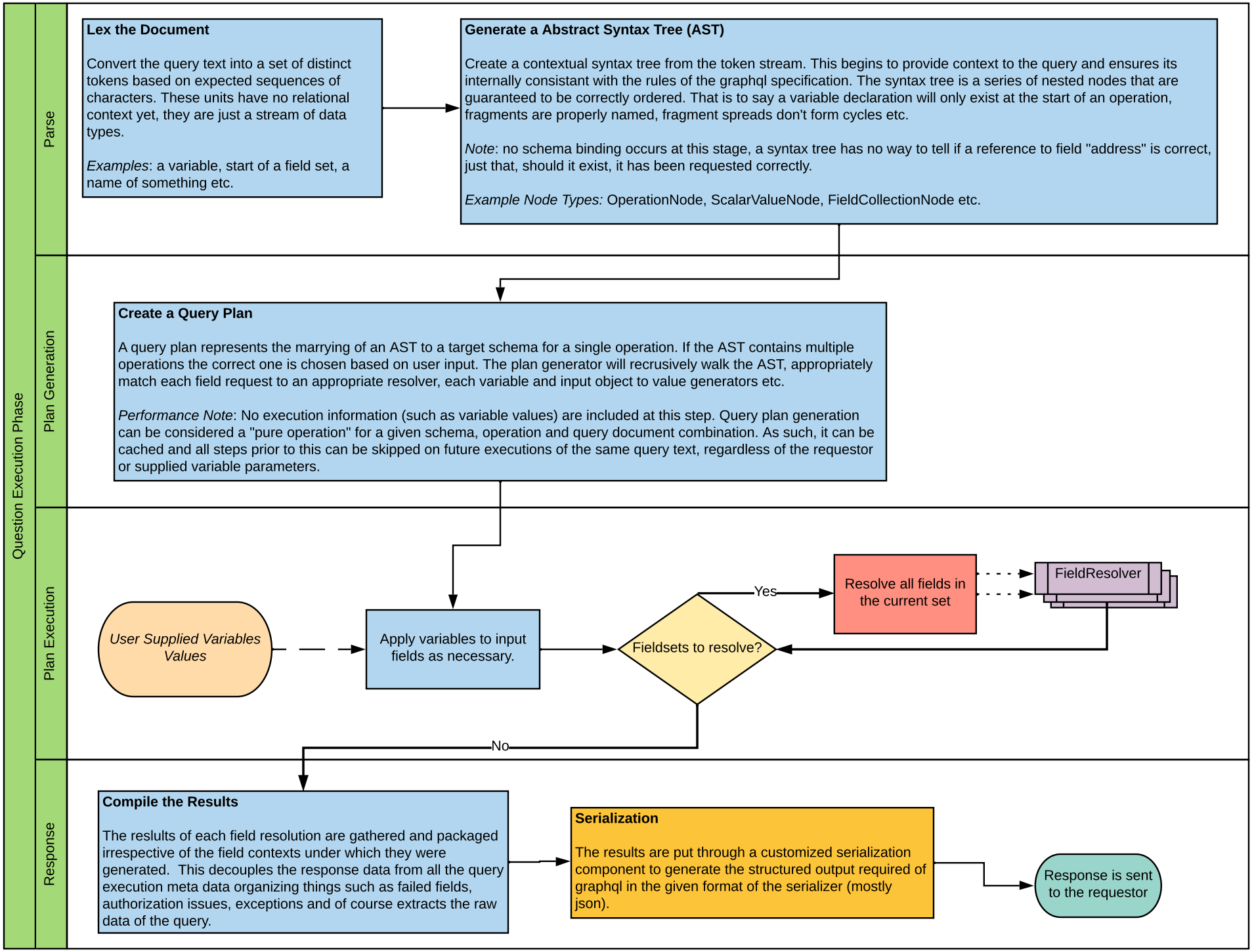
