WANAWARE RELATIONSHIP GRAPH + RELATIONSHIP DISCOVERY ENGINE (RDE)

Architecture Deep Dive for Technical Evaluation

How dependency structure is stored, kept current, and used for cause + impact reasoning.

How to use this deep dive

Who this is for

This document is for technical evaluators who need to verify that a dependency model holds up during real operations:

- **Platform and infrastructure architects** (model correctness, change impact)
- **SRE and incident response leads** (cause isolation, blast radius, noise reduction)
- **Security architects and technical evaluators** (scope of exposure, containment boundaries)

What this is (and isn't)

This is: an architecture deep dive focused on update behavior and traversal—how dependency structure is stored, kept current, and used for cause + impact reasoning.
This is not: a UI guide or feature catalog.

Quick navigation

- The modern dependency problem — why proving cause and scope is slow without dependency context
- What the Relationship Graph is — the data model and the questions it must answer
- Requirements driven by real operations — what "current, explainable, fast" must mean under stress
- Approaches that fall short — why lists, diagrams, and bolt-on graph views drift or slow down
- System overview: Evidence → RDE → Relationship Graph → Reasoning — how evidence becomes stored structure
- RDE: How relationships are built, updated, and kept trustworthy — normalization, deduplication, change, decay, conflicts
- Traversal and reasoning: cause, blast radius, and reducing noise — upstream cause vs downstream impact + explainable paths
- Data model and hierarchy: keeping services stable while assets change — assets → collections → service elements → structures

- Service mapping and health propagation: turning dependency structure into operational signals — roll-up health + redundancy awareness

- What to validate in the evaluation + close — a short checklist to test correctness and trustworthiness

After reading, you should be able to validate

- Understand what the Relationship Graph represents (data model, not a picture)
- Understand what RDE does (evidence → relationships → updates over time)
- See how traversal answers upstream cause and downstream impact questions
- Understand why the model stays current as environments change
- Use a short checklist to test correctness and trustworthiness during an **evaluation**

> If dependency structure is not current, explainable, and fast to traverse, it won't be used during incidents.

The modern dependency problem

Modern systems are made of interconnected parts. Applications rely on shared databases, identity services, networks, cloud infrastructure, and third-party platforms. These dependencies change frequently as workloads scale, move, and are reconfigured.

When something breaks, the slow part is often not the fix. The slow part is proving **what failed first** and **what else is affected**. Symptoms often show up downstream, far from the initiating cause.

A common incident pattern (one example)

A database tier degrades for a subset of requests. Minutes later:

- application error rates climb,
- user sessions fail,
- network tooling flags congestion,
- multiple teams see “their” alerts at the same time.

Without dependency context, responders investigate each signal independently. The result is parallel triage, duplicated work, and slow convergence on the initiating failure.

Why list-based tools struggle under pressure

Many operational tools treat infrastructure as a list. They can tell you what exists, but they often struggle to answer dependency questions consistently in real time.

During incidents, teams end up reconstructing “what depends on what” manually. That work is time-consuming, brittle, and hard to repeat—especially when the environment is changing while you troubleshoot.

What WanAware does differently

WanAware is built around a different assumption: **dependency structure is primary data.** Instead of rebuilding relationships during an investigation, WanAware maintains relationships continuously so teams can reason about cause and impact directly.

What the Relationship Graph is

The Relationship Graph is a continuously updated model of assets and their dependencies.

- **Nodes** represent entities in the environment, including infrastructure components, logical services, identities, and business structures.

- **Relationships** represent real “depends on” links, such as a service relying on a database, a workload running on a host, or a business service supported by an application tier.

The Relationship Graph is the platform’s data model. Dependency questions are answered by traversing stored relationships, not by reconstructing context from static diagrams or joins across tables.

Practical questions the graph must answer

- What does this rely on?
- What relies on this?
- If this fails or is compromised, what else is affected?
- Which services and business functions are involved?

