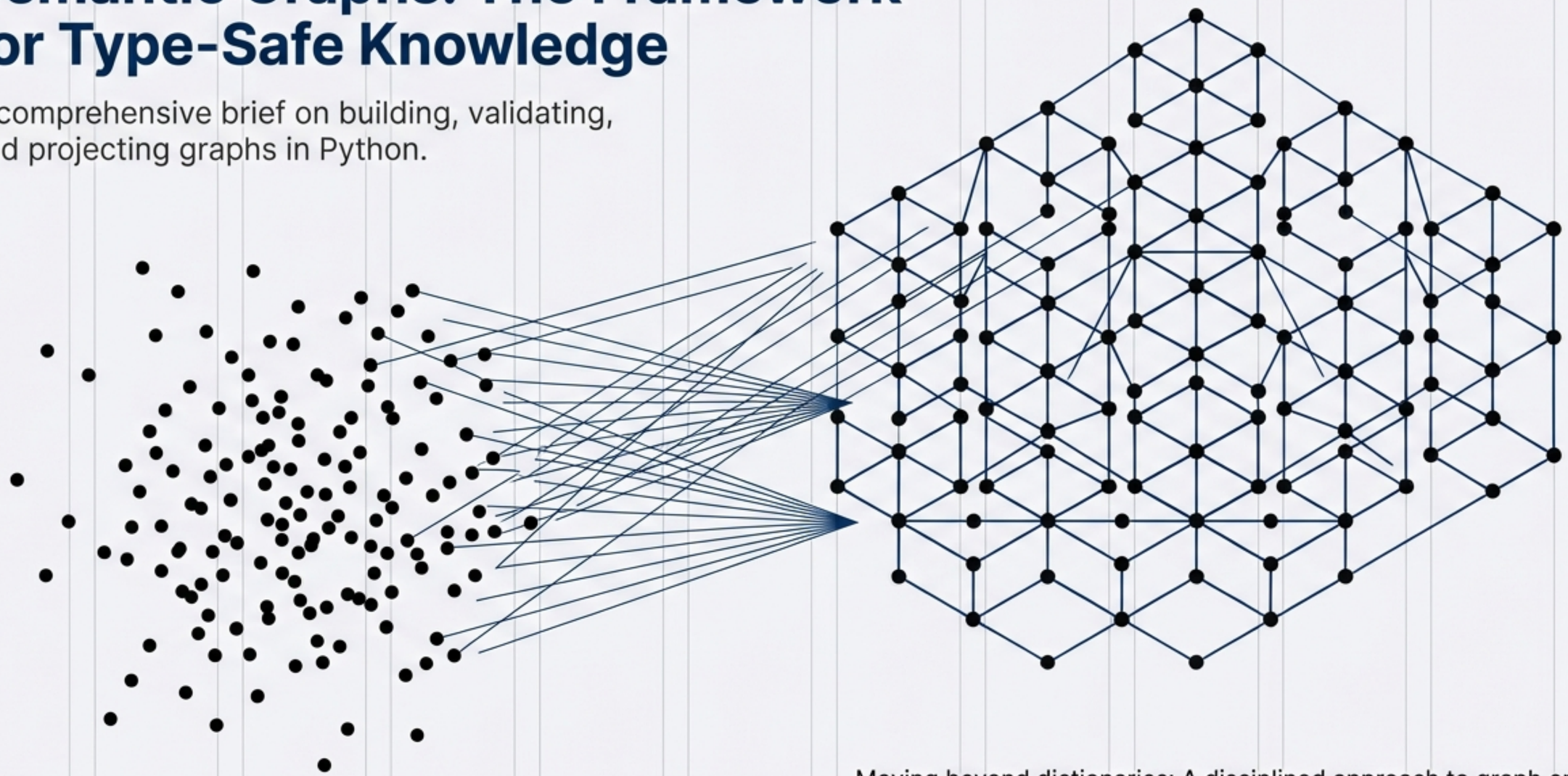
Semantic Graphs: The Framework for Type-Safe Knowledge

A comprehensive brief on building, validating, and projecting graphs in Python.



