STARISIAN TECHNOLOGIES

StarCache

v3.0 Architectural Specification

The Extraction

Status: FINAL DRAFT — Ready for Implementation
Author: MaximillianGroup (Max Barrett) / Starisian Technologies
Supersedes: StarCache v2.1.1 (WordPress MU-Plugin only)
Repos: starisian/starcache-core · starisian/starcache-wp

1. Purpose and Scope

This document is the governing technical specification for StarCache v3.0, codenamed The Extraction. It defines the architecture, contracts, failure-mode handling, repository boundaries, and implementation sequence for extracting StarCache from a WordPress-only MU-Plugin into a general-purpose PHP caching library that powers the entire Starisian Technologies platform.

1.1 What Changes

StarCache v2.x is a WordPress MU-Plugin. Every class in the v2.x stack assumes WordPress functions, constants, and the wp_cache_* API are present. This is correct for the WordPress presentation layer but makes StarCache unusable by the pure PHP engine components — Helios, Sirus, and Dheghom — which run in CLI workers, standalone API nodes, and edge functions where WordPress does not exist.

StarCache v3.0 extracts the core logic — version-based key generation and adapter-based storage — into a standalone, dependency-free PHP library. The WordPress-specific orchestration becomes a thin consumer of that core. The SPARXSTAR engine components consume the same core directly.

1.2 What Does Not Change

The public API surface that WordPress themes and plugins use today — star_cache_get(), star_cache_set(), star_cache_remember(), star_cache_delete() — does not change. The WordPress bridge remains fully backward-compatible with v2.x call sites. No existing WordPress integration requires modification.

1.3 The Governing Principle

Correctness Boundary

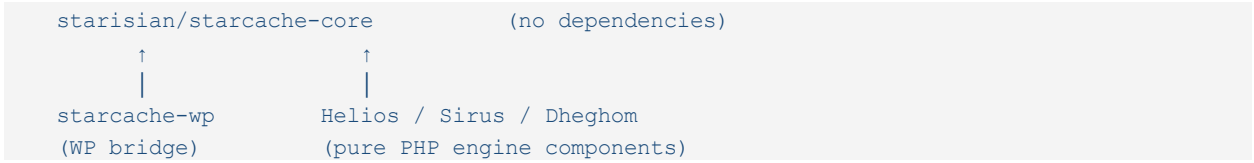
In the WordPress presentation layer, a cache correctness failure is a UX nuisance — a user sees a stale page. In Helios evaluating governance decisions, the same failure mode means a revoked permission or changed policy is served from cache as still-valid. That is not a UX problem. That is a sovereignty violation. The v3.0 architecture is designed against that failure mode, not the lesser one.

2. Repository Structure

StarCache v3.0 ships as two independent Composer packages with a strict dependency direction: the WordPress bridge depends on the core. The core has zero external dependencies.

Package	Contents and Constraints
starisian/starcache-core	The pure PHP caching engine. Contains StorageProviderInterface, StarVersionStore, StarCacheKey, and the concrete APCu, Redis, Filesystem, and Memory storage providers. Requires PHP 8.2+. Zero Composer dependencies.
starisian/starcache-wp	The WordPress bridge. Consumes starcache-core via Composer. Contains WordPressStorageProvider, StarPageCache, StarResponseController, StarAssetMinifier, StarCacheContext, and all WordPress hook registrations. Requires WordPress 6.0+ and PHP 8.2+.
Helios / Sirus / Dheghom	Consume starisian/starcache-core directly via Composer. Each engine injects its own StorageProvider at bootstrap. No WordPress dependency.

2.1 Dependency Direction



Boundary Rule

No class in starisian/starcache-core may reference any WordPress function, constant, or class. No class in starisian/starcache-core may call function_exists() to probe for WordPress APIs. The boundary is enforced by PHPStan at level 9 in the core CI pipeline with no WordPress bootstrap file.