Requirements driven by real operations

For a dependency model to be useful during incidents, change planning, and security response, it must behave well under stress.

Operational requirements

What evaluators should expect

Multi-hop traversal

Most investigations require multiple hops upstream and downstream. The system must return results quickly at depth.

Current state

If relationships lag reality, responders stop trusting the answers. The model must update as environments change.

Explainability

When the system identifies an initiating failure or an impact set, teams must be able to see the dependency path behind the conclusion.

Cause vs symptom separation

Downstream effects should be grouped as consequences when they share an upstream cause.

Service and business context

The model must connect technical conditions to services and organizational structures so teams can prioritize based on real impact.

Watch-out callout

A dependency model that only supports visualization is not sufficient. During incidents, teams need fast traversal and inspectable paths—not a static map.

Approaches that fall short

Many teams attempt dependency reasoning with tools not built for it:

- **List-based inventories and CMDBs** — Great for records. Dependency reasoning often requires joins and traversal at query time. As depth increases, investigations slow down.
- **"Graph views" on top of non-graph storage** — Helpful for visualization, but traversal is constrained by the underlying storage and update model.
- **Manual service maps and diagrams** — Useful for communication, difficult to keep current. Once drift sets in, they stop helping during incidents.
- **Correlation-based RCA systems** — Helpful for anomaly patterns, but often struggle to separate cause from coincidence when topology changes.

System overview: Evidence → RDE → Relationship Graph → Reasoning

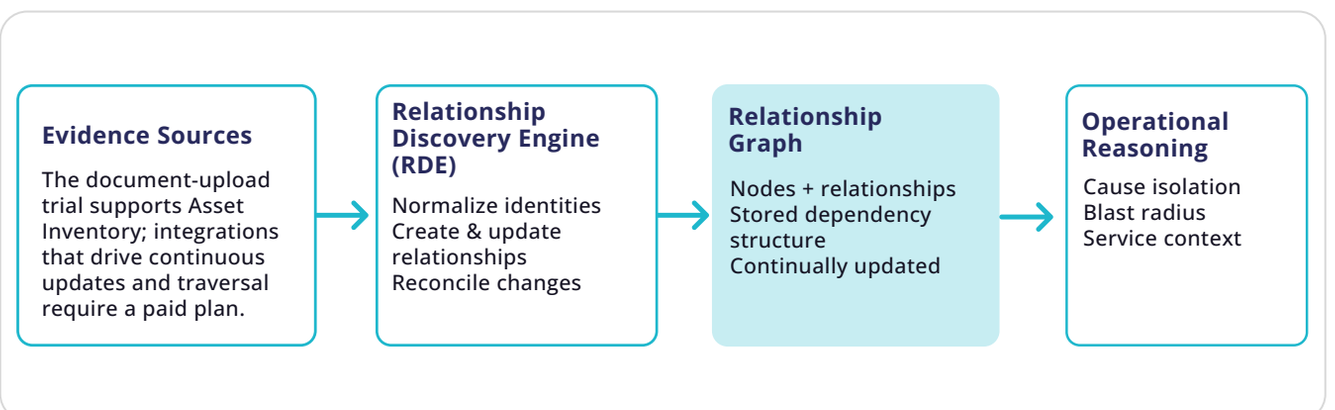


Figure 1: Relationships are created and updated as evidence arrives, then traversed directly to support cause isolation and impact analysis during real operation.

WanAware's architecture is designed around one requirement: dependency structure must be current, inspectable, and fast to traverse during real operations.

Traversal is optimized for multi-hop queries and returns results with pagination/limits when fan-out is large.

At a high level, the platform works in four layers:

- Evidence sources
- RDE ingestion and normalization
- Relationship Graph storage
- Traversal and reasoning consumers (incident response, change planning, security scoping)

Why the overview matters

Traditional approaches often reconstruct relationships during analysis. WanAware's model is different: relationships are created and updated when evidence arrives, so traversal is performed on stored structure.

Key idea

If the model is built "just in time," it will fail precisely when you need it most.

