Liferay
Clustering Deck

Must-Know Concepts, Explained

Coverage	Core Concepts, Database & File Store Prerequisites, Cluster Link & Cache Replication, Search Engine Clustering, Load Balancing & Sessions, Scheduler & Background Jobs, Cloud-Native & Containerized Clustering, Troubleshooting & Best Practices
Format	Question → Concise answer → configuration example where useful
Level	Liferay Infrastructure/DevOps Engineer to Solution Architect rounds

---

Reflects Liferay DXP 7.4+ / current Liferay DXP Cloud (quarterly release) clustering architecture. Configuration property names and defaults can shift between releases – always confirm against the release notes for your specific DXP version.

Generated September 20, 2026

1 Core Clustering Concepts 2

2 Database & File Store Prerequisites 4

3 Cluster Link & Cache Replication 6

4 Search Engine Clustering 8

5 Load Balancing & Session Management 10

6 Scheduler & Background Jobs 12

7 Cloud-Native & Containerized Clustering 14

8 Troubleshooting & Best Practices 16

Core Clustering Concepts

Q1.1

What does it mean to cluster a Liferay DXP deployment, and what two distinct goals – high availability and horizontal scalability – does clustering typically serve at the same time?

Clustering runs multiple independent Liferay DXP application server nodes, all pointed at the same shared backing services (database, file store, search engine), collectively serving the same site(s) as if they were one logical system. This simultaneously serves two related but distinct goals: **high availability** (if one node crashes or needs to be taken down for maintenance, the remaining nodes keep serving traffic, avoiding a single point of failure) and **horizontal scalability** (spreading incoming request load across multiple nodes lets the overall deployment handle more concurrent traffic than any single node could alone). A production Liferay cluster is almost always built to serve both goals together, since the same multi-node architecture that provides failover protection also naturally provides the extra request-handling capacity of running more than one server.

Q1.2

What are the five essential components every Liferay DXP cluster requires, at a minimum, for multiple nodes to function correctly as one coherent system?

- A **load balancer** distributing incoming requests across the available nodes.
- A **database** (or database cluster) accessible to, and shared by, every node – never an embedded database.
- A **file store** accessible to every node, for Document Library content.
- A **search engine** (Elasticsearch, running as a process external to DXP itself) accessible to every node, optionally clustered in its own right.
- **Cluster Link** configured on every node, enabling inter-node communication and cache replication.

Skipping or misconfiguring any one of these five leaves the deployment looking superficially like a cluster (multiple running nodes) while actually behaving inconsistently – e.g. different nodes serving stale cached data, or seeing different uploaded files – which is precisely why each is treated as a genuinely required, non-optional prerequisite rather than an optional tuning knob.

Q1.3

Why can't a Liferay cluster simply run each node as a fully independent, self-contained instance with its own local database and file storage, the way you might horizontally scale a stateless web service?

Unlike a purely stateless web service (where each request can, in principle, be handled by any instance with zero shared state at all), Liferay is a stateful content and application platform – the data a user creates or reads (a page edit, an uploaded document, an Objects entry) must be immediately visible to *every* other node, regardless of which specific node happens to handle a given user's next request. If each node had its own independent local database and file storage, a user's edit made via node A would simply be invisible to a subsequent request that happened to land on node B – an obviously broken, inconsistent experience. This is precisely why shared, common backing services (database, file store, search index) are non-negotiable architectural requirements for a Liferay cluster, not a matter of preference or optimization.

Q1.4

What is the difference between vertical scaling (a bigger single server) and horizontal scaling (clustering more servers) as two different strategies for handling a Liferay deployment's growing load, and why does clustering typically become necessary eventually regardless of how large a single server is made?

Vertical scaling increases a single server's own resources (more CPU cores, more RAM) – simple to implement (no clustering complexity at all) but bounded by the largest single machine practically available, and offers zero protection against that one machine failing entirely. **Horizontal scaling** (clustering) instead runs several separate, smaller servers together – more complex to set up and operate correctly (all five prerequisites above must be properly configured), but has no comparable hard ceiling on total capacity, and directly provides the failover protection vertical scaling alone cannot offer. Most real production Liferay deployments eventually need horizontal scaling specifically for this failover/high-availability property alone, even in cases where a single, sufficiently large vertically-scaled server could theoretically still handle the actual request volume – since no single server, however large, can ever protect against its own hardware or OS-level failure.

