Creating a project plan architecture for the given Cash and Electronic Trading use-case, leveraging Java Spring Boot, DART (Data Analysis and Reporting Tool) AI, and Generative AI (Gen AI), involves defining a structured approach to meet the requirements outlined in the image. Below is a detailed project plan architecture, broken down into phases, components, and technologies. This plan assumes a modern, scalable, and AI-driven system to handle the workflow for Equities Cash and similar trading scenarios.

**Project Plan Architecture: Cash and Electronic Trading System with Java Spring Boot, DART AI, and Gen AI**

**1. Project Overview**

The goal is to develop an intelligent system for Cash and Electronic Trading that automates impact assessments, generates requirements, identifies impacted code, and creates test cases and code using AI-driven models (DART AI and Gen AI). The system will integrate with existing workflows (e.g., Equities Cash), use historical documentation, and reduce reliance on manual SME (Subject Matter Expert) input over time.

**2. Key Objectives**

* Automate impact assessment for new regulatory/client tasks using historical data and AI models.
* Identify impacted classes/code in Git repositories and generate test cases and code for applications.
* Leverage DART AI for data analysis and reporting, and Gen AI for generating requirements, code, and test cases.
* Achieve 90% automation of requirements generation, minimizing SME resourcing.

**3. Architecture Components**

The architecture will consist of the following layers and components:

**3.1. Frontend Layer**

* **Technology**: HTML, CSS, JavaScript (React.js or Angular for a responsive UI).
* **Purpose**: Provide a user interface for SMEs and BAs (Business Analysts) to input manual data, view impact assessments, generated requirements, test cases, and code.
* **Integration**: Communicate with the backend via RESTful APIs.

**3.2. Backend Layer**

* **Technology**: Java Spring Boot (for RESTful services, business logic, and API management).
* **Purpose**:
  + Process historical documentation (Confluence, PDFs, etc.) and manual inputs.
  + Host AI models (DART AI and Gen AI) for impact assessment, requirement generation, and code/test case generation.
  + Manage database interactions and Git integrations.
* **Key Features**:
  + RESTful API endpoints for frontend-backend communication.
  + Spring Data JPA for database operations (e.g., PostgreSQL or MySQL).
  + Spring Security for authentication and authorization.

**3.3. AI and Machine Learning Layer**

* **Technology**: DART AI (custom or third-party AI tool for data analysis/reporting) and Gen AI (e.g., custom-trained models or platforms like Hugging Face, OpenAI, or Google AI for generative tasks).
* **Purpose**:
  + DART AI: Analyze historical data, workflows, and impacted systems to establish baselines and avoid stale data reuse.
  + Gen AI: Generate impact assessments, requirements (90% automation), test cases, and code snippets based on learned patterns.
* **Training Data**: Historical documentation, workflow data (e.g., Equities Cash), and existing codebases.
* **Model Iterations**: Train models over multiple iterations to improve accuracy and reduce SME dependency.

**3.4. Data Layer**

* **Technology**: PostgreSQL or MySQL (for structured data), MongoDB (for unstructured data like PDFs, Confluence pages).
* **Purpose**: Store historical documentation, workflow data, impacted systems, generated requirements, test cases, and code.
* **Integration**: Use Spring Boot’s JPA/Hibernate for ORM and JDBC for database connectivity.

**3.5. DevOps and Version Control**

* **Technology**: Git (for version control), Jenkins or GitHub Actions (for CI/CD), Docker (for containerization), Kubernetes (for orchestration).
* **Purpose**:
  + Identify impacted classes/code across Git projects (Step-2 requirement).
  + Automate build, test, and deployment pipelines.
  + Ensure scalability and reliability of the system.

**3.6. Testing Framework**

* **Technology**: JUnit, Mockito (for unit testing), Selenium (for UI testing), and custom scripts for AI-generated test cases.
* **Purpose**: Validate generated test cases (front to back) and ensure code quality.

**4. Project Phases**

**Phase 1: Requirement Analysis and Planning (2-4 weeks)**

* Gather historical documentation and list impacted systems (Equities Cash workflow).
* Define AI model requirements (DART AI for analysis, Gen AI for generation).
* Create a detailed project roadmap, including milestones and resource allocation.

**Phase 2: System Design and Architecture (4-6 weeks)**

* Design the frontend, backend, AI, and data layers as described above.
* Set up Git repositories for version control and establish CI/CD pipelines.
* Prototype DART AI and Gen AI models using sample data.

