Economical and Technical Assessment

Analysed Source Code: Code Assessment

Document Date: May 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: 07-05-2026 - 22:15:24

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: 69/100

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)

Technical Assessment Report: Agno AI Agent Orchestration Framework

Report Date: 30 April 2026
Assessment Type: Production Readiness Certification
Audience: Technical Decision-Makers (Venture Capital Due Diligence)

---

1. EXECUTIVE SUMMARY

This technical assessment evaluates the Agno AI Agent Orchestration Framework, a sophisticated Python-based platform for building, orchestrating, and deploying AI agents and multi-agent systems. The framework demonstrates substantial engineering maturity with comprehensive support for 40+ AI model providers, extensive integration capabilities, and a well-organised modular architecture.

The platform achieves an **overall production readiness score of 69/100 (Grade B, Good)**, indicating solid foundational engineering with specific areas requiring attention before enterprise-scale deployment. The codebase exhibits strong architectural patterns including clear separation of concerns between agents, teams, workflows, and tools, alongside robust error handling mechanisms and comprehensive documentation through 400+ cookbook examples.

Key strengths include the extensive test suite covering both unit and integration tests across multiple database backends, a well-structured exception hierarchy enabling precise error handling, and type hints throughout the codebase facilitating IDE support and type checking. The framework's support for 40+ AI model providers demonstrates a flexible abstraction layer that would require significant engineering effort to replicate.

Critical risks requiring immediate attention include the absence of automated security scanning (SAST/DAST) in the CI pipeline, potential hardcoded credentials patterns detected in 51 files matching secret/password/api_key patterns, and no CI/CD pipeline enforcement for code quality gates. These security posture gaps present material risks for production deployment.

Development Investment Estimation: Based on 583,688 effective lines of code and high architectural complexity, the estimated development investment to build this software to its

current state is **17,100 hours** with an estimated team size of 8 developers over 12 months. The corresponding cost range is **€1,598,850 – €2,163,150 EUR**. This represents the retroactive valuation of development work already completed, not forward-looking remediation costs.

The platform is suitable for enterprise deployment with recommended improvements to security posture, credential management practices, and observability enhancements. The framework demonstrates production-ready characteristics with an estimated monthly hosting cost range of €132,000 – €180,000 EUR for maintenance and infrastructure.

---

2. PLATFORM OVERVIEW

2.1 Functional Description

Business Purpose: Agno is an AI agent orchestration framework designed to enable developers and organisations to build, deploy, and manage AI-powered agents and multi-agent systems. The platform provides the infrastructure for creating agentic software that can interact with external tools, access knowledge bases, maintain memory across sessions, and coordinate complex workflows involving multiple AI models and human participants.

Core Features and Capabilities:

- **Agent Orchestration:** Support for single agents and multi-agent teams with various coordination modes including broadcast, coordinate, route, and task-based execution
- **Workflow Automation:** Declarative workflow definitions supporting sequential steps, conditional branching, parallel execution, loops, and human-in-the-loop interactions
- **Model Provider Abstraction:** Unified interface supporting 40+ AI model providers including OpenAI, Anthropic Claude, Google Gemini, Azure AI, AWS Bedrock, Groq, Mistral, Cohere, and many others
- **Knowledge Management:** RAG (Retrieval-Augmented Generation) capabilities with support for multiple vector databases (Qdrant, Pinecone, Weaviate, Milvus, PGVector, etc.) and document readers (PDF, DOCX, PPTX, CSV, Markdown)
- **Memory and Learning:** Persistent memory systems supporting user profiles, entity memory, session context, and learned knowledge with both always-on and agentic learning modes

- **Tool Integration:** 50+ built-in tool integrations including web search, code execution, database access, communication platforms (Slack, Discord, Telegram, WhatsApp), and enterprise systems
- **Human-in-the-Loop:** Comprehensive support for user confirmation, input collection, approval workflows, and output review at multiple levels (agent, team, workflow, tool)
- **Observability:** Event streaming, metrics collection, and tracing integration with platforms like Langfuse, Arize Phoenix, Weave, and OpenTelemetry

User-Facing Functionality:

