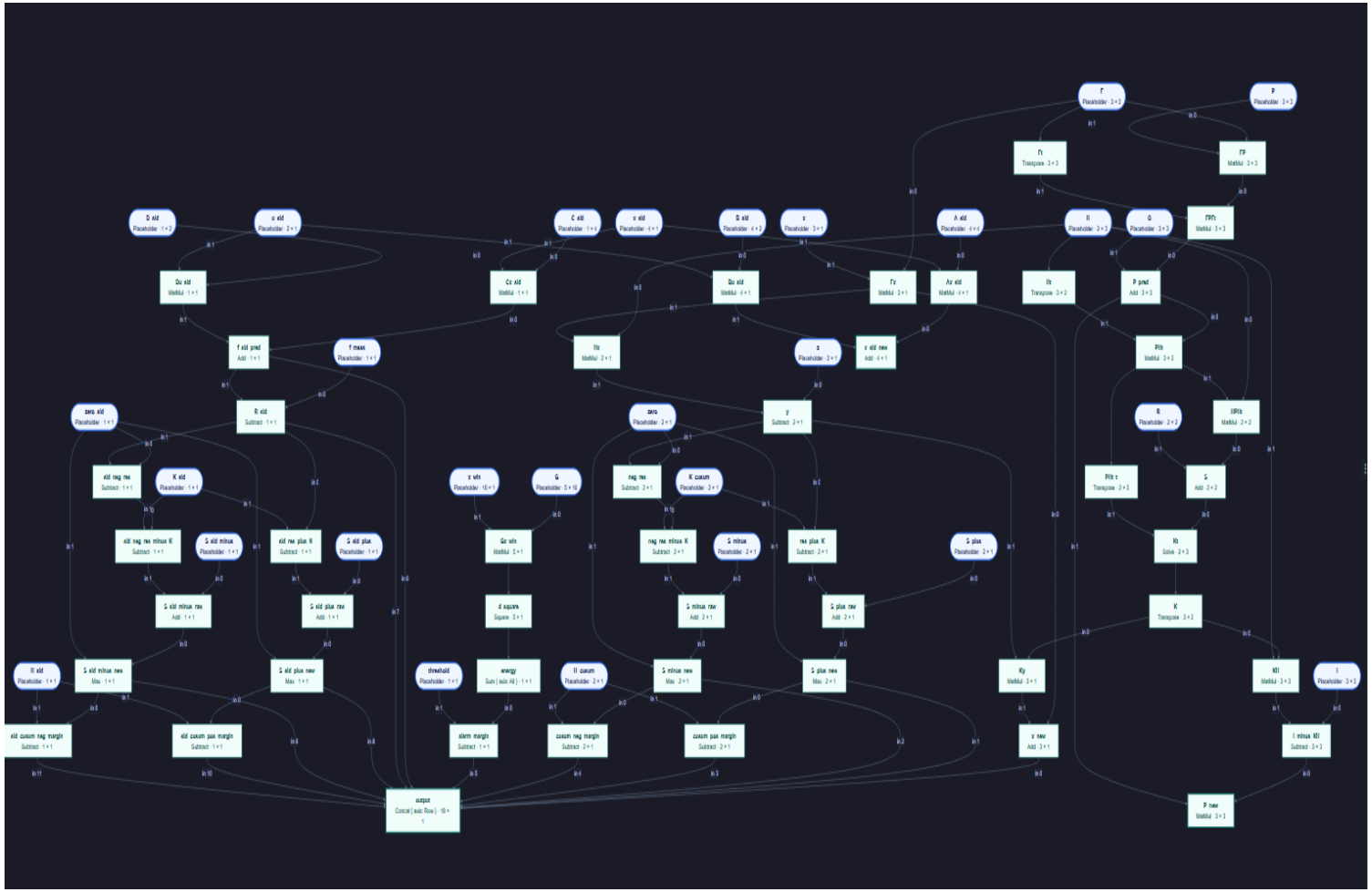
Adapt the offline prototype for continuous processing of PMU streams

Domain Expert Co-Design

The electrical engineers expressed their equations directly in our declarative DSL without software intermediaries.

UNIFIED HYBRID TOPOLOGY

Several Detection Methods, One Computation Graph



SECTION 01 — Kinematic

Kalman Estimation & CUSUM

Tracks frequency & RoCoF; monitors innovation residuals with dual-sided statistical cumulative sum.

SECTION 02 — Transients

Wavelet Energy Detection

Applies Daubechies 4 (db4) filter bank over 16-sample sliding windows to trap high-frequency energy spikes.

SECTION 03 — Subspace

State-Space Prediction (SID)

MISO 4th-order state-space prediction using V-I inputs, tripping CUSUM on persistent model drift.

