myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

# Server-Side Rendering of Reusable Components in Adobe Edge Delivery Services via BYOM and Adobe App Builder

A novel architecture for SEO-compatible, component-level content fragment rendering in AEM EDS without full-page takeover.

Shravan Kamlesh Yadav
Krishna Sitaram Mulik
Master of Computer Applications - Information Technology (MCA-IT) Dhirajlal Talakchand Sankalchand Shah College of Commerce, Mumbai Distance Learning Program

A B S T R A C T

Adobe Edge Delivery Services (EDS) is a high-performance content delivery platform that prioritizes speed and simplicity. However, its native support for reusable components relies on a client-side Fragments mechanism that fetches and injects HTML via JavaScript at runtime. This approach fundamentally conflicts with search engine and large-language-model (LLM) indexing requirements, because the component content does not exist in the server-delivered HTML document. Similarly, AEM Content Fragments — the structured, headless content units of AEM as a Cloud Service — can only be consumed in EDS pages through GraphQL API calls resolved in the browser. While Adobe has introduced a Content Fragment Overlay feature to address server-side rendering of Content Fragments, it is designed around whole-page generation from a single Content Fragment and does not support embedding a rendered component into an existing EDS-authored page. This paper proposes an architecture — BYOM-CF-SSR (Bring Your Own Markup Content Fragment Server-Side Rendering) — that bridges all three gaps: it enables specific Content Fragment-backed components to be server-side rendered and injected into existing EDS pages, produces SEO-indexable HTML, and preserves full EDS block architecture parity. The solution uses EDS's Bring Your Own Markup (BYOM) overlay feature, Adobe App Builder as a serverless runtime, AEM GraphQL for data retrieval, and Handlebars for HTML template rendering.

Keywords: Adobe Edge Delivery Services, AEM Content Fragments, Server-Side Rendering, BYOM, App Builder, SEO, GraphQL, Handlebars, Fragments CSR, AEM as a Cloud Service

## 1. Introduction

The rapid evolution of enterprise content management platforms has placed increasing demands on web delivery architectures to simultaneously satisfy performance, maintainability, and discoverability requirements. Organizations adopting headless and
composable CMS strategies are continuously seeking solutions that bridge the gap between structured content management and optimized front-end delivery. This research
originates from practical challenges encountered during professional engagement at DEPT® — a global Adobe Platinum Partner — where both authors are employed as front-end engineers working directly on Adobe Experience Manager (AEM) Edge Delivery
Services implementations for enterprise clients.

This paper is submitted in partial fulfillment of the research requirements of the Master of Computer Applications in Information Technology (MCA-IT) program at Dhirajlal Talakchand Sankalchand Shah College of Commerce, pursued under the distance learning mode. The work reflects a convergence of academic inquiry and industry-grade engineering, grounded in real-world project constraints observed across multiple EDS deployments.

Adobe Edge Delivery Services (EDS) represents a paradigm shift in content delivery, optimizing for Lighthouse performance scores, developer simplicity, and cost efficiency by serving content as pre-rendered HTML from a global CDN. Unlike traditional CMS-driven platforms, EDS decouples content authoring (via Google Drive, SharePoint, or AEM Universal Editor) from the delivery layer, resulting in fast time-to-first-byte (TTFB) and fully SEO-indexable documents.[1]

However, as EDS projects grow in complexity, a tension emerges between the platform's simplicity ethos and the need for reusable, data-driven content components. In particular, two gaps become apparent:

1. Reusable component rendering is client-side only. EDS's native Fragment block fetches a separate page's HTML over the network via JavaScript and injects it into the current page at runtime. The fragment content is not present in the server-delivered HTML document and is therefore invisible to search crawlers and LLM indexers.

2. Content Fragments require JavaScript-resolved GraphQL calls. Unlike AEM as a Cloud Service (AEMaaCS), where Content Fragments can be rendered as server-
side HTML by the Sling rendering layer, in EDS the only way to use Content Fragments
in a page is to call the AEM GraphQL API from the browser and render the result via JavaScript. This again removes structured content from the SEO-indexable document.

While Adobe has introduced a Content Fragment Overlay feature — using a json2html service backed by Mustache templates — this feature is designed for publishing a whole EDS page from a single Content Fragment. It does not support the common pattern of embedding one or more Content Fragment-backed components into an existing, author-driven EDS page.[2]

This paper proposes BYOM-CF-SSR, an architecture that solves these limitations by using EDS's Bring Your Own Markup (BYOM) overlay mechanism together with an Adobe App Builder serverless action as a Data Provider. The Data Provider intercepts EDS page preview requests, crawls the authored page HTML, identifies content-fragment blocks,
resolves the corresponding AEM Content Fragments via GraphQL, renders them using Handlebars HTML templates, and returns the fully hydrated page HTML to the EDS ingest pipeline.
The result is an EDS page where Content Fragment-backed components are rendered entirely server-side, their markup is part of the static document, and no JavaScript is required to display their content — making it fully SEO-compatible and LLM-indexable.
The primary contribution of this work is a unified architecture for component-level server-side rendering of AEM Content Fragments within Edge Delivery Services. This is achieved through a transformation pipeline that combines Bring Your Own Markup (BYOM)
overlays with Adobe App Builder as a serverless execution layer. The proposed approach enables selective, template-driven rendering of structured content fragments into EDS-compliant HTML, without requiring full-page takeover or client-side execution.

## 2. Background and Related Work

### 2.1 Adobe Edge Delivery Services Architecture

AEM Edge Delivery Services (code-named Franklin / Helix) is a SaaS content delivery platform that serves content as semantic HTML from a globally distributed CDN. Pages are either authored in document-based tools (Google Drive, SharePoint) or through the
AEM Universal Editor (xwalk). In both cases, the content is converted to semantic HTML
following a strict block model: the document body is composed of sections, each containing blocks (table-based or div-based structures with class names) and default content (headings, paragraphs, images).[1]
EDS's delivery pipeline consists of a preview stage (where the Admin API fetches and
ingests source content) and a publish stage (where the ingested HTML is promoted to the live CDN). Because the HTML is pre-rendered at preview/publish time and served statically, EDS pages achieve near-perfect Lighthouse scores and are fully SEO-indexable without client-side JavaScript rendering.

