PetriNet Studio: Architecting a SaaS Simulator in Modern C++

Eng. Gabriel Valenzuela

Meeting C++ 2025

October 2025

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened...

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened…
- What if your concurrency diagram could just *run*?

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened…
- What if your concurrency diagram could just *run*?
- That diagram is a *Petri Net*.

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened…
- What if your concurrency diagram could just *run*?
- That diagram is a *Petri Net*.
- We turned it into a *distributed simulator* in *C++23*.

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened…
- What if your concurrency diagram could just *run*?
- That diagram is a *Petri Net*.
- We turned it into a *distributed simulator* in *C++23*.
- We want share the architecture and performance story.

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened…
- What if your concurrency diagram could just *run*?
- That diagram is a *Petri Net*.
- We turned it into a *distributed simulator* in *C++23*.
- We want share the architecture and performance story.

What is this talk about?

- You've debugged a race condition that **shouldn't** have happened...
- What if your concurrency diagram could just *run*?
- That diagram is a *Petri Net*.
- We turned it into a *distributed simulator* in *C++23*.
- We want share the architecture and performance story.

"From formal models to real systems — where math meets C++ through the state equation."

About me

- **Gabriel Valenzuela**
  Computer Engineer (UNC), MSc Software Engineering (UNLP)
- Professor at UNC, IEEE (CS/YP) member, RUNIC network
- Senior C++ Developer at **Wazuh**
- Research: concurrency, microservices, complex systems, Petri Nets

UNC · Universidad Nacional de La Plata · IEEE Computer Society · IEEE young professionals Argentina Section · wazuh. · RUNIC

Agenda

1. Context & motivation
2. Objectives
3. Extended state equation
4. Architecture
5. Design patterns & modern C++
6. Algorithmic design
7. Performance & risks
8. Questions & closing

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.
- Represents systems as a bipartite graph:

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.
- Represents systems as a bipartite graph:
  - **Places** (circles): hold tokens (resources/states)

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.
- Represents systems as a bipartite graph:
  - **Places** (circles): hold tokens (resources/states)
  - **Transitions** (rectangles): consume/produce tokens

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.
- Represents systems as a bipartite graph:
  - **Places** (circles): hold tokens (resources/states)
  - **Transitions** (rectangles): consume/produce tokens
  - **Arcs**: define causal relationships

Carl Adam Petri (1926 – 2010)

Petri Nets: Recap

- Introduced by **Carl Adam Petri** in 1962 — in his PhD thesis at the University of Bonn.
- One of the first *mathematical models of concurrent computation*.
- Represents systems as a bipartite graph:
  - **Places** (circles): hold tokens (resources/states)
  - **Transitions** (rectangles): consume/produce tokens
  - **Arcs**: define causal relationships
- A marking (token distribution) = system's global state

Carl Adam Petri (1926 – 2010)

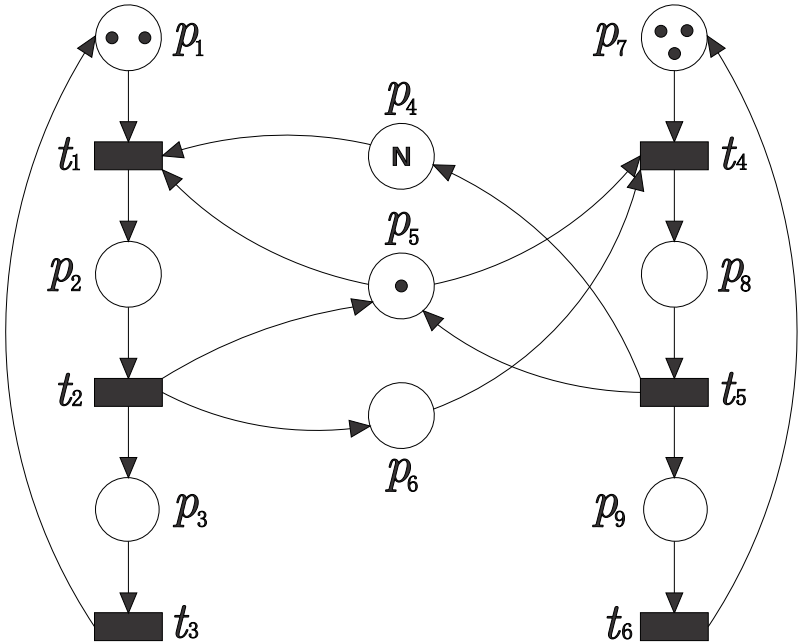
Petri Nets: Recap

So... what are they?