Q1.5

What is the general recommended approach for learning and validating Liferay clustering concepts before attempting a genuine production deployment, and why does Liferay's own documentation explicitly caution against using a simple local Docker-based cluster setup for production?

Liferay's own documentation recommends starting with a simple, fast-to-set-up two-node cluster running via Docker containers on a single machine – standing up two DXP server containers alongside containers for a database, search engine, and file store – specifically as a learning and validation exercise to understand how the pieces fit together and interact, without the overhead of a genuine multi-machine production environment. This same documentation explicitly notes that such a setup is intended purely **for learning purposes** and is *not* suitable for actual production use – a genuine production deployment additionally needs a real, dedicated HTTP load balancer, properly separated (and likely clustered/replicated) database servers with read/write splitting considered, and a genuinely clustered, multi-node file store and search engine, none of which a simple single-machine Docker learning setup is designed to provide.

Database & File Store Prerequisites

Q2.1

Why must every node in a Liferay cluster be configured to use the exact same database (or database cluster), and why is Liferay's built-in embedded/bundled database explicitly unusable in a clustered deployment?

Every piece of core Liferay data – users, sites, pages, permissions, and (as covered in the companion Liferay Objects deck) custom object entries – lives in the relational database, so every node must read and write against the *identical* shared database for all nodes to present a single, consistent view of that data to users regardless of which node handles their request. Liferay's bundled embedded database (used out of the box for quick evaluation and local development) runs as an in-process database tied to that one specific server instance, with no mechanism for a separate node to connect to or share it – making it fundamentally incompatible with clustering by its very nature, which is precisely why every Liferay clustering guide's first, non-negotiable prerequisite is migrating to a genuine, separately-hosted, network-accessible database (PostgreSQL, MySQL, Oracle, or another supported database) before clustering is even attempted.

Q2.2

What is the read/write database-splitting optimization sometimes used in larger Liferay cluster deployments, and what specific kind of workload does it help address?

Rather than every node sending both read and write database queries to one single database instance, a read/write-split architecture directs **write** operations to a primary database instance and distributes **read** operations across one or more separate, continuously-synchronized read-replica instances – reducing load on the primary instance and letting read capacity scale somewhat independently of write capacity. This optimization is specifically valuable for larger, read-heavy deployments (a high-traffic public website with comparatively infrequent content updates relative to its page-view volume) where the database's read load, rather than its write load, is the actual bottleneck – it adds real architectural complexity (the application and/or database layer must correctly route each specific query to the right instance, and replication lag between primary and replicas must be accounted for) and is therefore generally reserved for deployments that have a demonstrated, genuine need for it, rather than being a default recommendation for every cluster.

Q2.3

What is the shared file store requirement in a Liferay cluster, and what problem arises if two nodes are instead each configured to store uploaded Document Library content on their own separate local disk?

Liferay's Document Library stores uploaded files (documents, images, and other binary content) through a configurable **file store** abstraction – and in a cluster, this file store must be a location genuinely accessible to *every* node, exactly like the shared database. If each node instead independently wrote uploaded files to its own local disk, a file uploaded via a request that happened to land on node A would simply not exist on node B's local disk at all – so a subsequent request for that same file, if routed to node B, would fail to find it, an inconsistency directly analogous to (and just as broken as) the independent-per-node-database problem discussed in Section 1. Common shared file store options include a network-attached shared filesystem (NFS), Liferay's own Advanced File System Store pointed at shared network storage, or a cloud object storage service (like S3), any of which genuinely provide every node with access to the exact same underlying file content.

Q2.4

Where is the file store's specific configuration set, and what is the `FileSystemStoreConfiguration` OSGi configuration responsible for?

The file store implementation and its settings are configured through Liferay's OSGi configuration system, specifically via `com.liferay.portal.store.file.system.configuration.FileSystemStoreConfiguration` (for the file-system-based store) – specifying, among other settings, the root directory path where files should be stored. In a cluster, this configuration must be set identically on every node, with that configured path pointing to the same genuinely shared network location (rather than each node's own distinct local path) – a configuration mismatch here, where nodes are each configured with a superficially similar-looking but actually different local path, is a common and easy-to-overlook source of the exact file-inconsistency symptoms described in the previous question.

