From Scripts to Contracts





What Rust's type system taught me about infrastructure composition

Jesús Pérez Lorenzo · Rust developer · Libre cloud architect

RustLab 2026 · Bologna · Deep Space

co-organizer · Rust Las Palmas

Piazza Maggiore, Bologna · photo Max Nayman · Unsplash

⚠ CONTENT ADVISORY

Talk is cheap.
Show me the code.

— Linus Torvalds, linux-kernel mailing list, 2000

- This talk contains **code**. A fair amount.
- Code is where a claim becomes **real**: it compiles, it runs, or it fails. The rest, any intelligence can infer.
- Reading it is **optional**. Understanding it is optional. It is there so nothing I say floats.

`Side effects: programs that run on stage · contracts that break on purpose · one bug I didn't know I had.`



Jesús Pérez

Rust developer · Libre cloud architect

jesusperez.pro · jpl@jesusperez.pro



```rust
#[derive(Debug)]
enum Lang { COBOL, Informix4GL, SQL, C, Perl, PHP, Ruby, Bash, JavaScript, Nushell, Go, KCL, Nickel, Rust }
use Lang::*;
struct Engineer { since: u16, journey: &'static [Lang], builds: &'static [&'static str], beliefs: &'static str }

const ME: Engineer = Engineer {
    since: 1987,
    journey: &[COBOL, Informix4GL, SQL, C, Perl, PHP, Ruby, Bash, JavaScript, Nushell, Go, KCL, Nickel, Rust],
    builds: &["provisioning", "ontoref", "rustelo"],
    beliefs: "Taoism — yang is form, yin is flow; neither wins",
};

// Checked by rustc, not at run time: reorder `journey` and this does not compile.
const _: () = assert!(matches!(ME.journey.last(), Some(Rust)), "finally");

fn main() {
    println!("experience: {} years — software development and systems integrations", 2026 - ME.since);
    println!("journey: {:?}", ME.journey);
    println!("builds: {}", ME.builds.join(" · "));
    println!("community: ULPGC + SPEGC workshops · co-organizer, Rust Las Palmas");
    println!("beliefs: {}", ME.beliefs);
    println!("prequel: Rustikon 2026 — \"Why I Needed Rust, Finally\"");
}
```



Jesús Pérez

Rust developer · Libre cloud architect

jesusperez.pro · jpl@jesusperez.pro



compiled and ran

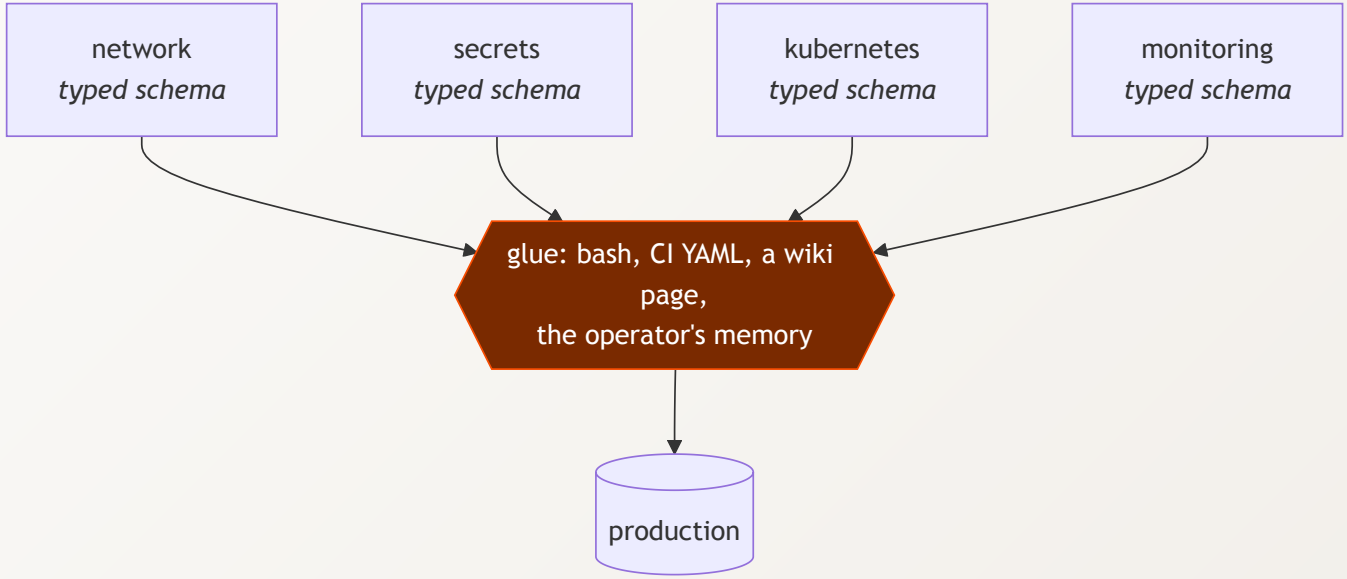
EXPERIENCE	39 years — software development and systems integrations
JOURNEY	COBOL · Informix4GL · SQL · C · Perl · PHP · Ruby · Bash · JavaScript · Nushell · Go · KCL · Nickel · **Rust**
BUILDS	provisioning · ontoref · rustelo
COMMUNITY	ULPGC + SPEGC workshops · co-organizer, Rust Las Palmas
BELIEFS	Taoism — yang is form, yin is flow; neither wins
PREQUEL	Rustikon 2026 — "Why I Needed Rust, Finally"

