SAND2019-1704PE

Geophysical Monitoring System Processing Service Architecture: Control Applications and Plugins

PRESENTED BY

Ben Hamlet

SAND2019-AAAAAA

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Outline

Control Based Architecture

Control Application Architecture Responsibilities

Plugin Architecture Responsibilities

Control Application Sequences

- Startup
- Processing Request

Architecture Runway

Control Based Architecture

Primary concept: Implement Monitoring Business Logic

Control Applications

- Entry point for automatic processing business logic
  - Filtering, beaming, FK, detection, association, location, magnitude, etc.
  - Accessed from automatic processing sequences and UI
- Independent of other control applications
  - Support novel processing sequences
  - Develop and replace in insolation
- Relocatable to multiple environments (testbeds, data center ops, field laptops, etc.)
- Intentionally dependent on the conventions and technologies of the broader GMS ecosystem
  - Expose service routes, interact with data persistence mechanism, application monitoring, …

Plugins

- Implement algorithms
- Extension point for new algorithm implementations
- Loosely dependent on the broader GMS ecosystem

Control Application Architecture Responsibilities

Provide access to common business logic via external interfaces

- Automatic processing interfaces
  - Streaming: Consume data objects available to process
  - Descriptor: Consume descriptions of the data objects available to process; load data from OSD
- Interactive: tailored to UI needs

Data Access and Persistence via OSD

- Load data based on descriptors
- Load additional data required to serve processing request
- Store processing results and create descriptors

Plugin Registry Management

- Discover and register plugins at startup
- Select and invoke correct plugins for each processing request

Configuration

- Load and cache at startup; receive updates at runtime
- Resolve processing parameters during each processing request

Implement general application responsibilities with project standard technologies and frameworks

- Logging, configuration, process monitoring, external service communication, etc. (see *Architecture Overview*)
- Consume and produce COI data objects

Plugin Architecture Responsibilities

Address GMS Project Principles

- Extensibility
  - Integrate new algorithms
  - Isolate algorithm implementations from GMS libraries, frameworks, etc.
    - Path to implement algorithms in languages other than Java
- Scalability
  - Control Applications deployed in different GMS environments (laptop through datacenter)
    - Same applications operate in each environment, possibly at reduced functionality
    - Access algorithm implementations appropriate to those environments
    - e.g. 3D earth models and waveform correlation may not be feasible on a laptop
  - Different algorithm deployments (service vs. in-memory) based on client application's requirements
- Maintainability
  - Access related algorithms through common interfaces from the same Control application logic

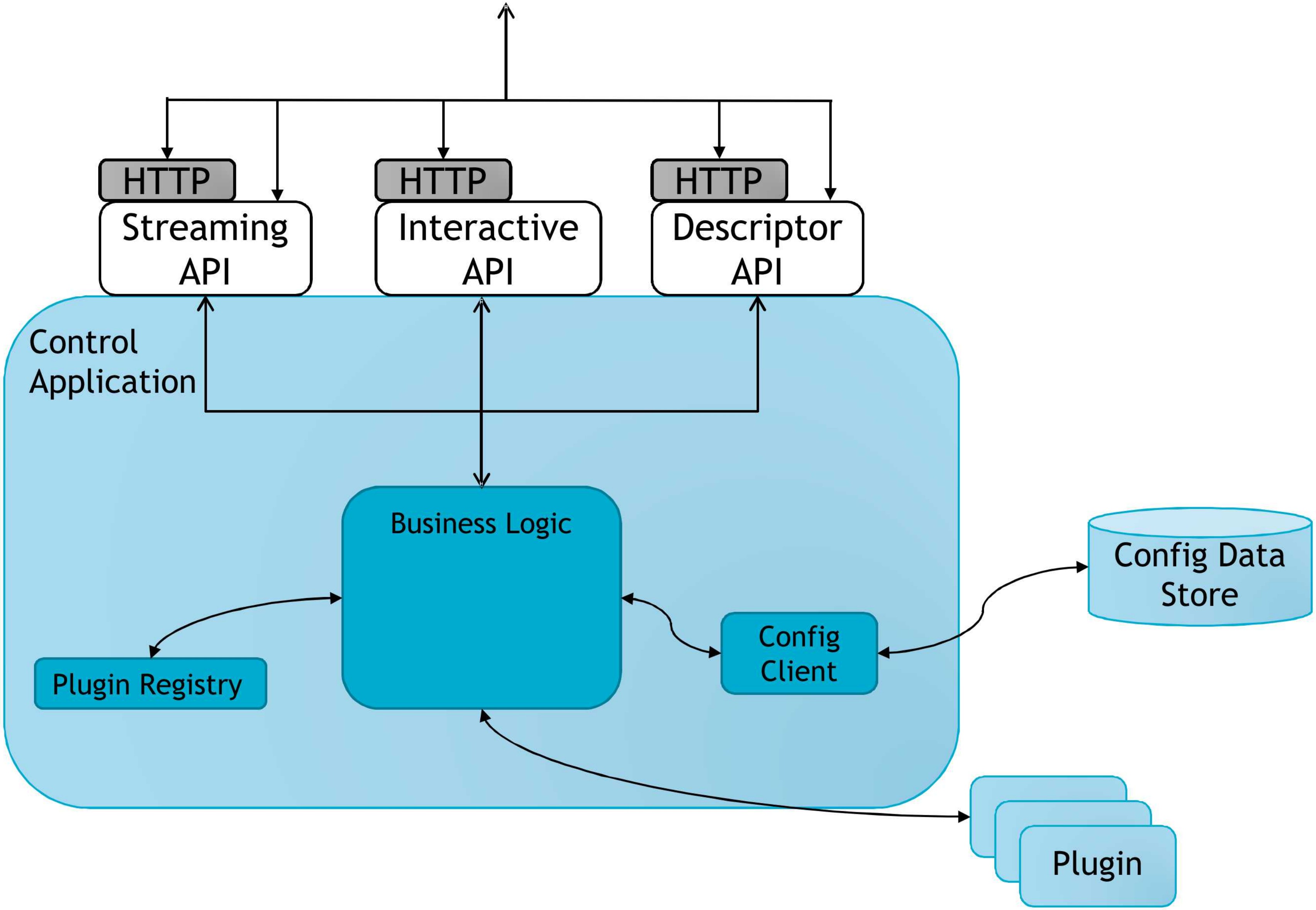
Design Goals

- Dynamically discoverable at runtime
- Isolate algorithm logic from GMS control applications, processing flows, and OSD interactions.
- Simple interfaces reimplemented by a variety of algorithms from the same family.

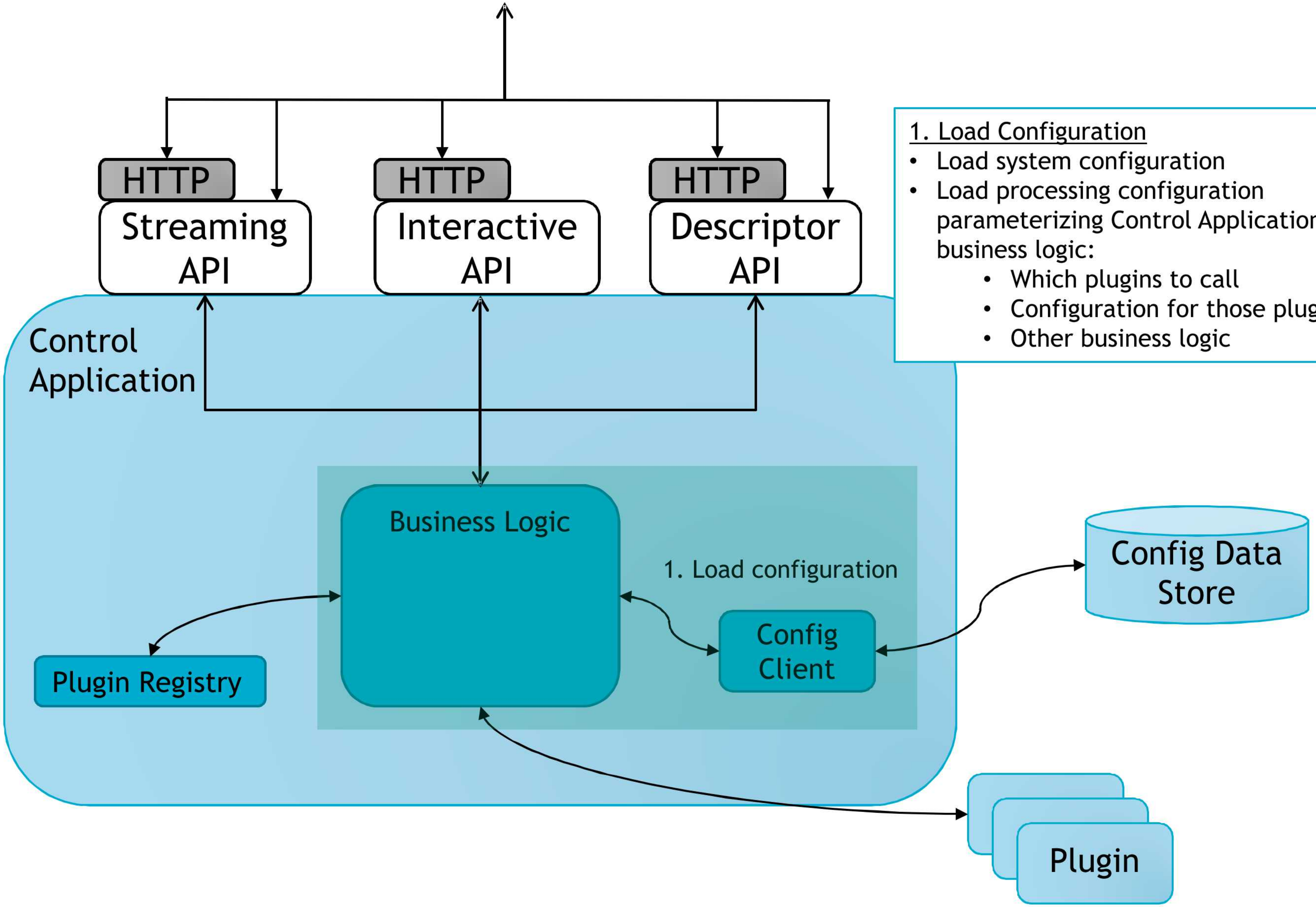
Control Application Runtime Sequences

Startup

Sequence: Control Application Startup (1/6)