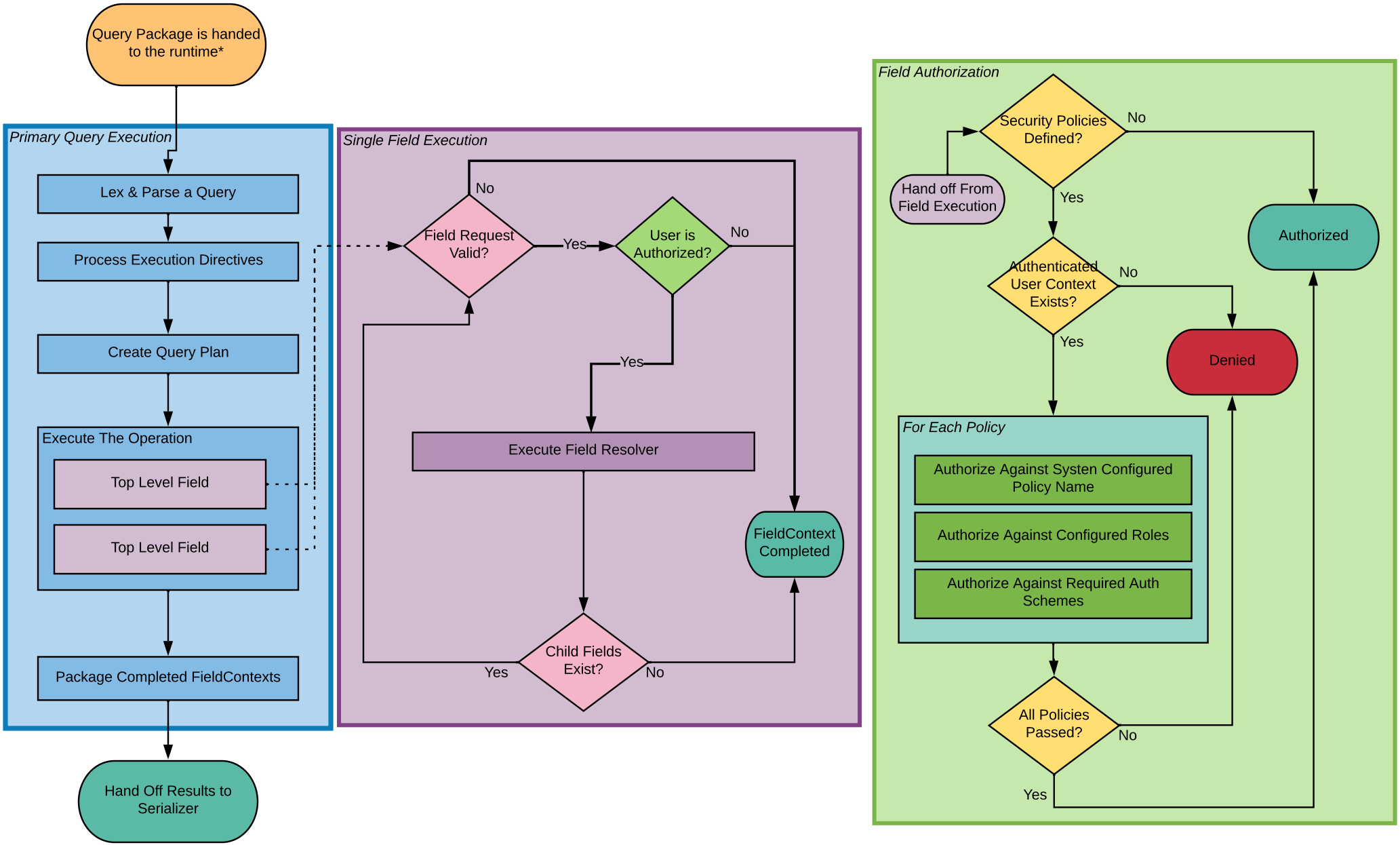
Graph Query Execution