You have seen this log

```
$ ./deploy.sh production
[ok]   network applied
[ok]   secrets propagated
[ok]   orchestrator updated
[ok]   monitoring reloaded
[ok]   deploy complete          exit 0
```

- Every step reported **success**.
- The **order** between them was **wrong** — and nothing knew there *was* an order.
- The pipeline was **green**. **Production was not.**

Typed leaves, untyped tree



*Modern IaC types every **resource**.*

*The **composition** — what depends on what, in which order, under which condition — is untyped.*

Three layers I kept confusing

Layer	Question	Who checks it
Shape	Does this field exist, with the right primitive?	`serde`, JSON Schema
Form	Do these fields agree with each other?	contracts, validated constructors
Composition	Is the *whole graph* of steps legal?	… usually nobody

- Rust made me fluent in the first two: newtypes, enums, `TryFrom`.
- The third layer is where every 3 AM alert I remember came from.
- This talk is what happened when I made layer three a contract.


Infrastructure deserves the same type discipline
as the code it carries.
If it doesn't typecheck, it doesn't touch a server.
(We'll come back to how true that sentence is. It's mostly true.)

The stack, honestly

why not all of it in Rust? →



Nickel

form — config time

Component contracts, formula DAGs, workspace composition. Rejects before anything runs.

~71k LOC · 984 files

Nushell

orchestration

The CLI, the live DAG executor today, provider glue. Structured data, not strings.

~199k LOC · 895 files

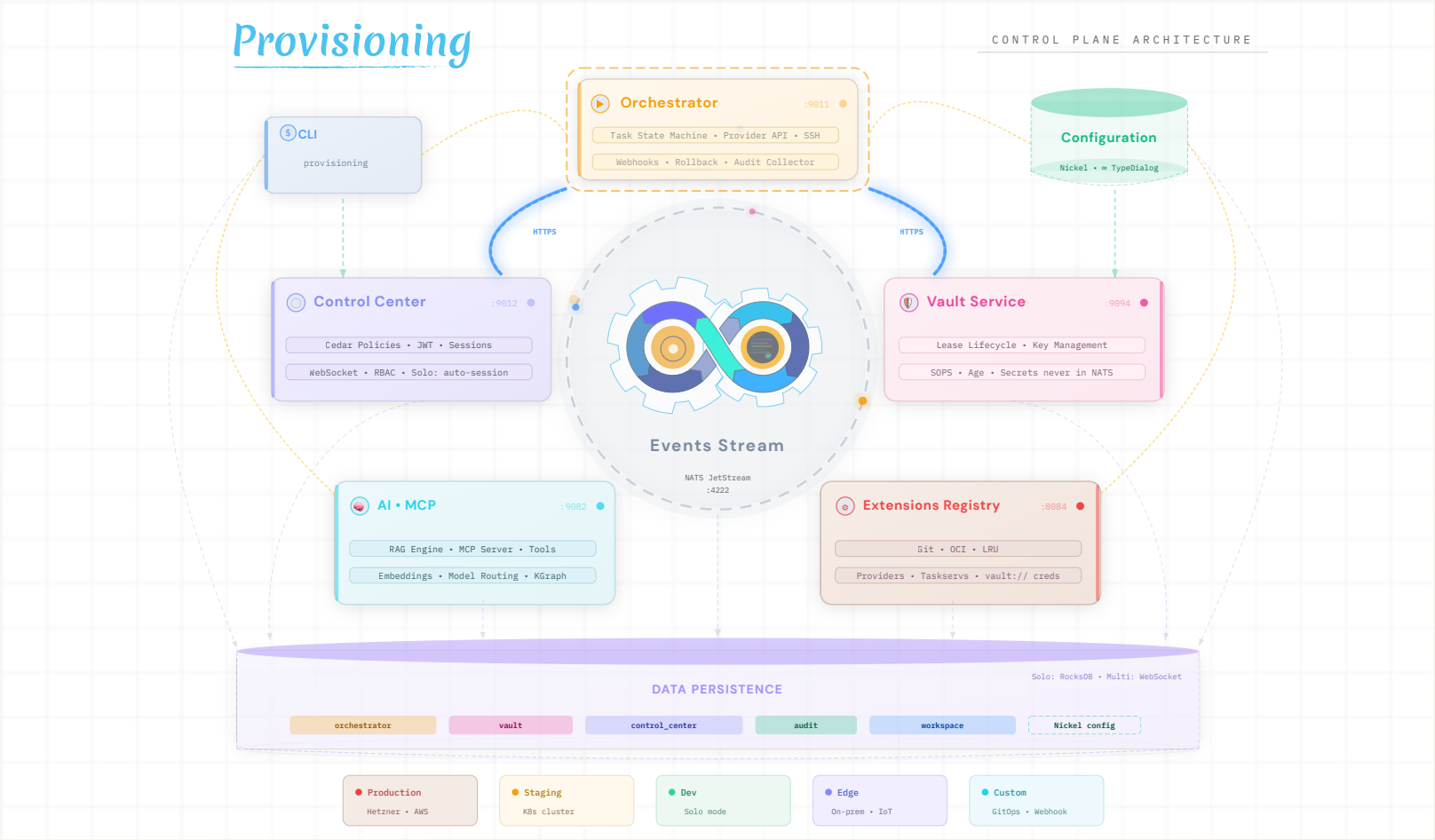
Rust

the platform

Orchestrator, provider traits, NATS services, control center, post-quantum vault.

