

WHITEPAPER

Delivering commerce experiences at scale

With Adobe Experience Manager, Commerce Integration Framework, Adobe Commerce

By Grant Naber, Customer Success Engineer Manager, Adobe



Table of Contents

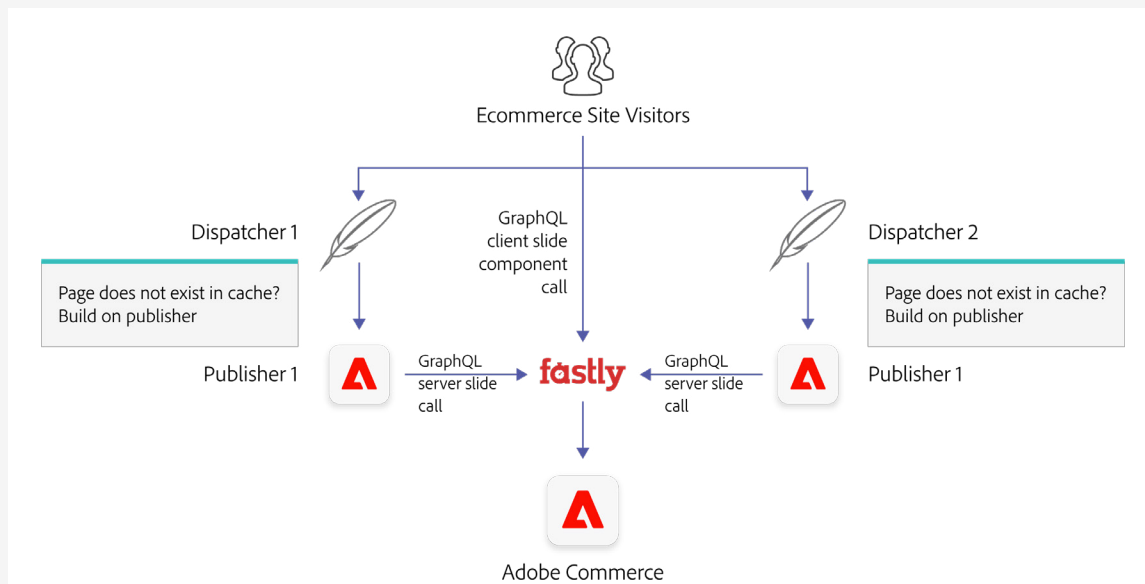
AEM/CIF/Adobe Commerce overview	3
Server-side rendering	4
Client-side rendering	4
Planning effective caching for Ecommerce success under load	5
AEM performance optimisations under load from default configurations	6
TTL based caching on the AEM Dispatchers	6
Browser caching	6
Dispatcher statfilelevel and gracePeriod settings optimisation	7
Adobe Experience Manager	4
Adobe Commerce	4
CIF – GraphQL caching via components	8
Hybrid Caching – client side GraphQL requests within cached dispatcher pages	8
Uncacheable GraphQL requests	9
Ignoring Tracking Parameters on AEM Dispatcher cache	9
MPM workers limits on dispatchers	10
AEM / Adobe Commerce / Infrastructure alignments – timeouts and connection limits	10
1. AEM Load Balancer	11
2. Dispatchers' timeout settings	11
3. Publishers – connection limits, timeouts and default configuration changes	12
Adobe Commerce – changes from default settings	13
Geographic location of AEM and Adobe Commerce infrastructure	13
AWS/Azure Privatelink – linking AEM And Adobe Commerce on cloud VPCs	13
GraphQL Caching in Adobe Commerce	13
Magento Indexer settings	13
Magento catalogue flat tables	14
Fastly origin shielding	14
Fastly image optimisation	14
Disabling unused Adobe Commerce modules	15
MySQL and Redis Slave connection activation	15
Moving to a Adobe Commerce on cloud scaled (split) architecture	15
Preparing for launch – AEM/CIF/Adobe Commerce performance testing tips	16
Conclusion	18



AEM/CIF/Adobe Commerce overview

A recommended integration pattern between AEM and Adobe Commerce using CIF as a connector is for AEM to own the presentation layer (the “glass”) and Adobe Commerce to power the commerce backend as a “headless” backend. This integration approach leverages the strengths of each application: the authoring, personalisation, and omnichannel capabilities of AEM and ecommerce operations of Adobe Commerce.

In an AEM/CIF/Adobe Commerce environment, ecommerce site visitors will initially arrive at AEM. AEM will check if it has the requested page available in its dispatcher cache. If the page exists, this cached page will be served to the visitor, and no further processing is required. If the dispatcher does not contain the requested page, or it has expired, then the dispatcher requests the AEM publisher to build the page, with the publisher calling Adobe Commerce for ecommerce data to build the page if necessary. The built page is then passed to the dispatcher to serve to the visitor and is then cached preventing the need for further load to be placed onto the servers on subsequent requests to the same page from other visitors.



A combination of server-side rendering, and client-side rendering can be used in the AEM/CIF/Adobe Commerce model: Server-side rendering to deliver static content and client-side rendering to deliver frequently changing or personal dynamic content by directly calling Adobe Commerce for specific components from within the user’s browser.