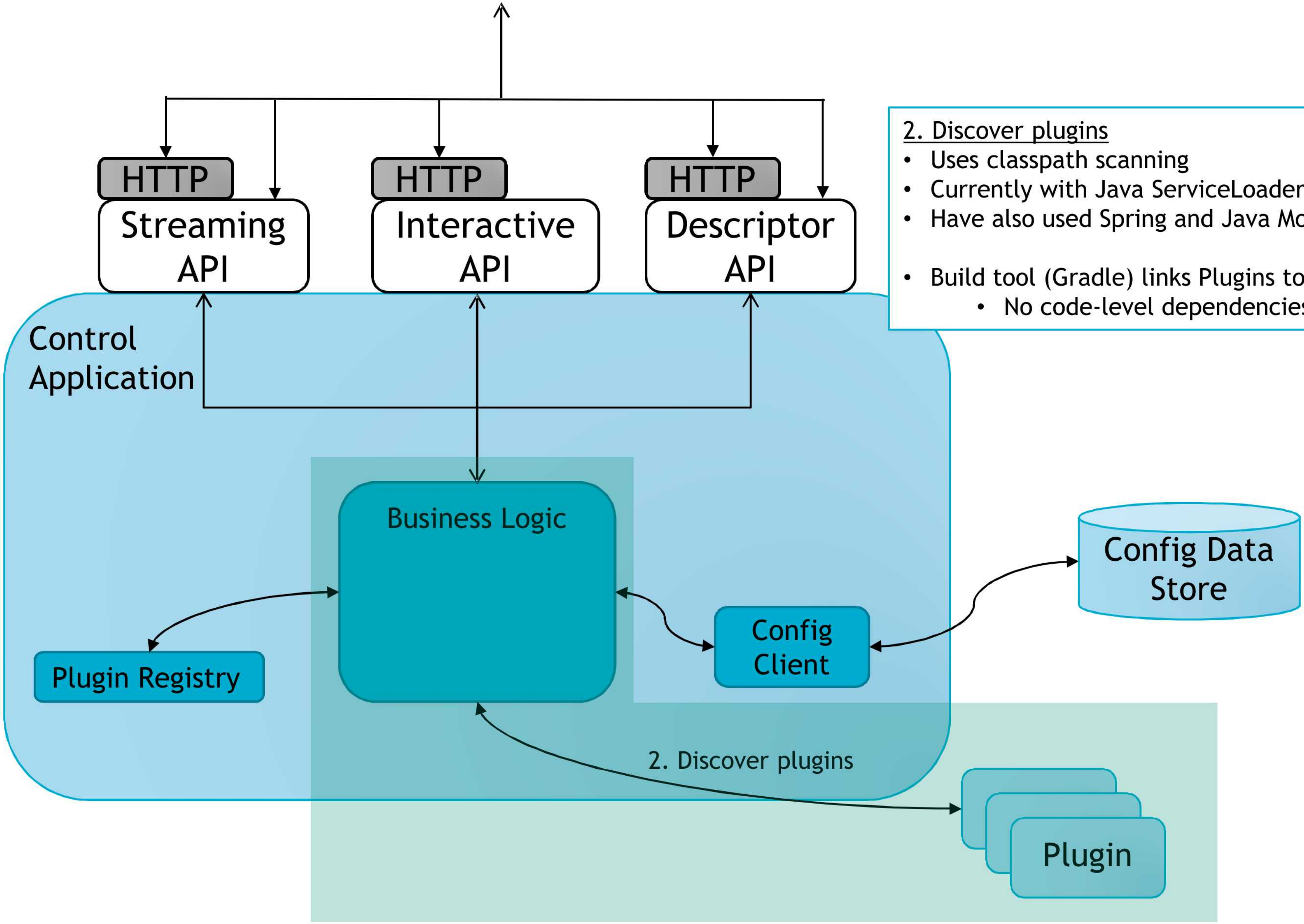
Sequence: Control Application Startup (2/6)



1. Load Configuration

- Load system configuration
- Load processing configuration parameterizing Control Application's business logic:
  - Which plugins to call
  - Configuration for those plugins
  - Other business logic

Sequence: Control Application Startup (3/6)

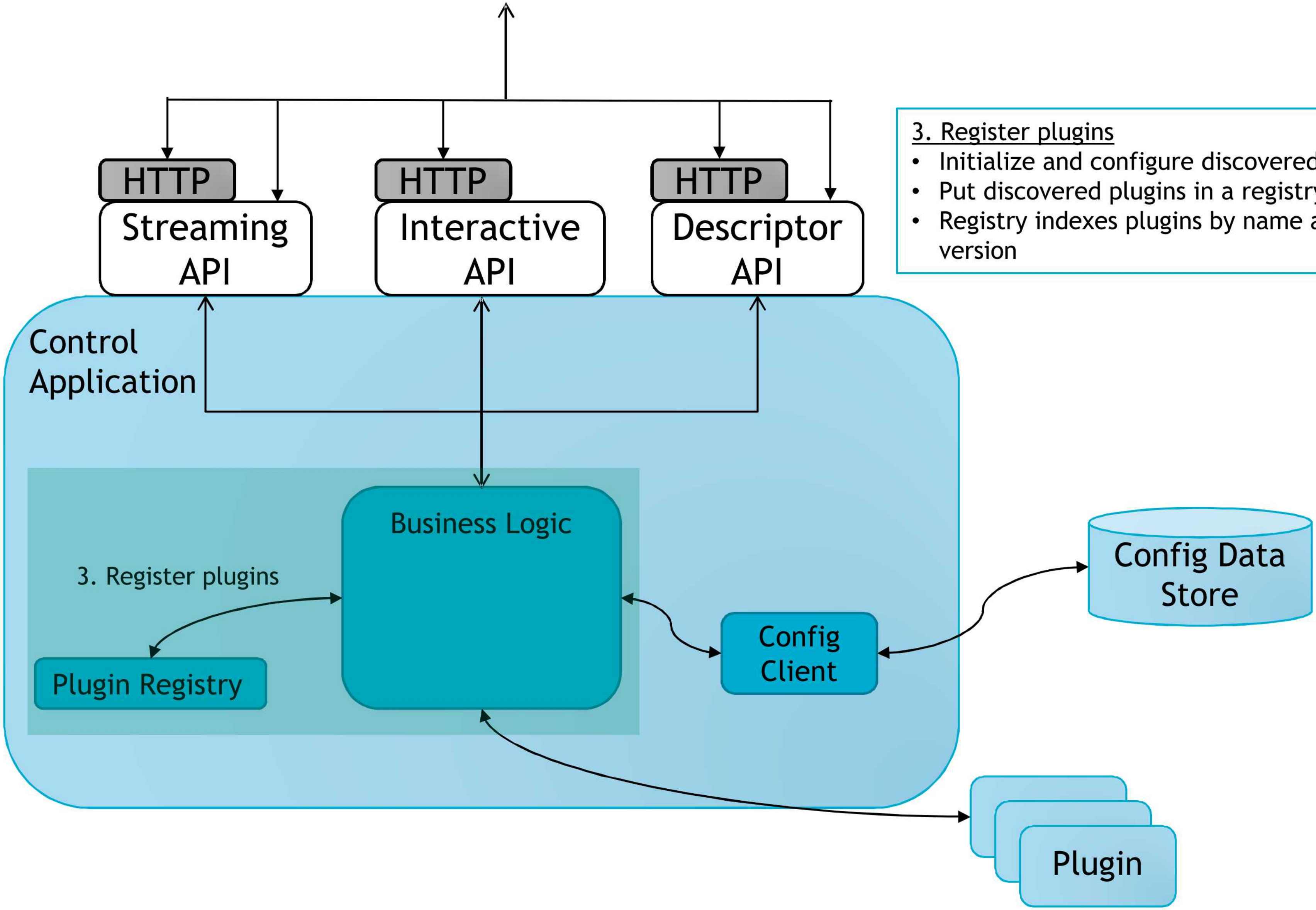


<u>2. Discover plugins</u>

- Uses classpath scanning
- Currently with Java ServiceLoader
- Have also used Spring and Java Modules

- Build tool (Gradle) links Plugins to Control
  - No code-level dependencies

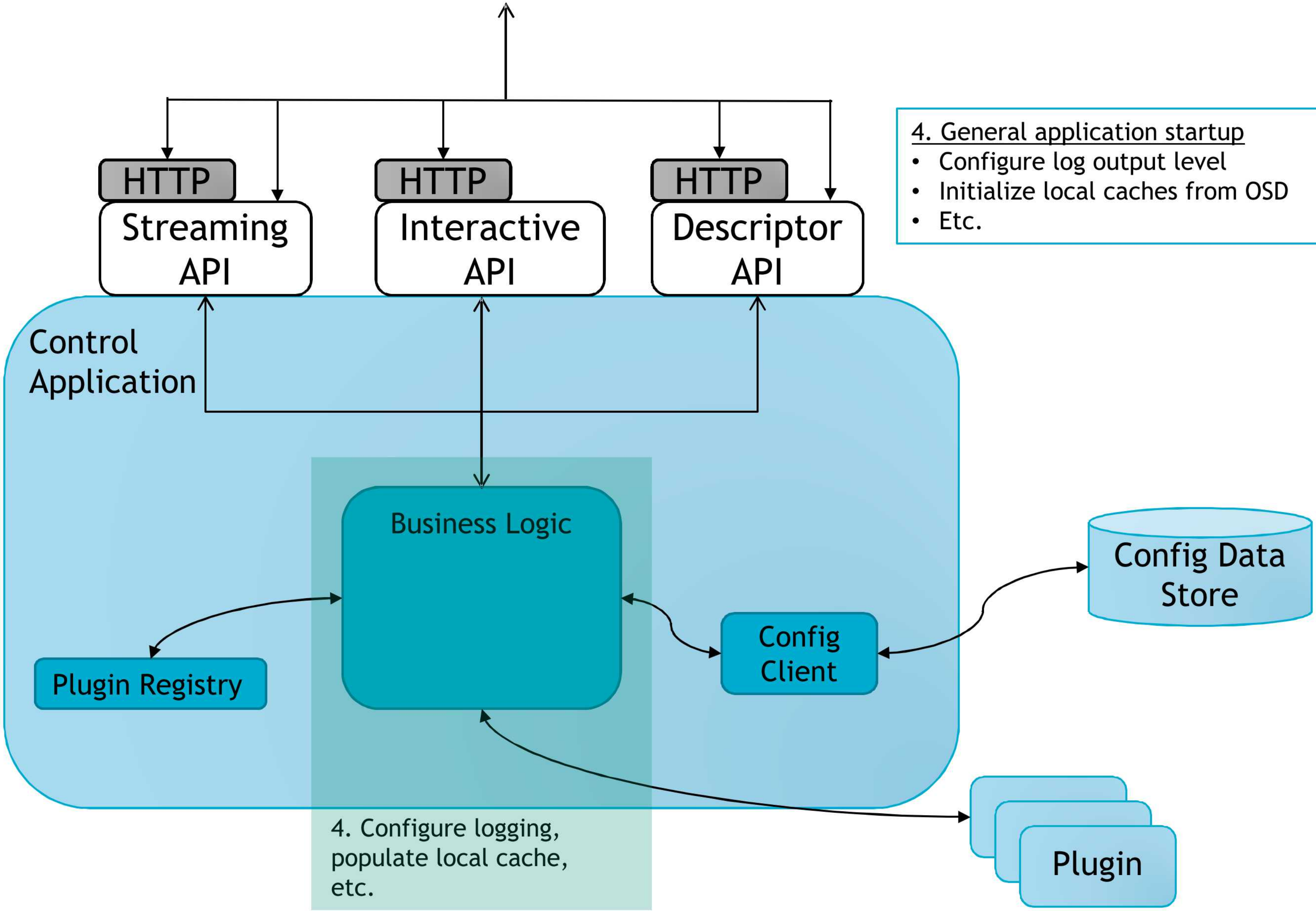
Sequence: Control Application Startup (4/6)



3. Register plugins

- Initialize and configure discovered plugins
- Put discovered plugins in a registry
- Registry indexes plugins by name and version

Sequence: Control Application Startup (5/6)

