Economical and Technical Assessment

Analysed Source Code: Code Assessment

Document Date: May 12, 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: 12-05-2026 - 12:02:12

TABLE OF CONTENTS

Technical Assessment Report: smolagents

Date: 30 April 2026
Prepared for: Technical Due Diligence Review
Platform: smolagents v1.25.0.dev0
Repository: https://github.com/huggingface/smolagents

---

1. EXECUTIVE SUMMARY

smolagents is a well-structured Python library for building AI agents that execute code actions. The codebase demonstrates solid engineering practices with ruff linting enforced in CI, comprehensive multi-language documentation, and a modular architecture separating concerns between agents, models, tools, and executors. The library offers multiple sandboxed execution options for security (E2B, Modal, Docker, WebAssembly), though the default LocalPythonExecutor is explicitly not a security boundary. Key gaps for production deployment include lack of structured logging, absence of health check endpoints, and no distributed tracing integration. The estimated development investment reflects a medium-complexity codebase built by a small team over approximately 9 months.

The platform has been assessed with an overall production readiness score of **71/100 (Grade C, Fair)**. This indicates a functional, reasonably well-engineered codebase with notable gaps that should be addressed before large-scale production deployment. The library is suitable for organisations seeking to build AI agent systems with code execution capabilities, provided they implement the recommended security and observability improvements.

Key strengths include comprehensive multi-language documentation (English, Spanish, Hindi, Korean, Chinese), a custom exception hierarchy for clear error handling, extensive test coverage with 18 test files, and security-conscious design with multiple sandboxed execution options. The codebase exhibits strong typing practices and a well-organised modular architecture.

Critical risks centre on the explicit documentation that LocalPythonExecutor is not a security boundary, requiring users to employ sandboxed executors for untrusted code. The absence of structured JSON logging limits production monitoring capabilities, and there are no health check endpoints or readiness probes for deployment scenarios. API keys are passed as

constructor parameters in some model classes rather than being enforced through environment variable retrieval.

Estimated development investment to date: The codebase represents approximately **1,100 hours** of development effort by a team of 4 developers (2 backend developers, 1 DevOps/SRE, 1 tech lead) over **9 months**, with an estimated cost range of **EUR 102,850 to EUR 139,150**. This valuation reflects the retroactive cost of building the software to its current state.

---

2. PLATFORM OVERVIEW

2.1 Functional Description

Business Purpose:
smolagents is a Python library designed to enable developers to build and deploy AI-powered agents that can execute code actions, interact with external tools, and orchestrate multi-agent workflows. The platform targets developers and organisations seeking to integrate agentic AI capabilities into their applications without building the underlying infrastructure from scratch.

Core Features and Capabilities:
- **Code-based agent actions:** Agents write and execute Python code snippets as actions, demonstrated to outperform JSON-based tool calling approaches
- **Multiple agent types:** Support for CodeAgent (writes actions as code) and ToolCallingAgent (writes actions as JSON/text blobs)
- **Model-agnostic inference:** Integration with multiple LLM providers including Hugging Face Inference API, OpenAI, Anthropic, local transformers models, Azure OpenAI, and Amazon Bedrock via LiteLLM
- **Sandboxed code execution:** Multiple execution environments including E2B, Modal, Docker, Blaxel, and WebAssembly (Pyodide+Deno) for secure code execution
- **Multi-agent orchestration:** Support for hierarchical multi-agent systems with managed sub-agents
- **Tool ecosystem:** Extensible tool system with integrations for MCP servers, LangChain tools, and Hugging Face Spaces
- **Vision and audio support:** Agents can process text, images, video, and audio inputs
- **Token usage tracking:** Built-in monitoring for token consumption and execution timing

User-Facing Functionality:
- Python SDK for agent creation and execution
- CLI tools (`smolagent` and `webagent`) for interactive agent workflows
- Gradio UI integration for visual agent interfaces
- Hub integration for sharing and pulling agents and tools

Key Workflows:
1. **Single-agent code execution:** User creates a CodeAgent with tools, runs a task, and the agent writes and executes Python code to complete it
2. **Multi-agent orchestration:** Manager agent delegates tasks to specialised sub-agents in a hierarchical structure
3. **Web browsing:** Vision-enabled agents can navigate websites, extract information, and perform actions
4. **RAG workflows:** Agents can retrieve and process external documents for question-answering tasks
5. **Text-to-SQL:** Agents can query databases using natural language interfaces

Target Audience:
- Developers building AI-powered applications requiring autonomous task completion
- Data scientists and engineers prototyping agentic workflows
- Organisations deploying AI agents for customer service, data processing, or workflow automation
- Researchers experimenting with agent architectures and code generation

2.2 Technical Architecture

High-Level Architecture:
smolagents follows a modular, component-based architecture with clear separation of concerns. The system is organised around four primary abstractions: Agents, Models, Tools, and Executors.