An example of the different components in a Product Detail Page on an example AEM ecommerce storefront can be seen in the example below:

The screenshot shows a product detail page for a 'Sabina Hooded Cardigan' on an AEM storefront. The page is annotated with red and purple boxes and callouts explaining the rendering strategy.

Red - Realtime calls to Adobe Commerce from Client browser - data always kept up to date, but does results in repeat requests and load on Adobe Commerce

Purple - calls to Adobe Commerce from AEM publisher during building of page

The page is then subsequently cached in the dispatcher - no further requests to, or load on, Magento needed

Server-side rendering

Ecommerce pages such as product detail pages (PDPs) and product listing pages (PLPs) are unlikely to change frequently and are suited to be fully cached after being rendered server-side using AEM CIF Core Components. The pages should be rendered on the AEM publisher using generic templates created in AEM. These components get data from Adobe Commerce via GraphQL APIs. These pages are created dynamically, rendered on the server, cached on the AEM dispatcher and then delivered to the browser. Examples of this are shown in the purple boxes in the example above.

Client-side rendering

Where more dynamic attributes such as stock levels/availability or price are displayed, for example on Product Detail Pages (PDP's), client-side components can be used. Whilst the template page can be built and cached on the dispatcher using the server-side rendering approach above, within the static page itself there can be dynamic client-side web components. These dynamic components can fetch data directly in the client's browser from Adobe Commerce via GraphQL APIs to check, for example, current price or stock level in real time on the PDP. This ensures that content that is usually critical to be shown in real time is always fetched on page load. Examples of this are shown in the yellow boxes in the example above.

A combination of AEM templates and client-side rendering can also be used during the checkout process: client-side cart components render the shopping cart, checkout form and integration with the payment service provider. This hybrid approach can also be used for Adobe Commerce's account management functionality such as create account, signing into account, and forgotten password.



Planning effective caching for Ecommerce success under load

Delivering a shopping experience under load will require caching strategy planned in advance. Whilst initially, the request from business stakeholders may be to always present real time product data to customers, this is not an optimal use of system resources, and the impacts of end user site performance would greatly outweigh the benefits of consistently showing real time information.

The initial step in caching strategy should therefore be to define with the relevant stakeholders a matrix of acceptable caching timings for the different areas of the site, for example:

Caching Area	How often changes?	Impact if stale content served from cache?	Acceptable caching TTL (Time to live)
Site content HTML pages, updated via CMS	Infrequently	Low	1 day
Site content template media/ assets – logo, CSS design images etc.	Infrequently	Low	1 week
Product Listing Pages (PLP)	Infrequently	Medium	1 day
Product Detail Pages (PDP)	Sometimes	Medium	1 hour
Product Categories	Infrequently	Medium	1 day
Prices	Frequently	High	No cache
Inventory / Stock	Frequently	High	No cache
Site Search	Most users unique	Medium	Cache results from top 100 search request phrases for 1 day
Checkout	Every user unique	Very high	No cache
Shopping cart	Every user unique	Very high	No cache
Payment pages	Every user unique	Very high	No cache

With this initial planning complete, the technical configuration can start to be put in place to configure caches based on these requirements.

Even if content is updated and needing to be made live within the caching TTL, it is, in most cases, possible to manually clear the caches for the AEM dispatcher and Adobe Commerce cache selectively for that content, meaning that urgent changes will be reflected immediately. The process around manual cache clearing should also be planned and tested in advance so that if there is a need to manually force an update on some content, then it is documented into a site operations runbook and clear how and who needs to be involved to action this. An example manual cache clear operation for AEM and Adobe Commerce is shown here.



AEM performance optimisations under load from default configurations

The AEM dispatcher is a reverse proxy, that helps deliver an environment that is both fast and dynamic. It works as part of a static HTML server, such as Apache HTTP Server, with the aim of storing (or “caching”) as much of the site content as is possible, in the form of static resources. This approach aims to minimize the need to access the AEM page rendering functionality and the Adobe Commerce GraphQL service as much as possible. The result of serving much of the pages as static HTML, CSS, & JS delivers performance benefits to users and reduces infrastructure requirements on the environment. Any page or query that is likely to be identically repeated from user to user should be considered for caching.

The following sections show at a high level the recommended technical focus area to be reviewed to enable effective caching on AEM in a CIF/Adobe Commerce environment.

TTL based caching on the AEM Dispatchers

Caching as much of the site as possible on the dispatchers is best practice for any AEM project. Using time-based cache invalidation will cache server side rendered CIF pages, for a set limited amount of time. After the set time has expired, the next request will rebuild the page from the AEM publisher and Adobe Commerce GraphQL and will store it in the dispatcher cache again until the next invalidation.

The TTL caching feature can be configured in AEM with using the “Dispatcher TTL” component within the ACS AEM Commons package and setting `/enableTTL “1”` in the dispatcher.any configuration file.

If enabled, the dispatcher will evaluate the response headers from the backend, and if they contain a Cache-Control max-age or Expires date, an auxiliary, empty file next to the cache file is created, with modification time equal to the expiry date. When the cached file is requested past the modification time it is automatically re-requested from the backend. This gives an effective caching mechanism which requires no manual intervention or maintenance, once the product update delay (TTL) has been acknowledged and accepted by business stakeholders.