**Phase 3: Development and Integration (8-12 weeks)**

* Develop the Java Spring Boot backend with RESTful APIs.
* Integrate DART AI for data analysis and baseline establishment.
* Train and integrate Gen AI for generating requirements, test cases, and code.
* Build the frontend UI for user interaction.
* Connect to Git for identifying impacted classes and generating code/test cases.

**Phase 4: Testing and Validation (4-6 weeks)**

* Test API endpoints, AI models, and generated outputs (requirements, test cases, code).
* Perform integration testing across frontend, backend, and AI layers.
* Validate that the system achieves 90% automation of requirements with minimal SME input.

**Phase 5: Deployment and Rollout (2-4 weeks)**

* Deploy the system using Docker and Kubernetes for scalability.
* Roll out new releases and ensure the model identifies new baselines to avoid stale data.
* Gather feedback from SMEs and BAs for iterative improvements.

**Phase 6: Maintenance and Iteration (Ongoing)**

* Monitor system performance and AI model accuracy.
* Retrain models with new data to improve over multiple iterations.
* Reduce SME resourcing within the organization as automation improves.

**5. Tools and Technologies Summary**

* **Programming Languages**: Java (Spring Boot), JavaScript (Frontend), Python (AI model development).
* **Frameworks**: Spring Boot, React.js/Angular, JUnit, Mockito, Selenium.
* **AI Tools**: DART AI (custom or third-party), Gen AI (custom-trained or platforms like Hugging Face).
* **Databases**: PostgreSQL/MySQL, MongoDB.
* **DevOps**: Git, Jenkins/GitHub Actions, Docker, Kubernetes.

**6. Risk Mitigation**

* **Data Quality**: Ensure historical documentation is accurate and well-structured for AI training.
* **AI Accuracy**: Continuously validate AI outputs (impact assessments, requirements, code) with SME reviews.
* **Scalability**: Use cloud-based solutions (e.g., AWS, Azure) for scalability and performance.

**7. Success Metrics**

* Achieve 90% automation of requirements generation within 6 months.
* Reduce SME resourcing by 50% within the first year.
* Ensure 99% accuracy in identifying impacted classes/code and generating valid test cases/code.

### Project Roadmap: Cash and Electronic Trading System

**Project Start Date: February 24, 2025**

**Estimated Completion Date: September 22, 2025**

**Total Duration: 30 weeks**

**Phase 1: Requirement Analysis and Planning (Weeks 1-4)**

**Duration**: February 24, 2025 – March 23, 2025  
**Objective**: Gather requirements, define scope, and create a project plan.

**Milestones and Deliverables:**

* **Week 1-2 (February 24 – March 8, 2025)**
  + **Milestone 1: Requirement Gathering Complete**
    - Collect historical documentation (e.g., Confluence, PDFs) and list impacted systems for Equities Cash workflow.
    - Conduct stakeholder interviews with SMEs and BAs to understand manual inputs and expectations.
  + **Deliverable**: Requirement Document (PDF/Word) outlining workflows, impacted systems, and AI model needs.
* **Week 3-4 (March 9 – March 23, 2025)**
  + **Milestone 2: Project Plan Finalized**
    - Define AI model requirements (DART AI for analysis, Gen AI for generation).
    - Create a detailed project roadmap, resource allocation, and risk assessment.
  + **Deliverable**: Project Roadmap Document, Resource Allocation Plan, and Risk Mitigation Strategy.

**Phase 2: System Design and Architecture (Weeks 5-10)**

**Duration**: March 24, 2025 – May 25, 2025  
**Objective**: Design the system architecture, including frontend, backend, AI, and data layers.

**Milestones and Deliverables:**

* **Week 5-6 (March 24 – April 6, 2025)**
  + **Milestone 3: Architecture Design Complete**
    - Finalize the layered architecture (frontend, backend, AI, data, DevOps).
    - Design database schema (PostgreSQL/MySQL, MongoDB) and API specifications (RESTful endpoints).
  + **Deliverable**: System Architecture Diagram, Database Schema, and API Specification Document.
* **Week 7-8 (April 7 – April 20, 2025)**
  + **Milestone 4: AI Model Prototyping Complete**
    - Prototype DART AI for data analysis and baseline establishment.
    - Prototype Gen AI for generating impact assessments, requirements, and code/test cases using sample data.
  + **Deliverable**: AI Model Prototypes (Python scripts/documentation), Sample Outputs (e.g., generated requirements).
* **Week 9-10 (April 21 – May 25, 2025)**
  + **Milestone 5: DevOps Setup Complete**
    - Set up Git repositories, CI/CD pipelines (Jenkins/GitHub Actions), and containerization (Docker).
    - Establish Kubernetes for orchestration (if cloud-based deployment is planned).
  + **Deliverable**: Git Repository Structure, CI/CD Pipeline Configuration, Docker/Kubernetes Setup Documentation.

