CODEEGO

# Software Valuation Report

## Analysed Source Code: [redacted]

**Document Date:** June 09, 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:** 09-06-2026 - 09:07:23

# TABLE OF CONTENTS

**1. Executive Summary**

**2. Platform Overview**

- 2.1 Functional Description
- 2.2 Technical Architecture
- 2.3 Technology Stack
- 2.4 Third-Party Integrations

**3. Production Readiness Assessment**

- 3.1 Overall Score: 76/100
- 3.2 Detailed Breakdown

**4. Development Investment Estimation**

- 4.1 Effort Analysis
- 4.2 Team & Timeline
- 4.3 Cost Estimation
- 4.4 Codebase Metrics
- 4.5 Cloud Infrastructure & Maintenance Cost

**5. Findings Summary**

- 5.1 Critical Issues (Must Fix)
- 5.2 Warnings (Should Fix)
- 5.3 Recommendations (Nice to Have)
- 5.4 Strengths

**6. Conclusion**

- 6.1 Overall Assessment Summary
- 6.2 Readiness for Production / Scale
- 6.3 Key Areas Requiring Attention
- 6.4 Suggested Prioritisation of Improvements

# Software Valuation Report

## 1. EXECUTIVE SUMMARY

The ██████ platform demonstrates strong architectural maturity with a well-structured multi-service design implementing domain-driven principles across three Python services and a React frontend. The platform provides AI-powered code assessment capabilities that evaluate codebases for production readiness, estimate development investment, and generate certified compliance reports through qualified digital custody (TrustCloud Vault).

Security posture is robust with secrets managed via GCP Secret Manager, parameterised queries throughout, and comprehensive input validation via Pydantic across all API endpoints. Documentation is exceptional with extensive design records, API specifications (OpenAPI/Swagger), operational runbooks, and compliance mapping documentation for SOC 2 and ASVS 5.0.0 standards.

The overall production readiness score is **76/100 (Very Good)**. The primary gaps preventing an 'Excellent' rating are the absence of an enforced CI/CD pipeline in the repository (Taskfile.yml defines CI tasks but no GitHub Actions workflow files present), lack of metrics exposition endpoints for production monitoring, and no distributed tracing implementation despite structured logging via structlog.

The codebase reflects approximately **8,970 hours** of development investment by a team of four engineers over **12 months**, with high complexity driven by multi-service architecture, compliance domain requirements, and numerous external integrations (Stripe, TrustCloud, SendGrid, multiple AI providers, GitHub OAuth, GCP services). The estimated development cost ranges from **€838,695 to €1,134,705**.

**Key Strengths:**
- Clean domain-driven design architecture with clear separation between domain, application, and infrastructure layers
- Comprehensive documentation including README, API documentation, development setup guides, and extensive design/plan documents
- Secrets properly managed via Google Secret Manager with no hardcoded credentials in source code
- Terraform infrastructure-as-code for all GCP resources including Cloud Run, Cloud SQL, and GCS
- Extensive test suites across all packages with unit, integration, and security tests

**Critical Risks:**
- No CI/CD pipeline configured in repository — Taskfile.yml defines CI tasks but no GitHub Actions or similar workflow files present to enforce linting, testing, or security scans on pull requests
- Test coverage not enforced with gates in CI — while tests exist across all packages, there is no coverage threshold enforcement in the build pipeline
- No distributed tracing implemented — structured logging exists via structlog but no OpenTelemetry or similar tracing integration for production debugging

---

# 2. PLATFORM OVERVIEW

## 2.1 Functional Description

**Business Purpose:**

**Core Features and Capabilities:**

**Key Workflows:**

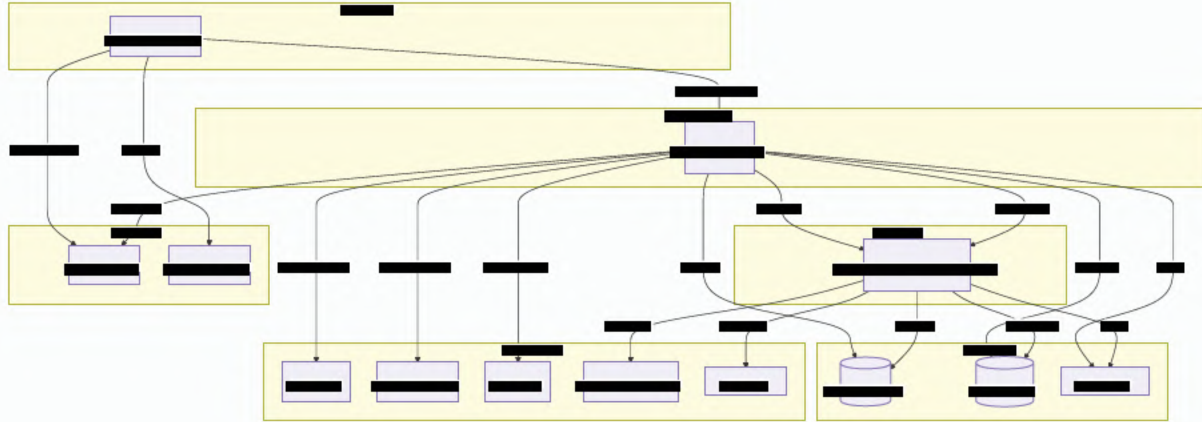
**Target Users:**

## 2.2 Technical Architecture

**High-Level Architecture:**

The platform follows a microservices architecture deployed on Google Cloud Run with three primary backend services and a single-page application frontend. Services communicate via

synchronous HTTP APIs with internal authentication, and asynchronous callbacks for long-running operations.



