AI Solution Architecture

Enterprise AI Developer Productivity Platform v4.0

1. Architectural Overview

The platform follows a **two-phase architectural evolution** designed to minimize early operational burden while establishing a scalable foundation for event-driven intelligence, graph reasoning, and optional model fine-tuning in later stages. This staged approach addresses the PRD's explicit concern about over-engineering during MVP development and ensures alignment with organizational AI maturity.

The overall architecture consists of:

- A **lightweight MVP stack** centered on retrieval, frontier LLMs, and developer-facing interfaces.
- A **full-scale architecture** adding a shared enterprise knowledge graph, durable workflow engine, event bus, and agentic automation.

The system is designed to act as a **unified semantic layer** across code, documentation, tickets, and architectural knowledge—ultimately supporting both human-in-the-loop workflows and safe, automated reasoning processes.

2. Phase 1 — MVP Architecture (Low Complexity, High Value)

The MVP architecture is intentionally simple, minimizing distributed components while still delivering high developer impact.

2.1 High-Level Components

1. **Weaviate Vector Database**
    - Stores embedding vectors for code, docs, Jira tickets, and Slack messages.
    - Enables hybrid search (BM25 + dense vectors) required by MVP-R1–R4.

    - Supports modular sharding for later expansion but in MVP operates as a single logical cluster.

2. **Redis Cache**
    - Provides response caching, token-budget caching, and short-term session state.
    - Significantly reduces LLM inference costs and latency for repeated context queries.

3. **Backend API (FastAPI or Node)**
    - Primary orchestration layer for RAG pipelines, UI backends, and LLM gateway routing.
    - Implements strict data minimization prior to LLM calls (MVP-R7).
    - Provides REST endpoints consumed by the IDE extension and Slack bot.

4. **LLM Gateway**
    - A dedicated routing layer that abstracts Anthropic/OpenAI providers, enforces timeouts, and handles failovers.
    - Supports tiered routing (cheap → classification; high-tier → synthesis) per MVP-R17.

5. **IDE Extensions (VS Code, JetBrains)**
    - Inline code explanations, context panel, and retrieval source view (MVP-R8–R12).
    - Communicates with backend over HTTPS with user-scoped auth tokens.

6. **Slack Bot**
    - Executes "/context" queries and displays thread-based responses.
    - Integrates Slack OAuth and uses user's permissions for RBAC inheritance (MVP-R25).

7. **Static Environment Diagnostic Engine**

    - Runs local static checks (env vars, ports, dependencies) then sends results to LLM for summarization.
    - Cannot mutate environments per MVP-R15.

8. **Observability Stack**
    - Prometheus for metrics, Grafana for dashboards, Loki for logs.
    - Captures latency (p95 target < 4s), token usage, RAG precision, and cost metrics (MVP-R22–R24).

2.2 Data Flow (MVP)

1. **Indexing Pipeline**
    - GitHub → cloning → AST chunker → embedding model → Weaviate
    - Confluence/Markdown → text chunker → embedding model → Weaviate
    - Jira & Slack → OAuth-based ingestion → cleaning/redaction → embedding → Weaviate
2. **Query Workflow**
    - User invokes query from IDE/Slack
    - Backend retrieves candidates (BM25 + vectors)
    - RRF merges rankings
    - Reranker refines top-k
    - LLM gateway synthesizes response
    - UI displays answer + source citations
3. **Environment Diagnostic Workflow**

    - Local diagnostic script → backend → LLM summarization → suggestions shown in IDE.

2.3 MVP Deployment Plan

- Kubernetes or ECS cluster running backend + Weaviate + Redis.
- Managed LLMs; no local inference.
- 1–2 FTE maintenance due to minimal distributed systems footprint.
- Designed for 10K daily queries with p95 < 4 seconds (NFR).

3. Phase 2 — Full-Scale Architecture (Enterprise-Grade, Extensible)

Once usage, data quality, and demand are established, the platform expands into an **event-driven, graph-enriched** architecture that supports reasoning, lineage tracking, cross-service impact analysis, and agentic workflows.