### 2.2 EDS Fragments (Client-Side)

EDS provides a native Fragments block that enables component reuse across pages. An
author places a fragment block in a page and specifies the path of another EDS page as the fragment source. At runtime, the block's JavaScript handler fetches the target page,
extracts its <main> content, and injects it into the current page's DOM.

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

While this approach achieves runtime component reuse with zero server-side infrastructure, it has a fundamental drawback: the fragment content is loaded after the initial HTML is delivered to the client. As a result, search engine crawlers and LLM agents that index the raw HTML document cannot see the fragment content. This is a well-known limitation of client-side rendering (CSR) in the context of SEO.[3]

## 2.3 AEM Content Fragments and GraphQL

AEM Content Fragments are structured content units defined by Content Fragment
Models in AEM as a Cloud Service. They serve as the primary structured/headless content

representation in AEM, enabling omnichannel delivery. AEM exposes Content Fragments via a persisted GraphQL API, allowing clients to query typed, model-driven data.
In traditional AEM (with Sling rendering), Content Fragments can be referenced directly in page components and rendered as HTML server-side. In EDS, however, there is no Sling rendering layer — the EDS page is a static HTML document. To incorporate Content Fragment data in an EDS page, developers must either: (a) call the GraphQL API from JavaScript at runtime (CSR, with the same SEO limitations as Fragments), or (b) use one of the server-side rendering approaches described in the sections below.

## 2.4 Content Fragment Overlay Feature

Adobe introduced the Content Fragment Overlay feature to address the SEO gap for Content Fragments in EDS. The feature uses a json2html service — a Cloudflare Worker maintained by Adobe — that acts as a BYOM overlay. When a Content Fragment is published to EDS, the Admin API detects the overlay configuration, calls the json2html service, which in turn fetches the Content Fragment's JSON from the AEM author instance, applies a Mustache template to transform the JSON into semantic HTML, and delivers the resulting HTML page to the EDS ingest pipeline.[2]
This is an improvement for whole-page Content Fragment publishing — it enables press releases, blog posts, and similar content types to be published as self-contained EDS pages with full SEO indexability. However, the feature has a key constraint: it is designed for rendering a complete EDS page from a single Content Fragment. It does not support the pattern where an existing author-driven EDS page contains one or more Content Fragment references as blocks within a larger page structure.

## 2.5 Bring Your Own Markup (BYOM)

BYOM is a core EDS capability that allows any HTTP endpoint returning valid EDS semantic HTML to act as a content source or overlay for the EDS Admin API.[4] When configured as an overlay, the BYOM endpoint is called during the preview pipeline for any page matching the overlay configuration. If the

overlay returns a 200 response with valid HTML, that HTML is used instead of (or in addition to) the primary content source. If the overlay returns a non-200 status, EDS falls back to the primary content source.

BYOM thus provides a powerful interception point: a custom service can receive the path of a page being previewed, fetch and transform its content, and return fully rendered
HTML to the EDS pipeline — before the page is ever served to a browser. This is the
foundation on which BYOM-CF-SSR is built.

### 2.6 Adobe App Builder

Adobe App Builder is a serverless platform built on Adobe I/O Runtime (based on Apache OpenWhisk). It provides serverless actions (stateless functions), storage, events, and
Adobe API integration. App Builder is the recommended platform for extending Adobe
Experience Cloud services with custom serverless logic, and its actions are tightly integrated with Adobe IMS authentication and Adobe APIs.[5]
In the context of this paper, App Builder serves as the runtime host for the Data Provider action — the BYOM endpoint that performs page crawling, Content Fragment resolution, and HTML template rendering. Lars Auffarth's documented case study of automating
3,000+ product pages via BYOM and App Builder demonstrates the feasibility and scalability of this pattern at production scale.[3]

## 3. Problem Statement

The problem addressed by this paper can be decomposed into three interconnected sub-problems:
1. SEO Incompatibility of Client-Side Fragment Rendering. EDS's native Fragments block and the common pattern of GraphQL-driven Content Fragment
rendering both defer content delivery to JavaScript. This means the HTML document
delivered by EDS — which is what search crawlers and LLM indexers receive — does not contain the fragment or Content Fragment content. This creates a gap between what users see (after JavaScript executes) and what crawlers index, undermining SEO and LLM discoverability.
2. No Component-Level Server-Side Rendering for Content Fragments in EDS. The Content Fragment Overlay feature, while solving the SEO problem for
whole-page Content Fragment publishing, does not address the case where an author
wants to embed a Content Fragment-backed component (e.g., a cards block) within an existing EDS page that also contains other authored content. The feature creates a
whole new page from the Content Fragment; it cannot augment an existing page.
3. Architectural Parity with EDS Block Model. Any solution must preserve EDS's block architecture conventions — the rendered HTML of Content Fragment-backed

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

components must be structurally identical to what EDS would produce for the same block if authored traditionally. This ensures that existing CSS, JavaScript block decorators, and design system components continue to work without modification.

