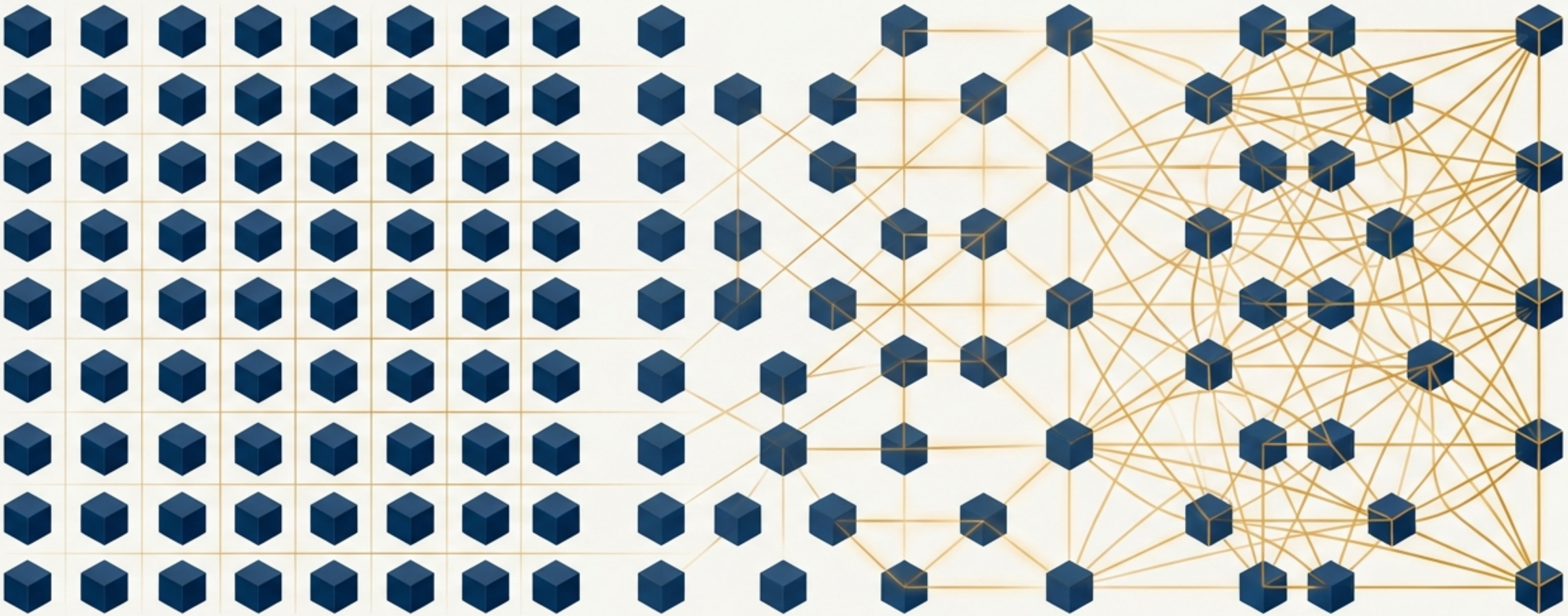
From Well-Formed to Meaningful

How Semantic Type Safety Infuses Runtime Checks with Real-World Context



We've Built Robust Systems with Runtime Type Safety

Modern frameworks like OSBot-Utils' Type_Safe library have already eliminated entire categories of bugs by moving beyond static hints to continuous runtime validation.

Continuous Runtime Checks

Every assignment and operation is validated instantly.

```
# Before: Fails deep in execution
store.prices["PROD-789"] = "not-a-number"

# After: Fails at assignment
# -> TypeError: Value 'nubr' is not a valid money value
```

Auto-conversion of Primitives

Wraps raw types in safe counterparts, improving developer experience without sacrificing safety.