- Mathematical model for concurrent systems
- Places (states) + Transitions (events) + Tokens (resources)
- Formal verification + Visual representation

Extended Petri Nets add

- Inhibitor arcs (must be zero)
- Reader arcs (test without consuming)
- Reset arcs (zero a place)
- Guards & time windows



Example: Producer-Consumer

Why they still matter

- Formal verification *and* visual reasoning

Why they still matter

- Formal verification *and* visual reasoning
- Natural fit for concurrent/distributed systems

Why they still matter

- Formal verification *and* visual reasoning
- Natural fit for concurrent/distributed systems
- Applications: workflow, embedded control, protocols, performance [1]

Why they still matter

- Formal verification *and* visual reasoning
- Natural fit for concurrent/distributed systems
- Applications: workflow, embedded control, protocols, performance [1]

Why they still matter

- Formal verification *and* visual reasoning
- Natural fit for concurrent/distributed systems
- Applications: workflow, embedded control, protocols, performance [1]

"Petri Nets are a visual debugger and design tool for complex systems."

Why this project?

- Existing PN tools: academic, monolithic, desktop-only

Why this project?

- Existing PN tools: academic, monolithic, desktop-only
- No cloud-native simulator with distributed compute backends

Why this project?

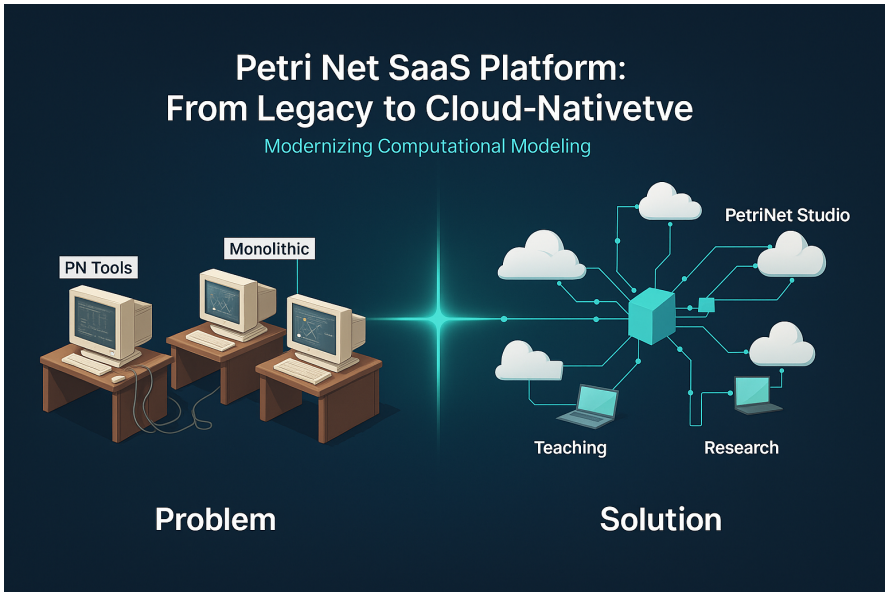
- Existing PN tools: academic, monolithic, desktop-only
- No cloud-native simulator with distributed compute backends
- **Need: an open SaaS platform for teaching, analysis, and research**

Why this project?

- Existing PN tools: academic, monolithic, desktop-only
- No cloud-native simulator with distributed compute backends
- **Need: an open SaaS platform for teaching, analysis, and research**

Why this project?

- Existing PN tools: academic, monolithic, desktop-only
- No cloud-native simulator with distributed compute backends
- **Need: an open SaaS platform for teaching, analysis, and research**