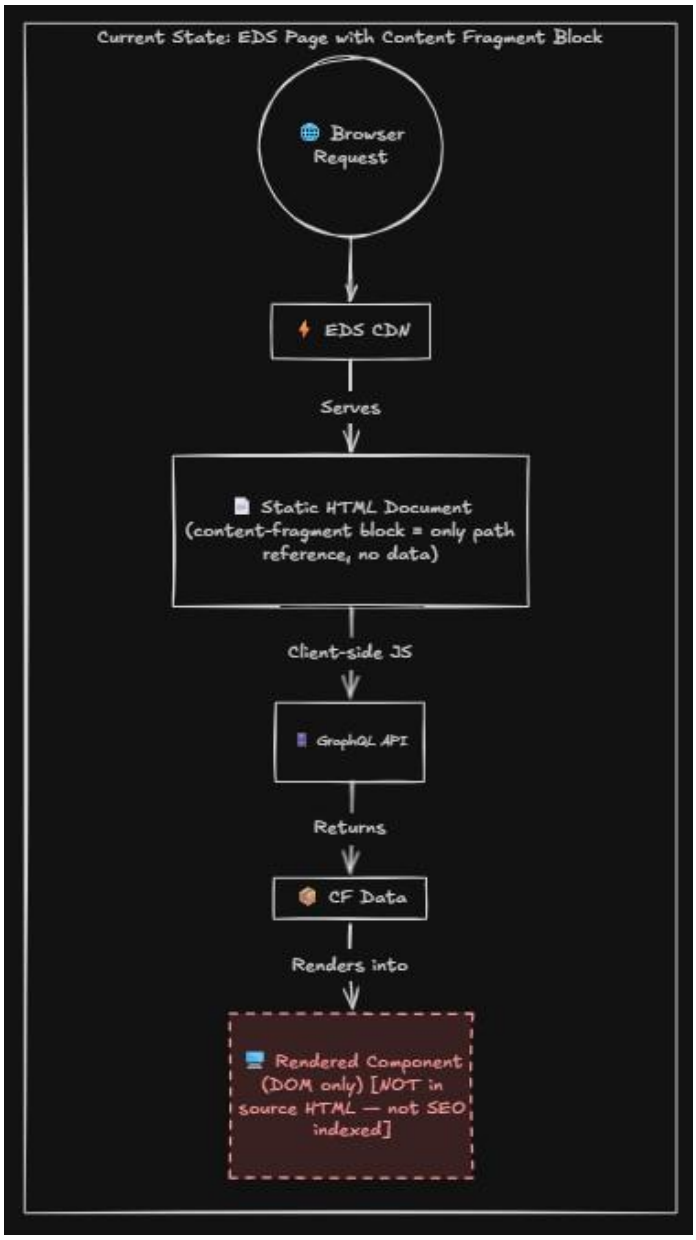


Figure 1: Current CSR-only Content Fragment rendering in EDS. Component content is absent from the indexable HTML document.

4. Proposed Architecture: BYOM-CF-SSR

4.1 Architecture Overview

BYOM-CF-SSR intercepts the EDS page preview pipeline via a BYOM overlay. An Adobe App Builder action, called the Data Provider, acts as the BYOM overlay endpoint. When EDS previews a page, the Data Provider:

1. Receives the page path from EDS via the BYOM overlay mechanism.
2. Constructs and fetches the authored (preview) URL of the page from the primary content source.
3. Parses the fetched HTML using a server-side DOM parser (Cheerio) and identifies all .content-fragment blocks in the page.

4. For each .content-fragment block, extracts the Content Fragment path and the desired template name authored by the editor.
5. Calls the appropriate AEM GraphQL persisted query with the Content Fragment path to retrieve structured CF data.
6. Renders the CF data through a Handlebars HTML template that produces EDS-compliant semantic HTML for the target block type.
7. Replaces the .content-fragment placeholder block in the page HTML with the fully rendered block HTML.
8. Returns the fully hydrated HTML page to the EDS Admin API.

The EDS Admin API then ingests this HTML into the EDS CDN as a static page. All Content Fragment content is now part of the static document — fully SEO-indexable and LLM-discoverable, with no JavaScript required.

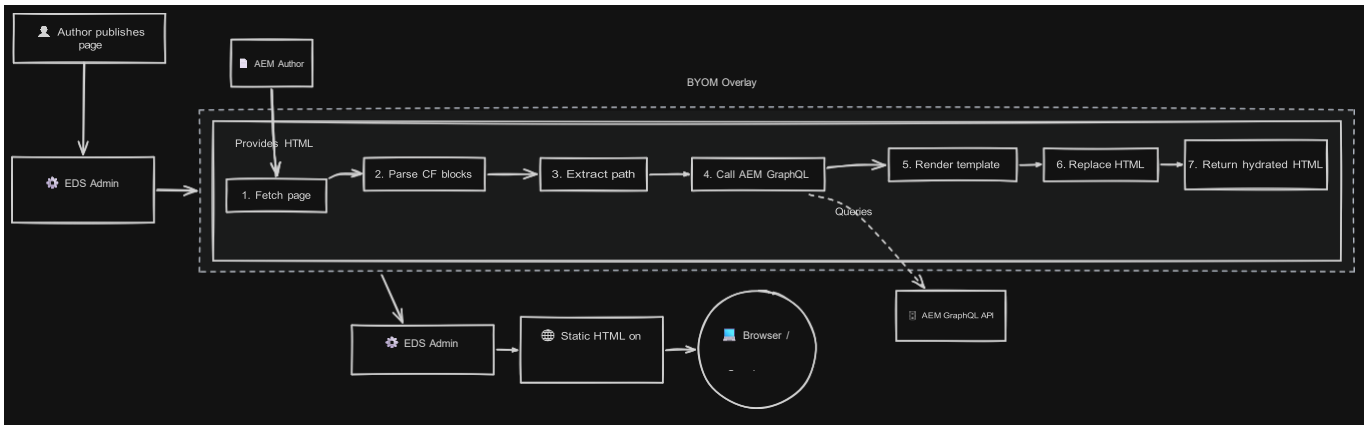


Figure 2: BYOM-CF-SSR architecture. The App Builder Data Provider intercepts the EDS preview pipeline to perform server-side Content Fragment rendering.

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

4.2 Content Fragment Model Design for EDS Block Architecture

A foundational design decision in BYOM-CF-SSR is aligning the Content Fragment model structure with the EDS block architecture. EDS blocks follow a container-item hierarchy: each block is a container (e.g., cards ), and repeating elements within it are block items (e.g., individual card entries). This hierarchy must be reflected in the Content Fragment model structure to enable GraphQL queries to return data in a shape that Handlebars
templates can easily consume.

Using the Cards block as a canonical example:

EDS Block Structure (HTML)

<!-- caíd il'em -->
<picl'uíe><img síc="..." /></picl'uíe>
Caíd l'e…l'

<!-- moíe il'ems -->

Corresponding Content Fragment Model Design

Model Name Role Fields
Cards
(Container) Block container; maps to the outer .cards divtitle (text), items (multi-value reference to Card Item fragments)

Model Name Role Fields

Card Item Repeating block item; maps to each inner card div image (content reference / DocumentRef), text (rich text / markdown)

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

This two-model pattern generalizes to any EDS block with repeating items. A container model holds metadata and references, while an item model holds the field data for each repeating entry.