3. StorageProviderInterface

This interface is the primary contract of the entire v3.0 architecture. All concrete storage implementations — Redis, APCu, Filesystem, Memory, WordPress — implement this interface. All consumers — StarVersionStore, StarCacheKey, Helios, Sirus — depend only on this interface, never on a concrete class.

3.1 Interface Definition

```
namespace StarCache\Core\Interfaces;

interface StorageProviderInterface
{
    public function get(string $key): mixed;

    public function set(string $key, mixed $value, int $ttl = 0): bool;

    public function delete(string $key): bool;

    /**
     * Existence check without deserialization.
     * Must not return true for expired keys.
     */
    public function has(string $key): bool;

    /**
     * Atomic increment. Required for race-free version bumping.
     * Implementations that cannot guarantee atomicity must document
     * the race window explicitly.
     */
    public function increment(string $key, int $offset = 1): int;

    public function flush(): bool;

    /**
     * Returns telemetry for admin tooling and health checks.
     * Minimum keys: name (string), status (string: connected|degraded|unavailable)
     *
     * @return array<string, mixed>
     */
    public function getStats(): array;
}
```

3.2 Method Contracts

Method	Contract	Notes
get()	Returns stored value or null on miss. Never throws. Null is a valid cached value — callers must use has() if they need to	Return mixed — never false for miss.

	distinguish a null value from a miss.	
set()	Stores value with optional TTL in seconds. TTL of 0 means no expiry for Redis/APCu; implementations must document their zero-TTL behavior.	Returns bool success.
has()	Existence check without deserialization overhead. Must return false for expired keys. Must not deserialize the value.	Performance-critical path.
increment()	Atomically increments an integer key by offset. If the key does not exist, initializes to offset. Redis: INCRBY. APCu: apcu_inc(). Filesystem/Memory: document the race window.	Returns new integer value.
flush()	Clears all keys managed by this provider instance. Scope is implementation-defined — a Redis provider scoped to a key prefix flushes only that prefix.	Use with care in shared environments.
getStats()	Returns name, status, and any implementation-specific metrics. Used by the WordPress admin bar and health-check endpoints.	Never throws.

4. StarVersionStore

StarVersionStore provides version-based cache invalidation without deleting entries from the storage backend. When a version group is bumped, all cache keys that embedded the previous version number become unreachable. Stale entries expire naturally on their own TTL. No flush is required and no thundering herd is triggered.

Why This Matters for Governance

When Helios evaluates a governance decision (Can User X access Resource Y?), the result is cached under a key that embeds the policy_version counter. When a policy changes, bumping policy_version makes all prior decisions unreachable in a single atomic operation. No enumeration of cached keys is required. The operation cost is O(1) regardless of how many decisions are cached.

4.1 Eviction-Proof Initialization

The critical failure mode in earlier versions: if Redis evicts a version key under memory pressure (maxmemory-policy allkeys-lru), the next read returns null, the store re-initializes to 1, and any cache entry built against a higher version silently becomes reachable again. This is a correctness violation in a governance context.

The fix uses hrtime(true) for initialization rather than a fixed constant. hrtime() returns a nanosecond-precision integer from a monotonic clock. If a version key is evicted and re-initialized, the new epoch will be a value such as 1715832948123456789 — numerically greater than any version number that could have been built by sequential increments since the system started. Old keys become unreachable orphans. They will never be served.