When a query (or mutation) is received from a client it is handed off to the graphql runtime where a set series of steps are executed to fulfil the request. In general these steps are grouped into four distinct phases and executed as follows.



Request Processing

This diagram illustrates the conceptual activities, segmented by middleware pipelines, for completing a single query.



*Invocation of the GraphQL ASP.NET runtime is traditionally done via an HttpProcessor mapped to an URI via ASP.NET, however; this is simply a convience interation point. The runtime can be safely invoked anywhere as long as the correct data fields are supplied to it.

Primary Middleware Pipelines

GraphQL ASP.NET performs its query resolutions, from start to finish, in a set of 4 middleware pipelines that perform small incremental actions on a data context in order to produce a result. This diagram contains a description of the default components included in each pipeline. All pipelines are extensible via startup configuration settings.

Query Execution Pipeline

The primary workflow of a query. The ASP.NET runtime will hand off the raw data of the request (the query text and variable package) from a caller to this pipeline for processing.

This pipeline passes forward a `GraphQueryExecutionContext` containing top level information describing the request data and the authorized user, if any.

1) ValidateQueryRequest

Ensures that the request data, handed off from ASP.NET, could be completed in its current state. i.e. is the query text not blank, was pipeline context properly generated and supplied etc.

2) RecordQueryMetrics

Manages the performance metrics of the plan starting the clock and ending any performance counters still running as the pipeline unwinds.

3) ParseQueryDocument

Lexes and parses the document text into a syntax tree, which is attached to the context.

4) ValidateQueryDocument

Performs a first pass validation to ensure the constructed query document is internally consistant.

5) AssignOperation

Chooses the correct operation to be executed in multi-operation documents.

6) Validate Variable Data

Validates any supplied variable values against the variable declarations in the chosen operation.

7) QueryAuthorization (optional)

When included, this component will invoke the authorization pipeline for every "secure" field and directive (those that define auth rules) on the chosen operation and approve or deny the user access to continue executing the query as a whole.

8) Apply Execution Directives

Invokes the Directive Execution Pipeline for each execution directive applied to the query document's chosen operation.

9) GenerateQueryPlan

Creates a query execution plan for the chosen operation.

10) ExecuteQueryOperation

Spawns and completes set of Field Execution Pipelines for each top-level field in the requested operation. The results of each are attached to the execution context.

11) PackageQueryResult

Packages the completed field execution results into a final `IGraphOperationResult.`

This document represents the middleware component order in its default configuration. Out of the box, GraphQL ASP.NET can support either authorization scheme (field or query level) on a "per schema" basis. Each pipeline can be extended, interjected or completely changed to suit the developer's needs.

Field Execution Pipeline

For any field that needs to be resolved (i.e. when data is needed from user code to fulfill a graph query), each field is processed through its own pipeline to invoke the user's code in a secure and isolated manner.

This pipeline passes forward a `GraphFieldExecutionContext` containing all required field level data.

1) ValidateFieldExecutionRequest

Ensures that the request for the individual field can be completed in its current state. I.E. given the supplied source object a mapped resolver exists that can handle the request and all the required parameterized information for said resolver is present.

2) Authorization (optional)

When included, this component will invoke the schema item authorization pipeline for the single field approving or denying the user access to the data field.

3) ResolveField

The field resolver itself (the developers's code) is invoked and the results are recorded to the field context.

4) Process Child Fields

Kicks off the appropriate field execution pipelines for any child fields requested of the field just resolved.

Directive Execution Pipeline

For any directive that needs to be executed this pipeline is invoked allowing the directive to execute its code against the query document part its been applied to.

This pipeline passes forward a `GraphDirectiveExecutionContext` containing all required field level data.

1) ValidateDirectiveExecutionRequest

Ensures that the request for the individual directive can be completed in its current state. Is the directive employed in a valid location? Is it defined on the schema? etc.

2) Authorization (optional)

When included, this component will invoke the schema item authorization pipeline for the single directive approving or denying the user access to the execute said directive.