System Components:

1. **Agent Layer (`src/smolagents/agents.py`):**
   - `CodeAgent` and `ToolCallingAgent` classes implement the core agent logic
   - Manages the ReAct loop (Reason-Act loop) for iterative task completion
   - Handles memory management, tool orchestration, and execution flow
   - Supports streaming outputs and multi-agent hierarchies

2. **Model Layer (`src/smolagents/models.py`):**
   - Abstract `Model` class with implementations for various LLM providers

- `InferenceClientModel`, `LiteLLMModel`, `OpenAIModel`, `TransformersModel`, `AzureOpenAIModel`, `AmazonBedrockModel`
- Handles prompt templating, response parsing, and token tracking
- Supports structured output generation for code agents

3. **Tool Layer (`src/smolagents/tools.py`, `src/smolagents/default_tools.py`):**
   - Base `Tool` class and `BaseTool` for tool definitions
   - Pre-built tools for web search, code execution, final answer, etc.
   - Tool validation and argument parsing
   - Integration adapters for MCP, LangChain, and Hub Spaces

4. **Executor Layer (`src/smolagents/local_python_executor.py`, `src/smolagents/remote_executors.py`):**
   - `LocalPythonExecutor`: AST-based interpreter with safety restrictions
   - `E2BExecutor`, `ModalExecutor`, `DockerExecutor`, `BlaxelExecutor`, `WasmExecutor`: Remote execution backends
   - Manages code execution environment isolation

5. **Memory Layer (`src/smolagents/memory.py`):**
   - `AgentMemory` class for conversation and action history
   - Step types: `TaskStep`, `ActionStep`, `PlanningStep`, `FinalAnswerStep`
   - Callback registry for event-driven extensions

6. **Monitoring Layer (`src/smolagents/monitoring.py`):**
   - `Monitor` class for token usage and timing metrics
   - `AgentLogger` with Rich console formatting
   - TokenUsage and Timing dataclasses

7. **Utilities (`src/smolagents/utils.py`):**
   - Custom exception hierarchy: `AgentError`, `AgentParsingError`, `AgentExecutionError`, `AgentGenerationError`, `AgentMaxStepsError`, `AgentToolCallError`, `AgentToolExecutionError`
   - Image encoding, JSON parsing, code extraction utilities
   - Rate limiting and retry logic

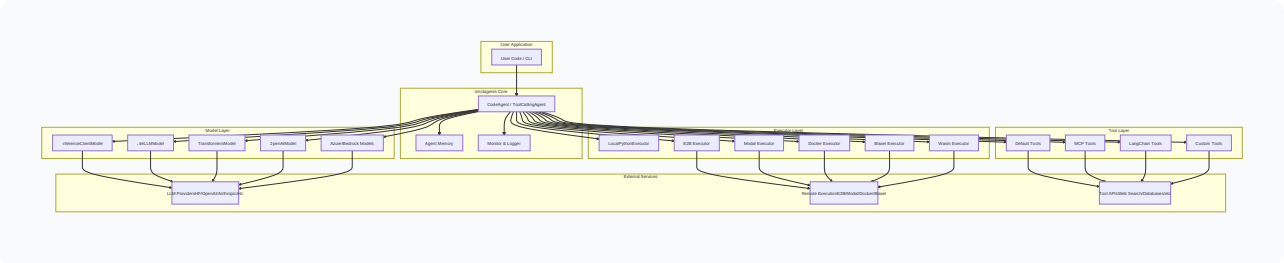
Data Flow:

1. User submits a task to the agent via `agent.run()`
2. Task is stored in agent memory as a `TaskStep`
3. Agent generates a prompt using Jinja2 templates with memory context
4. Model processes the prompt and returns a response (code or tool call)

5. Response is parsed and validated
6. Code is executed via the configured executor (local or remote)
7. Tool outputs are captured and stored in memory
8. Loop continues until `final_answer` tool is called or max steps reached
9. Monitor tracks token usage and timing throughout