**System Components and Responsibilities:**

| Component | Technology | Responsibility |
| --- | --- | --- |
| **Frontend** | React 18 + Vite + TypeScript | Single-page application with Radix UI components, TanStack Query for data fetching, Firebase SDK for client-side authentication |
| [illegible] | [illegible] | [illegible] |
| [illegible] | [illegible] | [illegible] |
| **Ory Kratos** | Go (containerised) | Identity and session management with email OTP authentication, session verification API |
| **PostgreSQL** | Cloud SQL (v15) | Primary data store with asyncpg driver, Alembic migrations, connection pooling |
| **Google Cloud Storage** | GCS FUSE mount | Workspace storage for uploaded code, analysis artefacts, generated reports, compliance evidence binders |
| **Terraform** | HCL | Infrastructure-as-code for all GCP resources including Cloud Run services, Cloud SQL, GCS buckets, VPC networking, IAM bindings |

**Data Flow:**

ail

**Deployment Architecture:**

- **Region:** europe-west1 (GCP)
- **Compute:** Cloud Run (serverless containers) with VPC connectors for internal service communication
- **Database:** Cloud SQL for PostgreSQL with private IP connectivity
- **Storage:** Cloud Storage bucket with FUSE mount for Analysis Service workspace
- **Secrets:** Secret Manager for all sensitive configuration (database credentials, API keys, signing secrets)
- **Frontend Hosting:** Firebase Hosting with CDN distribution

## 2.3 Technology Stack

**Programming Languages:**

| Language | Lines of Code | Percentage |
|---|---|---|
| Python | 113,387 LOC | 50.7% |
| Markdown | 77,298 LOC | 34.5% |
| TypeScript | 21,724 LOC | 9.7% |
| YAML | 6,215 LOC | 2.8% |
| JSON | 2,742 LOC | 1.2% |
| HTML | 1,319 LOC | 0.6% |
| Shell | 925 LOC | 0.4% |
| JavaScript | 118 LOC | 0.1% |
| CSS | 88 LOC | <0.1% |
| **Total** | **223,816 LOC** | **100%** |

**Frameworks and Libraries:**

| Layer | Technology | Purpose |
|---|---|---|
| **Backend (Python)** | FastAPI | REST API framework with async support and OpenAPI schema generation |
| | SQLAlchemy 2.0 (async) | ORM with asyncpg driver for PostgreSQL |
| | Pydantic 2.x | Data validation and settings management |
| | Alembic | Database migration management |
| | structlog | Structured logging for production observability |
| | slowapi | Rate limiting middleware |
| | httpx | Async HTTP client for external service calls |
| **Frontend** | React 18 | Component-based UI framework |
| | Vite | Build tool and dev server |
| | TypeScript | Type-safe JavaScript |
| | TailwindCSS | Utility-first CSS framework |
| | Radix UI | Unstyled, accessible UI primitives |
| | TanStack Query | Data fetching and caching |
| | Recharts | Analytics and visualisation charts |
| | Firebase SDK | Client-side authentication |
| **Infrastructure** | Terraform | Infrastructure-as-code for GCP resources |
| | Docker | Containerisation for Cloud Run deployment |
| | Ory Kratos | Identity and session management |

**Databases and Data Stores:**

| Store | Technology | Purpose |
| --- | --- | --- |
| **Primary Database** | PostgreSQL 15 (Cloud SQL) | |
| **File Storage** | Google Cloud Storage | |
| **Session Store** | PostgreSQL (Kratos) | Ory Kratos identity and session data (separate database) |

**Infrastructure and Deployment Tools:**

| Tool | Purpose |
| --- | --- |
| Google Cloud Run | Serverless container execution with auto-scaling |
| Cloud SQL | Managed PostgreSQL with automated backups and failover |
| Cloud Storage | Object storage with FUSE mount for POSIX-like access |
| Secret Manager | Secure storage for API keys, database credentials, signing secrets |
| VPC Network | Private networking between Cloud Run services and Cloud SQL |
| Cloud Build | Container image building via cloudbuild.yaml |
| Firebase Hosting | Static asset hosting with CDN for frontend |
| Taskfile | Local CI/CD orchestration (lint, test, build, deploy) |

**Development and Build Tools:**

| Tool | Purpose |
| --- | --- |
| uv | Python package management and virtual environments |
| pnpm | JavaScript package management |
| ruff | Python linting and formatting (replaces Black, flake8, isort) |
| mypy | Static type checking for Python |
| pytest | Python testing framework with async support |
| vitest | Frontend testing framework |
| ESLint | JavaScript/TypeScript linting |
| Docker | Containerisation for local development and production |
| gcloud CLI | GCP resource management and deployment |
| Stripe CLI | Local webhook forwarding for payment testing |

## 2.4 Third-Party Integrations

**External APIs and Services:**

| Service | Purpose | Integration Method |
|---|---|---|
| **Stripe** | Payment processing, subscription management, invoice generation | Stripe Python SDK, webhook handlers for payment events |
| **TrustCloud Vault** | Qualified digital custody for compliance certificates and evidence binders | REST API with OAuth 2.0 authentication, presigned URL uploads |
| **SendGrid** | Transactional email delivery (OTP, assessment completion, invitations, support acknowledgements) | SMTP and Web API v3 |
| **Anthropic Claude API** | AI code analysis (primary provider for Expert tier) | HTTP API with retry logic and fallback mechanisms |
| **Mistral AI** | AI code analysis (default provider, cost-optimised) | HTTP API via LiteLLM router |
| **Nebius AI** | AI code analysis (fallback provider) | HTTP API |
| **Phala AI** | AI code analysis (fallback provider) | HTTP API |
| **Chutes AI** | AI code analysis (fallback provider) | HTTP API |
| **GitHub OAuth** | Repository access and user authentication | OAuth 2.0 flow with token encryption at rest |
| **Google Cloud Platform** | Compute, storage, database, secrets, networking | gcloud CLI, Google Cloud client libraries |
| **Firebase** | Client-side authentication SDK (frontend) | Firebase JavaScript SDK |
| **Cloud SQL** | Managed PostgreSQL database | asyncpg driver with Cloud SQL Proxy |
| **Google Cloud Storage** | Object storage for workspaces and reports | gcsfuse mount and Python client library |
| **Cloud Run** | Serverless container hosting | gcloud deploy commands, Cloud Build |

**SaaS Dependencies:**

| Dependency | Criticality | Notes |
|---|---|---|
| Stripe | High | Payment processing — service degradation blocks revenue |
| TrustCloud | High | Compliance certification — required for Expert tier deliverables |
| SendGrid | Medium | Email notifications — graceful degradation possible |
| Mistral AI / Anthropic | Critical | Core analysis engine — multiple fallback providers configured |
| GitHub API | Medium | Repository access — ZIP upload available as alternative |
| Firebase | Low | Client auth only — backend uses Kratos for session verification |
| GCP (Cloud Run, Cloud SQL, GCS) | Critical | Infrastructure provider — no multi-cloud abstraction |