4.2 Implementation

```
namespace StarCache\Core;

use StarCache\Core\Interfaces\StorageProviderInterface;

class StarVersionStore
{
    public const GROUP_PAGES   = 'pages';
    public const GROUP_QUERIES = 'queries';
    public const GROUP_OBJECTS = 'objects';

    private const KEY_PREFIX   = 'sc_ver_';

    public function __construct(
        private readonly StorageProviderInterface $provider,
        private readonly string $tenantId = 'default'
    ) {}

    public function get(string $group): int
    {
        $key     = $this->buildKey($group);
        $version = $this->provider->get($key);
```

```
        if ($version === null) {
            // hrtime(true): nanosecond precision, monotonic clock.
            // Eviction-proof: new epoch is always > any increment sequence.
            $version = (int) hrtime(true);
            $this->provider->set($key, $version, 0);
        }

        return (int) $version;
    }

    public function bump(string $group): int
    {
        // Atomic increment via provider. Race-free on Redis and APCu.
        $newVersion = $this->provider->increment($this->buildKey($group));
        return $newVersion;
    }

    public function resetToTimestamp(
        ?string $group = null,
        ?int $timestamp = null  // Injectable for testing
    ): int {
        $ts = $timestamp ?? (int) hrtime(true);
        $groups = $group ? [$group] : [self::GROUP_PAGES, self::GROUP_QUERIES,
self::GROUP_OBJECTS];
        foreach ($groups as $g) {
            $this->provider->set($this->buildKey($g), $ts, 0);
        }
        return $ts;
    }

    private function buildKey(string $group): string
    {
        return self::KEY_PREFIX . $this->tenantId . '_' . md5($group);
    }
}
```

4.3 resetToTimestamp() Contract

resetToTimestamp() is the deliberate, auditable global invalidation operation. It is distinct from flush() — it does not clear the storage backend. It advances the version epoch so that all keys built against any prior version become unreachable. Stale entries drain on their TTL.

Parameter / Behavior	Definition
$group = null	Resets all three built-in groups (pages, queries, objects) in a single call. Use for full platform deploys or security events.
$group = 'policy_version'	Resets only the named group. All other groups remain at their current version.
$timestamp = null	Uses hrtime(true). Default for production.

$timestamp = (int)	Injectable value for test environments. Allows test suites to control epoch transitions without mocking hrtime.
Return value	The new epoch integer. Log this value in the audit trail for every call to resetToTimestamp() in production.
Idempotency	Two calls within the same process will always produce different values because hrtime() is monotonic. This guarantee holds within a single PHP process. Across concurrent processes, the last writer wins, which is the correct behavior.

5. StarCacheKey

StarCacheKey builds deterministic, collision-resistant, multisite-aware and multi-tenant-aware cache keys. Every key is a SHA-256 digest of a structured segment string. The raw segment string is preserved for debugging via buildDebug(). The hashed form is used for all storage operations.

5.1 Key Segments

Segment	Definition
namespace	Literal string "star". Prefixes all keys from this library.
tenantId	Caller-supplied tenant or environment identifier. WordPress bridge passes get_current_blog_id(). Engine components pass their own environment ID. Defaults to "default".
versionGroup	The version token from StarVersionStore for the named group. Format: v:{group}:{version}. Includes the group name to ensure keys for different groups cannot collide even when both are at the same version number.
reference	Caller-supplied logical identifier for the cached data. Maximum 250 characters. Validated on entry — empty or oversized references throw InvalidArgumentException.
userId	Optional caller-supplied user identifier. Defaults to "anon". Present in the segment string so user-scoped and anonymous keys are structurally distinct.
contextDimensions	Optional array of key-value pairs representing request context (device type, locale, experiment bucket, etc.). Array is ksorted before hashing to ensure identical context always produces the same hash regardless of insertion order.
salt	AUTH_KEY + SECURE_AUTH_SALT in WordPress environments. In pure PHP environments, an application secret injected at bootstrap. Prevents external key guessing.

