* Created by Unknown User (kiran.sahoo2@spglobal.com), last modified on [May 01, 2023](https://confluence.marketintelligence.spglobal.com/pages/diffpagesbyversion.action?pageId=143761787&selectedPageVersions=4&selectedPageVersions=5)
* [GraphQL/APIs Threat Protection And Monetization](https://confluence.marketintelligence.spglobal.com/pages/viewpage.action?pageId=143761787)

**Design Objectives**

* Purpose is to provide a high performant Implementation for securing GraphQL/API endpoints from Malicious queries/requests and enable Monetization Capabilities .
  + **Threat Protection**
    - Cost Analysis
    - Depth Limiting
    - Rate Limiting
    - Authentication/Authorization
    - Tracing
  + **Monetization**
    - Build Monetization schemes to open up Revenue Streams

**Rate and Depth Limiting**

* **Authentication/Authorization**
  + Checks Authorization header for JWT token and validates it against a Secret Key
* **Rate Limiting**
  + Objective is to Rate limit the number of queries/requests per second, is implemented using the Governor framework (Generic Cell Rate Algorithm) for Protecting the server from excessive load, for Rate Limiting. GCRA is a great option for providing a efficient, fair and flexible algorithm
    - Support for both **Direct** and **Keyed** Rate Limiting
    - Controls how much strain it is supposed to put on external services and **allow services to regulate how much load they take on from their users**
* **Depth Limiting**
  + Uses graphql\_depth\_limit for easy identification of possible malicious queries (high depth). This helps the server from being exploited from deep and cyclic queries.
  + Example -



**Cost Analysis**

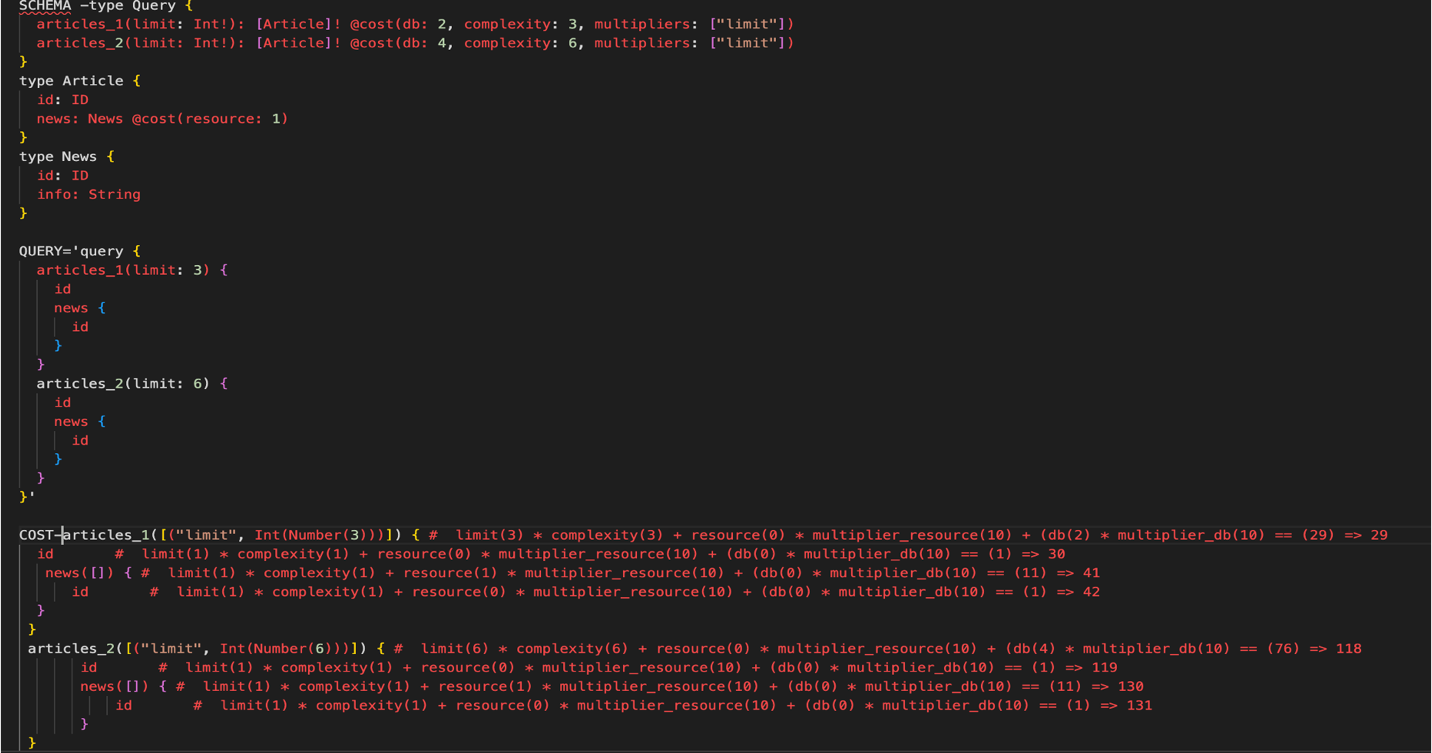
Cost Analysis is crucial for  **Threat Protection and Monetization**. Different types of Cost analysis