Q2.5

Why does a cluster's shared file store need to be considered for backup and disaster-recovery planning separately from the database's own backup strategy, rather than assuming database backups alone are sufficient?

Uploaded Document Library content is stored as actual binary files in the shared file store, with the database instead holding only *metadata* about those files (their names, folder structure, permissions, version history) – backing up the database alone captures none of the actual file content itself, only the records describing where that content is supposed to be and what it's called. A complete, restorable backup and disaster-recovery strategy for a Liferay cluster must therefore explicitly include the shared file store's own actual content as a separate, coordinated backup target alongside the database backup – and, critically, a restore must bring both the database and file store back into a *mutually consistent* state together (restoring one without the other, or restoring them from mismatched points in time, leaves the database referencing files that either don't exist or don't match what the metadata expects).

Cluster Link & Cache Replication

Q3.1

What is Cluster Link, and what specific problem does it solve that simply sharing a common database does not, on its own, already address?

Even with every node correctly sharing the same database, file store, and search engine, each node still maintains its own local, in-memory **cache** of frequently-accessed data (for performance – reading from an in-memory cache is far faster than a fresh database query every time). Without a way to keep these independent local caches synchronized, a change made on node A (updating a page's content, say) would correctly hit the shared database, but node B's own stale, unaware local cache might continue serving the *old*, pre-change version of that same data to its own users, unaware anything changed. **Cluster Link** is Liferay's mechanism for solving exactly this: it enables inter-node communication specifically so that a cache-invalidating change on one node is broadcast to and correctly invalidates the corresponding cached data on every other node in the cluster, keeping each node's local cache consistent with the shared, authoritative source of truth.

Q3.2

What is JGroups, and what role does it play underneath Liferay's Cluster Link feature?

JGroups is a mature, widely-used open-source Java toolkit specifically for building reliable, group-communication-based clustering functionality – providing the underlying mechanisms for node discovery (nodes finding and recognizing each other as members of the same cluster), reliable message delivery between cluster members, and handling nodes dynamically joining or leaving the group over time. Liferay's Cluster Link feature is built directly on top of JGroups, using it as the actual transport and group-membership layer – when you configure Cluster Link's specific communication settings (covered in the next questions), you are, underneath, directly configuring JGroups' own channel configuration, making a working understanding of JGroups' basic concepts genuinely useful background for correctly troubleshooting Cluster Link issues.

Q3.3

What is the difference between the `cluster.link.channel.properties.control` and `cluster.link.channel.properties.transport.0` configuration properties, and what distinct kind of traffic does each one's JGroups channel carry?

Cluster Link uses two logically separate JGroups channels, each configured by its own property: `cluster.link.channel.properties.control` defines the channel specifically used for **cluster management and coordination** messages – node discovery, heartbeats, and membership changes (a node joining or leaving). `cluster.link.channel.properties.transport.0` defines the channel used for **cache replication and cluster data** messages – the actual cache-invalidation notifications discussed above, propagating a data change from the node where it occurred out to every other node. In most real deployments, both properties are configured to point at the same underlying JGroups configuration file (commonly `tcp.xml`), letting both categories of traffic share the same basic transport configuration – though they remain conceptually and configurably distinct channels, and could, in principle, be tuned or routed differently if a specific deployment's needs called for it.

Q3.4

What is the difference between configuring Cluster Link with UDP Multicast versus TCP Unicast, and in what specific network scenario does Liferay's own documentation recommend switching from the (often simpler-to-configure) multicast default to unicast instead?

UDP Multicast lets nodes automatically discover each other by broadcasting to a shared multicast group address,

without needing each node to know every other node's specific address in advance – convenient and often simpler to initially configure, but it depends on the underlying network infrastructure actually supporting and correctly routing multicast traffic, which many real-world network environments (especially cloud environments, and deployments spanning nodes across a significant geographical distance) either don't support at all or actively block. **TCP Unicast** instead has each node communicate via ordinary point-to-point TCP connections to an explicitly configured list of other node addresses – more configuration-heavy (every node's address must be known and listed), but works reliably in network environments where multicast isn't available or practical. Liferay's documentation specifically recommends switching to TCP Unicast precisely when your network configuration, or the geographical distance between cluster nodes, prevents reliable use of UDP Multicast.