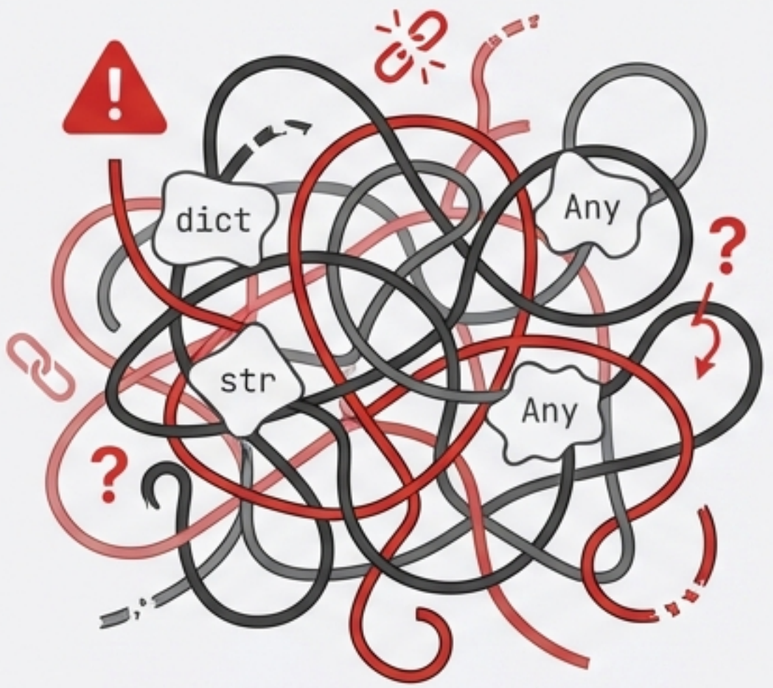
Moving beyond dictionaries: A disciplined approach to graph engineering.

Core Promise: Runtime type safety, ontology-driven validation, and deterministic reproducibility.

`osbot_utils.helpers.semantic_graphs | v3.70.x+`

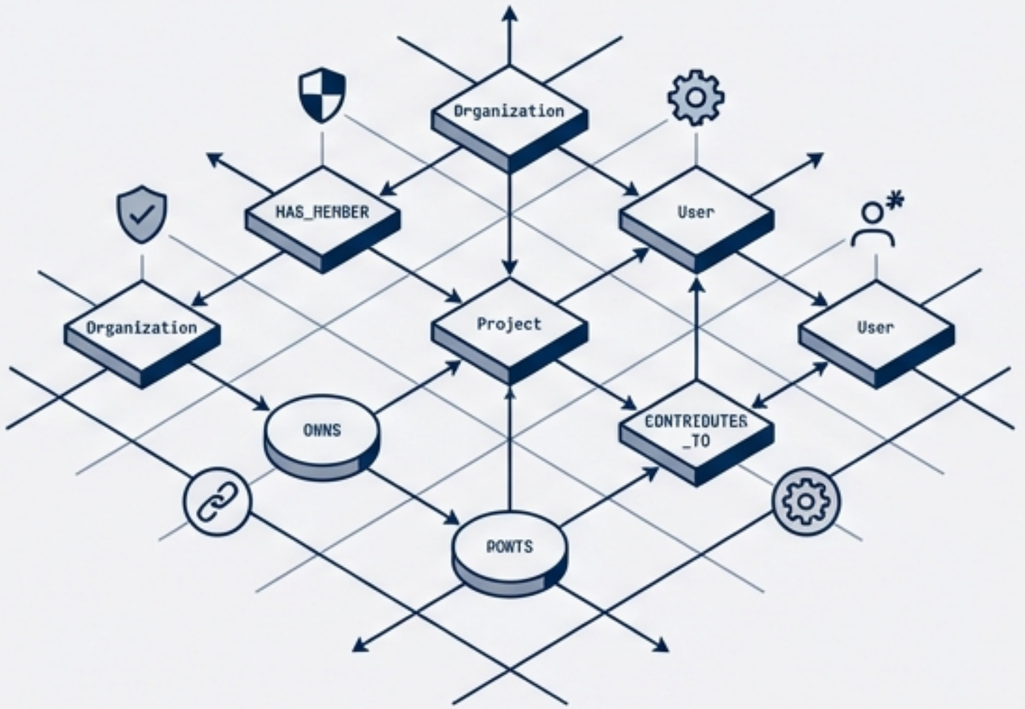
The Problem with Ad-Hoc Graphs

⚠ The Spaghetti (Status Quo)