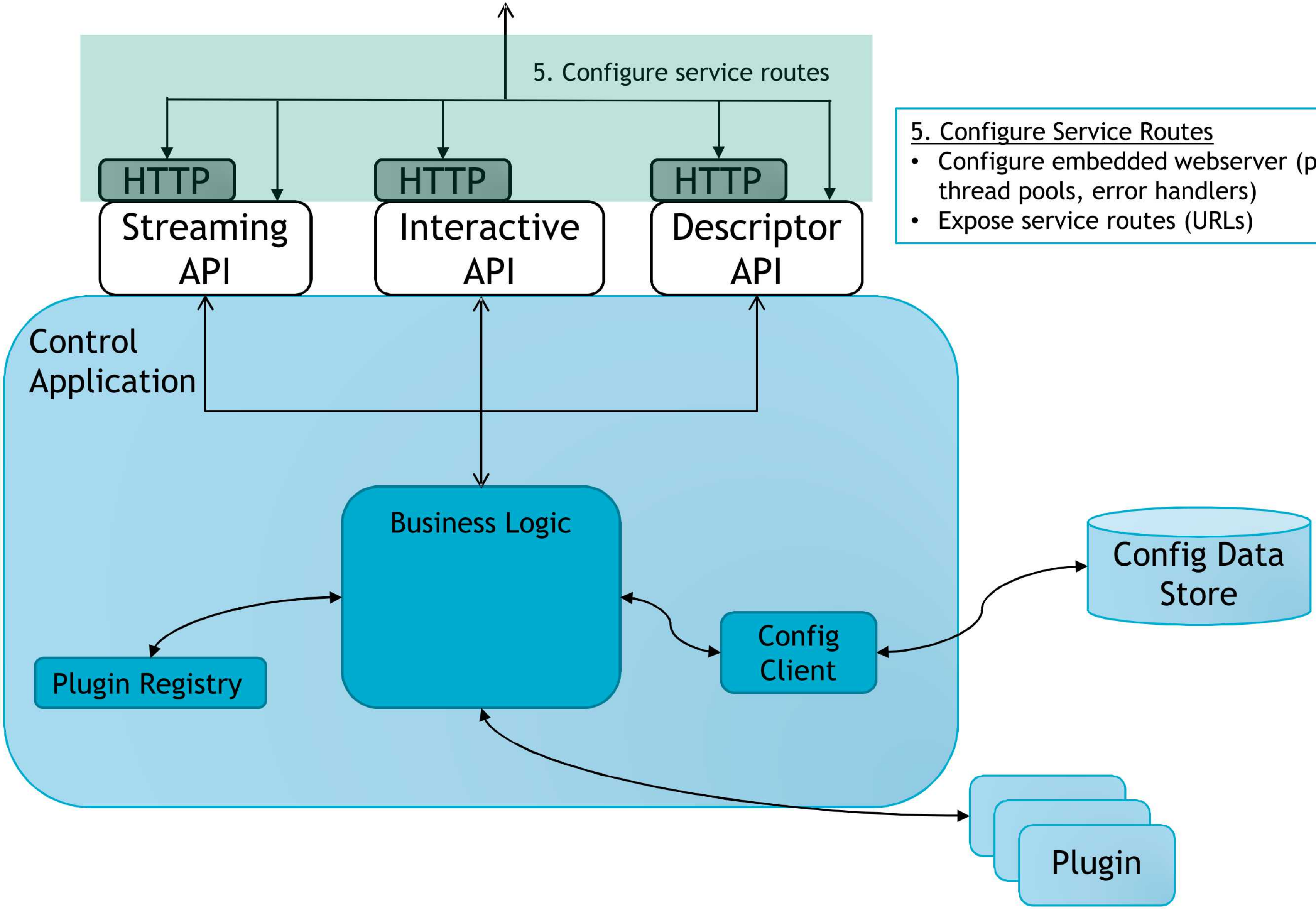


4. Configure logging, populate local cache, etc.

<u>4. General application startup</u>

- Configure log output level
- Initialize local caches from OSD
- Etc.

Sequence: Control Application Startup (6/6)



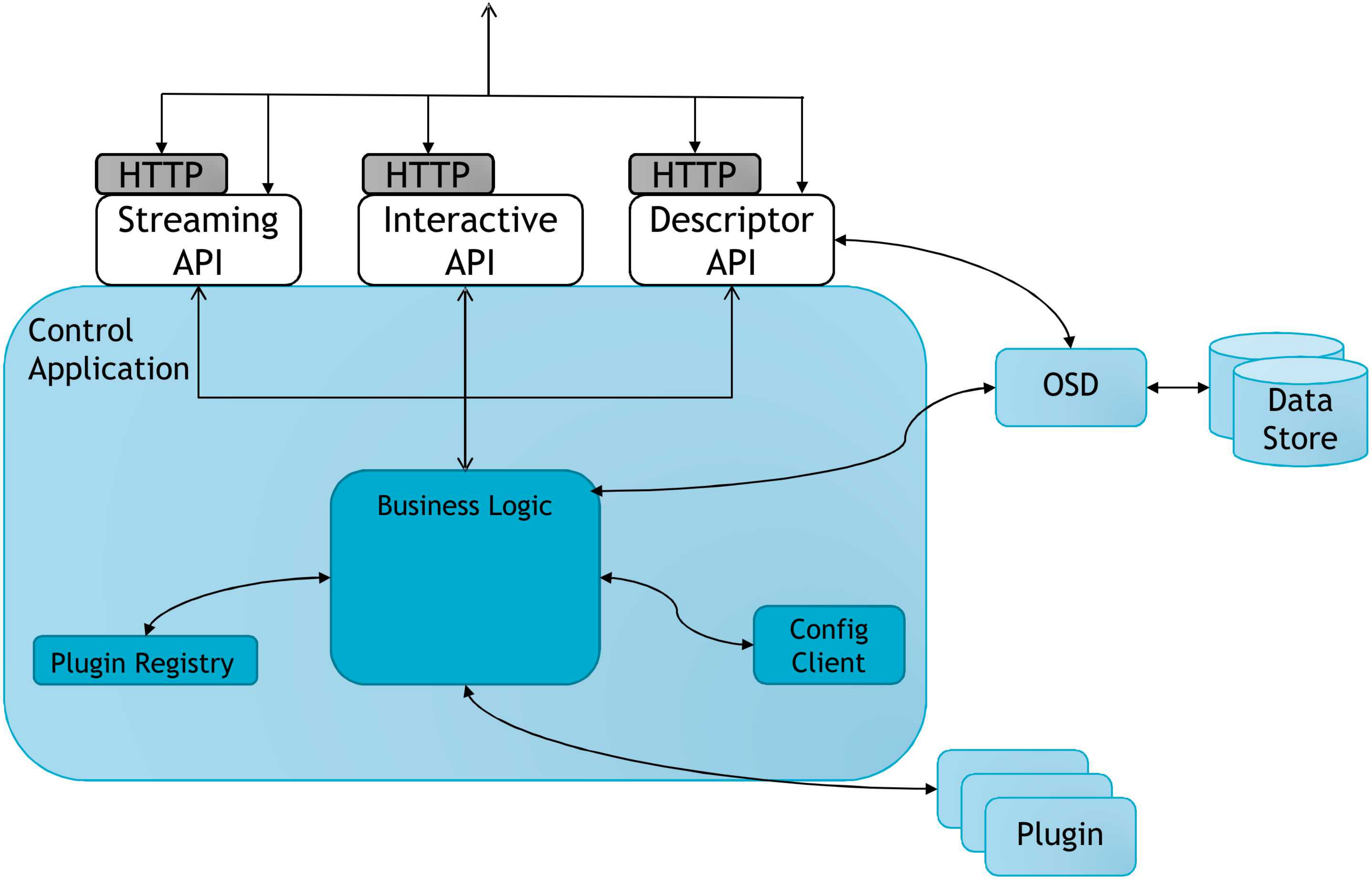
5. Configure Service Routes

- Configure embedded webserver (ports, thread pools, error handlers)
- Expose service routes (URLs)

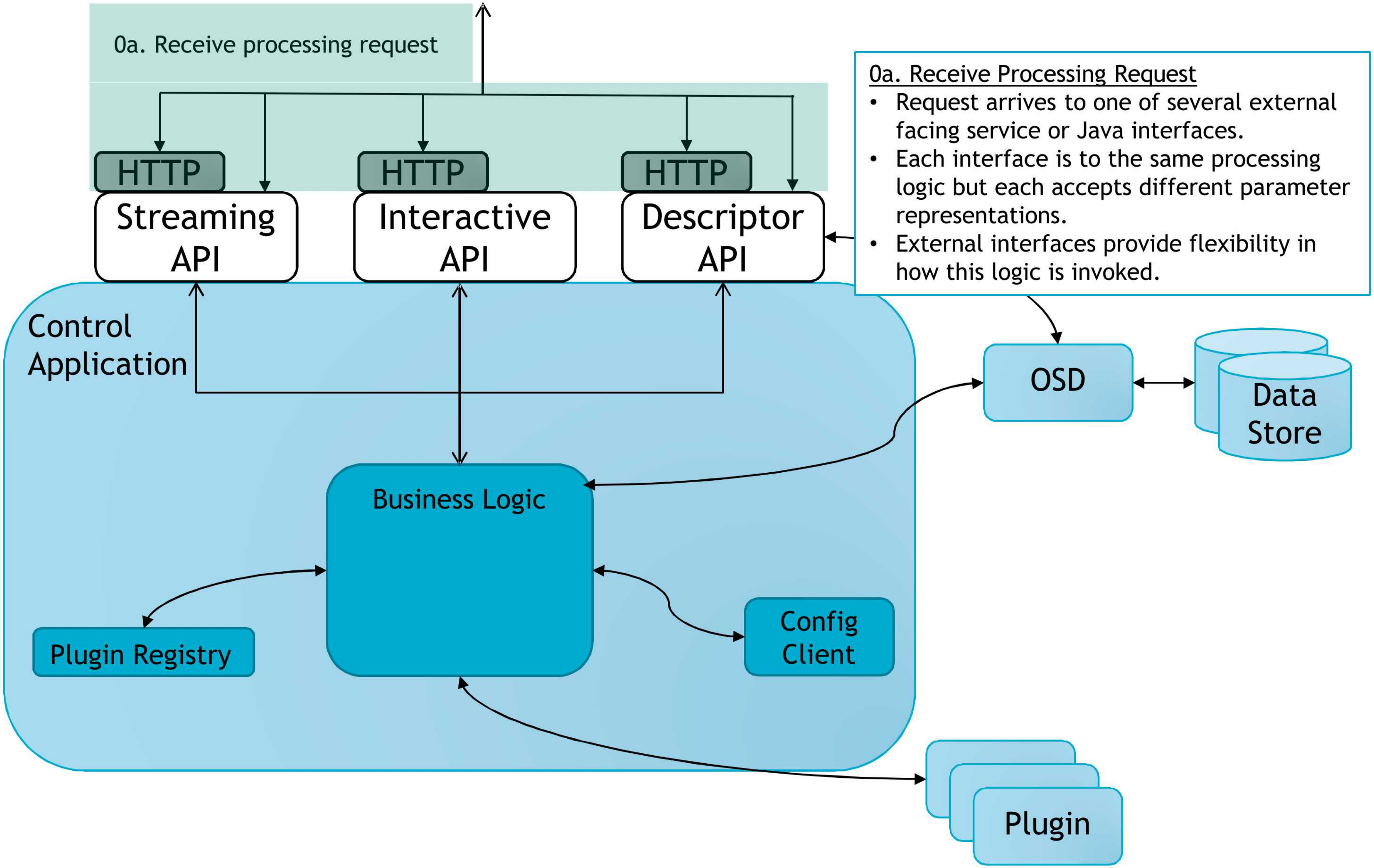
Control Application Runtime Sequences

Processing Request

Sequence: Control Serves Processing Request (1/13)



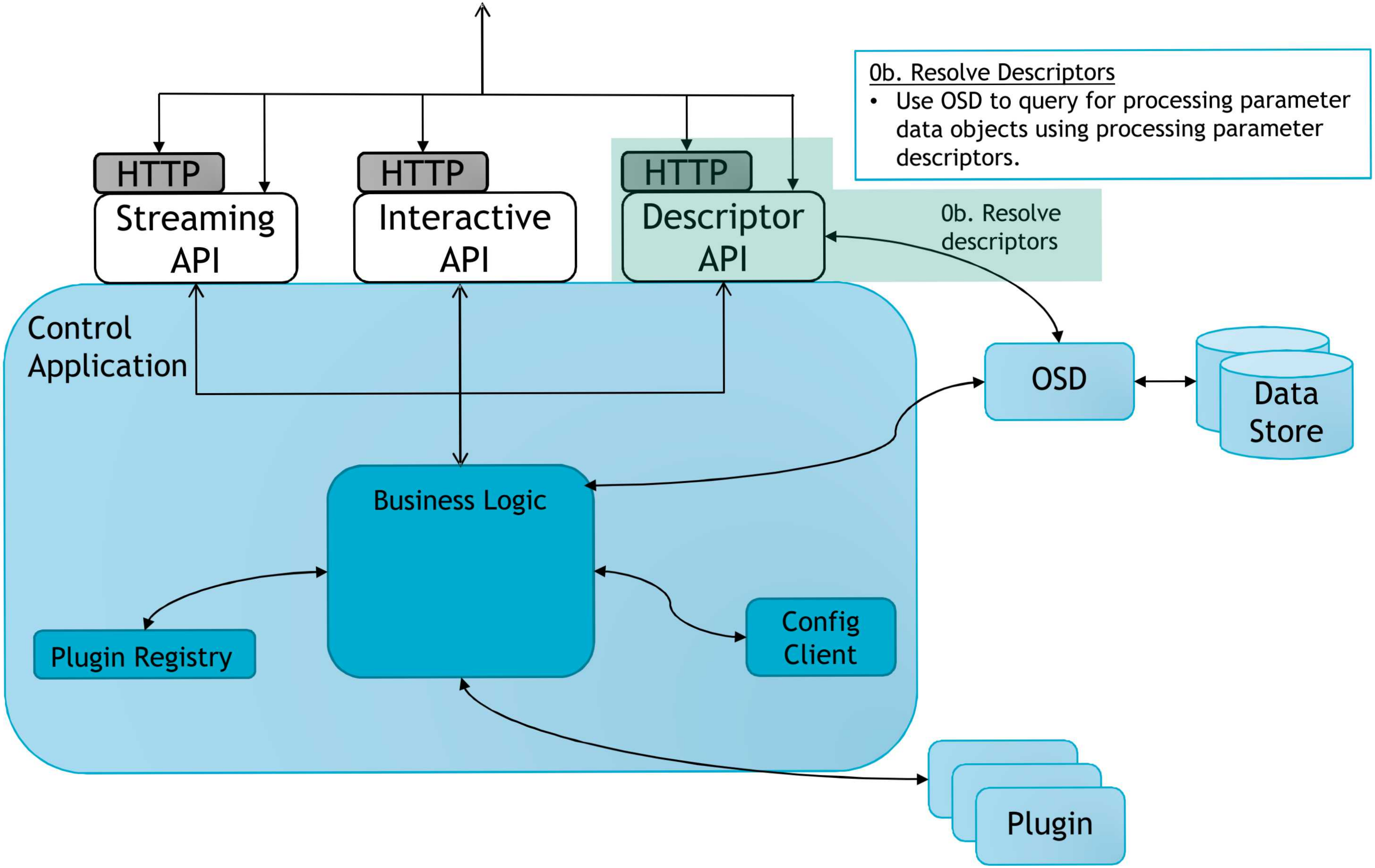
Sequence: Control Serves Processing Request (2/13)



