AI-Native Developer Assignment: Guidelines for 3-5 Years Experience

This assignment evaluates your development skills, understanding of core software design principles, and your ability to leverage AI-powered code assistants effectively within the context of an AI-native SaaS platform for ITSM and ITOM.

1. Assignment Goal & Context

The primary goal is to assess your proficiency in:

- **Building a functional full-stack application demonstrating integration.**
- Implementing basic AI simulation logic on the backend.
- Applying fundamental software design patterns.
- Writing **comprehensive test cases for both frontend and backend**.
- **Crucially: Effectively using an AI code assistant (e.g., GitHub Copilot, Cursor AI, Windsurf AI) to accelerate development, improve code quality, and demonstrate prompt engineering skills.** We also want to see how you might integrate AI assistants into broader project workflows using "rules" or "templates" if your chosen assistant supports it.
- **Integrating with a real AI API (OpenAI, Groq, or Google Gemini) for AI-driven insights.**

The assignments are framed around core features of an AI-native ITSM and ITOM platform, designed to provide intelligent assistance to IT operations teams.

2. General Guidelines for Candidates

- **Time Commitment:** Each assignment is designed to take approximately 2-3 hours of focused work. Please complete it at your own pace.
- **Technology Stack (Mandatory):**
    - **Backend:** Golang
    - **Frontend:** React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component
- **AI Code Assistant Usage is Mandatory:** You must use an AI code assistant throughout the development process.
- **The candidate must submit unit test cases achieving at least 80% code coverage to demonstrate a functional model, along with proper documentation and clearly commented code.**
- **AI API Integration is Mandatory:** You must integrate with at least one of the following AI APIs for your AI logic: OpenAI, Groq, or Google Gemini.
- **Logging:** You are required to maintain a detailed log of your AI code assistant interactions. This includes:
    - **Prompts:** The exact prompts you used.
    - **AI Responses/Suggestions:** The relevant code or text snippets suggested by the AI.

    - **Your Action:** What you did with the AI's suggestion (accepted, modified, rejected, asked for refinement, etc.) and why.
    - **Context:** A brief description of the task you were performing when you used the AI.
- **Prompt Engineering & Rules/Templates:** In your README.md, you should also discuss:
    - Your prompt engineering strategy (e.g., how you made prompts specific, provided context, iterated).
    - If your AI assistant (like Cursor AI) supports "rules" or "templates" for project-wide standards, show how you would use or could use them for this assignment (even if for conceptual benefit). Provide examples.
- **Completeness over Perfection:** Focus on delivering a working solution that meets the core requirements of your chosen modules. Clean, readable code and thoughtful design are more important than an exhaustive feature set.
- **Open-Book:** Feel free to use any online resources, documentation, or tutorials.

---

3. Deliverables

Please submit the following:

- **Working Codebase:** A complete, runnable application demonstrating your chosen modules. Submit it as a zipped folder or a link to a Git repository.
- **Comprehensive README.md File:** This is a critical component and must include:
    - **Setup Instructions:** Clear, step-by-step instructions on how to set up and run your implemented components (both frontend and backend). Include any necessary dependencies, environment variables (e.g., API keys for OpenAI/Groq/Gemini), and build/run commands.
    - **Core Functionality Overview:** A brief explanation of what the application does and how to use it, including the user journey and logic you defined for your chosen assignment.
    - **Software Design Choices & Justification:** A detailed explanation of key architectural decisions made for the modules you implemented. This should cover:
        - Chosen technologies (Golang, and your chosen frontend framework) and why.
        - Database schema design (if applicable) and relationships.
        - API design (endpoints, request/response formats) for implemented components.
        - Frontend component structure and state management approach.
        - How the AI API (OpenAI/Groq/Gemini) is integrated and consumed.

- **Test Strategy & Coverage:** Describe your approach to testing both the frontend and backend. Explain what types of tests you implemented and why.
- **AI Code Assistant Usage Log (Detailed):** As described in Section 2, provide a clear, structured log of your AI interactions, including prompts, AI responses, your actions, and context. Aim for at least 5-7 significant interactions.
- **Prompt Engineering & AI Assistant Strategy Discussion:** Describe your approach to crafting effective prompts. What techniques did you use (e.g., being specific, providing examples, iterating)? If your AI assistant supports "rules" (like Cursor AI's Rules) or "templates" for enforcing coding standards or generating boilerplate for larger projects, discuss how you might use or could have used them for this assignment. Provide a concrete example of a rule/template you would define for a larger project.
- **Assumptions Made:** Any assumptions or simplifications made during development.
- **Potential Improvements & Future Enhancements:** Ideas for how the application could be extended or improved with more time.