Domain-Specific Primitives

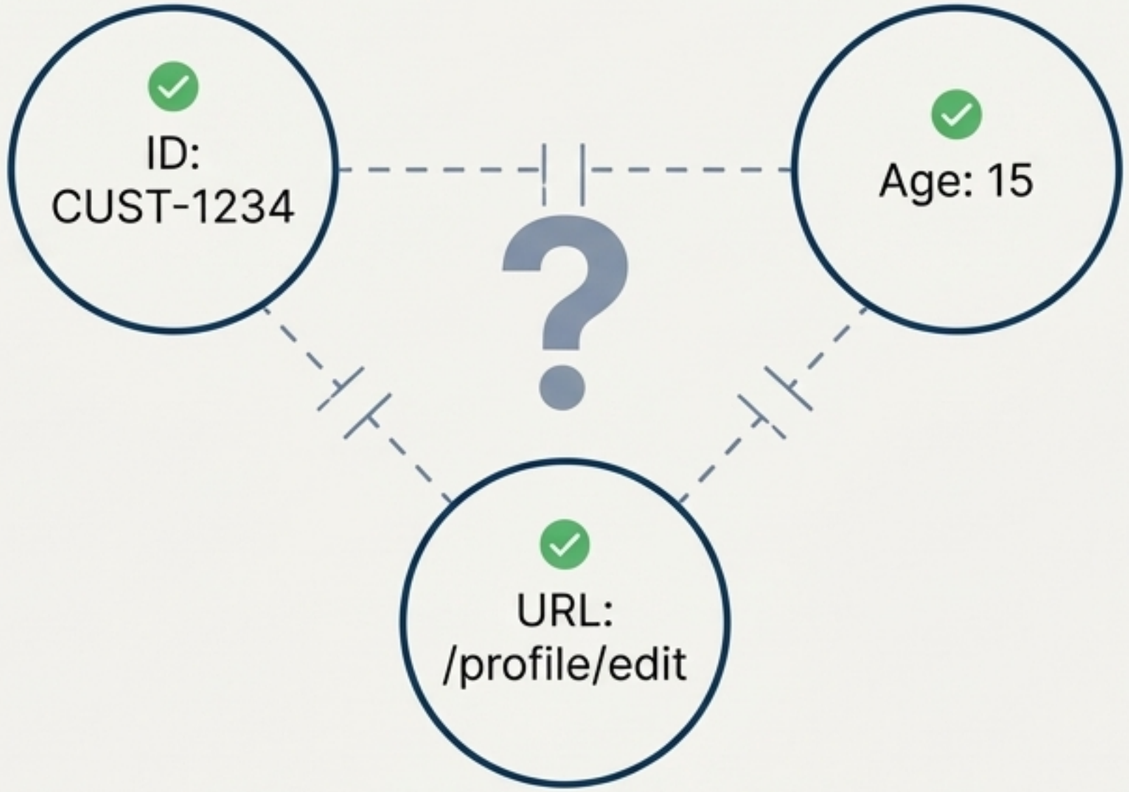
The real power lies in specialised types that encapsulate business rules.

```
Safe_Str__Username: # Enforces character/length limits

Safe_UInt__Age:     # Restricts values to 0-150

Safe_Str__Email:    # Ensures presence of '@'
```

By banning raw primitives (str, int, float), we have mastered **syntactic and format-level correctness.**

But Our Code Still Lacks a Critical Element: Context

Traditional type checks know nothing about the *meaning* of the data beyond its immediate constraints. They validate the *content format*, but not the *content meaning*.



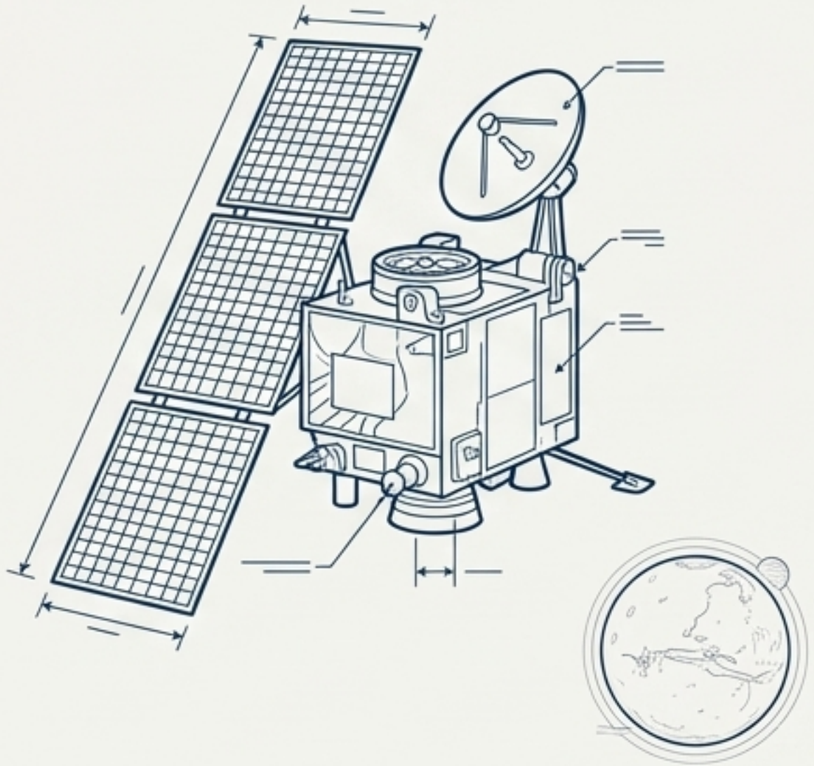
What if a value is perfectly formed, but fundamentally wrong in the bigger picture?