0a. Receive Processing Request

- Request arrives to one of several external facing service or Java interfaces.
- Each interface is to the same processing logic but each accepts different parameter representations.
- External interfaces provide flexibility in how this logic is invoked.

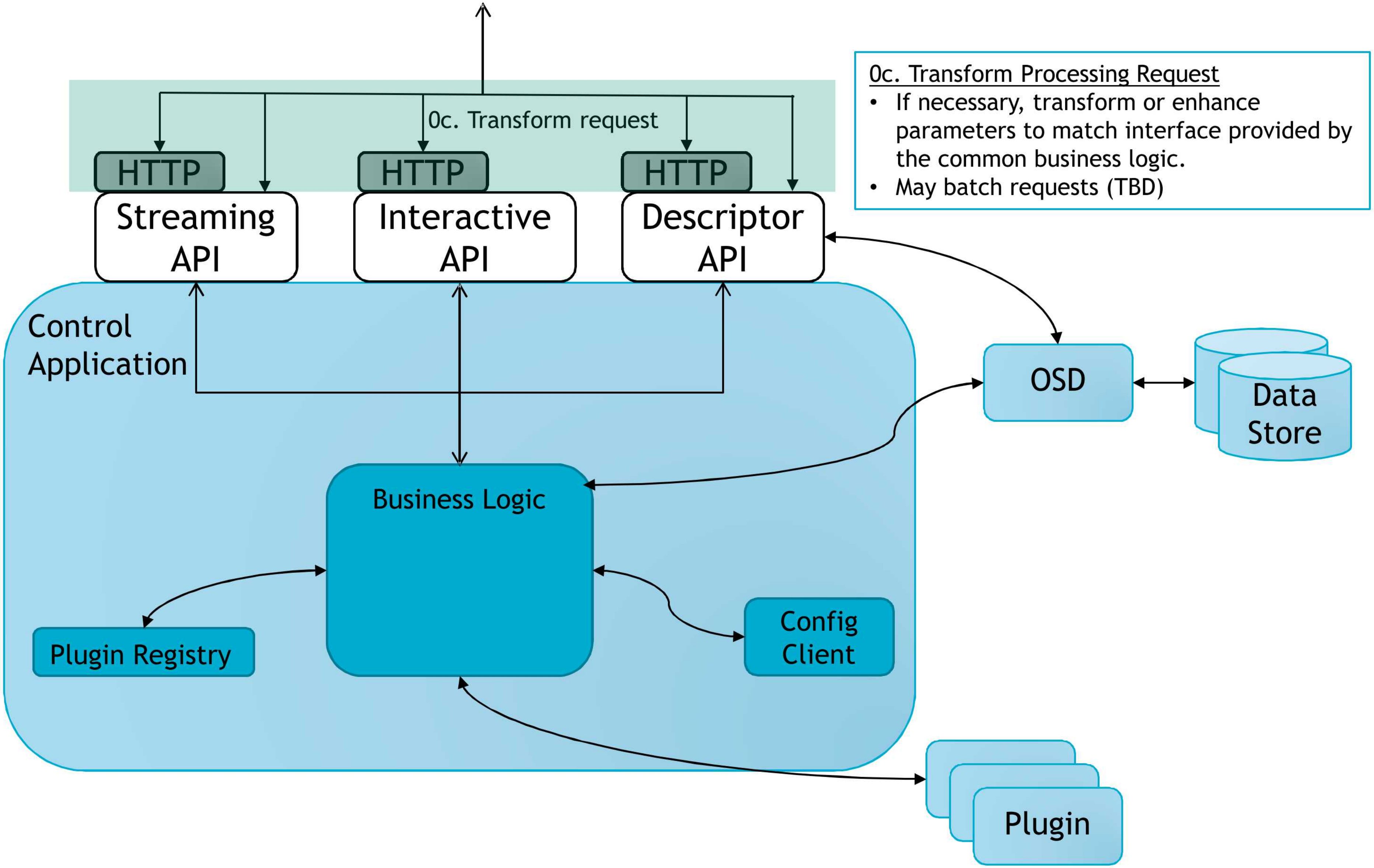
Sequence: Control Serves Processing Request (3/13)



0b. Resolve Descriptors

- Use OSD to query for processing parameter data objects using processing parameter descriptors.

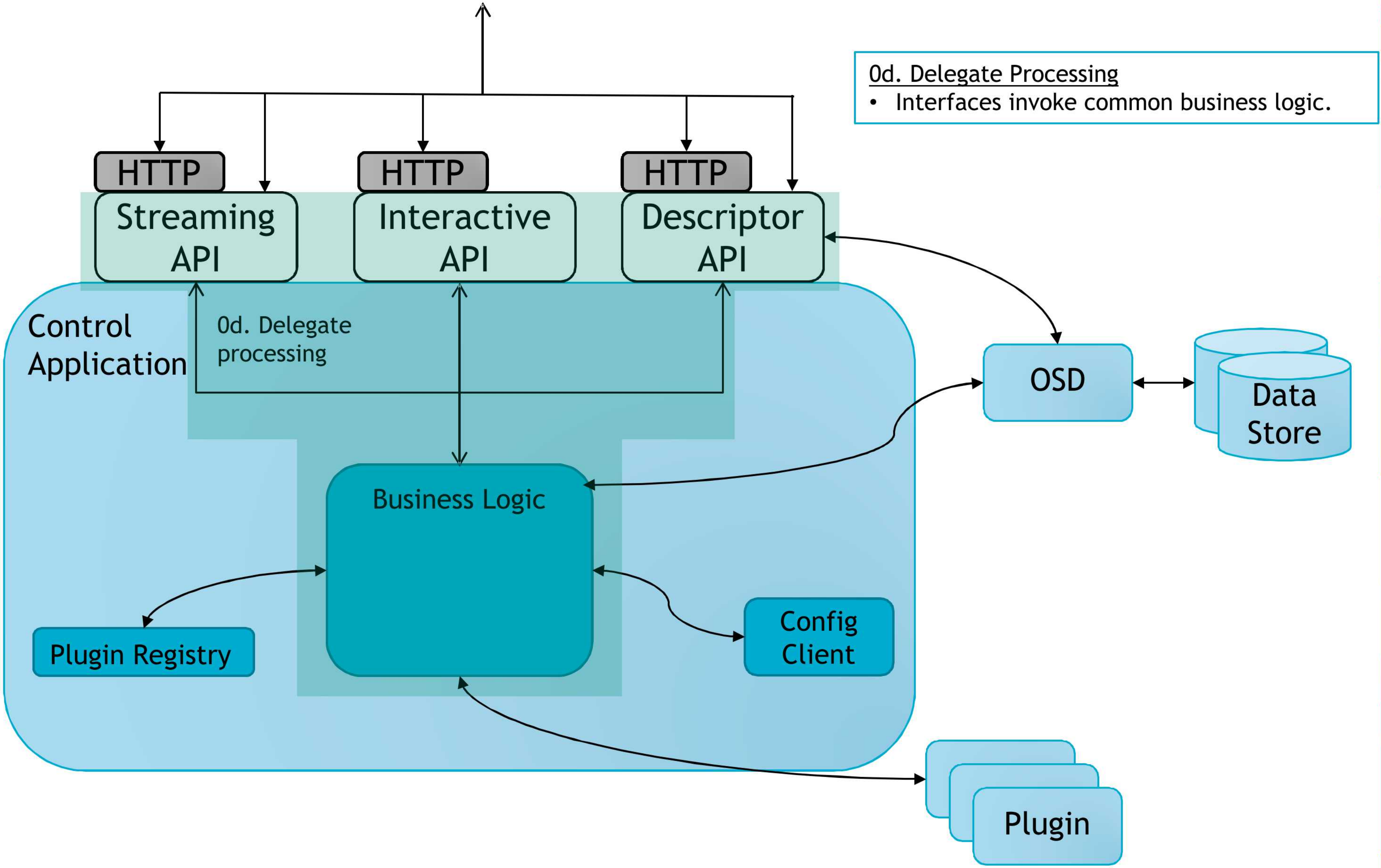
Sequence: Control Serves Processing Request (4/13)



<u>0c. Transform Processing Request</u>

- If necessary, transform or enhance parameters to match interface provided by the common business logic.
- May batch requests (TBD)

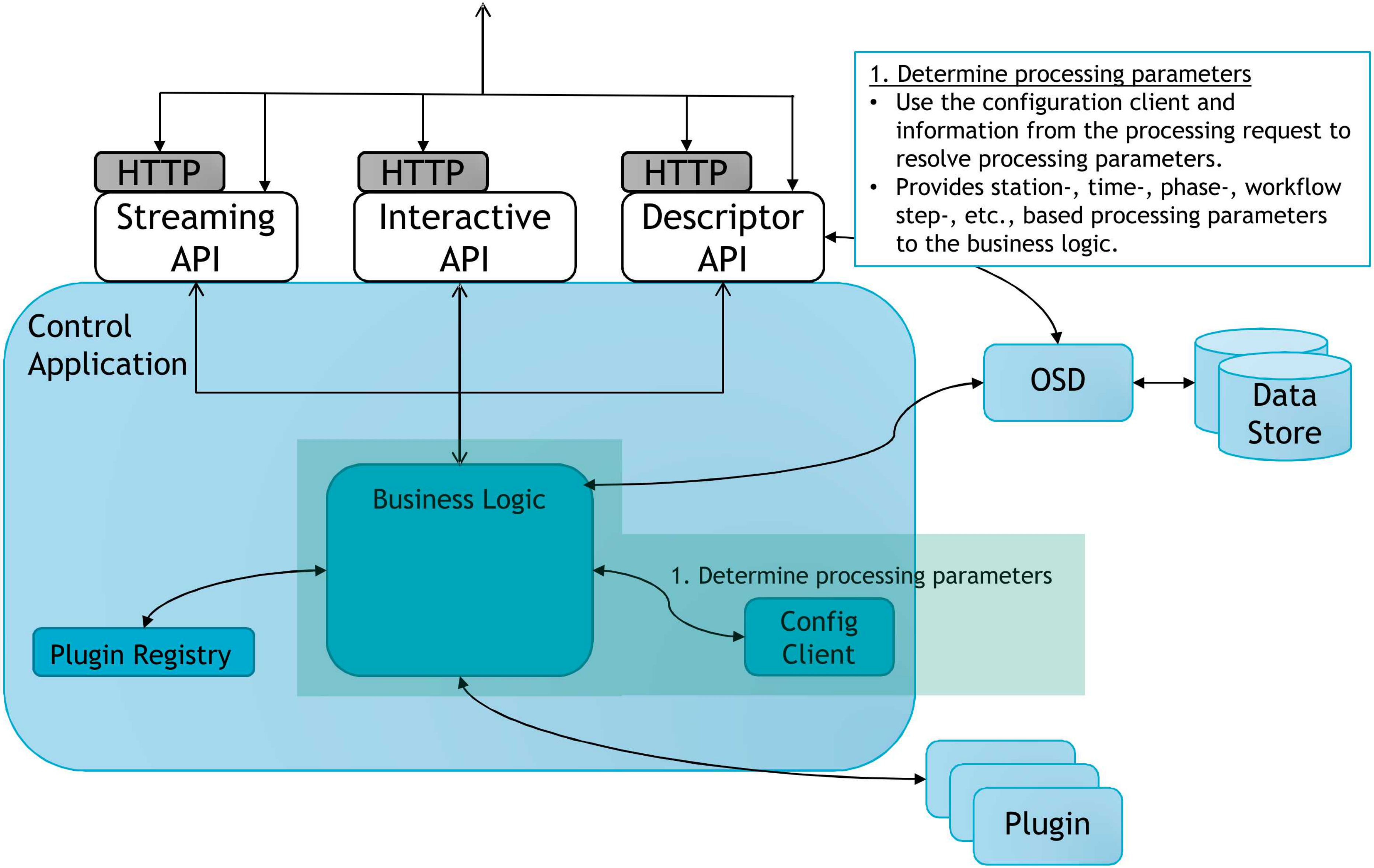
Sequence: Control Serves Processing Request (5/13)



0d. Delegate Processing

- Interfaces invoke common business logic.