AI Assignments for 3-5 Years Experience

You **MUST** implement a **end to end solution** for **ONE** of the following assignments. This includes a Golang backend and a frontend built with React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component. **Test cases for both frontend and backend are mandatory.** You are responsible for defining the full user journey, detailed logic, and specific data models to achieve the expected output.

Mandatory Core Modules for ALL Candidates:

- **AI Code Assistant Usage & Prompt Engineering:** As per Section 2.
- **Core AI Logic (API Integration):** Implement the logic that leverages an external AI API (OpenAI, Groq, or Google Gemini) to process raw operational event data and generate AI-driven insights (e.g., "Category", "Suggested Action").
    - **Expected Output (from this module):** A Golang function or module that takes an event object as input, calls the chosen AI API, processes its response, and returns the event object augmented with derived fields like ai_category and ai_suggested_action.
- **Data Persistence:** Store the ingested events and their AI-generated insights.
    - **Expected Output:** A defined database schema (e.g., for SQLite, PostgreSQL, or MongoDB) and basic CRUD operations (at least create and read) for your event data, implemented in Golang.

- **Backend Event Ingestion API (Golang):** Build a RESTful API endpoint to ingest raw operational events (e.g., POST /events). This API will call your "Core AI Logic" and then persist the event with its generated insights.
    - **Expected Output:** A working Golang API endpoint (using net/http, Gin, or Echo) that accepts JSON event data, processes it through your AI logic (which integrates with OpenAI/Groq/Gemini), stores it, and returns a success response. Basic error handling for API requests.
- **Frontend Insight Dashboard (React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component):** Build a simple web UI to interact with your Golang backend, displaying the operational events and their AI- generated insights. Allow for basic viewing and perhaps one simple filter.
    - **Expected Output:** A web page that allows users to input event data, submit it to your Golang backend, and then lists the ingested events with their AI-derived insights (ai_category, ai_suggested_action, etc.). A detail view for a single event should also be available. The UI should be responsive and user-friendly.

Assignment 1: AI-Powered Incident Triage Assistant

Scenario: Your platform aims to help IT support teams quickly prioritize and categorize incoming service incidents. You'll build a full-stack "Event-to-Insight" system where the "events" are incident reports. The system should ingest an incident via the frontend, process it through the Golang backend and an AI API to suggest its severity and category, and then display it in a web dashboard.

Expected Output:

- **Backend (Golang):**
    - A POST /incidents API endpoint that accepts incident data (e.g., title, description, affected_service).
    - This endpoint should call the chosen AI API (OpenAI/Groq/Gemini) to determine ai_severity (e.g., Low, Medium, High, Critical) and ai_category (e.g., Network, Software, Hardware, Security).
    - The processed incident, including AI-derived fields, should be stored in a database.
    - A GET /incidents endpoint to retrieve all incidents.
    - **Unit and Integration Tests** for API endpoints, AI integration logic, and database operations.
- **Frontend (React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component):**
    - A web interface with a form to input incident data (title, description, affected service).

    - Upon submission, the UI sends the data to your Golang POST /incidents endpoint.
    - A list view displaying all incidents retrieved from GET /incidents, showing their original details and the ai_severity and ai_category.
    - A detail view for a single incident.
    - **Unit Tests** for components and **End-to-End Tests** (e.g., using Playwright or Cypress) for the user journey.

Assignment 2: AI-Driven Alert Correlation & Suggestion

Scenario: Your platform processes raw alerts from various monitoring systems. You'll build a full-stack "Event-to-Insight" system where the "events" are incoming alerts. The system should ingest an alert via the frontend, use Golang and an AI API to suggest a potential root cause and action, and potentially correlate it with other recent alerts from the same source.

Expected Output:

- **Backend (Golang):**
    - A POST /alerts API endpoint that accepts alert data (e.g., source_system, metric_type, value_exceeded, timestamp).
    - This endpoint should call the chosen AI API (OpenAI/Groq/Gemini) to determine ai_suggested_root_cause (e.g., Disk Full, Network Latency, Service Down), ai_suggested_action (e.g., Restart Service, Check Logs, Scale Up), and optionally ai_correlation_id (if applicable, based on similar recent alerts from the same source system).
    - The processed alert, including AI-derived fields, should be stored in a database.
    - A GET /alerts endpoint to retrieve all alerts, possibly with a filter for source_system or ai_correlation_id.
    - **Unit and Integration Tests** for API endpoints, AI integration logic, and database operations.
- **Frontend (React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component):**
    - A web interface with a form to input alert data (source system, metric type, value exceeded).
    - Upon submission, the UI sends the data to your Golang POST /alerts endpoint.
    - A list view displaying all alerts retrieved from GET /alerts, showing their original details and the ai_suggested_root_cause, ai_suggested_action, and ai_correlation_id.
    - Optionally, the list view could group alerts by ai_correlation_id.
    - A detail view for a single alert.

