BlueSpice MediaWiki

BlueSpice

At Enterprise Scale

Hallo Welt! GmbH | Bruderwöhrdstraße 29 | 93055 Regensburg | +49 941 660 80 0

Executive Summary

This is a concept paper intended to demonstrate that large-scale instances are feasible and to provide an overview of how to approach them strategically. BlueSpice Farm is designed from the ground up as a multi-wiki platform, and it is a viable architecture for deployments at the 10,000 sub-wiki scale described in this document. This summary answers the three operational questions that typically dominate any enterprise wiki decision at this scale.

Can a Farm of this size be maintained without proportional growth in Ops headcount?

The Farm model is specifically designed to avoid that outcome. Every sub-wiki runs from a single shared application codebase, so maintenance effort scales with infrastructure capacity rather than wiki count. Tenant isolation is handled at the data tier through per-wiki databases, scoped search metadata, and namespace cache keys, while the application layer remains shared. New wiki provisioning completes within seconds and is a data operation, not an infrastructure operation, so adding the 10,000th wiki does not require an additional application instance or server build. Data-tier activities that do scale with wiki count, notably backup and schema migration, are addressed through orchestration tooling that enables parallelized backup pipelines and migration batching across tenants, with runtime sized to the available database cluster headroom.

Can platform updates be applied without touching every wiki individually?

Yes, through a coordinated path that operates once against the shared codebase. Updates are delivered as container images via a standard Helm and Kubernetes rolling deployment. Monthly PATCH updates produce zero downtime. MINOR updates follow a proven expand and contract pattern that keeps every wiki operational throughout the migration window, with no possibility of a codebase-to-database version mismatch causing failure. Tenant-aware rollout sequencing allows staged validation on representative sub-wiki cohorts before farm-wide deployment.

Is there an architectural ceiling on Farm size?

There is no hard ceiling in the architecture. BlueSpice is built on MediaWiki, which today serves approximately 24 billion page views per month across the Wikimedia Foundation's infrastructure, a traffic volume that dwarfs this deployment and demonstrates the shared-codebase Farm architecture performs under far more demanding load. Search operates on a shared OpenSearch cluster with horizontal data-node partitioning and per-tenant metadata scoping, sized to document volume rather than wiki count.

Bottom line

The platform and operational model are ready for a deployment of this scale. This document works through the architecture, scaling model, update mechanics, operational responsibility split, and risk posture in detail.

BlueSpice MediaWiki

Table of Contents

1. Introduction 5
   1.1 The concerns this document addresses 5
   1.2 Bottom line up front 5
2. Architecture Overview – How BlueSpice Farm Works 6
   2.1. What BlueSpice Farm Is 6
   2.2 Core Components 6
   2.3 How Sub-Wikis Are Provisioned and Isolated 7
   2.4 Dependency Stack 8
      2.4.1 WebSocket Routing 9
   2.5 Deployment Topology 10
3. Scaling to ~ 10,000 Sub-Wikis 11
   3.1 How the Farm Model Scales 11
   3.2 Shared vs. Per-Wiki Resources 11
   3.3 Database Architecture at Scale 12
   3.4 Caching Strategy 15
      3.4.1 Frontend Cache Layer 16
      3.4.2 WebSocket Connection Scaling 17
      3.4.3 Recommended approach for this deployment 18
   3.5 Search Infrastructure at Scale 18
   3.6 Application Pod Sizing 21
4. Update & Maintenance Model 22
   4.1 Update & Maintenance Model 22
   4.2 Update & Maintenance Model 23
   4.3 PATCH Updates – Procedure 24
   4.4 MINOR Updates – Expand/Contract Pattern 24
   4.5 MAJOR Updates 27
   4.6 Rollback Procedures 28
   4.7 Configuration Management 28
5. Operational Responsibility Split 30

BlueSpice MediaWiki

5.1 Responsibility Matrix 30
5.2 What the Ops Team Does Not Need to Do 34
5.3 Escalation Path 35
6. Operational Risk Assessment 36
6.1 Escalation Path 36
6.2 Single Points of Failure 40
6.3 Disaster Recovery Posture 42
6.4 Update Rollout Risk and Staged Deployment 44
6.5 Security Surface Area 45
6.6 Upgrade Failure Handling 46
7. Reference Points & Benchmarks 48
7.1 BlueSpice – Known Deployment Scale 48
7.2 MediaWiki at Wikipedia Scale 49
8. Open Questions / Gaps 50
8.1 Vendor roadmap 50
8.2 Customer-side Decisions Required 52
8.3 Architectural Gaps Requiring Further Design 53

Hallo Welt! GmbH | Bruderwöhrdstraße 29 | 93055 Regensburg | +49 941 660 80 0

1. Introduction

This document is a technical briefing, not a vendor sales document. It is written for the Lead Architect and the DevOps/Operations team evaluating the operational implications of running BlueSpice Farm at a scale of approximately 10,000 sub-wikis. It maps the terrain as accurately as available documentation and known reference deployments allow.

1.1 The concerns this document addresses

When an enterprise wiki platform is proposed at scale — specifically ~10,000 sub-wikis within a single BlueSpice Farm deployment — three questions dominate the operational conversation:

Maintenance burden: Does managing a Farm of this size require proportionally more Ops effort per wiki, or does the Farm model provide meaningful economy of scale?

Update complexity: When BlueSpice releases a new version, does applying that update mean touching 10,000 independent systems, or does the Farm architecture allow a single coordinated update path?

Scalability ceiling: Is there a known upper bound to how many sub-wikis a single Farm can support, and what does the path to that ceiling look like operationally?

1.2 Bottom line up front

BlueSpice Farm is architecturally designed so that the *codebase*— the PHP application, extensions, and skins — is **shared across all sub-wikis**. An update to the core platform is applied once to the shared *codebase*, not once per wiki. The per-wiki isolation is confined to **data** (separate databases or schema prefixes) and **configuration** (per-wiki "LocalSettings file"). This is the central operational characteristic the Ops team must understand: the maintenance surface does not scale linearly with the number of sub-wikis.

What the Ops team **is** responsible for scales with infrastructure, not with wiki count: Kubernetes and database cluster health, storage provisioning, cache layer operations, search cluster sizing, and the execution of deployments. The platform vendor (Hallo Welt! GmbH) provides the software update path; the Ops team owns the infrastructure on which that update runs.

This document works through the architecture, scaling model, update mechanics, operational responsibility split, and risk posture in detail. Open questions requiring direct clarification from BlueSpice are consolidated in Section 7.

2. Architecture Overview – How BlueSpice Farm Works

2.1. What BlueSpice Farm Is

BlueSpice Farm is a multi-wiki deployment architecture built on top of MediaWiki, the same open-source engine that powers Wikipedia. Where a standard BlueSpice installation serves a single wiki, a Farm deployment serves an arbitrary number of independent wikis — referred to as *sub-wikis*— from a **shared application codebase**.

The Farm model is not a multi-tenancy abstraction layered onto a single-wiki product. It is the native scaling architecture of MediaWiki: the core engine has supported Farm deployments since its early versions, and BlueSpice inherits and extends this capability. Each sub-wiki is a fully independent wiki with its own content, user permissions, configuration, and database (there are ways to share such things amongst the sub-wikis as well, but this will be covered in a dedicated section). What is shared is the PHP application layer — one codebase, one set of extensions, one set of skins — running against many data stores simultaneously.

This distinction is operationally significant: **a Farm of 10,000 sub-wikis does not mean 10,000 separately maintained application installations.**

2.2 Core Components

A BlueSpice Farm deployment at enterprise scale consists of the following logical layers:

Layer	Component	Deployment Location
Application (web)	BlueSpice Web container – handles HTTP requests	K8s pods
Application (tasks)	BlueSpice task container – background jobs, search index updates, mailings	K8s pods
Web / Proxy	Reverse proxy	K8s pods or ingress controller

Persistence	MariaDB / MySQL (one database per subwiki or prefixed schemas within a database)	External to K8s – customer managed MariaDB/MySQL cluster
Search	OpenSearch cluster	External to K8s — customer managed OpenSearch cluster
Object storage	S3-compatible store (file uploads)	External to K8s — customer managed (e.g., MinIO, Ceph, or cloud S3-compatible endpoint)
Cache	Redis (object cache, session store, jobqueue)	K8s pods (queue runner deployment)
Auth	OIDC / SAML 2.0 / LDAP integration	Native in “Application” or via header-based authentication by a Authn-Proxy
Companion services	AI/RAG, PDF Export, Math-Formulas,...	K8s pods
Real-time notifications	BlueSpice wire container – WebSocket push service	K8s pods
Collaborative editing state	MongoDB — stores active CollabPads session documents and revision records, scoped by wiki ID	External to — customer-K8smanaged MongoDB instance

2.3 How Sub-Wikis Are Provisioned and Isolated

Each sub-wiki in a BlueSpice Farm is defined by:

- A **wiki identifier** (used in URL routing, e.g., farm.example.com/wiki-id)
- A **dedicated database** (or database prefix) within the DB cluster
- A **per-wiki configuration block** within the Farm's central configuration layer (typically in pre-init-settings.php/ post-init-settings.php or included config files)
- A **dedicated file storage namespace** within the shared S3 bucket or storage backend

Critically, there is **no separate PHP process, container, or codebase per wiki**. The same running application pods serve all sub-wikis. The correct wiki context is selected at request time based on the incoming hostname or URL path, and the appropriate database connection and configuration are loaded dynamically.