Browser caching

The dispatcher TTL approach above will greatly reduce requests and load onto the publisher, however there are some assets which are very unlikely to change and therefore even requests to the dispatcher can be reduced by caching relevant files locally on a user’s browser. For example, the site’s logo, which is displayed on every page on the site in the site template, would not need to be requested each time to the dispatcher. This instead can be stored on the user’s browser cache. The reduction in bandwidth requirements for each page load would have a large impact on site responsiveness and page load times.

Caching at the browser level is commonly done via the “Cache-Control: max-age=” response header. The max-age setting tells the browser how many seconds it should cache the file for before attempting to “revalidate” or request it from the site again. This concept of cache max-age is commonly referred to as “Cache Expiration” or TTL (“Time to Live”).

Some areas of an AEM/CIF/Adobe Commerce site which can be set to be cached in the client's browser include:

- Images (within the AEM template itself, e.g. site logo and template design images – catalogue product images would be called from Adobe Commerce via Fastly, caching these images are discussed later on)
- HTML files (for infrequently changed pages – terms and conditions page etc)
- CSS files
- All site JavaScript files – including CIF JavaScript files

Dispatcher statfilelevel and grace period settings optimisation

The default dispatcher configuration uses /statfilelevel "0" setting – this means that a single ".stat" file is placed at the root of htdocs directory (document root directory). If a change is made to a page or file in AEM, the modification time of this single stat file is updated to the time of the change. If the time is newer than modification time of the resource, then the dispatcher will consider all resources are invalidated and any subsequent request for an invalidated resource will trigger a call to the Publish instance. So essentially, with this setting every activation will invalidate the whole cache.

For any site, especially commerce sites with heavy load, this would place an unnecessary amount of load onto the AEM Publish tier for the whole site structure to become invalidated with only a single page update.

Instead, the statfilelevel setting can be modified to a higher value, corresponding to the depth of sub-directories in the htdocs directory from the document root directory so that when a file located at a certain level is invalidated then only files at that .stat directory level and below are updated.

For example: let's say you have a product page template at:

content/ecommerce/us/en/products/product-page.html

Each folder level would have 'stat level' – as shown broken down in the table above.

content (docroot)	ecommerce	us	en	products	product-page.html
0	1	2	3	4	-

In this case, if you had left the statfilelevel property set to the default "0", and the product-page.html template is updated and activated triggering an invalidation then every .stat file from docroot to level 4 will be touched, and files invalidated, causing a further request from the AEM publish instances for all pages across the site (including other websites, countries and languages) from that single change.

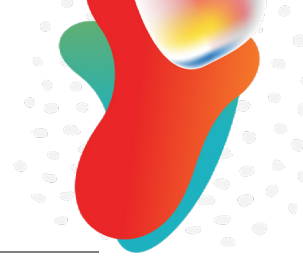
However, if the statfilelevel property is set to level 4, and a change is made to the product-page.html – then only the .stat file in products directory for that specific website/country/language would be touched.

Please note that the .stat file level shouldn't be set to a too high level – exceeding 20 can have performance impacts. Executing a bulk file activation whilst running a performance test should give you the correct level you should tune your stat level to.

Another dispatcher setting to optimise when configuring the statfilelevel is the gracePeriod setting. This defines the number of seconds a stale, auto-invalidated resource may still be served from the cache after the last activation occurred. Auto-invalidated resources are invalidated by any activation (when their path matches the dispatcher /invalidate section, and to the level specified in the statfilelevel property). Setting the gracePeriod setting to 2 seconds can be used to prevent a scenario where multiple requests are continually sent to the publisher, even while the publisher is still in the process of building the new page*.

* Further more detailed reading on this topic is available at the following link: <https://github.com/adobe/aem-dispatcher-experiments/tree/main/experiments/gracePeriod>





CIF – GraphQL caching via components

Individual components within AEM can be set to be cached, meaning that the GraphQL request to Adobe Commerce is called once and then subsequent requests, up to the configured time limit, are retrieved from the AEM cache and would not place further load onto Adobe Commerce. Examples would be a site navigation based on a category tree shown on every page and options within a faceted search functionality – these are just two areas which require resource intensive queries on Adobe Commerce to build yet would be unlikely to change regularly and therefore would be good choices for caching. This way, for example, even when a PDP or PLP is being rebuilt by the publisher, the resource intensive GraphQL request for the navigation build would not hit Adobe Commerce and could be retrieved from the GraphQL cache on AEM CIF.

An example below is for the navigation component to be cached because it sends the same GraphQL query on all pages in the site. The request below caches the past 100 entries for 10 minutes for the navigation structure:

```
venia/components/structure/navigation:true:100:600
```

The example below caches the past 100 faceted search options in a search page for 1 hour:

```
com.adobe.cq.commerce.core.search.services.SearchFilterService:true:100:3600
```

The request, including all custom http headers and variables, must match exactly in order for the cache to be 'hit' and to prevent a repeat call to Adobe Commerce. It should be noted there once set there is no easy way to manually invalidate this cache. This could mean, therefore that if a new category is added in Adobe Commerce, it would not start to appear in the navigation until the expiry time set in the cache above has expired and the GraphQL request is refreshed. The same for search facets. However, given the performance benefits to be achieved by this caching, this is usually an acceptable compromise .