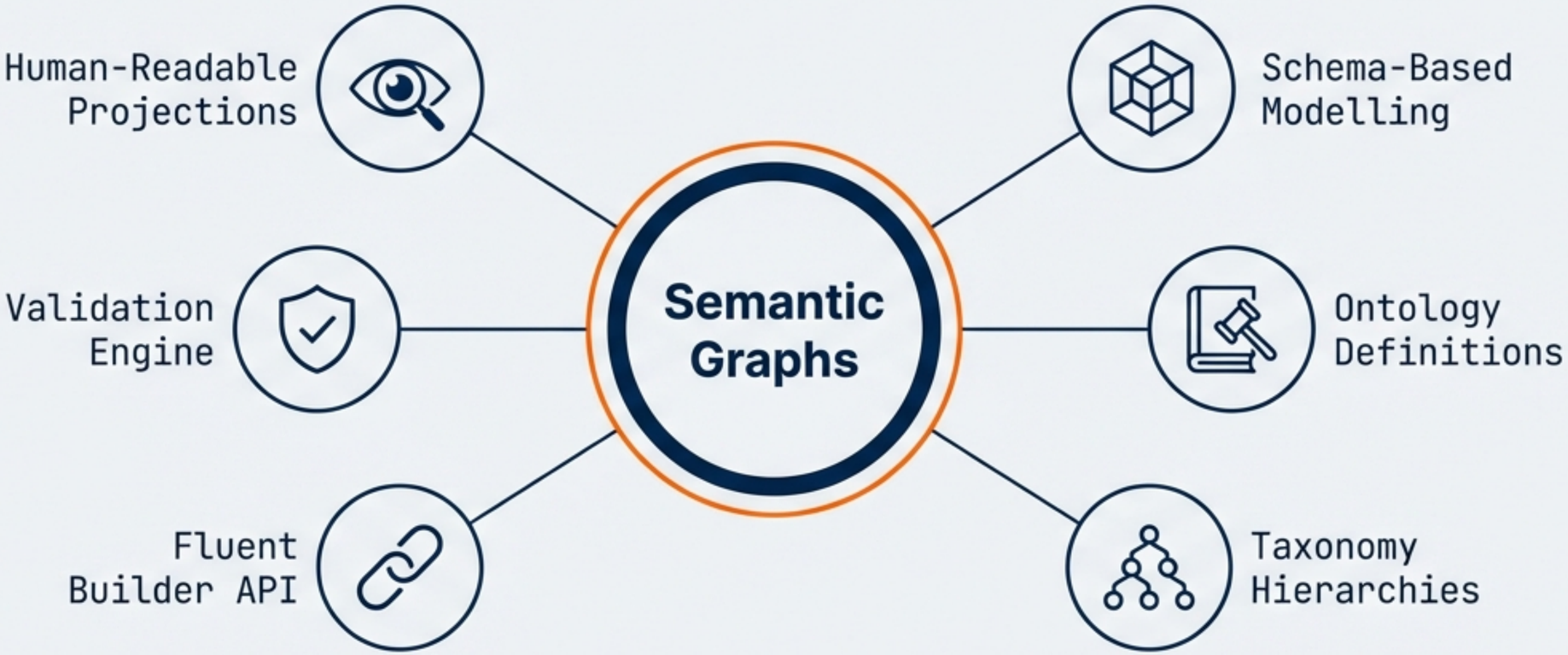
- **No Edge Rules:** Invalid relationships go unchecked.
- **Manual Integrity:** Broken links and data loss.
- **Fragile Validation:** Custom code is often incomplete.

The Blueprint (Semantic Graphs)



- **Enforced Constraints:** Logic dictates structure.
- **Type Safety:** Machine vs. Human IDs distinguished.
- **Rigorous Engineering:** Constraints ensure sanity.

A Complete Ecosystem for Graph Engineering

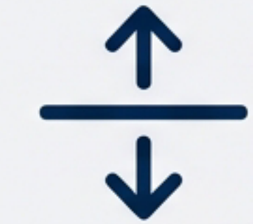


Semantic Graphs is not just a data structure. It is a type-safe framework for building, validating, and projecting knowledge graphs, ensuring that nothing enters your system without adhering to the laws you define.

Design Philosophy & Core Principles

Schemas are Pure Data

Schema classes contain only type annotations. They never contain methods.