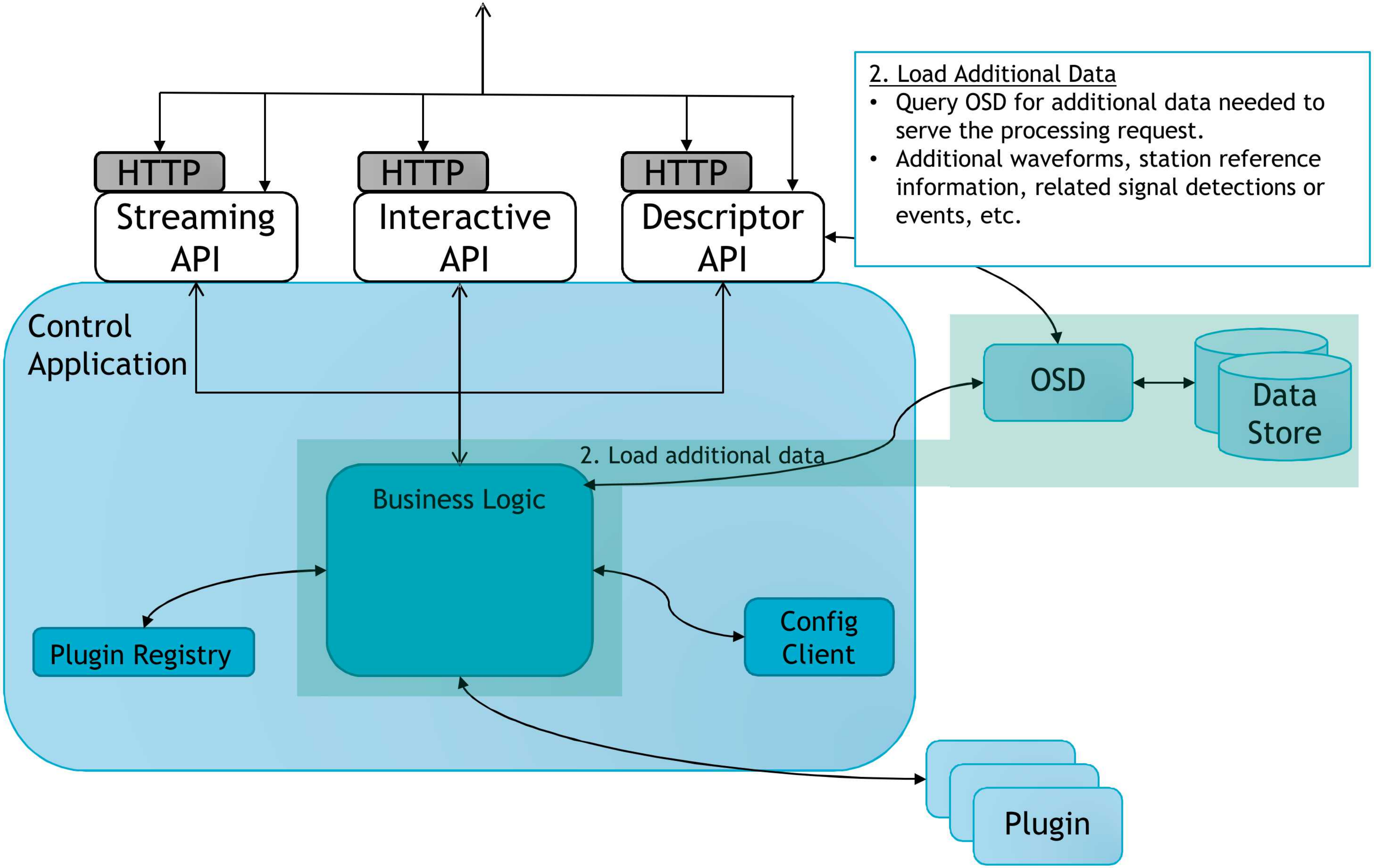
Sequence: Control Serves Processing Request (6/13)



1. Determine processing parameters

- Use the configuration client and information from the processing request to resolve processing parameters.
- Provides station-, time-, phase-, workflow step-, etc., based processing parameters to the business logic.

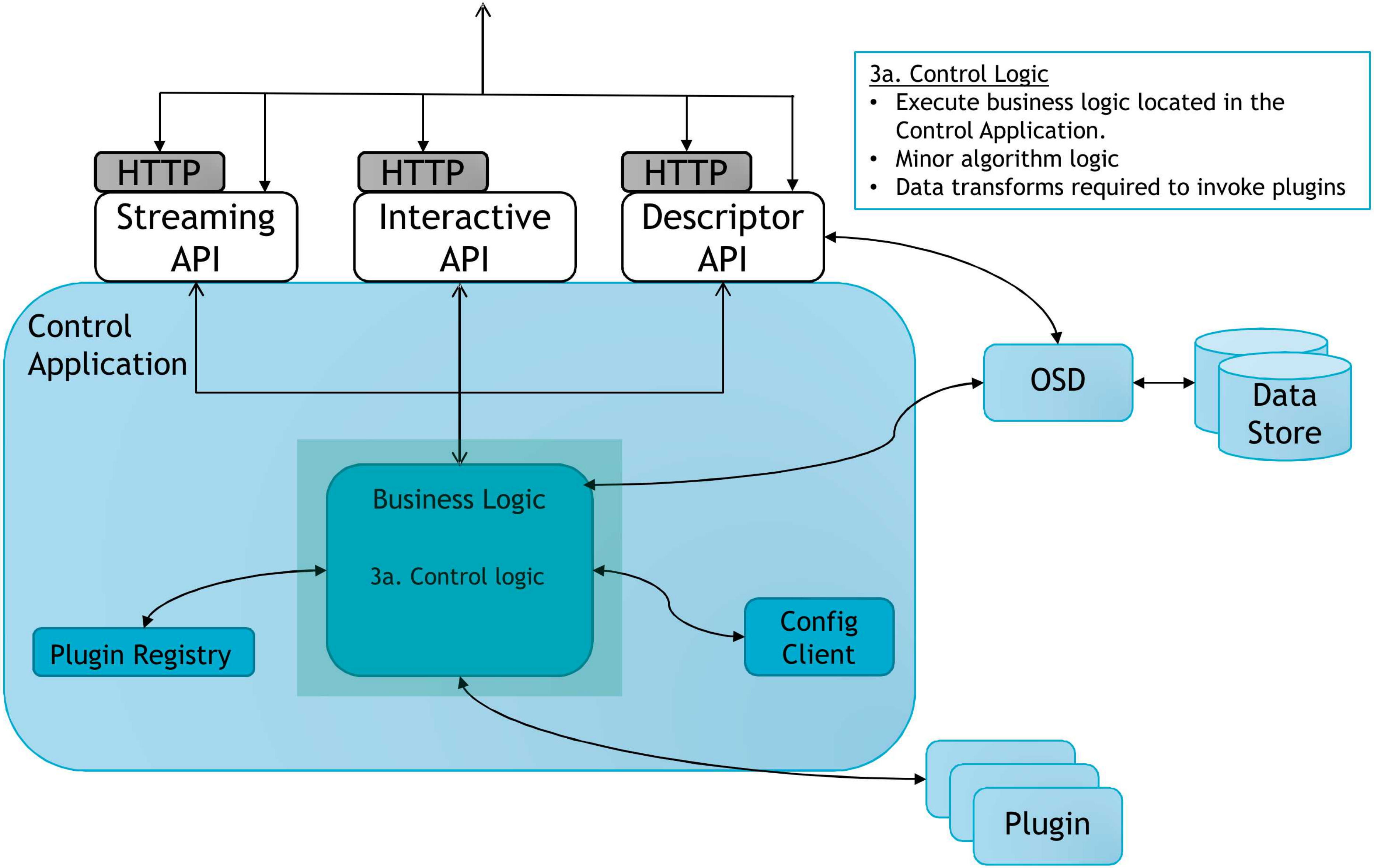
Sequence: Control Serves Processing Request (7/13)



2. Load Additional Data

- Query OSD for additional data needed to serve the processing request.
- Additional waveforms, station reference information, related signal detections or events, etc.

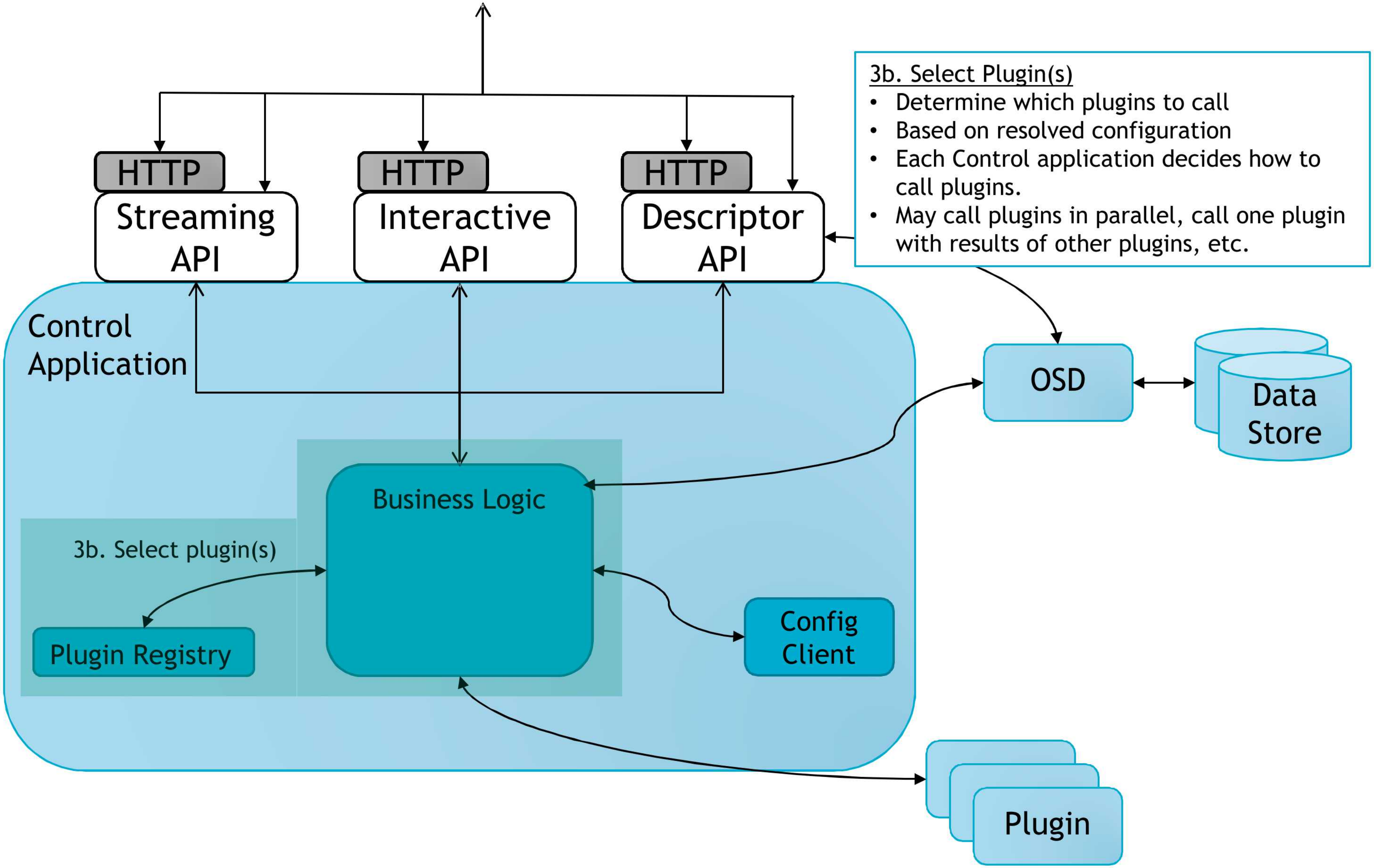
Sequence: Control Serves Processing Request (8/13)