Q3.5

Why is it considered a serious, deployment-breaking misconfiguration to run multiple Liferay nodes against a shared database without also properly configuring Cluster Link, rather than simply a performance sub-optimization to fix later?

Running nodes against a correctly shared database but *without* Cluster Link properly configured doesn't merely leave some minor performance optimization on the table – it produces a deployment that appears superficially functional (multiple nodes running, all pointed at the same real database) while actually serving **inconsistent, stale data** to different users depending purely on the arbitrary happenstance of which node a given request lands on, exactly the cache-staleness problem described above. This class of bug is particularly insidious precisely because it's often intermittent and hard to reproduce reliably in casual testing (it depends on the specific timing of when a change is made versus when a subsequent read happens to hit a different, not-yet-invalidated node's cache) – which is exactly why Liferay's own documentation and community guidance consistently treat Cluster Link as an absolutely mandatory, non-optional component of any multi-node deployment, never as an optional performance tuning step to be deferred or skipped.

Search Engine Clustering

Q4.1

Why does a Liferay cluster require Elasticsearch to run as a separate process, external to the DXP application server nodes themselves, rather than running an embedded search index within each individual node?

Liferay's search functionality is built on Elasticsearch, and in any real cluster deployment, Elasticsearch runs as its own separate, independently-managed process (or cluster of processes) rather than being embedded inside each Liferay node – exactly mirroring the shared-database and shared-file-store requirements already discussed. If each node instead maintained its own independent, embedded search index, the same fundamental consistency problem would recur once more: content indexed via a request handled by node A wouldn't be searchable by a user whose subsequent search request happened to land on node B's separate, unaware index. Running one shared (and, in production, typically separately-clustered for its own high availability) Elasticsearch deployment that every Liferay node queries against ensures every node sees the exact same, consistent search index, regardless of which node originally triggered the content being indexed.

Q4.2

Can Elasticsearch itself be clustered (running multiple Elasticsearch nodes), and how does this relate to, but remain conceptually distinct from, the Liferay DXP application-server clustering discussed throughout the rest of this deck?

Yes – Elasticsearch fully supports its own independent clustering, running multiple Elasticsearch nodes together for its own high availability and search-query capacity, entirely analogous in spirit to Liferay DXP's own application-server clustering, but as a genuinely separate, independently-managed clustering concern with its own configuration, its own node-discovery mechanisms, and its own operational considerations. In a production deployment, you're typically managing *two* conceptually distinct clusters simultaneously – a Liferay DXP application-server cluster (the primary focus of this deck) and a separate Elasticsearch cluster it depends on – and it's a common, explicitly-warned-against mistake to accidentally end up running what looks like "two Elasticsearch clusters" (e.g. through a cluster-name mismatch between nodes) rather than one genuine, properly joined Elasticsearch cluster backing the entire Liferay deployment.

Q4.3

What is Cross Cluster Replication (CCR), and what specific multi-data-center deployment scenario does it address that a single, geographically centralized Elasticsearch cluster cannot?

Cross Cluster Replication lets a "follower" Elasticsearch cluster, located at one data center, maintain a continuously-updated, read-only replicated copy of indexes originating from a "leader" Elasticsearch cluster located at a different, geographically separate data center – specifically enabling an architecture where Liferay DXP nodes at each separate data center can perform fast, **local** search reads against their own co-located follower cluster (avoiding the latency penalty of every search query crossing a long-distance network link to a single, centrally-located cluster), while writes still flow, ultimately, to the single authoritative leader cluster and are then automatically, continuously replicated back out to every follower. This is specifically valuable for genuinely geographically-distributed, multi-data-center Liferay deployments (as opposed to a single-region cluster, where CCR's added complexity generally isn't needed at all) where minimizing search-read latency for geographically distant users is a real, demonstrated requirement.

Q4.4

Why does upgrading Elasticsearch independently of a Liferay upgrade generally carry a lower operational risk (typically just requiring a backup, with no restore needed) compared to the scenario of upgrading Liferay and simultaneously switching it to connect to a genuinely different, new Elasticsearch cluster?