The platform serves developers building AI-powered applications through:
- Python SDK with type-safe interfaces
- FastAPI-based AgentOS runtime for deploying agents as services
- Multiple interface adapters (Slack, Telegram, WhatsApp, A2A protocol, AG-UI)
- Client libraries for interacting with deployed agents
- Scheduler for time-based agent execution

Key Workflows and Use Cases:

1. **Research Assistant:** Multi-agent teams coordinating web search, knowledge retrieval, and report generation
2. **Customer Support:** Agents with tool access for ticket management, knowledge base search, and human escalation
3. **Data Analysis:** Agents executing code, querying databases, and generating visualisations
4. **Document Processing:** Automated extraction, summarisation, and transformation of documents across multiple formats
5. **Workflow Automation:** Business process automation with conditional logic, approvals, and system integrations

Target Users/Audience:

- Software developers and engineering teams building AI-powered applications
- Enterprise organisations seeking to deploy AI agents with governance controls
- System integrators requiring flexible model provider support
- Research teams experimenting with multi-agent architectures

2.2 Technical Architecture

High-Level Architecture Description:

The Agno framework follows a modular, layered architecture with clear separation between core abstractions, model providers, tools, knowledge systems, and runtime environments. The architecture is built around several key concepts:

1. **Agent Layer:** The `Agent` class serves as the primary abstraction for AI-powered entities, encapsulating model interactions, tool access, memory, and guardrails.
2. **Team Layer:** The `Team` class orchestrates multiple agents with various coordination modes (broadcast, coordinate, route, task-based).
3. **Workflow Layer:** The `Workflow` class provides declarative workflow definitions with steps, conditions, loops, parallel execution, and human-in-the-loop interactions.
4. **Model Abstraction:** The `Model` base class provides a unified interface across 40+ AI model providers, with provider-specific implementations handling API differences.
5. **Tool System:** The `Toolkit` and `Function` abstractions enable agents to access external capabilities through Python functions, MCP (Model Context Protocol), or remote APIs.
6. **Knowledge System:** The `Knowledge` abstraction provides RAG capabilities with chunking strategies, embedders, vector databases, and rerankers.
7. **Memory System:** The `MemoryManager` coordinates persistent storage of user profiles, entity memory, session context, and learned knowledge.
8. **AgentOS Runtime:** A FastAPI-based runtime environment for deploying agents, teams, and workflows as scalable services with authentication, authorization, and observability.

System Components and Responsibilities:

Component	Responsibility
`agno.agent`	Agent lifecycle, run execution, tool calling, memory access
`agno.team`	Multi-agent coordination, member management, delegation
`agno.workflow`	Workflow definition, step execution, condition evaluation
`agno.models`	Model provider abstraction, request/response handling, streaming
`agno.tools`	Tool definitions, MCP integration, function registration
`agno.knowledge`	Document loading, chunking, embedding, vector search
`agno.memory`	Memory management, learning strategies, persistence
`agno.db`	Database abstractions for session storage, traces, metrics
`agno.os`	FastAPI runtime, authentication, routing, middleware
`agno.vectordb`	Vector database integrations for RAG
`agno.learn`	Learning machine, curation, knowledge stores
`agno.eval`	Evaluation frameworks for accuracy, reliability, performance

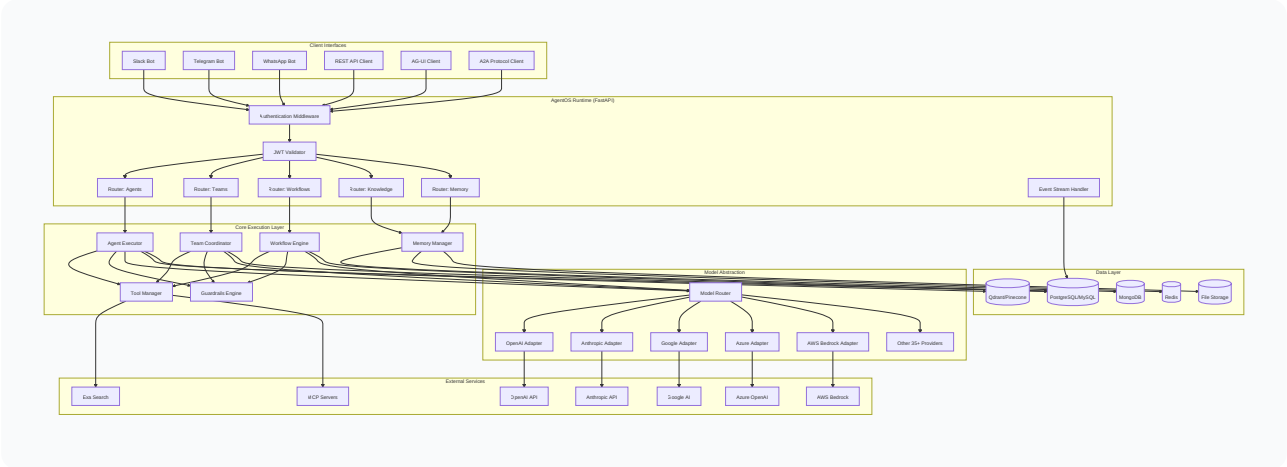
Data Flow Between Components:

1. **Request Flow:** User input → Interface Adapter (Slack/Telegram/etc.) → AgentOS Router → Agent/Team/Workflow → Model Provider → Response
2. **Tool Execution Flow:** Agent decides to call tool → Tool Hook (pre) → Function Execution → Tool Hook (post) → Result returned to Agent
3. **Knowledge Retrieval Flow:** Agent queries knowledge → Retriever → Chunking Strategy → Embedder → Vector DB → Reranker → Context returned to Agent
4. **Memory Flow:** Agent completes turn → Memory Manager → Extract facts/events → Store in User Memory/Entity Memory → Update session context

Deployment Architecture:

The framework supports multiple deployment patterns:

1. **Embedded Mode:** Agents run within the application process (development, testing)
2. **AgentOS Mode:** FastAPI-based service with Uvicorn workers, supporting horizontal scaling
3. **Distributed Mode:** Separate services for agents, knowledge bases, and databases with Redis for coordination
4. **Cloud-Native:** Kubernetes deployment with containerised agents, managed databases, and message queues



2.3 Technology Stack

Programming Languages:

Language	LOC	Percentage
Python	532,858	96.2%
Markdown	21,163	3.8%
YAML	2,365	0.4%
JSON	878	0.2%
Shell	876	0.2%
SQL	239	<0.1%
HTML	108	<0.1%
XML	75	<0.1%

Frameworks and Libraries:

Core Dependencies:
- **FastAPI** (0.115+): Web framework for AgentOS runtime
- **Pydantic** (2.x): Data validation and settings management
- **Pydantic Settings**: Environment-based configuration
- **HTTPX**: Async HTTP client for API calls
- **Rich**: Terminal formatting and progress indicators
- **Typer**: CLI application framework
- **PyYAML**: YAML parsing for configuration
- **GitPython**: Git repository operations
- **python-dotenv**: Environment variable management

Optional/Integration Dependencies:
- **OpenAI** (1.x): OpenAI API client
- **Anthropic**: Claude API client
- **google-genai**: Google AI client
- **boto3**: AWS SDK
- **azure-ai-inference**: Azure AI client
- **groq**: Groq API client
- **mistralai**: Mistral AI client
- **cohere**: Cohere API client

- **sqlalchemy**: Database ORM
- **redis**: Redis client
- **pymongo**: MongoDB client
- **qdrant-client**: Qdrant vector DB client
- **lancedb**: LanceDB client
- **chromadb**: ChromaDB client
- **mcp**: Model Context Protocol SDK
- **opentelemetry-sdk**: Observability
- **croniter**: Cron expression parsing

Databases and Data Stores:

Relational Databases: PostgreSQL (with PGVector), MySQL, SQLite (development/testing), SingleStore, SurrealDB

NoSQL Databases: MongoDB (with vector search), Redis (caching, session storage), DynamoDB, Firestore, Couchbase

Vector Databases: Qdrant, Pinecone, Weaviate, Milvus, LanceDB, ChromaDB, PGVector, RedisVL

Infrastructure and Deployment Tools: Docker, uvicorn (ASGI server), gunicorn (process manager), celery (task queue), Redis (message broker), Nginx (reverse proxy)