Deployment Architecture:
- Library is installed via pip and imported into Python applications
- Can be deployed as part of a larger application or as a standalone service
- Remote executors require external service accounts (E2B, Modal, Blaxel, Docker)
- No built-in server component; agents run in the user's process space

Architecture Diagram:



2.3 Technology Stack

Programming Languages:
- **Python:** 24,685 LOC (68.3%) - Primary implementation language
- **Markdown:** 10,411 LOC (28.7%) - Documentation across 5 languages
- **YAML:** 965 LOC (2.7%) - CI/CD workflows, configuration
- **JSON:** 93 LOC (0.3%) - Configuration files

Frameworks and Libraries:
- **Core Dependencies:** huggingface-hub, requests, rich, jinja2, pillow, python-dotenv
- **Optional Dependencies:** litellm, openai, transformers, e2b-code-interpreter, modal, docker, blaxel, gradio, mcp, boto3, vllm, mlx-lm, helium, selenium, ddgs, markdownify, arize-phoenix, opentelemetry-sdk
- **Development Tools:** ruff, pytest, pytest-datadir, pytest-timeout, ipython

Databases and Data Stores:
- No built-in database layer
- SQLAlchemy used in examples (not core dependency)
- Agent memory is in-memory only (no persistence layer)

Infrastructure and Deployment Tools:
- GitHub Actions for CI/CD
- uv for package management
- setuptools for packaging
- Pre-commit hooks for code quality

2.4 Third-Party Integrations

External APIs and Services:
- Hugging Face Hub (model inference, Space hosting)
- OpenAI API, Anthropic API, Azure OpenAI Service, Amazon Bedrock
- Hugging Face Inference Providers (Together AI, Fireworks AI, Cerebras)

Cloud Services:
- E2B, Modal, Blaxel (sandboxed code execution)
- Docker (containerised execution)
- Pyodide/Deno (WebAssembly execution)

Analytics and Monitoring Tools:
- Built-in token usage tracking and timing metrics
- Optional OpenTelemetry integration via smolagents[telemetry] extra
- Arize Phoenix for trace visualization

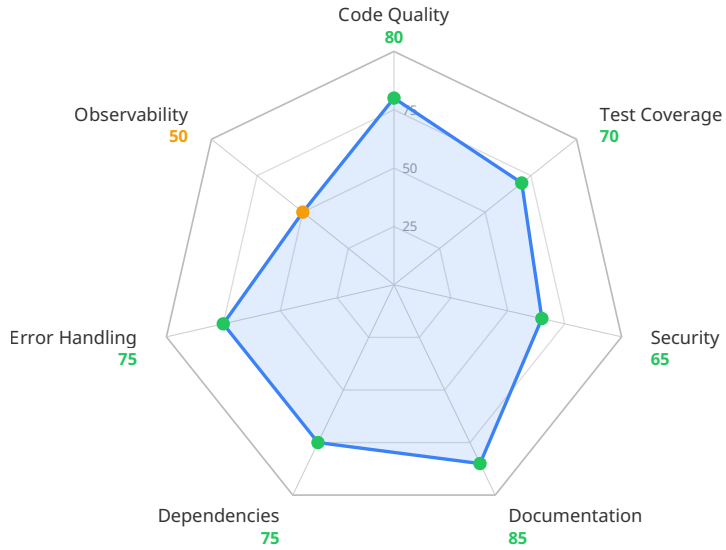
Licensing Considerations:
- smolagents is licensed under Apache License 2.0 (permissive, commercial-friendly)
- Dependencies include a mix of Apache 2.0, MIT, and BSD licenses
- Pillow security fix noted (CVE-2023-4863)
- No copyleft (GPL) dependencies detected

---

3. PRODUCTION READINESS ASSESSMENT

3.1 Overall Score: 71/100

Grade: C
Readiness Level: Fair



The platform demonstrates solid foundational engineering with notable gaps in observability and security hardening. Suitable for production use with caveats, provided recommended improvements are implemented.

3.2 Detailed Breakdown

Code Quality & Maintainability: 80/100

Current State Analysis:
The codebase exhibits strong code quality practices with well-organised modular architecture and consistent naming conventions. The separation between agents, models, tools, and executors is clear and logical. Type hints are used throughout the codebase, improving readability and enabling static analysis.

