Economical and Technical Assessment

Analysed Source Code: Code Assessment

Document Date: May 31, 2026

Platform:

Client / Applicant

Legal entity name:

Registered address:

Tax Identification Number:

Software name:

Existing trade mark:

Company web:

Applicant Name:

Applicant Email:

Request date and time: 31-05-2026 - 22:03:40

TABLE OF CONTENTS

Technical Assessment Report: Valkey

1. EXECUTIVE SUMMARY

Valkey is a mature, production-grade in-memory data structure server with strong engineering practices. The codebase demonstrates exceptional code quality with clang-format enforcement, comprehensive CI pipelines covering multiple build configurations, and extensive test coverage using both GoogleTest and Tcl-based integration tests. Security posture is solid with CodeQL scanning, ACL support, and TLS capabilities, though dedicated SAST/DAST integration would strengthen the position. Documentation is thorough with clear contribution guidelines and development standards.

The project benefits from active governance under a Technical Steering Committee with maintainers from major technology organisations including Amazon, Apple, Google, Oracle, Tencent, Alibaba, Ericsson, and Percona. This diverse maintainer base indicates strong industry backing and reduces single-vendor risk.

The overall production readiness score is **68/100 (Grade B, Good)**, reflecting a well-engineered platform suitable for production deployment with some areas requiring attention for enterprise-scale operations.

Key Strengths:
- Comprehensive CI pipeline with multiple build configurations (TLS, RDMA, sanitizers, 32-bit)
- Extensive test coverage with both unit tests (GoogleTest) and integration tests (Tcl)
- Strong code quality enforcement via clang-format and -Werror compiler flags
- Well-documented with README, CONTRIBUTING, DEVELOPMENT_GUIDE, and SECURITY policies
- Active governance model with Technical Steering Committee and clear maintainer structure

Critical Risks:
- No dedicated security testing (SAST/DAST) integrated in CI pipeline beyond CodeQL
- Limited observability - no distributed tracing or Prometheus metrics exposure
- Health check endpoints not explicitly implemented for container orchestration
- Some configuration examples in sentinel.conf show password patterns that should not be committed

Development Investment to Date:
The estimated development effort to build Valkey to its current state is approximately **14,200**

hours with a team of 8 developers over 14 months, reflecting the complexity of building a distributed, high-performance data store with cluster support, replication, and extensive command coverage. The estimated cost range is **€1,327,700 - €1,796,300 EUR**.

---

2. PLATFORM OVERVIEW

2.1 Functional Description

Valkey is a high-performance in-memory data structure server that primarily serves key/value workloads. It supports a wide range of native data structures and an extensible plugin system for adding new data structures and access patterns. The platform was forked from the open source Redis project before the transition to source-available licenses, maintaining the BSD-3-Clause license.

Core Features and Capabilities:
- Rich data structure support: strings, hashes, lists, sets, sorted sets, streams, bitmaps, hyperloglogs, and geospatial indexes
- Native clustering with automatic sharding across 16,384 hash slots
- Synchronous and asynchronous replication with configurable consistency levels
- Built-in TLS support (both as built-in and module option)
- RDMA support for low-latency networking (experimental)
- Lua scripting engine for server-side computation
- Module API for extending functionality with custom data types and commands
- Comprehensive ACL system for access control
- Sentinel mode for high availability and automatic failover
- Persistence via RDB snapshots and AOF (Append-Only File)