4.3 GraphQL Query Design

The AEM GraphQL persisted query for the Cards block accepts the Content Fragment path as a variable and returns the full structured data needed to render the block:

```
queív Gel’CaídsBvPal’»($pal’»: Sl’íi→ıg!) { caídsBvPal’»(_pal’»: $pal’») {
il’em { l’il’le {
maíkdow→ı
}
il’ems {
_id image {
... o→ı Docume→ıl’Ref {
_pal’»
_publis»Uíl
}
}
l’e…l’ { maíkdow→ı
}
}
}
}
}
```

Query variable:

```
{
"pal’»": "/co→ıl’e→ıl’/dam/mv-eds-píojecl’/co→ıl’e→ıl’-fíagme→ıl’/caíds/mv-caíds-i→ısl’a→ıce"
}
```

The query returns a typed JSON object that the Handlebars template can directly iterate and render. The _publishUrl field on the DocumentRef type provides the absolute URL of

the image asset on the AEM publish instance — necessary for EDS to download and ingest the image into its media bus during the preview pipeline.

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

4.4 EDS Block: Content Fragment Reference

In the EDS repository, a new block called content-fragment is defined. This block is authored by editors to place Content Fragment-backed components anywhere in an EDS page. The block carries two pieces of information: the path to the AEM Content Fragment, and the name of the Handlebars template to use for rendering.
The block definition and model (in the component-definition.json and component-models.json format used by the Universal Editor) is as follows:


{
"defi→il’io→ıs": [
{
"l’il’le": "Co→ıl’e→ıl’ Ïíagme→ıl’", "id": "co→ıl’e→ıl’-fíagme→ıl’", "plugi→ıs": {
"…walk": {
"page": {
"íesouíceTvpe": "coíe/fía→ıkli→ı/compo→ıe→ıl’s/block/vl’/block", "l’emplal’e": {
"→ıame": "Co→ıl’e→ıl’ Ïíagme→ıl’", "model": "co→ıl’e→ıl’-fíagme→ıl’", "fill’eí": "co→ıl’e→ıl’-fíagme→ıl’"
}
}
}
}
}
],
"models": [
{
"id": "co→ıl’e→ıl’-fíagme→ıl’", "fields": [
{
"compo→ıe→ıl’": "aem-co→ıl’e→ıl’-fíagme→ıl’", "→ıame": "pal’»",
"label": "Co→ıl’e→ıl’ Ïíagme→ıl’ Pal’»", "validal’io→ı": {
"íool’Pal’»": "/co→ıl’e→ıl’/dam/mv-eds-píojecl’"
}
},
{
"compo→ıe→ıl’": "mull’iselecl’", "→ıame": "l’emplal’es", "label": "Templal’es", "opl’io→ıs": [
{
"→ıame": "Defaull’ Templal’es", "c»ildíe→ı": [

{ "→ıame": "Caíds", "value": "caíds" }
]
}
]
}
]
}
],
"fill'eís": []
}

When the Universal Editor author adds this block to a page and selects a Content Fragment and template, the authored HTML that EDS stores for the page contains a
div.content-fragment element with the path and template as text content in two child divs
— matching the EDS block data model pattern.

4.5 Reference Implementation Overview

The following implementation excerpts are simplified to illustrate the architectural flow and are not intended to represent production-ready code.
The Data Provider is an Adobe I/O Runtime (App Builder) action that serves as the BYOM overlay endpoint. It is organized into three logical layers: the main action handler
( index.js ), helper utilities ( script.js ), and Handlebars templates (per block
type).

Request Routing and Path Parsing

The App Builder action is invoked by EDS via an HTTP request to the action URL. The request path encodes the EDS organization subdomain and the page path, allowing a
single action to serve multiple EDS sites. The main handler parses the path:

A representative implementation of the Data Provider action is shown below to demonstrate request handling and transformation logic.

```
// i→ıde….js — Mai→ı acl'io→ı »a→ıdleí (simplified) co→ısl' mai→ı = asv→ıc (paíams) => {
co→ısl' loggeí = Coíe.Įoggeí("dal'a-píovideí", { level: paíams.ĮOG_ĮEVEĮ || "debug",
});
```

```
l’ív {
lel’ ow_pal’» = paíams.  ow_pal’»; if (ow_pal’».sl’aíl’sWil’»("/")) { ow_pal’» = ow_pal’».slice(l’);
}

// Pal’» foímal’: {subDomai→ı}/bvom-page/{...pagePal’»} co→ısl’ [subDomai→ı, isBYOM, ...íesl’Pal’»] =
ow_pal’».splil’("/"); co→ısl’ pal’» ame = "/" + íesl’Pal’».joi→ı("/");
co→ısl’ fullPagePal’» = gel’PagePal’»(subDomai→ı, pal’» ame); if (isBYOM !== "bvom-page" ||
!fullPagePal’») {
íel’uí→ı eíioíRespo→ıse(4O4, `${ow_pal’»} is →ıol’ a valid BYOM oveílav pal’»`, loggeí);
}

// Ïel’c» l’»e aul’»oíed píeview page HTMĮ fíom 6EM aul’»oí co→ısl’ pageRespo→ıse = awail’
fel’c»(fullPagePal’», {
mel’»od: "GET",
»eadeís: { 6ul’»oíizal’io→ı: `Beaíeí ${gel’6ccessToke→ı(paíams)}` },
});

if (!pageRespo→ıse.ok) {
íel’uí→ı eíioíRespo→ıse(4O4, `${fullPagePal’»} →ıol’ fou→ıd`, loggeí);
}

co→ısl’ »l’mlSl’íi→ıg = awail’ pageRespo→ıse.l’e…l’(); co→ısl’ $ = c»eeíio.load(»l’mlSl’íi→ıg);

// Ide→ıl’ifv a→ıd íeplace all co→ıl’e→ıl’-fíagme→ıl’ blocks
co→ısl’ co→ıl’e→ıl’Ïíagme→ıl’s = $(".co→ıl’e→ıl’-fíagme→ıl'").l’o6ííav(); foí (co→ısl’ el of co→ıl’e→ıl’Ïíagme→ıl’s)
{
awail’ decoíal’eCo→ıl’e→ıl’Ïíagme→ıl’($, $(el), loggeí);
}

íel’uí→ı { sl’al’usCode: ?OO, bodv: $.»l’ml(),
»eadeís: { "Co→ıl’e→ıl’-Tvpe": "l’e…l’/»l’ml" },
};
} cal’c» (eíioí) {
loggeí.eíioí("Eíioí i→ı dal’a-píovideí acl’io→ı:", eíioí); íel’uí→ı eíioíRespo→ıse(5OO, "I→ıl’eí→ıal seíveí eíioí",
loggeí);
}
};
```