ENGINE DESIGN CHALLENGE

An Acyclic Graph with State Across Samples

kalman_dwt_cusum.dsl — State Transitions

```
# --- State transitions ---
x           | x_new;
P           | P_new;
S_plus      | S_plus_new;
S_minus     | S_minus_new;
x_sid       | x_sid_new;
S_sid_plus  | S_sid_plus_new;
S_sid_minus | S_sid_minus_new]
```

The challenge: Cycles Break DAG Invariants

A recursive state update (x_{k+1} depends on x_k) creates a cycle in a standard graph, destroying topological sorting.

Intra-Sample Feedforward Purity

Within tick k, placeholders (x, P) act as independent root inputs. Toposort runs with zero cycles.

Inter-Sample State Handoff

Targeting a placeholder registers a deferred transition.

LLM COMPILER FRONTEND

Anatomy of a Structured Compiler Prompt

LAYER 1 — Role & Guardrails

Role Definition

Explicitly sets the model as a compiler frontend. Forbids conversational filler, explanations, and markdown fences.

LAYER 2 — Syntax Specification

Grammar & Constraints

Defines case-sensitive operator tokens, shape parameters, and Bayesian conditional dependency topology rules.

LAYER 3 — Grounding

Few-Shot Demonstrations

Provides paired examples linking English queries directly to valid node declarations and edge connections.

LAYER 4 — Invocation

User Natural Prompt

Supplies the specific operational task. The LLM converts it into an AST with all required context in place.

prompts/dsl_instructions.txt — System Context

```
# 1. ROLE
You are a compiler frontend translating NLP requests into Graph DSL.
Output raw DSL lines ONLY.
No markdown, fences, or explanations.

# 2. GRAMMAR & CONSTRAINTS
Operator[label = , = ]
Supported: RequestUrl, RequestParam, Text, Get, Post, MatMul, Add...
Final line: [ | ]

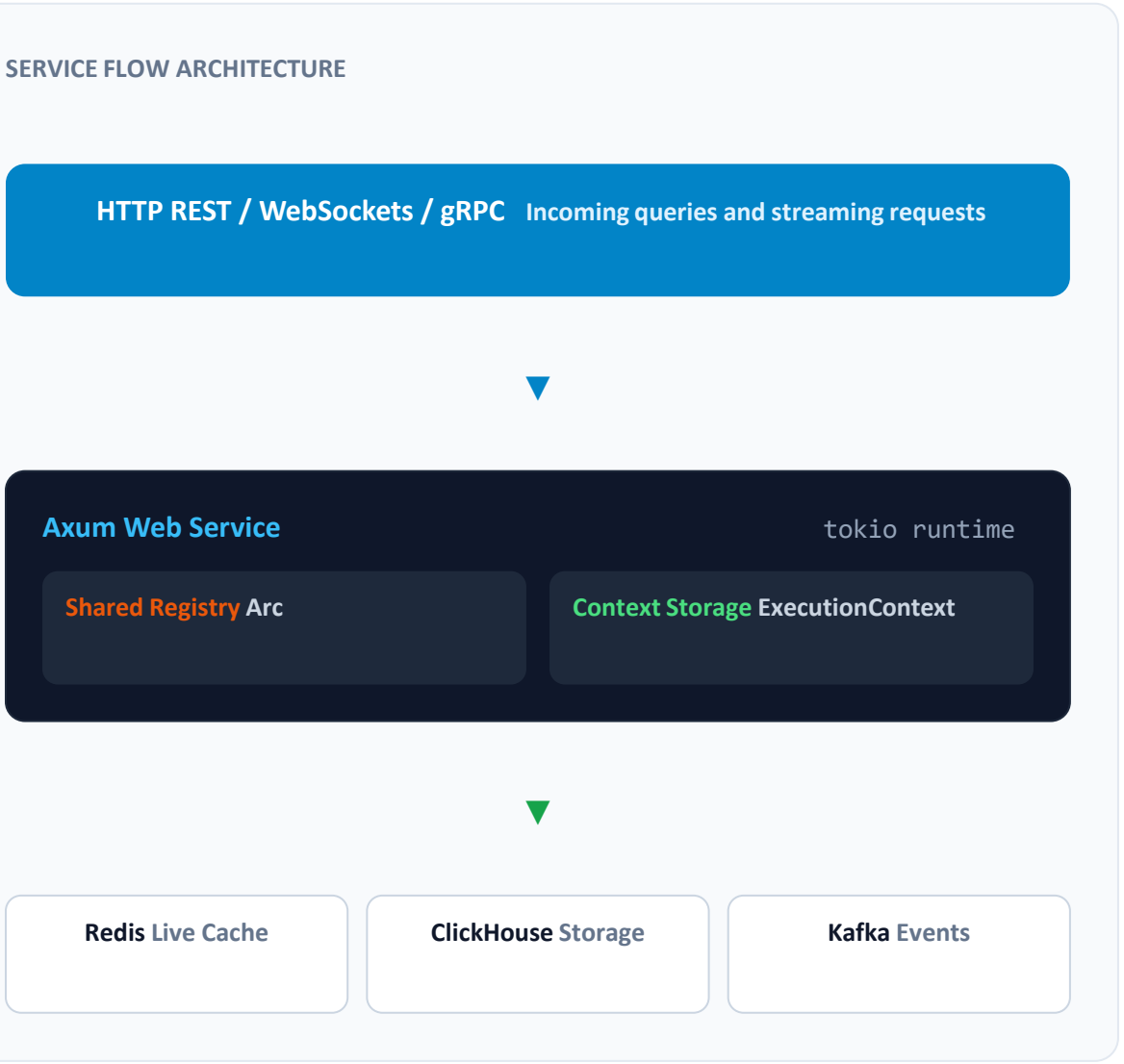
# 3. FEW-SHOT DEMONSTRATION
Input: "Fetch hourly temperatures from Open-Meteo for Athens."
Output:
  RequestUrl[label = openmeteo,
    name = "https://api.open-meteo.com/v1/forecast"]
  RequestParam[label = hourly, name = "hourly"]
  Text[label = temp_field]
  Get[label = fetch]
  [openmeteo; hourly; temp_field; fetch|openmeteo,hourly]

# 4. USER INVOCATION
Query: "Retrieve 50 Hz PMU frequency and compute CUSUM alarms."
```