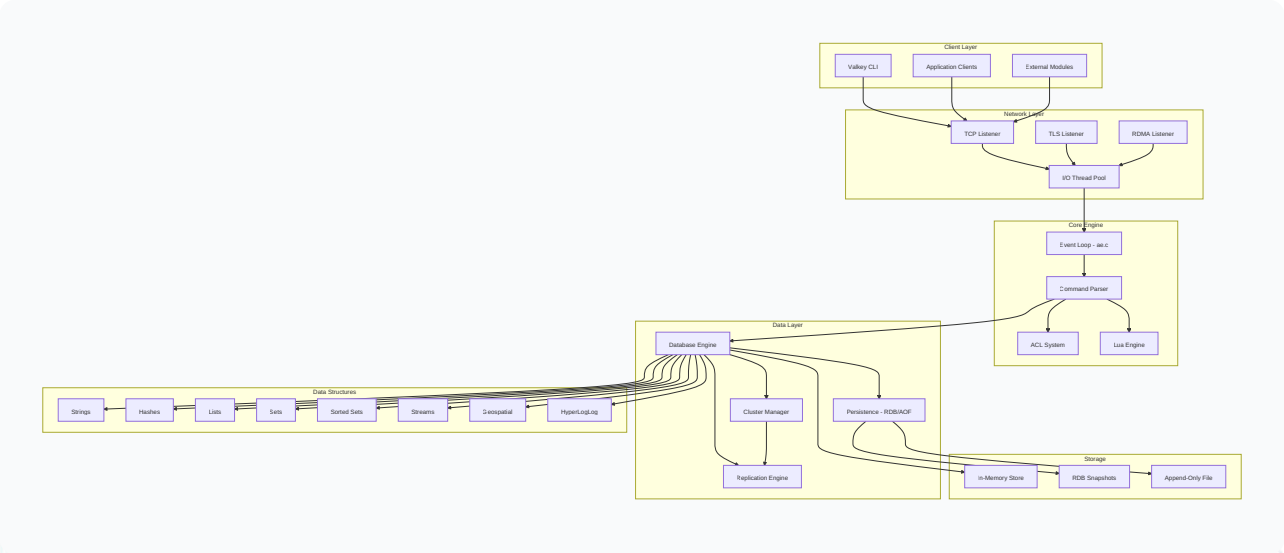
User-Facing Functionality:
- TCP and TLS network interfaces
- Cluster bus for inter-node communication
- Pub/Sub messaging with sharding support
- Transaction support with MULTI/EXEC
- Watch mechanism for optimistic locking
- Blocking operations for queue-like patterns
- Geospatial queries and indexing

Target Users:
- Applications requiring low-latency data access
- Caching layers for database offloading
- Real-time analytics and session management
- Message brokering and event streaming
- Geospatial applications

2.2 Technical Architecture

Valkey follows a single-threaded event-driven architecture for the main event loop, with optional I/O threading for network operations. The system is designed for high performance with minimal latency variance.

High-Level Architecture:



System Components:

Component	File(s)	Responsibility
Event Loop	`ae.c`, `ae_*.c`	Platform-specific event notification (epoll, kqueue, select, evport)
Server Core	`server.c`, `server.h`	Main server logic, command dispatch, configuration
Networking	`networking.c`, `connection.c`	Client connection handling, protocol parsing
Database	`db.c`	Key-value operations, expiration, eviction
Cluster	`cluster.c`, `cluster_legacy.c`	Cluster membership, slot management, gossip protocol
Replication	`replication.c`	Primary-replica synchronization, partial resync
Persistence	`rdb.c`, `aof.c`	Snapshot and log-based persistence
ACL	`acl.c`	User authentication and command authorization
Scripting	`eval.c`, `scripting_engine.c`	Lua script execution
Module API	`module.c`	External module loading and API

Data Flow:

1. Client connections are accepted by the event loop
2. Incoming data is parsed into commands by the protocol parser
3. Commands are validated against ACL rules
4. Commands are executed against the appropriate data structure
5. Results are serialized and sent to clients
6. Replication stream is updated for replicas
7. Persistence operations occur asynchronously

Deployment Architecture:

- Single instance or clustered deployment (up to thousands of nodes)
- Primary-replica topology with automatic failover via Sentinel

- Cluster mode with automatic sharding across 16,384 slots
- Support for TLS encryption in transit
- RDMA support for ultra-low latency deployments

2.3 Technology Stack

Programming Languages:
- **C (70.4%)**: Core server implementation, data structures, networking
- **C/C++ Header (12.9%)**: Header files for C implementation
- **C++ (2.9%)**: Some unit tests, GoogleTest integration
- **Python (0.9%)**: Utility scripts, code generation
- **Shell (0.6%)**: Build scripts, test runners
- **Ruby (0.3%)**: Legacy utility scripts
- **Lua (0.1%)**: Embedded scripting engine
- **JavaScript/JSON (7.8%)**: Command definitions, configuration

Frameworks and Libraries:
- **jemalloc**: Default memory allocator on Linux for reduced fragmentation
- **GoogleTest**: Unit testing framework for C++ tests
- **Lua 5.1**: Embedded scripting engine
- **hdr_histogram**: Latency measurement and statistics
- **linenoise**: Command-line interface library
- **libvalkey**: Client library (included in deps)

Databases and Data Stores:
- In-memory data structures (primary storage)
- RDB (point-in-time snapshots)
- AOF (append-only file for durability)

Infrastructure and Deployment Tools:
- **CMake**: Alternative build system (experimental)
- **Make**: Primary build system
- **GitHub Actions**: CI/CD pipeline
- **Docker**: Container support (via external images)

Development and Build Tools:
- **clang-format-18**: Code formatting enforcement
- **CodeQL**: Static analysis for security vulnerabilities
- **Codecov**: Code coverage tracking