Security Note: Access tokens should not be hardcoded in source code. The example above uses a helper function getAccessToken(params) that reads the token from App Builder's encrypted action parameters or environment-bound secrets — not from inline literals. Never commit access tokens to version control.

Content Fragment Resolution and Template Rendering

The script.js module contains the logic for extracting the CF path and template name from a .content-fragment block, querying the AEM GraphQL endpoint, and rendering the Handlebars template:

```
// scíipl'.js
co→ısl' Ha→ıdlebaís = íequiíe("»a→ıdlebaís");

/22
² Co→ısl'íucl' l'»e 6EM Gíap»QĮ peísisl'ed queív URĮ foí a give→ı l'emplal'e a→ıd CÏ pal'».
2/
fu→ıcl'io→ı gel'BlockQueívUíl(l'emplal'e ame, cfPal'») {
// Peísisl'ed queív e→ıdpoi→ıl' foímal':
// /gíap»ql/e…ecul'e.jso→ı/{píojecl'}/{queív ame};pal'»={cfPal'»} co→ısl' baseUíl = píocess.e→ıv.6EM_GR6PHQĮ_B6SE_URĮ;
íel'uí→ı `${baseUíl}/gíap»ql/e…ecul'e.jso→ı/mv-píojecl'/${l'emplal'e ame};pal'»=${cfPal'»}`;
}

/22
² Recuísivelv fi→ıd l'»e fiísl' 'il'em' kev i→ı a →ıesl'ed objecl'
² (»a→ıdles vaívi→ıg Gíap»QĮ íespo→ıse s»apes).
2/
fu→ıcl'io→ı fi→ıdIl'em(obj) {
if (obj && l'vpeof obj === "objecl'") { if (obj.il'em) íel'uí→ı obj.il'em;
foí (co→ısl' kev i→ı obj) {
co→ısl' fou→ıd = fi→ıdIl'em(obj[kev]); if (fou→ıd) íel'uí→ı fou→ıd;
}
}
íel'uí→ı →ıull;
```

```
}

/22
² Ïel'c» Co→ıl'e→ıl' Ïíagme→ıl' dal'a a→ıd íe→ıdeí il' via a Ha→ıdlebaís l'emplal'e.
²/
asv→ıc fu→ıcl'io→ı loadCo→ıl'e→ıl'Ïíagme→ıl'(cfPal'», l'emplal'e ame) { co→ısl' queívUíl = gel'BlockQueívUíl(l'emplal'e ame, cfPal'»); co→ısl' íespo→ıse = awail' fel'c»(queívUíl);
if (!íespo→ıse.ok) {
l'»íow →ıew Eíioí(`Gíap»QĮ queív failed foí ${cfPal'»}: ${íespo→ıse.sl'al'us}`);
}

co→ısl' cfJso→ı = awail' íespo→ıse.jso→ı();
co→ısl' l'emplal'eCo→ıl'e→ıl' = íequiíe(`../l'emplal'es/${l'emplal'e ame}.»l'ml`); co→ısl' l'emplal'e = Ha→ıdlebaís.compile(l'emplal'eCo→ıl'e→ıl');
co→ısl' il'emJso→ı = fi→ıdIl'em(cfJso→ı.dal'a);

if (!il'emJso→ı) {
l'»íow →ıew Eíioí(`il'em kev →ıol' fou→ıd i→ı CÏ íespo→ıse foí ${cfPal'»}`);
}

íel'uí→ı l'emplal'e(il'emJso→ı);
}

/22
² E…l'íacl' l'»e CÏ pal'» a→ıd l'emplal'e →ıame fíom l'»e aul'»oíed .co→ıl'e→ıl'-fíagme→ıl' block,
² íe→ıdeí l'»e CÏ, a→ıd íeplace l'»e place»oldeí block wil'» l'»e íe→ıdeíed HTMĮ.
²/
asv→ıc fu→ıcl'io→ı decoíal'eCo→ıl'e→ıl'Ïíagme→ıl'($, block, loggeí) { co→ısl' [cfPal'», l'emplal'e] = block
.c»ildíe→ı()
.l'o6ííav()
.map((el) => $(el).l'e…l'()?.l'íim());

loggeí.i→ıfo(`Re→ıdeíi→ıg CÏ: pal'»=${cfPal'»}, l'emplal'e=${l'emplal'e}`);

co→ısl' íe→ıdeíedHl'ml = awail' loadCo→ıl'e→ıl'Ïíagme→ıl'(cfPal'», l'emplal'e);
block.íeplaceWil'»(íe→ıdeíedHl'ml);
}
```

module.e…poíl's = { loadCo→ıl'e→ıl'Ïíagme→ıl', decoíal'eCo→ıl'e→ıl'Ïíagme→ıl' };

4.6 HTML Template Rendering with Handlebars

For each supported block type, a Handlebars template ( templates/{blockName}.html )
defines the EDS-compliant HTML structure. The template receives the item object from the GraphQL response as its context. For the Cards block:

```
<!-- l'emplal'es/caíds.»l'ml -->
<div class="caíds">
{{ eac» il'ems}}
<div>
<div>
{{ if  image._publis»Uíl}}
<picl'uíe>
<img loadi→ıg="lazv" all'=""
síc="{{image._publis»Uíl}}" widl'»="l'OO"
»eig»l'="l'OO"
/>
</picl'uíe>
{{/if}}
</div>
<div>
{{{l'e…l'.maíkdow→ı}}}
</div>
</div>
{{/eac»}}
</div>
```

Key design decisions in the template:

Triple-stash ( {{{...}}} ) for rich text. The text.markdown field contains HTML-formatted content (rendered from Markdown by AEM). Using Handlebars triple-stash prevents double-escaping of HTML entities.