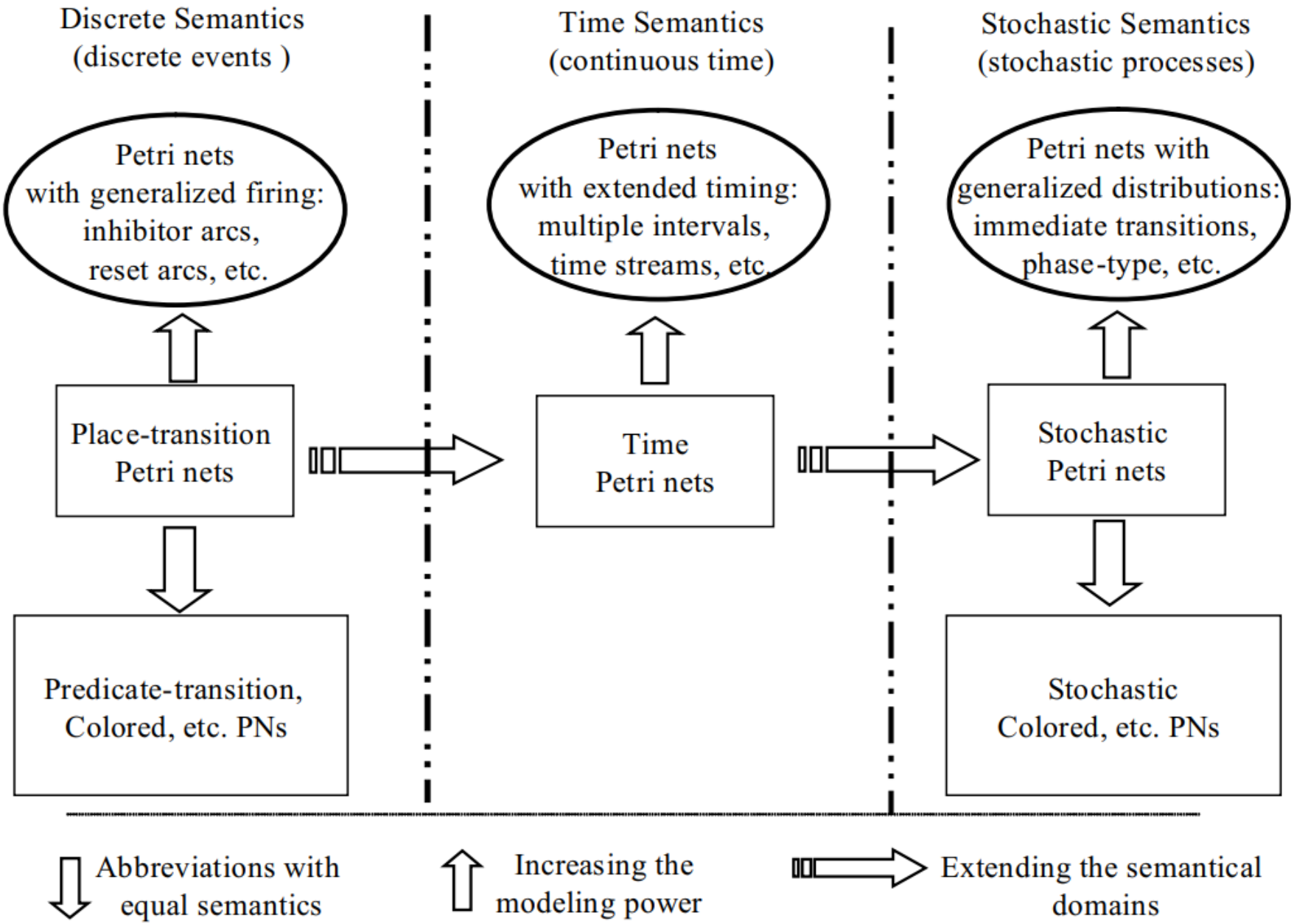
Problem & inspiration

Traditional flow:
Model in PN $\rightarrow$ hand-translate to code $\rightarrow$ lose guarantees, complex to do, error prone

Inspiration — PPX (Ventre, Micolini & Daniele) [2]:
Execute the **model directly** via an extended state equation on hardware (FPGA)

Goal:
Bring PPX's philosophy to a cloud-native C++23 backend (SaaS)

Problem & inspiration



Semantics domains of the Petri net-based models [3]

Classical vs extended equation

Classical:

$$M_{j+1} = M_j + I \cdot \sigma$$

Extended (PPX):

$$M_{j+1} = M_j + I \cdot (\sigma \wedge Ex) \# A$$

where

$$Ex = E \wedge B \wedge L \wedge G \wedge Z$$

E enabled, B inhibitor, L reader, G guards, Z time windows; A resets.

This equation is the core of our C++ engine.[4]

Extended step in C++23 (simplified)

```cpp
struct Marking { std::vector<int> m; };

auto step(Marking& M, mdspan<const int, dextents<2>> I,
span<const bool> E, span<const bool> B,
span<const bool> L, span<const bool> G,
span<const bool> Z, span<const bool> sigma) {
  // Ex = E & B & L & G & Z
  vector<bool> Ex(E.size());
  for (size_t t=0; t<Ex.size(); ++t)
  Ex[t] = E[t] && B[t] && L[t] && G[t] && Z[t];

  // Pick first enabled (policy pluggable)
  size_t t = find_first_enabled(sigma & Ex);

  if (t < Ex.size()) {
    auto col = I.column(t);
    for (size_t p=0; p<col.size(); ++p)
    M.m[p] += col[p]; // vectorize with std::simd
  }
}
```

System architecture (SaaS)

- **Frontend:** React + React Flow (editor + visualizer)

System architecture (SaaS)

- **Frontend:** React + React Flow (editor + visualizer)
- **Go BFF:** orchestration (gRPC/HTTP/WS), RabbitMQ producer

System architecture (SaaS)

- **Frontend:** React + React Flow (editor + visualizer)
- **Go BFF:** orchestration (gRPC/HTTP/WS), RabbitMQ producer
- **C++ engines:** extended equation executor + analyses (gRPC/AMQP)

System architecture (SaaS)

- **Frontend:** React + React Flow (editor + visualizer)
- **Go BFF:** orchestration (gRPC/HTTP/WS), RabbitMQ producer
- **C++ engines:** extended equation executor + analyses (gRPC/AMQP)
- **Storage:** PNML/JSON; schemas for zero/single copy (FlatBuffers)

System architecture (SaaS)

