HUS 26.1.1 FastAPI + GenAI Track

AI Product-to-Code: Multi-Agent Epic → User Stories → Implementation + Validation System

Overview

Build a sophisticated multi-agent system that transforms a high-level Product Request into structured Epics, then into User Stories, then into formal Specs (spec-driven development), and finally into working code with automated validation. The system employs specialized agents working together to plan, specify, implement, and verify features with real-time progress updates and full observability. Users can control execution (pause/resume), ask questions mid-run, and approve specs before implementation.

Use Case: Intelligent Product Planning + Implementation

Your system must handle:

- Product Request → Epic generation (with scope, priorities, dependencies)
- Epic → User Story generation (with acceptance criteria, estimates, edge cases)
- Spec-driven development step between story and implementation (Generate Spec → Approve Spec → Implement)
- Real-time progress visibility throughout (planning, spec, implementation, validation)
- Multi-agent orchestration where each agent performs one specific task
- Web-augmented best-practice lookup when needed (latest patterns, security guidelines, framework docs)
- User-controlled analysis/build flow with pause/resume functionality (checkpointing)
- Visual flow generation using Mermaid diagrams
- Role-based access with User/Admin distinction
- Complete observability and traceability of agent operations (including LLM calls)

Target Personas

Product Manager (PM)

Profile:

Business stakeholders who need to understand scope, priorities, and business logic.

Key Deliverables They Need:

- Epics with goals, scope, and priority suggestions
- User stories per epic with acceptance criteria
- User journey flows and business processes
- Feature dependencies
- Integration points with external services
- Business rules documentation
- Limitations and technical constraints

Software Engineer (SDE)

Profile: Technical team members who need implementable specs, code, and validation evidence.

Key Deliverables They Need:

- Architecture overview with module/component relationships
- API contracts (OpenAPI) and request/response schemas
- Data model/schema and data flow diagrams (if applicable)
- Implementation plan (files/modules to be created/modified)
- Generated code with clean structure and conventions
- Generated tests mapped to acceptance criteria
- Validation report (tests/lint/security) and auto-fix summaries
- Security/authentication flow documentation (if relevant)

Tech Stack

- LangGraph: Multi-agent orchestration with checkpointing (pause/resume)
- FastAPI: Backend API and WebSocket/SSE server for real-time updates
- pgvector: Semantic search over product requirements, stories, specs, and generated code artifacts
- External MCP Servers: File operations (create/read/patch) and knowledge tools
- Langfuse: Observability and token tracking for every LLM call and agent step
- LLM: OpenAI
- API Docs: Swagger
- Real-time updates: Your choice (WebSocket or SSE)
- Validation tooling (suggested; equivalents allowed): pytest (tests), ruff or black+flake8 (lint/format)

User Flow Overview

The system should support the following user journey:

1. Authentication & Onboarding

- User signs up or logs in (role: User or Admin)
- View existing projects

2. Requirement Intake:

- User provides the requirement as free text.
- Reference documents as optional

3. Epic Creation

- The system performs mandatory research and epics are generated.
- User can approve or provide suggestions to improve it.
- Once approved move on to the story generation phase.

4. Story Generation

- The user can provide the epic id for which the stories need to be created.
- Once approved move on the spec generation phase.

5. Spec Generation

- System generates a formal spec for the chosen story.
- Developer approves/rejects spec.
- Approved spec triggers implementation.

6. Code Generation

- Generate the code for the given Story ID using the spec.
- Validation for the given code generated.

7. Export & Share

- Downloads documentation and code in multiple formats
    - The docs should be saved as a markdown file or pdf.
- Saves analysis for future reference
- (Admin only) Manages user access and projects

Milestones

Milestone 1: Foundation - "User Can Start"

Goal:

Create a system where users can authenticate, create a project, submit a Product Request, and trigger backlog generation.

What Users Should Be Able to do:

- Authentication and Signup Flow
  - Basic Signup/Registration flow for new users (JWT authentication)
  - Login as either User or Admin
  - User can see their own projects
- Project Initiation
  - Submit Product Request as text
  - Optional supporting document upload
  - Create project with unique identifier
  - Trigger "Backlog Generation" run
- Basic Infrastructure
  - Project tracking and artifact storage
  - Configuration management
  - Error handling with user-friendly messages
  - Swagger available

Invalid Input Handling

System must gracefully handle and provide clear feedback for:

- Empty Product Request
- Unsupported or corrupted documents
- Files too large (define a limit, e.g., 20MB)
- Wrong input formats

Demo Scenario:

user signs up → logs in → Creates project → Submits Product Request → Starts run → Sees "Backlog Generation Started" + Project ID

Milestone 2: Research - "System Uses Web + Market Evidence"

Goal:

Implement web + market/best-practice research and store it as a first-class artifact used by downstream agents.

What Users Should Be Able to Do:

- Research Appendix Artifact
- View an artifact containing:
    - URLs consulted (citations)
    - key findings summary
    - how research impacted epics/stories/spec decisions

Research Requirement:

- Research must be web search (OpenAI web search or Tavily)
- Research must run before epic generation
- Results must be persisted per project run
- Research must contain the answer to which X was chosen.

Demo Scenario:

User starts backlog generation → System shows "Research in progress" → User sees 8 URLs + summary → System shows "Research Complete" → Epic generation begins using those findings

Milestone 3: Epic Generation

Goal:

Convert a Product Request into a prioritized epic backlog grounded in the research artifact.

