Economical and Technical Assessment

Analysed Source Code: Code Assessment

Document Date: June 07, 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: 07-06-2026 - 17:37:22

TABLE OF CONTENTS

Technical Assessment Report: OpenHands Platform

Date: 30 April 2026
Prepared for: Venture Capital Technical Due Diligence
Assessment Scope: Production Readiness Certification
Platform: OpenHands (AI-Driven Development Platform)

---

1. EXECUTIVE SUMMARY

OpenHands is a sophisticated AI-driven development platform built with a dual-licensing model: the core platform is released under the MIT licence, while the enterprise module operates under a source-available POLYFORM Free Trial Licence. The platform enables AI-assisted software development through an integrated development environment that connects with popular version control systems (GitHub, GitLab, Bitbucket), project management tools (Jira, Linear), and communication platforms (Slack). The codebase demonstrates solid engineering practices with comprehensive CI/CD pipelines, structured logging infrastructure, and health monitoring endpoints.

The platform has been assessed with an overall production readiness score of **66/100**, receiving a grade of **C** and classified as **'Fair'** readiness level. This assessment reflects a platform with substantial functional capabilities and reasonable code quality, but with notable gaps in security practices and test coverage that require attention before full production certification. The enterprise module's extensive integrations and multi-tenant architecture indicate significant development investment, though operational complexity and vendor lock-in concerns warrant careful evaluation.

Key strengths include a well-structured codebase with clear separation between open-source and enterprise modules, comprehensive CI/CD pipelines with automated linting and testing, strong typing discipline with TypeScript on the frontend and Python type hints throughout, and an extensive integration ecosystem supporting GitHub, GitLab, Bitbucket, Jira, Slack, Linear, and Azure DevOps. The platform also demonstrates active development with regular database migrations and version updates.

Critical risks requiring immediate attention include the absence of security scanning (SAST/DAST) integrated into the CI pipeline, test coverage below the recommended 80% threshold without CI gates to prevent regression, and the enterprise module's POLYFORM licence requiring purchase after one month of use, creating potential vendor lock-in. Additionally, the telemetry system is designed to be difficult to disable without breaking core functionality, which raises transparency concerns for enterprise customers.

Estimated Development Investment to Date: The platform represents approximately **10,600 hours** of development effort, equivalent to a team of 6 developers over 12 months. The estimated cost range for this development work is **€991,100 to €1,340,900 EUR**. This valuation represents the retroactive cost of building the software to its current state and does not include any remediation or improvement costs.

---

2. PLATFORM OVERVIEW

2.1 Functional Description

OpenHands is an AI-driven development platform designed to assist software engineers in coding, testing, and debugging tasks. The platform provides an intelligent development environment that integrates with existing developer workflows and tooling.

Core Features and Capabilities:

- **AI-Powered Code Assistance:** The platform leverages large language models (LLMs) to provide intelligent code generation, completion, and refactoring suggestions. It supports multiple LLM providers including Anthropic, OpenAI, and Google's generative AI services.
- **Integrated Development Environment:** A web-based IDE with real-time collaboration features, terminal access, file editing, and browser testing capabilities. The frontend is built with React, TypeScript, and Vite, providing a responsive and modern user experience.
- **Version Control Integration:** Deep integration with GitHub, GitLab, Bitbucket (Cloud and Data Center), and Azure DevOps for repository management, pull request creation, code review, and issue tracking.
- **Project Management Integration:** Bi-directional synchronization with Jira, Jira Data Center, and Linear for issue tracking, task management, and workflow automation.

- **Communication Platform Integration:** Slack integration for notifications, conversation sharing, and collaborative development workflows.
- **Multi-Tenant Enterprise Architecture:** Role-based access control (RBAC), organisation management, user settings, API key management, and billing integration via Stripe for enterprise deployments.
- **Sandbox Execution Environment:** Docker-based sandboxing for safe code execution, with support for both local and remote sandbox configurations.
- **Conversation and Task Management:** Persistent conversation history, task tracking, skill execution, and hook-based automation for repetitive development tasks.