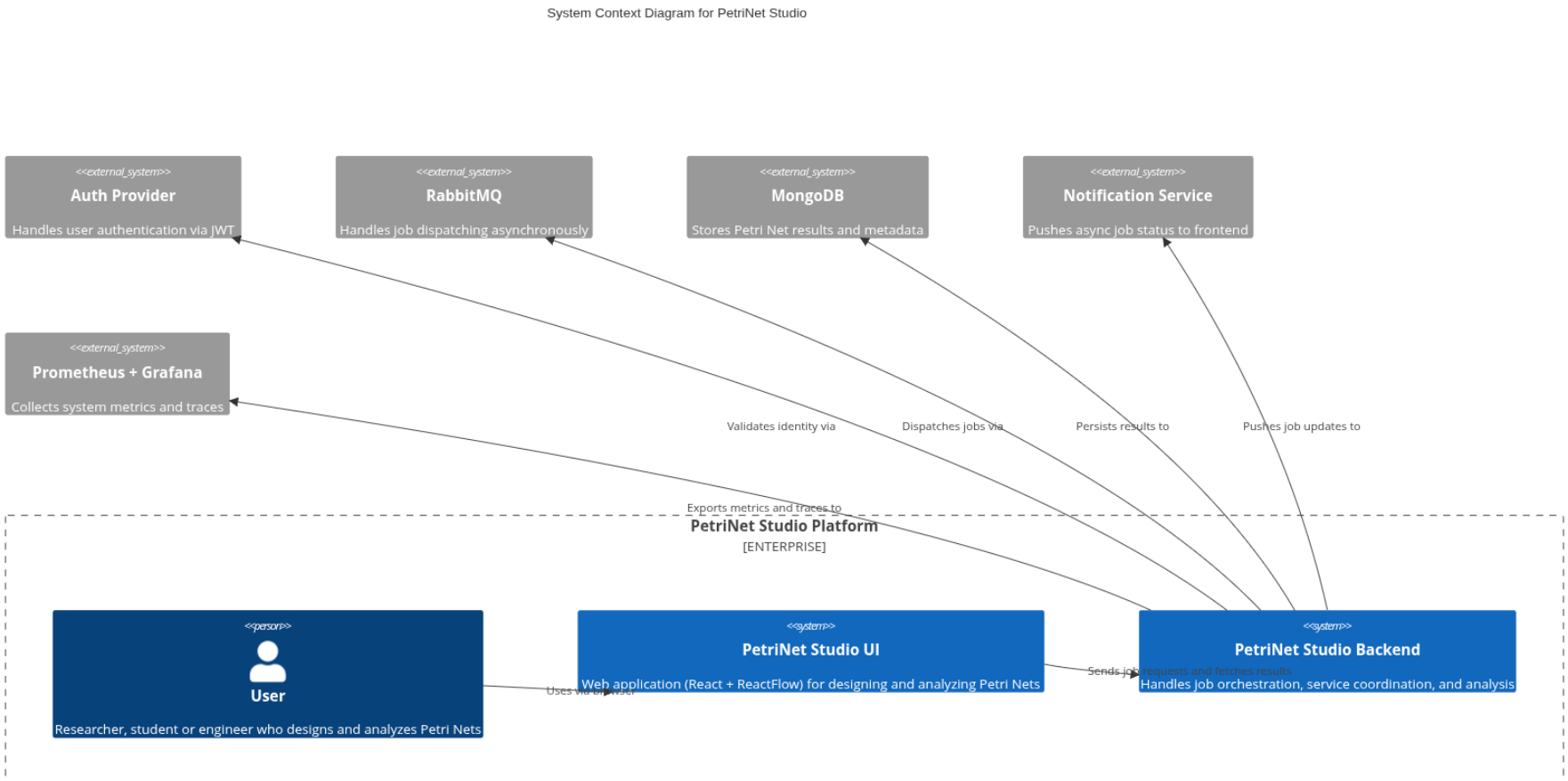
- **Frontend:** React + React Flow (editor + visualizer)
- **Go BFF:** orchestration (gRPC/HTTP/WS), RabbitMQ producer
- **C++ engines:** extended equation executor + analyses (gRPC/AMQP)
- **Storage:** PNML/JSON; schemas for zero/single copy (FlatBuffers)

System architecture (SaaS)

- **Frontend:** React + React Flow (editor + visualizer)
- **Go BFF:** orchestration (gRPC/HTTP/WS), RabbitMQ producer
- **C++ engines:** extended equation executor + analyses (gRPC/AMQP)
- **Storage:** PNML/JSON; schemas for zero/single copy (FlatBuffers)

Hybrid: software flexibility + optional HW/GPU acceleration

System architecture (SaaS)



System Context Diagram for PetriNet Studio [3]