This means that **provisioning a new sub-wiki is a data and configuration operation**, not an infrastructure operation. No new pods are deployed; no new application instances are started. A new database is created, storage namespace initialised, and the wiki registered in the Farm configuration.

2.4 Dependency Stack

Component	Technology	Notes
Runtime	PHP 8.4+	Shared across all wikis; runs inside the BlueSpice application container image
Application	MediaWiki 1.43+ (LTS) / BlueSpice 5.x	Single shared installation
Database	MariaDB 10.6+ (or MySQL 8.0+)	One database per wiki recommended at this scale; external cluster
Search	OpenSearch 2.x (Elasticsearch-compatible)	Shared cluster; per-wiki index namespacing
Cache	Redis 7.x / Valkey	Object cache + session store + job queue backend
Object Storage	S3-compatible API	File uploads, wiki exports, backup staging
Container orchestration	Kubernetes	Application pods, ingress, config management
Image registry	OCI-compatible registry	BlueSpice application images; update

TLS termination	Ingress controller (e.g., HAProxy)	Handles all inbound HTTPS
Real-time push	wire (Node.js / WebSocket)	Stateless; shared across all sub-wikis; clients pass wiki ID to identify context
Collaborative editing backend	collabpads (Node.js / socket.io)	Shared across all sub-wikis; wiki ID used to route ACL checks to the correct sub-wiki via wiki-web REST API
Collaborative session store	MongoDB 8.0	Stores active collaborative editing sessions and revision records; external to K8s cluster; requires backup

2.4.1 WebSocket Routing

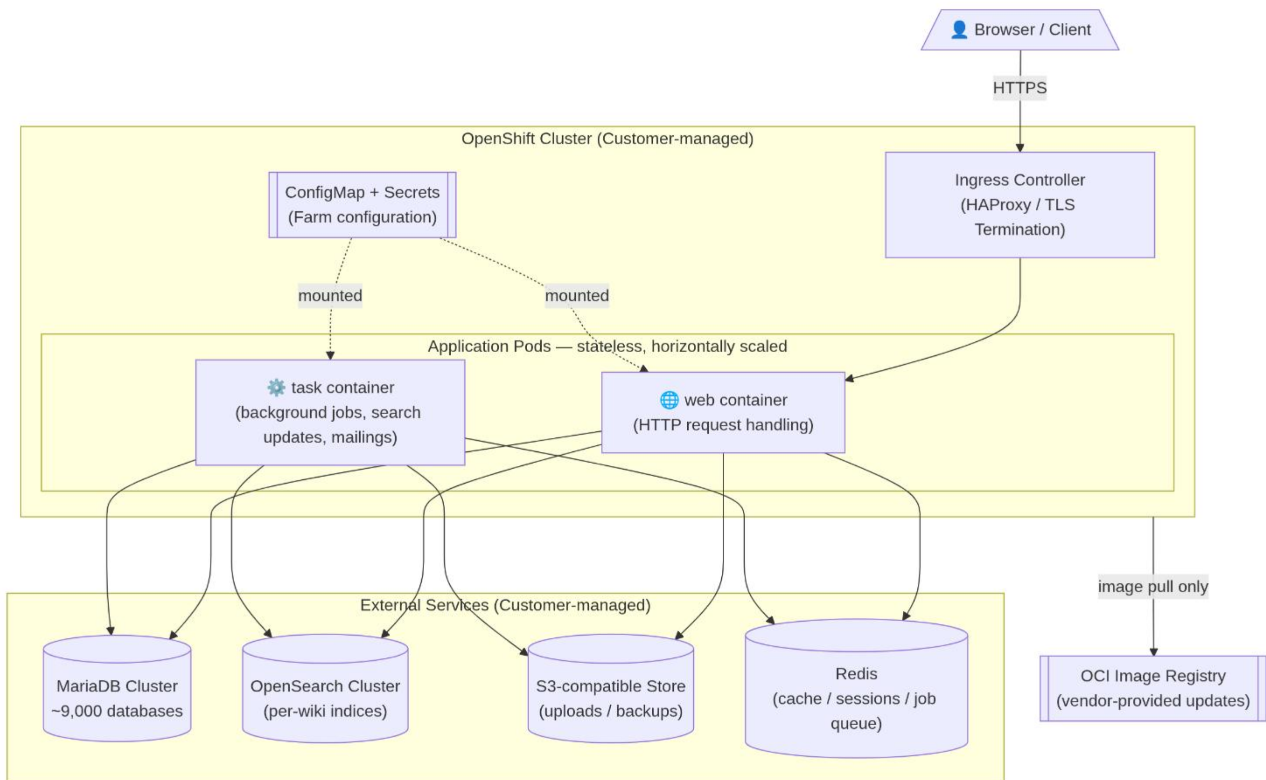
Both the and services communicate with browser clients via persistent WebSocket wirecollabpadsconnections. These connections must be routed by the Cluster ingress controller based on URL path, alongside standard HTTP traffic:

Path	Target Service	Protocol	Notes
/_collabpads/	Collabpads pod	WebSocket (socket.io)	Collaborative editing sessions; ingress must support WebSocket upgrade on this path
/_wire/ (or equivalent)	wire pod	WebSocket	Real-time push notifications; ingress must support WebSocket upgrade on this path
All other paths	Wiki-web pod	HTTP	Standard wiki traffic

The Cluster ingress controller (HAProxy) must support WebSocket proxying and WebSocket upgrade annotations. Long-lived WebSocket connections must be accounted for in ingress timeout configuration — the default HTTP timeout is typically too short for persistent collaborative editing sessions.

2.5 Deployment Topology

All inbound user traffic enters through the cluster ingress layer; Most application pods are stateless and horizontally scalable; all stateful services (database, search, object storage) live outside the cluster on customer-managed infrastructure. The only planned outbound connection from the cluster is to a container image registry for codebase updates — all other dependencies are internal.



Note on air-gap posture

The cluster has no general outbound internet connectivity. The image registry may be an internal mirror (e.g., OpenShift's integrated registry, Harbor, or Nexus) that proxies or manually receives image updates from the vendor. The operational procedure for applying updates must account for this: image delivery from the vendor → internal registry → cluster pull.

3. Scaling to ~ 10,000 Sub-Wikis

3.1 How the Farm Model Scales

The BlueSpice Farm architecture separates concerns that are *shared* from concerns that are *per-wiki*. This separation is the foundation of its scalability:

- Shared (scale once): Application codebase, PHP runtime, extension set, skin set, search cluster, cache layer, job queue infrastructure
- Per-wiki (scale with wiki count): Database, storage namespace, search index, configuration block

This means the operational scaling challenge at 10,000 sub-wikis is primarily a **data infrastructure challenge**, not an application infrastructure challenge. Adding the 10,000th wiki does not require deploying an additional application instance. It requires allocating a database, a storage namespace, and a search index — all of which are handled by provisioning tooling, not manual Ops work.

Horizontal scaling of the application layer (adding more web or task pods) responds to concurrent user load, not to wiki count. The number of wikis and the number of concurrent users are independent scaling axes.

3.2 Shared vs. Per-Wiki Resources

Resources	Shared across all wikis	Per-wiki isolated
PHP application codebase	✓	—
BlueSpice extensions and skins	✓	—
web pod pool	✓ (all wikis served by same pod pool)	—
task pod pool	✓	—
Redis cache layer	✓ (namespaced per wiki via key prefix)	—

OpenSearch cluster	✓ (shared index, shared cluster)	Per-wiki via document metadata
MariaDB cluster	✓ (shared cluster infrastructure)	Per-wiki database
S3 storage bucket(s)	✓ (shared bucket infrastructure)	Per-wiki key prefix / bucket
TLS certificate	✓ (single wildcard or SAN cert for path-based routing)	—
Farm configuration registry	✓	Per-wiki configuration block
User accounts	Shared	—
Permissions / namespaces	—	✓
Content (pages, files, history)	—	✓
wire service	✓ (all sub-wiki clients connect to shared wire pool; wiki ID passed per connection)	—
collabpads service	✓ (shared service; MongoDB documents scoped by wiki ID)	—
MongoDB	✓ (shared instance; wiki ID field scopes all documents)	—
CollabPads MariaDB tables	—	(3 tables per wiki database, added by the CollabPads extension schema update)

3.3 Database Architecture at Scale

At 10,000 sub-wikis, the choice of database isolation strategy has significant operational consequences. BlueSpice Farm supports two modes: one database per wiki and a shared

database with per-wiki table prefixes. At the given scale, the first approach is the way to go.

Each sub-wiki receives its own MariaDB database (e.g. wiki_projectalpha, wiki_projectbeta) The BlueSpice Farm configuration maps each wiki identifier to its corresponding database name.

Item	Pros	Cons
Data isolation	Clean: backup, restore, and deletion are per-database operations	—
Schema migrations	Migrations run per-database; a failed migration affects only one wiki	Running update.php across 10,000 databases requires orchestration (scripted iteration)
Operational clarity	Easy to identify which data belongs to which wiki	MariaDB has practical limits on simultaneous open databases; connection pool management becomes important
Backup granularity	Wiki-level backup and point-in-time restore is straightforward	Backup jobs must iterate 10,000 databases; scheduling and storage management are non-trivial
Deletion / deprovisioning	Drop the database; no residual data risk	—

Database Cluster Topology

The MariaDB cluster itself should be sized and configured for high availability and horizontal read scaling.