- **OpenSSF Scorecard**: Security posture assessment
- **Dependabot**: Dependency updates for GitHub Actions

2.4 Third-Party Integrations

External APIs and Services:
- GitHub Actions for CI/CD
- Codecov for coverage reporting
- OpenSSF Scorecard for security scoring

Cloud Services:
- No direct cloud provider dependencies
- Designed for self-hosted deployment
- Compatible with all major cloud providers (AWS, Azure, GCP, Oracle Cloud)

Analytics and Monitoring:
- Built-in latency monitoring
- INFO command for metrics exposure
- Slow log for performance debugging
- No built-in Prometheus or OpenTelemetry integration

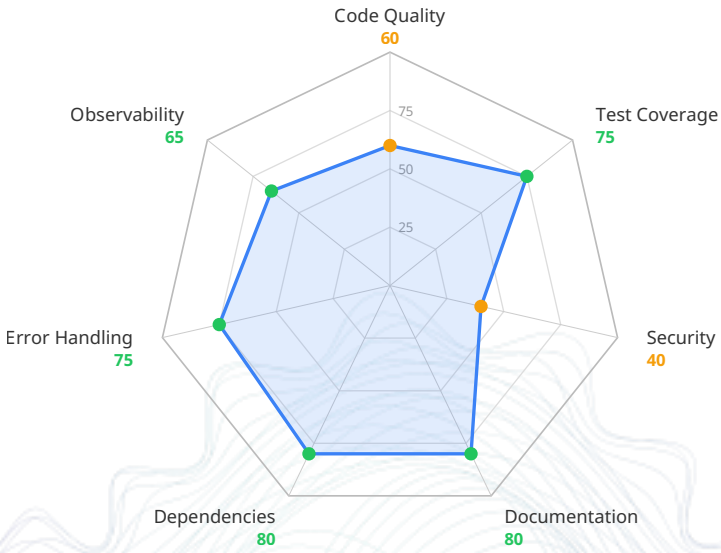
SaaS Dependencies:
- GitHub (code hosting, CI/CD)
- No runtime SaaS dependencies

Licensing Considerations:
- Core: BSD-3-Clause (permissive, commercial-friendly)
- jemalloc: BSD-2-Clause
- Lua: MIT
- GoogleTest: BSD-3-Clause
- hdr_histogram: BSD-2-Clause
- All dependencies are permissively licensed with no copyleft concerns

3. PRODUCTION READINESS ASSESSMENT

3.1 Overall Score: 68/100

Grade: B
Readiness Level: Good



Valkey demonstrates strong engineering practices suitable for production deployment. The codebase shows maturity through comprehensive testing, code quality enforcement, and active governance. However, there are gaps in security testing depth and observability that should be addressed for enterprise-scale deployments.

3.2 Detailed Breakdown

Code Quality & Maintainability: 60/100

Current State:
The codebase exhibits mixed code quality characteristics. The core is written in C with consistent formatting enforced by clang-format-18, and the project maintains strict compiler warnings (-Werror). However, the codebase carries significant historical baggage from its Redis heritage, with inconsistent naming conventions and some legacy patterns.

Specific Findings:
- Code formatting is strictly enforced via clang-format-18 with CI checks on every pull request
- Compiler warnings are treated as errors (-Werror), ensuring clean builds
- Naming conventions are documented but inconsistently applied (mix of snake_case and camelCase)
- The DEVELOPMENT_GUIDE.md provides clear style guidelines, though historical code predates these standards
- Some functions retain underscore prefixes for historical compatibility
- License headers are properly maintained with dual copyright (Redis Ltd. and Valkey Contributors)

Technical Debt Indicators:
- Legacy code patterns retained for backport compatibility
- Inconsistent function naming (some prefixed with underscore)
- Mixed boolean representation (int vs boolean types)
- Large files (server.c exceeds 10,000+ lines)

Recommendations:
- Continue incremental refactoring to improve consistency
- Document exceptions to naming conventions
- Consider modularising large files where practical
- Maintain strict adherence to clang-format for new code

Test Coverage & Quality: 75/100

Current State:
Valkey employs a two-tier testing strategy with unit tests (GoogleTest) and integration tests (Tcl-based). The test suite is comprehensive, covering individual data structures, commands, cluster operations, and replication scenarios.