The above caching options can be set using the AEM OSGi configuration console in "GraphQL Client Configuration Factory". Each cache configuration entry can be specified with the following format:

- NAME:ENABLE:MAXSIZE:TIMEOUT like for example mycache:true:1000:60 where each attribute is defined as:
 - > NAME (String): name of the cache
 - > ENABLE (true|false): enables or disables that cache entry
 - > MAXSIZE (Integer): maximum size of the cache (in number of entries)
 - > TIMEOUT (Integer): timeout for each cache entry (in seconds)

Hybrid Caching – client side GraphQL requests within cached dispatcher pages

It is also possible for a hybrid approach to caching of pages: it is possible for a CIF page to contain components which would always request the latest information from Adobe Commerce directly from the customer's browser. This can be useful for specific areas of the page within a template which are important to be kept up to date with real time information: Product prices within a PDP, for example. Where prices are changing frequently due to dynamic price matching, that information can be configured to be not cached on the dispatcher, rather the prices can be fetched client-side in the customer's browser from Adobe Commerce directly via GraphQL APIs with AEM CIF web components.



This can be configured via the AEM components settings – for Price information on product list pages, this can be configured in the product list template, selecting the product list component on the page settings and checking the “load prices” option. The same approach would work for stock levels.

The methods above should only be used in the case where real time, constantly up to date information is a requirement. In the example above with pricing, it could be agreed with business stakeholders to update prices only daily at low traffic times and perform cache flush operation then. This would remove the need for the real time pricing information requests and the subsequent extra load onto Adobe Commerce when building each page displaying pricing information.

Uncacheable GraphQL requests

Specific dynamic data components within pages should not be cached and will always require a GraphQL call to Adobe Commerce, such as for the shopping cart and calls throughout the checkout pages. This information is specific to a user and is changing constantly due to customer’s activity on the site – e.g. by adding products to their shopping cart.

GraphQL Query results should not be cached for logged in customers if the site’s design would give different responses based on the user’s role. For example, you can create multiple customer groups and set up different product prices or different product category visibility for each group. Caching results such as these may cause customers to see the prices of another customer group or to have incorrect categories showing.

Ignoring Tracking Parameters on AEM Dispatcher cache

Ecommerce sites may drive traffic to their site using PPC search adverts or social media campaigns. Use of these mediums will mean that a tracking ID is added onto the outbound link from that platform. For example, Facebook will add a Facebook Click ID (fbclid) to the URL, Google Adverts will add a Google Click ID (gclid), this will make incoming links to your AEM frontend appear like the below, as an example:

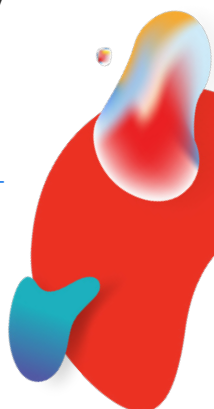
```
https://www.adobe.com/?gclid=oirhgj34y43yowiahg9u3t
```

The gclid and fbclid will change with every user that clicks the advert, this is intended for tracking purposes, but with its default settings, AEM would see every request as a unique page, which would bypass the dispatcher and generate unnecessary extra load on the publisher and Adobe Commerce. During a surge event this can even cause the AEM publishers to become overloaded and unresponsive.

When a parameter is set to be ignored for a page, the page is cached the first time that the page is requested. Subsequent requests for the page are served the cached page, regardless of the value of the parameter in the request[†].

It should therefore be configured to ignore all parameters by default in “ignoreUrlParams”, except where a GET parameter is used which would change the HTML structure of a page. An example of this would be with a search page where the search term is in the URL as GET parameter – in this case you should then manually configure ignoreUrlParams to ignore parameters such as gclid, fbclid and any other tracking parameters your advertising channels are using, leaving the GET parameters required for normal site operations unaffected.

[†] Further reading on the importance of setting ignoreUrlParams is available here: <https://github.com/adobe/aem-dispatcher-experiments/tree/main/experiments/ignoreUrlParams>



MPM workers limits on dispatchers

The MPM workers settings is an advanced Apache HTTP server configuration which would require thorough testing to optimise based on your Dispatcher's available CPU and RAM. However, in the scope of this whitepaper we would suggest that ServerLimit and MaxRequestWorkers, should be increased to a level that the server's available CPU and RAM would support, and then the MinSpareThreads and MaxSpareThreads be both increased to a level which matches the MaxRequestWorkers.

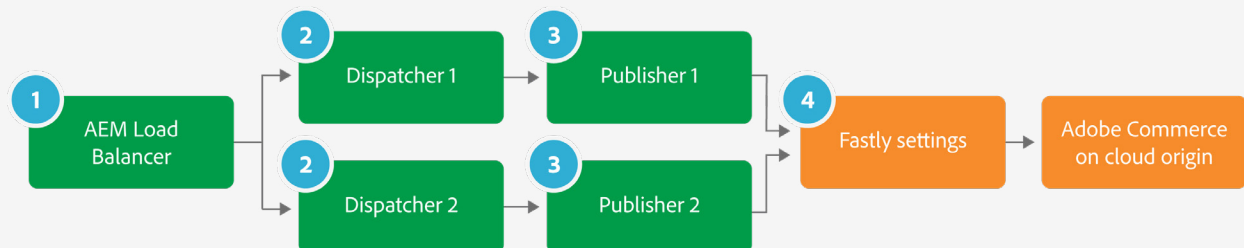
This configuration would leave Apache HTTP on an "full readiness setting" which is a high-performance configuration for servers with significant RAM and multiple CPU cores. This configuration will produce the best possible response times from Apache HTTP by maintaining persistent open connections ready to serve requests and would remove any delay in spawning new processes in response to sudden traffic surges, such as during flash sales.