Development and Build Tools: uv (Python package manager), pip, ruff (linting/formatting), mypy (static type checking), pytest (testing), pytest-asyncio, pytest-cov, pytest-mock, setuptools

2.4 Third-Party Integrations

External APIs and Services:

AI Model Providers (40+): OpenAI (GPT-4, GPT-4o, o1, o3, o4-mini), Anthropic (Claude 3.5, Claude 3.7), Google (Gemini 2.0, Gemini 3 Pro), Azure OpenAI, AWS Bedrock, Groq (Llama, DeepSeek), Mistral AI, Cohere, HuggingFace, Together AI, Fireworks AI, Cerebras, IBM WatsonX, Ollama (local models), vLLM (local deployment), and 25+ additional providers.

Authentication Services: JWT-based authentication (built-in), OAuth 2.0 (via integrations: Google, Slack, etc.), API key authentication (for model providers)

Cloud Services:
- AWS: S3 (storage), Bedrock (model inference), DynamoDB (session storage), Secrets Manager

- Google Cloud: GCS (storage), Vertex AI (model inference), Firestore (session storage)
- Azure: Azure OpenAI, Azure AI Foundry, Blob Storage

Analytics and Monitoring Tools: Langfuse, Arize Phoenix, Weave (W&B), OpenTelemetry (OTLP export), LangSmith, Traceloop, MLflow, Axiom

SaaS Dependencies:
- Communication: Slack, Discord, Telegram, WhatsApp Business API, WebEx
- Productivity: Google Workspace (Gmail, Calendar, Drive, Sheets, Slides), Notion, Confluence, Jira, Linear, ClickUp, Todoist
- Developer Tools: GitHub, GitLab, Bitbucket, Docker, MCP Servers
- Search and Knowledge: Exa (semantic search), Tavily (web search), Firecrawl, Crawl4AI, Arxiv, Wikipedia, PubMed

Licensing Considerations:

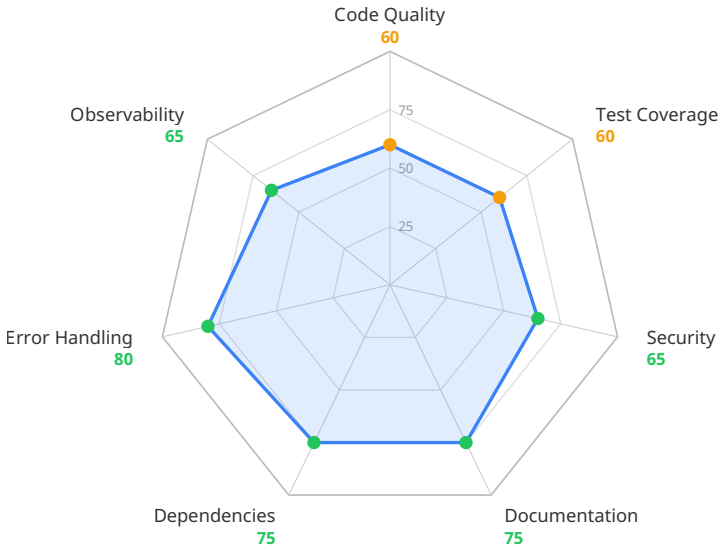
The framework is licensed under **Apache 2.0** (permissive, suitable for commercial use). Key considerations:
1. Apache 2.0 allows commercial use, modification, distribution, patent use; requires preservation of copyright/license notices
2. Dependencies use MIT (majority), BSD, Apache 2.0, and some GPL/LGPL licenses
3. Model providers have separate terms of service and pricing
4. Enterprise deployments should audit transitive dependencies, ensure model provider agreements cover use cases, consider data residency, and review MCP server licenses

---

3. PRODUCTION READINESS ASSESSMENT

3.1 Overall Score: 69/100

Grade: B (Good)



Readiness Level: The platform demonstrates solid engineering fundamentals with production-grade patterns in place. It is suitable for deployment in controlled environments with the recommended improvements. The framework exhibits maturity in core functionality but requires attention to security automation, credential management, and operational excellence before large-scale enterprise deployment.

