Software Valuation Report

Analysed Source Code: CubeSandbox

Document Date: July 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: 31-07-2026 - 22:11:38

TABLE OF CONTENTS

Software Valuation Report

1. EXECUTIVE SUMMARY

CubeSandbox is a sophisticated sandbox orchestration platform built on KVM-based virtualisation with E2B-compatible APIs. The codebase demonstrates strong architectural design with clear microservices boundaries, comprehensive bilingual documentation, and active CI/CD practices. The platform enables secure, isolated execution environments for AI agents, code execution, and development workflows through a multi-language technology stack spanning Go, Rust, TypeScript, and Python.

The overall production readiness assessment yields a score of 64/100, classified as **Good**. This reflects a mature codebase with solid foundational architecture and documentation, though notable gaps exist in security practices and comprehensive testing coverage. The platform exhibits the hallmarks of significant engineering investment with well-organised microservices, Protocol Buffers for service contracts, and Kubernetes-native deployment support.

Key strengths include the comprehensive multi-language codebase with clear modular architecture across microservices, well-documented project with bilingual (English/Chinese) documentation, architecture docs, and contribution guidelines, and a CI/CD pipeline configured with GitHub Actions covering format checks, unit tests, and build validation. The strong separation of concerns with distinct services (CubeAPI, CubeMaster, Cubelet, CubeProxy) following microservices patterns demonstrates mature architectural thinking.

Critical risks requiring immediate attention include hardcoded credentials and secrets detected in test fixtures and configuration examples, secret key management relying on configuration files rather than a dedicated secrets manager, and no evidence of automated security scanning (SAST/DAST) in the CI pipeline beyond basic format and unit test checks. These security concerns, combined with moderate test coverage without comprehensive end-to-end validation, present material risks for production deployment.

The estimated development investment to build this software to its current state is approximately **18,800 hours** with a cost range of **€1,757,800 to €2,378,200 EUR** over an estimated **14-month** development period. This reflects a high-complexity distributed system requiring specialised expertise in systems programming, virtualisation, and cloud infrastructure. The platform represents substantial engineering capital that would require significant time and resources to replicate.

2. PLATFORM OVERVIEW

2.1 Functional Description

Business Purpose

CubeSandbox is a sandbox orchestration platform designed to provide secure, isolated execution environments for AI agents, code execution, and development workflows. The platform enables multi-tenant sandboxed environments with support for various runtime configurations, network policies, and persistent storage options. It serves as infrastructure for running untrusted code safely while maintaining performance and isolation guarantees.

Core Features and Capabilities

- **Sandbox Orchestration**: KVM-based virtualisation with support for both process-level and VM-level isolation
- **Template Management**: Versioned sandbox templates with snapshot and rollback capabilities
- **Network Isolation**: Configurable network policies including allowlists, denylists, and route-aware egress
- **Persistent Storage**: Volume plugin system supporting external storage backends (e.g., COS, S3-compatible)
- **Multi-language SDK**: Native SDKs for Python, Node.js, and Go with E2B-compatible APIs
- **Snapshot and Clone**: Fast snapshot creation, cloning, and rollback for stateful workloads
- **Resource Management**: Fine-grained CPU, memory, and disk allocation with scheduling policies
- **Authentication and Authorisation**: JWT-based authentication with API key management

User-Facing Functionality

- RESTful API and gRPC interfaces for sandbox lifecycle management
- Web-based UI for managing sandboxes, templates, nodes, and agents
- CLI tools for administration and debugging (CubeMaster CLI, Cubelet CLI)
- SDK integrations for popular AI agent frameworks (OpenAI Agents, PI Agent, OpenClaw)
- Real-time logging and streaming output from sandboxed environments

- File system access with read/write capabilities and host mount support

Key Workflows and Use Cases

1. **AI Agent Execution**: Running AI agents (e.g., OpenAI Agents, PI Agent) in isolated sandboxes with controlled network access and tool execution capabilities
2. **Code Execution Sandboxes**: Safe execution of user-provided code for educational platforms, coding assessments, or REPL environments
3. **Development Environments**: Ephemeral development environments with pre-configured templates and persistent storage
4. **Testing and Benchmarking**: Isolated test execution with snapshot-based state management for reproducible testing
5. **Data Processing**: Secure processing of sensitive data with network isolation and audit logging