5.2 Method Signatures

```
namespace StarCache\Core;

class StarCacheKey
{
    public const MAX_REFERENCE_LENGTH = 250;

    public function __construct(
        private readonly StarVersionStore $versionStore,
        private readonly string $salt,
        private readonly string $tenantId = 'default'
    ) {}

    /**
```

```
     * Build a storage-ready cache key (SHA-256 digest).
     *
     * @param array<string, scalar> $contextDimensions  ksorted before hashing.
     * @throws \InvalidArgumentException  On empty or oversized $reference.
     */
    public function build(
        string $reference,
        ?string $userId = null,
        string $versionGroup = StarVersionStore::GROUP_OBJECTS,
        array $contextDimensions = []
    ): string {
        $this->guardReference($reference);
        return hash('sha256', $this->buildRaw($reference, $userId, $versionGroup,
$contextDimensions));
    }

    /**
     * Build the human-readable segment string for logging and tracing.
     * Never use the output of this method as a storage key.
     */
    public function buildDebug(
        string $reference,
        ?string $userId = null,
        string $versionGroup = StarVersionStore::GROUP_OBJECTS,
        array $contextDimensions = []
    ): string {
        return $this->buildRaw($reference, $userId, $versionGroup, $contextDimensions);
    }
}
```

5.3 Key Construction

```
    private function buildRaw(
        string $reference,
        ?string $userId,
        string $versionGroup,
        array $contextDimensions
    ): string {
        $version = $this->versionStore->get($versionGroup);

        ksort($contextDimensions);
        $dimHash = !empty($contextDimensions)
            ? md5(json_encode($contextDimensions, JSON_THROW_ON_ERROR))
            : 'global';

        return implode(':', [
            'star',
            $this->tenantId,
            "v:{$versionGroup}:{$version}",
            $reference,
            $userId ?? 'anon',
            $dimHash,
            $this->salt,
```

```
            ]);
        }
```

Do Not Use buildDebug() as a Storage Key

The raw segment string produced by buildDebug() can exceed the 250-byte Memcached key limit under normal operating conditions (a nanosecond-precision version token is 19 digits alone). Only the SHA-256 output of build() is safe to use as a storage key. buildDebug() is for log lines and trace output only.

5.4 versionGroup Default

The $versionGroup parameter has no default value in the constructor signature. Callers must explicitly specify which version group their keys belong to. This is intentional. Providing a default would mean a caller who omits the parameter silently shares a version group with every other caller who also omitted it. A single bump of that shared group invalidates the entire cache.

The WordPress bridge uses StarVersionStore::GROUP_OBJECTS as its default for the data API layer (star_cache_get / star_cache_set). It uses GROUP_PAGES for page cache keys. It uses GROUP_QUERIES for SQL result keys. Engine components define their own group constants.

6. Concrete Storage Providers

Four concrete implementations of StorageProviderInterface ship with starisian/starcache-core. A fifth — WordPressStorageProvider — ships with starisian/starcache-wp.

6.1 RedisStorageProvider

Property	Detail
Package	starisian/starcache-core
PHP extension	ext-redis (predis/predis as optional fallback)
get()	Deserializes with unserialize($value, ['allowed_classes' => false]). Object injection hardened.
set()	Uses setEx() for TTL > 0, set() for TTL = 0.
increment()	Uses Redis INCRBY. Atomic. No race window.
has()	Uses Redis EXISTS. Does not deserialize.
Configuration	Host, port, password, database via constructor. No WordPress constants.
Eviction note	Operators running maxmemory-policy allkeys-lru must scope the StarCache keyspace to a dedicated Redis database or use a separate Redis instance to prevent version key eviction.

6.2 APCuStorageProvider

Property	Detail
Package	starisian/starcache-core
PHP extension	ext-apcu
Use case	Single-node Helios deployments. Fastest possible get/set latency. Not suitable for distributed deployments.
increment()	Uses apcu_inc(). Atomic within a single process.
has()	Uses apcu_exists(). Does not deserialize.
TTL = 0	APCu interprets TTL 0 as no expiry. This matches the Redis behavior and is safe for version keys.

6.3 FilesystemStorageProvider

Property	Detail
Package	starisian/starcache-core

Use case	Fallback for environments where neither Redis nor APCu is available. Acceptable for development. Not recommended for production at scale.
increment()	Uses flock() for mutual exclusion. Documents the race window: concurrent processes on NFS mounts may experience flock() failures. In this case increment() falls back to non-atomic read-modify-write and logs a warning.
has()	File existence check. Checks file mtime against TTL. Does not read file contents.

