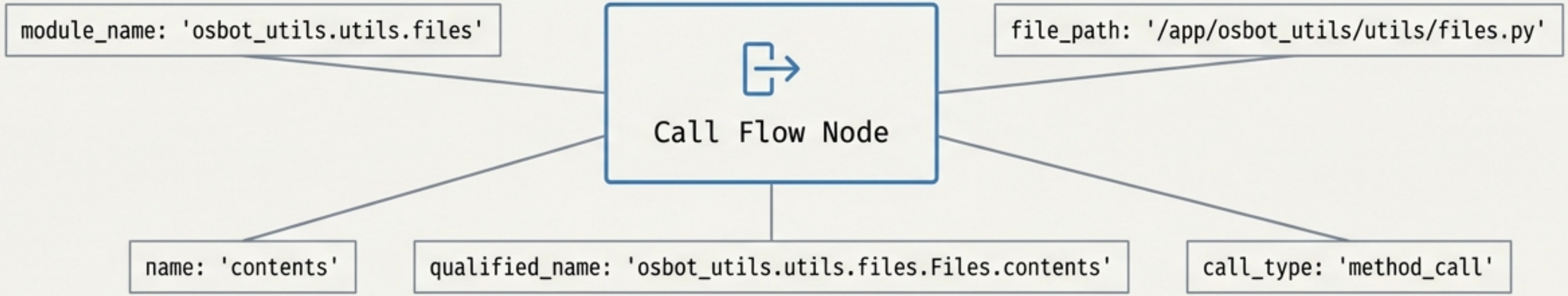
Refactoring Our Codebase: From Ad-Hoc Attributes to a Semantic Framework

A Technical Proposal for Implementing a Generic Semantic Graphs Framework and a First-Class Code Structure Model

Our Current Call Flow Graphs Mix Behaviour with Structure

The existing Call Flow Analyzer embeds structural metadata (module, file path, qualified name) as flat string attributes directly onto every behavioural node.



Key Issues

- **Redundancy:** The same module name is repeated on every node within that module.
- **Mixed Concerns:** Behavioural data (the call) and structural data (the module) are conflated in one object.
- **Implicit Relationships:** The 'module contains class' relationship is not explicitly modelled and cannot be queried.
- **No Hierarchy:** The critical Package → Module → Class → Method hierarchy is completely lost.

This Approach Creates Fragility and Limits Reusability

The current design is not just inefficient; it actively hinders our ability to analyse, query, and reuse our understanding of the codebase.

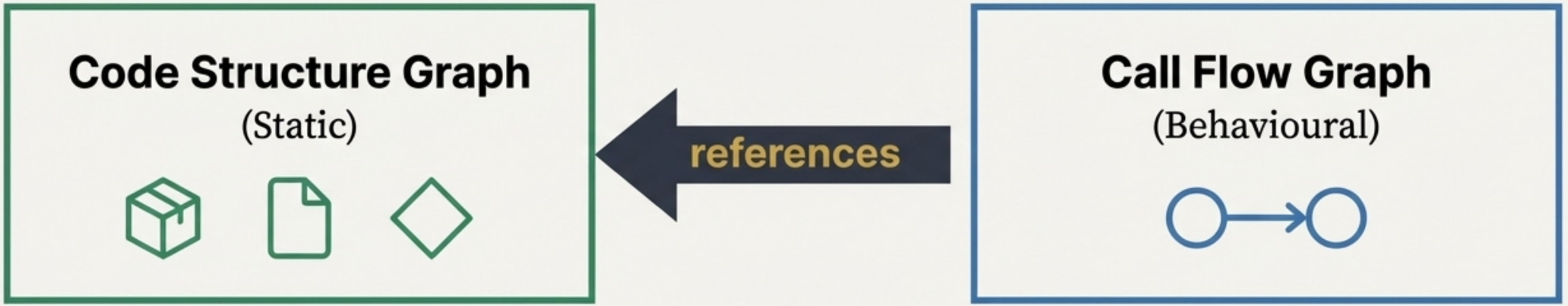
Technical Debt

- Fragile Queries: Finding ‘all classes in package X’ requires slow, error-prone string parsing instead of graph traversal.
- High-Risk Changes: Modifying how we represent code structure requires changing the schema of every call flow node.
- Inconsistent Data: No validation exists to ensure qualified names or module paths are internally consistent.

Strategic Limitation

- Knowledge Silo: The understanding of code structure is locked exclusively within the call flow tool.
- No Reusability: This structural model cannot be used for other purposes, such as dependency analysis or architectural validation.