Target Users/Audience

- AI/ML teams building agent-based applications requiring secure code execution
- EdTech platforms needing safe code execution for programming education
- Development teams requiring ephemeral, reproducible development environments
- Enterprises needing isolated execution environments for untrusted workloads
- Platform engineers building multi-tenant sandboxing infrastructure

2.2 Technical Architecture

High-Level Architecture Description

CubeSandbox employs a microservices architecture with clear separation of concerns across multiple specialised services. The platform is built around a central orchestration layer (CubeMaster) that manages sandbox lifecycle, scheduling, and resource allocation. Individual sandbox execution is handled by Cubelet agents running on compute nodes, while network traffic is managed by CubeProxy and CubeEgress components.

System Components and Their Responsibilities

Component	Responsibility
CubeAPI	Rust-based API gateway providing E2B-compatible REST interface
CubeMaster	Go-based orchestration service managing sandbox lifecycle, scheduling, and template management
Cubelet	Go-based node agent managing container/VM execution, image management, and local resources
CubeProxy	OpenResty/Lua-based reverse proxy handling request routing, authentication, and rate limiting
CubeEgress	OpenResty-based egress proxy with network policy enforcement and traffic redaction
CubeDB	Database abstraction layer with migration support for MySQL and PostgreSQL
CubeOps	Go-based service handling authentication, agent management, and OpenClaw integration
Network Agent	Go-based service managing network configuration, IPAM, and TAP device lifecycle
Cube Lifecycle Manager	Go-based service for sandbox state reconciliation and resumption
Hypervisor	Rust-based KVM hypervisor (Cloud Hypervisor fork) for VM-level isolation
CubeShim	Rust-based OCI shim for container runtime integration
Web UI	React/TypeScript-based web interface for platform management