3) ResolveDirective

The directive's resolver is invoked. Each directive may or may not alter the query document in some way.

Schema Item Authorization Pipeline

For any schema item (field, directive etc.) that needs to be resolved, the is processed through its own pipeline to ensure the currently active user is authorized to access the item they are requesting.

This pipeline passes forward a `GraphSchemaItemSecurityContext` containing the metadata to complete an authN and authZ check.

1) Policy Aggregation

Inspects the security groups and policies attached to the provided schema item and generates a set of authN and authZ requirements that must be met.

2) Authentication

Using the security requriements from step one authenticates the user in an appropriate manner and generates a representation of the user (ClaimsPrincipal).

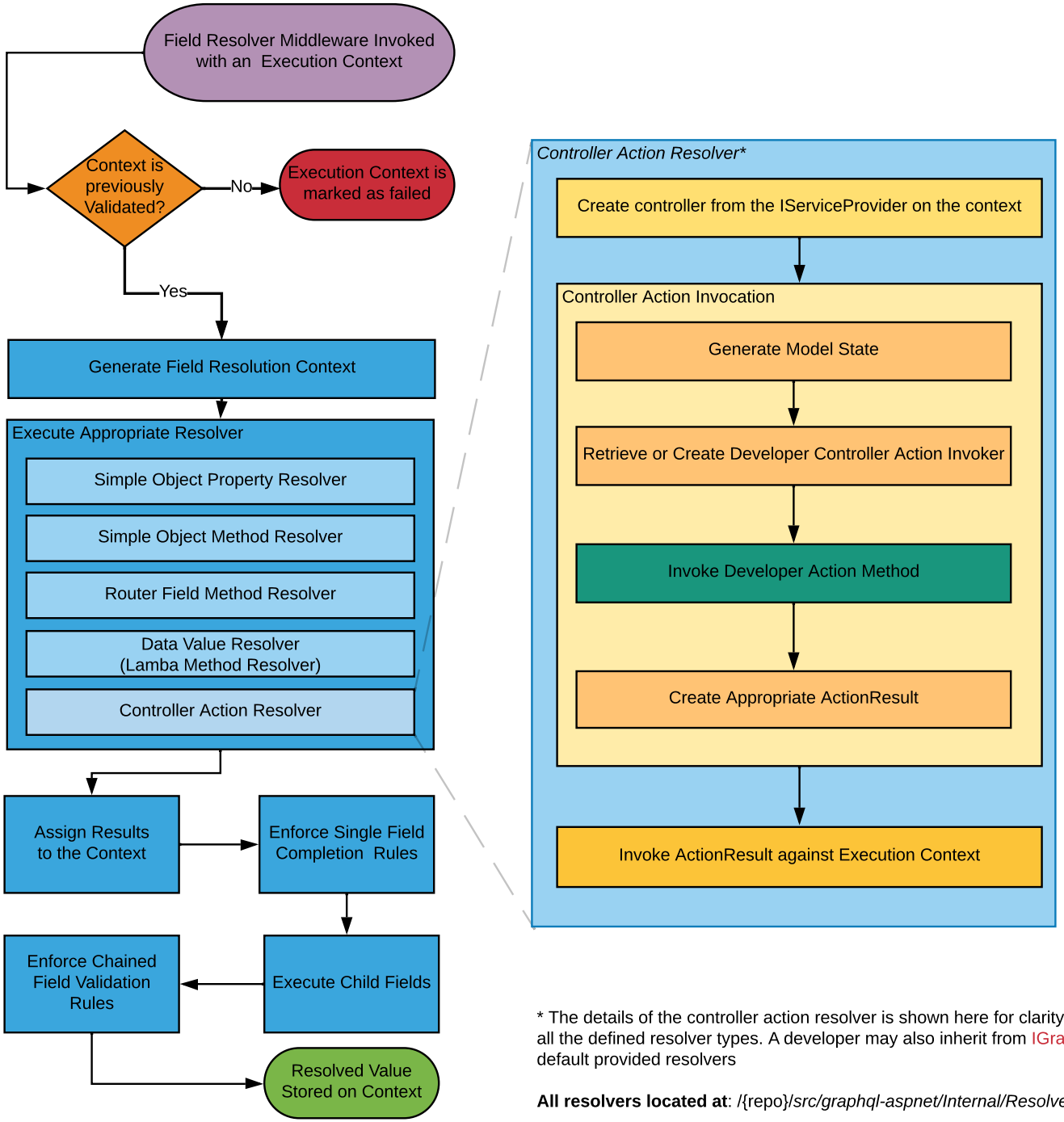
3) Authorization

Using the security requriements from step one and the authenticated user from step 2 authorizes the user against the required policies, roles etc. and generates a challenge result indicating success or failure.

This document represents the middleware component order in its default configuration. Out of the box, GraphQL ASP.NET can support either authorization scheme (field or query level) on a "per schema" basis. Each pipeline can be extended, interjected or completely changed to suit the developer's needs.

Invoking a Single Field Resolver

This diagram illustrates how a single field is eventually resolved marrying the library's internal code to the developer's code for performing meaningful business logic.