6.4 MemoryStorageProvider

Property	Detail
Package	starisian/starcache-core
Use case	Unit testing exclusively. In-process array storage. State does not persist between requests.
increment()	Non-atomic array increment. Race conditions are theoretically possible but irrelevant in single-threaded test execution.
Test pattern	Inject MemoryStorageProvider into StarVersionStore and StarCacheKey in test setUp(). Reset between tests via flush().

6.5 WordPressStorageProvider (in starcache-wp)

Property	Detail
Package	starisian/starcache-wp
Backend	Wraps wp_cache_get / wp_cache_set / wp_cache_delete. Backed by whatever object-cache drop-in is active (APCu, Redis via wp-redis, database fallback).
increment()	WordPress has no native atomic increment in the object cache API. WordPressStorageProvider uses wp_cache_get + wp_cache_set with a brief flock() guard. Documents the race window for shared Redis deployments.
autoDetect()	Static factory method. Probes Redis → Memcached → Memcache → WordPress object cache in order. Returns a concrete StorageProviderInterface. This is where the v2.x probe chain lives in v3.0.
has()	wp_cache_get() with $found parameter. Does not deserialize separately — WordPress always deserializes on get. This is a known inefficiency of the WordPress object cache API.

7. WordPress Bridge (starcache-wp)

The WordPress bridge is a thin consumer of starcache-core. Its sole responsibilities are: bootstrapping the correct StorageProvider at WordPress load time, registering WordPress hooks, and implementing the WordPress-specific cache behaviors (page cache output buffering, response headers, asset minification).

7.1 Bootstrap Sequence

The hook ordering in starcache.php is unchanged from v2.1.1. The only change is that StorageProvider injection replaces the inline adapter detection:

```
// starcache.php — bootstrap (abbreviated)

add_action('plugins_loaded', static function (): void {
    $provider = WordPressStorageProvider::autoDetect();
    $store    = new StarVersionStore($provider, (string) get_current_blog_id());
    $keyBuilder = new StarCacheKey($store, AUTH_KEY . SECURE_AUTH_SALT, (string)
get_current_blog_id());

    StarCache::boot($provider, $store, $keyBuilder);
}, 0);

add_action('plugins_loaded', [StarCacheContext::class, 'resolve'], 1);
add_action('init',           [StarPageCache::class,    'startPageCache'], 1);
add_action('send_headers',   static function (): void {
    StarCacheContext::lock();
    StarResponseController::apply();
}, 1);
```

7.2 Backward Compatibility

All global helper functions — star_cache_get(), star_cache_set(), star_cache_remember(), star_cache_delete() — remain in the global namespace and maintain their v2.x signatures. No breaking changes to any WordPress integration.

Function	v3.0 Behavior
star_cache_get()	Unchanged signature. Delegates to StarCache::getInstance()->star_getCachedData().
star_cache_set()	Unchanged signature. Delegates to star_setCachedData() or star_setCachedDataWithTtl().
star_cache_remember()	Unchanged signature. Get-or-set cache-aside pattern. Exceptions from $callback propagate unchanged.
star_cache_delete()	Unchanged signature. Delegates to star_deleteCachedData().
StarQueryCache	Remains deprecated. cachedWpdbQuery() utility is retained. The posts_pre_query and the_posts hooks remain unregistered. Removal target: v4.0.