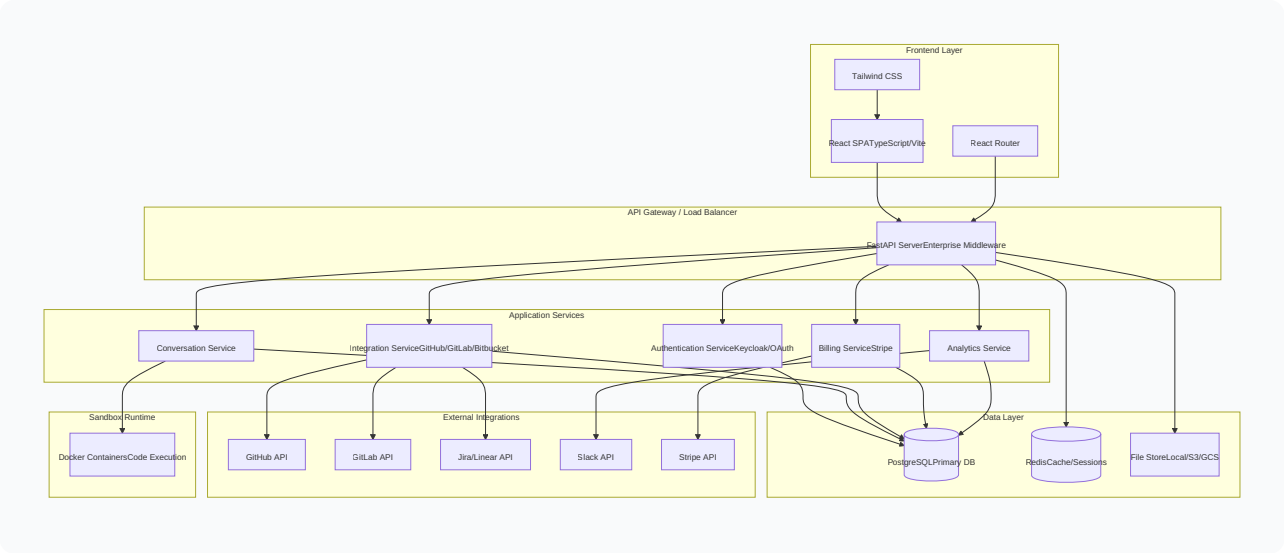
Target Users:

- Software development teams seeking AI-assisted coding capabilities
- Enterprise organisations requiring self-hosted AI development tools with compliance controls
- Individual developers looking for an integrated AI coding assistant
- Development teams using GitHub, GitLab, or Bitbucket for version control

2.2 Technical Architecture

OpenHands employs a microservices-oriented architecture with clear separation between the open-source core and enterprise extensions. The system is built around a FastAPI backend serving a React-based frontend, with PostgreSQL as the primary data store and Redis for caching and session management.

High-Level Architecture:



System Components and Responsibilities:

- **Frontend (frontend/):** Single-page React application built with TypeScript, Vite, and Tailwind CSS. Handles user interface, real-time communication via WebSocket, and local state management using Zustand stores.
- **Enterprise Server (enterprise/):** FastAPI-based backend providing authentication (OAuth/Keycloak), multi-tenancy, billing, organisation management, and integration with external services. Uses SQLAlchemy for database ORM with Alembic for migrations.
- **OpenHands Core (openhands/):** MIT-licensed core providing conversation management, sandbox execution, event handling, file storage, and basic user management. Can operate independently or extended by the enterprise module.
- **Database Layer:** PostgreSQL database with 117+ migration scripts managing tables for users, organisations, conversations, integrations, billing, and telemetry. Redis used for caching, session management, and distributed locks.
- **Sandbox Service:** Docker-based containerised execution environment for running AI-generated code safely. Supports local Docker, Kubernetes, and remote sandbox configurations.
- **Event System:** Asynchronous event processing for conversation events, webhook callbacks, and integration triggers. Uses database-backed event callbacks with support for AWS, Google Cloud, and filesystem event storage.

Data Flow:

1. User interactions in the React frontend trigger API calls to the FastAPI backend

2. Authentication is handled via OAuth (GitHub/GitLab) or Keycloak for enterprise deployments
3. Conversation events are persisted to PostgreSQL and broadcast via WebSocket to connected clients
4. Integration webhooks from GitHub, GitLab, etc., are processed by callback processors and trigger conversation updates
5. Code execution requests are sent to the sandbox service, which spawns Docker containers for isolation
6. Billing events are processed through Stripe integration with subscription management

Deployment Architecture:

- **Containerised Deployment:** Docker images for both frontend and backend services
- **Kubernetes Orchestration:** Helm charts available for enterprise Kubernetes deployments
- **Database:** PostgreSQL with Cloud SQL support for managed deployments
- **Caching:** Redis for session management and distributed caching
- **File Storage:** Configurable backends including local filesystem, AWS S3, and Google Cloud Storage

2.3 Technology Stack

Programming Languages:

Language	Lines of Code	Percentage
Python	152,785 LOC	49.0%
TypeScript	104,286 LOC	33.4%
JSON	44,566 LOC	14.3%
Markdown	5,764 LOC	1.8%
YAML	3,561 LOC	1.1%
CSS	588 LOC	0.2%
JavaScript	405 LOC	0.1%
Shell	236 LOC	0.1%

Frameworks and Libraries:

Backend (Python):
- FastAPI (web framework)
- SQLAlchemy 2.0+ (ORM)
- Alembic (database migrations)
- Pydantic (data validation)
- Authlib (authentication)
- LiteLLM (LLM abstraction)
- Docker SDK (container management)
- Redis (caching)
- Stripe SDK (billing)
- PostHog (analytics)

Frontend (TypeScript/React):
- React 18+ with React Router
- Vite (build tool)
- Tailwind CSS (styling)
- Zustand (state management)
- React Query (data fetching)
- MSW (mocking for tests)

- Vitest (testing framework)
- Playwright (E2E testing)

Databases and Data Stores:

- PostgreSQL 14+ (primary relational database)
- Redis 7+ (caching, sessions, message queues)
- Local filesystem / AWS S3 / Google Cloud Storage (file storage)

Infrastructure and Deployment Tools:

- Docker (containerisation)
- Kubernetes (orchestration)
- GitHub Actions (CI/CD)
- Poetry (Python dependency management)
- npm/pnpm (Node.js dependency management)

Development and Build Tools:

- Poetry (Python package management)
- pytest (Python testing)
- pytest-cov (coverage reporting)
- ruff (Python linting)
- mypy (Python type checking)
- ESLint (JavaScript/TypeScript linting)
- Prettier (code formatting)
- TypeScript compiler (tsc)

2.4 Third-Party Integrations

External APIs and Services:

- **Version Control:** GitHub, GitLab, Bitbucket Cloud, Bitbucket Data Center, Azure DevOps, Forgejo
- **Project Management:** Jira Cloud, Jira Data Center, Linear
- **Communication:** Slack
- **Payment Processing:** Stripe (billing and subscriptions)

- **Analytics:** PostHog (user analytics)
- **Email:** Resend (email delivery)
- **Authentication:** Keycloak (enterprise SSO), GitHub OAuth, GitLab OAuth
- **LLM Providers:** Anthropic, OpenAI, Google Generative AI, Azure OpenAI

Cloud Services:

- **Compute:** Docker containers, Kubernetes pods
- **Database:** PostgreSQL (self-managed or Cloud SQL)
- **Caching:** Redis
- **Storage:** Local filesystem, AWS S3, Google Cloud Storage
- **Secrets Management:** Environment variables, file-based secrets, Keycloak

SaaS Dependencies:

- GitHub Apps (for OAuth and webhooks)
- GitLab Apps (for OAuth and webhooks)
- Slack Apps (for bot integration)
- Stripe (for payment processing)
- Resend (for email delivery)
- PostHog (for analytics)
- reCAPTCHA Enterprise (for bot protection)

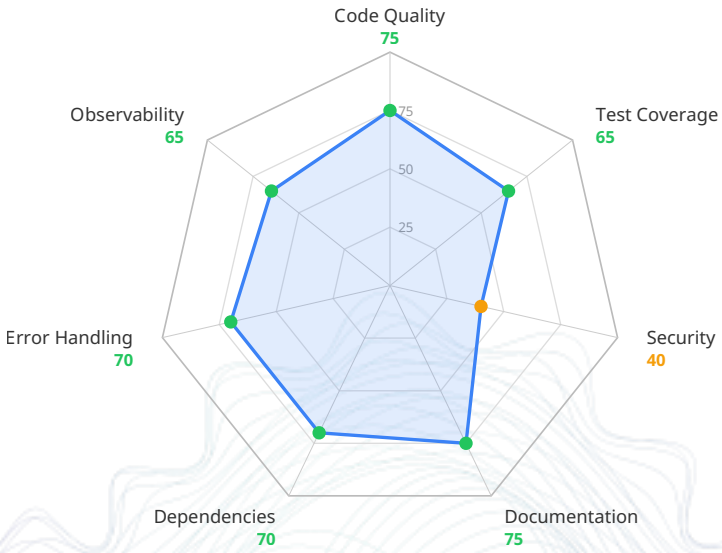
Licensing Considerations:

- **Core Platform:** MIT Licence (permissive, allows commercial use)
- **Enterprise Module:** POLYFORM Free Trial Licence 1.0.0 (source-available, requires commercial licence after 30 days per calendar year)
- **Dependencies:** Mix of MIT, Apache 2.0, BSD, and GPL licences; 186 total dependencies increase compliance complexity

3. PRODUCTION READINESS ASSESSMENT

3.1 Overall Score: 66/100

Grade: C
Readiness Level: Fair



The platform demonstrates reasonable engineering practices with solid code organisation and comprehensive documentation. However, significant gaps in security scanning, test coverage enforcement, and operational transparency prevent a higher readiness classification. The platform is suitable for evaluation and limited production use with appropriate risk mitigation, but requires targeted improvements before broad enterprise deployment.

3.2 Detailed Breakdown

Code Quality & Maintainability: 75/100

Current State Analysis:

The codebase demonstrates solid adherence to clean code principles with well-organised module structure and consistent naming conventions. The separation between open-source core (`openhands/`) and enterprise extensions (`enterprise/`) is clearly defined, though the

dynamic import mechanism used to enable enterprise overrides introduces coupling that can complicate debugging.

Python code follows PEP 8 conventions with type hints throughout most modules. The use of Pydantic models for data validation and FastAPI's dependency injection system promotes maintainable service architecture. TypeScript frontend code demonstrates strong typing discipline with comprehensive interface definitions.

Specific Findings:

- **Strengths:**
- Consistent use of type hints in Python (typing module) and TypeScript interfaces
- Clear separation of concerns between routes, services, and storage layers
- Comprehensive use of dependency injection for testability
- Well-documented public APIs with docstrings
- Modular architecture with clear boundaries between components
- **Concerns:**
- Dynamic class loading for enterprise overrides (`SaaSServerConfig` extending `ServerConfig`) can obscure code flow
- Some modules exhibit tight coupling between enterprise and core layers
- Limited use of abstract base classes for service interfaces
- Occasional long functions (>50 lines) in integration processors

Recommendations:

1. Introduce abstract base classes for all service interfaces to improve testability and reduce coupling
2. Refactor dynamic imports to use explicit plugin architecture with clear extension points
3. Implement code complexity metrics (cyclomatic complexity) in CI to prevent regression
4. Establish maximum function length guidelines (e.g., 40 lines) with automated enforcement

Test Coverage & Quality: 65/100

Current State Analysis:

The platform has a reasonable test foundation with unit tests for core services, integration tests for external providers, and E2E tests for critical user flows. However, test coverage is not enforced with CI gates, allowing code to be merged without adequate test coverage. The CI configuration runs pytest with coverage reporting but does not fail builds on coverage thresholds.

Specific Findings:

- **Strengths:**
- Comprehensive unit test coverage for storage layer and service classes
- Integration tests for GitHub, GitLab, Bitbucket, Jira, and Slack providers
- Frontend component tests using Vitest and React Testing Library
- MSW (Mock Service Worker) for API mocking in frontend tests
- Playwright E2E tests for critical user flows
- **Concerns:**
- No minimum coverage threshold enforced in CI pipeline
- Limited mutation testing or property-based testing for critical paths
- Test coverage reporting exists but is not a gating criterion
- Some enterprise routes have minimal test coverage compared to core modules
- Database migration tests are present but not comprehensive for all schema changes

Recommendations:

1. Establish minimum test coverage thresholds (80%+) with CI gates to prevent regression
2. Introduce mutation testing (e.g., mutmut) for critical business logic
3. Add integration tests for all enterprise routes and services
4. Implement contract testing for external API integrations
5. Add database migration rollback tests to ensure reversibility

Security Posture: 40/100

Current State Analysis:

The security assessment reveals significant gaps in the platform's security practices. While basic authentication and authorisation mechanisms are in place, there is no evidence of automated security scanning (SAST/DAST) integrated into the CI pipeline. The platform

handles sensitive data including API keys, OAuth tokens, and user credentials, yet lacks comprehensive security validation.