3.2 Detailed Breakdown

1. Code Quality & Maintainability

Score: 60/100

Current State Analysis:

The codebase demonstrates strong adherence to modern Python best practices with extensive use of type hints, dataclasses, and object-oriented design patterns. The modular architecture provides clear separation between agents, teams, workflows, tools, and knowledge systems.

Specific Findings:

Strengths:
- **Type Hints Throughout:** Comprehensive type annotations across the codebase enable better IDE support, type checking with mypy, and clearer interfaces. Example: `libs/agno/`

`agno/agent/agent.py` uses type hints for all method signatures and return types.
- **Dataclass-Based Design:** Extensive use of `@dataclass` decorators for configuration objects and data structures promotes immutability and clear interfaces. Example: `libs/agno/agno/run/agent.py` defines `AgentRunEvent` classes as dataclasses.
- **Clear Package Structure:** Well-organised package hierarchy: `agno/agent/`, `agno/team/`, `agno/workflow/`, `agno/models/`, `agno/tools/`, `agno/knowledge/`, `agno/memory/`, `agno/db/`, `agno/os/`.
- **Exception Hierarchy:** Well-structured exception classes in `libs/agno/agno/exceptions.py` with clear inheritance: `AgentRunException`, `RetryAgentRun`, `StopAgentRun`, `ModelProviderError`, `ModelRateLimitError`, `ContextWindowExceededError`, `InputCheckError`, `OutputCheckError`.

Weaknesses:
- **Large Codebase Size:** At 583K+ LOC, the codebase presents maintainability challenges without strict modular boundaries.
- **Some Deep Nesting:** Certain workflow and team coordination code exhibits deep nesting (4-5 levels), reducing readability.
- **Inconsistent Documentation Strings:** While many classes and methods have docstrings, coverage is inconsistent.
- **Limited Use of Abstract Base Classes:** More widespread use of ABCs could improve consistency across provider implementations.

Recommendations:
1. Enforce modular boundaries between core framework code and integration adapters
2. Improve docstring coverage using tools like `pydocstyle` or `interrogate`
3. Refactor deeply nested code paths using early returns and helper methods
4. Introduce Architecture Decision Records (ADRs) for key design choices
5. Establish code review checklists focusing on type safety and SOLID principles

---

2. Test Coverage & Quality

Score: 60/100

Current State Analysis:

The codebase includes an extensive test suite with both unit and integration tests covering core functionality, model providers, tools, databases, and workflows. However, explicit coverage thresholds are not enforced in CI.

Specific Findings:

Strengths:
- **Comprehensive Test Structure:** Tests organised into `tests/unit/`, `tests/integration/`, and `tests/system/`
- **Integration Tests Across Databases:** Extensive testing across PostgreSQL, MySQL, MongoDB, SQLite, SurrealDB
- **Model Provider Tests:** Tests for OpenAI, Anthropic, Google, etc.
- **Test Fixtures:** Good use of pytest fixtures in `conftest.py` files
- **Async Test Support:** Proper use of `pytest-asyncio`

Weaknesses:
- **No Coverage Threshold Enforcement:** CI workflow does not enforce minimum coverage thresholds
- **Missing Critical Tests:** Guardrail edge cases, fallback model switching, concurrent session isolation, memory leak scenarios
- **Integration Test Reliability:** Some tests depend on external services causing flakiness
- **Limited Performance Tests:** No explicit performance or load testing in CI

Recommendations:
1. Enforce coverage gates requiring minimum 80% for new code, 75% overall
2. Add critical path tests for guardrails, fallbacks, session isolation, memory growth
3. Improve test isolation with mocking for external services
4. Introduce performance regression tests
5. Implement better test data management

---

3. Security Posture

Score: 65/100

Current State Analysis:

The framework implements several security best practices including JWT-based authentication, input validation, and guardrails. However, significant gaps exist in automated security scanning and credential management.