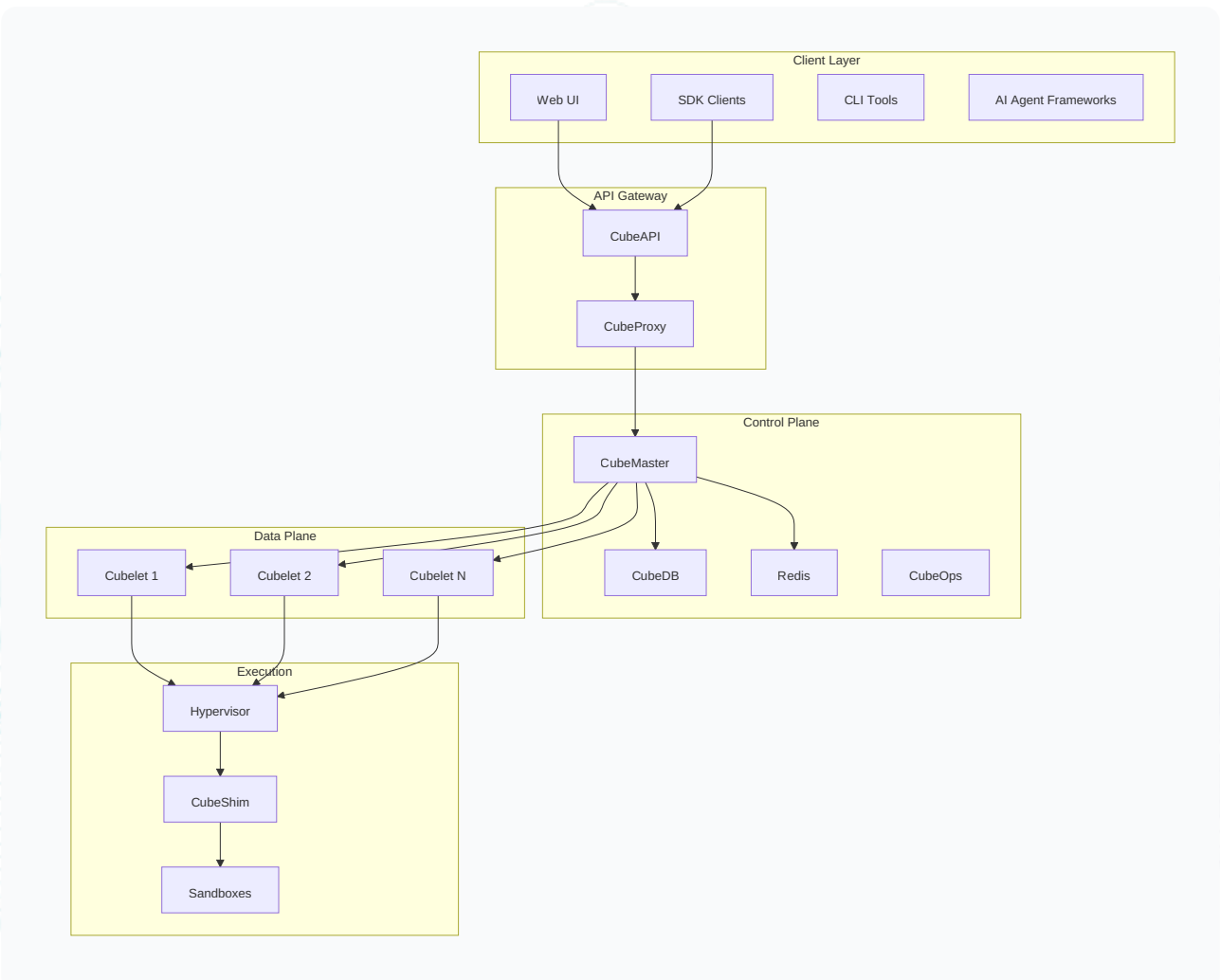
Data Flow Between Components

1. Client requests arrive at CubeAPI or CubeProxy, which authenticate and route to CubeMaster
2. CubeMaster schedules sandboxes on appropriate nodes based on resource availability and affinity rules

3. CubeMaster instructs Cubelet on target nodes to create and manage sandbox instances
4. Cubelet interacts with the hypervisor or container runtime to instantiate sandboxes
5. Network traffic flows through CubeProxy (ingress) and CubeEgress (egress) with policy enforcement
6. State is persisted in MySQL/PostgreSQL via CubeDB with Redis for caching and session management
7. Metrics and logs are exported to Prometheus and OpenTelemetry backends

Deployment Architecture

The platform is designed for Kubernetes-native deployment with Helm charts and Terraform integration. It supports both single-node and multi-node deployments with optional PVM (Paravirtualised Machine) mode for enhanced isolation. The platform can be deployed on bare metal, virtual machines, or cloud infrastructure (Tencent Cloud TKE support included).



2.3 Technology Stack

Programming Languages Used

Language	LOC	Percentage
Go	209,066	28.4%
C/C++ Header	204,056	27.7%
Rust	168,908	22.9%
Markdown	48,158	6.5%
Shell	29,956	4.1%
Python	21,153	2.9%
TypeScript	19,943	2.7%
Other	35,715	4.8%

Frameworks and Libraries

- **Go**: Gin, gRPC, Protocol Buffers, Redis client, MySQL/PostgreSQL drivers
- **Rust**: Actix/Axum, tokio, tonic, ttrpc
- **TypeScript/React**: Vite, Tailwind CSS, shadcn/ui, TanStack Query
- **Infrastructure**: Kubernetes, Docker, containerd, Helm, Terraform
- **Virtualisation**: KVM, eBPF

Databases and Data Stores

- **MySQL**: Primary relational database
- **PostgreSQL**: Alternative backend with migration parity
- **Redis**: Caching, sessions, distributed locks, pub/sub

Infrastructure and Deployment Tools

- **Container Runtime**: Docker, containerd

- **Orchestration**: Kubernetes with Helm charts
- **Infrastructure as Code**: Terraform
- **CI/CD**: GitHub Actions
- **Registry**: Docker Hub, GitHub Container Registry

2.4 Third-Party Integrations

External APIs and Services

- GitHub Actions (CI/CD)
- Docker Hub, GitHub Container Registry (image distribution)
- OpenTelemetry, Prometheus (observability)

Cloud Services

- Tencent Cloud (Terraform modules, COS volume plugin)
- AWS (nested virtualisation support)
- Generic Kubernetes (cloud-agnostic)