When Elasticsearch is upgraded on its own, independent of any Liferay version change, all of the existing indexes remain intact and in place throughout the upgrade – backing up the indexes beforehand is still recommended as a standard operational safety precaution, but actually restoring from that backup generally isn't needed afterward, since the upgrade process itself doesn't discard or require rebuilding the existing index data. The considerably higher-risk scenario is specifically when a Liferay upgrade is combined with connecting to an entirely *different* Elasticsearch cluster (a fresh cluster with no existing indexes at all) – in that case, a full restore from backup *is* necessary for all primary storage indexes, and if certain search-tuning features (like Result Ranking or Synonym Sets) are in use, their configuration data must additionally be separately re-imported into the Liferay database itself after the upgrade, since that specific tuning data lives in the database rather than purely within the search index.

Load Balancing & Session Management

Q5.1

What role does a load balancer (such as Apache HTTP Server or HAProxy) play as the entry point to a Liferay cluster, and why do end users and external systems typically never connect directly to any individual DXP node?

A load balancer sits in front of the entire cluster as the single, unified entry point external traffic actually connects to – distributing each incoming request across the available healthy DXP nodes according to some configured balancing strategy, and, critically, also monitoring node health so that a node which has crashed or become unresponsive is automatically and transparently removed from rotation until it recovers. End users and external integrations are configured to connect only to the load balancer's own address, never to any individual node directly, specifically so that the addition, removal, or temporary failure of any single node behind the load balancer remains completely invisible to those external clients – they simply keep talking to the same stable load-balancer address throughout, regardless of how the specific set of healthy nodes behind it happens to change over time.

Q5.2

What are "sticky sessions," and why does a Liferay cluster commonly configure the load balancer to route a given user's requests to the same specific node throughout their session, rather than freely load-balancing each individual request?

Sticky sessions (session affinity) configure the load balancer to consistently route all of a given user's requests, throughout their entire session, to the exact same backend node they were originally routed to – rather than potentially routing each individual request to a different, arbitrarily-chosen node. This is commonly configured because a user's in-progress session state (their authenticated login, and various other transient, session-scoped data) is, by default, held in that one specific node's own local memory – without sticky sessions, a user's subsequent request landing on a *different* node (one that never saw their original login) could unexpectedly find itself unauthenticated or otherwise missing session context the node handling their previous request already had. Sticky sessions sidestep this problem entirely by simply always routing a given user back to the one node that already holds their session state.

Q5.3

What is session replication, and how does it provide session-state continuity for a user even if the specific node their sticky session was pinned to crashes mid-session – a scenario sticky sessions alone cannot protect against?

While sticky sessions ensure a user is *normally* routed consistently back to one node, they offer no protection at all if that specific node actually crashes or is taken down mid-session – the user's session state, held only in that one node's local memory, would simply be lost entirely, forcing them to log in again. **Session replication** (supported by Liferay, using Tomcat's own session replication capability, from DXP 2024.Q3+/Portal GA92+) addresses this by synchronizing session data *across* all nodes in the cluster – so if a user's originally-assigned node fails, another node already holds an up-to-date copy of their session state and can seamlessly continue serving them, without the user needing to log in again or losing their in-progress session context, at the cost of the additional network and memory overhead required to keep every node's session data continuously synchronized with every other node.

Q5.4

What is the trade-off between relying purely on sticky sessions (without full session replication) versus enabling full session replication across the cluster, in terms of operational complexity versus resilience?

Sticky sessions *alone*, without session replication, are simpler to configure and impose no ongoing network/memory overhead for keeping every node's session data synchronized – but they leave a real, uncovered gap: a node failure genuinely disrupts every user session that happened to be pinned to that specific node, forcing those users to re-authenticate. Enabling full session replication closes that gap, providing genuinely seamless failover for user sessions even during a node failure, but at the cost of additional configuration complexity and continuous, ongoing overhead as every node keeps its session data synchronized with every other node throughout normal operation. Many real deployments make a deliberate, considered trade-off here based on their specific tolerance for occasional forced re-authentication during a rare node failure versus their willingness to accept the added operational overhead and complexity of full session replication – there's no single universally correct answer independent of a given deployment's own specific requirements.

Q5.5

Why must SSL/TLS termination and the overall network path between the load balancer and the individual DXP nodes be carefully configured, particularly regarding how the original client's actual IP address and protocol are communicated onward to the application layer?