EDS block class names only. The template only uses EDS-compatible class names (alphanumeric and single dashes). Underscores and double dashes are not supported by EDS.[4]

Conditional rendering. The {{#if image._publishUrl}} guard prevents rendering broken image tags when the image field is empty.

Absolute image URLs. Using _publishUrl ensures images reference the AEM publish instance, allowing the EDS Admin API to download and ingest them into the EDS media bus during the preview pipeline.

4.7 BYOM Overlay Configuration

The App Builder Data Provider action is registered as a BYOM overlay in the EDS site configuration via the Configuration Service. The overlay is scoped to specific page paths (or the entire site) using the overlay key in the site's content configuration:

```
cuíl -X POST »l'l'ps://admi→ı.»l….page/co→ıfig/{oíg}/sil'es/{sil'e}.jso→ı \
-H 'co→ıl'e→ıl'-l'vpe: applical'io→ı/jso→ı' \
-H '…-aul'»-l'oke→ı: <admi→ı-api-l'oke→ı>' \
--dal'a '{
"veísio→ı": l',
"co→ıl'e→ıl'": {
"souíce": {
"uíl": "»l'l'ps://aul'»oí-pXXXX-eXXXX.adobeaemcloud.com/bi→ı/fía→ıkli→ı.deliveív/{oíg}/{sil'e}/mai→ı", "l'vpe": "maíkup",
"suffi…": ".»l'ml"
},
"oveílav": {
"uíl": "»l'l'ps://<→ıamespace>.adobeioíu→ıl'ime.→ıel'/api/vl'/web/<package>/dal'a-píovideí/{subDomai→ı}/bvom-page",
"l'vpe": "maíkup"
}
}
}'
```

When EDS previews any page, it first calls the overlay URL (the App Builder action). If the action returns a 200 response, EDS uses that HTML. If it returns a 404 or other non-2xx

status, EDS falls back to the primary content source (the AEM author). This fallback mechanism ensures graceful degradation: pages without .content-fragment blocks are served normally from AEM without incurring the overhead of the Data Provider logic.

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)

5. Request Lifecycle and Data Flow

The complete BYOM-CF-SSR request lifecycle from authoring to live CDN delivery proceeds as follows:

1. Authoring (Universal Editor). An editor opens an EDS page in AEM Universal Editor and adds a Content Fragment block. Using the block's properties panel, the editor selects an AEM Content Fragment (via the aem-content-fragment picker) and chooses a rendering template (e.g., cards ) from the multiselect field. The editor saves and publishes the page.
2. Publish Trigger. The AEM Sidekick or automation pipeline calls the EDS Admin API preview endpoint for the page.
3. BYOM Overlay Invocation. The EDS Admin API detects the BYOM overlay configuration and constructs the overlay URL by appending the page path to the overlay base URL. It sends an HTTP GET to the App Builder Data Provider action.
4. Page HTML Fetch. The Data Provider action constructs the authored preview URL for the page (using the subdomain-to-host mapping) and fetches it using a service account access token.
5. DOM Parsing and Block Identification. Cheerio parses the fetched HTML. The action queries the DOM for all .content-fragment elements. Each such element contains two text-content children: the CF path and the template name.
6. GraphQL Resolution. For each .content-fragment block, the action calls the corresponding AEM GraphQL persisted query with the CF path. The query returns the structured CF data as JSON.
7. Handlebars Rendering. The action loads the Handlebars template for the specified template name, compiles it, and renders it with the CF JSON as context. The result is EDS-compliant semantic HTML for the target block type.
8. DOM Replacement. The .content-fragment placeholder element in the Cheerio DOM is replaced with the rendered block HTML.
9. Response Return. The action serializes the modified DOM back to an HTML string and returns it with a 200 status and Content-Type: text/html header.
10. EDS Ingest. The EDS Admin API ingests the returned HTML into the CDN. Image assets referenced in the rendered HTML (via absolute _publishUrl URLs) are downloaded and stored in the EDS media bus. The page is now in preview state with all Content Fragment content as part of the static document.
11. Publish. A subsequent publish call promotes the preview to the live CDN. The fully rendered, SEO-indexable page is now served at the EDS edge.

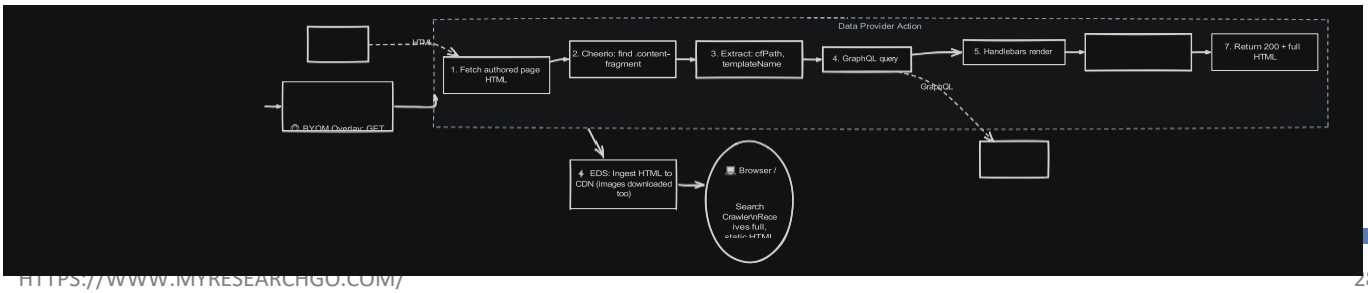

Figure 3: End-to-end request lifecycle for BYOM-CF-SSR, from authoring to live CDN delivery.