C++ backend modules

Module	Responsibility	PPX analogue
model	PN entities, validated builders	Matrix program
math	Incidence & kernels (dense/sparse)	Calc. state
engine	Extended eq., coverability, invariants	Core algorithm
runtime	Pools, monitor, timers, telemetry	Queues/policy
io	PNML/JSON, FlatBuffers/Glaze	Matrix loading
hw	SIMD/GPU facades (PImpl)	FPGA fabric

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense
- **Builder:** incidence & vectors from PNML/JSON

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense
- **Builder:** incidence & vectors from PNML/JSON
- **Observer:** progress/telemetry to BFF

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense
- **Builder:** incidence & vectors from PNML/JSON
- **Observer:** progress/telemetry to BFF
- **PImpl (Pointer to Implementation):** GPU/AVX backends, stable ABI

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense
- **Builder:** incidence & vectors from PNML/JSON
- **Observer:** progress/telemetry to BFF
- **PImpl (Pointer to Implementation):** GPU/AVX backends, stable ABI

Patterns mapped to execution

- **Strategy:** firing policy (priority, deterministic, random)
- **Policy-based:** arithmetic (checked/saturating), sparse/dense
- **Builder:** incidence & vectors from PNML/JSON
- **Observer:** progress/telemetry to BFF
- **PImpl (Pointer to Implementation):** GPU/AVX backends, stable ABI

Not using: Visitor, Command, or Abstract Factory — we execute matrices directly

Concepts for Plug-in Safety (C++23)

```cpp
template<typename Algo>
concept Algorithm =
requires(Algo a) {
  { Algo::name() } ->
      std::convertible_to<std::string_view>;
  { a.execute(span<const int>{}, span<int>{}) }
  -> std::same_as<std::expected<void,int>>;
};

struct Coverability {
  static constexpr std::string_view name() {
    return "coverability";
  }
  std::expected<void,int> execute(span<const int> I,
  span<int> out);
};

// Compile-time enforcement of the interface contract
static_assert(Algorithm<Coverability>);
```

Each analysis module (e.g., coverability, invariants, reachability) is validated at compile time to match the system's plugin contract — preventing ABI drift and runtime errors.

Zero-copy with mdspan

```cpp
// FlatBuffers schema yields contiguous int32_t array (row-major)
auto incidence_view(const uint8_t* fb)
-> std::mdspan<const int32_t, std::dextents<size_t,2>>
{
  auto mat = GetIncidenceMatrixFB(fb);
  return { mat->data()->data(), mat->rows(), mat->cols() }; // no copy!
}

// Use directly in engine
M_new = M + incidence_view.column(t);
```

Efficient dataflow: the incidence matrix is accessed in-place from FlatBuffers via `std::mdspan`*, avoiding copies and preserving cache locality during Petri Net firing updates.*

Vectorizing the update (Cycle 2)

```
#include <experimental/simd>
using std::experimental::native_simd;

void add_column_simd(span<int> M, span<const int> col) {
  size_t i = 0;
  constexpr size_t width = native_simd<int>::size();

  for (; i + width <= M.size(); i += width) {
    native_simd<int> vm(&M[i], element_aligned);
    native_simd<int> vc(&col[i], element_aligned);
    (vm + vc).copy_to(&M[i], element_aligned);
  }

  // Scalar tail
  for (; i < M.size(); ++i)
  M[i] += col[i];
}
```

Cycle 2 — Update phase of the PPX: the selected transition acts as a column selector of the incidence matrix. Vectorization with `std::simd` *accelerates the marking update* $M_{j+1} = M_j + I_{\cdot,t}$ *while preserving deterministic semantics.*