**Phase 3: Development and Integration (Weeks 11-22)**

**Duration**: May 26, 2025 – August 17, 2025  
**Objective**: Develop and integrate the frontend, backend, AI models, and Git integrations.

**Milestones and Deliverables:**

* **Week 11-14 (May 26 – June 22, 2025)**
  + **Milestone 6: Backend Development Complete**
    - Build Java Spring Boot backend with RESTful APIs for processing documentation, manual inputs, and AI outputs.
    - Integrate Spring Data JPA for database operations.
  + **Deliverable**: Backend Codebase (Java/Spring Boot), API Documentation.
* **Week 15-18 (June 23 – July 20, 2025)**
  + **Milestone 7: AI Model Integration Complete**
    - Integrate DART AI for analyzing historical data and establishing baselines.
    - Integrate Gen AI for generating 90% of requirements, test cases, and code snippets.
    - Train models with historical data and validate outputs with SME feedback.
  + **Deliverable**: Trained AI Models (DART AI, Gen AI), Integration Scripts, Validation Reports.
* **Week 19-22 (July 21 – August 17, 2025)**
  + **Milestone 8: Frontend and Git Integration Complete**
    - Develop frontend UI (React.js/Angular) for user interaction.
    - Integrate with Git to identify impacted classes/code and generate code/test cases per application stack > components.
  + **Deliverable**: Frontend Codebase, Git Integration Scripts, Generated Code/Test Case Samples.

**Phase 4: Testing and Validation (Weeks 23-28)**

**Duration**: August 18, 2025 – September 21, 2025  
**Objective**: Test the system, validate AI outputs, and ensure quality.

**Milestones and Deliverables:**

* **Week 23-25 (August 18 – September 7, 2025)**
  + **Milestone 9: Unit and Integration Testing Complete**
    - Test backend APIs, AI models, and frontend components using JUnit, Mockito, and Selenium.
    - Validate AI-generated outputs (requirements, test cases, code) against SME/BA feedback.
  + **Deliverable**: Test Reports, Bug Fixes, Validation Feedback Document.
* **Week 26-28 (September 8 – September 21, 2025)**
  + **Milestone 10: System Validation Complete**
    - Ensure 90% automation of requirements generation and 50% reduction in SME resourcing (initial target).
    - Perform load testing and security testing (e.g., Spring Security).
  + **Deliverable**: Validation Report, Performance Metrics, Security Audit Report.

**Phase 5: Deployment and Rollout (Weeks 29-30)**

**Duration**: September 22, 2025 – October 5, 2025  
**Objective**: Deploy the system, roll out new releases, and establish maintenance processes.

**Milestones and Deliverables:**

* **Week 29-30 (September 22 – October 5, 2025)**
  + **Milestone 11: System Deployment Complete**
    - Deploy the system using Docker and Kubernetes (or cloud platforms like AWS/Azure).
    - Roll out the first release and ensure the model identifies new baselines to avoid stale data.
    - Train SMEs/BAs on using the system.
  + **Deliverable**: Deployed System, User Training Manual, Release Notes.

**Phase 6: Maintenance and Iteration (Ongoing, Starting October 6, 2025)**

**Objective**: Monitor performance, retrain models, and improve automation over time.

**Milestones and Deliverables (Quarterly):**

* **Quarter 1 (October 2025 – December 2025)**
  + **Milestone 12: Initial Performance Review**
    - Assess system performance, AI accuracy, and SME resourcing reduction.
    - Retrain models with new data and feedback.
  + **Deliverable**: Performance Report, Updated AI Models.
* **Quarter 2 and Beyond (Ongoing)**
  + Continuously monitor, iterate, and refine the system to achieve long-term goals (e.g., 99% accuracy, full SME independence).

**Key Assumptions**

* Access to historical documentation, workflows, and Git repositories is available from the start.
* SME and BA availability for feedback and validation is ensured throughout the project.
* Cloud infrastructure (e.g., AWS, Azure) or on-premises servers are available for deployment.