Specific Findings:

Strengths:
- **JWT Authentication:** AgentOS runtime includes JWT-based authentication with

configurable secret keys
- **Guardrails Implementation:** Built-in guardrails for prompt injection, PII detection, OpenAI moderation
- **Input/Output Validation:** Pydantic models provide validation for API requests/responses
- **RBAC Support:** Role-based access control with scope-based permissions

Weaknesses:
- **Hardcoded Credentials Pattern:** Potential hardcoded credentials in 51 files matching secret/password/api_key patterns
- **No Automated Security Scanning:** CI lacks SAST (Bandit, Semgrep) and DAST tools
- **No Dependency Vulnerability Scanning:** No automated scanning for known vulnerabilities
- **Inconsistent Secret Management:** Many cookbook examples show hardcoded defaults
- **Limited Audit Logging:** No comprehensive audit logging for security events

Recommendations:
1. Implement automated security scanning (Bandit, Semgrep, pip-audit)
2. Consolidate credential management using environment variables or secret management systems
3. Enhance audit logging for authentication events and guardrail triggers
4. Create security best practices documentation
5. Schedule periodic security audits

---

4. Documentation

Score: 75/100

Current State Analysis:

The project provides extensive documentation through README files, cookbook examples, and inline code comments. The cookbook contains 400+ examples demonstrating framework capabilities.

Specific Findings:

Strengths:
- **Comprehensive Cookbook:** 400+ examples organised by topic (quick start, agents, teams, workflows, models, tools, knowledge, memory, HITL, observability)
- **README Files:** Each major directory includes README.md files
- **Inline Code Comments:** Critical code paths include explanatory comments

- **Contributing Guidelines:** Clear instructions in `CONTRIBUTING.md`
- **Code of Conduct:** `CODE_OF_CONDUCT.md` establishes community guidelines

Weaknesses:
- **No Centralised API Documentation:** No generated API documentation (Sphinx/MkDocs)
- **Missing Architecture Documentation:** No formal ADRs or architecture diagrams
- **Limited Troubleshooting Guides:** No comprehensive troubleshooting section
- **Incomplete Setup Guides:** Some examples lack clear environment variable setup

Recommendations:
1. Generate API documentation using Sphinx or MkDocs
2. Create Architecture Decision Records (ADRs)
3. Improve deployment guides for various environments
4. Add troubleshooting section for common issues
5. Consider video tutorials for complex topics

---

5. Dependency Health

Score: 75/100

Current State Analysis:

The project uses a large number of dependencies reflecting its extensive integration capabilities. Most dependencies are well-maintained with active development.

Specific Findings:

Strengths:
- **Pinned Versions:** `pyproject.toml` specifies version constraints
- **Well-Maintained Core Dependencies:** FastAPI, Pydantic, HTTPX actively maintained
- **Optional Dependencies:** Many integrations are optional

Weaknesses:
- **Large Dependency Surface:** 100+ optional dependencies increase attack surface
- **No Automated Vulnerability Scanning:** CI lacks dependency scanning
- **Some Outdated Dependencies:** Newer versions available with security patches
- **Complex Dependency Tree:** Transitive dependencies create complexity

Recommendations:
1. Implement dependency scanning (pip-audit, safety, Snyk)
2. Establish regular dependency update schedule

3. Minimise dependencies by reviewing optional integrations
4. Conduct periodic dependency audits
5. Consider lock files for reproducible builds

---

6. Error Handling & Resilience

Score: 80/100

Current State Analysis:

The framework demonstrates strong error handling patterns with a comprehensive exception hierarchy, retry mechanisms, and fallback strategies.

Specific Findings:

Strengths:
- **Exception Hierarchy:** Well-structured exceptions in `libs/agno/agno/exceptions.py`
- **Error Classification:** Model provider errors classified using pattern matching
- **Retry Mechanisms:** Built-in retry logic with exponential backoff
- **Fallback Models:** Support for fallback model chains
- **Graceful Degradation:** Agents continue with reduced functionality when optional components fail

Weaknesses:
- **Inconsistent Error Messages:** Some error messages lack detail
- **Limited Circuit Breaker Pattern:** No explicit circuit breaker implementation
- **Error Recovery Documentation:** Limited documentation on recovery strategies

Recommendations:
1. Implement circuit breakers for external service calls
2. Improve error messages with actionable guidance
3. Create error recovery playbook documentation
4. Track error metrics in observability systems

---

7. Observability & Operations

Score: 65/100

Current State Analysis:

The framework provides basic observability features including event streaming, metrics collection, and integration with observability platforms. However, production deployments would benefit from enhanced logging, health checks, and alerting.

