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 (as defined above) and expose it to external reasoning engines like Golem through a well-defined set of interfaces. The core of this change is the introduction of a modular, plugin-based enrichment pipeline and a robust API for graph data access.

2.2. Component Architecture: The Enrichment Pipeline

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**.[10] 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 (e.g., by identifying specific dependencies like spring-boot-starter-web). 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).[11]

2.3. Interface Specification: Golem Access API

Minotaur will expose the enriched CognitiveGraph to Golem via a dedicated API. A hybrid approach combining a bulk data export with a fine-grained query endpoint is recommended to support the different needs of the Golem engine.

2.3.1. Bulk Data Export Endpoint

This endpoint is designed for the initial loading of the entire project graph into Golem's Neural Perceiver.

- **Endpoint:** GET /v1/project/{projectId}/graph
- **Response Format:** The complete CognitiveGraph, serialized using **Protocol Buffers**.
- **Streaming:** For exceptionally large projects, the API should support streaming the response to avoid excessive memory consumption on the server.[12] The response would be a stream of Protobuf messages, each representing a chunk of nodes and edges.

2.3.2. Graph Query Endpoint

This endpoint is designed to be used by Golem's Symbolic Reasoner to perform targeted traversals and semantic queries. The recommended technology for this interface is **GraphQL**, as it is explicitly designed for querying graph-structured data and allows the client to specify its exact data needs, preventing over-fetching.[15]

- **Endpoint:** POST /v1/graphql
- **GraphQL Schema:** Minotaur will expose a GraphQL schema that mirrors the CognitiveGraph schema. This allows for powerful, typed queries.
- **Example Query:** A query from the Symbolic Reasoner to find all controller methods that use the "User" data model might look like this:

```
GraphQL
query GetUserControllers {
  architecturalPattern(patternType: "Controller") {
    name
    implementedBy {
     ... on Method {
       name
```

```
      usesDataModel(name: "User") {
        name
      }
    }
  }
 }
}
```

- **Pagination:** All connections (list-based fields) in the GraphQL schema must implement cursor-based pagination to handle large result sets efficiently. This aligns with best practices for scalable graph APIs.[17] The
  pageInfo object, containing cursors and hasNextPage flags, will be a standard part of any paginated response.