High Availability

These HA patterns are commonly used with MariaDB:

Pattern	How it works	Suitable for this deployment
Galera Cluster (synchronous multi-primary)	All nodes accept writes; synchronous replication ensures consistency	No — Galera uses optimistic concurrency control, which can lead to certification conflicts with BlueSpice Farm
Galera Cluster (synchronous, primary-replica)	Only primary accepts writes; synchronous replication ensures consistency; primary failover via ProxySQL	Yes — DB cluster is managed by Galera, stable connection, works for heavy workloads.
Primary / Replica (async replication)	One primary accepts writes; replicas follow asynchronously; failover via orchestrator or ProxySQL	Partially – harder to manage; acceptable for write-heavy workloads where galera's write amplification is a concern; no primary failover

For this kind of setup Galera Cluster (primary-replica) is recommended, as it is natively supported by the application core (MediaWiki) and follows the Wikipedia infrastructure without the management overhead.

Connection Proxy Layer

With multiple application pods (potentially 10–35 pods and additional pods) each maintaining a pool of database connections, the raw connection count arriving at the MariaDB cluster can become a significant operational problem. MariaDB has a hard connection limit (default 151, typically raised to several thousand) and each connection consumes memory on the server side regardless of whether it is actively executing a query.

A connection proxy layer is therefore not optional at this scale — it is a required architectural component.

ProxySQL is the recommended solution. It is fully open source (GPL v3), widely used in production with MariaDB and Galera Cluster, and its transaction-level connection pooling directly addresses the connection count problem: multiple logical connections from application pods share a significantly smaller pool of real connections to the database backend. ProxySQL also provides read/write splitting (routing queries to replica nodes and writes to the primary), Galera node state awareness, and query routing rules — all of which are relevant at this scale.

3.4 Caching Strategy

Redis/Valkey serves three distinct functions in this deployment, each with different operational characteristics:

Function	Redis usage	Failure impact	Persistence needed
Object cache	DB query results	Cache miss → increased DB load; no data loss	No
Session store	Stores authenticated user sessions	Active sessions lost on restart; users re-authenticate	Desirable but not critical (configure short TTL)
Job queue	Stores pending background jobs (search index updates, email, etc.)	Jobs in queue at restart time may be lost or requeued	Desirable

Since none of these functions require durable long-term persistence, Redis/Valkey is well-suited to run as a Kubernetes workload. The recommended configuration for Redis:

- Redis Sentinel (3-node: 1 primary + 2 replicas + sentinel processes) deployed as a K8s StatefulSet for HA without requiring an external operator
- Alternatively, Redis Cluster mode if horizontal sharding is needed — not likely required at this scale
- For the job queue function specifically, enable Redis AOF persistence (append-only file) to minimise job loss on pod restart
- Key namespacing per wiki is handled by the BlueSpice/MediaWiki cache key prefix — no additional configuration needed at the Redis level

Why keep Redis inside the cluster?

Latency between application pods and Redis is critical for cache performance. Running Redis inside the same cluster eliminates a network hop to an external service and simplifies the network security policy. The absence of a persistence requirement removes the primary argument for externalising it.

3.4.1 Frontend Cache Layer

The majority of requests to a BlueSpice Farm arrive in the context of an authenticated user session, and most application endpoints enforce access control on every request. These requests cannot be served from a frontend cache and must reach the application pods:

Endpoint	Purpose	Cacheable?
index.php	Page rendering (authenticated)	✗ — ACL enforced per request
api.php	MediaWiki Action API	✗ — session/auth context required
rest.php	MediaWiki REST API	✗ — session/auth context required
img_auth.php	Authenticated file delivery (protected wikis)	✗ — access check per file per user

However, a subset of requests serves content that is identical for all users and changes only on application updates. These are strong candidates for caching at the ingress or a dedicated frontend cache layer (e.g., Varnish, Nginx proxy cache, or an Cluster-level caching layer):

Endpoint / Resource	Purpose	Cache behaviour
load.php	Delivers combined, minified JavaScript and CSS bundles	content hash in the query string; cache indefinitely, invalidated by hash change on update
dynamic_file_dispatcher.php	Serves certain dynamically generated images (e.g., avatar images for user accounts)	Conditionally cacheable — safe to cache for wikis with no per-user access control on files; must bypass cache if img_auth.php is in use

Static assets (fonts, icons)	Served from the web container's static file path	Cacheable with long TTL (days to weeks); invalidated by image update

3.4.2 WebSocket Connection Scaling

The wire and collabpads services maintain persistent WebSocket connections with browser clients. Unlike HTTP requests, these connections are long-lived — a user actively editing a collaborative pad holds an open connection to collabpads for the duration of their session, and all connected users hold an open connection to wire for real-time notifications.

This has distinct scaling characteristics from HTTP request handling:

Service	Connection lifetime	Concurrent connections (estimate)	Scaling model
wire	Long-lived (for entire browser session)	Up to ~2,000–5,000 (one per active browser session)	Stateless — horizontally scalable; multiple pod replicas behind K8s Service; load balancer distributes connections
collabpads	Long-lived (for duration of collaborative editing session)	Lower — only users actively in a collaborative edit; typically a fraction of total concurrent users	Stateful per session via MongoDB; horizontal scaling requires session-aware load balancing (sticky sessions at ingress, or MongoDB-backed shared state)

wire scaling — current limitation

Although the wire service is stateless with respect to its own data, it cannot currently be scaled horizontally. See Section 7.

collabpads scaling — current limitation

The collabpads service does not currently support horizontal scaling. See Section 7.

MongoDB sizing

MongoDB holds **primary production data**: unsaved changes within active collaborative editing sessions are stored exclusively in MongoDB until the content is saved to MediaWiki. Data loss in MongoDB therefore means loss of in-progress collaborative edits — this is not a reconstructable cache. MongoDB must be treated with the same operational rigour as the MariaDB cluster.

The recommended configuration is a **MongoDB replica set** (1 primary + 2 secondaries) for automatic failover. A single MongoDB instance is not acceptable for production use.

3.4.3 Recommended approach for this deployment

Given the path-based URL routing and cluster ingress layer already in place, the simplest effective implementation is cache rules at the ingress controller level:

- Cache responses where the query string contains a version/hash parameter (MediaWiki always appends one)
- Cache static asset paths (/resources/ , /skins/ directories) with a long TTL
- Pass all index.php, api.php, rest.php, and img_auth.phprequests directly to the application pods with no caching

A dedicated Varnish layer is an option for more sophisticated caching logic (e.g., anonymous-user page caching for any publicly accessible wikis in the farm), but given that the majority of requests in this deployment are expected to be authenticated, the return on investment for a full Varnish deployment is modest unless public wikis form a significant portion of the farm.

There is a dedicated "frontendcache" container image available for direct use.

3.5 Search Infrastructure at Scale

BlueSpice uses a search extension to index and query wiki content via OpenSearch. In this deployment, the search infrastructure is configured to use a single shared index set across all sub-wikis, rather than per-wiki indices.

Index topology

All content from all 10,000 sub-wikis is written into three shared indices:

Index	Content
Wikipage	All wiki page content across all sub-wikis
Specialpage	Special page content, structured data
Repofile	Uploaded files and file metadata

This approach **eliminates the index-count scaling problem** entirely. Instead of 30,000+ indices, the cluster manages three indices regardless of wiki count. Wiki-level access control and scoping are enforced at the application layer (BlueSpice/MediaWiki), not at the index level — a search query is always executed in the context of an authenticated user with a known set of accessible wikis.

Operational implication

Provisioning a new sub-wiki does not require creating new OpenSearch indices or modifying cluster configuration. Content is indexed into the existing shared indices with wiki-identifying metadata fields, and BlueSpice query construction handles per-wiki scoping automatically.

Index sizing (indicative)

Parameter	Estimate	Notes
Average pages per wiki	~300	Weighted average; long tail of small wikis, minority of large wikis
Total indexed documents (wikipage)	~3,000,000	10,000 × 300
Additional documents (repofile, general)	~500,000–1,000,000	Dependent on file upload volume
Estimated total index storage	~100–300 GB	Highly dependent on content richness; validate against Confluence export data
Shard count per index	5–10 primary shards	Sized for total document volume and query throughput, not wiki count
Replicas per shard	1	Provides redundancy; increase to 2 for higher read throughput if needed

Cluster sizing (indicative)

With three indices and a total document volume in the low millions, the OpenSearch cluster can be sized conservatively:

Parameter	Indicative value	Notes
Data nodes (initial)	3–5	N+1 redundancy; scale up based on query latency under load
JVM heap per node	32 GB	Standard OpenSearch recommendation: 50% of available RAM, max 32 GB
Total storage per node	500 GB–1 TB	Provides headroom for index growth; SSDs strongly recommended
Dedicated master nodes	3	Separate from data nodes at this cluster size; prevents master instability under indexing load

These figures should be validated against actual content volumes extracted from the source environment prior to cluster provisioning.

Advantage of shared index approach at this scale

The shared index model significantly simplifies OpenSearch operations: there are no per-wiki index lifecycle management concerns, no shard count explosion, and cluster management overhead is equivalent to operating a single large search application rather than 10,000 small ones. The trade-off is that access control enforcement moves entirely to the application layer.