AEM / Adobe Commerce / Infrastructure alignments – timeouts and connection limits

There are settings with AEM and Adobe Commerce and surrounding infrastructure such as load balancers which need alignment, these are related to connection limits and timeout settings.

A misalignment between these limits would mean that connections could end up being throttled at AEM side, whilst Adobe Commerce is capable of handling more connections. Similarly, for the timeout settings, a misalignment could mean timeout errors occur on AEM side, while Adobe Commerce is still processing a request.

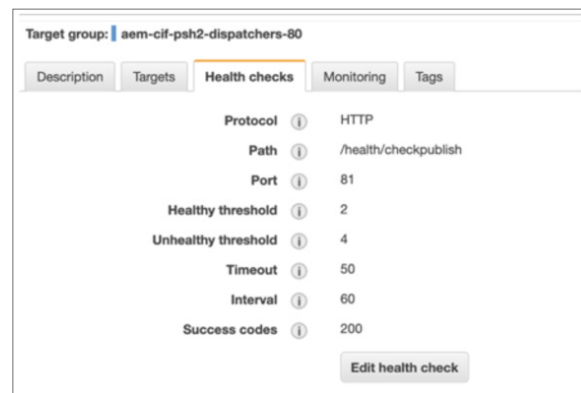
For the timeout settings, the settings should be reviewed and aligned to prevent 503 timeout errors appearing when under load. There are several infrastructure and application timeout settings to review:



1. AEM Load Balancer

Assuming there is an AWS application load balancer in the infrastructure and multiple dispatchers/publishers – the following settings should be considered for the load balancer:

- a. Publisher health checks should be reviewed to prevent dispatchers dropping out of service unnecessarily early from load surges. The timeout settings of the load balancer health check should be aligned with the publisher timeout settings.



Target group: aem-cif-psh2-dispatchers-80

Description Targets **Health checks** Monitoring Tags

Protocol ⓘ HTTP

Path ⓘ /health/checkpublish

Port ⓘ 81

Healthy threshold ⓘ 2

Unhealthy threshold ⓘ 4

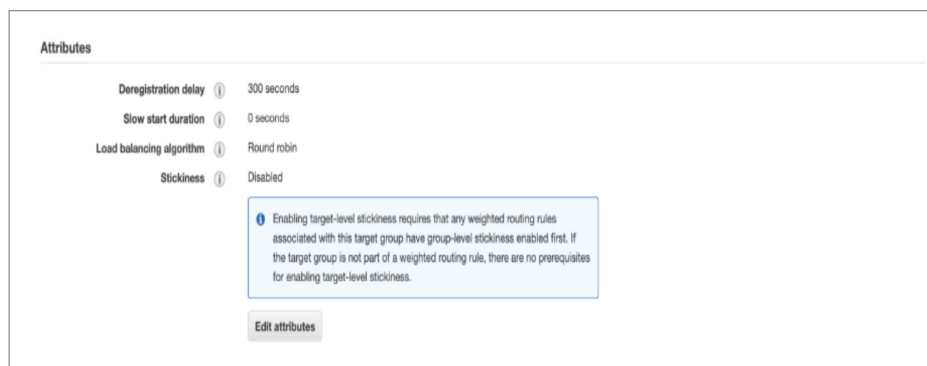
Timeout ⓘ 50

Interval ⓘ 60

Success codes ⓘ 200

Edit health check

- b. Dispatcher target group stickiness can be disabled, and Round Robin load balancing algorithm can be used. This is assuming there is no AEM specific functionality or AEM user sessions used that would require session stickiness to be set. It assumes that user login and session management is only on Adobe Commerce via GraphQL.



Attributes

Deregistration delay ⓘ 300 seconds

Slow start duration ⓘ 0 seconds

Load balancing algorithm ⓘ Round robin

Stickiness ⓘ Disabled

Enabling target-level stickiness requires that any weighted routing rules associated with this target group have group-level stickiness enabled first. If the target group is not part of a weighted routing rule, there are no prerequisites for enabling target-level stickiness.

Edit attributes

- c. Please note if you do enable session stickiness, this may cause requests to Fastly not be cached, as by default, Fastly does not cache pages with the Set-Cookies header. Adobe Commerce sets cookies even on cacheable pages (TTL > 0), but the default Fastly VCL strips those cookies on cacheable pages in order for Fastly caching to work. If pages are not caching, check any custom cookies you may be using and also upload the Fastly VCL and recheck the site.

2. Dispatchers' timeout settings

The /timeout in the dispatcher "renders" options specifies the connection timeout accessing the AEM publish instance in milliseconds. This should be reviewed and the default setting of "0" (indefinite timeout) should be used if a separate load balancer is present to handle the timeout settings.