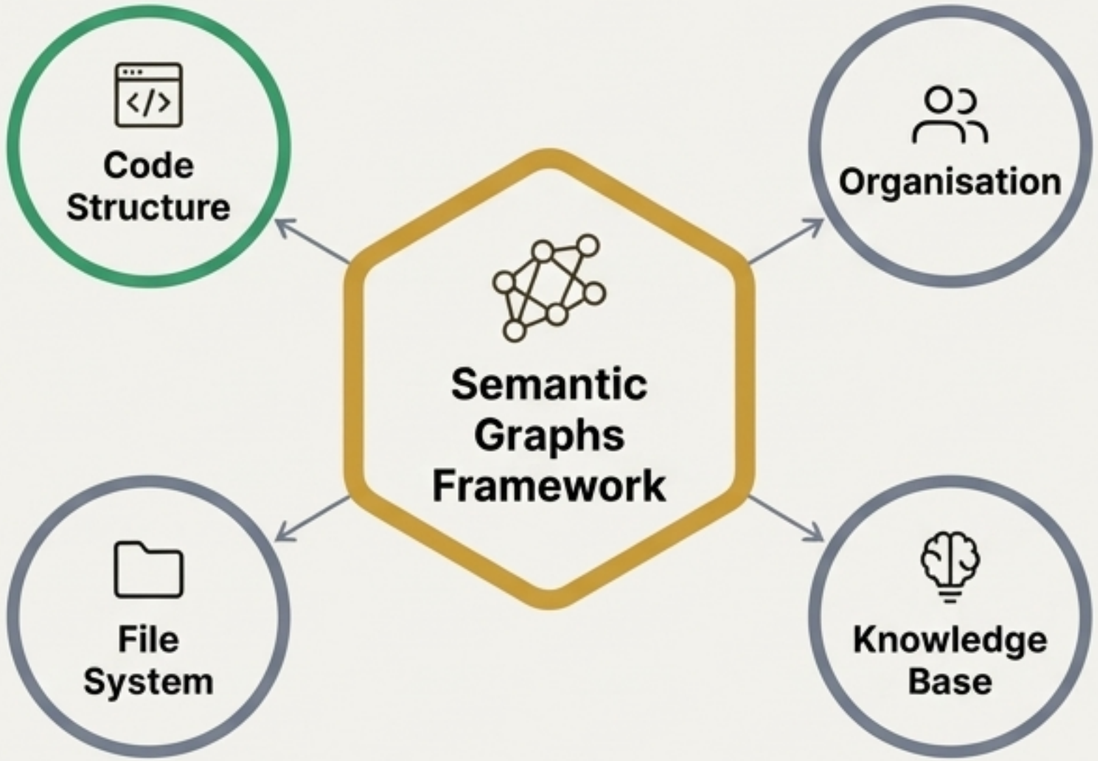
The Insight: Separate *What Code Is* from *What It Does*



- **Structure Graph**: A first-class, queryable graph representing Python's static hierarchy: packages, modules, classes, and methods.
- **Call Flow Graph:** A lean graph of behavioural nodes that simply *point to* their corresponding nodes in the structure graph.
- This separation of concerns provides clarity, eliminates redundancy, and enables reusability.

This Pattern Is a Reusable Asset: A Generic Semantic Graphs Framework

The need for typed, validated graphs with formal relationships is a recurring pattern. We can solve this problem once, robustly.



Key Capabilities of the Framework

- Formal definition of node and edge types (`Ontology` in Fira Code).
- Validation to ensure graph integrity.
- Support for bidirectional navigation ("class has method" AND "method in class").
- A system for domain-specific constraints (`Rules` in Fira Code).

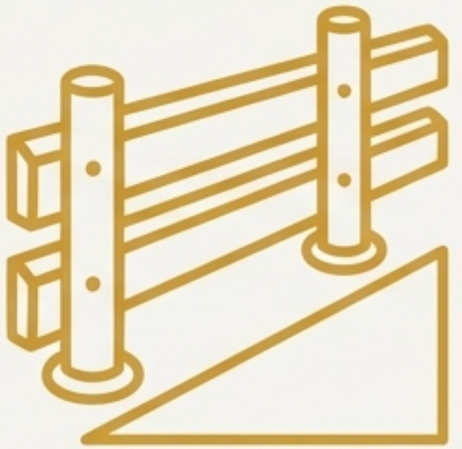
The Framework Provides Three Pillars for Building Rich Models

The Blueprint (Ontology)

Formally defines the ‘what’ and the ‘how’. It specifies the allowed node types (e.g., `Class`, `Method`) and the valid relationships between them (e.g., a `Class` *defines* a `Method`).