Separation of Concerns

All business logic lives in Utils, Builders, and Validators. Logic is distinct from data.

Deterministic Reproducibility

IDs are generated from seeds for testing. Run it 100 times, get the same IDs 100 times.

Ontology-Driven

The Map (Ontology) dictates the Territory (Graph). If the map says it is invalid, the graph rejects it.

The Dual Identity System: ID vs. Ref

Separation of Internal Reliability and External Readability.

ID (The Machine)

Internal Precision

`Node_Type_Id('abc123...')`

Unique identifiers, foreign keys, linking nodes, maintaining integrity.

Projector

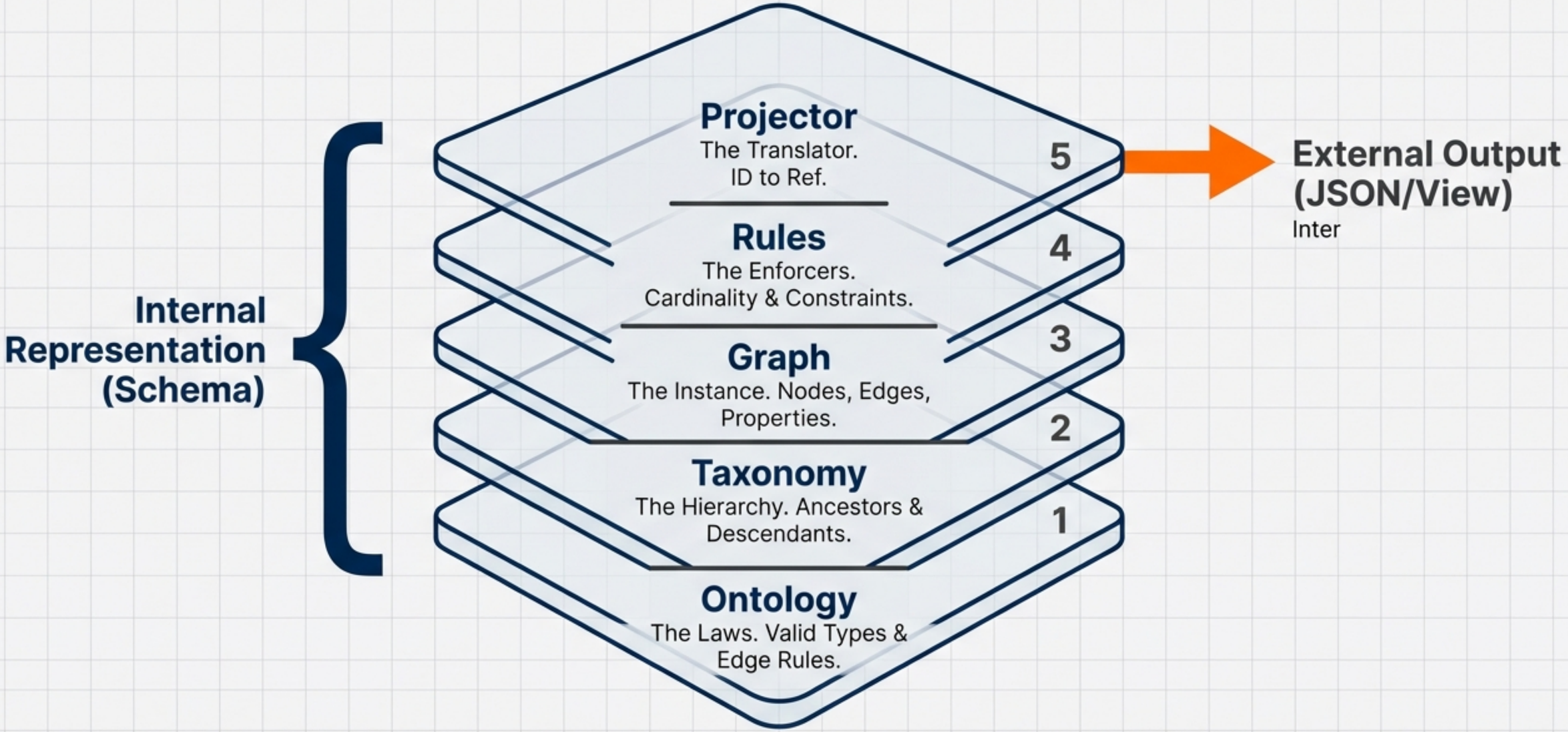
Ref (The Human)

External Clarity

`Node_Type_Ref('class')`

Human-readable labels, configuration, debugging, UI display.