3.1 New Major Components

1. **Neo4j Enterprise Knowledge Graph**
    - Stores service dependencies, API relationships, ownership, incident lineage, and architectural decisions (FS-R1–R5).
    - Enriches retrieval pipelines: RAG retrieves semantic context; KG provides structural/causal context.
    - Supports relationship projection for cross-service impact analysis.
2. **Kafka Event Bus**
    - Streams build/deploy events, PR merges, cluster health signals, and runtime traces (FS-R6–R8).
    - Enables real-time KG updates and agent triggers.

3. **Temporal Workflow Engine**
    - Coordinates multi-step workflows with retries, human approval stages, and audit logs (FS-R9–R12).
    - Used for environment remediation, meeting → ticket pipelines, impact assessors, and documentation agents.
4. **LangGraph Agent Framework**
    - Encapsulates deterministic multi-node reasoning flows with tool access (FS-R13–FS-R17).
    - Agents have restricted capabilities: sandbox testing, read-only access, or controlled write capabilities gated by approvals.
5. **Model Lifecycle Service (MLS)**
    - Handles dataset curation, fine-tuning pipelines, evaluation, and automated rollback (FS-R18–R22).
    - Stores model versions and evaluation reports in a dedicated registry.
6. **CI/CD Integrations**
    - Pre-merge impact analysis, dependency risk scoring, auto-generated PR summaries, and test plan suggestions.
    - Uses KG + RAG + agentic workflows.

3.2 Full-Scale Data & Control Flow

Event→Graph→Agent Loop

1. **Kafka event arrives** (PR merged, new deploy, incident alert)
2. **Temporal workflow triggers** a LangGraph agent
3. Agent queries:

    - RAG for context,
    - KG for lineage, dependencies, and ownership,
    - Traces for runtime dependencies
4. Agent produces:
    - Risk report,
    - Suggested next steps,
    - Optional change requests (feature flag updates, documentation patches)
5. Human approval → execution (if relevant)

Environment Remediation Loop

1. Developer runs local diagnostic
2. Temporal launches workflow
3. Agent proposes fix → sandbox tests → human approves → final execution
4. Data logged to KG as environment incident lineage.

4. Security & Governance Architecture

- **RBAC inheritance** from GitHub/Jira/Slack ensures no new role systems (MVP-R25).
- **Redaction engine** for Slack/Jira PII and secrets.
- **LLM data minimization** via structured context assembler (MVP-R7).
- **Agent sandboxing**: Docker-in-Docker, ephemeral namespaces.
- **Approval gates** for any write-capable workflow.
- **Audit logs** captured via Temporal + centralized logging.

5. Deployment Model

MVP Deployment

- Single Kubernetes cluster
- Managed Weaviate (or self-hosted minimal cluster)
- Redis instance
- Backend, LLM Gateway, Observability stack
- Rolling upgrades with zero agents or workflow orchestrators to maintain simplicity

Full-Scale Deployment

- Multi-node Neo4j cluster
- Kafka cluster (managed recommended: MSK/Confluent/Aiven)
- Temporal Cloud or self-hosted Temporal cluster
- Scalable LangGraph agent workers
- Optional GPU node pool for Llama fine-tuning/inference
- Integration with org's CI/CD platform (GitHub Actions, GitLab, Argo)

6. Scaling & Performance Considerations

- **Caching-first strategy** reduces LLM load by targeting ≥80% hit rate.
- **Retrieval tier scaling**: Weaviate sharding across services or code domains.
- **Async pipelines** for indexing large repositories.
- **Latency targets**:

- MVP p95 < 4s
- Full-scale p95 < 2.5s via smarter context assembly and shorter inference paths.

- **Cost optimization** via:
  - Tiered model routing
  - Result caching
  - Fine-tuned small Llama models for high-volume repetitive tasks