When a load balancer terminates SSL/TLS itself (decrypting incoming HTTPS traffic before forwarding plain HTTP requests on to the backend DXP nodes), the DXP application itself would, without further configuration, only ever see the load balancer's own IP address as the apparent source of every request, and would see every request arriving as plain HTTP rather than the original HTTPS – both of which can cause real, concrete problems: incorrect IP-based audit logging or access-control decisions, and Liferay incorrectly believing it's being accessed over an insecure, non-HTTPS connection when generating absolute URLs or setting security-sensitive cookie attributes. This is why load balancer configurations must correctly forward this information onward using standard HTTP headers (commonly `X-Forwarded-For` for the original client IP, and `X-Forwarded-Proto` for the original protocol), with Liferay's own server configuration correspondingly set up to trust and correctly interpret those specific forwarded headers from the load balancer.

Scheduler & Background Jobs in a Cluster

Q6.1

Why does running Liferay's scheduled/background jobs (such as a nightly cleanup task) naively on every node in a cluster risk that job running multiple times redundantly, rather than exactly once as intended?

Liferay includes a built-in scheduler for running background jobs on a defined schedule – but in a cluster, every single node is, by default, an independently-running Liferay instance with its own active scheduler component, all pointed at the same shared configuration. Without specific cluster-aware handling, a scheduled job configured to run "every night at 2 AM" would naively trigger on *every* node simultaneously at that time, potentially causing the same cleanup, notification, or data-processing logic to run redundantly multiple times in parallel – at best wasting resources, and at worst causing genuine correctness problems for jobs that aren't safely idempotent when run concurrently (e.g. a job that sends a notification email, which you'd very much not want sent once per cluster node).

Q6.2

How does Liferay's clustered scheduler avoid the duplicate-job-execution problem, and what general mechanism (a cluster-wide "master" designation) does it use to ensure only one node actually runs a given scheduled job?

Liferay's scheduler is cluster-aware precisely to address this: rather than every node independently executing every scheduled job, the cluster elects (via Cluster Link's underlying coordination mechanism) one node to act as the scheduler "master" for a given point in time, and only that currently-designated master node actually triggers and executes the scheduled jobs – the other nodes remain aware the jobs exist and are scheduled, but defer actual execution to whichever node currently holds the master role. If the current master node fails or is taken down, the cluster automatically elects a new master node from among the remaining healthy nodes, ensuring scheduled jobs continue running exactly once on schedule even through a node failure, without ever falling back to the naive "every node runs everything" behavior that would cause duplicate execution.

Q6.3

Why does correct scheduled-job behavior in a cluster depend directly on Cluster Link being properly configured, tying this topic directly back to Section 3's coverage of cluster coordination?

The master-node election and coordination mechanism underlying clustered scheduling relies on the exact same inter-node communication and coordination infrastructure that Cluster Link provides more broadly (Section 3) – nodes need to reliably discover each other, agree on which one currently holds the master role, and correctly detect when a master node has failed and a new election is needed, all of which depends on Cluster Link's control-channel communication functioning correctly. This means a Cluster Link misconfiguration doesn't just risk the cache-staleness symptoms discussed in Section 3 – it can *also* directly cause the duplicate (or, in some failure modes, entirely missed) scheduled-job-execution problems discussed in this section, reinforcing why Cluster Link is treated as such a foundational, cross-cutting prerequisite underlying correct cluster behavior across many different subsystems, not merely a caching-specific concern in isolation.

Q6.4

What kind of custom background logic (a custom scheduled job implemented via a client extension or module, for instance) requires particular developer attention to ensure it behaves correctly when deployed into a clustered environment, rather than assuming Liferay's

built-in cluster-awareness automatically covers it?

Liferay's own built-in scheduled jobs correctly use the cluster-aware master-election mechanism described above automatically – but a *custom* scheduled job a developer implements themselves (whether as a traditional module or via an appropriate client extension type, as covered in the companion Liferay Client Extensions deck) must be deliberately written to use Liferay's own scheduler API correctly, rather than, for instance, naively using a plain Java/OS-level cron-style mechanism that has no awareness of the cluster at all and would trigger independently on every node. This is an important, easy-to-overlook detail for developers building custom background/scheduled functionality on Liferay – correctly integrating with Liferay's own cluster-aware scheduler API, rather than reaching for a simpler-seeming but cluster-unaware scheduling mechanism, is essential for the same custom logic to behave correctly (running exactly once, cluster-wide) once deployed into a genuinely clustered production environment.