If there is no load balancer in the infrastructure, the timeout settings should instead be specified in the dispatcher /timeout settings, with a value that matches the GraphQL timeout settings in the publisher.

3. Publishers – connection limits, timeouts and default configuration changes

Publisher GraphQL connection limits and timeouts: Initially, the Max HTTP connections in Adobe Commerce CIF GraphQL Client Configuration Factory OSGI settings should be set to the default Fastly maximum connections limit, which is currently set to 200. Even if there are multiple publishers in the AEM farm, the limit should be set the same across each publisher, matching the Fastly setting. The reason for this is that in some cases, one publisher could be handling more traffic than the other publishers, if an associated dispatcher is taken out of the farm for example. This would mean that all traffic would be routed through the single remaining dispatcher and publishers, in this case the single publisher may then need all the HTTP connections.

The “Default HTTP method” should be set from POST to GET. Only GET requests are cached in Adobe Commerce GraphQL cache and so the default method should always be set to GET.

The http connection timeout and http socket timeout should be set to a value that matches the Fastly timeout.

The screenshot shows the 'CIF GraphQL Client Configuration Factory' dialog box. It contains several configuration fields: 'GraphQL Service Identifier' (Default), 'GraphQL Service URL' (https://www.magentosite.com/graphql), 'Default HTTP method' (GET), 'Accept self-signed SSL certificates' (unchecked), 'Allow HTTP communication' (unchecked), 'Max HTTP connections' (200), 'HTTP connection timeout' (30000), 'HTTP socket timeout' (30000), 'Request pool timeout' (2000), 'Default HTTP Headers' (empty), and 'GraphQL cache configurations' (a list of cache entries). At the bottom, there is a 'Configuration Information' section showing the 'Persistent Identity (PID)' and 'Factory Persistent Identifier (Factory PID)'.

Magento CIF GraphQL Client Configuration Factory (settings shown are examples only and need to be tuned on a case by case basis)

The screenshot shows the 'Advanced options' section of the Fastly backend configurations. It includes fields for 'Maximum connections' (200), 'Error threshold' (0), and a 'TIMEOUTS' section with 'Connection timeout' (Customer specific setting), 'First byte timeout' (Customer specific setting), and 'Between bytes timeout' (Customer specific setting). There is also an 'Override host' field.

Fastly backend configurations (settings shown are examples only and need to be tuned on a case by case basis)

Adobe Commerce – changes from default settings

Geographic location of AEM and Adobe Commerce infrastructure

To reduce latency between the AEM publisher and Adobe Commerce GraphQL when building pages, the initial provisioning of the two separate infrastructures should be hosted within the same AWS (or Azure) Region. The geographical location chosen for both clouds should also be closest to the majority of your customer base, so that client side GraphQL requests are served from a geographically close location to the majority of your customers.

AWS/Azure Privatelink – linking AEM And Adobe Commerce on cloud VPCs

To improve response times and reduce latency further still for sites with extreme load expectations, an AWS (or Azure) Privatelink connection can be considered between AEM's VPC and Adobe Commerce on cloud's VPC. This would have the effect of all network traffic between the AEM publishers and Adobe Commerce staying on the global AWS backbone and not needing to traverse the public internet.

GraphQL Caching in Adobe Commerce

When the user's browser or AEM publisher calls Adobe Commerce's GraphQL, certain calls will be cached in Fastly. The queries that are cached are generally those which contain non personal data and are not likely to change often. These are for example: categories, categoryList and products. Those that are explicitly not cached are those which change regularly and if cached could pose risks to personal data and site operations for example queries such as cart and customerPaymentTokens.

GraphQL allows you to make multiple queries in a single call. It is important to note that if you specify even one query that Adobe Commerce does not cache with many others that are not cacheable, Adobe Commerce will bypass the cache for all queries in the call. This should be considered by developers when combining multiple queries to ensure potentially cacheable queries are not unintentionally bypassed[‡].

Magento Indexer settings

By default, indexers are set to "Update on Save". This would not be an issue for a low traffic site which is not regularly updated, but for a site with heavy load, this setting would cause issues with site response times. For sites expecting heavy load, all indexers should be set to "Update on Schedule". This setting performs targeted updates to only updated content's indexer references in the background with the use of cron jobs and will not impact on GraphQL response times or site performance.

[‡] Further information on cacheable and non-cacheable queries can be found in the Adobe Commerce documentation: <https://devdocs.magento.com/guides/v2.4/graphql/caching.html>

Magento catalogue flat tables

The use of flat tables for products and categories is not recommended. Use of this deprecated feature can result in performance degradations and indexing issues, therefore flat catalogue should be disabled via the Adobe Commerce admin, in the storefront section. Some third-party modules and customisations do require flat tables to function correctly – it is recommended that an evaluation be done to understand impacts and risks associated with having to use flat tables when choosing to utilize these extensions or customisations.

Fastly origin shielding

By default, Fastly origin shielding is not enabled. The purpose of Fastly's origin shielding is to reduce traffic directly to the Adobe Commerce origin: when a request is received, a Fastly edge location (or "point of presence" / POP) checks for cached content and provides it. If it is not cached, it continues to the Shield POP to check if it is cached there (if the content has previously been requested even from another global POP, it will be cached). Finally, if not cached on the Shield POP, it will only then proceed to the Origin server.