3.6 Application Pod Sizing

Application pod count is driven by **concurrent user sessions**, not by wiki count. With a target of several hundred to a few thousand concurrent sessions across the Farm:

Scenario	Concurrent sessions	Indicative web pod count	Notes
Conservative baseline	500	8–12 pods	Starting point; scale up based on observed load
Mid-range target	2,000	20–32 pods	Assumes ~150–200 sessions handled per pod
Peak planning	5,000	50–70 pods	Accounts for burst; K8s HPA should handle auto-scaling

BlueSpice web pods are fully stateless – session state lives in Redis, file uploads go to S3, no local disk state. This makes horizontal auto-scaling via Kubernetes HPA (Horizontal Pod Autoscaler) straightforward and reliable.

The task pod pool is sized independently based on job queue throughput requirements (search re-indexing frequency, mailing volume, etc.) and does not directly track concurrent user load.

4. Update & Maintenance Model

This section describes how BlueSpice updates are structured, how they propagate across a Farm deployment, and what the Ops team is responsible for at each update tier. It is the operationally critical section of this document.

4.1 Update & Maintenance Model

BlueSpice releases follow a three-tier SemVer versioning model. Each tier has a different operational impact profile:

Tier	Cadence	DB schema change	Search index change	Infrastructure change	Typical Ops effort
PATCH (e.g. 5.1.1 → 5.1.2)	~Monthly	✗	✗	✗	Low — rolling pod restart only
MINOR (e.g. 5.1 → 5.2)	1–2× per year	Possible	Possible	Rarely	Medium — batched migration procedure required
MAJOR (e.g. 5.x → 6.x)	~Every 3 years	Very likely	Very likely	Likely	High — treated as a special case; bespoke migration plan required

MongoDB schema changes (if any) for the collabpads service are handled at the service level on container restart and do not require a separate per-wiki migration step.

Long-Term Support (LTS) releases

BlueSpice designates certain MINOR releases as Long-Term Support (LTS) releases. An LTS release receives PATCH updates (security fixes, bug fixes) for an extended period — typically several years — without requiring the customer to track each MINOR release as it is published.

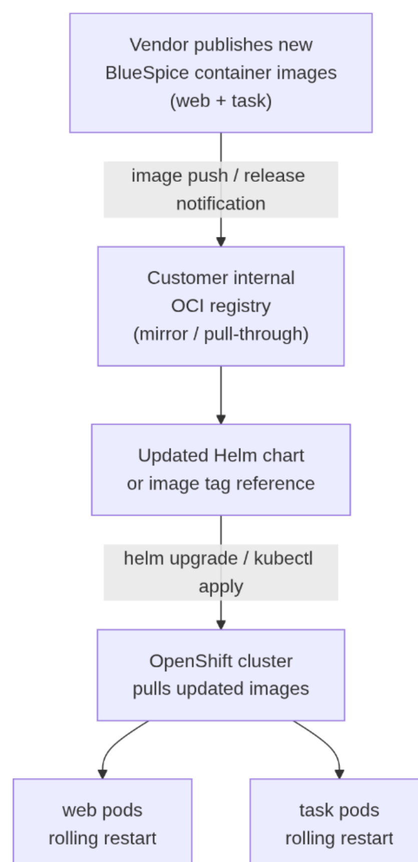
This is the recommended operational posture for this deployment:

- Adopt an LTS release at initial deployment
- Apply PATCH updates on the monthly cadence (low effort, no schema changes)
- Plan MINOR updates as deliberate, scheduled events — typically once or twice per year, or skip to the next LTS
- Treat MAJOR updates as infrastructure projects requiring dedicated planning capacity

The LTS model means the Ops team is **not required to track every MINOR release**. Please note, that there is only one LTS MINOR per MAJOR version.

4.2 Update & Maintenance Model

All BlueSpice application updates are delivered as updated container images published to the vendor's image registry. The delivery path for this deployment is:



The application codebase — PHP, extensions, skins, configuration defaults — is entirely contained within the container images. There is no in-place file update on a shared filesystem. **Updating the codebase means updating the image tag and performing a rolling pod restart.**

This delivery model has a direct operational consequence: the customer's image registry must be able to receive updated images from the vendor. In an air-gap-adjacent environment (no general outbound connectivity), a defined process for image ingestion — whether via a pull-through cache registry, a manual transfer procedure, or a scheduled sync job — must be established and tested before the first production update.

4.3 PATCH Updates – Procedure

PATCH updates are the routine maintenance cycle. They do not modify the database schema or search index structure, and the update procedure is operationally simple:

1. Vendor publishes updated web and task container images
2. Customer ingests images into internal registry
3. Helm chart image tag is updated (or a new chart version is published by vendor)
4. Helm upgrade is executed against the K8s cluster
5. Kubernetes performs a rolling restart of web and task pods — old pods continue serving traffic until new pods pass health checks
6. No database or search index changes are required
7. Rollback, if needed: helm rollback to the previous release — restores previous image tag and restarts pods

Downtime: zero (rolling restart). **Ops effort: low** (image ingestion + single Helm command). This is the update procedure the Ops team will execute most frequently — approximately monthly.

The wire and collabpads pods are also restarted as part of the rolling update. Active WebSocket connections will be dropped on pod restart; clients reconnect automatically. For collabpads, any active collaborative editing sessions will be interrupted — users will need to rejoin their sessions after the restart. **For PATCH updates, this is considered acceptable**. For MINOR updates affecting collabpads, a maintenance window notification to active users is recommended.

4.4 MINOR Updates – Expand/Contract Pattern

MINOR updates may introduce changes to the MariaDB schema and/or the OpenSearch index mappings. These changes must be applied to all 10,000 databases and/or all affected indices. The procedure follows the **expand/contract (zero-downtime) pattern**:

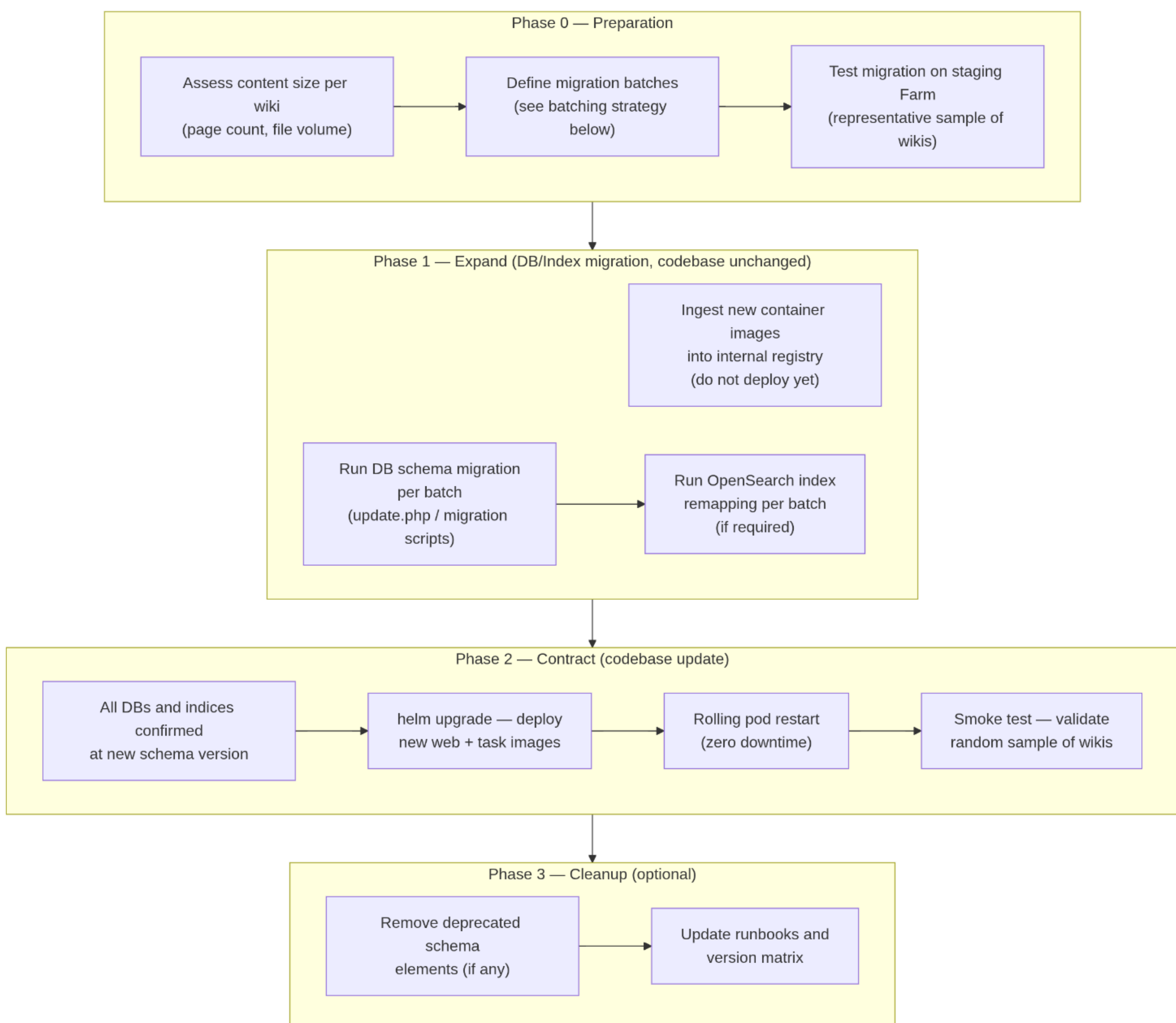
The expand/contract principle

The expand/contract pattern ensures that a schema migration never creates a hard dependency between the codebase version and the database state. It works as follows:

1. **Expand phase:** The new schema version (e.g. 5.2 schema) is designed to be **backward-compatible with the current codebase version** (5.1). New columns, tables, or index fields are added in a non-breaking way. The 5.1 codebase continues to operate correctly against the updated schema.
2. **Contract phase:** Once all databases/indices have been migrated to the new schema, the codebase is updated to 5.2 (rolling pod restart). The new codebase may then use the new schema features. Old schema elements that are no longer needed may be cleaned up.