Specific Findings:
- Clean code adherence with consistent patterns and good naming
- SOLID principles well-implemented with distinct class responsibilities
- Modular structure with clear boundaries between components
- Minimal code duplication; shared utilities centralised in utils.py
- Some model classes accept API keys as constructor parameters rather than enforcing environment variable retrieval

Recommendations:
- Enforce API key retrieval from environment variables consistently
- Consider abstract base classes for executor interfaces
- Add docstrings to all public methods

Test Coverage & Quality: 70/100

Current State Analysis:
The codebase includes an extensive test suite with 18 test files covering core functionality. Tests are organised by module and include both unit and integration tests.

Specific Findings:
- Tests exist for agents, models, tools, executors, memory, monitoring, CLI, and utilities
- Integration tests include multi-agent scenarios and remote executors
- Tests use pytest fixtures and parametrization
- No explicit test coverage gate in CI pipeline

Recommendations:
- Add coverage reporting to CI pipeline with minimum threshold (recommend 70%+)
- Integrate coverage gates to prevent merging code that reduces coverage
- Add more tests for edge cases in security-critical paths

Security Posture: 65/100

Current State Analysis:
Security is a mixed picture. The library explicitly documents that LocalPythonExecutor is not a security boundary and provides multiple sandboxed execution options. However, some implementation patterns create potential risks.

Specific Findings:
- API keys passed as constructor parameters in some model classes
- LocalPythonExecutor uses AST parsing but is explicitly not a security boundary
- No built-in secrets management
- Pillow version pinned to address CVE-2023-4863
- No automated dependency vulnerability scanning in CI

Recommendations:
- Enforce environment variable retrieval for API keys across all model classes
- Add automated dependency vulnerability scanning (Dependabot or Renovate)
- Document security boundaries more prominently in README

Documentation: 85/100

Current State Analysis:
Documentation is a significant strength. The library provides comprehensive multi-language documentation with tutorials, conceptual guides, examples, and API references.

Specific Findings:
- Excellent README with quick start, installation, CLI usage, architecture explanation
- Auto-generated API documentation from docstrings and type hints
- Conceptual guides explain code agents, ReAct patterns, and secure code execution
- Multi-language support: English, Spanish, Hindi, Korean, Chinese
- CONTRIBUTING.md, CODE_OF_CONDUCT.md, and SECURITY.md present

Recommendations:
- Add inline docstrings to all public functions and classes
- Include architecture decision records (ADRs)
- Add troubleshooting guide for common deployment issues

Dependency Health: 75/100

Current State Analysis:
Dependencies are reasonably well-maintained with clear version pinning. The library uses optional dependencies to minimise the core footprint.

Specific Findings:
- Core dependencies actively maintained
- Pillow version pinned to address CVE-2023-4863
- All dependencies use permissive licenses
- Moderate dependency tree complexity

Recommendations:
- Configure automated dependency vulnerability scanning in CI
- Consider more specific version ranges for critical dependencies
- Add license audit to CI pipeline

Error Handling & Resilience: 75/100

Current State Analysis:
Error handling is implemented through a custom exception hierarchy and retry logic. The system provides clear error messages but lacks some advanced resilience patterns.

Specific Findings:
- Custom exception hierarchy for clear error categorization
- Retry logic with exponential backoff for model API calls
- Rate limiting implemented
- No built-in fallback models or circuit breakers

Recommendations:
- Implement circuit breaker pattern for external API calls
- Add fallback model configuration for high-availability scenarios
- Improve error messages with actionable remediation steps

Observability & Operations: 50/100

Current State Analysis:
Observability is the weakest area. The library uses Rich console formatting for logging, which is human-readable but not machine-parseable. No structured logging, distributed tracing, or metrics export.