The Library (Taxonomy)

Provides a classification system. It allows us to group nodes into hierarchical categories (e.g., a `Class` and a `Function` are both ‘Code Elements’).



The Guardrails (Rules)

Enforces domain-specific constraints on the graph instance. Examples include transitivity (if A contains B, and B contains C, then A contains C) or cardinality (a method can only be defined in one class).

The Framework Is Built on a Foundation of Precision and Clarity

1. Bidirectional Relationships with Explicit Names

Every edge has both a forward (`defines`) and an inverse (`defined_in`) name. This eliminates ambiguity and simplifies graph traversal.

2. Relationships as Verbs

Edge types are always verbs that describe the relationship from the source node's perspective, ensuring semantic consistency.

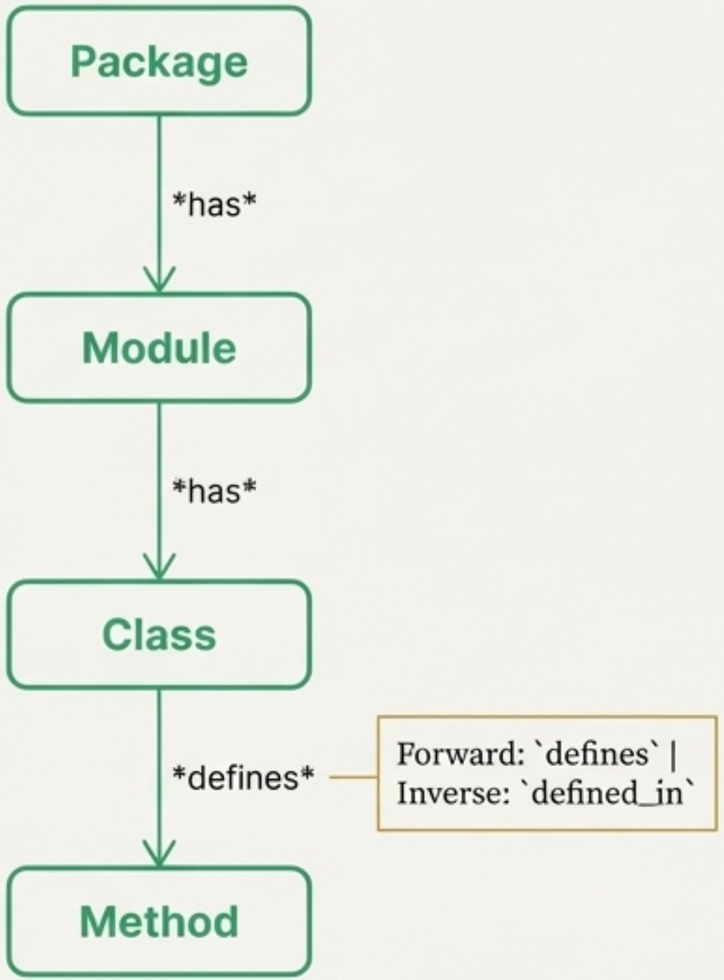
3. Separation of Structure and Rules

The `Ontology` defines what *can* exist. The `Rules` define constraints on what *does* exist in a specific graph instance. This separation enhances flexibility.

4. Domain-Specific Primitives

We use precise, type-safe primitives like `Safe_Str__Python__Class` instead of generic strings to enforce correctness at the data level.