Resolver Descriptions

Object Property Resolver: Extracts a single value of a single property from the source object.

Object Method Resolver: Executes a single public method on the source object and uses the returned value as the result.

Route Field Method Resolver: An internally used resolver to handle virtual fields generated to support the developer's requested graph structure when the developer doesn't specifically create a controller action for each level of their graph hierarchy

Data Value Resolver: Similar to the method resolver this resolver executes a supplied lambda to perform some action and generate a result.

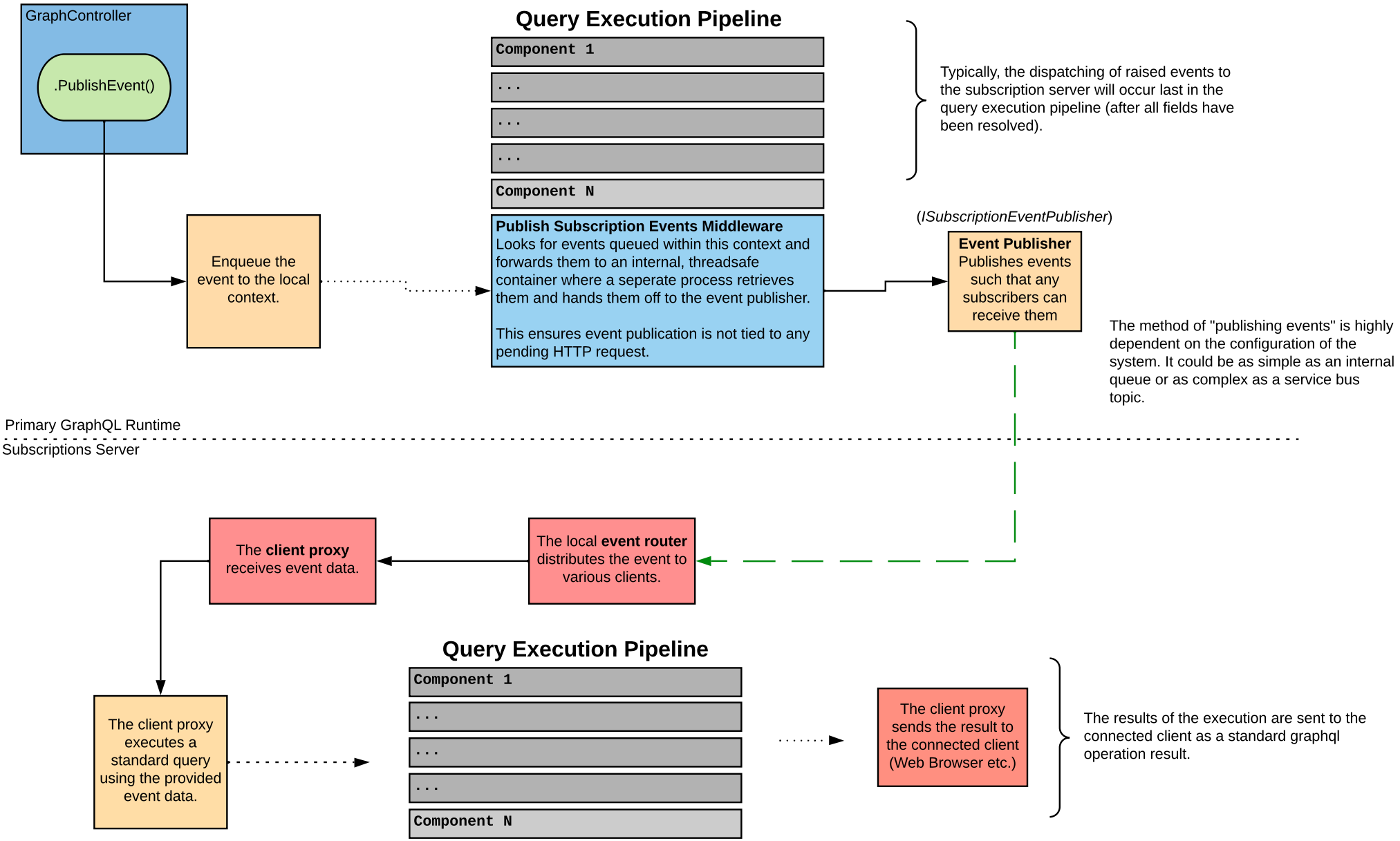
Controller Action Resolver: Executes an ASP.NET MVC style controller with appropriate model level validation and processing of generated ActionResults.