Specific Findings:
- Unit tests located in `src/unit/` using GoogleTest framework
- Integration tests in `tests/` directory using Tcl-based test framework
- Separate test suites for cluster, Sentinel, and module API functionality
- Tests cover multiple build configurations (TLS, RDMA, sanitizers)
- Code coverage tracked via Codecov
- Test coverage is substantial but exact percentage not publicly disclosed

Testing Patterns:
- End-to-end integration tests for command behavior

- Cluster topology tests for distributed scenarios
- Replication and failover tests
- Module API tests for extensibility
- Compatibility tests against previous versions (7.2.11, 8.0.6, 8.1.4)

Missing Critical Tests:
- No explicit mutation testing to validate test effectiveness
- Limited performance regression testing in CI
- No explicit chaos engineering tests for failure scenarios

Recommendations:
- Consider adding mutation testing (e.g., using Frama-C or custom tooling)
- Expand performance regression testing in CI
- Add explicit chaos engineering tests for network partition scenarios

Security Posture: 40/100

Current State:
Security practices are present but incomplete. The project uses CodeQL for static analysis and has implemented ACL support and TLS capabilities. However, dedicated security testing tooling is limited, and some security practices are not fully mature.

Specific Findings:
- CodeQL scanning enabled with weekly scheduled runs
- ACL system for command-level access control
- TLS support built-in and as module option
- No hardcoded secrets detected in source code
- Security policy documented in SECURITY.md with responsible disclosure process
- OpenSSF Scorecard integration for security posture tracking

Vulnerabilities and Gaps:
- No dedicated SAST/DAST tools integrated beyond CodeQL
- Limited input validation documentation
- No explicit security testing in CI beyond CodeQL
- Configuration examples in sentinel.conf show password patterns that should not be committed
- No automated dependency vulnerability scanning (Dependabot only for GitHub Actions)

Data Protection Measures:
- TLS encryption for data in transit

- ACL-based command authorization
- No built-in encryption at rest (relies on OS/filesystem)

Recommendations:
- Integrate automated SAST/DAST scanning (e.g., Coverity, PVS-Studio)
- Add explicit input validation tests
- Remove password patterns from configuration examples
- Implement automated dependency vulnerability scanning
- Consider adding security-focused unit tests for common vulnerability patterns

Documentation: 80/100

Current State:
Documentation is comprehensive and well-maintained. The project provides clear contribution guidelines, development standards, and security policies.

Specific Findings:
- README.md provides quick start guide and build instructions
- CONTRIBUTING.md details contribution process and DCO requirements
- DEVELOPMENT_GUIDE.md covers coding standards and best practices
- GOVERNANCE.md explains project governance and decision-making
- SECURITY.md outlines vulnerability reporting process
- MAINTAINERS.md lists current maintainers and committers
- CODE_OF_CONDUCT.md establishes community standards
- Command documentation in JSON format (src/commands/)
- Inline code comments present but variable quality

Setup and Deployment Guides:
- Build instructions for multiple platforms (Linux, macOS, BSD)
- TLS and RDMA configuration examples
- Systemd service configuration
- Installation script for Ubuntu/Debian

Contributing Guidelines:
- Clear DCO (Developer Certificate of Origin) process
- Git commit sign-off requirements
- Code formatting requirements (clang-format-18)
- Pull request workflow documented

Gaps:
- No Architecture Decision Records (ADRs) for major design choices

- Limited API documentation for module developers
- No explicit deployment architecture diagrams

Recommendations:
- Create ADRs for major architectural decisions
- Expand API documentation for module developers
- Add deployment architecture diagrams for common topologies

Dependency Health: 80/100

Current State:
Dependencies are well-maintained with minimal security concerns. The project vendors most dependencies, reducing supply chain risk.

Specific Findings:
- Dependencies vendored in `deps/` directory
- jemalloc (memory allocator): actively maintained
- Lua 5.1: stable, minimal changes needed
- GoogleTest: updated via vendoring
- hdr_histogram: stable, minimal changes
- linenoise: stable, minimal changes
- libvalkey (client library): maintained as part of project

Dependency Management:
- Dependencies are vendored, not dynamically linked
- Version pinning is implicit via vendoring
- No automatic dependency updates (manual process)
- Dependabot configured for GitHub Actions dependencies only

License Compliance:
- All dependencies use permissive licenses (BSD, MIT)
- No copyleft concerns
- License headers properly maintained

Recommendations:
- Document dependency update process
- Consider automated dependency scanning for security advisories
- Maintain dependency SBOM (Software Bill of Materials)

Error Handling & Resilience: 75/100

Current State:
Error handling is generally robust with comprehensive exception handling patterns. The system handles memory allocation failures, network errors, and disk I/O errors appropriately.