RDE: How relationships are built, updated, and kept trustworthy

The Relationship Discovery Engine (RDE) is responsible for turning evidence into a usable dependency model.

It performs three continuous jobs:

1. Create or update nodes (assets, services, identities, structures)
2. Create or update relationships ("depends on," "runs on," "connects to," "supports")
3. Reconcile change over time so the graph reflects current state

Update behaviors evaluators should expect

Normalization

Evidence is normalized so equivalent entities resolve to consistent nodes (for example, the same service or workload referenced across tools).

Deduplication and reconciliation

When multiple sources describe the same entity or relationship, RDE reconciles

them into one representation. This reduces “double counting” and prevents parallel graphs from forming.

Change handling

When assets scale, move, or are reconfigured, RDE updates relationships so downstream reasoning reflects the new topology.

Retirement and decay

When evidence disappears for an entity or relationship, the model should reflect that change rather than keeping stale links indefinitely. Evaluators should understand how “no longer observed” is represented.

Conflict awareness

When evidence sources disagree, the system should avoid silent certainty. When sources disagree, we retain per-source evidence and expose a “conflict” state on the relationship until resolved. Evaluators should be able to see what relationship exists, why it exists, and what evidence supports it.

What breaks trust in dependency models

Teams stop using a model when they observe:

- dependencies that persist after changes,
- missing relationships they know exist,
- duplicate entities that fragment impact analysis,
- answers that cannot be explained with a path.

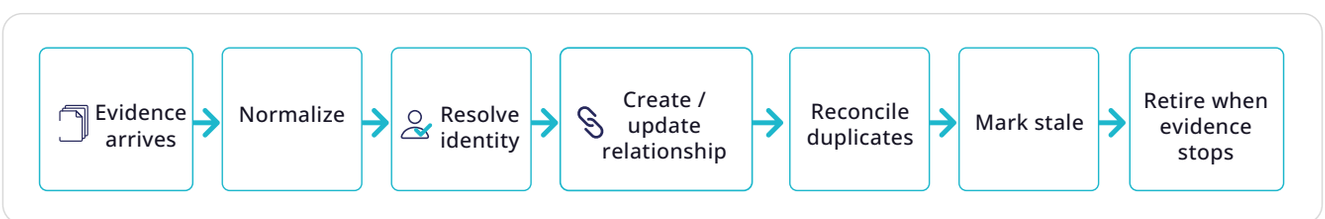


Figure 2: RDE update lifecycle
Evidence arrives→ normalize→resolve identity→create/update relationship → reconcile duplicates → mark stale/retire when evidence stops

Caption: Trust comes from current state, consistent identity, and visible update behavior, not from visualization alone.

Watch-out callout

A dependency model can be visually impressive and still operationally unreliable if it cannot stay current or explain why a relationship exists.

Traversal and reasoning: cause, blast radius, and reducing noise

The operational value of the Relationship Graph comes from traversal: walking dependency links to answer "what depends on what" at any depth.

Because nodes maintain direct links to related nodes, traversal proceeds step-by-step through stored relationships rather than repeated table joins or ad hoc reconstruction.

Two primary traversal directions

Upstream traversal (initiating cause)
Used to isolate shared dependencies and identify what failed first.
Typical question: "What common upstream dependency explains these downstream symptoms?"

Downstream traversal (blast radius / scope of impact)
Used to identify everything that relies on a starting point.
Typical question: *"If this component fails—or is compromised—what else is affected?"*