Deterministic Output: Temperature 0.1 locks syntax generation into strict AST emission.

PRODUCTION ARCHITECTURE

Infrastructure Integration via Axum



Exposing the Engine as a Service

A straightforward solution to expose the engine is to integrate it into a web service using a framework like Axum.

In-Memory Compiled Graphs

The service compiles blueprints once and shares them across requests without recompilation or lock contention.

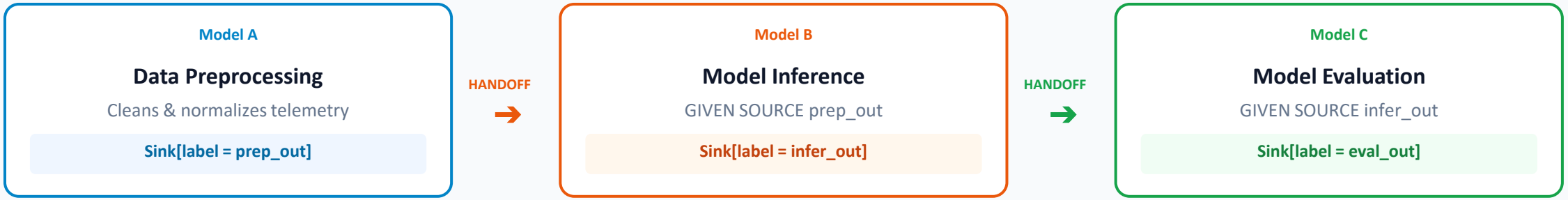
Standardized Protocol Access

External applications query the engine over HTTP, WebSocket, or gRPC without needing native Rust bindings.

MULTI-MODEL ORCHESTRATION

Composable Pipelines: Sinks to Sources

CHAINED MULTI-MODEL PIPELINE



Definition DSL

Sink Nodes Route Output

The Definition DSL includes dedicated Sink nodes to direct computation results to files, ClickHouse, or Kafka.

```
Sink[label = out, table = ...]
```

Execution DSL

Source Nodes Feed Input

Downstream models bind upstream outputs as their inputs using the GIVEN SOURCE clause in the query.

```
GIVEN SOURCE ./data/prep_out
```

Modular Topology

Independent Lifecycle

Preprocessing, inference, and evaluation remain separate, auditable graphs composed without monolithic coupling.

```
Model A ➔ Model B ➔ Model C
```

THE ROAD AHEAD

Engine Evolution & Future Milestones

Pillar 01

Decision-Making Algorithms

- Probablistic
- Deterministic

- Variable Elimination & Factor Products
- MCMC Sampling (Gibbs & Likelihood)
- Automated Causal Structure Learning

Pillar 02

Graph Execution Optimizations

Compiler passes that detect independent computational subgraphs for concurrent multi-core scheduling.

- Parallel Subgraph Forking (Tokio / Rayon)
- Kernel Fusion (Add + MatMul into SIMD)

Pillar 03

Engine-as-a-Service (Axum)

Exposing compiled graphs via high-throughput HTTP/gRPC microservices with shared zero-copy registries.

- Low-Latency Axum HTTP & gRPC Endpoints
- WebSocket & SSE for 50 Hz Live Feeds
- Dynamic Multi-Tenant Model Registry

RUSTLAB 2026 • BOLOGNA • NOV 03

Building a Declarative Computation Engine in Rust

Questions & Discussion