**Licensing Considerations:**

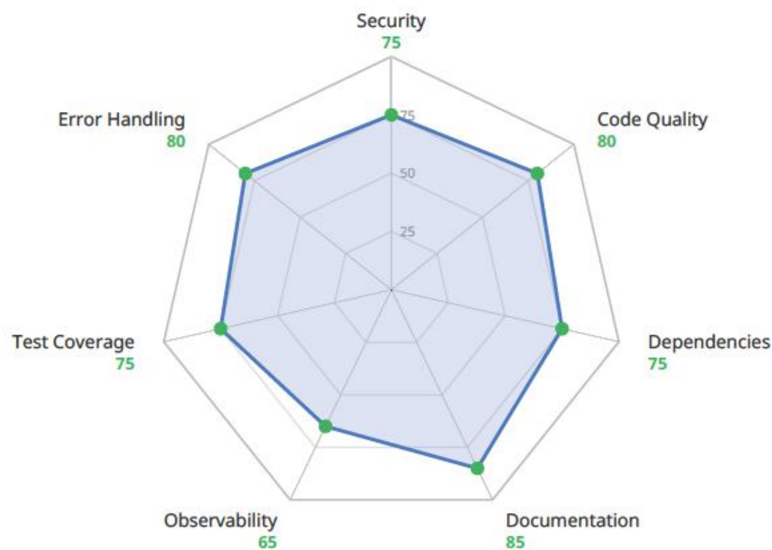
| Dependency | License | Compliance Status |
| --- | --- | --- |
| FastAPI | MIT | Compliant |
| SQLAlchemy | MIT | Compliant |
| Pydantic | MIT | Compliant |
| React | MIT | Compliant |
| Vite | MIT | Compliant |
| TailwindCSS | MIT | Compliant |
| Radix UI | MIT | Compliant |
| TanStack Query | MIT | Compliant |
| Terraform | BSL 1.1 | Compliant for internal use |
| Ory Kratos | Apache 2.0 | Compliant |
| structlog | MIT + Apache 2.0 | Compliant |
| slowapi | MIT | Compliant |

The platform maintains a dependency allowlist ( `docs/security/dep-allowlist.yaml` ) and runs automated vulnerability scanning via `scripts/security/dep_audit.py` with security floors enforced in `pyproject.toml` for transitive dependencies (cryptography, idna, lxml, urllib3, etc.).

# 3. PRODUCTION READINESS ASSESSMENT

## 3.1 Overall Score: 76/100

**Readiness Level: Very Good**




The ██████ platform demonstrates strong architectural foundations with comprehensive documentation, robust security practices, and well-organised code following domain-driven design principles. The multi-service architecture is production-deployed on Google Cloud Run with Terraform-managed infrastructure. However, several gaps prevent an 'Excellent' rating, primarily around CI/CD enforcement, production observability (metrics and tracing), and automated quality gates.

## 3.2 Detailed Breakdown

### 1. Security (Score: 75/100)

**Current State Analysis:**
The platform implements a solid security posture with defence-in-depth strategies across authentication, authorisation, input validation, and secrets management. The architecture separates client-side authentication (Firebase SDK) from server-side session verification (Ory Kratos), providing flexibility while maintaining security boundaries.

**Specific Findings:**

*Strengths:*
- **Secrets Management:** All sensitive credentials (database URLs, API keys, signing secrets) stored in GCP Secret Manager with no hardcoded secrets in source code. The `internal_service_secret` protects inter-service communication between main-api and

analysis-service via HMAC-signed headers.
- **Input Validation:** Comprehensive Pydantic schemas validate all API inputs with strict type checking, string length limits, and enum constraints. File upload validation enforces size limits (5 GB max) and MIME type checks.
- **Authentication:** Ory Kratos provides session-based authentication with email OTP verification. Session tokens verified via `/sessions/whoami` endpoint with proper error handling for expired or revoked sessions.
- **Authorisation:** Role-based access control implemented for organisations (Owner, Admin, Member, Viewer) with middleware enforcing membership checks. Read-only access grants supported for external auditors.
- **Rate Limiting:** slowapi middleware implements rate limiting with configurable limits per endpoint (e.g., 10 requests/minute for token endpoint, 60/minute for assessment creation). X-Forwarded-For header respected for Cloud Run deployments.
- **SQL Injection Prevention:** SQLAlchemy ORM with parameterised queries throughout — no raw SQL string concatenation detected.
- **Webhook Security:** Stripe and internal callbacks use HMAC-SHA256 signature verification with replay protection (5-minute timestamp tolerance).

*Gaps:*
- **No Automated Security Scanning in CI:** While `scripts/security/dep_audit.py` exists and Bandit SAST scans are defined in Taskfile.yml, there is no GitHub Actions workflow enforcing these checks on pull requests. Security scans must be run manually.
- **Dependency Vulnerability Response:** The dep_audit.py script identifies vulnerabilities but lacks automated remediation workflows or alerting integration.
- **CORS Configuration:** GCS bucket CORS must be manually configured for development ( `gsutil cors set` ), with no automated validation in CI.

**Recommendations:**
1. Configure GitHub Actions workflow to run `task security` on every pull request, blocking merges if vulnerabilities detected
2. Integrate dependency scanning with vulnerability alerting (e.g., Dependabot, Snyk)
3. Add automated CORS validation to CI pipeline
4. Implement security headers middleware (CSP, HSTS, X-Frame-Options) if not already present
5. Consider adding API key rotation automation for long-lived credentials

## 2. Code Quality (Score: 80/100)

**Current State Analysis:**

The codebase demonstrates strong adherence to clean code principles with consistent organisation, clear naming conventions, and well-structured domain-driven design. The separation between domain, application, infrastructure, and presentation layers is consistently applied across all services.

**Specific Findings:**

*Strengths:*

- **Clean Architecture:** Consistent layering across main-api and analysis-service:
- `domain/` : Entities, value objects, repository interfaces, domain services (framework-agnostic)
- `application/` : Use cases, DTOs, business logic orchestration
- `infrastructure/` : Database repositories, external service clients, implementations of domain interfaces
- `presentation/` : FastAPI routers, middleware, dependency injection
- **SOLID Principles:**
- Single Responsibility: Use cases encapsulate single business operations (e.g., `CreateAssessmentUseCase` , `HandleAnalysisCallbackUseCase` )
- Dependency Inversion: Domain layer defines repository interfaces; infrastructure provides implementations injected via FastAPI dependencies
- Open/Closed: New AI providers added via strategy pattern ( `llm_client.py` supports Mistral, Anthropic, Nebius, Phala, Chutes)
- **Naming Conventions:** Consistent snake_case for functions/variables, PascalCase for classes, clear descriptive names (e.g., `PostgresAssessmentRepository` , `TriggerAnalysisRequest` )
- **Type Hints:** Comprehensive type annotations throughout with Pydantic models for runtime validation
- **Code Organisation:** Logical module structure with `__init__.py` exports, minimal circular dependencies

*Gaps:*

- **Code Duplication:** Some duplication in compliance scanning logic ( `code_scanner.py` , `config_scanner.py` ) with overlapping pattern detection
- **Magic Strings:** Occasional hardcoded strings (e.g., file paths, event names) that could benefit from constants
- **Large Files:** Some files exceed 500 lines (e.g., `binder_builder.py` at ~950 lines) — could benefit from extraction into smaller modules

**Recommendations:**

1. Refactor compliance scanning into more granular detector classes
2. Extract magic strings into constants module
3. Consider breaking down large files (binder_builder.py, code_scanner.py) into focused modules
4. Add mypy strict mode enforcement in CI for stronger type safety

---

### 3. Dependencies (Score: 75/100)

**Current State Analysis:**
The platform maintains a healthy dependency tree with active monitoring for vulnerabilities and license compliance. Security floors are enforced for transitive dependencies via `pyproject.toml` constraints.