What Users Should Be Able to Do:

Generate epics that include:

- Epic title + goal
- In-scope / out-of-scope
- Priority suggestion (e.g., P0/P1/P2) with reasoning
- Dependencies between epics
- Risks, assumptions, and open questions
- Success metrics / acceptance signals (high-level)

Epic Review & Approval Loop

- User can approve epics, or request edits
- User can add constraints (budget/time/tech constraints) and regenerate
- Once approved user stories can be created

Mermaid Diagram Output

Generate at least 1 Mermaid diagram: epic dependency graph (flowchart)

Demo Scenario:

System generates 6 epics → User edits constraints ("must support SSO") → System regenerates → User approves → Epic status becomes "Approved"

Milestone 4: User Story Generation - "Epics → Stories with Acceptance Criteria"

Goal:

For a chosen epic, generate implementable user stories with acceptance criteria and edge cases.

What Users Should Be Able to Do:

- Story Generation per Epic
  - User selects an Epic ID and triggers story generation
  - Stories must include:
    - User story statement (As a Frontend/Backend Developer)
    - Acceptance criteria (Given/When/Then)
    - Edge cases & failure modes
    - Non-functional requirements (performance/security/privacy if relevant)
    - Estimates (t-shirt size or points) with rationale
    - Dependencies (on other stories, epics, external systems)
- Story Review & Approval
  - Approve/reject stories
  - Edit or add constraints and regenerate

Demo Scenario:

User selects Epic #2 → System generates 10 stories → User rejects 2 and adds constraint "must be GDPR compliant" → Regenerate those 2 → Approve all → Stories ready for spec

Milestone 5: Spec-Driven Development - "Spec First, Then Code"

Goal:

Introduce a formal spec step that must be explicitly approved before any implementation begins.

What Users Should Be Able to Do:

For a chosen Story ID, generate a formal spec including:

- Overview + goals
- Detailed functional requirements mapped to acceptance criteria

- API contracts (OpenAPI snippets or endpoint definitions) if applicable
- Data model changes (tables/fields/relationships) if applicable
- Security considerations (authz/authn, input validation, rate limits)
- Error handling strategy
- Observability plan (logs/metrics/traces)
- Test plan mapping each acceptance criterion → test cases
- Implementation plan (files/modules to create/modify)

Approval Gate

- Spec must be explicitly approved before code generation can run
- Support reject-with-feedback → regenerate spec

Mermaid Diagram Output

Generate at least 2 Mermaid diagrams across spec work (examples):

- Sequence diagram for a primary user flow
- ER diagram (if DB involved)

Demo Scenario

User picks Story #7 → System generates spec → User rejects ("need rate limiting") → System regenerates with rate limiting section → User approves → Implementation becomes available

Milestone 6: Implementation + Validation - "Generate Working Code and Prove It"

Goal:

Generate code from the approved spec and run automated validation with clear reporting and auto-fix loops.

What Users Should Be Able to Do:

- Code Generation
    - Generate code for the provided Story ID
    - Use structured project layout and conventions
    - Generate tests aligned to acceptance criteria
- Validation Pipeline
    - Run validation steps and store a validation report:
    - Unit tests (pytest)
    - Lint/format (ruff or black+flake8)
    - Store failures with actionable messages and file references

- Auto-Fix Loop (Controlled)
    - If validation fails, an agent can propose fixes
    - User can choose: auto-apply fixes or request manual review
    - Re-run validation after fixes
    - Final output includes "Validation Passed" evidence

Export Outputs

- Epics (Markdown/JSON)
- Stories (Markdown/JSON)
- Specs (Markdown/PDF optional)
- Code bundle (zip)
- Validation report (Markdown/JSON)

Demo Scenario:

- User story ID → Code generation runs → Tests fail → System proposes 2 patches → Agent does auto-fix → Tests pass → User downloads code zip + validation report

Milestone 7: Real-Time Control, Observability & Admin - "System Transparency"

Goal:

Provide full real-time visibility, pause/resume control, and admin oversight; add Langfuse traces for every LLM call.

What Users Should Be Able to Do:

- Real-Time Progress Updates
    - Live stage tracking: Research → Epics → Stories → Specs → Code → Validation
    - File-by-file updates during code generation (e.g., "patching app/routes/auth.py")
    - Percent completion estimates per stage
    - Activity feed with warnings/errors
- Pause/Resume with Checkpointing (LangGraph)
    - Pause at any time
    - Resume from the exact step with state restored
    - Preserve user feedback injected mid-run
- Interactive Q&A Mid-Run
    - Questions about current state ("What are you working on now?", "What's blocked?")

    - Add instructions ("Prioritize P0 security", "Don't use Redis", "Prefer async endpoints")
- Admin Dashboard - Project & User Management
    - View all users and projects
    - CRUD users and projects
    - See run statuses and basic analytics (active runs, completion rates)

Separate Flow: Langfuse Integration for Token Tracking

Goal: Track and optimize LLM usage independently.

What Should Be Tracked:

- Every LLM call during research/planning/spec/implementation/validation
- Token usage per project, per run, per agent
- Cost breakdown by operation type
- Performance metrics (latency, success rate)
- Complete execution traces

How to Implement:

- Integrate Langfuse SDK in backend
- Wrap all LLM calls with Langfuse tracing
- Create separate observability dashboard (can be Langfuse UI)
- Link traces back to projects for debugging

Demo Scenario:

Developer opens Langfuse dashboard → Selects a project → Sees complete trace tree with 15 agent calls → Views token usage: 12,500 tokens → Cost: $0.40 → Identifies one agent using 40% of tokens → Optimizes that agent's prompt

Brownie Points: End-to-End Traceability Matrix

- Generate an artifact that maps: Requirement paragraphs → Epics → Stories → Spec sections → Code files/functions → Tests
- Stored as JSON + rendered table
- Helps prove "nothing is missing" and aids validation

Remember: Start simple, then iterate. Each milestone builds on the previous one. Focus on making each component work well before moving to the next!