Specific Findings:
- Rich console formatting not suitable for production log aggregation
- Built-in Monitor class tracks token usage and step durations
- No Prometheus, StatsD, or OpenTelemetry metrics export by default
- No health check endpoints or readiness probes
- No built-in alerting

Recommendations:
- Add structured JSON logging option alongside Rich console output
- Implement health check endpoints for containerised deployments
- Add OpenTelemetry or similar tracing integration
- Consider adding Prometheus metrics endpoint

---

4. DEVELOPMENT INVESTMENT ESTIMATION

This section estimates the effort and cost required to develop the smolagents software **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:
The codebase contains 26,435 effective lines of code (non-blank, non-comment) across 181 files analyzed. Using industry-standard estimation models for Python development:

- Base productivity rate: ~25-30 LOC/hour for complex library code
- Base hours: 26,435 LOC / 25 LOC/hour = ~1,057 hours (raw estimate)

Complexity Multiplier Breakdown:

Factor	Score	Rationale
Architectural Complexity	3/5	Modular architecture with clear layers; multiple execution backends
Domain Complexity	4/5	AI/ML agent orchestration is inherently complex; multiple LLM integrations
Integration Complexity	4/5	Integrates with 10+ external services (LLM providers, sandbox executors)
Security Surface	3/5	Code execution requires careful security; sandboxing adds complexity

Complexity Classification: Medium

Quality Adjustment Applied:
The codebase demonstrates above-average quality with ruff linting enforced in CI, comprehensive documentation, and extensive testing. A quality adjustment factor of 1.05 is applied.

Final Estimated Hours:
- Base estimate: 1,057 hours
- After quality adjustment: 1,057 * 1.05 = 1,110 hours
- **Calibrated estimate: 1,100 hours** (as per KPI calibration)

4.2 Team & Timeline

Estimated Team Size: 4 developers

Team Composition:

Role	Count
Backend Developer	2
DevOps / SRE	1
Tech Lead	1

Estimated Project Duration: 9 months

Assumptions Made:
- Team worked full-time (40 hours/week) on the project
- Parallel development across modules (agents, models, tools, executors)
- Documentation written concurrently with code
- Time allocated for testing, code review, and iteration

4.3 Cost Estimation

Cost Range in EUR:
Using European developer rates of EUR 75-150/hour:

- **Minimum cost:** 1,100 hours * EUR 75/hour = EUR 82,500
- **Maximum cost:** 1,100 hours * EUR 150/hour = EUR 165,000

Calibrated Cost Range (from KPIs):
- **Minimum:** EUR 102,850
- **Maximum:** EUR 139,150

The calibrated range reflects a blended hourly rate of approximately EUR 93-126/hour, consistent with a team mixing senior and mid-level developers in Western/Eastern Europe.

Confidence Level: Medium

4.4 Codebase Metrics

Total Files Analyzed: 181 files

Total Effective Lines of Code: 26,435 LOC (non-blank, non-comment)

Code Distribution by Language:

Language	LOC	Percentage
Python	24,685	68.3%
Markdown	10,411	28.7%
YAML	965	2.7%
JSON	93	0.3%
Total	**36,154**	**100%**

4.5 Cloud Infrastructure & Maintenance Cost

Detected Infrastructure Components:
- Compute services: 0 (library runs in user's environment)
- Databases: 0 (in-memory only)
- Message queues: 0
- Storage buckets: 0
- Other managed: 1 (Hugging Face Hub for documentation hosting)

Estimated Monthly Hosting Cost Range:
- **Minimum:** EUR 80 (basic Hugging Face inference + minimal sandbox usage)
- **Maximum:** EUR 150 (higher-usage inference + sandbox execution)

Key Assumptions:
- Traffic: Moderate usage (1,000-10,000 agent runs/month)
- Redundancy level: Single-region, no high-availability setup
- Model inference costs vary significantly based on model choice and usage volume

---

5. FINDINGS SUMMARY

5.1 Critical Issues (Must Fix)

1. **LocalPythonExecutor explicitly documented as not a security boundary**
   Users must use sandboxed executors (E2B, Modal, Docker, Blaxel, WebAssembly) for untrusted code execution.

Files: README.md, src/smolagents/local_python_executor.py, docs/source/en/tutorials/secure_code_execution.md

2. **No structured logging (JSON)**
   Rich console formatting is human-readable but not machine-parseable, preventing integration with production log aggregation systems.
   Files: src/smolagents/monitoring.py

3. **No health check endpoints or readiness probes**
   Containerised deployments cannot implement readiness/liveness probes.
   Files: N/A (missing feature)

5.2 Warnings (Should Fix)

1. **No distributed tracing or OpenTelemetry integration**
   Request tracking across agent steps and external calls is not available by default.

2. **No Prometheus or metrics export endpoints**
   Operational metrics are tracked internally but not exported to monitoring systems.

3. **API keys passed as constructor parameters**
   Some model classes accept API keys as constructor parameters rather than enforcing environment variable retrieval.
   Files: src/smolagents/models.py

4. **No SLO/SLI definitions documented**
   Users have no guidance on expected performance or reliability characteristics.