* **Static Query Analysis**(Phase 1) – **Extensively implemented at  Github,Shopify,IBM**
  + Statically analyze queries prior to execution weather to allow them based on the information provided on top of the schema. This will be applied at the Proxy Layer. (**Can grossly overestimate cost**)
* **Query Response Analysis** (Phase 2)
  + Collect the query response data ,query response size and tag them as labels
  + Use query complexity (Number of fields, depth), query, frequency, cache hit rate cpu/memory/Network usage, db performance as features
  + Use ML to predict the query cost with High accuracy
  + Ideal approach and can predict cost at high accuracy
* **Dynamic Query Analysis**
  + Set up Timeouts and abort execution if it surpasses the SLA
  + Add cost during query execution and abort at threshold (**Dangerous , may lead to partial query execution**)

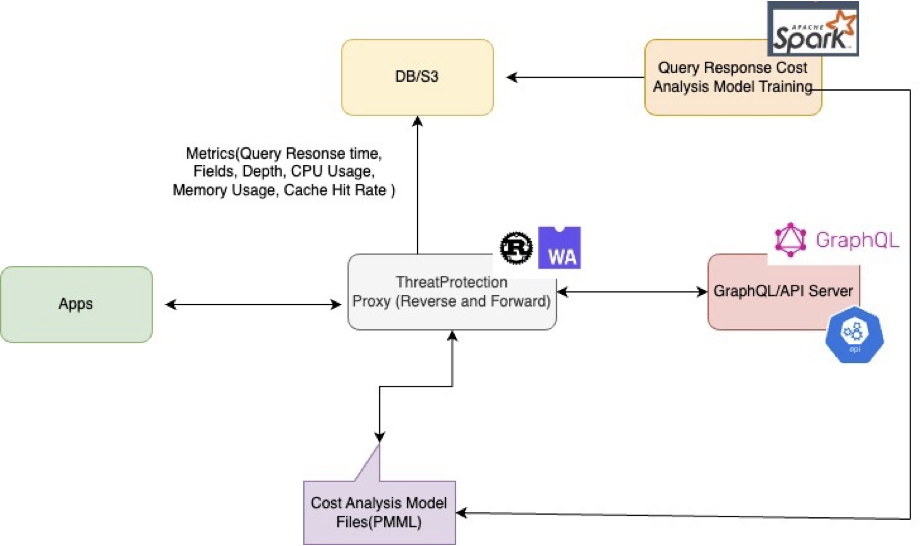
**Static Query Analysis  (Implementation Details- Phase 1)**

* Statically analyze queries prior to execution weather to allow them based on the information provided on top of the schema. This specification defines a directive that advertise enough information to calculate the cost of executing GraphQL queries.
  + Cost Directive Attributes-
    - Complexity
      * Fields that are moderate Objects, Complexity is 2
      * Fields that interact with external systems or Apis, complexity could be higher for example 5
      * Scalar Objects, Cost is 1
    - Dbcost
      * Simple Optimized queries, (cost is 1)
      * DB at a different AZ or region or highly analytical queries or large amount of data being returned can also have high network impact (cost is 10)
    - Servercost
      * Computationally expensive (High CPU/Memory/Network Usage) – example cost is 10 that has a high server resources usage
    - Multipliers
      * Field arguments for multiplying the complexity
      * Multipliers[Limit] :- How many objects in a list - Field argument for multiplying with complexity
      * multiplier\_dbcost: 10
      * multiplier\_recursion: 10
      * multiplier\_servercost: 10
    - Recursion
      * If recursion is detected cost gets affected exponentially – For example for one level of recursion , cost is 10, for the second level it’s set to 10^2 , so on and so forth

