Revised Technical Design Specification: Minotaur Enrichment and Interface Changes

2.1. Introduction

This document specifies the required architectural changes to the Minotaur code analysis platform. These modifications will enable Minotaur to generate the semantically enriched CognitiveGraph and expose it to external reasoning engines like Golem. The design prioritizes a high-throughput, low-latency data exchange mechanism for co-located processes by leveraging a zero-copy serialization strategy, while retaining a conventional API for auxiliary access and debugging.

2.2. Component Architecture: The Enrichment Pipeline

(This section remains unchanged from the previous specification.)
Minotaur's existing graph construction process will be augmented with a new, final stage: the **Semantic Enrichment Pipeline**. This pipeline will execute after the base Code Property Graph has been fully constructed and linked.

2.2.1. Plugin-Based Architecture

The Enrichment Pipeline will be designed around a **plugin architecture**. This is critical for maintainability and extensibility, as it allows framework-specific analysis logic to be developed and deployed independently of Minotaur's core.

- **Plugin Manager:** A central component within Minotaur responsible for discovering, loading, and executing registered enrichment plugins at runtime.
- **Plugin Interface:** A well-defined interface that all plugins must implement. This interface will provide methods that are called by the pipeline, passing a mutable reference to the CognitiveGraph.

```
interface IEnrichmentPlugin {
  // Returns the name of the framework this plugin targets (e.g., "spring-boot").
  string GetTargetFramework();

  // Executes the enrichment logic on the provided graph.
  void Enrich(CognitiveGraph graph);
}
```

- **Execution Flow:** When analyzing a project, Minotaur will first detect the frameworks in use. It will then invoke the corresponding registered plugins, which will traverse the graph to identify patterns, create the new semantic nodes (DataModel, ArchitecturalPattern), and add the new semantic edges (PART_OF_PATTERN, USES_DATA_MODEL).

2.3. Interface Specification: Golem Access

Minotaur will expose the enriched CognitiveGraph to Golem through a tiered interface model designed to optimize for different deployment scenarios.

2.3.1. Primary Data Exchange Interface (Zero-Copy IPC)

This interface is the primary mechanism for Golem to access the complete, project-wide CognitiveGraph. It is optimized for the scenario where Minotaur and Golem are running as separate processes on the same host, enabling data exchange with minimal overhead.

- **Technology: Cap'n Proto** serialization over a **shared memory segment**.[1]
- **Mechanism:** This approach avoids the traditional serialize-transmit-parse-deserialize cycle. The in-memory representation of the data structure is identical to its on-wire format, allowing Golem to access the graph data instantly by mapping the shared memory region.[2] This is a true zero-copy operation from the perspective of the applications.
- **Workflow:**
  1. After the Enrichment Pipeline completes, Minotaur allocates a shared memory segment of sufficient size.
  2. Minotaur builds the Cap'n Proto message representing the entire CognitiveGraph directly within this shared memory segment.
  3. Minotaur signals to the Golem process that the graph is ready, passing an identifier for the shared memory segment (e.g., via a local socket or another IPC mechanism).
  4. Golem maps the shared memory segment into its own address space, gaining immediate, read-only access to the complete graph data structure without any parsing or data copying.[3] The CognitiveGraph effectively becomes a shared, in-memory database.
- **Handling Graph Structures:** A key consideration is that Cap'n Proto's native encoding is a tree, not a graph, and does not support cycles or multiple references to the same object.[5] To accommodate the graph nature of the CognitiveGraph, one of the following strategies must be employed within the Cap'n Proto schema:
  - **ID-Based Referencing:** Nodes that are targets of back-edges or multiple forward

    edges are referenced by their unique ID. The Golem engine will be responsible for resolving these IDs to pointers upon initial traversal.
  - **Tree-Like Unfolding:** The graph is traversed and serialized in a way that avoids cycles, duplicating nodes where necessary. This is less efficient in terms of space but can simplify the access logic.
  - The ID-based referencing approach is strongly recommended as it preserves the true graph structure most efficiently.

2.3.2. Auxiliary Access API (GraphQL)

While the zero-copy IPC interface serves the Golem engine's need for bulk data access, a secondary API remains valuable for debugging, human-driven exploration, and integration with other tools that do not require the entire graph.

- **Endpoint:** POST /v1/graphql
- **Technology: GraphQL**. This provides a flexible, strongly-typed query language ideal for exploring graph structures without the overhead of fetching the entire dataset.
- **Use Cases:**
  - **Developer Tooling:** Allowing developers to write queries to inspect specific parts of a project's CognitiveGraph.
  - **Lightweight Clients:** Enabling other services to query for small, specific pieces of information without the need to map a potentially massive shared memory segment.
  - **API Introspection:** GraphQL's schema introspection capabilities make the API self-documenting.
- **Implementation Notes:** This endpoint would run alongside the primary IPC mechanism. It would query the same in-memory CognitiveGraph representation that Minotaur prepares for Golem, ensuring data consistency across both interfaces. All connections must implement cursor-based pagination to handle potentially large query results efficiently.