The £250 Million Misunderstanding

The 1999 NASA Mars Climate Orbiter was lost due to a software error where one system calculated thruster impulse in pound-force seconds (`lbf·s`) while another interpreted it as Newton-seconds (`N·s`).

The Root Cause

A failure of semantic context. To the software, both values were just 'numbers' (`float`).
The system had no concept of physical units.



What the code saw:

```
# Both systems saw this:
thruster_impulse: float = 4.45
```

What the context required:

```
# Lockheed Martin system (imperial)
thruster_impulse: { value: 1.0, unit: 'lbf·s' }
```

```
# JPL system (metric)
thruster_impulse: { value: 4.45, unit: 'N·s' }
```

Syntactically identical, semantically disastrous.

The Subtle, Everyday Bugs We Tolerate

This context gap isn't just for rocket science. It appears in our applications daily as logical flaws that syntactic checks cannot see.

The Phantom Customer

```
customer_id: Safe_Str__CustomerId
= "CUST-9999"
```

✅ Syntactic Check

Passes. The string matches the required alphanumeric pattern.

❌ Semantic Problem

Fails. Customer 'CUST-9999' does not actually exist in our database. The ID is valid in form, but not in reality.

The Underage Adult


```
user_age: Safe_UInt__Age = 15
```

✅ Syntactic Check

Passes. 15 is a valid integer within the 0-150 range.

❌ Semantic Problem

Fails. A business rule requires an adult (age ≥ 18) for a specific operation. The concept of 'adulthood' is semantic, not syntactic.

The Misleading Snippet

```
bio: Safe_Str__Html =
"<script>...</script>"
```

✅ Syntactic Check

Passes. The string may pass minimal filtering if it's technically valid HTML.