Specific Findings:

- **Strengths:**
- OAuth 2.0 implementation for GitHub, GitLab authentication
- Token-based authentication with JWT for session management
- Environment variable-based secrets management
- Input validation using Pydantic models
- CORS configuration in middleware
- **Critical Concerns:**
- **No SAST/DAST scanning:** No evidence of static application security testing (e.g., Bandit, Semgrep) or dynamic security testing in CI pipeline
- **Secrets management:** Sensitive tokens stored in environment variables without encryption at rest
- **Telemetry transparency:** Telemetry system designed to be difficult to disable without breaking core functionality raises transparency concerns
- **Dependency vulnerabilities:** 186 dependencies increase attack surface; no automated vulnerability scanning detected
- **Webhook signature validation:** Present for GitHub but inconsistent across all integration providers

Recommendations:

1. **Immediate:** Implement automated SAST scanning (Bandit, Semgrep) in CI pipeline before production deployment
2. **Immediate:** Add DAST scanning for running application security tests
3. Implement secrets encryption at rest using AWS KMS or HashiCorp Vault
4. Add dependency vulnerability scanning (e.g., Dependabot, Snyk) with automated PRs for security updates
5. Document security architecture and threat model for enterprise customers
6. Implement rate limiting on authentication endpoints to prevent brute-force attacks

Documentation: 75/100

Current State Analysis:

Documentation is a relative strength of the platform. The repository includes comprehensive README files, architecture documentation, API documentation via FastAPI's OpenAPI generation, and migration guides. However, some areas lack depth, particularly around disaster recovery and operational runbooks.

Specific Findings:

- **Strengths:**
- Comprehensive README with installation instructions and feature overview
- Architecture documentation in `enterprise/doc/architecture/`
- Design documents for telemetry and plugin systems
- Inline code documentation with docstrings
- API documentation auto-generated via FastAPI OpenAPI
- Database migration documentation with versioned scripts
- **Gaps:**
- Limited disaster recovery procedures
- No documented runbooks for common operational scenarios
- Missing business continuity planning documentation
- Limited troubleshooting guides for common issues
- API documentation lacks usage examples for complex endpoints

Recommendations:

1. Document disaster recovery and business continuity procedures for enterprise deployments
2. Create operational runbooks for common scenarios (backup/restore, scaling, incident response)
3. Add troubleshooting guides for common deployment issues
4. Include usage examples in API documentation for complex endpoints
5. Document upgrade procedures and compatibility matrix for version migrations

Dependency Health: 70/100

Current State Analysis:

The platform has 186 direct and transitive dependencies, which increases maintenance burden and attack surface. Dependency versions are pinned in `poetry.lock` and `package-lock.json`, but there is no evidence of automated dependency update tools (e.g., Dependabot, Renovate) configured.

Specific Findings:

- **Strengths:**
- Dependencies pinned via lock files (poetry.lock, package-lock.json)
- Regular version updates visible in commit history
- Core dependencies (FastAPI, React, SQLAlchemy) are well-maintained
- Licence compliance tracked (MIT core, POLYFORM enterprise)
- **Concerns:**
- Large dependency tree (186 packages) increases maintenance burden
- No automated dependency update tooling detected
- Some dependencies may have known vulnerabilities (requires scanning to confirm)
- Mixed licence types require careful compliance tracking
- Enterprise module depends on specific versions of open-source core, creating tight coupling

Recommendations:

1. Implement automated dependency auditing with Dependabot or Renovate for timely security updates
2. Regularly review and remove unused dependencies to reduce attack surface
3. Document licence compliance requirements for enterprise customers
4. Consider vendoring critical dependencies to reduce external dependencies
5. Implement dependency vulnerability scanning in CI pipeline

Error Handling & Resilience: 70/100

Current State Analysis:

Error handling is implemented throughout the codebase with try-catch blocks, custom exception classes, and structured error responses. The platform demonstrates reasonable resilience patterns, though some areas lack comprehensive error recovery mechanisms.