This pattern guarantees that at no point during the migration is there a version mismatch between the running codebase and the database state that causes an application failure.

MINOR update procedure



Batching strategy for schema migrations

At 10,000 databases, running schema migrations serially (one at a time) or in a single bulk operation are both suboptimal. Serial execution is too slow; bulk execution risks overwhelming the MariaDB cluster and makes failure recovery harder.

The recommended approach is **parallel batched execution**, with batch size calibrated to database content volume:

Batch type	Criteria	Suggested parallelism	Estimated time per batch
Small wikis	< 500 pages, < 1 GB	50–100 wikis in parallel	30 sec – 2 min per wiki
Medium wikis	500–5,000 pages, 1–10 GB	10–20 wikis in parallel	1–5 min per wiki
Large wikis	5,000–50,000 pages, > 10 GB	3–5 wikis in parallel	5–30 min per wiki
Very large wikis	> 50,000 pages	1 wiki at a time	Profile individually

These figures are indicative. Actual migration duration depends on the nature of the schema change (adding a nullable column is near-instant; rebuilding an index on a large table is not) and must be profiled on a staging environment with representative data before production execution.

A migration orchestration script (provided or co-developed with the vendor) should:

- Read the wiki list and pre-assessed content size classification
- Execute migration batches in parallel using the appropriate concurrency level
- Log success/failure per wiki with timestamps
- Halt the batch on unexpected errors and alert the Ops team before proceeding
- Produce a completion report (wikis migrated, wikis failed, total elapsed time)

Failed individual wiki migrations do not block the overall migration — the 5.1 codebase continues to run correctly against both migrated (5.2 schema) and unmigrated (5.1 schema) databases until all wikis are complete. Failed wikis can be retried independently.

Indicative total migration time (MINOR update)

Assuming a realistic content distribution (majority small wikis, minority large) and the parallelism levels above:

Wiki category	Estimated count	Batched migration time
Small (bulk batches of 100)	~7,800	~1–2.5 hours total
Medium (batches of 15)	~1,700	~2–4.5 hours total
Large (batches of 5)	~400	~2–6 hours total
Very large (individual)	~100	~2–5 hours total
Total (sequential batch execution)	**10,000**	**~8–18 hours**

This estimate is provided for planning purposes only. A realistic figure requires profiling on staging with production-representative data. The migration window can be reduced by increasing parallelism where MariaDB cluster headroom allows.

4.5 MAJOR Updates

MAJOR version updates (e.g. BlueSpice 5.x → 6.x) are released approximately every three years. They typically involve:

- Infrastructure-level changes (new required services, changed container structure, updated dependency versions)
- Non-backward-compatible schema changes that may not follow the standard expand/contract pattern
- Potential changes to extension set, configuration format, or Helm chart structure

MAJOR updates cannot be handled by the standard PATCH or MINOR procedures. Each MAJOR update must be treated as an infrastructure project with its own scoping, planning, staging validation, and rollback strategy. A bespoke Standard Operating Procedure (SOP) will be developed jointly between the vendor and the customer Ops team in advance of each MAJOR release.

Given the ~3-year cadence, this represents a significant but infrequent operational event. The LTS model (Section 3.1) ensures the customer has maximum advance notice and a stable maintenance window before a MAJOR update becomes necessary.

BlueSpice MediaWiki

4.6 Rollback Procedures

Update tier	Rollback mechanism	Conditions	Complexity
PATCH	helm rollback to previous release	Any time after deployment	Very low — pod restart only
MINOR (codebase only, Phase 2)	helm rollback to previous release	Before schema cleanup (Phase 3) — old codebase is compatible with new schema	Low — pod restart only; expand/contract guarantees compatibility
MINOR (schema migration)	Restore individual database(s) from pre-migration backup	Per-wiki; only for wikis where migration failed or produced corrupt state	Medium — requires tested per-wiki backup /restore procedure
MAJOR	Bespoke rollback plan, defined per release	Full environment rollback may be required	High — must be defined in the MAJOR update SOP
wire / collabpads (any tier)	helm rollback restores previous image tag for wire and collabpads pods	Any time	Very low — pod restart; MongoDB data is unaffected by application rollback

Key principle: The expand/contract pattern means that rolling back the codebase (Phase 2) is always safe — the old codebase runs correctly against the new schema. Schema rollback (reverting a database migration) is a last resort, requires a backup restore, and should only be triggered for wikis where data integrity issues are confirmed.

4.7 Configuration Management

Farm-level configuration is managed through Kubernetes-native mechanisms:

- **ConfigMaps and Secrets** — per-wiki and Farm-wide configuration values are mounted into application pods at startup; changes trigger pod restarts via standard K8s rolling update mechanisms
- **Helm chart values** — infrastructure-level configuration (replica counts, resource limits, image tags, service endpoints) is managed in Helm values files, version-controlled in the customer's GitOps repository
- **GitOps workflow** — the recommended operational model is a GitOps pipeline (e.g., ArgoCD or OpenShift GitOps) where all configuration and deployment changes are committed to a Git repository and applied to the cluster via a reconciliation loop; this provides audit trail, rollback capability, and change review for all infrastructure changes

The vendor does not prescribe a specific GitOps toolchain. Ansible and Puppet-based automation can wrap the Helm/kubectl operations for customers who prefer an imperative SOP-driven model; the vendor can assist with SOP design for these workflows.

5. Operational Responsibility Split

This section defines what the customer Ops team owns, what the vendor provides, and where the boundary lies. It is the primary reference for scoping the Ops team's workload.

5.1 Responsibility Matrix

Area	Vendor (Hallo Welt! GmbH)	Customer Ops Team	Notes
Application lifecycle			
Application updates (PATCH)	Publishes updated container images monthly; provides release notes and security advisories	Ingests images into internal registry; executes helm upgrade ; validates deployment	Routine monthly task; no schema changes; rolling restart, zero downtime
Application updates (MINOR)	Provides updated images, migration scripts, migration tooling, and documented procedure; supports staging validation	Plans and executes batched DB/index migration; schedules maintenance window; executes codebase rollout	Requires advance planning; see Section 3.4 for procedure detail
Application updates (MAJOR)	Provides bespoke migration guide and SOP; active support engagement during migration	Executes migration per agreed SOP; manages infrastructure-level changes	Treated as an infrastructure project; joint planning required
Security advisories	Publishes advisories when vulnerabilities are identified; provides impact assessment and severity rating (including downgrade justification where applicable); out-of-schedule image releases for critical CVEs	Subscribes to and monitors vendor advisories; decides on deployment timing within own change management process	Customer must establish an advisory notification channel with the vendor

OS / runtime security patching	Rebuilds and publishes container images with updated OS packages in each PATCH release; out-of-schedule for high-severity CVEs	Applies updated images via standard update procedure	No direct customer access to container base image required; all OS-level patching is delivered through image updates
Instance / Sub-wiki lifecycle			
Sub-wiki provisioning (create / archive)	Platform provides self-service Web UI and API for authorised wiki admins	Grants provisioning permissions to authorised users; monitors provisioning activity if required	No Ops action required for standard create/archive operations
Sub-wiki unarchiving	Platform provides no automated unarchive capability. Archive must be restored manually.	Executes unarchive operation (Ops-level action required)	Unarchive is intentionally gated at Ops level to prevent uncontrolled reactivation
Stateless services management			
wire service	Provides container image and configuration recommendations; updates delivered via PATCH/MINOR releases	Operates wire pod pool (replica count, resource limits, HPA configuration); monitors WebSocket connection metrics	Inside K8s cluster; stateless — operationally simple; restart drops active connections transiently
collabpads service	Provides container image, configuration recommendations, and guidance on horizontal scaling model	Operates collabpads pod(s); configures ingress WebSocket routing for /_collabpads/ ; monitors active session counts and connection stability	Inside K8s cluster; session state in MongoDB; confirm scaling model with vendor before sizing