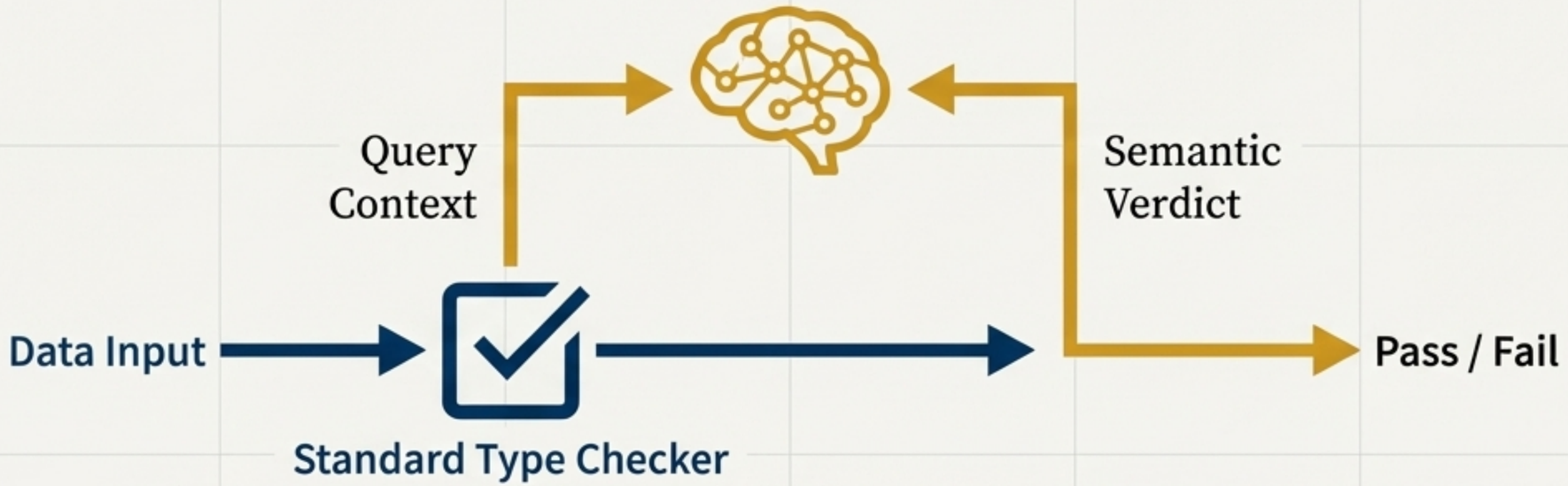
❌ Semantic Problem

Fails. The context expects a simple `<img>` tag or paragraph, not an executable script. The type doesn't understand the intended HTML structure.

Introducing Semantic Type Safety

Evolving Runtime Validation from Form to Meaning

Semantic Type Safety integrates knowledge graphs and ontologies directly into the validation process. Instead of checking a value against a fixed pattern or range, the system checks it against a model of the data's real-world meaning.



The New Paradigm

We move from asking 'Is this data well-formed?' to asking

"Does this data make sense according to our knowledge of the world?"

How It Works: Integrating a Knowledge Graph at Runtime

Ontologies & Knowledge Graphs

The foundation. A model defining concepts (`Person`, `Adult`), properties (`age`), and relationships. The system leverages this graph as its source of "truth".

An ontology defines `Adult` as a subclass of `Person` with the constraint `age ≥ 18`.

Semantic Constraints (SHACL/OWL)

The rules engine. Languages like SHACL (Shapes Constraint Language) define the conditions data must meet.

A SHACL shape can declare that any entity of type `Adult` *must* have an `ex:age` property with a value greater than or equal to 18.

Runtime Knowledge Queries

The validation step. The type system performs live lookups against the knowledge base.

When assigning a `UserAccountActive` type, the validator queries the graph: "Does an entity exist with this ID and an `isActive` status of `true`?"

Solving the Phantom Customer: Verifying Existence at the Type Level

The Old Way (Syntactic Check)

Type Definition

```
customer_id: Safe_Str__CustomerId
```

Validation Logic

Checks if the string matches a regex pattern (e.g., `^CUST-\d{4}$`).

Code Example

```
update_order("CUST-9999")
```

Outcome

✅ **Passes.**

The code proceeds, only to fail later when the database is queried, or worse, silently corrupts data.

The New Way (Semantic Check)

Type Definition

```
customer_id: ExistingCustomerId
```

Validation Logic

On assignment, the type's validator queries a knowledge graph or database: `ASK WHERE { :CUST-9999 rdf:type :Customer }`.

Code Example

```
update_order("CUST-9999")
```

Outcome

❌ **`SemanticTypeError`.**

Thrown immediately at the function call, because the ID is not found in the knowledge base. The error is caught at the boundary.

Enforcing Business Logic: From Manual `if` Statements to Semantic Types

The Old Way (Manual Check)

Type Definition

`age: Safe_UInt__Age`

Business Logic

Scattered throughout the codebase:

```
def grant_access(user: User):
    # This check is manually added
    if user.age < 18:
        raise PermissionError("User must be an adult.")
    # ... proceed
```

Problem

Inconsistent, easy to forget, not part of the function signature.

The New Way (Semantic Check)

Type Definition

The function signature itself enforces the rule:

```
# The type signature guarantees the user is an adult
def grant_access(user: AdultPerson):
    # ... proceed with confidence
```

Validation Logic

The `AdultPerson` semantic type is linked to an ontology rule (`Person.age >= 18`). When a `User` object is passed, the type system automatically validates this constraint.