~270k LOC · 28 crates

Rust didn't type my YAML. **Rust taught me what to demand** from everything around it.



Every arrow here is a contract or a typed event. The centre is the bus, not a service.

A component is a contract

```
# schemas/lib/contracts.ncl  (excerpt)
ComponentDef = {
  name       | String,
  mode       | [| 'taskserv, 'cluster, 'container |] | default = 'taskserv,
  depends_on | Array {
    name | String,
    kind | [| 'Requires, 'PrefersBefore, 'ConflictsWith |] | default = 'Requires,
  } | default = [],
  on_error   | [| 'Stop, 'Continue, 'Retry |] | default = 'Stop,
  requires   | {
    storage | { size | String, persistent | Bool } | optional,
    ports   | Array { port | Number, exposure | [| 'public, 'private, 'internal |] },
  } | default = {},
  provides   | {
    service   | String | optional,
    port      | Number | optional,
    endpoints | Array String | default = [],
  } | default = {},
  # mandatory — never omit
  concerns   | ServiceConcerns,
  ..
}
```

Concerns: typing what nobody types

```
# schemas/lib/concerns.ncl
ConcernState = { kind | [| 'enabled, 'disabled, 'pending, 'inherited |], .. }

ServiceConcerns = {
  tls | ConcernState,  dns | ConcernState,  certs    | ConcernState,
  backup | ConcernState,  observability | ConcernState,  security | ConcernState,
}

# a component that is honest about its gaps
concerns = {
  tls    = c.disabled "stateless service, no TLS termination",
  backup = c.pending  "policy to be defined" "BACKUP-001",
  ..
}
```

- You cannot **omit** a concern. You can only say *why* it's off — or *which ticket* closes it.
- "Which components have `backup = 'pending` ?" becomes a **query**, not an audit.

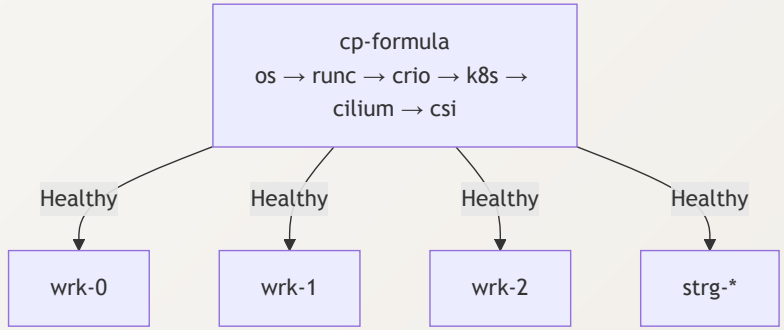
The graph is a value — and it is checked

```
# schemas/lib/formula.ncl (condensed) — runs at config time
let _Formula = std.contract.custom (fun label value =>
  let node_ids = value.nodes |> std.array.map (fun n => n.id) in
  let bad_deps = value.nodes |> std.array.flat_map (fun node =>
    node.depends_on
      |> std.array.filter (fun dep => !(std.array.elem dep.node_id node_ids))
      |> std.array.map (fun dep => "node '%{node.id}' depends_on unknown '%{dep.node_id}'")
  ) in
  if bad_deps != [] then
    std.contract.blame_with_message "invalid depends_on: %{std.string.join ", " bad_deps}" label
  else 'Ok value
)
```

- Duplicate node ids, dangling `depends_on`, bad edge endpoints → **rejected**.
- A formula = DAG of components on one server. `max_parallel`, `on_error` are policy **on the graph**.

Graphs of graphs: the composition

```
# workspaces/libre-wuji/infra/libre-wuji/dag.ncl
composition.formulas = [
  { formula_id = "libre-wuji-cp-formula",
    depends_on = [],
    health_gate = {
      check_cmd = "kubectl get nodes … | grep -v NotReady
      retries   = 18,
    } },
  { formula_id = "libre-wuji-wrk-0-formula",
    depends_on = [{ formula_id = "libre-wuji-cp-formula",
                    condition  = 'Healthy }],
    parallel   = true },
  # wrk-1, wrk-2, strg-* …
]
```



Same discipline, one level up: components → formulas → infrastructure.