Stateful services management			
MariaDB cluster	Provides ProxySQL configuration recommendations; assists with schema migration tooling; advises on cluster sizing	Operates MariaDB cluster (HA, replication, failover, upgrades, capacity management); manages ProxySQL deployment	External to K8s cluster; fully customer-managed infrastructure
OpenSearch cluster	Provides index configuration and mapping recommendations; assists with reindex tooling for MINOR/MAJOR updates	Operates OpenSearch cluster (node management, upgrades, capacity, snapshot management)	External to K8s cluster; fully customer-managed infrastructure
Redis/Valkey	Provides configuration recommendations for object cache, session store, and job queue roles	Operates Redis/Valkey (cluster deployment, availability, capacity)	Inside K8s cluster; customer-managed
S3-compatible object storage	Provides configuration guidance for bucket structure and access policy	Operates S3 backend (capacity, availability, access control, lifecycle policies)	External to K8s cluster; fully customer-managed infrastructure
MongoDB	Provides configuration guidance and sizing recommendations for collabpads workload; advises on backup strategy	Operates MongoDB instance (availability, capacity, upgrades, backup and restore); integrates MongoDB backups into overall backup schedule	External to K8s cluster; fully customer-managed; production data — backup is mandatory
Application management			
Helm chart management	Provides and versions Helm charts; publishes chart updates alongside image releases	Maintains chart values in GitOps repository; executes helm upgrade;	Customer-owned values files; vendor-owned chart templates

Backup and recovery	Provides basic backup tooling as part of the product; advises on backup strategy for this scale; can adapt tooling based on findings; co-develops approach with customer	Designs and operates backup solution for this scale (DB dumps or snapshots, S3 versioning /replication, OpenSearch snapshots); defines and tests RTO/RPO targets; schedules and monitors backup jobs	Standard product backup tooling may not be sufficient at 10,000-wiki scale; backup strategy requires joint assessment and will likely involve custom or extended tooling
Kubernetes management			
Kubernetes / OpenShift cluster	Provides Helm charts and deployment manifests; supports troubleshooting of application-layer issues	Manages K8s/OpenShift platform (upgrades, node capacity, network policies, RBAC, ingress configuration)	Vendor has no access to the cluster; all cluster operations are customer responsibility
Monitoring and alerting	Provides Prometheus exporter integrations for application services (wiki-web, wiki-task,etc.); provides recommendations and guidelines based on internal cloud experience; assists with setup	Deploys and operates monitoring stack (Prometheus, Grafana, alerting rules); integrates vendor exporters into existing observability infrastructure; defines alert thresholds	No out-of-the-box dashboards or alert rules provided at product level; vendor can share internal references
Disaster recovery	Provides architectural guidance; assists in DR planning	Defines DR runbooks; tests failover and restore procedures; operates DR environment if required	See Section 5 for risk and DR posture

TLS certificates	No involvement	Manages wildcard or SAN certificate lifecycle for path-based Farm URL scheme; integrates with Cluster ingress	Path-based routing requires one certificate covering the Farm domain
Capacity planning	Provides scaling recommendations and guidance based on observed usage patterns	Monitors resource utilisation; makes infrastructure scaling decisions; executes scaling operations	Vendor provides input; customer owns the decision and execution
External service integration			
Identity and access (SSO/ SAML/LDAP)	Provides supported integration configuration (OIDC/SAML/LDAP/Header-Based-Auth); supports troubleshooting	Operates and maintains IdP integration; manages LDAP/AD connectivity from within the cluster network	Auth traffic stays on internal network; no outbound auth connections expected

5.2 What the Ops Team Does Not Need to Do

To make the boundary explicit: the following operational concerns that would apply to a traditional self-hosted application are **handled by the platform model** and do not require per-wiki Ops action:

- **Per-wiki application updates** — there is one codebase; updating it updates all 10,000 wikis simultaneously
- **Per-wiki extension or skin management** — extensions and skins are part of the shared codebase; they are not installed or managed per wiki
- Per-wiki web server configuration — all wikis are served by the same pod pool behind the same ingress
- **Per-wiki security patching of the application layer** — OS and runtime patches are delivered via container image updates, applied once across all wikis
- **Creating or decommissioning application infrastructure per new wiki** — provisioning a new sub-wiki is a data operation (database + storage namespace + config entry), not an infrastructure operation

- **Per-wiki CollabPads configuration** — the collabpads and wire services are shared across all sub-wikis; wiki context is determined at runtime via wiki ID; no per-wiki service configuration is required

The number of sub-wikis does not drive Ops workload at the application layer. It drives workload at the data layer (DB migration batching, backup scope, storage capacity) and at the one-time setup of provisioning automation and monitoring coverage.

5.3 Escalation Path

Situation	First responder	Escalation to vendor
Application pod crash / restart loop	Customer Ops (check pod logs, resource limits)	If root cause is in application code or configuration
Wiki rendering error or functional regression after update	Customer Ops (check update logs, attempt rollback)	If issue persists after rollback or is confirmed application bug
Database performance degradation	Customer Ops (query analysis, ProxySQL stats, cluster health)	If degradation correlates with application query patterns (vendor can advise)
Security advisory received	Customer Ops assesses; schedules update per change management process	If advisory requires clarification on exploit applicability or if out-of-schedule patch is required
Schema migration failure (individual wiki)	Customer Ops (retry, check migration logs)	If migration script produces unexpected errors not covered in runbook
MAJOR update planning	Joint planning engagement initiated	Vendor leads SOP development

6. Operational Risk Assessment

This section maps the known failure modes of the architecture, assesses their impact and likelihood, and describes the mitigation posture. It is intended to give the Ops team a realistic view of where operational risk is concentrated.

6.1 Escalation Path

#	Component	Failure mode	Blast radius	Likelihood	Mitigation
1	Application pods (web)	All pods unavailable (bad deploy, OOM, crash loop)	All wikis unavailable — entire Farm offline	Low (K8s health checks prevent bad deploys reaching full rollout)	Rolling update strategy; health check gates; fallback cluster for pre-production validation; helm rollback as immediate recovery
2	Application pods (task)	All task pods unavailable	Background jobs suspended (search index updates delayed, mailings queued) — wikis remain readable and editable	Low	Task pods are independent of web pods; degraded mode is acceptable short-term; Redis job queue retains jobs across restarts
3	MariaDB cluster	Primary node failure	Write unavailability until failover completes	Low (Galera auto-failover)	Galera multi-primary — another node assumes writes automatically; ProxySQL detects node state change; RTO: seconds to low minutes

4	MariaDB cluster	Full cluster failure (all nodes)	**All wikis unavailable** — no reads or writes possible	Very low	N+1 node redundancy; restore from backup to secondary cluster if unrecoverable; geo-redundant secondary cluster (see Section 5.3)
5	ProxySQL	ProxySQL unavailability	All application-to-DB connections fail — **all wikis unavailable**	Low	Deploy ProxySQL as a replicated K8s StatefulSet or run multiple instances behind a K8s Service; ProxySQL itself is stateless between connections
6	OpenSearch cluster	Node failure (s)	Degraded search performance; search may return errors for some queries	Low (shard replication absorbs single node loss)	Shard replicas ensure availability across node loss; dedicated master nodes prevent split-brain; cluster auto-recovers on node restart
7	OpenSearch cluster	Full cluster failure	**Search unavailable** across all wikis — page browsing and editing remain functional; search returns errors	Very low	All wikis remain readable and editable without search; restore from snapshot; geo-redundant secondary
8	Redis	Full Redis unavailability	Cache misses increase DB load significantly; sessions lost (users re-authenticate); job queue suspended	Low (Sentinel HA inside cluster)	Redis Sentinel provides automatic failover; AOF persistence reduces job queue loss; application degrades gracefully on cache miss (no data loss)

9	S3 object storage	Unavailability	File uploads fail; existing file serving fails for files not cached	Low (depends on customer S3 backend HA)	S3 backend should be operated with its own HA posture; application remains functional for text content without file access
10	K8s cluster	Full cluster failure	**All application services unavailable**	Very low	Geo-redundant secondary cluster (see Section 5.5); cluster-level HA via multi-node control plane
11	Schema migration (MINOR update)	Migration failure on a subset of wikis	Affected wikis may be temporarily unavailable or in degraded state; remainder of Farm unaffected	Medium (complex migrations on large wikis carry higher risk)	Expand/contract pattern ensures codebase continues running on unmigrated wikis; failed wikis retried independently; pre-migration backup per wiki batch
12	Image registry	Internal registry unavailable	New pod starts fail (existing running pods unaffected)	Low	Running pods continue operating; new pod scheduling blocked until registry recovers; critical path only during updates or pod restarts

13	wire service	Pod unavailable (crash, OOM, restart)	Real-time push notifications suspended across all wikis — wikis remain fully readable and editable; users do not receive live update notifications until pod recovers	Medium (single pod; no redundancy)	Single instance is the current deployment model — no HA failover; K8s restarts the pod automatically (RTO: seconds to low minutes); no data loss on failure; notifications missed during downtime are not replayed; horizontal scaling blocked pending Redis/Valkey pub/sub implementation (see Section 7)
14	collabpads service	Pod unavailable (crash, OOM, restart)	Collaborative editing unavailable across all wikis; active sessions and any unsaved collaborative changes lost — standard editing remains fully functional	Medium (single pod; no redundancy)	Single instance by design — no failover pod; K8s restarts the pod automatically but in-flight session state is lost; unsaved collaborative changes are unrecoverable; users must restart their session; acceptable only because standard editing remains available as a fallback
15	MongoDB	Full MongoDB failure	All active collaborative editing sessions lost; collabpads service unavailable until MongoDB recovers	Low (depends on MongoDB HA configuration)	MongoDB should be operated with replica set (minimum 3 nodes) for automatic failover; session data loss on unclean failure is acceptable (sessions are resumable); draft revision records should be recoverable from backup