Fastly origin shielding can be enabled in your Adobe Commerce admin Fastly configuration backend settings. You should choose a shield location which is closest to your Adobe Commerce origin datacentre for the best performance.

Fastly image optimisation

Once Fastly origin shielding is enabled, this allows you to also activate Fastly Image Optimizer. Where product catalogue images are stored on Adobe Commerce, this service gives the ability to offload all resource intensive, product catalogue images transformation processing onto Fastly and off from the Adobe Commerce origin. End user response times are also improved for page load times, as images are transformed at the edge location which eliminates latency by reducing the number of requests back to the Adobe Commerce origin.

Fastly Image optimization can be enabled by "enable deep image optimization" in Fastly configuration in admin, although only after your origin shield has been activated. More details on configurations for Fastly Image optimisation is available on Adobe Commerce Devdocs.



Image optimization

Fastly IO snippet Enable/Disable **Current state: enabled**
VCL snippet upload is required in order to funnel image requests to Fastly optimizers.

Default IO config options Configure
Allows you to (re)configure items such as default quality levels lossy image formats, auto WebP, JPG types

Enable deep image optimization [store view]
Turns off Magento built-in image resizing and manipulation and offloads it onto Fastly IO. Please read the [image optimization guide](#) for caveats and details.

Enable adaptive device pixel ratios [store view]
Image sources will be rewritten to use srcsets supporting [adaptive device pixel ratios](#). Useful for Progressive Web Apps.

Disabling unused Adobe Commerce modules

If running Adobe Commerce headless, only serving requests through the GraphQL endpoint and no front-end store pages are being served directly from Adobe Commerce, then many modules become redundant and not used. By disabling unused modules, your Adobe Commerce code base becomes smaller, less complex and therefore could offer performance improvements. Disabling modules on Adobe Commerce can be managed using composer. Which modules that can be disabled would depend on the requirements for your site, and so no recommended list can be given as it would be specific to each customer's implementation of Adobe Commerce.

MySQL and Redis Slave connection activation

By default, MySQL and Redis Slave connections are not activated in Adobe Commerce on cloud. This is because these settings are only suitable for customers that are expecting very high load. The Cross-AZ (cross-Availability Zones) latency is higher with slave connections activated and so this setting actually reduces performance of a Adobe Commerce on cloud instance in the case the instance is receiving only regular load levels.

If the Adobe Commerce instance is expecting extreme load, then activating master-slave for MySQL and Redis will help with performance by spreading out the load on the MySQL Database or Redis across different nodes.

As a guide, on environments with normal load, enabling Slave Connections will slow down performance by 10-15%. But on clusters with heavy load and traffic, there is a performance boost of around 10-15%. Therefore, it is important to load test your environment with expected traffic levels to evaluate if this setting would be beneficial to your performance times under load.

To enable/disable slave connections for mysql and redis you should edit your ".magento.env.yaml" file to include the following:

```
stage:
  deploy:
    MYSQL_USE_SLAVE_CONNECTION: true
    REDIS_USE_SLAVE_CONNECTION: true
```

For scaled architecture (split architecture – see below), Redis slave connections should not be enabled, as this will cause errors to appear. In the case of a split architecture, it is instead recommended to implement L2 caching for Redis.

Moving to a Adobe Commerce on cloud scaled (split) architecture

If after all the configurations above, load test results or analysis of live infrastructure performance still indicates that the load levels to Adobe Commerce are of a level which consistently maxes out CPU and other system resources, then a move to a scaled (split) architecture should be considered.

With a standard Pro architecture, there are 3 nodes, each of which contains a full tech stack. By converting to split tier architecture, this changes to a minimum of 6 nodes: 3 of which contain Elasticsearch, MariaDB, Redis and other core services; the other 3 for processing web traffic contain php-fpm and NGINX. There are greater scaling possibilities with split tier: core nodes containing databases can be scaled vertically; web nodes can be scaled horizontally and vertically, giving a large amount of flexibility to expand infrastructure on demand for set period of high load activity and on nodes where the extra resources are needed.

If a decision has been made to switch to a split tier architecture due to heavy load expectations for your site, then a discussion should be engaged with your Customer Success Manager on the steps to enable this.



Preparing for launch – AEM/CIF/Adobe Commerce performance testing tips

To evaluate the effectiveness of all of the changes above, thorough performance testing should be run before go-live and before any future major deployments to your production environments. When planning your load testing, it is important to simulate real life consumer traffic as much as possible.

The most resource intensive areas of the AEM/CIF/Adobe Commerce site are those which are not cacheable such as the checkout process and site search. Static, and therefore cacheable, page browsing such as for Produce Detail Pages (PDP's) and Product Listing Pages (PLP's) make up the majority of the traffic to an ecommerce site generally and so the scripts and scenarios in the test should reflect that to measure the limits of the platform.

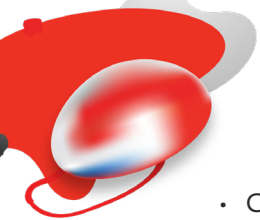
Having a single script running for your load test which navigates through the site with no wait time between steps, and also always completes the checkout process every time would not give a reliable indication of the limits of the platform, as that is not what a real-life scenario would be.