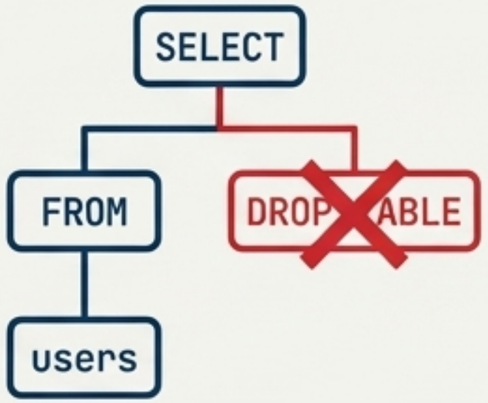
Benefit

The rule is declarative, centralised, and impossible to bypass. The code becomes self-documenting.

Beyond Primitives: Validating Complex Data Semantics

Semantic checks can go far beyond simple values, ensuring the integrity and safety of structured content.

Example 1: SQL Query Validation



Type: `Safe_Str__SelectQuery` (i `JetBrains Mono`)
Syntactic Check: Might ensure basic SQL syntax.
Semantic Check: Parses the query into an Abstract Syntax Tree (AST). Validates the AST against an allowed query model:

- Verifies that referenced tables and columns exist in the database schema.
- Disallows forbidden statements (e.g., `` `DROP TABLE` ``).
- Catches tautologies in `` `WHERE` `` clauses that could indicate an injection attempt.

Result: Prevents broken or malicious SQL from ever reaching the database engine.

Example 2: HTML Content Validation

Type: `Safe_Html__ProfileBio` (i `JetBrains Mono`)
Syntactic Check: Strips some known-bad tags.
Semantic Check: Uses a DOM parser and an HTML schema/ontology.

- Ensures the snippet is well-formed (tags are properly closed).
- Validates against a whitelist of allowed tags and attributes for that specific context (e.g., only `` `<b>` ``, `` `<i>` ``, `` `<p>` ``).
- Understands the destination context (e.g., rejecting block-level elements if it's meant for an inline span).

The Payoff: A Paradigm Shift in Data Quality and Security

Enhanced Data Quality

Catches logically inconsistent data at the source. Moves from "well-formed" to "meaningful and consistent" data, preventing corruption downstream.

Drastic Security Improvements

Moves beyond simple regex to understand the *intent* of content. A semantic validator can act as an intelligent whitelist, detecting SQL injection or XSS that is syntactically valid but semantically malicious.

Discovering "Unknown Unknowns"

Validates relationships between fields that isolated checks would miss (e.g., ensuring a city is in the correct country by querying a geographic knowledge base).

Truly Self-Documenting Code

Business rules like "must be an existing user" or "must be an adult" become part of the type signature (`ExistingUser`, `AdultPerson`), making intent clear and enforcement automatic.

Pragmatic Considerations for Implementation

Adopting semantic type safety is powerful, but requires thoughtful engineering.

Challenges

Performance Overhead

Semantic checks (e.g., a database lookup) are more expensive than an `isinstance` check.

Knowledge Availability

The validation is only as good as the underlying knowledge graph.

Complexity & Tooling

Defining semantic types is more involved than a simple regex.

Error Handling

Failures are more complex (e.g., 'Unknown Customer ID' vs. 'Expected int').

Mitigations & Considerations

Mitigation

Use aggressive caching, apply checks only at system boundaries (e.g., API inputs), or use lazy validation.

Consideration

Requires building and maintaining an ontology, or ensuring validators have low-latency access to up-to-date production data.

Path Forward

Requires frameworks to create a developer-friendly API for registering ontology-driven validators.

Best Practice

Design clear semantic error messages while being careful not to leak internal system state.

Building on a Foundation of Related Concepts

While a coherent runtime framework is novel, the principle of embedding more meaning into types exists across different domains.

Academia & Functional Languages

Refinement Types & Dependent Types (e.g., Liquid Haskell, Dafny).

```
type AgeOver18 = int
  where x >= 18
```

These allow types to be refined with logical predicates. Our approach is the runtime, data-driven sibling to these compile-time verification concepts.

The Semantic Web Community

SHACL & OWL Constraints

These are used to ensure data integrity within triple stores. We are essentially bringing SHACL-like validation into the application's in-memory object model, applying it continuously at runtime.