3a. Control Logic

- Execute business logic located in the Control Application.
- Minor algorithm logic
- Data transforms required to invoke plugins

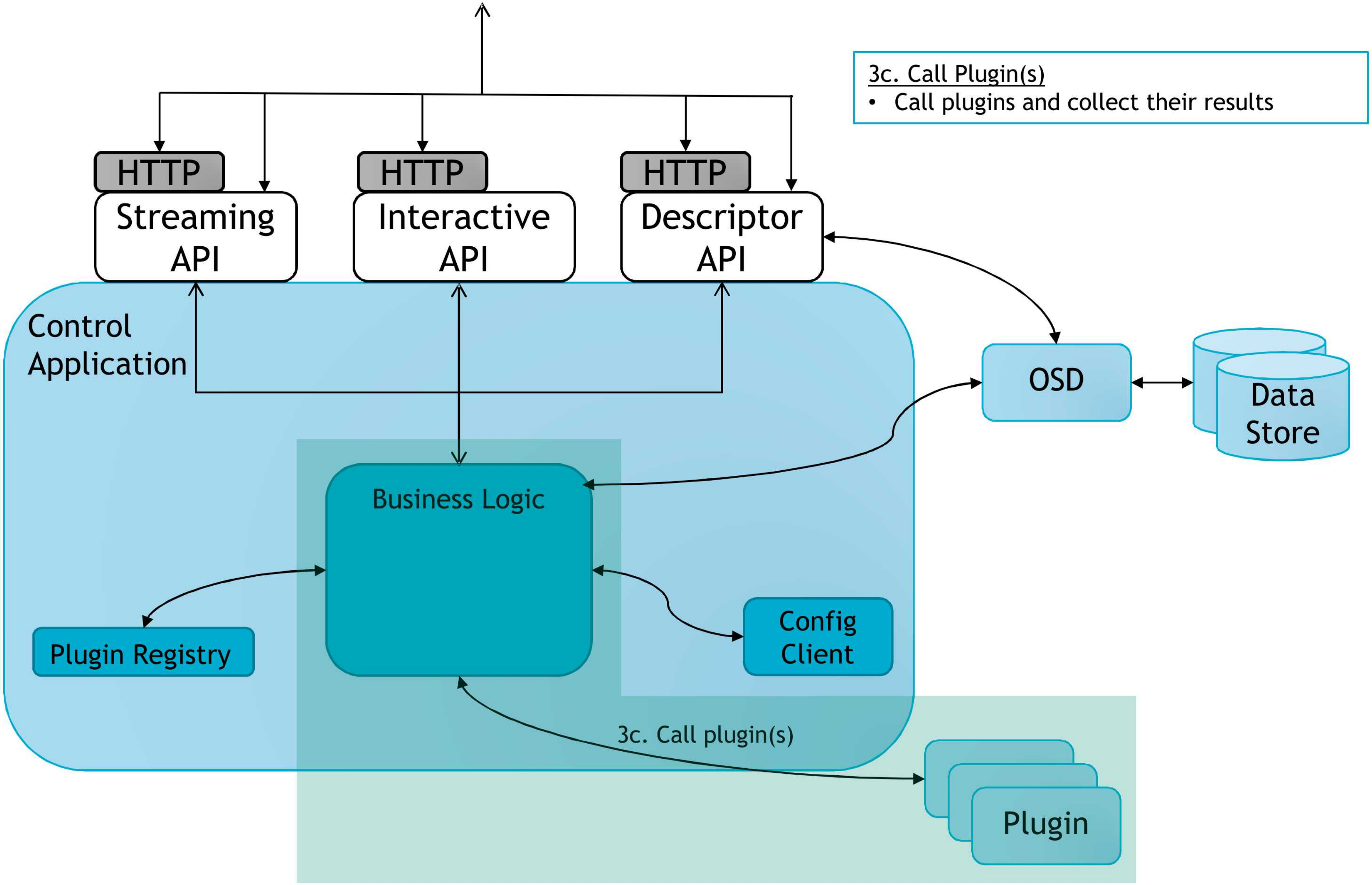
Sequence: Control Serves Processing Request (9/13)



3b. Select Plugin(s)

- Determine which plugins to call
- Based on resolved configuration
- Each Control application decides how to call plugins.
- May call plugins in parallel, call one plugin with results of other plugins, etc.

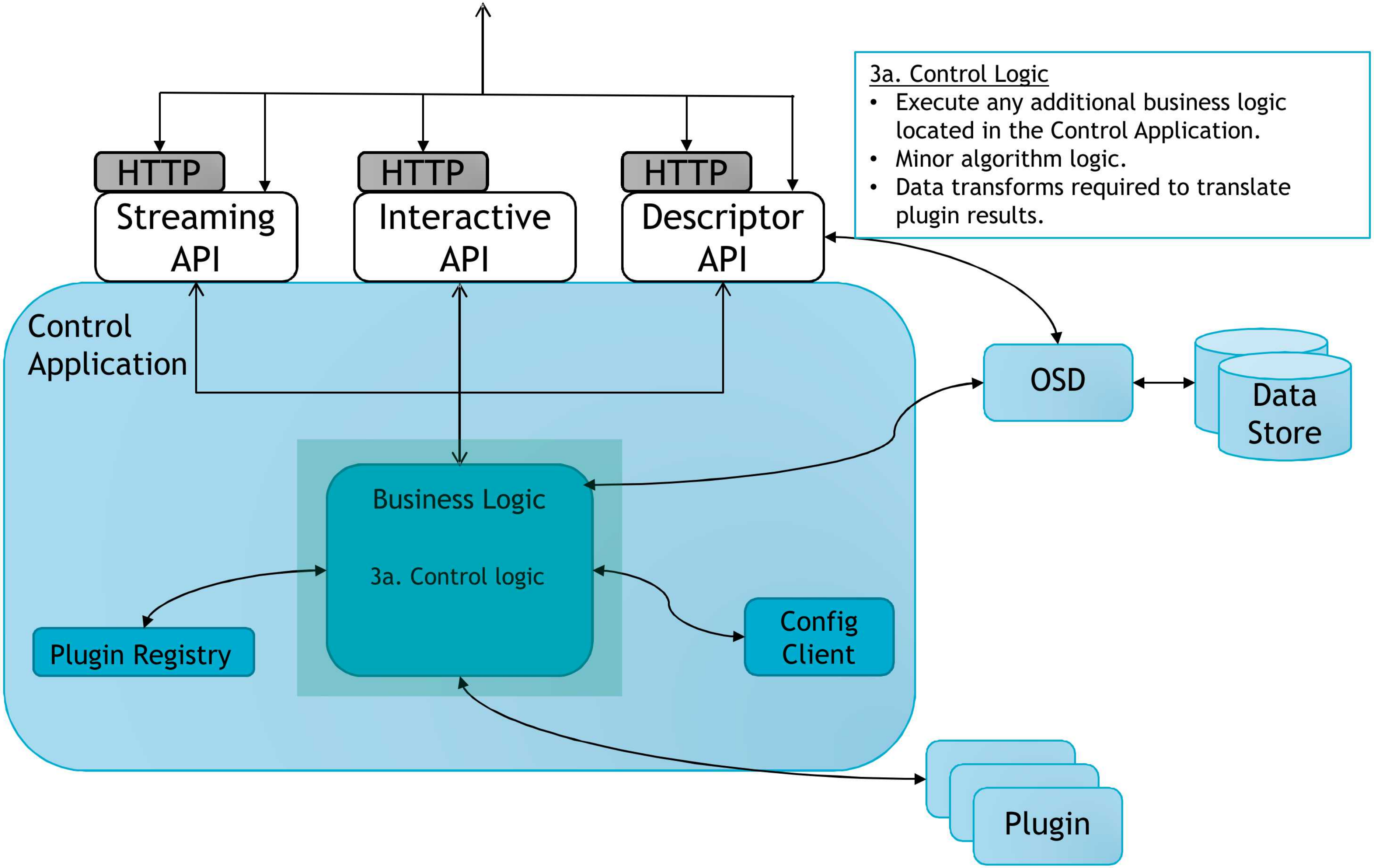
Sequence: Control Serves Processing Request (10/13)



3c. Call Plugin(s)

- Call plugins and collect their results

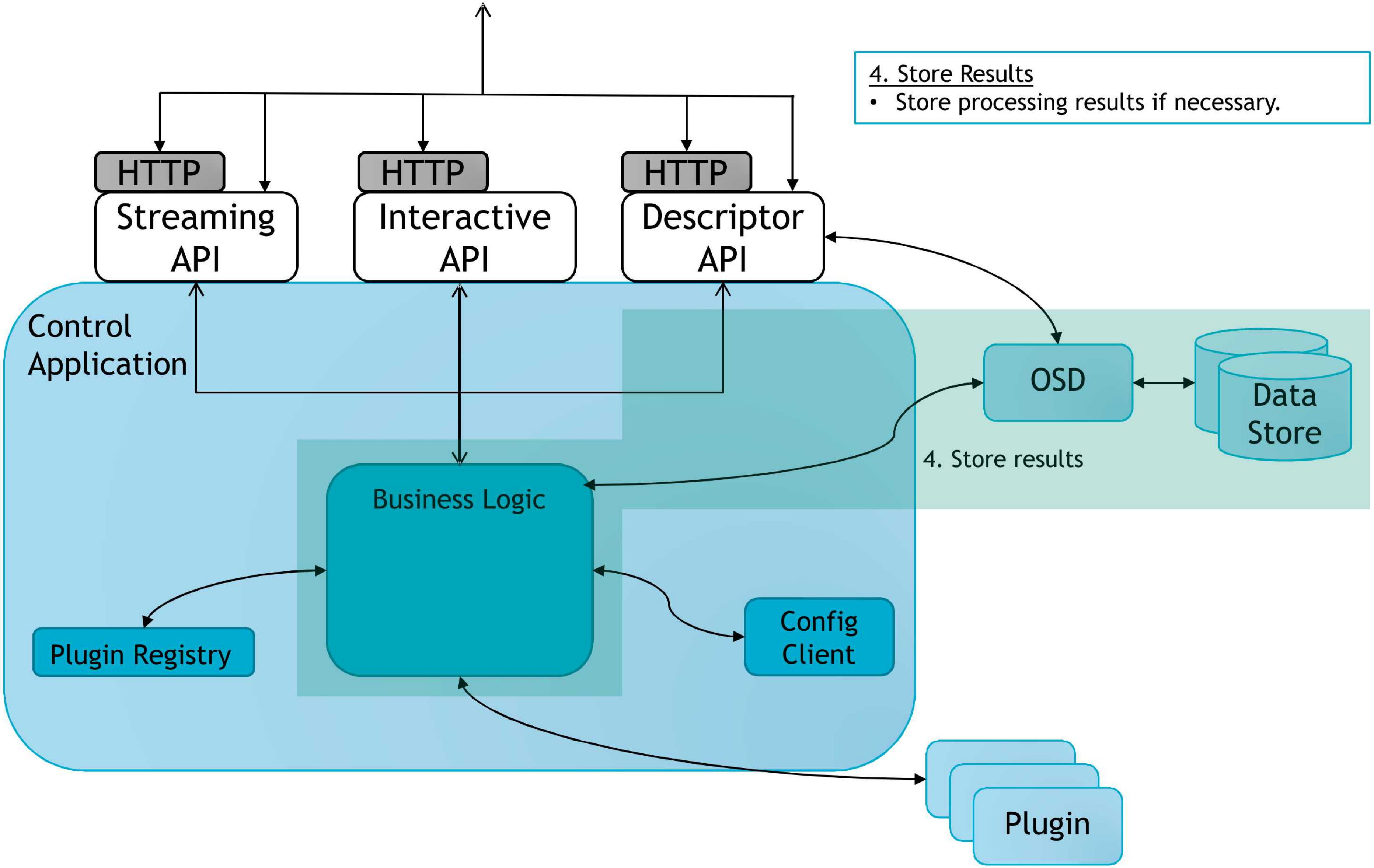
Sequence: Control Serves Processing Request (11/13)



3a. Control Logic

- Execute any additional business logic located in the Control Application.
- Minor algorithm logic.
- Data transforms required to translate plugin results.