System Architecture: The Five Subsystems



Step 1: Defining the Ontology

Configuring the Valid Universe.

```
# Using Ontology__Utils to define the laws
node_type = create_node_type(ref='function',
    category_id=cat_id, seed='func')
predicate = create_predicate(ref='calls',
    inverse_ref='called_by', seed='call')

# Defining the Edge Rule (The Constraint)
create_edge_rule(source_id=function_id,
    pred_id=call_id, target_id=function_id)
```

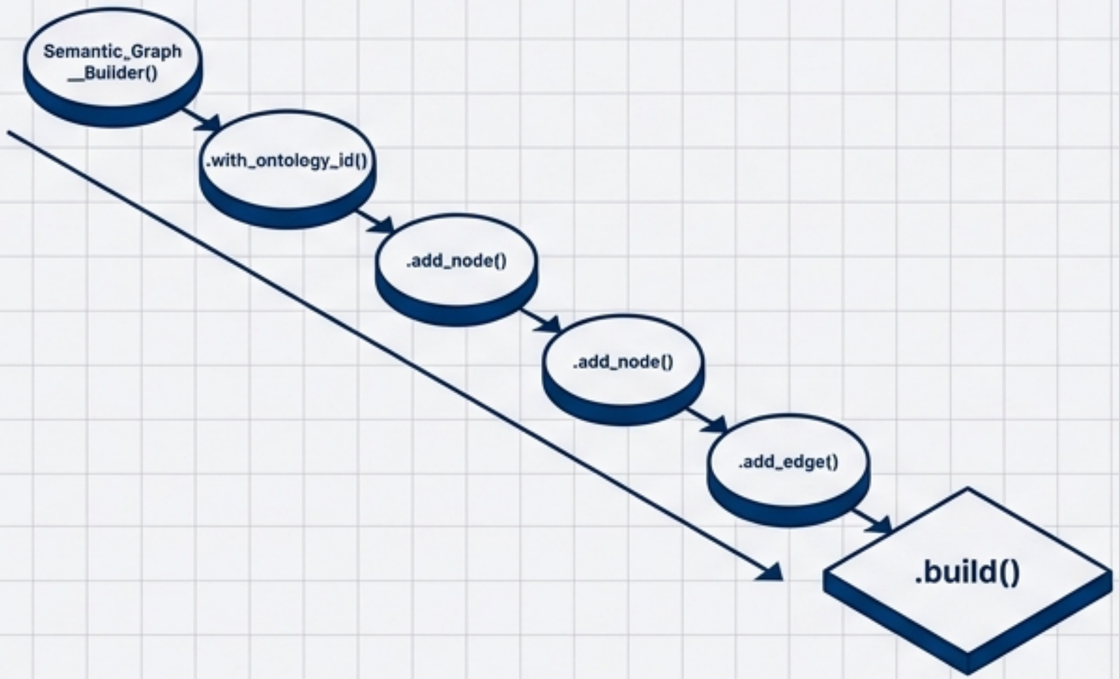
Key Concepts

- **Node Types**: What entities can exist?
- **Predicates**: Relationships and their inverse pairs.
- **Edge Rules**: Legal connections. (e.g., A “Function” can call a “Function”, but not a “Variable”).

Step 2: Building the Graph

The Fluent Builder Pattern

Chaining




```
graph = (Semantic_Graph__Builder()
    .with_ontology_id(my_ontology_id)
    .add_node(type_id=func_id,
        name='main')
    .add_node(type_id=func_id,
        name='helper')
    .add_edge(from_id=main_id,
        pred_id=calls_id,
        to_id=helper_id)
    .build())
```

Binds Laws

Constructs Data

Returns Schema__Semantic_Graph

Advanced Building: Determinism & Properties

Deterministic IDs

Randomness is the enemy of debugging.
Use seeds to guarantee reproducible tests.


```
add_node_with_seed(..., seed='my-seed')
```