Exactly one of — the lesson I hadn’t applied

```rust
// what Rust wanted from the start: two states, not four
pub enum NodeKind {
    TaskServ { taskserv: TaskServDef },
    Component { component: ComponentDef },
}

pub struct FormulaNode {
    pub id: String,
    pub kind: NodeKind,
}

match &node.kind {
    NodeKind::TaskServ { taskserv } => run_taskserv(taskserv),
    NodeKind::Component { component } => run_component(component),
}
```

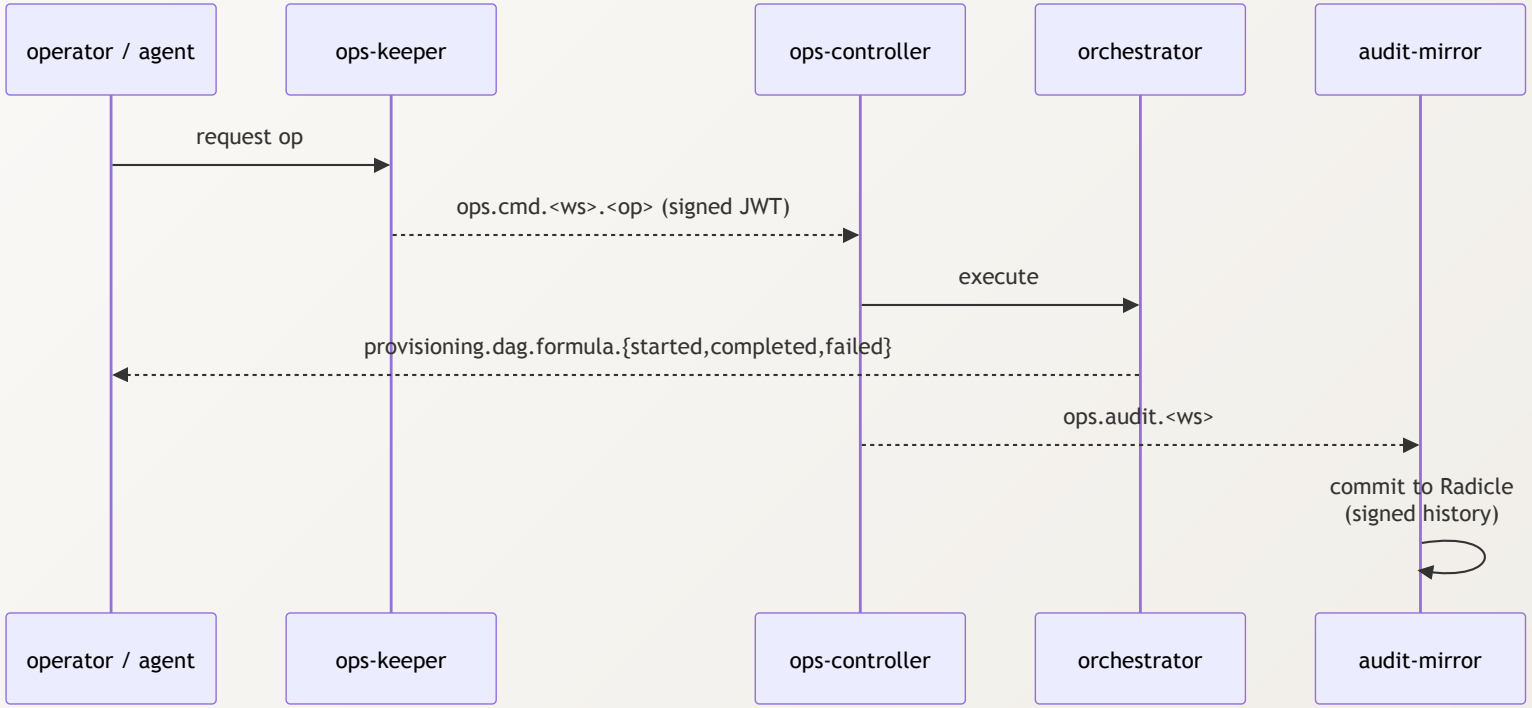
The `bail!` arms disappear because the state can't be built.

Providers satisfy a contract — the core doesn't move

```rust
// platform/crates/provisioning-compute/src/trait_def.rs
#[async_trait]
pub trait ComputeProvider: Send + Sync + Debug + 'static {
    fn kind(&self) -> ProviderKind;
    async fn spawn(&self, spec: InstanceSpec) -> Result<Instance, ComputeError>;
    /// MUST be idempotent: destroying an unknown handle returns Ok(()).
    /// MUST NOT return NotFound — that case maps to Ok(()).
    async fn destroy(&self, handle: &InstanceHandle) -> Result<(), ComputeError>;
    async fn status(&self, handle: &InstanceHandle) -> Result<InstanceStatus, ComputeError>;
    async fn list(&self, label_selector: &str) -> Result<Vec<Instance>, ComputeError>;
    async fn health(&self) -> Result<ProviderHealth, ComputeError>;
}
```

- Adding a provider = implementing a trait + a Nickel schema. Not editing a core — even your own local infra, or any Nushell script behind a JSON envelope.
- `#[non_exhaustive] enum ComputeError` — failure modes are part of the contract.
- But read the doc comments: **"MUST be idempotent"** is an invariant the type can't hold. Only tests can — so every provider, yours included, runs the same conformance suite: `run_suite(provider)`.

Coordination through events, not calls



- Payloads are Rust structs serialized with `serde` (JSON on the wire), on JetStream.
- Every mutation is signed going in and **auditable as a git history** coming out.