* The details of the controller action resolver is shown here for clarity of process. It is the most used and most complex of all the defined resolver types. A developer may also inherit from IGraphFieldResolver and create new or replace any default provided resolvers

All resolvers located at: /{repo}/*src/graphql-aspnet/Internal/Resolvers/**

Raising a Subscription Event

This diagram illustrates how a named event is raised from a GraphController and is passed through to a subscribed client.

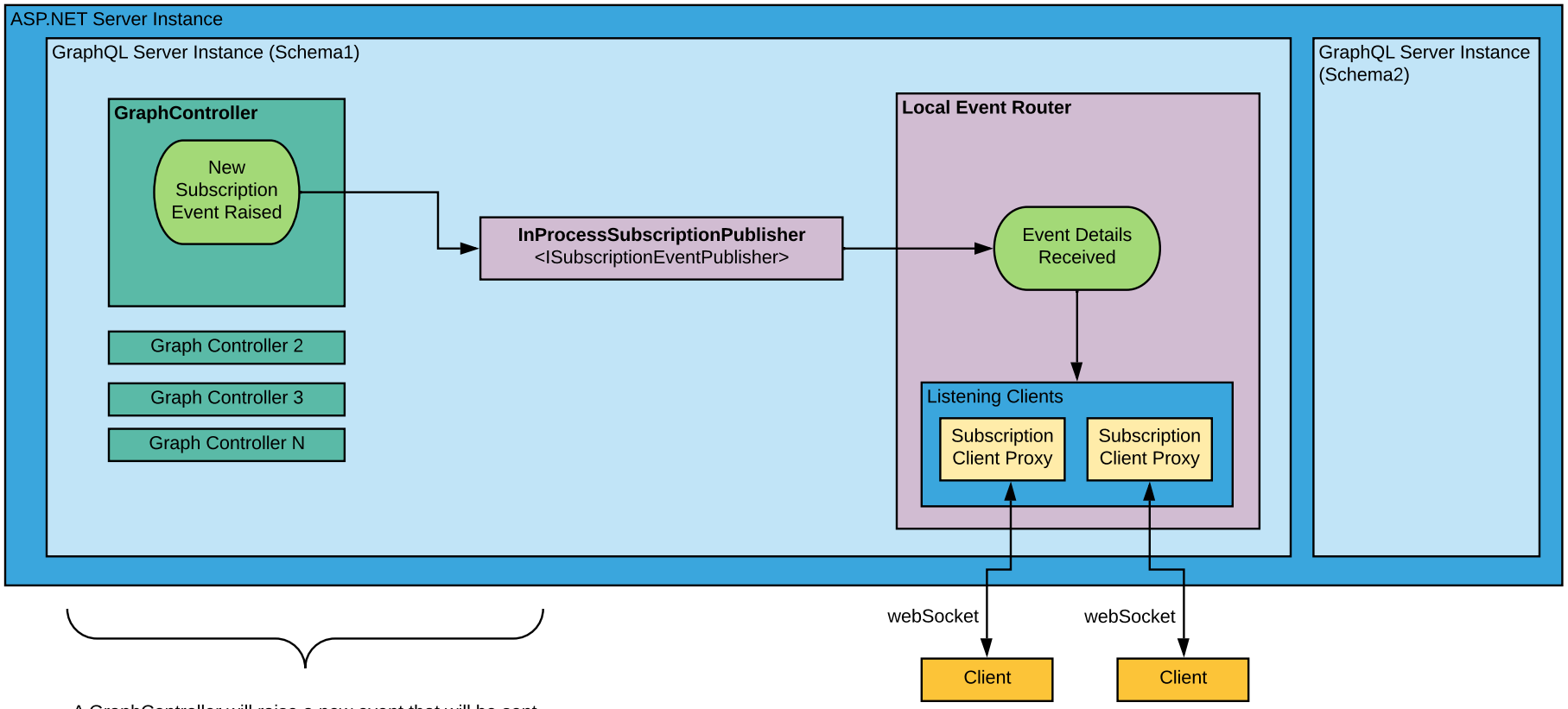


Subscriptions (In Process)

This diagram represents the primary logical components of the graphql subscription server when it is hosted in the same instance as the primary runtime.

Pros: No moving parts. The subscriptions are hosted right along side your query/mutation requests. There is near zero delay from when a subscription event is raised to when its dispatched to a subscription.

Cons: This solution provides no scalability. For smaller implementations this may work fine. If you ever have to introduce a second GraphQL server to balance query load or reach a prohibitive number of websocket connections this solution will fail. When scaled horiztonally some subscribed clients will not receive events raised by another server instance.



A GraphController will raise a new event that will be sent to an in-process publisher and raised to the local event router. Events are not published beyond the boundry of the ASP.NET server instance.

Subscriptions (Out of Process)

This diagram represents the primary logical components of the graphql subscription operation and how the primary runtime interacts with the server(s) hosting the websocket connections.