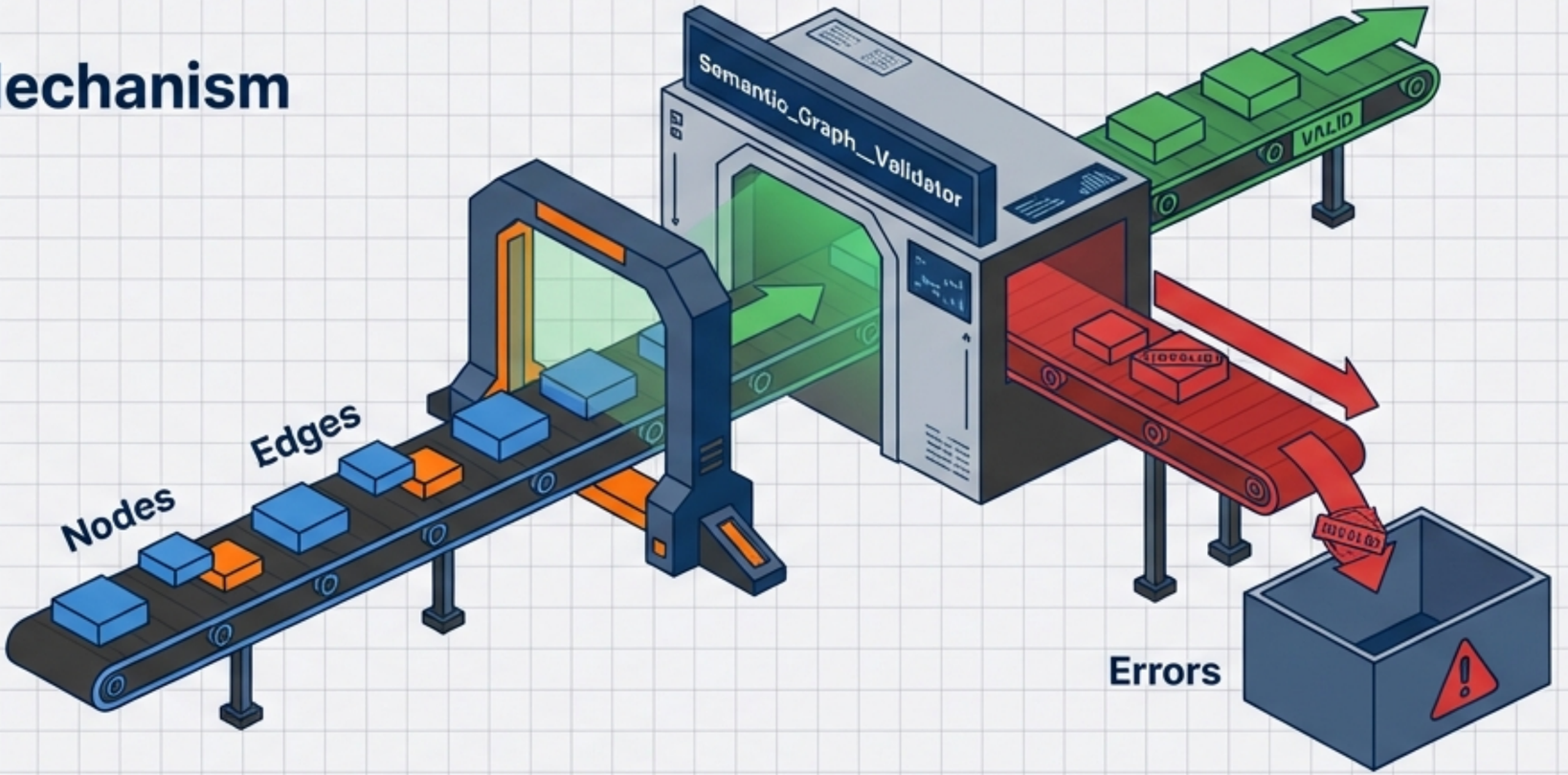
The Property System

Type-safe key-value pairs attached strictly to Nodes or Edges.


```
add_node_property(node_id, prop_id, value)
```