Live: break the contract

```
let f = import "schemas/lib/formula.ncl" in
let Acyclic = import "acyclic.ncl" in

f.make_formula {
  id          = "rustlab-live",
  description = "control plane, typed",
  provider    = "hetzner",
  server      = "libre-wuji-cp-0",
  nodes = [
    f.make_node { id = "os",         taskserv = { name = "os" } },
    f.make_node { id = "runc",       taskserv = { name = "runc" },       depends_on = [{ node_id = "os" }] },
    f.make_node { id = "crio",       taskserv = { name = "crio" },       depends_on = [{ node_id = "runc" }] },
    f.make_node { id = "kubernetes", taskserv = { name = "kubernetes" }, depends_on = [{ node_id = "crio" }] },
  ],
}
```

Live: break the contract

as written

✓ **contracts satisfied — nothing touched a server**

OS	∅
RUNC	← os
CRIO	← runc
KUBERNETES	← crio

runc → rnuc

✗ **contract broken**

```
Formula 'rustlab-live' has invalid depends_on:
node 'crio' depends_on unknown 'rnuc'
```

os depends_on kubernetes

✓ **contracts satisfied — the cycle passes**

OS	← kubernetes
RUNC	← os
CRIO	← runc
KUBERNETES	← crio

… } | Acyclic

✗ **contract broken**

```
Formula 'rustlab-live' has a dependency cycle
through: os, runc, crio, kubernetes
```


What Rust taught me
...and where the analogy breaks

Ownership, mapped — as analogy, not mechanism

Rust	Infrastructure
Single owner	One component owns a resource; others **borrow** its `provides`
`&` vs `&mut`	Read endpoints freely; mutating someone else's resource is a design error
Lifetimes	"This credential is valid for *this* deploy session"
`TryFrom` / validated constructors	Contracts that reject at the boundary
`enum` over `Option × Option`	`'pending "reason" "BACKUP-001"` instead of a missing field
Trait bounds	"Satisfy `ComputeProvider` and you're a provider"

> *Aliased mutable state is the root of most data races.*
> ***Two systems believing they own the same resource*** *is the root of most infra incidents.*

Why rollback is not `Drop`

I wanted this:

```
{
    let plan = Plan::apply(&ctx)?;
    plan.verify()?;
    plan.commit();
} // dropped uncommitted → rollback
```

- `Drop` is **synchronous**. Cloud APIs are not.
- `Drop` **cannot fail**. Deleting a volume can.
- `Drop` order is lexical. Teardown order is a **graph**.

What I have:

```
#[async_trait]
pub trait ProviderRollbackHandler: Send + Sync {
    async fn rollback_resource(&self, r: &ResourceSnapsho
        -> Result<()>;
    async fn cleanup_resources_after(&self, cp: &Checkpoi
        -> Result<Vec<String>>;
    async fn validate_rollback(&self, cp: &Checkpoint)
        -> Result<Vec<String>>;
}
```

Explicit, checkpointed, per provider — and **validated before it runs**.

The bug I found preparing this talk

```rust
use std::collections::{HashMap, HashSet, VecDeque};

#[derive(Default)]
struct DependencyGraph {
    nodes: HashSet<String>,
    edges: HashMap<String, Vec<String>>,         // task -> its dependencies
    reverse_edges: HashMap<String, Vec<String>>, // dependency -> its dependents
}

impl DependencyGraph {
    fn add_dependency(&mut self, from: &str, depends_on: &str) {
        self.nodes.extend([from.to_string(), depends_on.to_string()]);
        self.edges.entry(from.into()).or_default().push(depends_on.into());
        self.reverse_edges.entry(depends_on.into()).or_default().push(from.into());
    }

    // same logic as orchestrator/src/workflow.rs:174
    fn topological_sort(&self) -> Result<Vec<String>, String> {
        let mut in_degree: HashMap<&str, usize> =
            self.nodes.iter().map(|n| (n.as_str(), 0)).collect();
        for deps in self.edges.values() {
            for dep in deps {
                *in_degree.entry(dep).or_default() += 1;
```

The bug I found preparing this talk

as in workflow.rs

compiled — and the logic is wrong

```
ERROR       Circular dependency detected in
            workflow

property FAILED — case 0: acyclic graph of 4 nodes
rejected: Circular dependency detected in workflow
```

in-degree counted on the task

compiled and ran

```
ORDER       runc → crio → kubernetes → cilium

PROPERTY    1000 random DAGs, every edge
            respected ✓
```

The bug I found preparing this talk — continued

```
impl DependencyGraph {
    // same logic as orchestrator/src/workflow.rs:174
    fn topological_sort(&self) -> Result<Vec<String>, String> {
        let mut in_degree: HashMap<&str, usize> =
            self.nodes.iter().map(|n| (n.as_str(), 0)).collect();
        for deps in self.edges.values() {
            for dep in deps {
                *in_degree.entry(dep).or_default() += 1;
            }
        }
        let mut queue: VecDeque<&str> =
            in_degree.iter().filter(|(_, d)| **d == 0).map(|(n, _)| *n).collect();
        let mut result = Vec::new();
        while let Some(node) = queue.pop_front() {
            result.push(node.to_string());
            for dependent in self.reverse_edges.get(node).into_iter().flatten() {
                let d = in_degree.get_mut(dependent.as_str()).ok_or("unknown node")?;
                *d -= 1;
                if *d == 0 { queue.push_back(dependent); }
            }
        }
        if result.len() != self.nodes.len() {
            return Err("Circular dependency detected in workflow".into());
        }
        Ok(result)
    }
}
```

Types check shape. Tests check meaning.