Defining KPIs should be the first step in your performance test plan: define metrics that you can test during your initial test, but then measure again in the future, and on a recurring basis after the site is live. This allows you track the performance of your site's performance over time – pre and post launch. Example KPIs to define could be:

- Average response time - time to first byte or last byte
- Latency
- Bytes/s (Throughput)
- Error rate
- Orders per hour
- Page views per hour
- Unique users per hour (concurrent shoppers)

The following Jmeter high level guidelines should be considered when developing your AEM/CIF/Adobe Commerce load testing:

- Split your script into configurable scenarios, which should cover, for example:
 - › Open Home Page
 - › Open Category Page (PLP)
 - › View Simple Products (PDP) – 2 loops within each iteration
 - › View Configurable Products – 2 loops within each iteration
 - E.g. set the steps above to 60% of traffic
 - › Product Search
 - E.g. set searching the catalogue to 37% of traffic
 - › Cart and Checkout
 - E.g. a user completing the Cart and checkout part of the script should default to an industry standard ecommerce site conversion rate of around 3%
 - As the checkout flow is uncached and usually a resource intensive operation, setting an unrealistically high figure for the numbers of people completing orders vs. the number of site browsers would give an unreliable result for the volume of traffic your site could handle.



- Clean all caches before each test run:
 - › The AEM dispatcher cache should be fully cleaned
 - › Adobe Commerce's Fastly and internal cache should be fully flushed and cleaned – this can be done via the cache control in Adobe Commerce admin.
- Include a ramp period in Jmeter test: Having no ramp period set means no gradual ramp up of traffic and no chance for the site to cache any of the commonly visited pages and components of the page. In real life, it would be unusual for all peak traffic to arrive on a fully uncached site at exactly the same time, therefore a ramp period should be included in Jmeter test scripts to allow the cache to build up as would happen on a real ecommerce site.
- A "Wait time" between each step within an iteration should be used – in reality, a user would not immediately jump to the next page on the site during their journey – there would be a wait time whilst the user read the page and decided on their next action.
- Setting the thread groups to loop infinitely, but for a set time of x (e.g. 60 minutes), will give a repeatable test, with median response times to comparable against previous test runs. This means that after the set ramp up period, there will be the target number of Virtual Users running concurrently and this will continue for the set loop time.
- Median time should be used to give an improvement /decline in average response time, not average. If there are several edge results that take a lot longer than the other results then this would skew this average result, but what we are interested in is the end user response time for the majority of users, which is more suited to the median measure.
- Embedded resources are not collected by default in jmeter (e.g. JS, CSS and other resources downloaded when a real user visits page). This can be enabled, but only for the domain you are testing –external resource calls should still be excluded (e.g. we don't want to include response times from externally hosted services, eg. google analytics code, as we have no control over them).
- HTTP Cache Manager should be enabled, this enables Jmeter to cache page elements during a journey as a real user's journey would during their browsing of the website on their own browser. During their journey though the site, the user's browser would download the related embedded resources only once and then these would be cached by the user's browser. Also, if the same user returns to the site some time after their original visit then it could still be the cache that those assets are cached.
- Listeners should be kept within the actual load test runs (e.g. "View Results Tree" and "Aggregate Report"). Including this in the non-GUI real load test run can impact on the performance results being reported by Jmeter, as resources are used during the real test run to generate the reports. These listeners were removed from the test script to be replaced with a JTL results file, which can then be processed using Jmeter's Report Dashboard functionality.
- A target response time for evaluated so that the dashboard report's "Apdex score" can be used as a quick way to measure the effect of changes on performance between test runs. The Apdex score is based on a certain amount of people being able to access the site in a tolerable time . If the response time is over a certain "frustrating" amount, this lowers the score. The times can be set using the "apdex_satisfied_threshold" and "apdex_tolerated_threshold" parameters.
- Set a target "Orders per hour" metric to present to business users, not a Virtual User count. "Virtual users" can be a complex topic to understand what in real life the test is measuring. By calculating the site conversion rate, orders per hour, average time a user spends on the site and think time in between each page load, industry standard calculations can be used to present different load test scenarios based on orders per hour to be achieved.

- Finally, your Jmeter test server should be run on a server geographically close to where the majority of your user traffic is coming from and where your cloud infrastructure is hosted – hopefully these would be the same.

Conclusion

The above documentation is intended to be a high-level guide to provide some areas to investigate initially while preparing your AEM/CIF/Adobe Commerce environment for heavy load. There are many areas of Adobe Commerce, CIF, and Fastly that are not covered by the above guide which will require optimisations specific to your environment. Also, custom code and 3rd party modules may have effects on response times which cannot be covered here.

To mitigate the risk of changes or code being introduced that will negatively affect the AEM/CIF/Adobe Commerce end user response times, it should be ensured that a thorough and repeatable performance test strategy is put in place, including defining KPI's that can be measured over time. Performance testing should be carried out not only prior to go live but also continuing after go-live, ideally before each major release or change to your production environments to measure any performance impacts.



© 2021 Adobe. All rights reserved.

Adobe and Adobe logo are either registered trademarks or trademarks of Adobe in the United States and / or other countries.