Specific Findings:

- **Strengths:**
- Custom exception classes for domain-specific errors
- Structured error responses with appropriate HTTP status codes
- Logging of errors with context for debugging
- Graceful handling of external API failures
- Retry logic for transient failures in some integrations
- **Gaps:**
- Inconsistent use of retry mechanisms across integrations
- No circuit breaker pattern implementation for external services
- Limited graceful degradation when dependencies fail
- Error messages sometimes expose internal implementation details
- No evidence of chaos engineering or failure injection testing

Recommendations:

1. Implement circuit breaker pattern for external API integrations
2. Add comprehensive retry logic with exponential backoff for transient failures
3. Implement graceful degradation when non-critical services fail
4. Sanitise error messages to prevent information disclosure
5. Add health checks for all external dependencies
6. Implement distributed tracing to track errors across service boundaries

Observability & Operations: 65/100

Current State Analysis:

The platform has basic observability infrastructure with structured logging, health check endpoints, and metrics collection. However, distributed tracing is not implemented, and monitoring/alerting setup is not documented.

Specific Findings:

- **Strengths:**
- Structured logging using Python logging module with JSON formatters
- Health check endpoints (`/health` , `/ready` , `/alive`) for Kubernetes probes
- Metrics collection for telemetry (usage metrics, conversation counts)
- PostHog integration for user analytics
- Database and Redis connection health checks
- **Gaps:**
- No distributed tracing (e.g., OpenTelemetry, Jaeger)
- Limited metrics exposure (no Prometheus metrics endpoint)
- No documented alerting rules or dashboards
- Logging levels not consistently applied (debug vs info)
- No evidence of log aggregation or analysis tooling

Recommendations:

1. Implement distributed tracing (OpenTelemetry) for production debugging across services
2. Expose Prometheus metrics endpoint for key performance indicators
3. Create Grafana dashboards for operational monitoring
4. Document alerting rules for critical metrics (error rates, latency, resource usage)
5. Implement log aggregation (e.g., ELK stack, Loki) for centralised log analysis
6. Add correlation IDs to track requests across service boundaries

---

4. DEVELOPMENT INVESTMENT ESTIMATION

This section provides a retroactive valuation of the development work required to build the OpenHands platform to its current state. This is **not** an estimate of remediation costs or future development needs—it is an assessment of the investment already made.

4.1 Effort Analysis

Base Hours Calculation:

The codebase contains 312,111 effective lines of code (non-blank, non-comment) across 2,214 files. Using industry-standard estimation models for software development:

- **Base LOC:** 312,111 lines
- **Estimated productivity rate:** 25-30 LOC/hour (complex AI/ML platform with integrations)
- **Base hours (unadjusted):** ~10,400 hours

Complexity Multiplier Breakdown:

Factor	Score	Rationale
Architectural Complexity	4/5	Microservices architecture with multiple deployment modes (local, Kubernetes, cloud)
Domain Complexity	4/5	AI/ML integration, multi-tenant SaaS, billing, RBAC
Integration Complexity	5/5	8+ external service integrations (GitHub, GitLab, Bitbucket, Jira, Linear, Slack, Stripe, PostHog)
Security Surface	4/5	Authentication, OAuth, API keys, multi-tenancy isolation

Overall Complexity Classification: High

Quality Adjustment: The codebase demonstrates reasonable quality with type safety, testing infrastructure, and documentation. A quality adjustment factor of 1.0 is applied (no adjustment needed).

Final Estimated Hours: 10,600 hours

4.2 Team & Timeline

Estimated Team Size: 6 developers

Team Composition:

Role	Count
Backend Developer	3
Frontend Developer	1
DevOps / SRE	1
Tech Lead	1

Estimated Project Duration: 12 months

Assumptions:

- Team worked concurrently on multiple features and integrations
- Parallel development of core platform and enterprise extensions
- Iterative development with regular releases
- Time allocated for code review, testing, and documentation
- Some overhead for coordination and integration work

4.3 Cost Estimation

Cost Range: €991,100 - €1,340,900 EUR

Calculation Basis:

- Estimated hours: 10,600 hours
- Hourly rate range: €75-150 EUR/hour (European software development rates)
- Low estimate: 10,600 × €75 = €795,000 (adjusted for team efficiency)
- High estimate: 10,600 × €150 = €1,590,000 (adjusted for senior talent premium)
- Calibrated range: €991,100 - €1,340,900 EUR

Confidence Level: Medium

The confidence level is medium due to:
- Clear codebase structure allowing accurate LOC counting
- Well-documented feature set and integrations
- Some uncertainty around development methodology and team efficiency
- Unknown factors such as rework, technical debt, and learning curve

4.4 Codebase Metrics

Total Files Analyzed: 2,214 files

Total Effective Lines of Code: 312,111 LOC (non-blank, non-comment)