Data Science & Industry

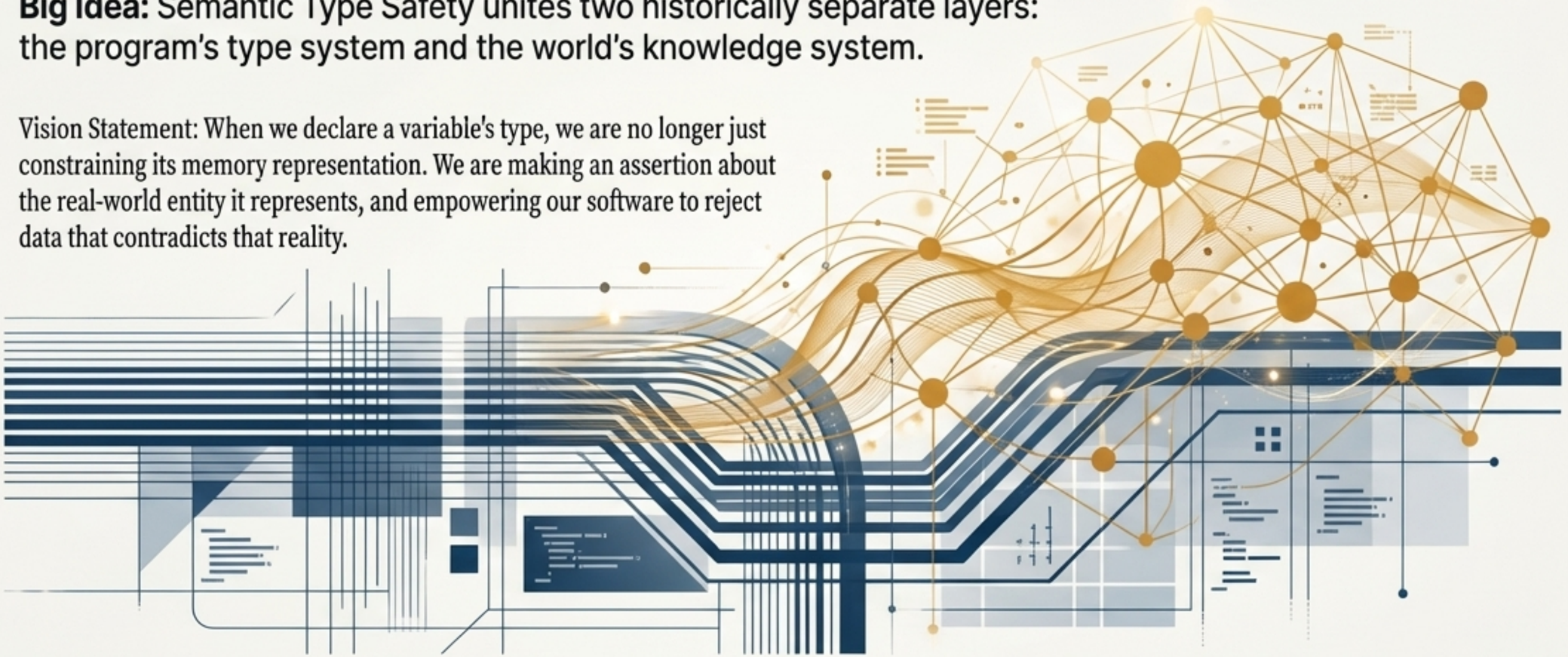
Ontology-Driven Validation & Semantic Data Platforms

Fields like healthcare and finance already use ontologies to flag inconsistent data (e.g., a pregnant male patient). We are making this a first-class citizen of the programming language's type system.

The Next Evolution: From Type Systems to Knowledge Systems

Big Idea: Semantic Type Safety unites two historically separate layers: the program's type system and the world's knowledge system.

Vision Statement: When we declare a variable's type, we are no longer just constraining its memory representation. We are making an assertion about the real-world entity it represents, and empowering our software to reject data that contradicts that reality.



Future Implications:

- Truly context-aware systems.
- More resilient AI and data-centric applications.
- A foundational piece for building systems that are not just correct, but truthful.

Code shouldn’t just be well-typed.

```
process_order(order: ExistingOpenOrder)
```

It should reflect the truth.