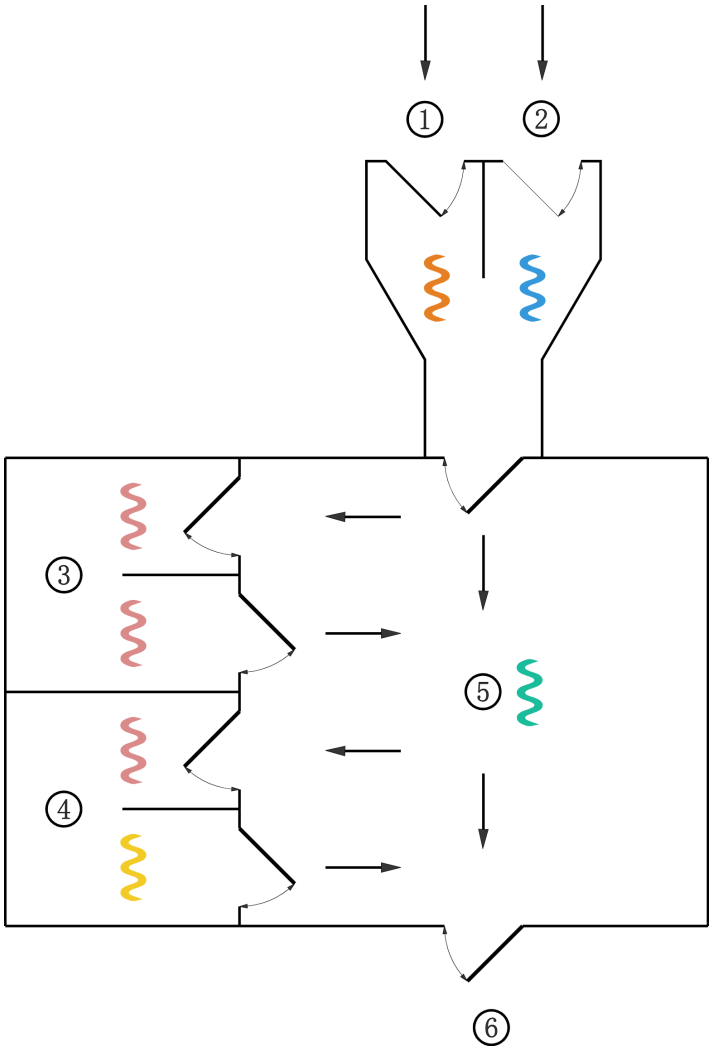
Structured Concurrency & Monitor Object Pattern

```
1         std::jthread worker([&](std::stop_token st) {
2           while (!st.stop_requested()) {
3             std::unique_lock lock(monitor_mutex); // Monitor
                  guard
4
5             auto Ex = compute_Ex(M);    // Enabled transitions
6             fire_transition(M, Ex);     // Update marking safely
7             notify_observers();         // Push telemetry
8           }
9         });
10
11        // Cancel from controller
12        worker.request_stop(); // RAII: stop + join on
              destruction
```

Monitor Object Pattern: Encapsulates both *state* and *synchronization* inside an object, ensuring that only one synchronized method executes at a time. Each simulation thread acts as a **monitor**, serializing access to shared state (`M`) and using `wait`/`notify` semantics for cooperative execution.

Unlike the Active Object pattern, the monitor does not run on a separate thread — each request executes in the client's thread, maintaining invariants and preventing data races.

Structured Concurrency & Monitor Object Pattern



Block diagram

Execution Model: Single vs Parallel

Deterministic (single server):

```
        while (!stop) {
          auto Ex = compute_Ex(M);
          auto t  = pick_by_priority(Ex);
          M += I.column(t);
        }
```

Parallel (maximal independent set):

```
        while (!stop) {
          auto Ex = compute_Ex(M);
          parallel_for_each(enabled_clusters(Ex), [&](auto t) {
            if (try_acquire_tokens(M, t))
            M += I.column(t);
          });
        }
```

The deterministic model enforces serial firing order—suitable for validation and debugging. The parallel model computes a maximal independent set of non-conflicting transitions, allowing concurrent firing when token dependencies do not overlap. Each firing unit behaves as a monitor, preserving marking consistency through atomic token acquisition.

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Invariants:

- P-invariants: $y^T \cdot I = 0$ (token conservation)

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Invariants:

- P-invariants: $y^T \cdot I = 0$ (token conservation)
- T-invariants: $I \cdot x = 0$ (cyclic firing sequences)

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Invariants:

- P-invariants: $y^T \cdot I = 0$ (token conservation)
- T-invariants: $I \cdot x = 0$ (cyclic firing sequences)
- Integer nullspace via Smith/Hermite normal form

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Invariants:

- P-invariants: $y^T \cdot I = 0$ (token conservation)
- T-invariants: $I \cdot x = 0$ (cyclic firing sequences)
- Integer nullspace via Smith/Hermite normal form

Analyses beyond execution

Coverability:

- MinCovVec algorithm with ω (unbounded)
- Cutoff heuristics (depth limit, hash pruning)
- GPU candidate: parallel frontier expansion

Invariants:

- P-invariants: $y^T \cdot I = 0$ (token conservation)
- T-invariants: $I \cdot x = 0$ (cyclic firing sequences)
- Integer nullspace via Smith/Hermite normal form

Structural:

- Siphons/traps via SCC(Strongly Connected Components) and feedback sets

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.
- Extends Karp–Miller, Monotone-Pruning (MP), and MinCov with:

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.
- Extends Karp–Miller, Monotone-Pruning (MP), and MinCov with:
  - Vectorized state updates in C++20.

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.
- Extends Karp–Miller, Monotone-Pruning (MP), and MinCov with:
  - Vectorized state updates in C++20.
  - Parallel firing and hash-based redundancy filtering.

MinCovVec algorithm detailed

- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.
- Extends Karp–Miller, Monotone-Pruning (MP), and MinCov with:
  - Vectorized state updates in C++20.
  - Parallel firing and hash-based redundancy filtering.
  - Controlled accelerations (ω propagation) with bounded memory.