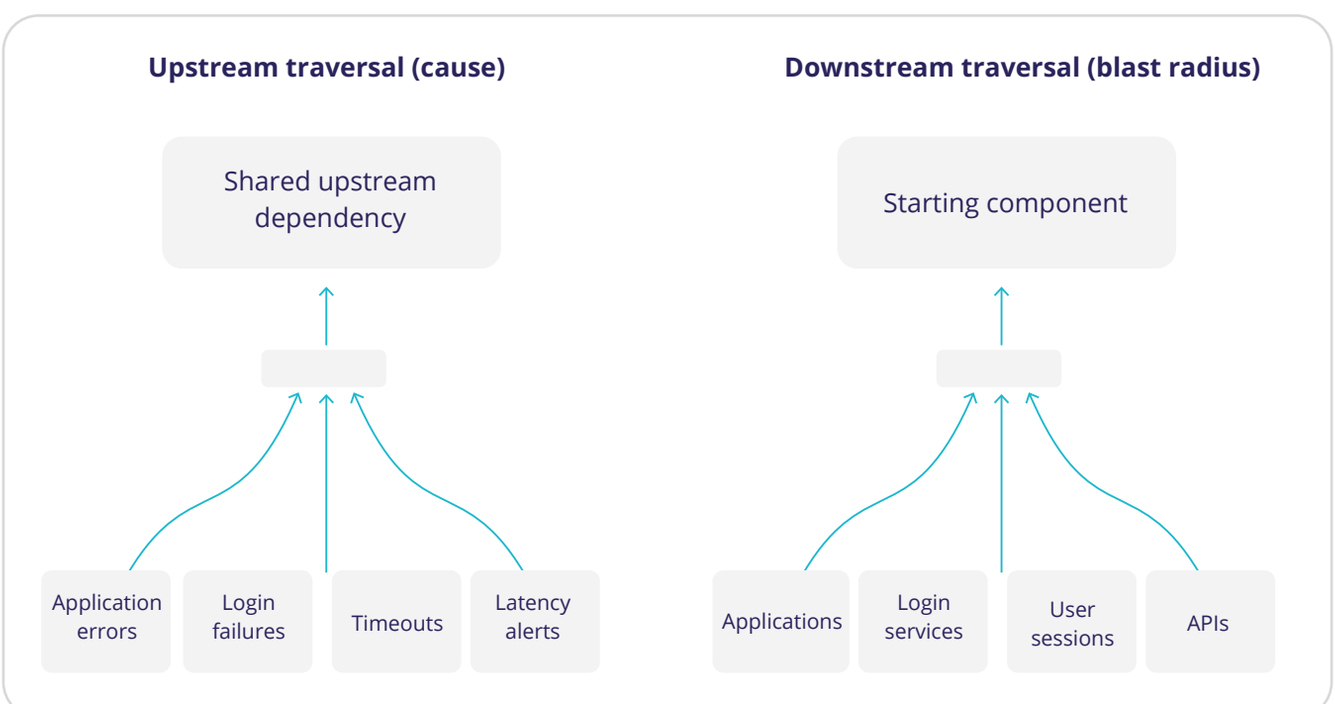


Figure 3: Same graph, different direction: cause (upstream) vs blast radius (downstream).

Explainability: the path is part of the answer

For evaluators, the key question is not only “what is the answer?” but “why should I trust it?”

WanAware supports explainability by preserving the dependency path that produced a result:

- which nodes were traversed
- which relationships connected them
- how the system moved from starting point to conclusion

This matters when teams need to validate an initiating cause or defend an impact set in a post-incident review.

Topology-based grouping: separating cause from symptom

Incidents often generate multiple alerts at the same time. Without dependency context, teams investigate each alert independently.

With topology-based reasoning:

- start from impacted nodes,
- traverse upstream to find shared dependencies,
- group downstream symptoms under the initiating cause.

The outcome is fewer parallel investigations and faster convergence on the component that best explains observed behavior.

> Given the same graph state and starting set, traversal makes cause, scope, and grouping repeatable.

Data model and hierarchy: keeping services stable while assets change

WanAware's Relationship Graph is organized to keep services stable while underlying assets change. This matters because evaluators need to test reasoning against an environment that is constantly scaling, moving, and being reconfigured.

The hierarchy (from dynamic to stable)

Assets (most dynamic)
Physical and virtual components such as hosts, workloads, network devices, storage resources, identities, and cloud resources.