16	collabpads + MongoDB	MongoDB data corruption or loss	Historical collaborative revision records lost; active sessions lost	Very low	Regular MongoDB backups (daily minimum) allow point-in-time restore; canonical MediaWiki page revision history is stored in MariaDB and is unaffected

6.2 Single Points of Failure

The following components represent single points of failure that require explicit HA treatment. Each is addressable within the described architecture:

Component	SPOF risk	HA treatment
ProxySQL	If a single ProxySQL instance fails, all DB connections fail	Run as a replicated deployment (2+ instances) behind a K8s Service; stateless between connections — no session state to synchronise
Cluster ingress controller	Single ingress controller failure makes the entire Farm unreachable	Cluster supports multiple ingress controller replicas; configure accordingly
MariaDB primary (non-Galera)	If using primary /replica without Galera, primary failure requires manual or orchestrated failover	Prefer Galera Cluster for automatic multi-primary failover; or use an external orchestrator (e.g., Orchestrator) with primary /replica
Redis/Valkey primary (before Sentinel)	Single Redis /Valkey instance failure loses cache, sessions, and job queue	Deploy Redis/Valkey Sentinel (3-node) as described in Section 2.4; Sentinel handles automatic primary election

Internal image registry	Unavailability blocks all pod restarts and updates	Operate registry with its own HA posture; consider a pull-through cache for resilience
MongoDB (single instance)	CollabPads session state and revision history unavailable; collabpads service fails	Operate MongoDB as a replica set (1 primary + 2 secondaries); automatic primary election on failure
collabpads (single pod— architectural constraint)	Collaborative editing unavailable Farm-wide on pod failure; unsaved changes in active sessions lost	Single instance is the current supported deployment model — no HA option available; mitigation is limited to K8s automatic pod restart (RTO: seconds to low minutes) and user awareness that unsaved collaborative changes may be lost; horizontal scaling is on the vendor roadmap (see Section 7)
wire (single pod — architectural constraint)	Real-time push notifications unavailable Farm-wide on pod failure	Single instance is the current supported deployment model; horizontal scaling requires Redis/Valkey pub/sub fan-out support, which is not yet implemented; mitigation is limited to K8s automatic pod restart; no data loss on failure

6.3 Disaster Recovery Posture

Recovery objectives

Formal RTO (Recovery Time Objective) and RPO (Recovery Point Objective) targets must be defined by the customer in conjunction with their business requirements. The architecture supports the following posture:

Failure scope	Recovery mechanism	Indicative RTO	RPO dependency
Single wiki data corruption	Per-wiki database restore from backup	Minutes to low hours (depending on DB size + restore procedure)	Time since last backup of that wiki's database
Partial DB cluster failure	Galera auto-failover	Seconds to low minutes	Near-zero (synchronous replication)
Full application layer failure	helm rollback or redeploy from Helm chart	Minutes	N/A (no data in application layer)
Full primary site failure	Failover to geo-redundant secondary cluster	Hours (depending on data replication lag and recovery procedure)	Time since last replication sync to secondary
Catastrophic data loss (primary + secondary)	Restore from offsite backup	Hours to days (depending on backup size and restore infrastructure)	Time since last offsite backup

Backup scope

A complete backup strategy must cover all stateful components:

Component	Backup method	Recommended Frequency	Notes
MariaDB (10,000 databases)	Logical dump (mysqldump / mariadb-dump) per database, or physical snapshot (Percona XtraBackup / MariaDB Backup) of full cluster	Daily minimum; hourly for high-traffic wikis	At 10,000 databases, backup orchestration requires scripted tooling; see Section 4.1
OpenSearch indices	OpenSearch snapshot API to S3	Daily	Snapshots are incremental after the first full snapshot; storage-efficient
S3 object storage	S3 versioning + cross-bucket or cross-site replication	Continuous (replication) + periodic snapshot	Depends on customer S3 backend capabilities
Redis/Valkey	AOF file backup (for job queue recovery)	On a best-effort basis; Redis/Valkey is reconstructable	Low priority relative to DB and S3
Helm values / GitOps repository	Git repository backup	Continuous (Git replication)	Configuration-as-code— loss is recoverable from Git history

Geo-redundant secondary cluster

The customer is planning a geo-redundant secondary infrastructure. While the detailed architecture of the secondary site is not yet defined, the following principles apply:

- The secondary cluster should be capable of running the full BlueSpice Farm stack independently (application pods, Redis/Valkey, connectivity to a replicated DB and OpenSearch cluster)

- Database replication to the secondary site should be asynchronous (acceptable RPO) or synchronous (near-zero RPO, higher latency cost) depending on the customer's RTO/RPO requirements
- The secondary cluster serves a dual purpose: DR failover target and pre-production validation environment for updates (see Section 5.4)
- Detailed DR architecture, failover runbooks, and replication topology require a dedicated design phase; this document flags them as open items (see Section 7)

6.4 Update Rollout Risk and Staged Deployment

The highest-probability operational risk event is not infrastructure failure but a regression introduced by an application update. The following staged rollout options are available, in increasing order of rigour:

Option	Description	Downside	Recommended for
K8s rolling update with health checks	Kubernetes deploys new pods one at a time, running health and readiness probes before proceeding; a failing probe halts the rollout	Health checks catch crashes and startup failures but not functional regressions that pass startup	All updates as a baseline minimum
Canary pod deployment	One new-version pod is added to the pool while remaining pods stay on the previous version; a sample of live traffic hits the canary; rollout proceeds only on confirmation	Requires traffic splitting at ingress level; adds complexity to the rollout procedure	MINOR updates; any update where a functional regression is a concern
Secondary cluster validation	The full update (including schema migrations) is applied to the secondary/fallback cluster first; the Ops team validates against production-representative data before applying to primary	Requires the secondary cluster to hold a recent copy of production data; adds time to the update window	MINOR and MAJOR updates; strongly recommended given the secondary cluster already exists

Recommendation: The existence of a geo-redundant secondary cluster makes **secondary cluster validation the natural pre-production gate** for all MINOR updates. The update procedure in Section 3.4 should include an explicit secondary-cluster validation phase before primary cluster rollout.

6.5 Security Surface Area

BlueSpice Farm supports centralised authentication via OpenIDConnect, SAML 2.0, LDAP/Active Directory or Header-Based-Authentication. At Farm level, the following security boundaries apply:

- **Farm-level admin access** — a small set of Farm administrator accounts can manage all sub-wikis; this access must be tightly controlled and audited
- **Per-wiki permissions** — each sub-wiki has its own permission model; users can be granted access to individual wikis without Farm-wide visibility
- **Shared session (optional)** — if enabled, user sessions are shared across wikis; permissions remain per-wiki; this simplifies user management but creates a shared authentication surface

Network security

- All inbound traffic enters via the Cluster ingress controller over HTTPS; no direct pod exposure
- The application cluster has no general outbound internet connectivity; the only planned outbound connection is image pulls from the container registry
- All companion services (MariaDB, OpenSearch, S3) communicate over internal network segments; access should be restricted by network policy to application pod IP ranges only
- img_auth.phpensures that file access is subject to the same permission checks as page access-files in protected wikis cannot be accessed by unauthenticated or unauthorised users by guessing URLs

Attack surface reduction

- Container images are rebuilt monthly with updated OS packages; high-severity CVEs trigger out-of-schedule image releases
- The vendor publishes security advisories with impact assessments, including downgrade justifications where BlueSpice's use of an affected component does not permit practical exploitation
- Customers should subscribe to vendor security advisories through a defined notification channel and integrate them into their vulnerability management process

Privilege escalation risk

The primary privilege escalation risk in a Farm deployment is a wiki administrator gaining Farm-level access through misconfiguration. Farm-level administrative access should be:

- Restricted to a named list of Ops/admin accounts
- Protected by MFA (enforced at the IdP level via SSO)
- Audited via MediaWiki's built-in log system (Special:Log)

WebSocket services

The wire and collabpads services introduce a WebSocket attack surface that is distinct from standard HTTP:

- WebSocket endpoints exposed via the ingress (/_collabpads/ , /_wire/) should be subject to the same network policy controls as HTTP endpoints; unauthenticated WebSocket upgrade requests should be rejected at the application layer
- MongoDB should not be accessible from outside the internal network; access should be restricted by network policy to collabpads pod IP ranges only

6.6 Upgrade Failure Handling

The following describes what happens if an update fails mid-execution, by update tier:

PATCH update failure

K8s rolling update halts when the health check fails on a new pod. Existing pods (previous version) continue serving traffic. The Ops team executes helm rollback. The failed image is never fully deployed. **No user impact.**

MINOR update — schema migration failure (mid-batch)

The migration orchestration script detects the failure, logs the affected wiki(s), and halts the current batch. The codebase has not been updated (Phase 2 has not started). Wikis with failed migrations continue running on the current codebase against their current schema — **no user impact.** The Ops team investigates the failure, resolves the root cause, and retries the affected batch. The overall migration continues from where it stopped.

The critical invariant of the expand/contract pattern is maintained: **at no point during a MINOR migration is there a state where the running codebase is incompatible with any database in the Farm.**

MINOR update — codebase rollout failure (Phase 2)

K8s rolling update halts on health check failure. helm rollback is executed. Because the new schema is backward-compatible with the old codebase, rolling back the codebase is safe regardless of how many databases have already been migrated. **No user impact.**

MAJOR update failure

Failure handling is defined in the bespoke MAJOR update SOP, developed jointly before the update is executed. The secondary cluster serves as the primary safety net — the MAJOR update is validated there before any production changes are made.