**Risks and Mitigation**

* **Risk**: Poor data quality for AI training.
  + **Mitigation**: Clean and preprocess data before training, involve SMEs for validation.
* **Risk**: AI model inaccuracy.
  + **Mitigation**: Use iterative training, SME reviews, and validation checkpoints.
* **Risk**: Delays in development/integration.
  + **Mitigation**: Allocate buffer time in the timeline and prioritize critical features.

**Resource Allocation**

* **Team Roles**:
  + 2 Java Developers (Spring Boot, backend).
  + 2 Frontend Developers (React.js/Angular).
  + 2 AI/ML Engineers (DART AI, Gen AI).
  + 1 DevOps Engineer (Git, Jenkins, Docker, Kubernetes).
  + 1 Project Manager (overseeing timeline, milestones).
  + 2 SMEs/BAs (for input and validation).

**High-Level Design (HLD)**

**1. Overview**

The HLD provides a high-level architectural overview of the Cash and Electronic Trading system, focusing on its major components, interactions, and technology stack. The system aims to automate workflows, reduce SME dependency, and integrate AI-driven solutions for impact assessment and code generation.

**2. System Architecture**

The system is structured in a layered, modular architecture to ensure scalability, maintainability, and performance.

**2.1. Layers**

* **Presentation Layer (Frontend)**
  + **Description**: User interface for SMEs, BAs, and developers to interact with the system, input manual data, and view outputs (e.g., impact assessments, generated requirements, test cases, and code).
  + **Technology**: HTML, CSS, JavaScript (React.js or Angular).
  + **Interaction**: Communicates with the backend via RESTful APIs.
* **Business Logic Layer (Backend)**
  + **Description**: Handles core business logic, processes historical documentation, manages AI model integration, and generates outputs.
  + **Technology**: Java Spring Boot (RESTful services, Spring Data JPA, Spring Security).
  + **Interaction**: Interfaces with the frontend, AI layer, data layer, and Git repositories.
* **AI and Machine Learning Layer**
  + **Description**: Contains DART AI for data analysis and baseline establishment, and Gen AI for generating requirements, test cases, and code.
  + **Technology**: Custom-trained models (Python/TensorFlow/PyTorch), third-party platforms (e.g., Hugging Face, OpenAI).
  + **Interaction**: Receives inputs from the backend, processes data, and returns generated outputs.
* **Data Layer**
  + **Description**: Stores historical documentation, workflow data, impacted systems, generated requirements, test cases, and code.
  + **Technology**: PostgreSQL/MySQL (structured data), MongoDB (unstructured data like PDFs, Confluence pages).
  + **Interaction**: Accessed via Spring Boot’s JPA/Hibernate for ORM and JDBC.
* **DevOps and Integration Layer**
  + **Description**: Manages version control, CI/CD pipelines, and deployment for scalability and reliability.
  + **Technology**: Git (version control), Jenkins/GitHub Actions (CI/CD), Docker (containerization), Kubernetes (orchestration).
  + **Interaction**: Integrates with the backend to identify impacted classes/code and deploy updates.

**2.2. Major Components**

* **Impact Assessment Engine**: Uses DART AI to analyze historical documentation and manual inputs, generating impact assessments for new regulatory/client tasks.
* **Requirement Generator**: Leverages Gen AI to produce 90% of requirements, reducing SME resourcing.
* **Code and Test Case Generator**: Uses Gen AI to generate code and test cases for impacted classes/components, integrated with Git repositories.
* **Workflow Manager**: Manages Equities Cash and similar workflows, ensuring new baselines are established to avoid stale data reuse.
* **User Interface**: Provides a dashboard for input, visualization, and output review.

**3. Data Flow**

1. **Input**: Historical documentation (PDFs, Confluence), manual inputs from SMEs/BAs, and Git repository data.
2. **Processing**: Backend processes inputs, sends them to DART AI for analysis and Gen AI for generation.
3. **Output**: Generated impact assessments, requirements, test cases, and code, displayed via the frontend or stored in the data layer.
4. **Integration**: Git identifies impacted classes, and CI/CD pipelines deploy updates.

**4. Non-Functional Requirements**

* **Scalability**: Handle increasing volumes of documentation and transactions using cloud-based solutions (e.g., AWS, Azure).
* **Performance**: Process impact assessments and generate outputs within seconds for real-time use.
* **Security**: Ensure data privacy with Spring Security, SSL, and role-based access control.
* **Reliability**: Achieve 99.9% uptime with Kubernetes and monitoring tools.