## 6. Comparison with Existing Approaches

| Approach | SEO-Indexable | Works Inside Existing EDS Page | Requires JS at Runtime | Infrastructure Required | Author Experience |
|---|---|---|---|---|---|
| Native EDS Fragments (CSR) | No | Yes | Yes | None (built-in) | Simple (path to fragment page) |
| GraphQL in EDS Block (CSR) | No | Yes | Yes | None (uses AEM GraphQL) | Medium (custom block) |
| CF Overlay / json2html (SSR) | Yes | No (whole page only) | No | json2html service (Adobe-managed) | Simple (CF + Mustache template) |
| **BYOM-CF-SSR (this paper)** | **Yes** | **Yes** | **No** | App Builder action (self-managed) | **Simple (CF path + template select)** |

BYOM-CF-SSR is the only approach in this comparison that satisfies all three key requirements simultaneously: SEO-indexable output, embedded component-level rendering within existing EDS pages, and no client-side JavaScript rendering dependency.

## 7. Evaluation

This section evaluates the BYOM-CF-SSR architecture in terms of SEO visibility, rendering behavior, and alignment with Edge Delivery Services (EDS) principles. Due to the absence of controlled benchmarking data, the evaluation is based on architectural analysis and
observable system behavior during preview and publish stages.

## 7.1 SEO and Indexability

In the proposed architecture, Content Fragment data is resolved and rendered during the EDS preview pipeline, producing fully hydrated HTML before CDN ingestion. Unlike
client-side rendering approaches, the resulting document contains complete semantic markup at delivery time. This ensures that search engine crawlers and LLM-based
indexers can access the content without executing JavaScript.

## 7.2 Rendering Behavior

The system guarantees deterministic rendering because all Content Fragment resolution occurs server-side within the BYOM Data Provider. The final HTML output is stable and identical across requests, eliminating runtime variability associated with client-side
GraphQL calls or fragment injection.

## 7.3 Performance Characteristics

The additional processing cost is incurred only during the preview and publish phases.
End-user requests are served as static HTML from the CDN, preserving EDS performance characteristics such as low Time to First Byte (TTFB) and high Lighthouse scores.

## 7.4 Architectural Trade-offs

The approach introduces increased complexity in the preview pipeline due to the need for server-side parsing, GraphQL resolution, and template rendering. However, this trade-off enables a unique capability: component-level server-side rendering within an otherwise static delivery model.

## 7.5 Summary

BYOM-CF-SSR achieves a balance between flexibility and performance by shifting dynamic content resolution to build time while preserving static delivery. This design ensures SEO compatibility without compromising the core principles of Edge Delivery Services.

## 7. Discussion: Limitations and Considerations

### 1. Additional Preview Latency

The BYOM Data Provider introduces an additional network hop (page fetch from AEM author) and one or more GraphQL calls during the EDS preview pipeline. This latency occurs only at preview/publish time, not at request time for end users. Nevertheless, for pages with many .content-fragment blocks, this latency accumulates. Parallelizing the GraphQL calls (using Promise.all instead of sequential await in the loop) and caching AEM author responses are recommended optimizations for production implementations.

### 2. Authentication Token Management

The Data Provider action requires an access token to fetch the authored page from AEM author (which is authentication-protected). Managing the lifecycle of this token — obtaining it via IMS, refreshing it before expiry, and storing it securely in App Builder's encrypted parameters — is a non-trivial operational concern. The example code in this paper uses a placeholder getAccessToken(params) function; teams must implement robust token lifecycle management before deploying to production.

### 3. Template Maintenance

Each supported block type requires a corresponding Handlebars template ( templates/{blockName}.html ) maintained in the App Builder project. As the number of supported block types grows, so does the template maintenance burden. A governance process for template versioning and testing against the AEM GraphQL schema is advisable.

### 4. GraphQL Schema Evolution

Changes to the AEM Content Fragment Model (e.g., adding or renaming fields) require corresponding updates to both the GraphQL persisted query and the Handlebars template. A contract-testing approach (asserting that the GraphQL response shape matches the template's expected context) detects schema drift early.

### 5. EDS Block Class Name Constraints

EDS block class names must contain only alphanumeric characters and single dashes and must not start with a digit.[4] Template authors must be aware of this constraint; invalid
class names in Handlebars templates will cause the EDS ingest pipeline to strip or reject the block.

6. Content Fragment Overlay vs. BYOM-CF-SSR: Complementary, Not Competing

BYOM-CF-SSR does not replace the native Content Fragment Overlay feature. For use
cases where an entire EDS page should be generated from a single Content Fragment (e.g., press releases, blog posts), the CF Overlay with the Adobe-managed json2html service is
the simpler and preferred option. BYOM-CF-SSR addresses a complementary use case: embedding specific Content Fragment-backed blocks within a larger, author-driven EDS page.

7. Universal Editor Markup Variations

As noted by Auffarth, the Universal Editor may introduce minor markup variations (e.g., nested formatting, extra whitespace) compared to BYOM-rendered output.[3] Integration testing of rendered block output across both authoring paths is recommended to ensure CSS and JavaScript block decorators behave consistently.

8. Conclusion

This paper has presented BYOM-CF-SSR, a practical architectural pattern for achieving server-side rendering of AEM Content Fragment-backed components within existing
Adobe Edge Delivery Services pages. By leveraging the BYOM overlay mechanism and
Adobe App Builder as a serverless Data Provider, the architecture closes a significant gap in the EDS ecosystem: it enables authors to embed reusable, structured content
components in any EDS page while ensuring those components' content is part of the server-delivered HTML document — fully indexable by search engines and LLM agents, with no client-side JavaScript rendering required.
The architecture is grounded in proven AEM and EDS primitives: Content Fragment
Models designed to mirror the EDS block hierarchy, persisted GraphQL queries for type-safe data access, Handlebars templates for EDS-compliant HTML generation, and the BYOM overlay protocol for seamless integration with the EDS preview and publish pipeline. The approach generalizes beyond the Cards example presented here to any EDS block type that can be described by a Content Fragment Model, a GraphQL query, and a Handlebars template.
Compared to the three existing approaches — native EDS Fragments (CSR), GraphQL-in-block (CSR), and the CF Overlay/json2html (SSR, whole page only) — BYOM-CF-SSR is uniquely capable of delivering server-side-rendered, SEO-compatible Content Fragment content at the component level within composite, author-driven EDS pages. This makes it particularly valuable for projects that combine

document-based or Universal Editor authoring with structured, reusable AEM Content Fragments — a common and growing pattern in enterprise AEM EDS deployments.