Collections (stabilize change)
Groupings that represent technical tiers, clusters, or functional sets. Collections can remain stable even as individual assets scale up, down, or rotate.

Service elements (logical services)
Logical services or applications that deliver functionality. Ownership and service context can be associated here.

Structures (business and organizational context)
Groupings such as business units, regions, vendors, environments, or projects that help evaluators tie technical conditions to operational priority.

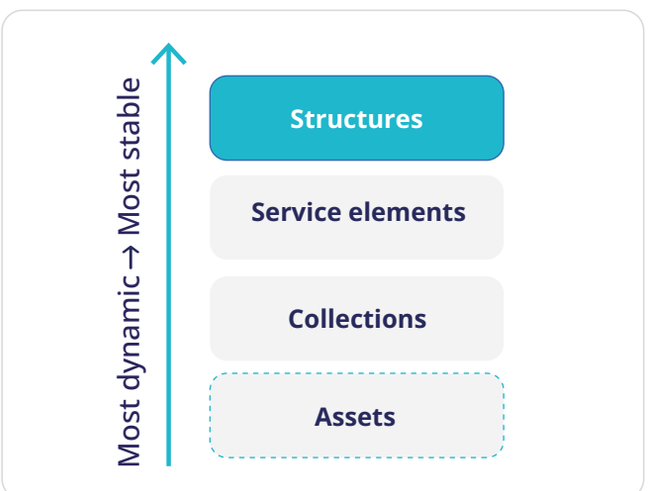


Figure 4: Dependency Stability Hierarchy. This model organizes system components by their rate of change. By anchoring high-churn Assets to more stable logical Structures, the architecture maintains a consistent operational view despite frequent underlying telemetry or infrastructure updates.

Why this hierarchy supports operations

Reasoning roll-up (bottom → top)
Technical conditions can roll up from assets to services and structures so teams can prioritize based on the service and business impact.

Targeting flow-down (top → bottom)
Teams can start with a service or structure, then traverse downward to identify the specific collections and assets that support it.

Key idea callout
A stable service model only helps if it stays connected to the changing asset reality underneath it.

A tiny example (one path)
Host runs workload → supports service element belongs to business structure

Evaluators should be able to inspect this path and confirm it updates as assets change.

Service mapping and health propagation: turning dependency structure into operational signals

Service mapping is most useful when it behaves like a live operational model, not a documentation artifact.

Health roll-up is policy-driven and configurable. Roll-up behavior reflects evaluator-defined policies, including which signals contribute to health and how thresholds are applied. Redundancy context is derived from observed structure (such as replicas, tiers, or capacity signals), not assumed from a single asset's status.

In WanAware:

- Services are represented as stable nodes.
- Supporting components connect beneath them through collections and assets.
- As assets scale, move, or rotate, the service representation remains intact.

Health propagation (how conditions roll up)
Health rolls up through the hierarchy so teams can answer:

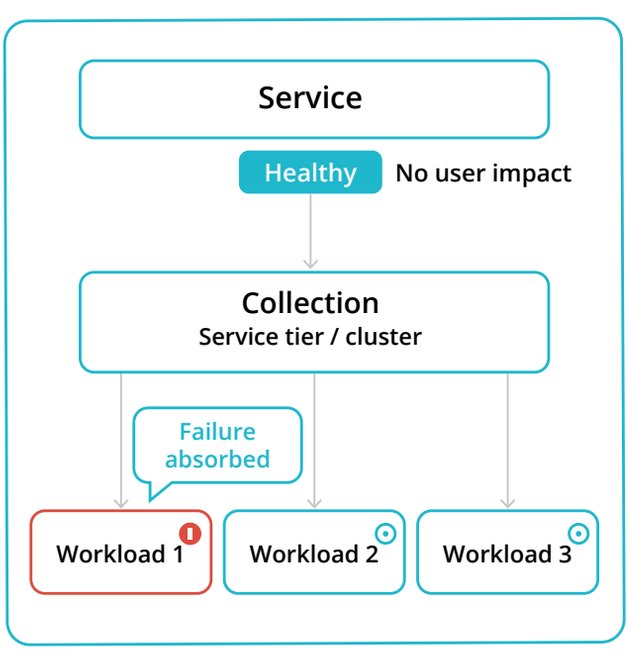
- "Is the service actually degraded?"
- "Which supporting components are driving risk?"
- "Is redundancy absorbing the issue?"