Application: A Formal Graph for Python Code Structure

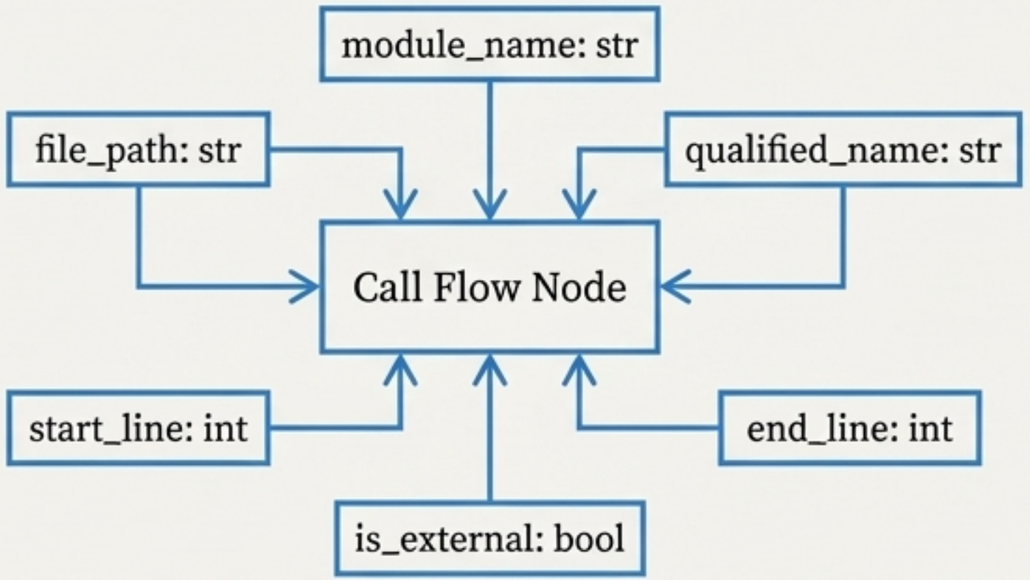


From `ontology__code_structure.json`

```
{
  "node_types": {
    "Class": {
      "description": "Represents a Python class.",
      "relationships": [
        {
          "verb": "defines",
          "target_node_type": "Method",
          "inverse_verb": "defined_in",
          "cardinality": "one-to-many"
        }
      ]
    },
    "Method": {
      "description": "Represents a method within a class."
    }
  }
}
```

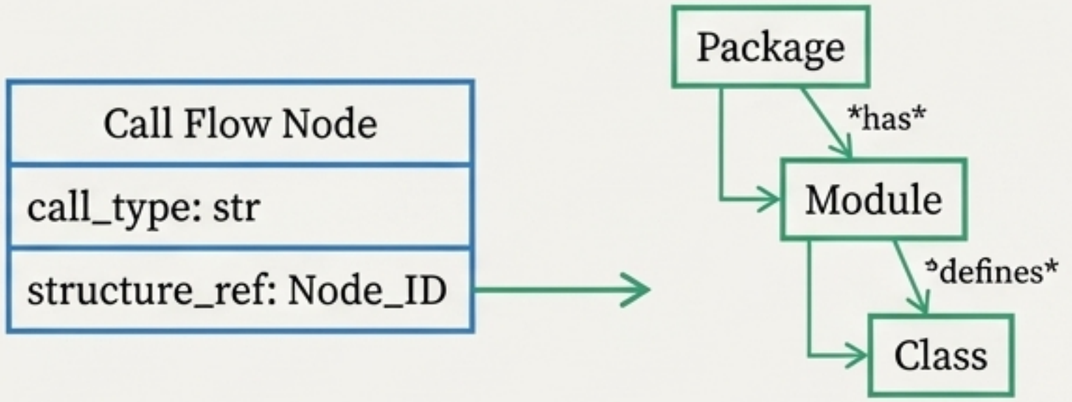
Call Flow Nodes Become Lightweight Pointers to a Rich Structural Model

`Schema__Call_Flow__Node (Old)`



Redundant & Brittle

`Schema__Call_Flow__Node (New)`



Code Structure Graph
(from slide 2)

Lean & Robust

Our Implementation Is a Phased Rollout from Foundation to Integration

Phase 1: Build the Foundation

Create the domain-agnostic framework for typed, validated graphs.

Deliverable: The generic Semantic Graphs framework.

→

Phase 2: Define the Domain

Implement the specific data types required to model Python code.

Deliverable: Python-specific primitive types.

→

Phase 3: Implement the Model

Build the analyser that consumes Python code and produces a structure graph.

Deliverable: The Code Structure analyser and graph builder.

→

Phase 4: Integrate and Refactor

Update the existing Call Flow tool to leverage the new structure graph.

Deliverable: Update the Call Flow Analyser to use the new framework.

Phase 1 Establishes the Core Semantic Graphs Framework

This phase delivers all the reusable components required to build any semantic graph, independent of a specific domain.

Core Primitives

- `Ontology_Id`
- `Taxonomy_Id`
- `Node_Type_Id`
- `Safe_Str__Ontology__Verb`

Schema Definitions

Ontology Schemas
`Schema__Ontology`
`Schema__Ontology__Node_Type`

Taxonomy Schemas
`Schema__Taxonomy`

Rules Schemas
`Schema__Rule_Set`

Graph Instance Schemas
`Schema__Semantic_Graph`

Framework Utilities

`Ontology__Registry`
(loads from JSON)

`Semantic_Graph__Validator`
`Semantic_Graph__Builder`

Success Is Defined by Clear, Testable Criteria

Minimum Viable Product (MVP)

Goal: The core refactoring is complete and functional.