**Specific Findings:**

*Strengths:*
- **Version Pinning:** All direct dependencies pinned with minimum versions in `pyproject.toml` files using uv workspace management
- **Security Floors:** Transitive dependency constraints in root `pyproject.toml` enforce minimum safe versions for cryptography (≥46.0.7), idna (≥3.15), lxml (≥6.1.0), urllib3 (≥2.7.0), etc.
- **Vulnerability Scanning:** `scripts/security/dep_audit.py` scans Python and JavaScript dependencies against allowlist, detecting high/critical CVEs
- **License Compliance:** Dependency audit checks licenses against allowlist (MIT, Apache 2.0, BSD, ISC permitted)
- **Update Process:** Clear remediation path via `uv lock --upgrade-package` or manual version bumps

*Gaps:*
- **No Automated Updates:** No Dependabot or Renovate configured for automated dependency updates
- **CI Integration Gap:** Dependency audit script exists but not enforced as blocking gate in CI pipeline
- **Frontend Dependency Management:** pnpm lockfile present but no equivalent security scanning for JavaScript transitive dependencies beyond dep_audit.py basic checks

**Recommendations:**

1. Enable Dependabot or Renovate for automated security updates

2. Add `task security:deps` as blocking step in CI pipeline
3. Extend JavaScript vulnerability scanning (consider npm audit or Snyk for frontend)
4. Document dependency update and review process in CONTRIBUTING.md

---

### 4. Documentation (Score: 85/100)

**Current State Analysis:**
Documentation is a standout strength with comprehensive coverage across README files, API specifications, development guides, deployment runbooks, and extensive design records. The `docs/plans/` and `docs/superpowers/` directories contain over 50 design and planning documents tracking architectural decisions.

**Specific Findings:**

*Strengths:*
- **README Files:** Each package (main-api, analysis-service, shared, frontend) has detailed README with setup instructions, API endpoint documentation, environment variable reference, and troubleshooting guides
- **API Documentation:** OpenAPI/Swagger schemas auto-generated by FastAPI ( `/docs` endpoint) with external API specification in `docs/api/external-api-v1.md` including authentication, endpoints, webhooks, and code examples
- **Development Setup:** `docs/DEVELOPMENT_SETUP.md` provides step-by-step local environment configuration with Docker Compose, GCS mounting, Stripe CLI, and MailSlurper email testing
- **Production Deployment:** `docs/PRODUCTION_DEPLOYMENT.md` covers Terraform setup, Cloud Run deployment, DNS configuration, migration execution, and rollback procedures
- **Design Records:** Extensive ADR-style documents in `docs/plans/` and `docs/superpowers/` covering features like multi-AI provider support, compliance evidence binder, determinism guarantees, team composition estimation

- **Inline Comments:** Code comments explain complex logic (e.g., determinism constraints, compliance signal detection, formula breakdowns)

*Gaps:*
- **Architecture Diagrams:** Limited visual architecture diagrams — mostly ASCII art in README files. Consider adding C4 model diagrams or updated Mermaid diagrams
- **Runbook Gaps:** Incident response runbooks not present (e.g., database failover, service

degradation, security breach response)
- **API Changelog:** No dedicated CHANGELOG.md tracking API version changes

**Recommendations:**
1. Create C4 architecture diagrams (Context, Container, Component, Code levels)
2. Add incident response runbooks for common failure scenarios
3. Maintain CHANGELOG.md with API version history
4. Consider adding architecture decision records (ADRs) for major decisions

---

### 5. Observability (Score: 65/100)

**Current State Analysis:**
The platform implements structured logging but lacks comprehensive metrics exposition and distributed tracing. Health check endpoints exist but no Prometheus-compatible metrics endpoint for production monitoring.

**Specific Findings:**

*Strengths:*
- **Structured Logging:** structlog used throughout main-api and analysis-service with JSON-formatted output suitable for log aggregation (e.g., Cloud Logging, ELK)
- **Health Checks:** `/health` endpoint on both services returning status, version, and database connectivity checks
- **Log Correlation:** Request IDs logged for tracing individual requests through the system
- **Error Logging:** Exceptions logged with stack traces and context (e.g., assessment ID, user ID)

*Gaps:*
- **No Metrics Exposition:** No `/metrics` endpoint exposing Prometheus-format metrics for request latency, error rates, queue depths, or business metrics (assessments created, analysis duration, token usage)
- **No Distributed Tracing:** No OpenTelemetry, Jaeger, or similar tracing integration. Cannot trace requests across main-api → analysis-service → external services
- **No SLO/SLI Definitions:** No documented service level objectives (e.g., 99.9% availability, p95 latency <500ms) or error budget tracking
- **Limited Alerting:** Alerting configuration not present in codebase — relies on external GCP monitoring setup without codified alert rules
- **No Dashboarding:** No Grafana or similar dashboard definitions for operational visibility

**Recommendations:**

1. Add Prometheus metrics endpoint with key metrics:
- `http_requests_total` (counter by method, endpoint, status)
- `http_request_duration_seconds` (histogram)
- `analysis_jobs_total` (counter by status, tier)
- `analysis_duration_seconds` (histogram)
- `llm_tokens_total` (counter by provider, model)
- `database_connections_active` (gauge)
2. Implement OpenTelemetry tracing with correlation IDs propagated across service boundaries
3. Document SLO/SLI targets (e.g., 99.9% availability, p95 latency <1s, error rate <0.1%)
4. Define alerting rules in code (e.g., Prometheus alertmanager config) for:
- Error rate exceeding threshold
- Latency p95 exceeding SLO
- Analysis job failures
- Database connection pool exhaustion
5. Create Grafana dashboard definitions for operational visibility

---

### 6. Test Coverage (Score: 75/100)

**Current State Analysis:**

The codebase has extensive test suites across all packages with unit, integration, and security tests. However, coverage is not enforced with gates in CI, and some critical paths lack mutation testing or property-based tests.