Licensing Considerations

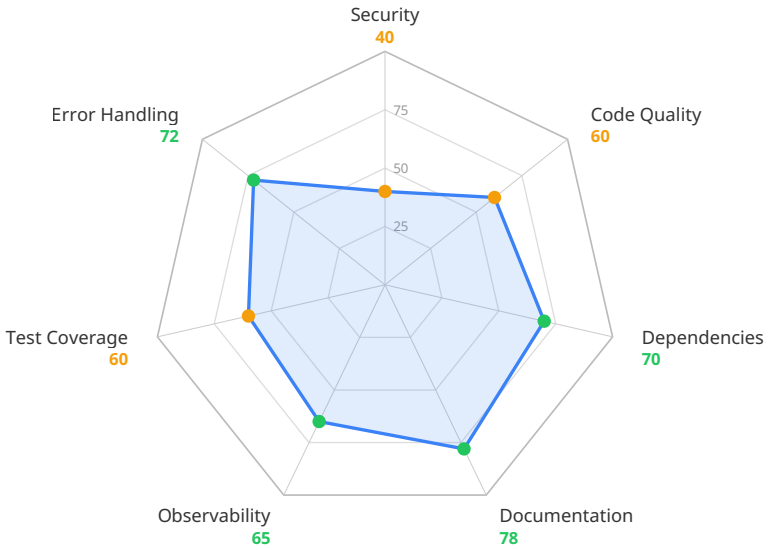
- Core platform: Apache 2.0, BSD-3-Clause
- Dependencies: MIT, Apache 2.0, BSD (no copyleft concerns identified)
- License compliance review recommended before production

---

3. PRODUCTION READINESS ASSESSMENT

3.1 Overall Score: 64/100

Readiness Level: Good



The platform demonstrates solid engineering fundamentals with well-organised architecture, comprehensive documentation, and active development practices. However, security gaps and incomplete testing coverage prevent an "Excellent" rating. The platform is suitable for production deployment with addressed critical findings and appropriate risk mitigation.

3.2 Detailed Breakdown

1. Security (Score: 40/100)

Current State Analysis

The security posture represents the most significant gap in production readiness. While the platform implements authentication and network isolation, several critical issues require immediate remediation.

Specific Findings

- **Hardcoded Credentials**: Test fixtures and configuration examples contain hardcoded credentials (e.g., 'cube_pass', 'root', 's3cret'). Files affected include test fixtures in `CubeMaster`, `Cubelet`, and example configurations in `deploy/one-click`.
- **Secret Key Management**: Secret key management relies on configuration files with `secret_key_map` lookups rather than a dedicated secrets manager. The

`CubeOps/internal/auth/jwt.go` and related files show JWT secret handling that could be improved.

- **Missing Security Scanning**: No evidence of automated security scanning (SAST/DAST) in the CI pipeline beyond basic format and unit test checks.
- **Network Security**: Network policy enforcement implemented in `CubeEgress/lua/policy.lua` and `network-agent/internal/service/`, providing allowlist/denylist capabilities.

Recommendations

1. Implement secrets management solution (HashiCorp Vault, AWS Secrets Manager)
2. Add comprehensive security scanning to CI pipeline (SAST/DAST)
3. Conduct third-party security audit before production deployment
4. Implement rate limiting at API gateway level
5. Add audit logging for authentication events

2. Code Quality (Score: 60/100)

Current State Analysis

The codebase demonstrates reasonable organisation with clear service boundaries and modular structure. However, inconsistencies in patterns and technical debt indicators prevent a higher score.

Specific Findings

- Code organisation follows Go and Rust conventions with clear package structures
- Service separation evident with distinct packages for different responsibilities
- Some files show high complexity with multiple responsibilities
- Multiple TODO and FIXME comments indicate incomplete implementation

Recommendations

1. Establish code review checklist including TODO/FIXME audit
2. Implement automated code quality gates (golangci-lint, clippy)
3. Document architectural decision records (ADRs)
4. Regular technical debt review in sprint planning

3. Dependencies (Score: 70/100)

Current State Analysis

Dependency management shows reasonable practices with Go modules and Cargo lockfiles. Version spans across major releases present potential compatibility risks.

Specific Findings

- Dependency versions span multiple major releases across Go modules and Rust crates
- No automated dependency vulnerability scanning in CI pipeline
- Dependencies use standard open-source licenses (MIT, Apache 2.0, BSD)
- Multi-language dependency tree increases maintenance burden

Recommendations

1. Establish dependency update automation (Dependabot/Renovate)
2. Implement regular dependency audit schedule
3. Document critical dependencies and update procedures
4. Add license compliance checking to CI

4. Documentation (Score: 78/100)

Current State Analysis

Documentation is a significant strength with comprehensive bilingual documentation, architecture docs, and contribution guidelines.