7. Reference Points & Benchmarks

This section provides the available evidence base for BlueSpice Farm at scale.

7.1 BlueSpice – Known Deployment Scale

Hallo Welt! GmbH operates BlueSpice in production at the following scales (named references not available for publication):

Deployment type	Scale	Traffic profile	Notes
BlueSpice Cloud (bluespice.cloud)	Multi-tenant Farm; undisclosed tenant count	Mixed; vendor-operated	Architecturally equivalent to the self-hosted Farm model described in this document; the vendor's own operational experience on K8s with the same stack is directly applicable
Large enterprise deployment (manufacturing sector, Europe)	Single installation; ~10–15 wiki instances	High traffic; predominantly anonymous users; CDN in front of application layer	Demonstrates application performance under high anonymous read load; CDN layer consistent with recommendations in Section 2.4.1; not comparable on wiki count
Largest known Farm deployment (public sector IT provider)	~200 sub-wikis	Moderate	Largest known BlueSpice Farm by wiki count prior to this project; approximately 2% of the target scale of this deployment

7.2 MediaWiki at Wikipedia Scale

BlueSpice is built on MediaWiki, and the Wikimedia Foundation operates MediaWiki at a scale that dwarfs the target deployment by several orders of magnitude. This provides the strongest available evidence that the underlying engine architecture is sound.

Metric	Wikimedia / Wikipedia	This deployment (target)
Wikis hosted	~900 (across all Wikimedia projects)	~10,000
Total pages	Hundreds of millions	~2.7 million (estimated)
Monthly page views	~20 billion	Not yet estimated
Concurrent users	Tens of thousands	Hundreds to low thousands
Search backend	Elasticsearch / CirrusSearch (same extension as BlueSpice)	OpenSearch / CirrusSearch
Database backend	MariaDB (same engine)	MariaDB
Cache layer	Redis/Valkey (same role)	Redis/Valkey
Architecture pattern	Shared codebase, per-wiki databases, centralised cache and search	Identical

The Wikimedia infrastructure is publicly documented in detail at mediawiki.org and in the Wikimedia Foundation's annual infrastructure reports. It represents the most rigorous available proof-of-concept for the MediaWiki engine's scalability characteristics.

Relevant difference

Wikimedia operates far fewer wikis (900), but at vastly higher traffic per wiki. This deployment inverts that ratio: many more wikis, most with low individual traffic. The scaling challenge here is breadth of data management (databases, indices, backup scope), not peak request throughput. The MediaWiki engine handles both profiles; the operational challenge is primarily at the infrastructure layer.

8. Open Questions / Gaps

This section consolidates items that are either not confirmed in available documentation, require direct clarification from Hallo Welt! GmbH, or depend on customer-side decisions not yet made. It is structured as a list of questions suitable for a technical call between the customer Ops team and the vendor.

8.1 Vendor roadmap

Database

1. Schema migration tooling: A migration orchestration tool is provided for MINOR updates at Farm scale. Customers and vendor collaborate on developping batching and parallelism tooling around . The recommended interface for triggering per-wiki migrations programmatically is documented in the Farm administration guide.
2. Migration duration profiling: Tooling is available to pre-assess migration duration per wiki before a MINOR update is executed in production, enabling batch sizing decisions (Section 3.4) to be made on data rather than estimates.
3. ProxySQL configuration: A reference ProxySQL configuration for BlueSpice Farm deployments is provided by the vendor, including connection pool sizing recommendations per pod count.

Provisioning and lifecycle

1. Unarchive procedure: The Ops-level unarchive operation is a documented SOP, performed via [CLI command / UI action / direct configuration change]. The procedure is covered in the operational runbook.
2. Wiki deletion (permanent): Permanent deletion of a sub-wiki beyond archiving is supported. The safe purge procedure ensures no residual data remains in the shared OpenSearch index or S3 bucket, and the scope of removed data is documented.

Updates and migrations

1. Expand/contract: Backward compatibility of new schema versions with the previous codebase version — the expand/contract pattern described in Section 3.4 — is covered by processes for MINOR releases.
2. MAJOR update roadmap: The current roadmap and expected timeline for the next MAJOR release (BlueSpice 6.x) are available. Anticipated infrastructure changes are outlined in the upgrade planning guide.
    a. A parallel-run migration strategy with a high degree of automation is planned for MAJOR updates. A concrete timeline is not yet available and will be communicated as part of the roadmap process.

Monitoring and observability

1. Prometheus exporter coverage: The vendor-provided Prometheus exporters for wiki-web and wiki-task expose a defined set of metrics. A reference metric catalogue and example scrape_config are available in the monitoring documentation.
   a. The vendor's own cloud offering (bluespice.cloud) already uses Prometheus to monitor its wiki Farm at scale. The vendor can provide advice and implementation support to customers adopting the same approach.
2. Application-level logging: BlueSpice emits structured logs in [JSON / plain text] format. A documented log schema for ingestion into a customer SIEM or log aggregation platform is provided.

Security

1. Security advisory notification: BlueSpice security advisories are published at https://en.wiki.bluespice.com/wiki/Security:Security_Advisories.
   a. Future security advisories will be published using the Common Security Advisory Framework (CSAF), making them machine-readable and enabling automated monitoring and ingestion into customer security tooling.
2. Farm-level privilege audit log: Farm-level administrative actions — wiki creation, permission changes, and user grants — are written to a dedicated audit log. We evaluate support for forwarding to a SIEM.
3. Container image signing: Container image signing via Sigstore/Cosign is on the vendor roadmap, allowing customers with supply-chain security requirements to cryptographically verify image provenance before deployment into their internal registry.

Backup

1. Backup tooling at scale: The product-level backup tooling and its behaviour at ~10,000-wiki scale are documented. A joint assessment process for developing a fit-for-purpose backup solution for large Farm deployments is available and recommended.
2. Dedicated tooling for restoration of individual sub-wikis is not yet in development. This is expected to be addressed as a joint customer–vendor effort, scoped during the deployment planning phase.

Collaborative Services

1. wire — Redis/Valkey pub/sub fan-out: The current direct HTTP notification model — which prevents horizontal scaling of the wire service — is a known limitation. A Redis/Valkey pub/sub channel as the notification transport is on the product roadmap, with delivery planned for a named release.

2. wire — single pod connection ceiling: Until pub/sub fan-out is available, wire operates as a single pod. Vendor-observed connection count ceilings for the wire service under a single Node.js process are documented, along with recommended resource limits (memory, CPU) for deployments targeting several thousand concurrent connections.
3. collabpads — horizontal scaling roadmap: Support for a Redis/Valkey-backed socket.io adapter — or an equivalent multi-instance architecture — to remove the single-pod constraint on the collabpads service is on the product roadmap, with delivery planned for a named release.
4. collabpads — single pod sizing guidance: Vendor-recommended resource limits (memory, CPU) for a single collabpads pod handling several hundred concurrent collaborative WebSocket connections across a large Farm are documented, including known connection count ceilings under the current Node.js implementation.
5. MongoDB schema migrations: MongoDB schema changes are handled automatically on collabpads container startup and do not require manual intervention. Breaking MongoDB schema changes are not introduced in MINOR updates; any such changes in MAJOR updates are accompanied by a documented migration procedure.

8.2 Customer-side Decisions Required

The following items require a decision or design input from the customer before the corresponding architecture or runbook can be finalised:

Item	Decision required	Impact
Secondary cluster architecture	Replication topology, RPO target, and whether secondary serves as active pre-production or cold standby	Affects DR runbook, update validation procedure, and data replication design
Monitoring stack	Which Prometheus/Grafana/alerting platform the customer operates	Required to scope vendor exporter integration and dashboard development
Image registry architecture	Internal mirror, pull-through cache, or manual transfer for vendor image ingestion	Affects update delivery procedure and security advisory response time
SSO / IdP integration	SAML 2.0, OpenID Connect (OIDC), or LDAP/AD — which protocol(s) and which IdP	Affects auth configuration, network policy (IdP connectivity), and MFA enforcement

8.3 Architectural Gaps Requiring Further Design

The following items are flagged in this document as requiring further design work that goes beyond the scope of this whitepaper:

- DR architecture detail: The geo-redundant secondary cluster exists as a stated requirement but its topology, replication design, and failover runbooks are not yet defined. This is a first-class design workstream.
- Backup solution at scale: The backup strategy for 10,000 databases, the shared OpenSearch index, and S3 storage at production scale requires a dedicated design phase and likely custom tooling development. Standard product backup tooling is a starting point, not a complete solution.
- Migration orchestration tooling: The batched, parallel schema migration procedure described in Section 3.4 requires orchestration tooling. Whether this is provided by the vendor, co-developed, or built by the customer needs to be agreed.
- MAJOR update SOP: No SOP exists for a MAJOR version update. This must be developed jointly well in advance of the next MAJOR release.
- Image supply chain security: Until vendor-side image signing is available, the customer should establish an internal process for verifying image integrity at ingestion into the internal registry (e. g., digest pinning, checksum verification against vendor-published digests).
- Scaleable wire service: Allowing multiple pods of this service requires some architectural changes (including a possible introduction of an additional service) which are yet to be researched.
- Scaleable collabpads service: Allowing multiple pods of this service requires some architectural changes (including a possible introduction of an additional service) which are yet to be researched.