**5. Technology Stack**

* **Languages**: Java (Spring Boot), JavaScript (Frontend), Python (AI models).
* **Frameworks**: Spring Boot, React.js/Angular, JUnit, Mockito, Selenium.
* **AI Tools**: DART AI (custom or third-party), Gen AI (custom-trained or platforms like Hugging Face).
* **Databases**: PostgreSQL/MySQL, MongoDB.
* **DevOps**: Git, Jenkins/GitHub Actions, Docker, Kubernetes.

**Low-Level Design (LLD)**

**1. Overview**

The LLD provides a detailed, granular design of the system’s components, including class diagrams, API endpoints, database schemas, and AI model workflows. This design focuses on implementation details for developers.

**2. Class Diagram (Backend - Java Spring Boot)**

Below is a simplified class structure for key components:

**2.1. ImpactAssessmentService**

java

WrapCopy

public class ImpactAssessmentService {

private DARTAIClient dartAIClient;

private DocumentRepository documentRepo;

public ImpactAssessment generateImpactAssessment(Document input, List<String> impactedSystems) {

*// Process historical data and manual inputs*

DataAnalysisResult analysis = dartAIClient.analyze(input, impactedSystems);

return new ImpactAssessment(analysis.getImpactedAreas(), analysis.getBaseline());

}

}

**2.2. RequirementGenerator**

java

WrapCopy

public class RequirementGenerator {

private GenAIClient genAIClient;

public List<Requirement> generateRequirements(ImpactAssessment impact, Workflow workflow) {

*// Use Gen AI to generate 90% of requirements*

String prompt = "Generate requirements for " + impact.getDescription() + " in " + workflow.getName();

return genAIClient.generate(prompt, Requirement.class);

}

}

**2.3. CodeTestCaseGenerator**

java

WrapCopy

public class CodeTestCaseGenerator {

private GenAIClient genAIClient;

private GitClient gitClient;

public GeneratedOutput generateCodeTestCases(String impactedClass, ApplicationStack stack) {

*// Identify impacted code in Git*

String classCode = gitClient.getCode(impactedClass);

*// Generate code and test cases using Gen AI*

String code = genAIClient.generateCode(classCode, stack);

String testCases = genAIClient.generateTestCases(classCode, stack);

return new GeneratedOutput(code, testCases);

}

}

**2.4. WorkflowManager**

java

WrapCopy

public class WorkflowManager {

private BaselineRepository baselineRepo;

public void updateBaseline(Release release) {

*// Establish new baseline to avoid stale data*

baselineRepo.save(new Baseline(release.getData(), release.getTimestamp()));

}

}

**3. API Endpoints (RESTful - Spring Boot)**

* **GET /api/impact/assessment/{workflowId}**
  + **Description**: Retrieve impact assessment for a specific workflow (e.g., Equities Cash).
  + **Input**: Workflow ID, historical data.
  + **Output**: JSON object containing impacted systems and assessment details.
* **POST /api/requirements/generate**
  + **Description**: Generate requirements using Gen AI.
  + **Input**: ImpactAssessment object, manual inputs.
  + **Output**: List of Requirement objects (JSON).
* **POST /api/code-test-cases/generate**
  + **Description**: Generate code and test cases for impacted classes.
  + **Input**: ImpactedClass, ApplicationStack.
  + **Output**: GeneratedOutput object (code and test cases in JSON).
* **POST /api/baseline/update**
  + **Description**: Update baseline after a new release to avoid stale data.
  + **Input**: Release object.
  + **Output**: Success/Failure response.

**4. Database Schema**

**4.1. Documents Table (PostgreSQL)**

sql

WrapCopy

CREATE TABLE documents (

id BIGINT PRIMARY KEY,

content TEXT NOT NULL,

workflow\_id VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**4.2. ImpactAssessments Table**

sql

WrapCopy

CREATE TABLE impact\_assessments (

id BIGINT PRIMARY KEY,

document\_id BIGINT REFERENCES documents(id),

impacted\_systems JSONB,

baseline JSONB,

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**4.3. Requirements Table**

sql

WrapCopy

CREATE TABLE requirements (

id BIGINT PRIMARY KEY,

impact\_assessment\_id BIGINT REFERENCES impact\_assessments(id),

description TEXT NOT NULL,

status VARCHAR(50),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**4.4. GeneratedCode Table**