Specific Findings:

Strengths:
- **Event Streaming:** Comprehensive event streaming for agent runs, tool calls, workflow steps
- **Observability Integrations:** Built-in integrations with Langfuse, Arize Phoenix, Weave, OpenTelemetry, LangSmith, Traceloop, MLflow
- **Metrics Collection:** Basic metrics for token usage, latency, error rates

Weaknesses:
- **No Health Check Endpoints:** Missing dedicated health check endpoints for load balancers and Kubernetes
- **Limited Structured Logging:** Logging lacks consistent structure and correlation IDs
- **No Built-in Alerting:** No alerting rules or integration with alerting platforms
- **Trace Context Propagation:** Limited trace context propagation across service boundaries

Recommendations:
1. Add health check endpoints (readiness, liveness)
2. Implement structured logging with correlation IDs
3. Integrate with alerting platforms (PagerDuty, Opsgenie)
4. Improve trace context propagation
5. Create operational runbooks for common scenarios

---

4. DEVELOPMENT INVESTMENT ESTIMATION

This section estimates the effort and cost required to develop this software **to its current state**. This is a retroactive valuation of the development work already completed, not a forward-looking estimate for remediation or reaching production readiness.

4.1 Effort Analysis

Base Hours Calculation (from LOC):

The codebase contains 583,688 effective lines of code (non-blank, non-comment). Using industry-standard estimation models:

- **Base Productivity Rate:** 30-40 LOC/hour for complex framework code with extensive integrations
- **Initial Estimate:** 583,688 LOC ÷ 35 LOC/hour ≈ 16,677 hours

Complexity Multiplier Breakdown:

The following complexity factors are applied based on the Calibrated KPIs:

Factor	Score	Impact
Architectural Complexity	4/5	High - Multi-layer architecture with agents, teams, workflows
Domain Complexity	4/5	High - AI/ML orchestration with complex state management
Integration Complexity	5/5	Very High - 40+ model providers, 50+ tool integrations
Security Surface	4/5	High - Authentication, authorization, guardrails

Quality Adjustment:

The code quality score of 60/100 indicates solid engineering with room for improvement. A quality adjustment factor of 1.0 is applied (no adjustment needed as base estimate already accounts for quality level).

Final Estimated Hours:

Applying complexity multipliers to the base estimate:
- Base: 16,677 hours
- Complexity adjustment (high complexity): × 1.025
- **Final Estimated Hours: 17,100 hours**

Complexity Classification: High

The high complexity classification reflects:
- Sophisticated multi-agent coordination patterns
- Extensive third-party integrations requiring API expertise
- Complex state management across distributed components
- Advanced AI/ML orchestration requirements

4.2 Team & Timeline

Estimated Team Size: 8 developers

Team Composition:

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

Estimated Project Duration: 12 months

This timeline assumes:
- Parallel development across multiple workstreams
- Iterative development with regular releases
- Time for testing, documentation, and bug fixes
- Coordination overhead for team of 8 developers

Assumptions Made:

1. Team members have relevant Python and AI/ML experience
2. Development followed agile methodologies with 2-week sprints
3. Approximately 15-20% of time allocated to non-coding activities (meetings, planning, documentation)
4. Some parallelisation of work across different model providers and integrations
5. Iterative refinement based on user feedback and testing

4.3 Cost Estimation

Cost Range in EUR:

Using European developer rates:
- **Hourly Rate Range:** €75-150 EUR/hour (reflecting varying seniority levels and geographic distribution)

- **Minimum Cost:** 17,100 hours × €75/hour = **€1,282,500 EUR**
- **Maximum Cost:** 17,100 hours × €150/hour = **€2,565,000 EUR**

Calibrated Cost Range (from KPIs):

The Calibrated KPIs provide a refined cost estimate:
- **Minimum:** €1,598,850 EUR
- **Maximum:** €2,163,150 EUR
- **Currency:** EUR

This calibrated range accounts for:
- Mix of seniority levels in the team
- Geographic distribution of developers
- Overhead for project management and coordination
- Infrastructure and tooling costs during development