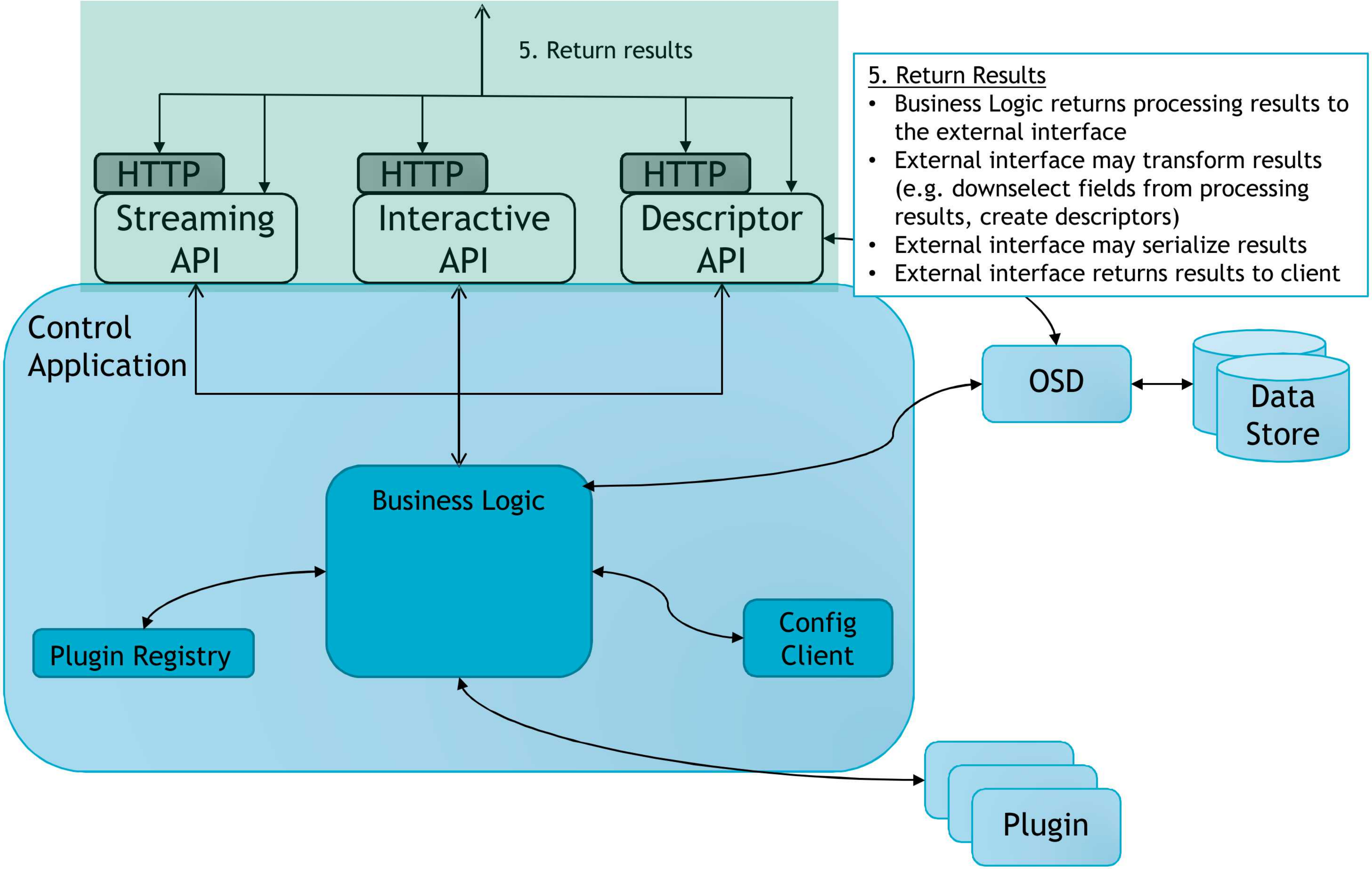
Sequence: Control Serves Processing Request (12/13)



4. Store Results

- Store processing results if necessary.

Sequence: Control Serves Processing Request (13/13)

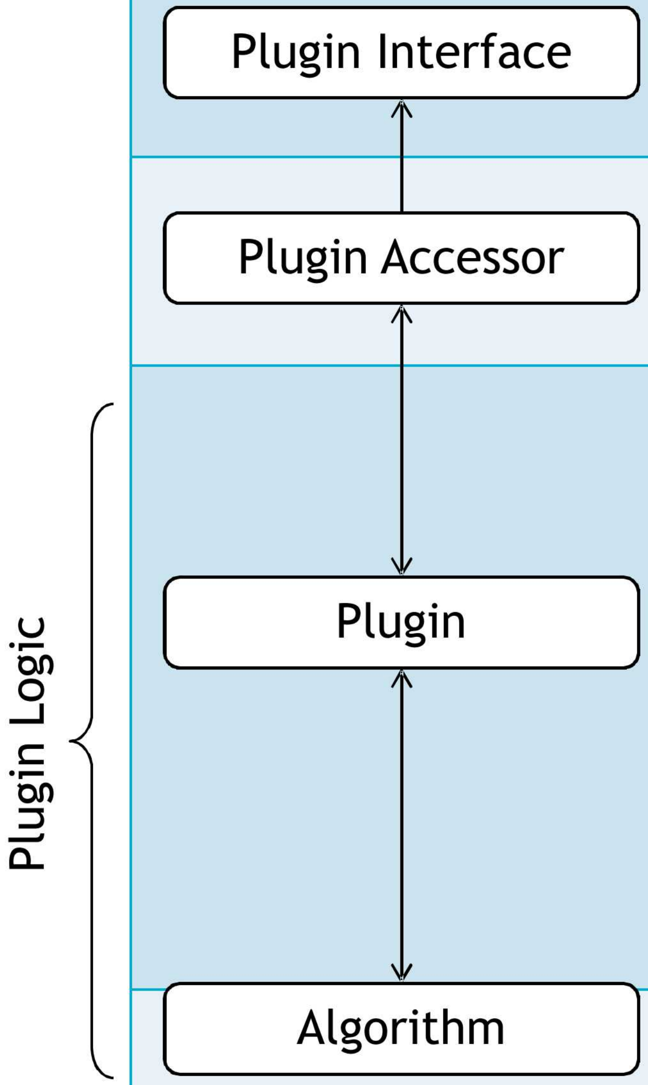


5. Return Results

- Business Logic returns processing results to the external interface
- External interface may transform results (e.g. downselect fields from processing results, create descriptors)
- External interface may serialize results
- External interface returns results to client

Plugin Design

Plugin Design



Declares operations for:

1. General plugin information (name and version)
2. Plugin initialization
3. Processing operation(s) for the algorithm family

Provides a realization of the plugin interface by

1. Delegating operation implementations to the plugin class (for in-memory plugins)
2. Invoking a remote plugin service (for remote plugins)

Prepares data for algorithm

1. Extract necessary fields from COI data objects
2. Transform fields into representations used by the algorithm
   a) Basic signal processing (merge adjacent waveforms, demean, normalize, etc.)
   b) Convert values from absolute times to sample counts
   c) Etc.
3. Algorithm invocation logic

Postprocesses algorithm results

1. Transform fields into COI representations (e.g. convert from sample counts to absolute times, etc.)
2. Associate metadata with algorithm results
3. Construct COI objects (if required by plugin interface)

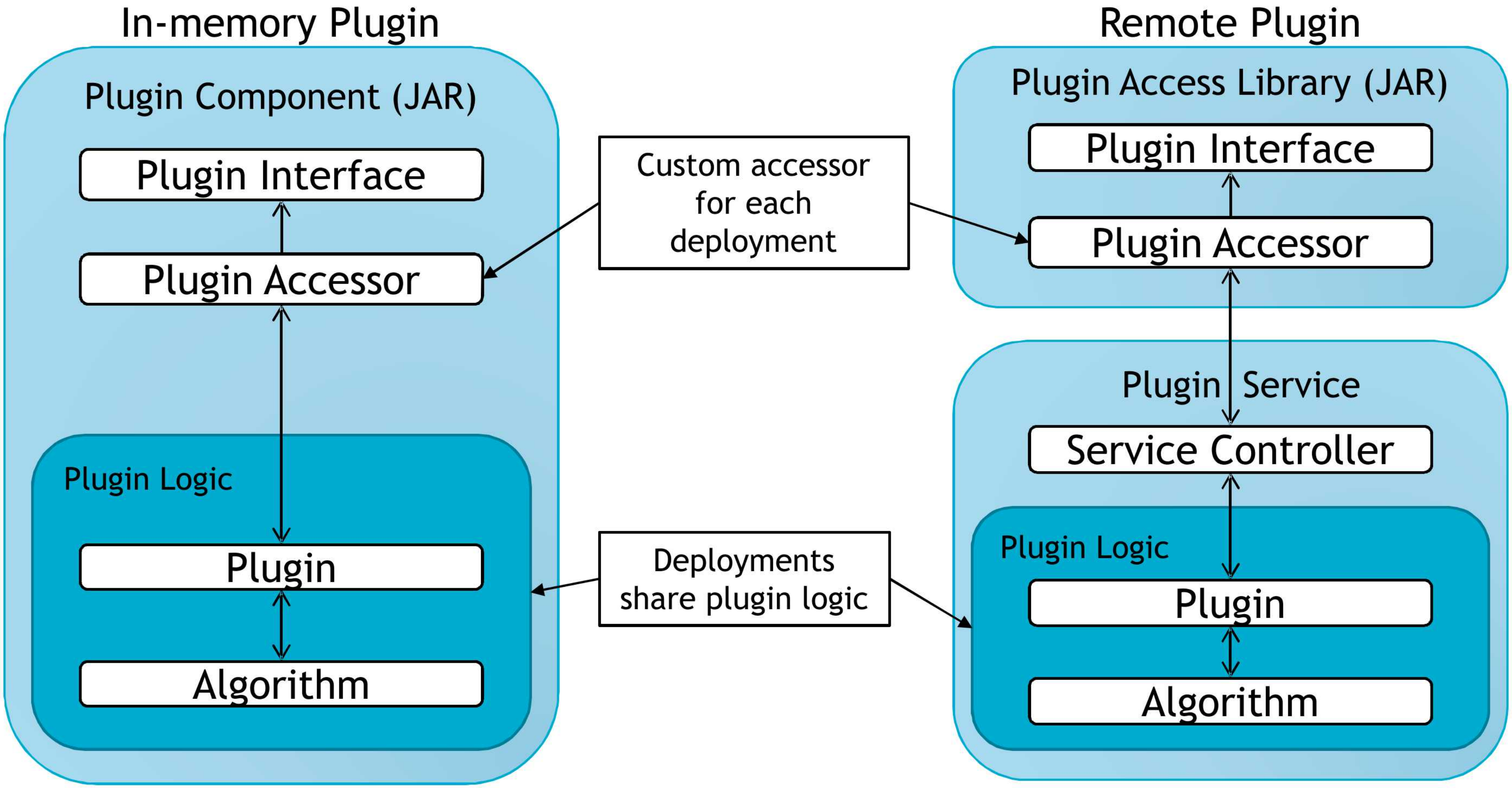
1. Performs scientific calculations
2. Ideally no dependencies on other GMS software (frameworks, libraries, COI classes, etc.)