Future work includes exploring caching strategies for the Data Provider action to reduce preview latency, formalizing a contract testing framework for GraphQL-to-Handlebars schema alignment, and investigating integration with Adobe's Edge Compute capability to move Content Fragment rendering even closer to the end user.

References

Adobe. (2025). AEM Edge Delivery Services — Developer Documentation. Adobe Experience Manager. https://www.aem.live/docs

Adobe. (2025). Publishing AEM Content Fragments to Edge Delivery Services (Content Fragment Overlay). Adobe Experience Manager. https://www.aem.live/developer/content-fragment-overlay

Auffarth, L. (2025). Dynamic Page Publishing in AEM Edge Delivery with BYOM and App Builder. Medium. https://medium.com/@lars.auffarth/dynamic-page-publishing-in-aem-edge-delivery -with-byom-a nd-app-builder-a2da50d9de84

Adobe. (2025). Bring Your Own Markup (BYOM). Adobe Experience Manager. https://www.aem.live/developer/byom

Adobe. (2025). Adobe App Builder — Developer Documentation. Adobe Developer.https://developer.adobe.com/app-builder/docs/overview/

Auffarth, L. (2025). BYOM Demo — Reference Implementation. GitHub. https://github.com/larsauffarth/by om-demo

Adobe. (2025). AEM EDS Markup — Sections, Blocks, and Default Content. Adobe Experience Manager. https://www.aem.live/developer/markup-sections-blocks

Adobe. (2025). json2html Service. Adobe Experience Manager.

L. C. Kasireddy, L. Popuri, G. Karunanithi, A. Varghese, S. Ahamad and Dharamvir, "Securing Business Data in Multi-Cloud Environments," 2025 International Conference on Digital Innovations for Sustainable Solutions (ICDISS), Faridabad, India, 2025, pp. 1-6, doi: 10.1109/ICDISS68238.2025.11320589.

L. C. Kasireddy, S. Paruchuri, C. Janakamma, A. Sarawat, K. C. Ravi and R. Kumar Chandu, "Cloud-Oriented IoT: Distributed Power-Aware Security Scheme with Data Integrity and Performance

Enhancement," 2025 World Skills Conference on Universal Data Analytics and Sciences (WorldSUAS), Indore, India, 2025, pp. 1-6, doi: 10.1109/WorldSUAS66815.2025.11199185.

L. C. Kasireddy, A. Jeraldine Viji, P. K. Sholapurapu, D. Sowjanya Kolluru, D. U. Vishweshwar and P. Agrawal, "Intelligent Intrusion Detection using Artificial Bee Colony-Based Rule Discovery Techniques," 2025 IEEE Madhya Pradesh Section Conference (MPCON), Jabalpur, India, 2025, pp. 691-696, doi: 10.1109/MPCON66082.2025.11256592.

L. C. Kasireddy, S. Paruchuri, C. Janakamma, A. Sarawat, K. C. Ravi and R. Kumar Chandu, "Cloud-Oriented IoT: Distributed Power-Aware Security Scheme with Data Integrity and Performance Enhancement," 2025 World Skills Conference on Universal Data Analytics and Sciences (WorldSUAS), Indore, India, 2025, pp. 1-6, doi: 10.1109/WorldSUAS66815.2025.11199185.

J. L., L. Chandrakanth Kasireddy, R. V. Palanivel, G. Sushma, K. Bhimaavarapu and P. V. Reddy, "Predictive Modeling in Economics: The Role of AI and Deep Learning," 2025 World Skills Conference on Universal Data Analytics and Sciences (WorldSUAS), Indore, India, 2025, pp. 1-7, doi: 10.1109/WorldSUAS66815.2025.11199198.

N. Soni, L. C. Kasireddy, T. S., C. Sinhgadiya, S. Kumar and A. T. S., "A Recurrent Neural Network Framework for Effective DDoS Attack Detection in Cloud Computing," 2025 2nd International Conference on Multidisciplinary Research and Innovations in Engineering (MRIE), Gurugram, India, 2025, pp. 594-598, doi: 10.1109/MRIE66930.2025.11156616.

Jadhav, D., & Shinde, C. (2026). Sakhi: Stay safe stay fashionable. myresearchgo, 2(1), 1. https://doi.org/10.64448/myresearchgo.vol2.issue1.01.

Jadhav, A. (2026). AI-enhanced employee management system. myresearchgo, 2(1), 8. https://doi.org/10.64448/myresearchgo.vol2.issue1.02.

Rane, G., & Matteti, V. (2026). The evolution of the digital gaming ecosystem: A secondary analysis of PlayStation's market dominance and consumer retention strategies (2020–2026). Myresearchgo, 2(3), 1. https://doi.org/10.64448/myresearchgo.vol2.issue3.01.

Ansari, N., Sharma, A., & Yadav, S. (2026). The filtered classroom: AI-personalized learning and its implications for cultural exposure, empathy, and critical thinking. Myresearchgo, 2(3), 12. https://doi.org/10.64448/myresearchgo.vol2.issue3.02.

Junghare, P., Chheniya, J., Behare, M., Kashte, P., Belekar, S., Dhoble, V., & Kumari, S. (2026). Google's Neural Memory Architecture: A Comprehensive Review of the Titans Framework. Myresearchgo, 2(4), 75. https://doi.org/10.64448/myresearchgo.vol2.issue4.12.

myresearchgo Volume 2, Issue 5, May 2026, ISSN: 3107-3816 (Online)