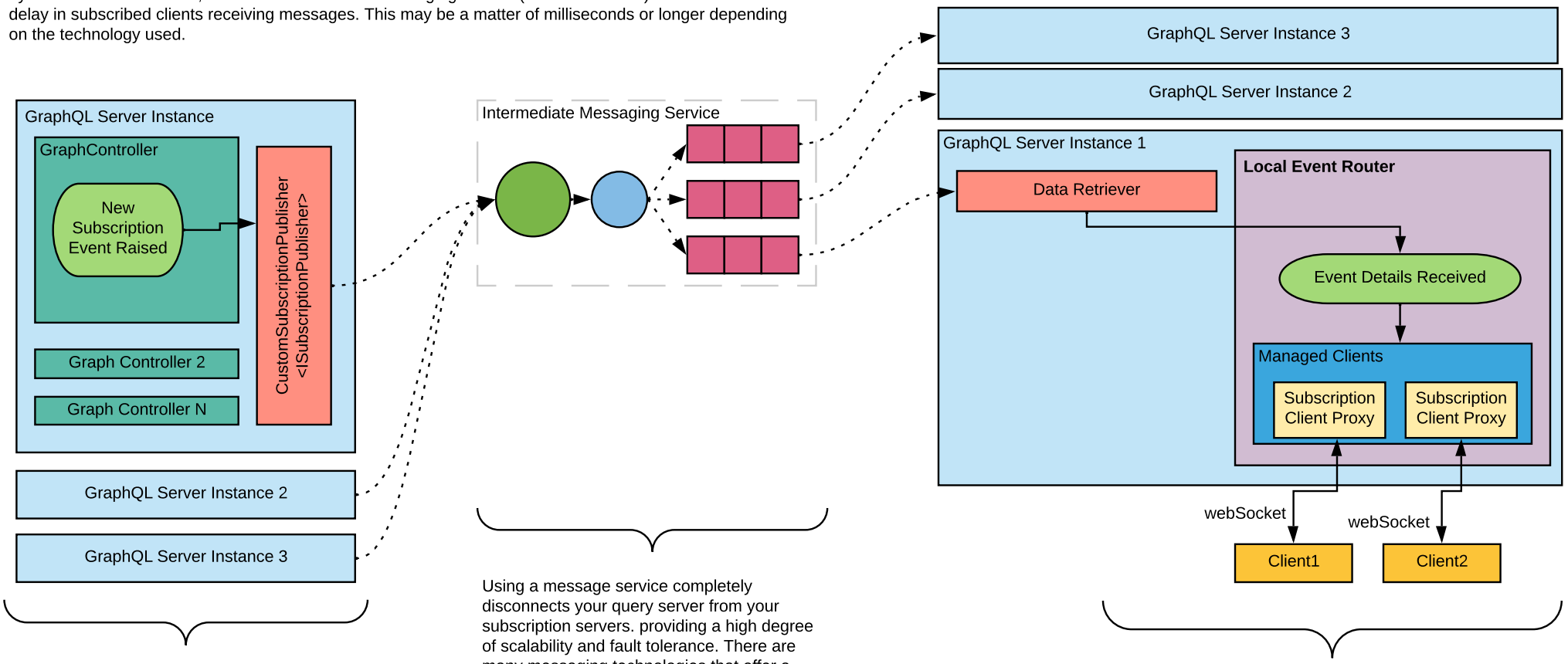
PROS: This approach gives you a high level of scalability by using an intermediary (like a broked message queue) to completely disconnect the graphql query/mutation server from the subscription server instances. Each "raised event" is emmited once by any given query server and the event is delivered to each server listening for it.

CONS: This approach is more complex to implement (it has more moving parts) and requires an out of process mechanism to manage messages and process events. Your graphQL subscriptions will be limited by such constraints. Also, use of an intermediate messaging service (or database etc.) will introduce some delay in subscribed clients receiving messages. This may be a matter of milliseconds or longer depending on the technology used.

An out of process subscription server requires creating two components:

Subscription Publisher: Implement *ISubscriptionPublisher* and deliver subscription events to some intermediate data source like a message queue. This object must be registered with your DI container.

Data Retriever: Create some sort of data retrieval process such that each server instance can pull a copy of a subscription event when needed. Deserialize the event and forward it to the server's ISubscriptionEventRouter (available through DI). You do not need to perform any processing on the message other than deserializing it.



Under high loads you may have multiple, load-balanced instances of your application. Any of which may raise events at any time..

Using a message service completely disconnects your query server from your subscription servers. providing a high degree of scalability and fault tolerance. There are many messaging technologies that offer a wide range of features for different scenarios.

Potential Tech: Redis, RabitMQ, Azure Service Bus, Amazon MQ

In this setup, the data retriever may be a custom built IHostedService that monitors a message queue.

The data retriever then deserializes subscription events from the queue and forwards them to the local router to be processed and distributed to any connected clients.