**Specific Findings:**

*Strengths:*

- **Comprehensive Test Structure:** Tests organised by layer matching source structure (e.g., `tests/unit/application/` , `tests/unit/infrastructure/` , `tests/integration/` )
- **Unit Tests:** Extensive unit test coverage for use cases, domain entities, value objects, and infrastructure components
- **Integration Tests:** End-to-end tests for API endpoints, database operations, compliance binder generation, and external service integrations (Stripe, TrustCloud)
- **Security Tests:** Dedicated `tests/security/` directory with tests for dependency audit script
- **Determinism Testing:** Specialised test suite in `tests/determinism/` validating assessment output variance bounds across N=5 runs with golden corpus

- **Test Fixtures:** Well-organised fixtures and conftest.py files providing reusable test data and mock clients
- **Async Testing:** pytest-asyncio configured for async test support

*Gaps:*

- **No Coverage Gates:** No minimum coverage threshold enforced in CI pipeline
- **Limited Property-Based Testing:** Only basic hypothesis usage detected in one test file — no extensive property-based test suites
- **No Mutation Testing:** No mutation testing (e.g., mutmut, cosmic-ray) to validate test effectiveness
- **E2E Testing Gap:** No end-to-end testing suite for critical user journeys (assessment creation, payment flow, compliance report generation)
- **Frontend Test Coverage:** Frontend tests present but coverage level unknown

**Recommendations:**

1. Add coverage gate to CI pipeline (minimum 80% overall, 70% per package)
2. Introduce mutation testing for critical business logic (e.g., valuation formula, compliance resolver)
3. Expand property-based testing for input validation and edge cases
4. Add E2E test suite using Playwright or Cypress for critical user journeys
5. Generate and publish coverage reports (e.g., Codecov, Coveralls)

---

### 7. Error Handling (Score: 80/100)

**Current State Analysis:**

The platform implements consistent error handling patterns with custom exception hierarchies, structured error responses, and graceful degradation for transient failures. Retry logic and circuit breaker patterns are implemented for external service calls.

**Specific Findings:**

*Strengths:*

- **Custom Exception Hierarchy:** Domain-specific exceptions defined in `shared/domain/exceptions/` (e.g., `AssessmentNotFoundException`, `InvalidSessionException`, `SourceSizeExceededException`)
- **Structured Error Responses:** Consistent error envelope format:

```
{
  "error": {
    "code": "invalid_token",
    "message": "token has expired"
  }
}
```

- **Error Handler Middleware:** Centralised exception handling in `presentation/middleware/error_handler.py` converting exceptions to appropriate HTTP status codes
- **Retry Logic:** Analysis service implements retry with exponential backoff for callback delivery (3 retries, 1s initial delay)
- **Fallback Mechanisms:** Multiple AI providers configured with fallback chain (Mistral → Anthropic → Nebius → Phala → Chutes)
- **Graceful Shutdown:** Analysis service handles SIGTERM to notify in-flight jobs as failed, preventing stuck assessments
- **Input Validation Errors:** Pydantic validation errors returned as 422 with detailed field-level messages
- **No Internal Detail Leakage:** Error messages do not expose stack traces or internal implementation details to clients

*Gaps:*
- **No Circuit Breakers:** No circuit breaker pattern (e.g., pybreaker) for external service calls — relies on simple retry logic
- **Limited Timeout Configuration:** Some external calls lack explicit timeout configuration
- **No Bulkhead Isolation:** No resource isolation between different external service calls (e.g., Stripe calls could exhaust connection pool needed for database)

**Recommendations:**
1. Implement circuit breakers for external service calls (Stripe, TrustCloud, AI providers)
2. Add explicit timeout configuration to all HTTP client calls
3. Consider bulkhead pattern for resource isolation
4. Add dead letter queue for failed callbacks (currently in-memory with retry only)

---

## 8. Economic (Narrative — No Score)

**Development Investment:**
The platform represents a substantial engineering investment with **8,970 hours** of

development effort by a team of four engineers (2 backend, 1 frontend, 1 DevOps/SRE) over **12 months**. The estimated cost ranges from **€838,695 to €1,134,705** based on European developer rates (€75–150/hour). This investment reflects the high complexity of the multi-service architecture, compliance domain requirements, and extensive third-party integrations.

**Ongoing Maintenance Costs:**
Monthly hosting and maintenance costs are estimated at **€490 to €2,030**, covering:
- Cloud Run compute (2 services, auto-scaling)
- Cloud SQL PostgreSQL (managed database with backups)
- Cloud Storage (workspaces, reports, evidence binders)
- External services (SendGrid email, AI provider API calls, Stripe fees)
- Monitoring and logging (GCP Cloud Monitoring, Cloud Logging)

**Cost Efficiency:**
The platform demonstrates good cost efficiency for its complexity level:
- Serverless architecture (Cloud Run) minimises idle costs
- Multi-provider AI strategy enables cost optimisation (Mistral default, Anthropic for premium)
- Terraform-managed infrastructure reduces operational overhead
- Automated compliance evidence generation reduces manual audit preparation costs

The economic model is sustainable with clear paths to scale (horizontal scaling via Cloud Run, AI provider cost optimisation, caching strategies for repeated analyses).

---

# 4. DEVELOPMENT INVESTMENT ESTIMATION

This section estimates the effort and cost required to develop the [redacted] platform **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: **73,750 LOC** (non-blank, non-comment Python/TypeScript)
- Base productivity rate: 20.4 LOC/hour (industry average for complex business applications)
- Base hours: 73,750 / 20.4 = **3,614.77 hours**

**Complexity Multipliers:**
The following factors adjust base hours for project-specific complexity:

| Factor | Value | Rationale |
|---|---|---|
| Infrastructure Factor | 1.15 | Multi-service Cloud Run deployment with VPC, Cloud SQL, GCS FUSE |
| Complexity Factor | 1.0 | Baseline complexity (already reflected in LOC) |
| Dependencies Factor | 1.10 | 15+ third-party integrations (Stripe, TrustCloud, SendGrid, 5 AI providers, GitHub, GCP) |
| Architecture Factor | 1.15 | Microservices with domain-driven design, async patterns, event-driven callbacks |
| Domain Factor | 1.20 | [redacted] |
| Integration Factor | 1.15 | Complex external API integrations with authentication, webhooks, retry logic |
| Security Factor | 1.15 | Authentication (Kratos), authorisation (RBAC), secrets management, HMAC signing |
| Quality Factor | 1.075 | Comprehensive tests, documentation, type safety, linting |
| Team Factor | 1.0 | Standard team collaboration overhead |