- `Result<Vec<String>>` is satisfied by the right order, the wrong order, and an `Err` that lies.
- Graph algorithms need **properties**, not examples:

```
proptest! {
    #[test]
    fn topo_order_respects_every_edge(g in arb_dag(1..64)) {
        let order = g.topological_sort().unwrap();
        let pos: HashMap<_, _> = order.iter().enumerate().map(|(i, n)| (n, i)).collect();
        for (task, deps) in &g.edges {
            for dep in deps { prop_assert!(pos[dep] < pos[task]); }
        }
    }
}
```

Typed composition moved my bugs. It did not delete them.

The boundary is where types leak

Nickel `ComponentDef` — 20+ fields

```
mode      | [| 'taskserv, … |] | default = 'taskserv,
requires  | { … },
provides  | { … },
concerns  | ServiceConcerns,    # mandatory
..                               # open record
```

Rust `ComponentDef` — 7 fields

```rust
#[derive(Deserialize)]
pub struct ComponentDef {
    pub name: String,
    #[serde(default = "default_deploy_mode")] // Cluster!
    pub mode: DeployMode,
    // requires, provides, concerns:
    // silently dropped by serde
}
```

- Two type systems, one concept, **two defaults**.
- `serde` ignores unknown fields unless you say `#[serde(deny_unknown_fields)]`.
- Fix: one source of truth + a CI test that exports every Nickel fixture and deserializes it strictly.

A contract protects only where it’s applied

```
provides | {
  service   | String | optional,
  port      | Number | optional,
  # Added 2026-08-18: the surrealdb catalog component has always declared
  # provides.protocol = "http+ws" and this closed record rejected it, so the
  # component could be listed in settings.ncl (which does not apply
  # ComponentDef) but not used as a DAG component node, which does. The
  # incompatibility stayed invisible until a workspace first tried to
  # schedule it through the unified model.
  protocol  | String | optional,
```

> *Gradual typing is a feature — and a place for bugs to wait. Know **where** your contracts run.*

External state doesn’t typecheck

- `condition = 'Healthy` is a typed enum. What *Healthy* means is `kubectl get nodes | grep -v NotReady`.
- `200 OK` from a control plane often means *queued*, not *done*.
- Eventual consistency, rate limits, idempotency windows: none of it is in a signature.
- I can validate the **plan**. I cannot validate **the cloud**.

So the seam is explicit: health gates with budgets, retries, events and an audit trail.
Operational, not structural. Observed, not proven.

What I'd do differently

1. **Sum types end to end.**
   If the config language can't express them, generate the Rust side — don't hand-copy `Option × Option`.
2. **Typed errors in the composition layer.**
   It still returns `anyhow`. The providers already have `#[non_exhaustive]` enums; the graph deserves one.
3. **One executor.**
   Nushell runs the live DAG; a Rust engine sat beside it with no tests on its sort. Two engines = two semantics.
4. **Property tests on every graph algorithm.**
   From day one — a `Result<Vec<String>>` checks the shape of the answer, not its order.
5. **Check acyclicity where the graph is declared.**
   Not where it's executed — the cycle passed `dag.ncl` and only Rust saw it.

Same paradigm, another subject

	Rust · the code	provisioning · infrastructure	ontoref · projects
Declare	types, enums	`ComponentDef` , formulas	axioms, tensions, practices — `core.ncl`
State	typestate	op status, health gates	`state.ncl` : current → desired, with blockers
Check before acting	the compiler	contract before deploy	a decision checked against invariants · pre-commit `act-gate`
What counts as done	`Result` , not exit 0	the health gate, not "applied"	a step's verdict is its check's exit code, not its report
Trace	—	signed ops → Radicle ledger	signed work order + receipt

Ontology — what the project *is*

form · the plan

Reflection — what it *becomes*

flow · the cloud

What this changes for AI agents

Helps

- **Context is queried, not remembered.** The agent asks the project what is true (`ontoref describe … --actor agent`) instead of guessing from files.
- **The agent proposes, the contract disposes.** Same validators over MCP as on the CLI; proposals land unsigned on `ops.pending.<ws>` , only signed commands reach `ops.cmd.<ws>` .
- **Its word is not the verdict.** A step's status is its check's exit code; a contradicting self-report is refused.
- **Human acts stay human.** Only a person ratifies or accepts; an agent signs only as itself. Its reach is a declared axis: `'AgentBoundary` .

Doesn't solve

- **Well-typed and wrong.** The `topological_sort` compiled.
- **Declared ≠ read.** An agent cites a mechanism nothing executes — exactly as people do.
- **"There is no check" is a claim too.** Absence needs the same proof as presence.
- **Every contract is adoption cost.** More form, more friction.