MinCovVec algorithm detailed

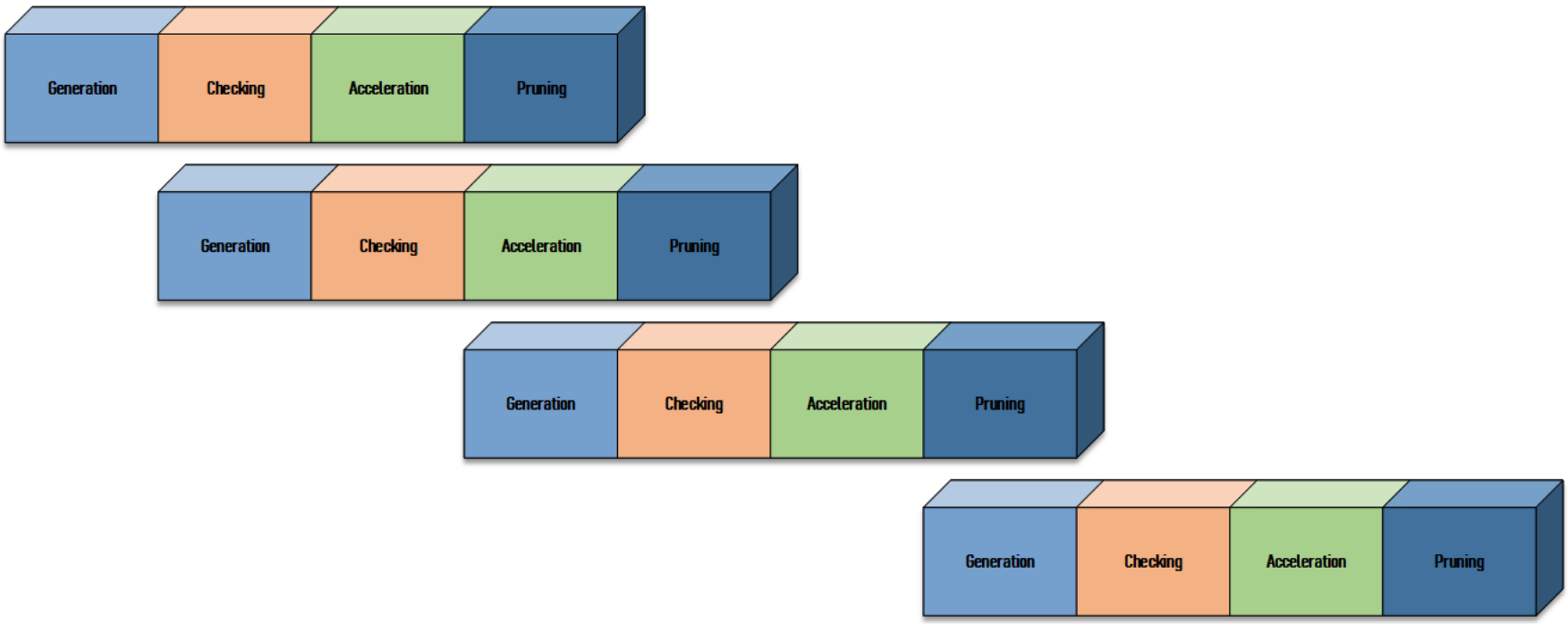
- Based on the **extended state equation** and the **PPX model**.
- Implements an optimized minimal coverability algorithm, [5] **MinCovVec**.
- Extends Karp–Miller, Monotone-Pruning (MP), and MinCov with:
  - Vectorized state updates in C++20.
  - Parallel firing and hash-based redundancy filtering.
  - Controlled accelerations (ω propagation) with bounded memory.
- Achieves significant reduction in node count and runtime vs. MP and MinCov.

MinCovVec algorithm detailed: Pseudocode (simplified)

```
// MinCovVec: minimal coverability tree (vectorized)
initialize(root = M0);
while (!pending.empty()) {
  auto node = pending.pop();
  for (auto t : enabled(node.M)) {
    auto M_new = fire(node.M, t);
    if (!exists_in(verSet, M_new)) {
      accelerate(M_new, node.ancestors);
      prune(verSet, M_new);
      verSet.insert(M_new);
      pending.push(M_new);
    }
  }
}
```

Parallel firing, hash filtering and ω-acceleration based on MinCovVec [5].

MinCovVec algorithm detailed: Block diagram



Pipeline of the algorithm

Performance plan

- **Data layout:** SoA; contiguous incidence columns

Performance plan

- **Data layout:** SoA; contiguous incidence columns
- **Kernels:** sparse vs dense; runtime dispatch via traits

Performance plan

- **Data layout:** SoA; contiguous incidence columns
- **Kernels:** sparse vs dense; runtime dispatch via traits
- **SIMD:** AVX-512 fastpath; baseline `std::simd`

Performance plan

- **Data layout:** SoA; contiguous incidence columns
- **Kernels:** sparse vs dense; runtime dispatch via traits
- **SIMD:** AVX-512 fastpath; baseline `std::simd`
- **Zero-copy:** FlatBuffers $\rightarrow$ `mdspan` views