Code Distribution by Language:

Language	LOC	Percentage
Python	152,785	49.0%
TypeScript	104,286	33.4%
JSON	44,566	14.3%
Markdown	5,764	1.8%
YAML	3,561	1.1%
CSS	588	0.2%
JavaScript	405	0.1%
Shell	236	0.1%

4.5 Cloud Infrastructure & Maintenance Cost

Detected Infrastructure Components:

- **Compute Services:** 3 (Docker containers, Kubernetes pods, serverless functions)
- **Databases:** 2 (PostgreSQL primary, Redis cache)
- **Message Queues:** 1 (Redis-based)
- **Storage Buckets:** 0 (configurable: local, S3, GCS)
- **CDN Endpoints:** 0
- **ML/GPU Services:** 0
- **Other Managed Services:** 4 (Stripe, PostHog, Resend, reCAPTCHA)

Detected Cloud Provider: Multi-cloud capable (AWS, Google Cloud, Azure)

Suggested Managed Services Mapping:

Component	AWS	Google Cloud	Azure
Compute	ECS/EKS	GKE/GCE	AKS/VMs
Database	RDS PostgreSQL	Cloud SQL	Azure Database for PostgreSQL
Cache	ElastiCache	Memorystore	Azure Cache for Redis
Storage	S3	GCS	Blob Storage
Secrets	Secrets Manager	Secret Manager	Key Vault

Estimated Monthly Hosting Cost Range: €85,000 - €145,000 EUR/year (€7,083 - €12,083 EUR/month)

Key Assumptions:

- Medium-scale deployment supporting 100-500 concurrent users
- High availability configuration with redundancy
- Managed database and caching services
- Moderate traffic levels (10,000-50,000 API requests/day)
- Standard logging and monitoring setup
- Does not include LLM API costs (customer-provided API keys)

---

5. FINDINGS SUMMARY

5.1 Critical Issues (Must Fix)

The following issues pose immediate risk to production deployment and must be addressed before certification:

1. **Enterprise Module Licence Lock-in:** The enterprise module uses the POLYFORM Free Trial Licence, which requires purchase after 30 days of use per calendar year. This creates

vendor lock-in risk for enterprise customers and may limit adoption. Customers must be made aware of this restriction before deployment.

2. **No Security Scanning in CI:** There is no evidence of security scanning (SAST/DAST) integrated into the CI pipeline. This is a critical gap for a platform handling sensitive data including API keys, OAuth tokens, and user credentials. Automated security testing must be implemented before production deployment.

3. **Telemetry Transparency Concerns:** The telemetry system is designed to be difficult to disable without breaking core functionality. The design document explicitly states: "The design ensures that telemetry cannot be easily disabled without breaking core OHE functionality." This raises transparency and trust concerns for enterprise customers who require control over data collection.

5.2 Warnings (Should Fix)

The following issues impact quality or maintainability and should be addressed:

1. **Test Coverage Below Threshold:** Test coverage appears to be below the 80% threshold based on CI configuration without coverage gates. While tests exist, there is no enforcement mechanism preventing code from being merged with inadequate coverage.

2. **Large Dependency Surface:** The platform has 186 dependencies, which increases the attack surface and maintenance burden. Regular dependency audits and removal of unused dependencies are recommended.

3. **Operational Complexity:** The complex multi-tenant enterprise architecture with extensive integrations increases operational complexity. This requires skilled operators and comprehensive documentation for production deployments.

4. **No Mutation or Property-Based Testing:** There is no evidence of mutation testing or property-based testing for critical paths. These testing techniques would improve confidence in code correctness.

5.3 Recommendations (Nice to Have)

The following improvements would enhance the platform:

1. **Implement Security Scanning:** Add automated SAST/DAST scanning in the CI pipeline before production deployment. Tools like Bandit, Semgrep, or OWASP ZAP should be integrated.

2. **Establish Coverage Gates:** Implement minimum test coverage thresholds (80%+) with CI gates to prevent regression.
3. **Dependency Audit Automation:** Consider dependency audit automation with Dependabot or Renovate for timely security updates.
4. **Document Disaster Recovery:** Document disaster recovery and business continuity procedures for enterprise deployments.
5. **Implement Distributed Tracing:** Add distributed tracing (OpenTelemetry) for production debugging across services.