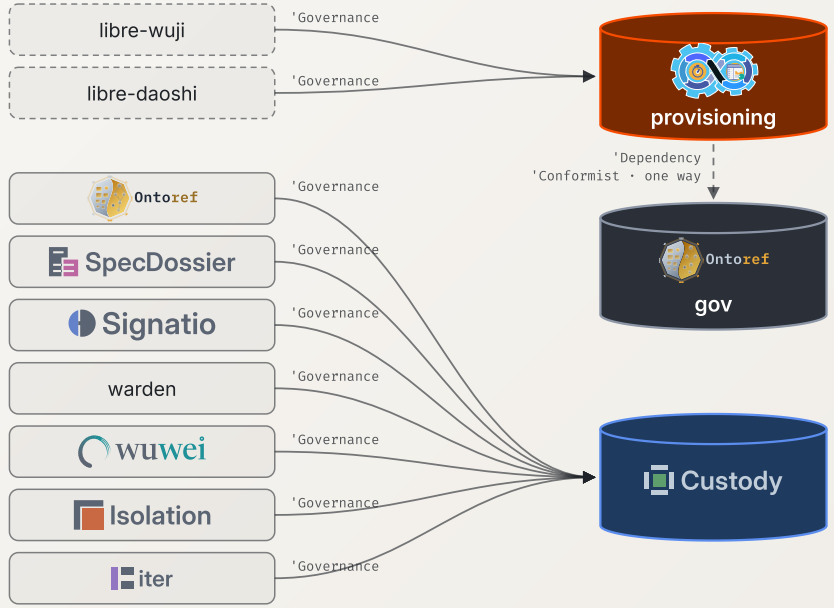
Agents scale authorship. Contracts and witnesses scale trust.

The same discipline, one level up

```
# a workspace → its domain (libre-daoshi/.domains-ontoref/…)
{ id = "libre-daoshi-to-provisioning",
  from = "libre-daoshi", to = "provisioning",
  axis = 'Governance, state = 'Node, resolution = 'Merge }

# a project → gov, one way (provisioning/.domains-ontoref/…)
{ id = "provisioning-to-gov",
  from = "provisioning", to = "gov",
  axis = 'Dependency, state = 'Linked,
  dep_kind = 'Conformist, mutual = false }
```

- `'Governance` **is a DAG.** `bond validate` walks it and reports a cycle as an error.
- `'Dependency` **is a plane.** Never walked for cycles: a mutual pair would be legal — by type, not by exception.
- `'Merge` : the node wins per id; the rest is inherited from the domain.



labels are bond axes · dashed = private · full URLs on the last slide

The cycle that slipped through my DAG is an error one level up.

Where this talk's ideas came from

career.ncl → core.ncl · only edges the ontology declares · scripts/cv-graph.nu

manifests in — validated by

1987–2004
Olivetti Dealer (EMISA) → DDM Information Systems → DataSys - Hispatec Group → AdmiSys S.L. → SCO / Caldera / Tarantella

2005–2009
Freelance
Programmer

2010–2017
Soluciones Digitales
Consultant, Programmer & Online Business...

2018–today
LibreCloud
Provisioning

2022–2024
Technology Innovation Institute
Software Developer — Rust & Integration...

2024–today
ontoref · operational ontology
Ontoref · Operational Ontology

Tool Boundary YANG
Rust gives determinism within the model you build. Building the wrong model is yours

Form and Flow SPIRAL
Form without flow is rigidity. Flow without form is chaos

DAG Retrospective YIN
The DAG does not forbid looking back — it forbids causal flow returning

Three Layers SPIRAL
Nickel: correctness at definition time. Nushell: observability at operation time

Typed DAG YANG
A typed DAG is a control flow graph the compiler verifies before you deploy

Yang/Yin Projects SPIRAL
The ecosystem has a yang pole (Provisioning, StratumlOps, TypeDialog, SecretumVault — form, structure, determinism) and a y...

Types Shape Water YANG
Types give water its shape. Without form: continuous hope

The Map Is Not The Territory YANG
Any model captures aspects of reality, never the totality

Provisioning
Declarative IaC with Nickel + Nushell + Rust. Typed dependency DAG. What sustains. v3.0.11

Ontoref
Self-describing project ontology protocol. Structure that remembers why. v0.1.0

Operational Ontology
Personal operational ontology (jpl-ontoref)

Takeaways

1. **Type the composition, not only the resources.** The edges are where incidents live.
2. **Absence must be explicit.** `'pending "why" "TICKET-42"` beats a missing field.
3. **The discipline recurses.** Components → formulas → infrastructure → projects and their bonds.
4. **Seams need their own tests.** Config ↔ Rust, plan ↔ cloud, type ↔ meaning.
5. **Agents propose; contracts, checks and people dispose.** Know where each guarantee ends.

If it doesn't typecheck, it doesn't touch a server.

And if it typechecks, it still has to be true.

Thank you

Questions?

Jesús Pérez Lorenzo
jesusperez.pro
jpl@jesusperez.pro

Feedback · 1 min





from the Canary Islands
rust-laspalmas.dev


Going deeper
numbers · comparisons · demo outputs · links

Numbers

Rust	28 workspace crates · ~270k LOC
Nickel	984 files · ~71k LOC
Nushell	895 files · ~199k LOC
Production workspace	`libre-wuji` · Hetzner fsn1 · 71 component definitions
Compute providers	Hetzner · UpCloud · AWS · DigitalOcean · local
Bus	NATS JetStream · JSON payloads · `serde` structs
Audit	`ops.audit.*` → Radicle signed history

← the stack

Why not all of it in Rust?