8. Engine Integration (Helios, Sirus, Dheghom)

8.1 Helios: Governance Decision Caching

Helios evaluates complex RBAC/ABAC rules that are computationally expensive to recompute on every request. StarCache Core allows Helios to cache compiled policy results against a policy_version group. When a policy changes, a single bump of policy_version invalidates all prior decisions without enumerating or deleting cached keys.

```
// Helios bootstrap
$provider   = new RedisStorageProvider($host, $port, $password, $database);
$store      = new StarVersionStore($provider, $environmentId);
$keyBuilder = new StarCacheKey($store, $appSecret, $environmentId);

// Caching a governance decision
$key = $keyBuilder->build(
    reference:          'governance:can_access',
    userId:             $userId,
    versionGroup:       'policy_version',
    contextDimensions:  ['resource' => $resourceId, 'action' => $action]
);

$decision = $provider->get($key);
if ($decision === null) {
    $decision = $this->evaluatePolicy($userId, $resourceId, $action);
    $provider->set($key, $decision, ttl: 300); // 5-minute TTL
}

// On policy change — invalidate all cached decisions atomically
$store->bump('policy_version');
```

8.2 Sirus: Context Resolution Caching

Sirus assembles the resolved context object for each request by merging user data, environment variables, and session state. When a user's permissions are updated in Dheghom, Sirus bumps that user's specific version key. Cached context for that user is invalidated while the rest of the cache remains hot.

```
// Per-user context invalidation in Sirus
$store->bump('ctx_user_' . $userId);

// Key construction with user-scoped version group
$key = $keyBuilder->build(
    reference:    'context:resolved',
    userId:       $userId,
    versionGroup: 'ctx_user_' . $userId
);
```

8.3 Dheghom: Hydrated Entity Caching

Dheghom caches hydrated entity objects after mapping raw database rows to Star-compliant structures. When an entity type is updated, bumping the entity_type_version group ensures no stale objects are served, even if the underlying database row has not changed its structure.

```
// Entity type version bump on schema change
$store->bump('entity_' . $entityType);

// Key construction
$key = $keyBuilder->build(
    reference:    'entity:' . $entityType . ':' . $entityId,
    versionGroup: 'entity_' . $entityType
);
```

9. Failure Mode Reference

This section documents every known failure mode in the caching layer, its severity classification, and the mitigation built into v3.0.

Failure Mode	Severity	Mitigation
Redis version key eviction	CRITICAL in governance contexts. MINOR in WP presentation layer.	hrtime(true) initialization. New epoch is always > any prior increment sequence. Stale keys become unreachable orphans.
Concurrent version bump race	MINOR. At worst, two increments produce the same version number. One bump is lost.	StorageProviderInterface.increment() uses atomic INCRBY on Redis and apcu_inc() on APCu. Race window is zero on these backends.
Full Redis reboot	MODERATE. All version keys are lost simultaneously.	hrtime(true) initialization applies on first read after reboot. All prior cache entries become unreachable. Cache warms from cold. No stale data served.
StorageProvider unavailable	MINOR to MODERATE depending on fallback chain.	StarVersionStore.get() returns hrtime(true) on null. Effectively treats every key as a unique epoch — full cache miss behavior. No stale data. Performance degrades gracefully.
StarCacheKey.build() with oversized reference	MINOR. Caught at call site.	InvalidArgumentException thrown with message. Caller must handle. Key is never constructed.
json_encode() failure in contextDimensions	MINOR. Rare. Only on non-JSON-serializable values.	JSON_THROW_ON_ERROR flag. JsonException propagates to caller. Key is never constructed.
Memory provider in production	CRITICAL. State does not persist between requests.	getStats() returns a warning flag when MemoryStorageProvider is detected outside a test environment (defined by an STARCACHE_ENV constant). Health check endpoints surface this warning.

10. Implementation Sequence

The v3.0 extraction follows a strict sequence. No phase begins before the previous phase is complete and its artifacts are committed.

Phase 0 — Stabilize v2.2 (WordPress)

Merge the current PR #1 with the three required pre-merge fixes, then immediately apply:

1. Correct the README feature table to reflect actual behavior. The posts_pre_query / the_posts hooks are not registered. The WP-CLI flush command bumps versions, not storage.
2. Apply @runInSeparateProcess to testShouldBypassReturnsTrueWhenDoNotCachePage. The DONOTCACHEPAGE constant pollution will cause intermittent test failures.
3. Raise PHPStan from level 5 to level 9. No exceptions.
4. Rename StarAssetMinifier::minifyJs() to normalizeJs() and add a DOES NOT MINIFY header to the docblock.
5. Add method-level @deprecated tags to every public method in StarQueryCache, not only the class docblock.