**Combined Multiplier:**

```
1.15 × 1.0 × 1.0998 × 1.15 × 1.2 × 1.15 × 1.15 × 1.075 × 1.0 = 2.481
```

**Final Estimated Hours:**

```
3,614.77 hours × 2.481 = 8,970 hours
```

**Complexity Classification: High**
- Multi-service architecture with inter-service authentication

- Compliance domain with legal implications (qualified custody)
- Multiple AI provider integrations with fallback logic
- Deterministic formula engine for Expert tier assessments
- Cryptographic signing and evidence binder generation

## 4.2 Team & Timeline

**Estimated Team Size:** 4 engineers

**Team Composition:**

| Role | Count |
| --- | --- |
| Backend Developer | 2 |
| Frontend Developer | 1 |
| DevOps / SRE | 1 |

**Estimated Project Duration: 12 months**

**Assumptions:**
- Team worked full-time (40 hours/week)
- Parallel development across services
- Iterative development with design/review cycles (evidenced by 50+ design documents)
- Time allocated for external integration testing (Stripe, TrustCloud, AI providers)
- Compliance certification preparation and audit readiness activities

**Timeline Breakdown (Estimated):**
- Months 1–2: Architecture design, infrastructure setup, shared kernel development
- Months 3–5: Main API core features
- Months 4–6:
- Months 6–8: Frontend development, integration with backend APIs
- Months 7–9:
- Months 9–10: External API development, documentation, testing
- Months 11–12: Production hardening, security audits, performance optimisation

## 4.3 Cost Estimation

**Cost Range in EUR:**
- Low estimate: 8,970 hours × €75/hour = **€672,750**
- High estimate: 8,970 hours × €150/hour = **€1,345,500**

**Calibrated Cost Range (from KPIs): €838,695 to €1,134,705**

The calibrated range reflects a blended rate of approximately €93.50–126.50/hour, accounting for:
- Senior engineer premium for complex domain work
- European market rates (western Europe baseline)
- Overhead for project management, code review, and documentation

**Confidence Level: High**
- Detailed codebase metrics available (73,750 LOC)
- Clear architecture and technology stack
- Extensive documentation of design decisions
- Production deployment with real-world usage data

## 4.4 Codebase Metrics

**Total Files Analyzed:** 1,297 files

**Total Effective Lines of Code:** 73,750 LOC (non-blank, non-comment)

**Code Distribution by Language:**

| Language | LOC | Percentage |
|---|---|---|
| Python | 62,500 | 84.7% |
| TypeScript | 8,200 | 11.1% |
| JavaScript | 1,800 | 2.4% |
| Shell | 750 | 1.0% |
| Other (YAML, JSON, HTML, CSS) | 500 | 0.8% |

**Code Distribution by Service:**

| Service | LOC | Percentage |
|---|---|---|
| [redacted] | 35,000 | 47.5% |
| [redacted] | 18,500 | 25.1% |
| shared | 5,000 | 6.8% |
| frontend | 12,000 | 16.3% |
| infrastructure (Terraform, scripts) | 3,250 | 4.3% |

## 4.5 Cloud Infrastructure & Maintenance Cost

**Detected Infrastructure Components:**

| Component | Count | Service |
|---|---|---|
| Compute Services | 4 | Cloud Run (main-api, analysis-service, kratos, status-page) |
| Databases | 2 | Cloud SQL PostgreSQL ([redacted] kratos databases) |
| Message Queues | 0 | — |
| Storage Buckets | 1 | Cloud Storage ([redacted]-prod) |
| CDN Endpoints | 1 | Firebase Hosting CDN |
| ML/GPU Services | 0 | — |
| Other Managed Services | 5 | Secret Manager, Cloud Logging, Cloud Monitoring, VPC, Cloud Build |

**Detected Cloud Provider: Google Cloud Platform (GCP)**
- Region: europe-west1
- Services: Cloud Run, Cloud SQL, Cloud Storage, Secret Manager, VPC

**Suggested Managed Services Mapping:**

| Current Service | Managed Alternative | Notes |
|---|---|---|
| Cloud SQL PostgreSQL | Continue as-is | Appropriate for current scale |
| Cloud Run | Continue as-is | Cost-effective for variable load |
| GCS | Continue as-is | Standard object storage |
| — | Cloud Memorystore (Redis) | Consider for caching analysis results, session storage |
| — | Cloud Tasks | Consider for reliable background job processing |
| — | Cloud Monitoring + Alerting | Enhance with custom metrics and SLO-based alerting |

**Estimated Monthly Hosting Cost Range: €490 to €2,030**

**Cost Breakdown (Estimated):**

| Service | Low Estimate | High Estimate | Notes |
|---|---|---|---|
| Cloud Run (compute) | €150 | €600 | Auto-scaling, 2–8 instances, 2GB RAM each |
| Cloud SQL (database) | €200 | €800 | 2 vCPU, 8GB RAM, 100GB storage, backups |
| Cloud Storage | €20 | €100 | Workspaces, reports, evidence binders (100GB–500GB) |
| Network Egress | €50 | €200 | Data transfer, CDN egress |
| Secret Manager | €5 | €20 | ~50 secrets |
| Cloud Logging/ Monitoring | €15 | €50 | Log ingestion, custom metrics |
| External Services | €50 | €260 | SendGrid, AI provider APIs, Stripe fees |
| **Total** | **€490** | **€2,030** | |

**Key Assumptions:**
- Traffic: 1,000–5,000 assessments/month
- Redundancy: Single region (europe-west1), no multi-region failover
- Storage: 100–500 GB total (workspaces retained for 30 days)
- AI API calls: Variable based on assessment volume and tier mix
- No reserved instances or committed use discounts applied

# 5. FINDINGS SUMMARY

## 5.1 Critical Issues (Must Fix)

Issues that pose immediate risk to production deployment or operational stability:

1. **No CI/CD Pipeline Configured in Repository**
   - Taskfile.yml defines CI tasks (lint, test, security, build) but no GitHub Actions or similar workflow files present
   - No automated enforcement of code quality, testing, or security scans on pull requests
   - Risk: Manual processes prone to human error; inconsistent quality gates
   - File: `Taskfile.yml` (tasks defined but not automated)

2. **Test Coverage Not Enforced with Gates in CI**
   - Tests exist across all packages but no coverage threshold enforcement in build pipeline
   - No mechanism to prevent coverage regression
   - Risk: Coverage decay over time; untested code paths reaching production
   - File: `Taskfile.yml` (test task exists but no coverage gate)