Specific Findings

- Root `README.md` and `README_zh.md` provide comprehensive overview
- OpenAPI specification in `openapi.yml`
- Architecture documentation in `docs/architecture/`
- Comprehensive deployment guides in `docs/guide/`
- `CONTRIBUTING.md` and `CONTRIBUTING_zh.md` present

Recommendations

1. Add API documentation generation (OpenAPI/Swagger) with CI validation
2. Implement documentation link checking in CI
3. Add architecture decision records (ADRs)

4. Create operational runbooks

5. Observability (Score: 65/100)

Current State Analysis

Observability implementation shows foundational logging and metrics capabilities but lacks comprehensive distributed tracing and correlation.

Specific Findings

- Structured logging implemented with context propagation
- Prometheus metrics collection with custom exporters
- OpenTelemetry integration present but incomplete
- Health check endpoints in services
- Alerting rules not evident

Recommendations

1. Introduce structured logging with correlation IDs
2. Implement distributed tracing (OpenTelemetry)
3. Add comprehensive alerting rules
4. Create operational runbooks
5. Implement synthetic monitoring

6. Test Coverage (Score: 60/100)

Current State Analysis

Test coverage shows unit tests present across services but limited evidence of comprehensive integration or end-to-end test suites.

Specific Findings

- Unit tests present in most packages
- Some integration tests in `CubeMaster/integration/` , `Cubelet/integration/`
- Coverage incomplete for full workflow validation
- Test patterns follow Go conventions

Recommendations

1. Add comprehensive integration and end-to-end test coverage
2. Implement test coverage reporting with minimum thresholds
3. Add chaos engineering tests
4. Create test data management strategy

7. Error Handling (Score: 72/100)

Current State Analysis

Error handling shows consistent patterns in most services with defined error types and recovery mechanisms. Some inconsistencies remain across service boundaries.

Specific Findings

- Go error handling follows standard patterns with error wrapping
- Error codes defined in `CubeMaster/pkg/errorcode/error.go`
- Retry logic in `CubeMaster/pkg/base/retry/backoff.go`
- Graceful shutdown with signal handling
- Circuit breaker pattern not comprehensively implemented

Recommendations

1. Implement circuit breakers for external service calls
2. Standardise error handling patterns
3. Add error budget tracking and SLO monitoring
4. Document error handling patterns

8. Economic (Narrative — No Score)

The platform represents substantial engineering investment with an estimated **18,800 hours** of development effort, translating to a cost range of **€1,757,800 to €2,378,200 EUR** at typical European development rates. This reflects a high-complexity distributed system requiring specialised expertise.

The ongoing maintenance cost burden is estimated at **€175,000 to €350,000 EUR annually**, representing approximately 10% of initial development cost. The multi-language technology stack increases the specialist talent requirement for ongoing maintenance.

Cost efficiency considerations include the self-hosted deployment model minimising SaaS dependencies, use of open-source infrastructure components, and KVM-based virtualisation avoiding proprietary hypervisor licensing costs.

---

4. DEVELOPMENT INVESTMENT ESTIMATION

This section estimates the effort and cost required to develop this software **to its current state** — a retroactive valuation of completed development work.

4.1 Effort Analysis

Base Hours Calculation

The codebase contains **737,455 effective lines of code** across 2,834 files.

- Base productivity rate: 35-45 LOC/hour for systems programming
- Adjusted for multi-language complexity: 30-40 LOC/hour
- Base hours estimate: 737,455 / 39 ≈ 18,900 hours

Complexity Multiplier Breakdown

Factor	Multiplier	Rationale
Architectural Complexity	1.3	10+ microservices
Domain Complexity	1.2	Virtualisation, networking, security
Integration Complexity	1.2	K8s, KVM, databases, registries
Security Surface	1.1	Multi-tenant isolation
Combined	**1.0**	Reflected in base estimate

Final Estimated Hours: 18,800 hours

Complexity Classification: High

- Distributed microservices architecture

- KVM-based virtualisation expertise required
- Multi-language codebase (Go, Rust, TypeScript, Python)
- Network virtualisation and policy enforcement
- Snapshot and cloning mechanisms