Nickel

WHO WRITES IT: OPERATORS, AGENTS — NOT THE PLATFORM BUILD

- Config edited and **validated without recompiling** the platform
- Contracts and merge are the language, not a library you call
- Blame points at the line of config that broke the contract

in Rust: a DSL or `build.rs` — every config change is a rebuild, every contract lives where only Rust readers see it

Nushell

WHAT IT TOUCHES: PROVIDER CLIS, SSH, FILES

- Glue as **structured data** — tables and records, not strings
- A provider can be a script: `ScriptProvider`, checked by the same conformance suite
- Changed in place, on the machine that runs it

the cost: ~199k LOC, the live DAG executor runs here — and that seam is where types leak

Rust

WHAT MUST NOT FAIL

- Orchestrator, NATS services, control center, post-quantum vault
- No GC, one static binary, `Send + Sync` checked
- Failure in the signature: `Result`, `#[non_exhaustive]`

and still: the `topological_sort` bug compiled — tests caught it

Each layer is typed in the tool its authors can change.
Rust set the bar; it doesn't have to do all the typing.

Why not Terraform / Pulumi / CUE / KCL / Crossplane?

	Terraform OpenTofu	Pulumi	CUE	KCL	Crossplane	provisioning
Typed resources	● provider schemas	● host-language types	● unification	● schemas + `check`	● CRD / OpenAPI	● Nickel contracts
Composition is a **declared value**	◐ implicit from references	○ built by running code	◐ data, no executor	◐ data; KusionStack runs it	◐ Compositions, eventual order	● **formula + composition DAG**
Policy on the graph error policy · parallelism · health gates	○ global `-parallelism`	◐ written in code	○ —	○ not in the language	◐ readiness checks	● `on_error` · `max_parallel` · `health_gate` **per node**
Cross-cutting concerns explicit	○ tags, convention	○ convention	◐ encodable	◐ encodable	○ convention	● **6 typed concerns,** `'pending` **+ ticket**
Illegal graph rejected **before anything runs**	● refs + cycles at validate	◐ preview runs the program	● what you encode	● what you encode	○ found at reconcile	◐ refs at config time · cycles only in Rust
Drift: continuous reconciliation	◐ plan shows drift	◐ refresh	○ —	○ —	● the model	◐ gates + events · no reconcile loop
Ecosystem · providers	● thousands	● large	◐ growing	◐ CNCF sandbox	● large	◐ 5 in the catalog · **any by contract** — a script, your own infra — checked by a conformance suite · one maintainer

Not a replacement — a different layer:
the edges between resources, typed and checked before anything runs.

● strong · ◐ partial · ○ absent — my assessment, for the composition layer only

"You could do this in any language"

Yes — the discipline travels

- The composition is typed in **Nickel** and run by **Nushell**. Port the executor to Go and the contracts still hold.
- That is the point of the talk: a Rust habit, exported to the part of the system that is *not* Rust.
- And Rust did not save me from a wrong `topological_sort`. Tests did.

What Rust carries — all in one compiler

Sum types + exhaustive `match`	illegal states unbuildable — `NodeKind`, not `Option × Option`
`Result` + `#[non_exhaustive]`	failure is part of the signature — `ComputeError`
`Send + Sync + 'static`	sharing a provider across tasks checked at compile time
No GC · one static binary	daemons that must not stall: ops-controller, ncl-sync, a post-quantum vault
`serde` **at the boundary**	the seam typed both ways — and you see where it leaks

Rust is where I learned the discipline — and where the parts that must not fail live.

Federation (ADR-042)

- Projects declare **Integration Modes**; contracts ship as **OCI artifacts** (`application/vnd.ontoref.mode.v1`) via `oras` .
- Workspaces consume them as typed imports.
- Cross-project impact is a bounded BFS: `impact_graph(slug, node, max_depth)` , depth ≤ 5.

Same three layers — shape, form, composition — one level further up.

Links

Jesús Pérez Lorenzo jesusperez.pro linkedin.com/in/jesusperez jpl@jesusperez.pro

INFRASTRUCTURE

Provisioning	the platform in this talk provisioning.systems
SecretumVault	post-quantum secrets vault · ML KEM-768 + DSA-65 repo.jesusperez.pro/jesus/secretumvault
lian build	ephemeral build substrate · provider-pluggable, content-addressed cache lian.build
lamina layers	builder layers · base images + pre-cooked dependency caches lamina.lian.build
solera runtime	runtime service images · built with lian-build solera.lian.build
Rustelo dev	full-stack Rust web framework · Axum + HTMX · Leptos optional rustelo.dev

COMMUNITY

RUST Las Palmas	Rust Las Palmas · Gran Canaria · co-organizer rust-laspalmas.dev
Feedback	1-minute survey about this talk · thank you jesusperez.pro/feedback/rustlab-2026

PROTOCOL · THE GOV AND CUSTODY DOMAINS, PART OF ONTOREF

Ontoref	ontology + reflection for projects · bonds ontoref.dev
Custody	the contract a safeguarded project is held to custody.ontoref.dev
SpecDossier	bounded, reviewed, agreed by digest specdossier.dev
Signatio	the signing capability signatio.dev
wuwei	agent sessions, observed wuwei.observer
Isolation	owner and agent on one host · what is shared, private, denied isolation.ontoref.dev
iter	part of isolation · work in a shared jj store, owner + agent iter.ontoref.dev

Cover: Piazza Maggiore, Bologna — photo Max Nayman, via Unsplash