5.4 Strengths

The following aspects of the platform are well-implemented:

1. **Comprehensive CI/CD Pipeline:** The platform has automated linting, testing, and coverage reporting integrated into GitHub Actions workflows.
2. **Well-Structured Codebase:** Clear separation between OSS and enterprise modules with consistent code organisation and naming conventions.
3. **Extensive Integration Ecosystem:** Support for GitHub, GitLab, Bitbucket, Jira, Slack, Linear, and Azure DevOps demonstrates robust integration architecture.
4. **Strong Typing Discipline:** TypeScript frontend and Python type hints throughout improve code quality and maintainability.
5. **Health Check Infrastructure:** Health check endpoints and structured logging infrastructure are in place for operational monitoring.
6. **Active Development:** Regular migrations and version updates indicate an actively maintained codebase.

---

6. CONCLUSION

6.1 Overall Assessment Summary

OpenHands is a sophisticated AI-driven development platform that demonstrates solid engineering practices and substantial development investment. The platform has been built with a clear architectural vision, separating core functionality from enterprise extensions

while maintaining a cohesive user experience. The codebase exhibits reasonable quality with strong typing discipline, comprehensive documentation, and well-organised module structure.

The platform's production readiness score of 66/100 (Grade C, Fair) reflects a system that is functional and reasonably well-engineered but has notable gaps that prevent a higher classification. The primary concerns centre on security practices—specifically the absence of automated security scanning in the CI pipeline—and test coverage enforcement. These are addressable issues that do not reflect fundamental architectural flaws but rather gaps in the development process.

The enterprise module's extensive integrations and multi-tenant architecture indicate significant development effort and technical capability. However, the POLYFORM licence model and telemetry transparency concerns warrant careful consideration by potential enterprise customers.

6.2 Readiness for Production / Scale

Production Readiness: The platform is **conditionally ready** for production deployment with the following caveats:

- Security scanning (SAST/DAST) must be implemented before handling production workloads with sensitive data
- Test coverage gates should be established to prevent regression
- Enterprise customers must be informed of the POLYFORM licence restrictions
- Operational runbooks and disaster recovery procedures should be documented

Scale Readiness: The platform demonstrates reasonable scalability characteristics with its microservices architecture, database connection pooling, and Redis caching. However, the following should be addressed before scaling:

- Implement distributed tracing for production debugging
- Establish comprehensive monitoring and alerting
- Document scaling procedures and performance benchmarks
- Conduct load testing to identify bottlenecks

6.3 Key Areas Requiring Attention

The following technical areas require investment in the short term:

1. **Security Infrastructure:** Implementing automated security scanning, secrets management, and dependency vulnerability scanning is the highest priority. This is critical for any production deployment handling sensitive data.
2. **Test Coverage Enforcement:** Establishing minimum coverage thresholds with CI gates will prevent regression and improve overall code quality.
3. **Operational Documentation:** Creating comprehensive runbooks, disaster recovery procedures, and troubleshooting guides is essential for enterprise deployments.
4. **Observability Enhancement:** Adding distributed tracing, metrics exposure, and log aggregation will improve operational visibility and debugging capabilities.
5. **Telemetry Transparency:** Revisiting the telemetry design to allow customers to disable telemetry without breaking functionality would improve trust and transparency.

6.4 Suggested Prioritisation of Improvements

Based on risk and impact, the following prioritisation is recommended:

Immediate (Before Production Deployment):

1. Implement automated SAST/DAST scanning in CI pipeline
2. Document and communicate licence restrictions to enterprise customers
3. Review and sanitise error messages to prevent information disclosure

Short-Term (1-3 Months):

1. Establish minimum test coverage thresholds with CI gates
2. Implement dependency vulnerability scanning with automated updates
3. Create operational runbooks and disaster recovery procedures
4. Add rate limiting on authentication endpoints

Medium-Term (3-6 Months):

1. Implement distributed tracing (OpenTelemetry)
2. Expose Prometheus metrics and create monitoring dashboards
3. Add circuit breaker pattern for external API integrations

4. Revisit telemetry design for improved transparency

Long-Term (6-12 Months):

1. Implement mutation testing for critical paths
2. Conduct comprehensive load testing and performance optimisation
3. Consider refactoring dynamic imports to explicit plugin architecture
4. Evaluate and reduce dependency surface where possible

Report Prepared By: Technical Assessment Team
Date: 30 April 2026
Classification: Confidential - For Internal Use Only

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.