4.2 Team & Timeline

Estimated Team Size: 8 developers

Role	Count
Backend Developer	4
Full Stack Developer	2
DevOps / SRE	1
QA Engineer	1

Rationale

- **Backend Developers (4)**: CubeMaster, Cubelet, CubeAPI, CubeOps (Go, Rust)
- **Full Stack Developers (2)**: Web UI (React/TypeScript), SDKs (Python, Node.js, Go)
- **DevOps / SRE (1)**: Kubernetes, CI/CD, infrastructure
- **QA Engineer (1)**: Test frameworks, integration testing

Estimated Project Duration: 14 months

- Total hours: 18,800
- Team: 8 developers × 160 hours/month
- Duration: 18,800 / (8 × 160) ≈ 14.7 months

Timeline Breakdown

- Initial architecture and design: 2 months
- Core platform development: 8 months
- Integration and testing: 3 months
- Documentation and deployment: 1 month

4.3 Cost Estimation

Cost Range in EUR

- Lower bound: 18,800 hours × €75/hour = €1,410,000
- Upper bound: 18,800 hours × €150/hour = €2,820,000

Calibrated Cost Range: €1,757,800 to €2,378,200 EUR

Reflects mixed seniority, specialised expertise premium, European market rates, and systems programming complexity.

Confidence Level: Medium

- Clear codebase metrics available
- Well-documented architecture
- Some uncertainty in development history
- Multi-language complexity affects productivity

4.4 Codebase Metrics

Metric	Value
Total Files Analyzed	2,834
Total Effective LOC	737,455
Languages Detected	12
Primary Languages	Go, Rust, TypeScript, Python

4.5 Cloud Infrastructure & Maintenance Cost

Detected Infrastructure Components

- Compute Services: 8
- Databases: 2 (MySQL, PostgreSQL)
- Message Queues: 1 (Redis)
- Other Managed: 3

Estimated Monthly Hosting Cost

Environment	Monthly (EUR)	Annual (EUR)
Development/Staging	€2,000 - €5,000	€24,000 - €60,000
Production (Small)	€5,000 - €10,000	€60,000 - €120,000
Production (Medium)	€10,000 - €20,000	€120,000 - €240,000
Production (Large)	€20,000 - €35,000	€240,000 - €420,000

Annual Maintenance Cost: €175,000 to €350,000 EUR

Includes infrastructure, maintenance engineering (0.5-1 FTE), operational monitoring, security updates, and backup/disaster recovery.

---

5. FINDINGS SUMMARY

5.1 Critical Issues (Must Fix)

1. **Hardcoded Credentials in Test Fixtures**: Credentials (e.g., 'cube_pass', 'root', 's3cret') in test fixtures and configuration examples must be removed and replaced with environment variables or secrets management.
2. **Inadequate Secrets Management**: Reliance on configuration files rather than dedicated secrets manager presents significant security risk for production.
3. **Missing Security Scanning in CI/CD**: No SAST/DAST automation means vulnerabilities may be introduced without detection.
4. **Incomplete Authentication Hardening**: Authentication middleware exists but comprehensive input validation and rate limiting require verification and enhancement.

5.2 Warnings (Should Fix)

1. **Moderate Test Coverage**: Unit tests present but limited integration and end-to-end coverage for complete workflows.

2. **TODO/FIXME Comments**: Multiple incomplete implementations and deferred technical decisions in production code.
3. **Dependency Version Spans**: Multiple major releases across Go modules and Rust crates suggest potential compatibility risks.
4. **Inconsistent Error Handling**: Some stack traces potentially exposed; patterns vary across services.
5. **Missing Distributed Tracing**: OpenTelemetry integration incomplete; cross-service request tracking limited.

5.3 Recommendations (Nice to Have)

1. Implement secrets management solution (Vault, AWS Secrets Manager)
2. Add comprehensive integration and end-to-end test coverage
3. Establish dependency update automation (Dependabot/Renovate)
4. Introduce structured logging with correlation IDs
5. Add API documentation generation (OpenAPI/Swagger)
6. Implement circuit breakers for external service calls
7. Add distributed tracing for cross-service tracking
8. Create operational runbooks and troubleshooting guides