Phase 1 — Initialize starcache-core Repository

6. Create the starisian/starcache-core repository.
7. Define StorageProviderInterface as the first and only commit. Treat it as frozen until two concrete implementations consume it.
8. Define composer.json: type library, php ^8.2, zero require dependencies.
9. Configure PHPStan at level 9 with no WordPress bootstrap. The CI pipeline must fail if any WordPress function is referenced.

Phase 2 — Port Core Logic

10. Port StarVersionStore with hrtime(true) initialization, atomic increment(), and resetToTimestamp().
11. Port StarCacheKey with the tenantId parameter, build() / buildDebug() split, ksorted context dimensions, and the explicit versionGroup requirement.
12. Implement MemoryStorageProvider. This is required before any other provider so that the core test suite can run without external dependencies.
13. Write the core test suite against MemoryStorageProvider. Version bump determinism, eviction simulation, key collision tests, and resetToTimestamp() epoch transition tests must all pass.

Phase 3 — Concrete Providers

14. Implement RedisStorageProvider. Atomic increment via INCRBY. Unserialize with allowed_classes: false.
15. Implement APCuStorageProvider. Atomic increment via apcu_inc.
16. Implement FilesystemStorageProvider. Document flock() race window explicitly.
17. Each provider ships with its own integration test that can be run against a live backend.

Phase 4 — Refactor WordPress Bridge

18. Add starisian/starcache-core as a Composer dependency of starcache-wp.
19. Implement WordPressStorageProvider with autoDetect() static factory.
20. Replace StarCacheAdapter with WordPressStorageProvider in all WordPress-side call sites.
21. Update starcache.php bootstrap to inject StorageProvider, StarVersionStore, and StarCacheKey as shown in Section 7.1.
22. Confirm all existing test suite tests pass without modification. The public API is unchanged.
23. Tag as starcache-wp v3.0.0.

Phase 5 — Engine Integration

24. Helios, Sirus, and Dheghom add starisian/starcache-core to their composer.json require blocks.
25. Each engine defines its own version group constants (policy_version, ctx_user_{id}, entity_{type}).
26. Each engine bootstraps its own StorageProvider at startup — RedisStorageProvider for distributed deployments, APCuStorageProvider for single-node.
27. Integration tests verify cross-engine version bump isolation: a bump in Helios does not invalidate Sirus cache entries.

11. Open Questions

The following items are deferred from v3.0 scope. They are documented here to prevent them from being re-litigated during implementation.

Item	Decision and Rationale
Persisted Version Store	Should StarVersionStore optionally write version counters to Dheghom to survive a full Redis reboot? DEFERRED to v4.0. The hrtime(true) initialization provides probabilistic correctness sufficient for v3.0. A synchronous DB read on every cache key construction is too high a cost to justify before production evidence of full-reboot correctness violations exists.
starcache_version_ttl filter	Should operators be able to set a non-zero TTL on version keys as a hedge against allkeys-lru eviction? DEFERRED. The correct operational fix is to configure Redis noeviction for the StarCache keyspace. A TTL-based version key introduces a predictable invalidation event that is worse than the random eviction it hedges against.
StarQueryCache removal	Removal target is v4.0, not v3.0. The cachedWpdbQuery() utility remains available. The posts_pre_query / the_posts hooks remain unregistered. Third-party code depending on the deprecated class must be notified via the v3.0 release notes.
Distributed lock for WordPressStorageProvider.increment()	The WordPress object cache API has no native atomic increment. The current flock() approach has a documented race window. A Redis-specific fast path (direct INCRBY bypassing the WP object cache) is a v3.1 optimization once RedisStorageProvider is in production.