Performance plan

- **Data layout:** SoA; contiguous incidence columns
- **Kernels:** sparse vs dense; runtime dispatch via traits
- **SIMD:** AVX-512 fastpath; baseline `std::simd`
- **Zero-copy:** FlatBuffers $\rightarrow$ `mdspan` views
- **Execution modes:** deterministic for pedagogy; parallel for throughput

Performance plan

- **Data layout:** SoA; contiguous incidence columns
- **Kernels:** sparse vs dense; runtime dispatch via traits
- **SIMD:** AVX-512 fastpath; baseline `std::simd`
- **Zero-copy:** FlatBuffers $\rightarrow$ `mdspan` views
- **Execution modes:** deterministic for pedagogy; parallel for throughput
- **Metrics:** P50/P95 step time, IPC, memory BW, contention

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps
- **SIMD/GPU portability:** complexity vs performance gains

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps
- **SIMD/GPU portability:** complexity vs performance gains
- **Zero-copy safety:** mdspan lifetime tied to arena scope

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps
- **SIMD/GPU portability:** complexity vs performance gains
- **Zero-copy safety:** mdspan lifetime tied to arena scope
- **Plugin ABI:** C boundary + PImpl for stability

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps
- **SIMD/GPU portability:** complexity vs performance gains
- **Zero-copy safety:** mdspan lifetime tied to arena scope
- **Plugin ABI:** C boundary + PImpl for stability
- **Determinism vs speed:** which should be default?

Risks & trade-offs

- **Coverability blow-up:** heuristics, time/memory caps
- **SIMD/GPU portability:** complexity vs performance gains
- **Zero-copy safety:** mdspan lifetime tied to arena scope
- **Plugin ABI:** C boundary + PImpl for stability
- **Determinism vs speed:** which should be default?
- **Numerical stability:** exact integer arithmetic for invariants

We’re still analyzing…

1. Plugin ABI: C interface + C++ Concepts — good balance?

We’re still analyzing…

1. Plugin ABI: C interface + C++ Concepts — good balance?
2. Determinism: default on (pedagogy) or opt-in (perf)?

We’re still analyzing…

1. Plugin ABI: C interface + C++ Concepts — good balance?
2. Determinism: default on (pedagogy) or opt-in (perf)?
3. Concurrency: coroutines vs thread pool in practice?

We’re still analyzing…

1. Plugin ABI: C interface + C++ Concepts — good balance?
2. Determinism: default on (pedagogy) or opt-in (perf)?
3. Concurrency: coroutines vs thread pool in practice?
4. Observability: which metrics/traces would we add?

We’re still analyzing…

1. Plugin ABI: C interface + C++ Concepts — good balance?
2. Determinism: default on (pedagogy) or opt-in (perf)?
3. Concurrency: coroutines vs thread pool in practice?
4. Observability: which metrics/traces would we add?
5. Hardware accel: when FPGA (PPX) vs software (C++) in production?

Thank you for your time!

Let's keep building bridges between models and systems.

Slides & code: `github.com/GabrielEValenzuela/petrinetstudio`

Connect: `@gabriel__val` — `github.com/GabrielEValenzuela`

Thanks to the Petri Net community, the RUNIC network, and everyone pushing open research forward.

References I

[1] R. David and H. Alla, *Discrete, Continuous, and Hybrid Petri Nets*. Springer, 2005.

[2] L. O. Ventre, O. Micolini, and E. Daniele, "Extended petri net processor for embedded systems," in *Congreso Argentino de Ciencias de la Computación (CACIC 2020)*, Córdoba, Argentina, 2020, pp. 450–459.

[3] M. Diaz, *Petri nets: fundamental models, verification and applications*. John Wiley & Sons, 2013.

[4] O. Micolini, "Arquitectura asimétrica multicore con procesador de petri," PhD in Ciencias Informáticas, Doctoral thesis, Universidad Nacional de Córdoba, Córdoba, Argentina, 2015.

References II

[5] L. O. Ventre, O. Micolini, G. Valenzuela, and M. Ludemann, “Redes de petri: Algoritmo para la construcción de árboles de mínima cobertura,” in *SAIC 2024 (53° JAIIO)*, MinCovVec: vectorized minimal coverability algorithm, Córdoba, Argentina: SADIO, 2024.

Backup: Extended Arc Types

Arc Type	**Semantics**	**Matrix**
Classic (input)	Consume tokens to enable	I (pre)
Classic (output)	Produce tokens after firing	I (post)
Inhibitor	Enabled only if place is empty	H
Reader	Test tokens without consuming	R
Reset	Zero the place after firing	Rst

Example: Mutual exclusion

- Place p_1 = "Resource available"
- Transition t_1 = "Acquire" (inhibitor from p_2 = "In use")
- t_1 enabled only if p_1 has tokens **AND** p_2 is empty