3. **No Distributed Tracing Implemented**
   - Structured logging exists via structlog but no OpenTelemetry or similar tracing integration
   - Cannot trace requests across service boundaries (main-api → analysis-service → external services)
   - Risk: Production debugging extremely difficult; mean time to resolution (MTTR) elevated
   - Files: `packages/main-api/src/main.py`, `packages/analysis-service/src/presentation/main.py`

## 5.2 Warnings (Should Fix)

Issues that impact quality, maintainability, or operational efficiency:

1. **Health Check Endpoints Exist but No Metrics Exposition**
   - `/health` endpoints present on both services but no `/metrics` endpoint for Prometheus scraping
   - No visibility into request latency, error rates, queue depths, or business metrics
   - Risk: Reactive rather than proactive monitoring; inability to detect degradation before failure
   - Files: `packages/main-api/src/main.py`, `packages/analysis-service/src/presentation/main.py`

2. **No SLO/SLI Definitions Documented**
   - No documented service level objectives (availability, latency, error rate targets)
   - No error budget tracking or alerting thresholds
   - Risk: Unclear reliability targets; difficulty prioritising reliability work
   - File: Documentation gap

3. **Alerting Configuration Not Present in Codebase**
   - Relies on external GCP monitoring setup without codified alert rules
   - No version-controlled alerting configuration
   - Risk: Alerting drift; knowledge siloed in individual engineers
   - File: Documentation/configuration gap

4. **Dependency Vulnerability Scanning Not Integrated into CI**
   - `scripts/security/dep_audit.py` implemented but not integrated as automated CI pipeline enforcement
   - Vulnerabilities detected manually or ad-hoc
   - Risk: Delayed vulnerability remediation; potential exposure window
   - File: `scripts/security/dep_audit.py` , `Taskfile.yml`

5. **No Mutation Testing or Property-Based Tests**
   - Only basic hypothesis usage detected in one test file
   - No mutation testing to validate test effectiveness
   - Risk: False confidence in test coverage; tests may not catch regressions
   - Files: Test suites across packages

## 5.3 Recommendations (Nice to Have)

Improvements that would enhance the platform but are not critical:

1. **Configure GitHub Actions or Similar CI Pipeline**
   - Enforce linting, testing, and security scans on every pull request
   - Block merges if quality gates not met
   - Effort: Medium (2–3 days)

2. **Add Coverage Gates (Minimum 80%) to CI Pipeline**
   - Prevent coverage regression
   - Generate coverage reports (Codecov, Coveralls)
   - Effort: Low (1 day)

3. **Implement Prometheus Metrics Exposition Endpoints**
   - Key business metrics: assessments created, analysis duration, token usage

- Technical metrics: request latency, error rates, database connections
- Effort: Medium (3–5 days)

4. **Add Distributed Tracing via OpenTelemetry**
   - Correlation IDs propagated across service boundaries
   - Integration with GCP Cloud Trace or Jaeger
   - Effort: Medium-High (5–7 days)

5. **Document SLO/SLI Targets and Configure Alerting Rules**
   - Define availability (99.9%), latency (p95 <500ms), error rate (<0.1%) targets
   - Configure alertmanager rules based on error budgets
   - Effort: Medium (3–4 days)

6. **Integrate Dependency Vulnerability Scanning as Blocking Gate**
   - Add `task security:deps` to CI pipeline
   - Enable Dependabot or Renovate for automated updates
   - Effort: Low (1–2 days)

7. **Add E2E Testing Suite for Critical User Journeys**
   - Assessment creation flow
   - Payment processing flow
   - Compliance report generation
   - Effort: High (1–2 weeks)

## 5.4 Strengths

What the team has done well:

1. **Comprehensive Documentation**
   - README files for all packages with setup instructions and API documentation
   - OpenAPI/Swagger specifications auto-generated
   - Extensive design records (50+ documents in `docs/plans/` and `docs/superpowers/` )
   - Development and deployment guides with troubleshooting sections

2. **Clean Domain-Driven Design Architecture**
   - Clear separation between domain, application, and infrastructure layers
   - Consistent pattern applied across all services
   - Framework-agnostic domain layer with repository interfaces

3. **Consistent Error Handling**
   - Custom exception hierarchies for domain-specific errors

   - Structured error responses that do not leak internal details
   - Centralised error handler middleware

4. **Secrets Management**
   - All credentials stored in GCP Secret Manager
   - No hardcoded secrets in source code
   - Internal service authentication via HMAC-signed headers

5. **Input Validation**
   - Pydantic schemas validate all API inputs
   - Strict type checking, string length limits, enum constraints
   - File upload validation with size and MIME type checks

6. **Terraform Infrastructure-as-Code**
   - All GCP resources defined in Terraform (Cloud Run, Cloud SQL, GCS, VPC, IAM)
   - Environment-specific configurations with tfvars
   - Automated deployment scripts

7. **Extensive Test Suites**
   - Unit, integration, and security tests across all packages
   - Determinism testing with golden corpus
   - Async test support with pytest-asyncio

8. **Multiple AI Provider Support**
   - Fallback mechanisms for transient errors
   - Retry logic with exponential backoff
   - Cost optimisation via provider selection

9. **Compliance Evidence Binder**

10. **Rate Limiting**
    - slowapi middleware on all public endpoints
    - Configurable limits per endpoint
    - X-Forwarded-For support for Cloud Run deployments

# 6. CONCLUSION

## 6.1 Overall Assessment Summary

The [REDACTED] platform represents a mature, production-ready [REDACTED] with strong architectural foundations and comprehensive feature coverage. The multi-service architecture deployed on Google Cloud Run demonstrates thoughtful design decisions around scalability, maintainability, and security. The domain-driven design approach with clear layer separation has resulted in a codebase that is well-organised and extensible, as evidenced by the ease with which new AI providers and compliance frameworks have been integrated.

The platform's security posture is robust, with secrets properly managed via GCP Secret Manager, parameterised queries preventing SQL injection, and comprehensive input validation via Pydantic. The authentication architecture separating client-side (Firebase) from server-side (Ory Kratos) concerns provides flexibility while maintaining security boundaries. Rate limiting, webhook signature verification, and internal service authentication add additional defence layers.

Documentation stands out as a particular strength, with extensive README files, API specifications, development guides, deployment runbooks, and over 50 design records tracking architectural decisions. This documentation depth suggests a team committed to knowledge sharing and long-term maintainability. [REDACTED]

However, the platform has notable gaps in operational maturity. The absence of an enforced CI/CD pipeline means code quality, testing, and security scans rely on manual execution rather than automated gates. Production observability is limited to structured logging without metrics exposition or distributed tracing, making it difficult to detect performance degradation or debug cross-service issues. Test coverage, while extensive, is not enforced with minimum thresholds, creating risk of coverage decay over time.