- **(If added it will be plus point)Unit Tests** for components and **End-to-End Tests** (e.g., using Playwright or Cypress) for the user journey.

Assignment 3: AI-Assisted Knowledge Base Search

Scenario: Your platform needs a self-service component where users can quickly find answers to IT questions. You'll build a full-stack "Event-to-Insight" system where the "events" are user search queries. The system should accept a user's problem query via the frontend, use Golang and an AI API to find relevant knowledge base articles (simulated as hardcoded data or a simple in-memory store in the backend), and provide a summarized answer.

Expected Output:

- **Backend (Golang):**
  - A POST /search-query API endpoint that accepts a user's query (string).
  - This endpoint should simulate a knowledge base (e.g., a hardcoded slice of structs representing articles with title and content).
  - It should then call the chosen AI API (OpenAI/Groq/Gemini) with the user's query and the simulated article content to:
    - Determine ai_summary_answer (a concise answer based on the relevant articles).
    - Identify ai_relevant_articles (a list of titles/IDs of articles).
  - The search query and its AI-generated results should be stored in a database (optional, but good for demonstrating persistence).
  - **Unit and Integration Tests** for API endpoints, AI integration logic, and knowledge base simulation.
- **Frontend (React.js, Vue.js, Next.js, Angular, Plain JavaScript or Custom Script based UI Component):**
  - A web interface with a search bar for users to type their queries.
  - When a user submits a query, the UI sends it to your Golang POST /search-query endpoint.
  - The UI then displays the ai_summary_answer generated by the AI.
  - It should also list the ai_relevant_articles (titles/IDs), allowing users to click to view full simulated article content (e.g., in a modal or a new section on the page).
  - **Unit Tests** for components and **End-to-End Tests** (e.g., using Playwright or Cypress) for the user journey.

---

4. Evaluation Criteria

Each assignment will be evaluated based on the following:

- **Functionality & Robustness (30%):**
    - Does the candidate's chosen full-stack solution work as expected?
    - Is the implemented solution stable and robust?
    - **Golang:** Demonstrated understanding of concurrency patterns and error handling.
    - **Frontend (React.js/Vue.js/Next.js):** Responsiveness, interactivity, and effective state management.
    - **AI API Integration:** Correct and effective integration with OpenAI, Groq, or Google Gemini APIs.
- **Overall System Design (20%):**
    - How well do the implemented frontend and backend modules fit into a cohesive system?
    - **Golang Backend Design:** Idiomatic Go design, effective use of Go features (e.g., goroutines, channels, interfaces). Clean API design.
    - **Frontend Design:** Component architecture, state management, and clear API interaction strategy.
    - **Database Schema:** Is the data model appropriate for the problem (even if simple)?
- **Code Quality & Best Practices (15%):**
    - High readability, maintainability, and consistency within chosen languages (Golang and chosen frontend framework).
    - Adherence to language-specific best practices (e.g., Go idioms, React Hooks rules, Vue.js reactivity principles, Next.js conventions).
    - Comprehensive error handling and logging within implemented modules.
- **Test Coverage & Quality (10%):**
    - Presence of meaningful **unit tests for both backend (Golang) and frontend (chosen framework)** components.
    - Presence of **integration tests for the backend (Golang)**.
    - Presence of **end-to-end tests for the full-stack application (e.g., Playwright, Cypress)** covering core user journeys.
    - Tests demonstrate an understanding of testing best practices.
- **AI Code Assistant Utilization & Prompt Engineering (15%):**
    - **Usage Log Quality:** Is the AI code assistant usage log detailed, clear, and comprehensive (prompts, AI responses, actions, context)?
    - **Effectiveness:** Does the log demonstrate how the AI assistant genuinely aided productivity (e.g., boilerplate, function stubs, refactoring, debugging) in the implemented modules?
    - **Prompt Engineering Skill:** How well did the candidate craft their prompts? Did they iterate on prompts? Did they use specific instructions, context, or examples?

    - **Rules/Templates Discussion:** Does the candidate demonstrate an understanding of how AI assistant "rules" or "templates" could be used for larger projects, and provide a concrete example relevant to their chosen technology stack?
- **Documentation & Communication (10%):**
    - Clarity and completeness of the README.md (setup instructions for both frontend and backend, design choices, AI usage log, assumptions, future improvements).
    - Effectiveness in explaining the overall system flow and interactions between frontend and backend.