**Examples -**

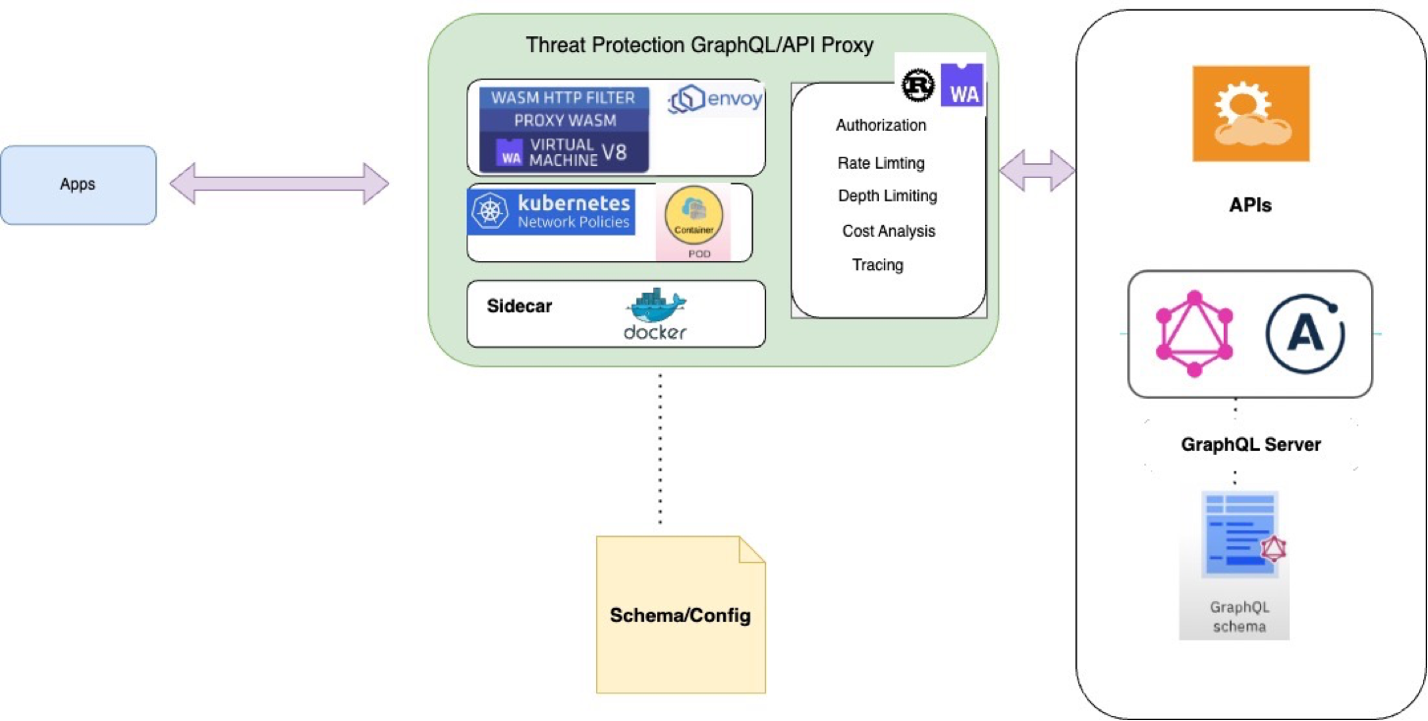


**Query Response Cost Analysis (Phase 2)**

****

* Expose Threatprotection module as both forward and reverse proxy
* Collect the query response data ,query response size from the server and tag them as labels
* Parse the query and use query complexity (Number of fields, depth), query, frequency, cache hit rate cpu/memory/Network usage, DB performance as features
* Build ML Model out of the collected information
* Use ML Model (PMML) to predict the query cost with High accuracy
* Ideal approach and can predict cost at high accuracy

**Implementation  Architecture**

****

**Built using Rust and WASM**

1.**High Performant**:-Blazing Fast-designed to be executed at near native speed **with Zero Runtime Overhead**

2.**Reliable**

3.**Scalability-** High concurrency and Scalable Asynchronous Design

4.HTTP/1 and **HTTP/2** support

5.**Excellent  Memory Efficient**:- There is no garbage collection heap-allocation by default. Rust would know when the variable gets out of scope or its lifetime ends at compile time and thus insert the corresponding LLVM/assembly instructions to free the memory.