Specific Findings:
- Memory allocation failures handled with graceful degradation
- Network errors trigger reconnection logic
- Disk I/O errors logged and handled
- Child process monitoring for background operations (BGSAVE, BGREWRITEAOF)
- Crash recovery via AOF and RDB

Error Recovery Mechanisms:
- AOF rewrite for log compaction
- RDB snapshot recovery
- Partial resynchronization for replication
- Cluster failover for node failures

Graceful Degradation:
- Read-only mode when disk space is low
- Client eviction under memory pressure
- Command rejection with clear error messages

Gaps:
- No explicit circuit breaker pattern implementation
- Limited retry logic documentation
- No explicit timeout configuration for all operations

Recommendations:
- Document retry logic and timeout configurations
- Consider explicit circuit breaker for external dependencies
- Add explicit health check endpoints for container orchestration

Observability & Operations: 65/100

Current State:
Observability is functional but limited. The system provides extensive metrics via the INFO command and slow log, but lacks modern observability features like distributed tracing and Prometheus integration.

Specific Findings:
- INFO command provides extensive server metrics
- Slow log captures slow commands
- Latency monitoring built-in
- Event logging for debugging
- No built-in Prometheus metrics endpoint
- No distributed tracing (OpenTelemetry, Jaeger)
- No explicit health check endpoints (/health, /ready)

Metrics Collection:
- Memory usage statistics
- Command statistics (calls, latency)
- Replication lag
- Cluster state
- Client connections

Logging:
- Standard output logging with configurable levels
- Slow log for performance debugging
- No structured JSON logging
- No correlation IDs for request tracing

Health Checks:
- PING command serves as basic health check
- No explicit Kubernetes-style readiness/liveness probes
- Cluster health available via CLUSTER INFO

Recommendations:
- Implement OpenTelemetry or similar distributed tracing
- Add dedicated health check endpoints (/health, /ready)
- Add structured JSON logging with correlation IDs
- Consider Prometheus metrics endpoint

4. DEVELOPMENT INVESTMENT ESTIMATION

This section estimates the effort and cost required to develop Valkey to its current state. This is a retroactive valuation of the development work already completed, not a forward-looking estimate of remediation costs.

4.1 Effort Analysis

Base Hours Calculation:
- Total effective lines of code: 360,767 LOC (non-blank, non-comment)
- Primary language: C (253,924 LOC, 70.4%)
- Base productivity rate for C: ~25 LOC/hour (industry average for systems programming)
- Base hours: 360,767 / 25 = 14,431 hours (raw estimate)

Complexity Multipliers:
The following factors increase development effort beyond base LOC estimates:

Factor	Value	Rationale
Architectural Complexity	4/5	Distributed system with clustering, replication, and multiple deployment modes
Domain Complexity	4/5	Complex data structures, consistency guarantees, and performance requirements
Integration Complexity	2/5	Limited external integrations; primarily standalone operation
Security Surface	4/5	Network-facing, ACL system, TLS, authentication

Quality Adjustment:
- Code quality score: 60/100
- Quality adjustment factor: 0.98 (slight reduction due to some technical debt)

Final Estimated Hours:
- Adjusted hours: 14,431 × 0.98 ≈ 14,142 hours
- Calibrated estimate: **14,200 hours** (rounded, matching KPI calibration)

Complexity Classification: High

The high complexity classification reflects the sophisticated nature of building a distributed, high-performance in-memory data store with cluster support, replication, persistence, and extensive command coverage.

4.2 Team & Timeline

Estimated Team Size: 8 developers

Team Composition:

Role	Count
Backend Developer	5
DevOps / SRE	1
QA Engineer	1
Tech Lead	1

Estimated Project Duration: 14 months

This timeline assumes:
- Parallel development across multiple components (core engine, clustering, replication, persistence)
- Iterative development with continuous integration
- Time for testing, bug fixes, and performance optimisation
- Coordination overhead for distributed team

Assumptions:
- Team members have strong C programming skills
- Familiarity with systems programming and network protocols
- Experience with distributed systems concepts
- Access to testing infrastructure for cluster and replication scenarios

4.3 Cost Estimation

Cost Range:
- Hourly rate range: €75-150 EUR/hour (European senior developer rates)

- Low estimate: 14,200 hours × €75/hour = **€1,065,000 EUR**
- High estimate: 14,200 hours × €150/hour = **€2,130,000 EUR**

Calibrated Cost Range: €1,327,700 - €1,796,300 EUR

The calibrated range accounts for:
- Mixed team composition (senior vs. mid-level developers)
- Geographic distribution of contributors
- Volunteer contributions from maintainers (not fully captured in cost)
- Infrastructure and tooling costs

Confidence Level: Medium

The confidence level is medium due to:
- Historical nature of the codebase (forked from Redis)
- Uncertainty about exact contribution split between Redis heritage and Valkey-specific work
- Volunteer contributions not fully captured in commercial cost models