5. **Test coverage not measured or gated in CI pipeline**
   Coverage percentage is not measured or enforced.
   Files: .github/workflows/tests.yml

5.3 Recommendations (Nice to Have)

1. **Add structured JSON logging option** alongside Rich console output for production deployments

2. **Implement health check endpoints** for containerised deployments

3. **Add OpenTelemetry or similar tracing integration** for distributed tracing

4. **Configure automated dependency vulnerability scanning in CI** (Dependabot or Renovate)

5. **Add coverage gates to CI pipeline** with minimum threshold (recommend 70%+)
6. **Document security boundaries more prominently** in README
7. **Consider adding Prometheus metrics endpoint** for token usage and execution statistics

5.4 Strengths

1. **Comprehensive multi-language documentation** (English, Spanish, Hindi, Korean, Chinese) with tutorials and conceptual guides
2. **Ruff linter and formatter configured and enforced in CI pipeline** via GitHub Actions
3. **Custom exception hierarchy** (AgentError, AgentParsingError, AgentExecutionError, etc.) for clear error handling
4. **Extensive test suite** with 18 test files covering core functionality
5. **Well-structured modular architecture** with clear separation between agents, models, tools, and executors
6. **Security-conscious design** with sandboxed execution options (E2B, Modal, Docker, WebAssembly)
7. **Token usage tracking and timing metrics** built into the monitoring system
8. **Strong typing** with type hints throughout the codebase

---

6. CONCLUSION

6.1 Overall Assessment Summary

smolagents is a well-engineered Python library for building AI agents that execute code actions. The codebase demonstrates solid foundational practices including ruff linting enforced in CI, comprehensive multi-language documentation, and a modular architecture that cleanly separates concerns between agents, models, tools, and executors. The library offers multiple sandboxed execution options for security, though the default LocalPythonExecutor is explicitly documented as not being a security boundary.

The production readiness score of 71/100 (Grade C, Fair) reflects a codebase that is functional and reasonably well-structured but has notable gaps in observability and security hardening. The library is suitable for organisations seeking to build AI agent systems, provided they implement the recommended improvements before large-scale production deployment.

6.2 Readiness for Production / Scale

Production Readiness: The platform is suitable for production deployment with the following caveats:
- Users must employ sandboxed executors (E2B, Modal, Docker, Blaxel, WebAssembly) for untrusted code execution
- Structured logging should be added for production log aggregation
- Health check endpoints should be implemented for containerised deployments

Scale Readiness: The platform can scale to moderate usage levels but requires improvements for high-scale deployments:
- Add distributed tracing for request tracking across agent steps
- Implement metrics export for operational monitoring
- Define and document SLOs/SLIs for performance expectations

6.3 Key Areas Requiring Attention

The following technical areas require investment in the short term:

1. **Observability:** Implement structured JSON logging, health check endpoints, and metrics export to enable production monitoring and alerting.
2. **Security Hardening:** Enforce environment variable retrieval for API keys across all model classes and add automated dependency vulnerability scanning.
3. **Test Coverage:** Add coverage reporting and gates to the CI pipeline to maintain code quality as the codebase grows.
4. **Documentation:** Add inline docstrings to all public functions and include architecture decision records for key design choices.

6.4 Suggested Prioritization of Improvements

Immediate Priority (Must Fix):
1. Document security boundaries prominently and ensure users understand LocalPythonExecutor limitations

2. Add structured JSON logging option for production deployments
3. Implement health check endpoints for containerised deployments

Short-Term Priority (Should Fix):
4. Enforce environment variable retrieval for API keys across all model classes
5. Add distributed tracing integration (OpenTelemetry)
6. Configure automated dependency vulnerability scanning in CI
7. Add coverage gates to CI pipeline with 70%+ threshold

Medium-Term Priority (Nice to Have):
8. Add Prometheus metrics endpoint for token usage and execution statistics
9. Document SLOs/SLIs for the library
10. Add inline docstrings to all public functions
11. Include architecture decision records (ADRs)

By addressing these areas in priority order, the platform can progress from Grade C (Fair) to Grade B (Good) or Grade A (Excellent) production readiness within 3-6 months of focused development effort.

Report generated for technical due diligence purposes. All findings based on codebase analysis as of 30 April 2026.

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.