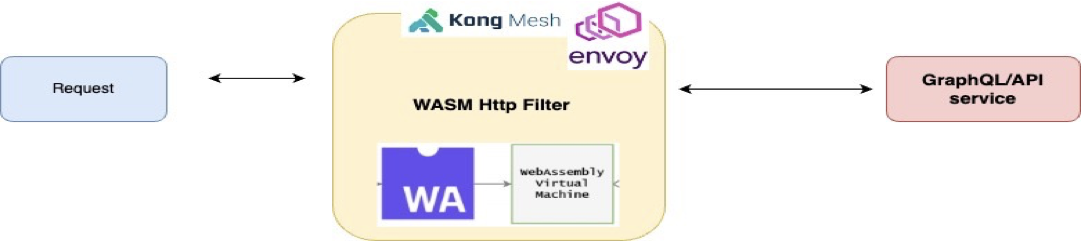
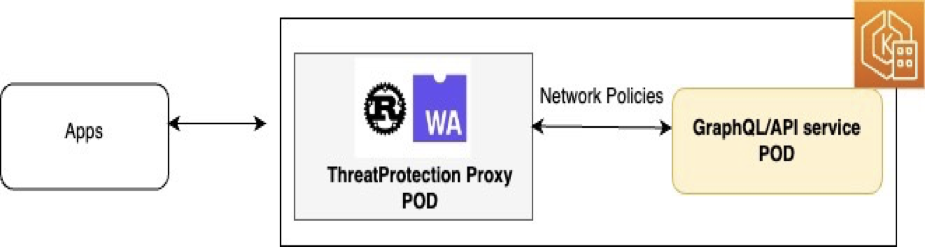
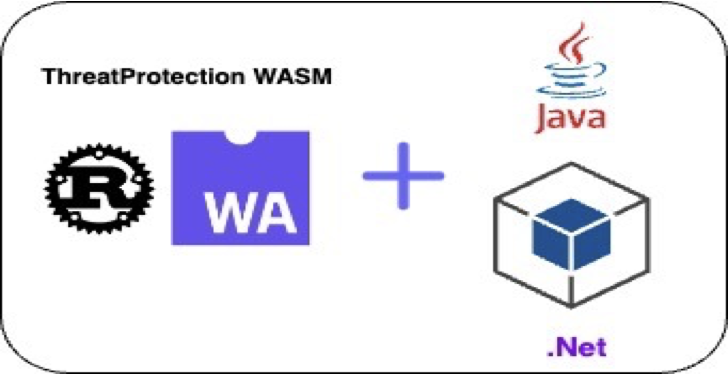
6.**Safety** :- Each WASM module executes within a sandboxed environment separated from the Host Runtime , communicates to Host Via an API

7.**Isolation**:- Crashes in  WASM VM don’t impact other parts of the system

8.**Portable and Interoperability-** Wide Range of Language Bindings

9.**Tracing-**Can be consumed by Prometheus, pushed as Open Telemetry, or collected in a centralized dashboard

**Integration Options**

* Deploy the Threat Protection Proxy **as a WASM Filter by taking advantage of the power of Envoy’s Proxy’s Extensibility**
  + ****
* Deploy the Threat Protection Proxy as a **Forward and Reverse proxy**for the main Server(Router or API). This can be achieved by using **Kubernetes Network Policies**.
* 
* Deploy as **Sidecar container**alongside Apollo Router that would intercept the requests by integrating this using a custom Apollo Router Native Rust Plugin.
* Deploy WASM workload directly in Kubernetes **Krustlet**(Support for Running WASM in Kubernetes- Experimental)
* **Interoperability-** Execute ThreatProtection Wasm modules within Java/.Net (Future State)
  + 

**Code Link-**