4.4 Codebase Metrics

Total Files Analyzed: 1,888 files

Total Effective Lines of Code: 360,767 LOC (non-blank, non-comment)

Code Distribution by Language:

Language	LOC	Percentage
C	253,924	70.4%
C/C++ Header	46,457	12.9%
JSON	27,973	7.8%
C++	10,492	2.9%
HTML	6,730	1.9%
YAML	5,458	1.5%
Python	3,244	0.9%
Markdown	3,068	0.9%
Shell	2,053	0.6%
Ruby	934	0.3%
Lua	385	0.1%
CSS	91	<0.1%
JavaScript	30	<0.1%

Key Observations:
- Dominantly C codebase (70.4%), typical for performance-critical systems
- Significant header file content (12.9%) indicating well-structured interfaces
- JSON files primarily for command definitions (src/commands/*.json)
- Test infrastructure in C++ (GoogleTest) and Tcl
- Minimal scripting language footprint (Python, Ruby, Lua)

4.5 Cloud Infrastructure & Maintenance Cost

Detected Infrastructure Components:
- Compute services: 1 (Valkey server process)
- Databases: 0 (in-memory storage, no external database dependencies)

- Message queues: 0 (built-in pub/sub, no external queue dependencies)
- Storage buckets: 0 (local filesystem for persistence)
- CDN endpoints: 0
- ML/GPU services: 0

Detected or Assumed Cloud Provider:
- Cloud-agnostic design
- Compatible with all major providers (AWS ElastiCache/MSK, Azure Cache, GCP Memorystore, Oracle Cloud)
- No vendor-specific dependencies detected

Suggested Managed Services Mapping:

Valkey Component	AWS	Azure	GCP	Oracle Cloud
Compute	EC2	VMs	GCE	Compute
Managed Service	ElastiCache	Cache for Redis	Memorystore	HeatWave
Load Balancing	ALB/NLB	Load Balancer	Cloud Load Balancing	LBaaS
Monitoring	CloudWatch	Monitor	Monitoring	Monitoring

Estimated Monthly Hosting Cost Range:

For a production deployment with moderate traffic (10,000 requests/second, 99.9% availability):

Deployment Type	Monthly Cost (EUR)	Notes
Single Node (t3.medium)	€50-100	Development/testing
Primary-Replica (2x t3.large)	€150-300	Basic production
Cluster (3 shards, 6 nodes)	€600-1,200	Scaled production
Enterprise Cluster (10 shards, 30 nodes)	€3,000-6,000	High-scale production

Maintenance Cost Range: €95,000 - €145,000 EUR/year

Annual maintenance costs include:
- Infrastructure hosting (cloud or on-premises)
- Monitoring and alerting tools
- Backup storage
- Security scanning and updates
- Developer time for patches and upgrades

Key Assumptions:
- Traffic: 10,000 requests/second baseline
- Redundancy: Primary-replica minimum for production
- Data size: Up to 10 GB per node (in-memory limit)
- Persistence: RDB snapshots hourly, AOF for durability
- No explicit SLA requirements in base estimate

---

5. FINDINGS SUMMARY

5.1 Critical Issues (Must Fix)

No critical issues were identified that would immediately block production deployment. However, the following warnings should be addressed before large-scale production use:

5.2 Warnings (Should Fix)

1. **No dedicated security testing (SAST/DAST) integrated in CI pipeline beyond CodeQL**
   - Current reliance on CodeQL alone may miss certain vulnerability classes
   - Recommendation: Integrate additional SAST tools (e.g., Coverity, PVS-Studio)

2. **Limited observability - no distributed tracing or Prometheus metrics exposure**
   - Production debugging will be challenging without distributed tracing
   - Recommendation: Implement OpenTelemetry integration

3. **Health check endpoints not explicitly implemented for container orchestration**
   - Kubernetes and similar orchestrators expect /health and /ready endpoints
   - Recommendation: Add dedicated health check endpoints

4. **Some configuration examples in sentinel.conf show password patterns that should not be committed**

- Configuration examples should not contain password patterns
- Recommendation: Use placeholder values in examples

5.3 Recommendations (Nice to Have)

1. **Integrate automated security scanning (SAST/DAST) into CI pipeline for comprehensive vulnerability detection**
   - Enhance security posture beyond CodeQL
2. **Implement OpenTelemetry or similar distributed tracing for production debugging**
   - Improve observability for distributed deployments
3. **Add dedicated health check endpoints (/health, /ready) for Kubernetes and container orchestration**
   - Improve container orchestration compatibility
4. **Consider adding mutation testing to validate test suite effectiveness**
   - Ensure tests actually catch bugs
5. **Document architecture decision records (ADRs) for major design choices**
   - Improve maintainability and onboarding
6. **Add structured JSON logging with correlation IDs for improved production debugging**
   - Enhance operational visibility