Redundancy awareness (avoid false emergencies)

A single component failure should not automatically imply service failure when enough capacity remains. A service view should reflect:

- component issues that are contained by redundancy, and
- component issues that exceed redundancy and create user impact.

Contained by redundancy
Single component issue; capacity still sufficient.



Redundancy exceeded (user user impact)
Multiple component issue; capacity no longer sufficient.

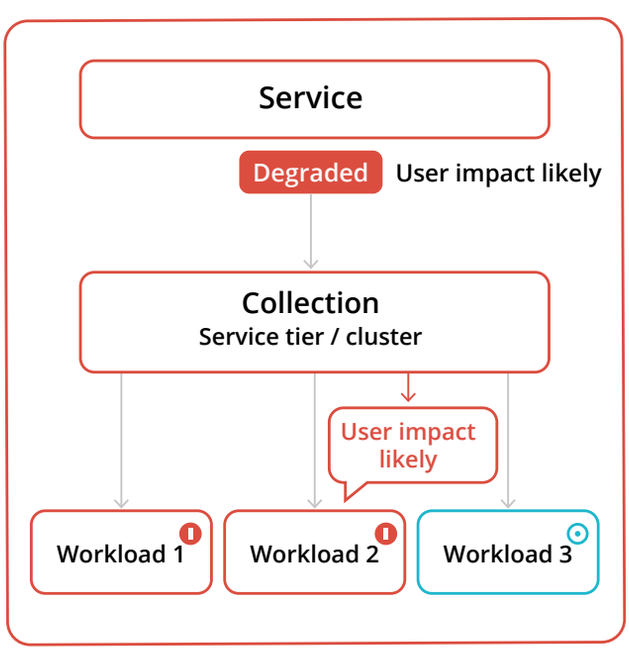


Figure 5: Service health reflects whether redundancy is still absorbing failures or has been exceeded.

Two views, same underlying model

The same dependency model can be viewed two ways, depending on who needs to act.

Engineering view
Shows the failing component and the dependency path that explains downstream symptoms.

Service view
Shows rolled-up service health and impact without promoting every component issue to a service degradation.

Watch-out
If service health flips due to single-asset noise, teams stop trusting it. Evaluators should test stability during partial failures and recoveries.

What to validate in the evaluation + close

Technical evaluators validate the system by testing a small set of behaviors that confirm it supports operational reasoning, not just visualization.

#	Do this	You should see	Red flags
1	Trigger a known change (scale, move, redeploy).	Nodes and relationships update automatically.	Stale links or duplicate entities.
2	Traverse 3–5 hops upstream and downstream.	Complete, reasonable paths at depth.	Shallow-only results or performance collapse.
3	Inspect the path behind a cause or impact result.	Readable node-by-node chain with labels.	Black-box answers or unclear inclusion.assets.
4	Select multiple related downstream symptoms	Convergence on a shared upstream dependency.	Symptoms treated as separate root causes.
5	Run downstream impact from a critical node.	Services and business structures affected.	Only immediate neighbors returned.asset list.
6	Simulate a partial failure where redundancy holds.	Service health remains stable unless exceeded.	Service health flips on single-component noise.

Figure 6: These checks validate correctness, trust, and operational usefulness.

Close

If these checks pass, you've validated that WanAware maintains dependency structure as current, inspectable data, and can use traversal to support cause isolation, impact scoping, and stable service reasoning during change and incidents.

Next step

Start Asset Inventory Trial

(Document-upload trial only. Integrations require a paid plan.)

Related: • **KDE**

www.wanaware.com