Cloud-Native & Containerized Clustering

Q7.1

What is Liferay's Cloud Native Experience (CNE), and how does it package the same underlying clustering architecture discussed throughout this deck specifically for deployment on Kubernetes?

Cloud Native Experience is Liferay's officially supported approach and tooling for deploying and operating Liferay DXP on Kubernetes – packaging DXP nodes, along with the supporting services they depend on (a search engine, and optionally a database), as Kubernetes-native resources, letting Kubernetes' own orchestration capabilities (automatic pod restart on failure, horizontal pod scaling, rolling updates) manage much of what would otherwise require manual configuration and operational tooling in a traditional, non-containerized cluster deployment. Critically, CNE doesn't replace or bypass the fundamental clustering requirements discussed throughout this deck (a shared database, shared file store, Cluster Link, a clustered search engine) – it instead provides a standardized, Kubernetes-native way of provisioning and wiring up those same essential components together, rather than introducing some entirely different clustering model unique to Kubernetes.

Q7.2

What is a k3d-based Liferay quickstart environment, and why is it explicitly positioned as a local development/validation tool rather than a production deployment pattern, echoing the Docker-based learning setup discussed in Section 1?

k3d runs a lightweight, local Kubernetes cluster (built on the k3s distribution) directly inside Docker containers on a developer's own machine, and Liferay provides a CNE-based quickstart specifically for spinning up a local Liferay deployment on such a k3d cluster, complete with ephemeral (non-persistent, throwaway) PostgreSQL and Elasticsearch instances running alongside it. Exactly like the simple Docker-based two-node cluster discussed in Section 1, this k3d quickstart is explicitly intended for local development, learning, and validating Kubernetes-based deployment configurations *before* deploying to a genuine, persistent, production-grade Kubernetes cluster – the ephemeral nature of its database and search engine (data is lost when the local environment is torn down) makes it fundamentally unsuitable, and not intended, for any actual production workload.

Q7.3

Why does running Liferay on Kubernetes still require exactly the same shared database, shared file store, and clustered search engine prerequisites discussed throughout this deck, rather than Kubernetes' own native capabilities somehow eliminating the need for them?

Kubernetes excels at orchestrating the lifecycle of **stateless or ephemeral** workloads – automatically restarting a crashed pod, scaling the number of running pod replicas up or down, and rolling out updates with minimal disruption – but it does not, on its own, solve the fundamentally different problem of *stateful data consistency* across multiple replicas of a genuinely stateful application like Liferay. Kubernetes can readily ensure that, say, three Liferay pod replicas are always running and healthy, automatically replacing any that crash – but it has no inherent knowledge of, or mechanism for, ensuring those three replicas all see the same database records, the same uploaded files, or the same search index, unless they're *explicitly configured*, exactly as in a traditional non-containerized cluster, to all connect to the same shared database, file store, and search engine – Kubernetes changes *how* a cluster's nodes are provisioned and orchestrated, not the fundamental architectural requirements for those nodes to behave consistently together as one coherent cluster.

Q7.4

How does a Kubernetes Service and Ingress resource, in a CNE deployment, functionally correspond to the traditional load balancer's role discussed in Section 5?

In a traditional, non-containerized Liferay cluster, a dedicated load balancer (Apache HTTP Server, HAProxy) sits in front of the individually-running DXP node processes, distributing traffic and monitoring node health. In a Kubernetes-based CNE deployment, this same functional role is instead fulfilled by Kubernetes-native resources – a **Service** providing a stable, single network identity that automatically load-balances traffic across whichever set of healthy DXP pods currently match its label selector (directly analogous to the load balancer distributing traffic across healthy backend nodes), and an **Ingress** resource typically handling the external-facing routing and, often, TLS termination – Kubernetes' own built-in health-checking and pod-readiness mechanisms take over the node-health-monitoring role a traditional load balancer would otherwise need to be separately configured to perform.

Troubleshooting & Best Practices

Q8.1

If different users on a live cluster report seeing inconsistent or ”flickering” content – the same page appearing to change depending on when they refresh – what is the first, most likely misconfiguration category to investigate?