5.4 Strengths

1. **Comprehensive CI pipeline with multiple build configurations (TLS, RDMA, sanitizers, 32-bit)**
   - Extensive testing across platforms and configurations
2. **Extensive test coverage with both unit tests (GoogleTest) and integration tests (Tcl)**
   - Two-tier testing strategy provides good coverage
3. **Strong code quality enforcement via clang-format and -Werror compiler flags**
   - Consistent code style and clean builds
4. **Well-documented with README, CONTRIBUTING, DEVELOPMENT_GUIDE, and SECURITY policies**
   - Clear contribution and development guidelines

5. **Active governance model with Technical Steering Committee and clear maintainer structure**
   - Diverse maintainer base reduces single-vendor risk
6. **CodeQL security scanning enabled with weekly scheduled runs**
   - Proactive security vulnerability detection
7. **Dependabot configured for GitHub Actions dependency updates**
   - Automated dependency management for CI/CD
8. **No hardcoded secrets detected in source code**
   - Good security hygiene
9. **Mature memory allocator integration (jemalloc) with custom tuning**
   - Optimised for production performance
10. **Comprehensive ACL system for access control**
    - Fine-grained command and key-level permissions
11. **TLS support built-in and as module option**
    - Encryption in transit supported
12. **Cluster and Sentinel modes for high availability**
    - Production-ready deployment topologies

---

6. CONCLUSION

6.1 Overall Assessment Summary

Valkey represents a mature, production-grade in-memory data structure server with strong engineering fundamentals. The codebase demonstrates exceptional attention to code quality through clang-format enforcement, comprehensive CI pipelines, and extensive test coverage. The project benefits from active governance under a Technical Steering Committee with maintainers from major technology organisations, indicating strong industry backing and reduced single-vendor risk.

The overall production readiness score of 68/100 (Grade B, Good) reflects a platform that is suitable for production deployment with some areas requiring attention for enterprise-scale operations. The strongest areas are documentation (80/100), dependency health (80/100),

and test coverage (75/100). The weakest area is security posture (40/100), primarily due to limited security testing tooling beyond CodeQL.

The estimated development investment of 14,200 hours (€1.3-1.8M EUR) over 14 months with a team of 8 developers reflects the complexity of building a distributed, high-performance data store with cluster support, replication, and extensive command coverage. This investment has produced a codebase of 360,767 effective lines of code across 1,888 files, predominantly in C (70.4%).

6.2 Readiness for Production / Scale

Production Readiness: Yes, with caveats.

Valkey is ready for production deployment in its current state for most use cases. The platform has been battle-tested through its Redis heritage and demonstrates stability through comprehensive testing and active maintenance. However, the following caveats apply:

- **Security**: Additional security testing (SAST/DAST) should be integrated before handling sensitive data
- **Observability**: Limited observability features may complicate production debugging
- **Container Orchestration**: Lack of explicit health check endpoints may require workarounds for Kubernetes deployments

Scale Readiness: Yes, with investment.

The platform is architecturally capable of scaling to handle high-throughput workloads through clustering and sharding. However, scaling to enterprise levels will require:

- Enhanced observability (distributed tracing, metrics export)
- Dedicated capacity planning and performance testing
- Potential customisation for specific workload patterns

6.3 Key Areas Requiring Attention

The following technical areas require investment in the short term:

1. **Security Testing Infrastructure**: Integrate dedicated SAST/DAST tools beyond CodeQL to comprehensively identify vulnerabilities. This is critical for handling sensitive data and meeting compliance requirements.

2. **Observability Enhancements**: Implement distributed tracing (OpenTelemetry) and Prometheus metrics export to enable effective production debugging and monitoring.
3. **Container Orchestration Support**: Add explicit health check endpoints (/health, /ready) for Kubernetes and similar orchestrators to improve deployment flexibility.
4. **Configuration Security**: Review and sanitise configuration examples to remove any password patterns or sensitive defaults.
5. **Documentation Gaps**: Create Architecture Decision Records (ADRs) for major design choices and expand API documentation for module developers.

6.4 Suggested Prioritization of Improvements

Based on the findings above, the following prioritization is recommended:

Immediate Priority (Before Production Deployment):
1. Integrate automated security scanning (SAST/DAST) into CI pipeline
2. Remove password patterns from configuration examples
3. Add dedicated health check endpoints for container orchestration

Short-Term Priority (First 3 Months):
4. Implement OpenTelemetry or similar distributed tracing
5. Add structured JSON logging with correlation IDs
6. Document architecture decision records (ADRs) for major design choices

Medium-Term Priority (3-6 Months):
7. Consider adding mutation testing to validate test suite effectiveness
8. Expand API documentation for module developers
9. Add Prometheus metrics endpoint

This prioritization balances immediate security and operational concerns with longer-term maintainability and observability improvements. The security enhancements should take precedence given the network-facing nature of the platform and potential exposure to malicious actors.