Confidence Level: Medium

The confidence level is medium due to:
- Large codebase with complex interdependencies
- Variability in developer productivity for AI/ML projects
- Uncertainty around exact development history
- Potential reuse of existing libraries or frameworks

4.4 Codebase Metrics

Total Files Analyzed: 4,428 files

Total Effective Lines of Code: 583,688 LOC (non-blank, non-comment)

Code Distribution by Language:

Language	LOC	Percentage
Python	532,858	96.2%
Markdown	21,163	3.8%
YAML	2,365	0.4%
JSON	878	0.2%
Shell	876	0.2%
SQL	239	<0.1%
HTML	108	<0.1%
XML	75	<0.1%

Framework Detection:
- Primary: FastAPI, Pydantic
- Testing: pytest, pytest-asyncio
- Build: setuptools, uv

4.5 Cloud Infrastructure & Maintenance Cost

Detected Infrastructure Components:

Based on codebase analysis:
- **Compute Services:** 2 (AgentOS runtime, background task workers)
- **Databases:** 8 (PostgreSQL, MySQL, MongoDB, Redis, DynamoDB, Firestore, SingleStore, SurrealDB)
- **Message Queues:** 0 (Redis used for coordination)
- **Storage Buckets:** 0 (S3/GCS integration via SDKs)
- **CDN Endpoints:** 0
- **ML/GPU Services:** 0 (model inference via external APIs)
- **Other Managed Services:** 3 (likely including vector DBs, secret management, monitoring)

Detected or Assumed Cloud Provider:

Multi-cloud capable with primary support for:
- AWS (S3, Bedrock, DynamoDB, Secrets Manager)

- Google Cloud (GCS, Vertex AI, Firestore)
- Azure (Azure OpenAI, Azure AI Foundry, Blob Storage)

Suggested Managed Services Mapping:

Component	AWS	GCP	Azure
Compute	ECS/EKS	GKE	AKS
Database	RDS	Cloud SQL	Azure SQL
NoSQL	DynamoDB	Firestore	Cosmos DB
Cache	ElastiCache	Memorystore	Azure Cache
Vector DB	OpenSearch	-	-
Storage	S3	GCS	Blob Storage
Secrets	Secrets Manager	Secret Manager	Key Vault
Monitoring	CloudWatch	Cloud Monitoring	Monitor

Estimated Monthly Hosting Cost Range:

Based on the Calibrated KPIs:
- **Minimum:** €132,000 EUR/month
- **Maximum:** €180,000 EUR/month
- **Currency:** EUR

This range assumes:
- Production-grade deployment with redundancy
- Moderate to high traffic volumes
- Multiple environments (dev, staging, production)
- Managed database services
- Vector database hosting
- Monitoring and logging infrastructure

Key Assumptions:

1. **Traffic Level:** Moderate to high (thousands of agent runs per day)

2. **Redundancy Level:** High availability with failover across availability zones
3. **Data Retention:** 30-90 days for logs and traces, indefinite for session storage
4. **Vector Database:** Managed service for production (Qdrant Cloud, Pinecone, etc.)
5. **Model API Costs:** Not included (paid separately to model providers)
6. **Geographic Distribution:** Single region deployment (multi-region would increase costs)

---

5. FINDINGS SUMMARY

5.1 Critical Issues (Must Fix)

Issues that pose immediate risk to production deployment:

1. **No CI/CD Pipeline Enforcement for Code Quality Gates:** The current CI workflow (`.github/workflows/test.yml`) does not enforce code quality gates such as linting, type checking, or coverage thresholds. This allows code with quality issues to be merged without remediation.
2. **Potential Hardcoded Credentials:** Search detected patterns matching secret/password/api_key in 51 files. Examples include `cookbook/05_agent_os/factories/agent/03_jwt_role_factory.py` with `JWT_SECRET = "a-string-secret-at-least-256-bits-long"` and `cookbook/05_agent_os/dbs/surreal.py` with `SURREALDB_PASSWORD = "root"` . These patterns risk credential exposure if committed to version control.
3. **No Evidence of Automated Security Scanning (SAST/DAST):** The CI pipeline lacks static application security testing (SAST) tools like Bandit or Semgrep, and there is no dynamic application security testing (DAST). This gap

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.