- ✓ Semantic Graphs framework is functional.
- ✓ Code structure ontology is defined in JSON.
- ✓ Code structure analyser produces a valid semantic graph.
- ✓ Call flow nodes successfully reference structure nodes.
- ✓ All existing call flow tests pass after refactoring.

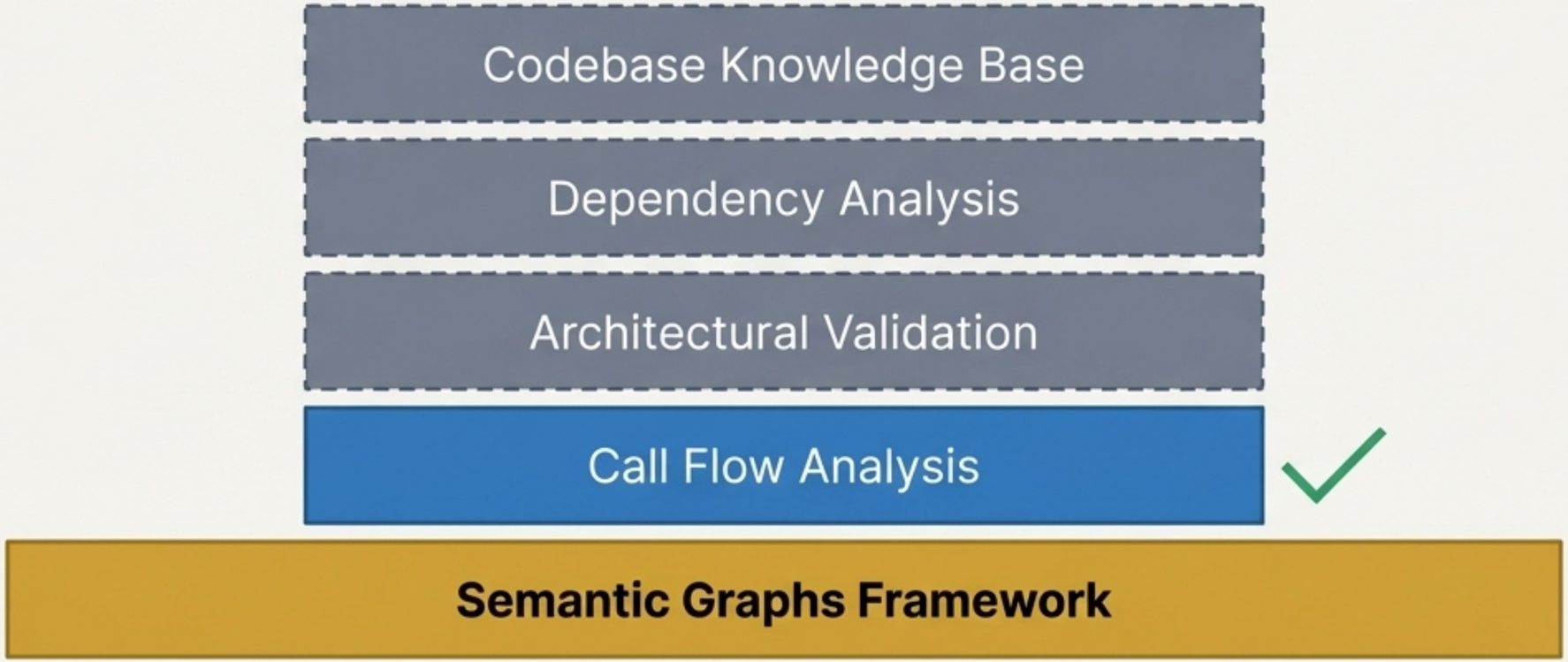
Full Implementation

Goal: The framework is robust, documented, and ready for new applications.

- ✓ Ontology validation and rule engine are fully operational.
- ✓ Taxonomy classification is working.
- ✓ Registries correctly cache definitions.
- ✓ Complete documentation is published.
- ✓ The framework is ready for other domain ontologies.

This Is a Foundational Asset for Future Code Intelligence

By making this strategic investment, we move from a single-purpose tool to a reusable platform for understanding and analysing our software.



This refactoring provides a robust, queryable, and reusable model of our code—an essential asset for any future development or analysis tools.