This specific symptom – content that appears different or seemingly reverts depending on exactly when a page is loaded – is the classic, textbook signature of a **Cluster Link** misconfiguration (Section 3): different nodes are each serving from their own locally-cached version of the data, with cache-invalidation messages either not being sent, not being received, or not being correctly processed between nodes, and the user's browser happening to land on a different node (with a different, not-yet-invalidated cached state) on each successive request or reload. The first, most productive troubleshooting steps are confirming that every node's Cluster Link configuration (both the control channel and transport channel properties) is genuinely, identically correct, and that the underlying JGroups transport (multicast or unicast, per Section 3) is actually able to successfully communicate between every pair of nodes given the real network environment they're actually running in.

Q8.2

Why is it considered good practice to deliberately test a Liferay cluster's failover behavior – actually killing or stopping a node mid-operation during a controlled test – rather than only validating that the cluster works correctly under normal, all-nodes-healthy conditions?

Validating that a cluster correctly serves traffic when every node is healthy confirms the basic configuration is functionally correct, but says nothing at all about whether the cluster's *failure-handling* behavior – the load balancer correctly detecting and routing around a failed node, session replication (or the accepted sticky-session trade-off) actually behaving as expected during a real failure, and the scheduler's master-election mechanism (Section 6) correctly failing over to a new master node – genuinely works as intended, since that failure-handling logic, by its very nature, is never actually exercised at all during normal, everything-healthy operation. Deliberately, proactively testing failover in a controlled, non-production environment (or during a planned, communicated maintenance window) – actually stopping a node and observing exactly what happens to in-flight user sessions, scheduled jobs, and overall traffic handling – is the only way to genuinely validate that a cluster's failure-handling behavior matches what was actually intended and configured, rather than discovering a gap in that behavior for the first time during a real, unplanned production incident.

Q8.3

Why is documenting and consistently maintaining configuration parity across every node in a cluster – ensuring every node runs the identical Liferay version, the identical set of deployed modules/client extensions, and identical relevant configuration – such a critical, foundational operational discipline?

A cluster is fundamentally designed around the assumption that any given user request could, in principle, be served by *any* of the available nodes with an identical result – an assumption that silently breaks down the moment nodes actually drift out of sync with each other (one node running a slightly different Liferay patch version, one node missing a client extension another node has deployed, or one node's configuration property set to a different value than its peers). Such drift produces exactly the same kind of confusing, hard-to-reproduce, node-dependent inconsistent-behavior symptoms discussed throughout this deck (particularly Section 3's cache-staleness scenario), but stemming from an entirely different root cause (genuine configuration/deployment drift, rather than a cache-invalidation failure) – making rigorous, disciplined configuration management (ideally through the same Configuration as Code and automated deployment-pipeline practices discussed in the companion Liferay Client Extensions deck, ensuring every

node is provisioned identically and reproducibly rather than manually, individually configured by hand) an essential, foundational operational discipline for any cluster that needs to remain reliably, predictably consistent over its full operational lifetime.

Q8.4

What monitoring and observability practices are particularly important specifically for a clustered Liferay deployment, beyond the kind of basic single-server monitoring (CPU, memory, disk) any production system would already need?

Beyond standard single-server resource monitoring (which remains necessary on every individual node), a clustered deployment specifically benefits from monitoring that spans and correlates *across* the whole cluster: tracking Cluster Link's own health and message-delivery latency (a degrading or increasingly-delayed Cluster Link can produce the cache-staleness symptoms discussed above well before it fails outright and becomes obviously, unambiguously broken), monitoring the shared Elasticsearch cluster's own health and replication status independently of any individual DXP node's health, monitoring the load balancer's own view of which nodes it currently considers healthy versus unhealthy (which may not always match what each node itself believes about its own health), and, where session replication is enabled, monitoring the actual overhead and latency that ongoing session synchronization is imposing across the cluster – collectively, this cluster-spanning observability is what actually lets an operations team distinguish "one node is having a problem" from "the cluster's own coordination and consistency mechanisms are themselves degrading," two very different classes of problem that can otherwise look deceptively similar from a single node's own local monitoring perspective alone.

End of deck – see the companion Liferay Objects and Liferay Client Extensions decks for the application layer running on top of this infrastructure.