Plugin Deployment

GMS has two primary plugin deployment schemes

- All current plugins are in-memory libraries
- Designed to support plugins deployed as services

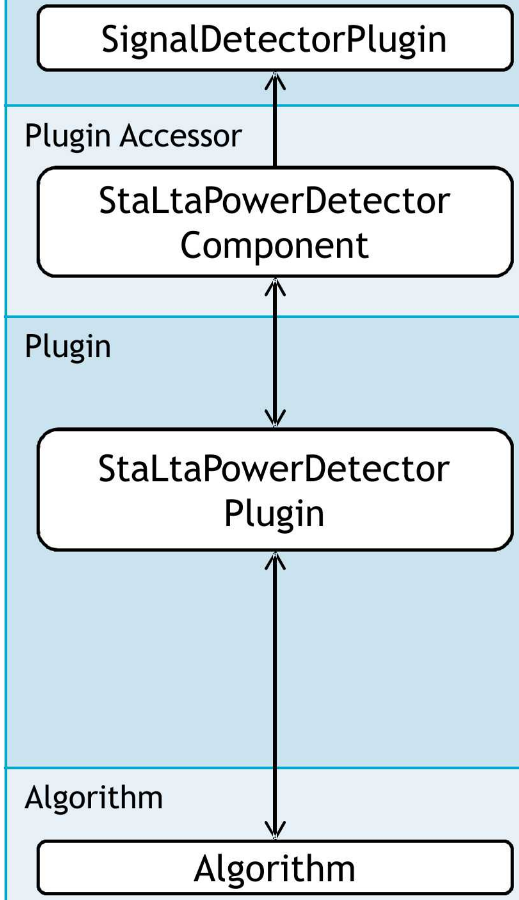
Packaging the same plugin logic in both schemes requires implementing a Plugin Accessor for each deployment.



Example Plugin Implementation – Signal Detector STA/LTA Plugin

In-memory plugin

See /gms/core/signal-detection in GMS software release

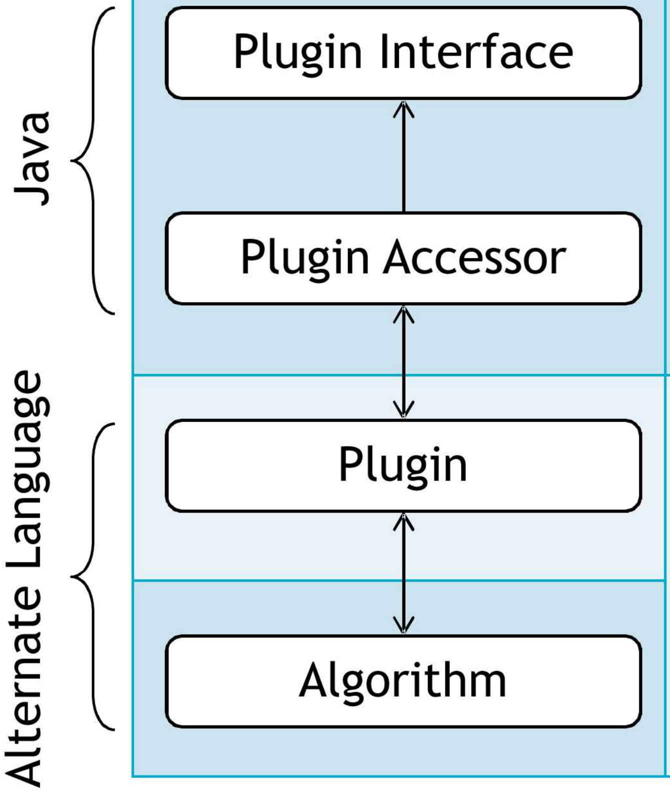


Plugin Interface SignalDetectorPlugin	1. getName and getVersion 2. initialize(configuration) 3. detectSignals(waveform[]) : Instant[]
Plugin Accessor StaLtaPowerDetector Component	1. Implements getName and getVersion 2. Initialize() and detectSignals() implementations delegate to StaLtaPowerDetectorPlugin
Plugin StaLtaPowerDetector Plugin	detectSignals implementation: a) Condition waveforms i. Interpolate over gaps ii. Merge adjacent waveforms iii. Convert STA/LTA window parameters from time units to sample counts iv. Extract double[] from waveforms b) Invoke STA/LTA algorithm c) Convert algorithm results from triggered sample indices to absolute times
Algorithm Algorithm	1. Implements STA/LTA transform and trigger on a double[] 2. Returns triggers as sample indices

Alternate Language Plugins – Notional (1/2)

GMS plugin architecture supports implementing plugins in non-Java languages

Integrating the plugin requires minimal Java (a Plugin Accessor)



Java

Plugin Interface

Plugin Accessor

Alternate Language

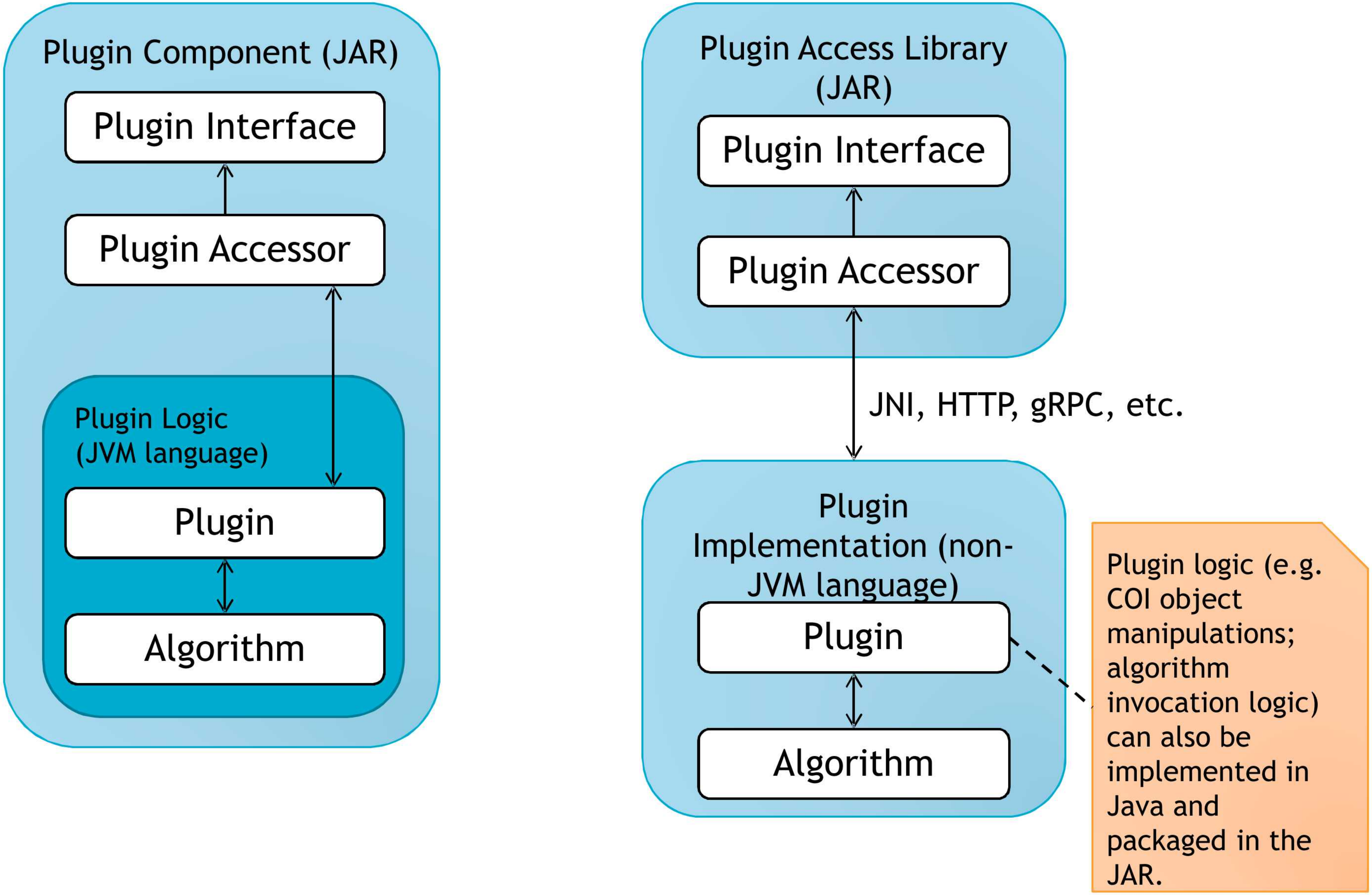
Plugin

Algorithm

1. Use Java to implement Plugin Accessor to call a Plugin implementation in an alternate language

1. Consume and produce COI data objects
   a) Use a COI implementation in the plugin language
   b) Deserialize necessary fields into plugin specific data objects
2. Implement Plugin pre- and post-processing logic
3. Implement plugin Algorithm

Alternate Language Plugins – Notional (2/2)



Plugin Component (JAR)

Plugin Interface

Plugin Accessor

Plugin Logic (JVM language)

Plugin

Algorithm

Plugin Access Library (JAR)

Plugin Interface

Plugin Accessor

JNI, HTTP, gRPC, etc.

Plugin Implementation (non-JVM language)

Plugin

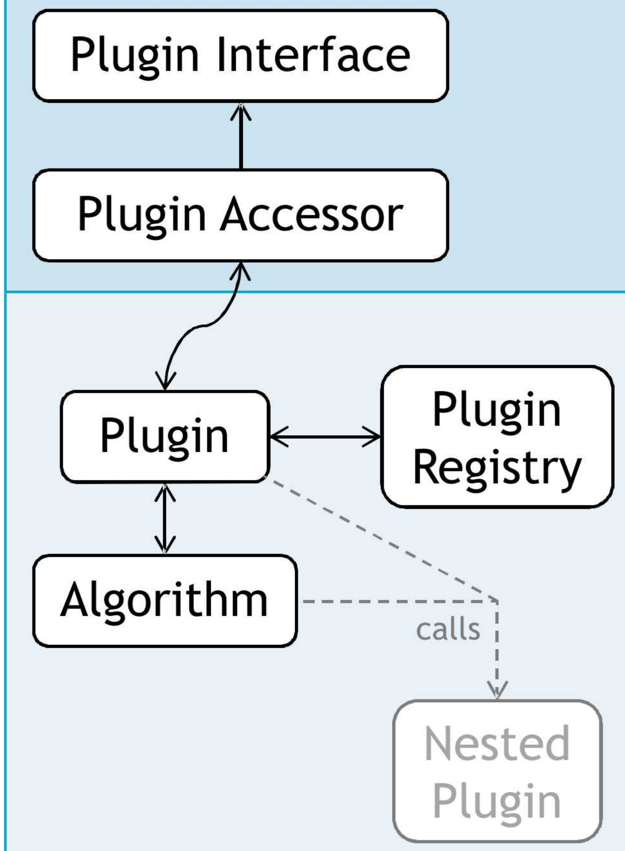
Algorithm

Plugin logic (e.g. COI object manipulations; algorithm invocation logic) can also be implemented in Java and packaged in the JAR.

Complication: Plugins calling Plugins

A GMS plugin may call other GMS plugins, e.g.

- Locator plugin calling Feature Prediction plugin
- Feature Prediction plugin calling Earth Model plugin



Structure

- Plugin contains a Plugin Registry
  - Discovers nested plugins
  - Registers nested plugins
  - Selects nested plugins to use during each processing request

Calling Nested Plugins

- Prefer for Plugin to make calls to nested plugins to isolate Algorithm from GMS libraries
- Algorithm calls nested plugins if necessary (performance; avoid complicated interactions between Plugin and Algorithm, etc.)