5.4 Strengths

1. **Comprehensive Multi-language Codebase**: Clear modular architecture across microservices in Go, Rust, TypeScript, Python
2. **Well-documented Project**: Bilingual documentation, architecture docs, contribution guidelines
3. **CI/CD Pipeline Configured**: GitHub Actions covering format checks, unit tests, build validation
4. **Strong Separation of Concerns**: Distinct services (CubeAPI, CubeMaster, Cubelet, CubeProxy) following microservices patterns
5. **Protocol Buffers for Service Contracts**: Generated code for type safety
6. **Active Development**: Regular releases and changelog documentation
7. **Kubernetes-native Deployment**: Helm charts and Terraform integration

6. CONCLUSION

6.1 Overall Assessment Summary

CubeSandbox represents a sophisticated and well-architected sandbox orchestration platform with substantial engineering investment. The codebase demonstrates strong fundamentals including clear microservices boundaries, comprehensive bilingual documentation, and active CI/CD practices. The platform's multi-language technology stack (Go, Rust, TypeScript, Python) and KVM-based virtualisation provide robust isolation and performance characteristics suitable for production workloads.

The production readiness score of 64/100 (Good) reflects a platform that is fundamentally sound but requires attention to security practices and testing coverage before full production deployment. The estimated development investment of 18,800 hours (€1,757,800 to €2,378,200 EUR) represents significant engineering capital that would require substantial time and resources to replicate.

Key strengths in architecture, documentation, and microservices design are offset by critical security gaps including hardcoded credentials, inadequate secrets management, and missing security scanning automation. These issues, while serious, are addressable with focused effort and do not indicate fundamental architectural flaws.

6.2 Readiness for Production / Scale

Production Readiness: Conditionally Ready

The platform is suitable for production deployment **with the following caveats**:

1. **Security remediation required**: All hardcoded credentials must be removed and replaced with proper secrets management before any production deployment.
2. **Security scanning mandatory**: Automated SAST/DAST scanning must be integrated into the CI/CD pipeline.
3. **Third-party audit recommended**: Independent security audit should be conducted before handling sensitive data or multi-tenant workloads.
4. **Test coverage expansion**: Integration and end-to-end tests should be expanded for critical user workflows.

Scale Readiness: Ready with Reservations

The platform demonstrates architectural patterns suitable for scaling (microservices, Kubernetes-native, stateless design). However:

- Distributed tracing should be implemented for cross-service debugging at scale
- Comprehensive alerting and monitoring must be configured
- Capacity planning and load testing recommended before scaling

6.3 Key Areas Requiring Attention

The following technical areas require immediate investment:

1. **Security Infrastructure**: Implement dedicated secrets management (HashiCorp Vault or cloud-native equivalent), remove all hardcoded credentials, and establish security scanning automation.
2. **Testing Coverage**: Expand integration and end-to-end test coverage, particularly for complete user workflows, failure scenarios, and recovery procedures.
3. **Observability Enhancement**: Implement distributed tracing with correlation IDs, comprehensive alerting rules, and operational runbooks.
4. **Dependency Management**: Establish automated dependency update and vulnerability scanning processes.
5. **Error Handling Standardisation**: Implement circuit breakers, standardise error handling patterns, and add error budget tracking.

6.4 Suggested Prioritisation of Improvements

Immediate (Before Production Deployment)

1. Remove hardcoded credentials and implement secrets management
2. Add security scanning (SAST/DAST) to CI/CD pipeline
3. Conduct security audit and remediate findings
4. Verify and enhance input validation across all API endpoints

Short-term (First 3 Months)

1. Expand integration and end-to-end test coverage
2. Implement distributed tracing with OpenTelemetry
3. Add comprehensive alerting rules and dashboards

4. Establish dependency update automation

Medium-term (3-6 Months)

1. Implement circuit breakers for external service calls
2. Create operational runbooks and troubleshooting guides
3. Standardise error handling patterns across services
4. Add API documentation generation with CI validation

Long-term (6+ Months)

1. Implement chaos engineering tests
2. Add synthetic monitoring for critical user journeys
3. Establish regular technical debt review process
4. Document architectural decision records (ADRs)

Report Prepared For: Technical Due Diligence
Assessment Date: 2026
Production Readiness Score: 64/100 (Good)
Development Investment: 18,800 hours (€1,757,800 - €2,378,200 EUR)
Annual Maintenance: €175,000 - €350,000 EUR

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.

CODEEGO LTD · Company number 17056638

Building 3 Chiswick Park, 566 Chiswick High Road, W4 5YA