Step 3: The Validation Engine

The Gatekeeper Mechanism

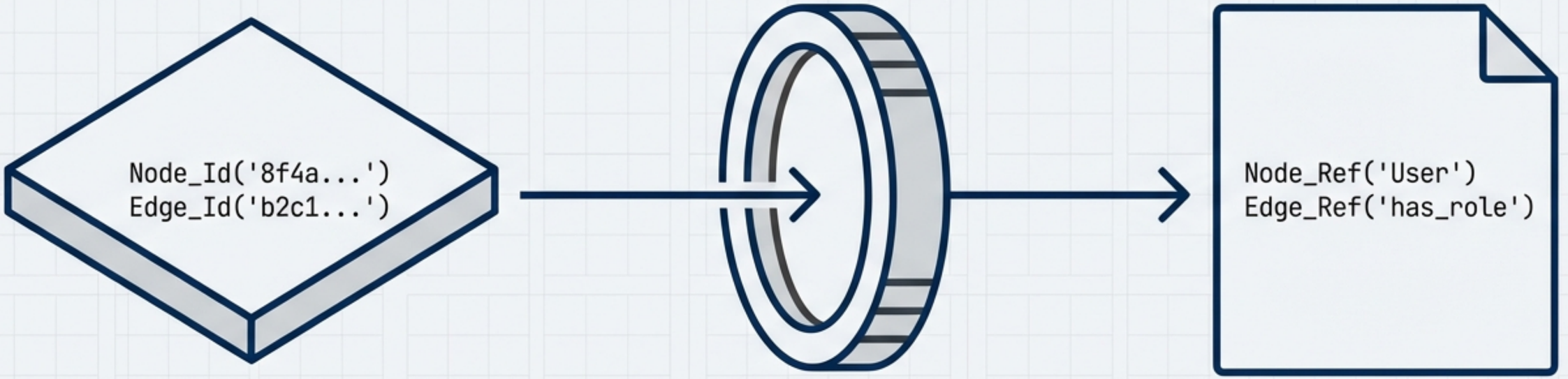



```
result = Semantic_Graph__Validator().validate(graph)
if not result.valid:
    print(result.errors)
    # Output: "Invalid edge: Function cannot call Package"
```

- **What is Validated?**
  - ☑ Unknown Nodes? (Must exist in Ontology)
  - Illegal Edges? (Must match Edge Rules)
  - Property Mismatch? (Must match Defined Types)

Step 4: Projections & Readability

Transforming Schemas into Insights.



Projections resolve internal IDs using the Registry, enabling debugging, logging, exporting, and UI integration.

Engineering Best Practices

DO	DON'T
✓ **DO:** Use Deterministic IDs for tests and reproducible builds.	✗ **DON'T:** Put logic inside Schema classes.
✓ **DO:** Keep Schemas Pure Data (no methods, only types).	✗ **DON'T:** Mix IDs and Refs in the same field.
✓ **DO:** Use the Registry Pattern to resolve References.	✗ **DON'T:** Use raw strings where Safe_Str or typed IDs are expected.
✓ **DO:** Validate the graph before processing logic.	✗ **DON'T:** Ignore Edge Rules (validation will fail).

Troubleshooting Common Friction Points

Diagnosing and Resolving Key Integration Errors.

`ERROR: Unknown node_type_id`

Cause: Node Type not registered.
Fix: Check Ontology__Registry.

`ERROR: Invalid edge`

Cause: No Edge_Rule exists.
Fix: Add rule via Ontology__Utils.create_edge_rule.

`ERROR: Registry required to resolve ref`

Cause: Used *_by_ref without registry.
Fix: Call **.with_registry(...)** on builder.

`ERROR: Empty Projection`

Cause: Ontology not found.
Fix: Ensure Ontology is registered before projection.

Essential API Reference

1. Building

`Semantic_Graph__Builder` (Fluent API)

`Semantic_Graph__Utils` (Query nodes/edges)

2. Ontology & Taxonomy

`Ontology__Utils / Ontology__Registry`

`Taxonomy__Utils / Taxonomy__Registry`

3. Validation & Rules

`Semantic_Graph__Validator`

`Rule_Engine`

4. Projection

`Semantic_Graph__Projector`

All classes available under osbot_utils.helpers.semantic_graphs

Ready to Build

The Semantic Graph Checklist

- [ ] **Schema Design:** Ensure pure data schemas; use ID types for internals.
- [ ] **Ontology Setup:** Create Node Types, Predicates, and Edge Rules.
- [ ] **Graph Construction:** Use `Semantic_Graph__Builder` with deterministic seeds.
- [ ] **Validation:** Run `Semantic_Graph__Validator` and check `.valid`.
- [ ] **Projection:** Use the Projector for human-readable output.

Semantic Graphs: Bringing rigour, type safety, and sanity to your data structures.