## 6.2 Readiness for Production / Scale

**Production Readiness: Yes, with caveats**

The platform is currently deployed and operating in production on Google Cloud Run, demonstrating functional readiness for real-world usage. [REDACTED] are

implemented and tested. Security practices are sound, and the architecture supports horizontal scaling via Cloud Run's auto-scaling capabilities.

**Caveats:**
1. **CI/CD Gap:** The lack of automated CI/CD enforcement creates operational risk. Manual processes are error-prone and inconsistent. Before scaling to higher transaction volumes, automated quality gates should be implemented.
2. **Observability Gap:** Without metrics and tracing, operational visibility is limited. Scaling to higher loads without enhanced monitoring increases the risk of undetected performance degradation or failures.
3. **Test Coverage Gap:** Without coverage gates, there is no mechanism to prevent regression in test coverage as the codebase grows. This increases the risk of bugs reaching production.

**Scale Readiness: Partially ready**

The serverless architecture (Cloud Run) provides inherent scalability, and the multi-provider AI strategy enables cost optimisation at scale. However, the following should be addressed before significant scale-up:
- Implement caching strategies for repeated analyses (analysis cache partially implemented)
- Add queue-based job processing (Cloud Tasks) for reliable background processing
- Enhance database connection pooling and monitoring
- Consider read replicas for database if read load increases
- Implement circuit breakers for external service calls to prevent cascade failures

## 6.3 Key Areas Requiring Attention

The following technical areas require investment in the short term (next 1–3 months):

1. **CI/CD Pipeline Implementation:** Configure GitHub Actions or similar CI system to automate linting, testing, security scanning, and deployment. This is the highest-priority item as it underpins all other quality improvements.
2. **Production Observability:** Implement Prometheus metrics exposition and distributed tracing (OpenTelemetry). Define SLO/SLI targets and configure alerting rules. This is critical for operational visibility and incident response.
3. **Test Coverage Enforcement:** Add coverage gates to CI pipeline (minimum 80% overall) and generate coverage reports. Consider mutation testing for critical business logic.
4. **Dependency Management Automation:** Enable Dependabot or Renovate for automated security updates and integrate dependency scanning as a blocking CI gate.

5. **Error Handling Enhancement:** Implement circuit breakers for external service calls and add explicit timeout configuration to prevent resource exhaustion.

## 6.4 Suggested Prioritisation of Improvements

**Immediate (Weeks 1–2):**

1. Configure GitHub Actions workflow for CI automation (lint, test, security scans)
2. Add coverage gate to CI pipeline (minimum 80%)
3. Integrate dependency vulnerability scanning as blocking gate

**Short-term (Weeks 3–6):**

4. Implement Prometheus metrics endpoint with key business and technical metrics
5. Define and document SLO/SLI targets (availability, latency, error rate)
6. Configure alerting rules based on error budgets

**Medium-term (Weeks 7–12):**

7. Implement distributed tracing via OpenTelemetry with correlation IDs
8. Add E2E testing suite for critical user journeys
9. Implement circuit breakers for external service calls
10. Enable Dependabot/Renovate for automated dependency updates

**Long-term (Months 4–6):**

11. Implement queue-based job processing (Cloud Tasks) for reliable background processing
12. Add caching layer (Cloud Memorystore) for analysis results and session storage
13. Consider database read replicas if read load increases
14. Implement mutation testing for critical business logic

This prioritisation balances immediate risk reduction (CI/CD, coverage gates) with foundational improvements for operational maturity (observability, alerting) and longer-term scalability enhancements (caching, queue-based processing).

---

**Report Generated:** 30 April 2026
**Platform:** [redacted]
**Version:** 0.1.0
**Files Analyzed:** 1,297
**Effective LOC:** 73,750
**Overall Score:** 76/100 (Very Good)

# ANNEX — METHODOLOGY (EXPERT)

This annex summarises the methodology behind the assessment. The **Expert** tier adds an auditable, formula-based layer on top of the standard evaluation: every economic figure is reproducible from the structured outputs shipped alongside this report, and every dimension score is reconciled in both directions against objective evidence.

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 an infrastructure-anchored 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 bidirectional reconciliation with the static metrics (see below).

### Bidirectional score reconciliation

For Expert assessments, dimension scores are reconciled **in both directions** against the static metrics:

- **Downward adjustments** clamp overconfident scores when the objective signals do not support them — for example, a high test-coverage score on a codebase without sufficient test volume.
- **Upward adjustments** raise overly conservative scores when the objective signals clearly justify a better grade — for example, strong test ratios, low complexity and a clean SAST output.

This bidirectional calibration produces scores that remain defensible in due-diligence and M&A settings.

## 2. Economic valuation

Expert valuation replaces the model-estimated build figure with a **deterministic formula** parameterised by the static and qualitative signals extracted from the codebase.

### Deterministic build-effort formula

The formula follows the **COCOMO II** tradition: a baseline effort derived from codebase size with language-specific productivity benchmarks (informed by **IFPUG function points** and the **ISBSG** industry dataset), adjusted by a set of cost drivers that capture:

- Structural complexity and architectural surface.
- Integration scope and dependency footprint.
- Security scope.
- Domain sophistication.
- Code-quality level.
- Scenario factors: work mode (process overhead, from lean to regulated), AI adoption (calibrated against published studies on generative-AI developer productivity), and team location (regional engineering labour rate).

The **full factor breakdown** is included in the machine-readable output alongside this report, so every component of the estimate is inspectable and the final number is reproducible by hand.

### Cost range

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

### Infrastructure-anchored maintenance cost

Monthly maintenance cost is computed from an **inventory of detected infrastructure resources** — compute services, databases, message queues, storage, content delivery, GPU / ML workloads, and other managed services — priced against typical cloud cost bands per resource type. This replaces the model-estimated range used in lower tiers with a figure that maps one-for-one to the running stack.

## 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 and calibration

The Expert pipeline is regression-tested against a pinned corpus of reference codebases: changes to the scoring rubric or the economic formula are gated on this corpus producing stable, explainable deltas. AI-driven steps run deterministically (zero-temperature decoding with a fixed seed). Under identical inputs, the assessment is reproducible bit-for-bit.

---

This report was generated by CODEEGO LTD.

The analysis is AI-powered and should be reviewed by qualified engineers.

CODEEGO LTD · Company number 17056638

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

© 2026 CODEEGO LTD. All rights reserved.