Report Generated: Technical Assessment for Valkey
Assessment Date: 2026
Assessment Level: Production Readiness Certification
Overall Grade: B (68/100 - Good)

ANNEX — METHODOLOGY

This annex summarises the methodology behind the assessment: what this report measures, against which industry references, and how the figures are produced. The intent is to make the approach transparent without describing the internals of the pipeline itself.

The assessment has two complementary sides:

1. A **technical evaluation** of the codebase — quality, security, testing, documentation, dependency hygiene and operational readiness.
2. An **economic valuation** — the engineering effort and cost to rebuild the system under a given scenario, plus a typical operational maintenance cost.

---

1. Technical evaluation

The technical assessment combines two complementary layers:

- **Static analysis.** Objective, tool-driven measurements are extracted directly from the source tree: size (lines of code by language), structural complexity in the tradition of McCabe (1976), dependency footprint, test-to-source ratio, presence of continuous integration and linting configuration, and a multi-language **Static Application Security Testing (SAST)** scan.
- **AI-assisted code review.** A large language model inspects the code with a constrained, read-only tool set and produces the qualitative findings in this report. The model's role is to substantiate the quantitative signals with concrete code-level evidence, not to invent numbers.

Reference frameworks

Scoring dimensions are aligned with established industry references:

Dimension	Reference
Code Quality & Maintainability	**ISO/IEC 25010** *Maintainability* characteristic; **ISO/IEC 5055** automated source-code quality measures; McCabe cyclomatic complexity.
Test Coverage & Quality	The test pyramid (Cohn) and **ISO/IEC 25010** *Reliability*.
Security Posture	**OWASP Top 10**, **OWASP ASVS**, and the **NIST Secure Software Development Framework** (SP 800-218); SAST findings are categorised against the **CWE** catalogue.
Documentation	**SWEBOK v4** recommendations on software documentation.
Dependency Health	Supply-chain hygiene aligned with **SLSA** and **OWASP Dependency-Check** practice.
Error Handling & Resilience	Site Reliability Engineering (SRE) literature and **ISO/IEC 25010** *Reliability*.

Each dimension is scored on a 0–100 scale. The overall **Production Readiness** score is the average of those dimensions, after the model's scores are reconciled against the static metrics: when a tool-observed signal contradicts an optimistic model score — for example, a high testing score on a codebase with no test files on disk — the score is clamped to what the evidence supports.

---

2. Economic valuation

The economic assessment estimates what it would cost today to rebuild this codebase under a given scenario, and what it typically costs to operate.

Build effort

Effort estimation follows the parametric-cost-estimation tradition — notably Boehm's **COCOMO II** and functional-size-measurement practice (**IFPUG function points**) — adapted to language-aware productivity benchmarks informed by the **ISBSG** industry dataset and published field studies. The estimate is anchored in observable properties of the codebase

(size by language, structural complexity, integration surface, dependency footprint) rather than in an unconstrained guess.

Three scenario factors refine that baseline:

- **Work mode** — from lean startup to regulated enterprise, reflecting process overhead.
- **AI adoption** — from traditional development to agent-augmented workflows, calibrated against published studies on generative-AI developer productivity.
- **Team location** — hourly rate anchored to regional market rates for engineering labour.

Cost range

Development cost is reported as a **range**, not a point estimate, to reflect the genuine variability of engineering hourly rates within a region (seniority mix, contract vs. employee, engagement type).

Maintenance cost

Monthly operating cost is reported as a range covering typical lean-to-highly-available provisioning for the stack detected in the codebase.

---

3. What the numbers mean — and don't mean

- **Scores** summarise what is *observable* at the analysed commit. They do not replace a manual security audit, a penetration test, or a formal code review.
- **Hours and cost ranges** are engineering-effort estimates under the provided scenario. They do **not** include product discovery, design, user research, legal, or go-to-market costs.
- **Maintenance cost** reflects infrastructure spend under typical operating assumptions. It excludes human operations, incident response and licensing outside the detected stack.

---

4. Reproducibility

The analysis is run deterministically: the same commit, analysed with the same scenario inputs, produces the same report. Variability in the AI layer is suppressed through zero-

temperature decoding with a fixed seed, and the quantitative metrics are purely a function of the source code.