sql

WrapCopy

CREATE TABLE generated\_code (

id BIGINT PRIMARY KEY,

class\_name VARCHAR(255),

code TEXT NOT NULL,

test\_cases TEXT NOT NULL,

application\_stack VARCHAR(255),

created\_at TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

**5. AI Model Workflow (DART AI and Gen AI)**

* **DART AI**:
  + **Input**: Historical documents, manual inputs, impacted systems.
  + **Process**: Use natural language processing (NLP) and machine learning to analyze data, identify patterns, and establish baselines.
  + **Output**: DataAnalysisResult (impacted areas, baseline data).
* **Gen AI**:
  + **Input**: Impact assessments, workflow descriptions, impacted code/classes.
  + **Process**: Use generative models (e.g., transformers) to generate requirements, code, and test cases based on prompts.
  + **Output**: Requirements (text), Code (Java/JavaScript), Test Cases (JUnit/Selenium scripts).

**6. Git Integration**

* **Process**:
  1. Use Git API (e.g., GitHub API) to fetch repository data.
  2. Identify impacted classes/files using commit history and diffs.
  3. Pass impacted code to Gen AI for code/test case generation.
  4. Store generated outputs in the data layer and push updates to Git.

**7. Testing Framework**

* **Unit Tests**: Use JUnit and Mockito to test backend services (e.g., ImpactAssessmentService, RequirementGenerator).
* **Integration Tests**: Use Spring Boot Test to validate API endpoints and AI integrations.
* **UI Tests**: Use Selenium to test frontend interactions.
* **AI Validation**: Compare AI-generated outputs (requirements, code, test cases) with SME/BA feedback for accuracy.

**8. Deployment Details**

* **Containerization**: Dockerize the Spring Boot application, frontend, and AI models.
* **Orchestration**: Use Kubernetes for scaling and managing containers.
* **CI/CD**: Automate builds, tests, and deployments using Jenkins/GitHub Actions.

Below, I’ll provide a detailed **Sequence Diagram**, **Flow Diagram**, and **Rules of Engagement (ROE)** for the Cash and Electronic Trading system, as outlined in the previous HLD and LLD. These artifacts will help visualize the interactions, processes, and operational guidelines for the system using Java Spring Boot, DART AI, and Generative AI (Gen AI).

**1. Sequence Diagram**

A sequence diagram illustrates the interaction between components (actors, frontend, backend, AI models, etc.) during a typical use-case, such as generating an impact assessment, requirements, and code/test cases for the Equities Cash workflow.

**Diagram Description (Text-Based Representation)**

**Actors and Components**:

* **SME/BA (User)**: Subject Matter Expert or Business Analyst.
* **Frontend**: User interface (React.js/Angular).
* **Backend**: Java Spring Boot services (e.g., ImpactAssessmentService, RequirementGenerator).
* **DART AI**: Data analysis and reporting tool for baseline establishment.
* **Gen AI**: Generative AI for requirement, code, and test case generation.
* **Data Layer**: Databases (PostgreSQL/MySQL, MongoDB).
* **Git**: Version control system for identifying impacted classes.

**Scenario**: Generating an impact assessment, requirements, and code/test cases for a new regulatory task in the Equities Cash workflow.

text

WrapCopy

@startuml

actor "SME/BA" as User

participant "Frontend" as FE

participant "Backend" as BE

participant "DART AI" as DART

participant "Gen AI" as GEN

participant "Data Layer" as DB

participant "Git" as GIT

loop For each new regulatory task

User -> FE: Input historical docs & manual data

FE -> BE: POST /api/impact/assessment (workflowId, data)

BE -> DART: analyze(input, impactedSystems)

DART --> BE: DataAnalysisResult (impacted areas, baseline)

BE -> DB: Store ImpactAssessment

DB --> BE: Confirmation

BE -> GEN: generateRequirements(impact, workflow)

GEN --> BE: List<Requirement>

BE -> DB: Store Requirements

DB --> BE: Confirmation

BE -> GIT: getImpactedClasses(workflow)

GIT --> BE: List<ImpactedClass>

BE -> GEN: generateCodeTestCases(impactedClass, stack)

GEN --> BE: GeneratedOutput (code, testCases)

BE -> DB: Store GeneratedCode

DB --> BE: Confirmation

BE -> FE: Return JSON (impact, requirements, code, testCases)

FE -> User: Display results

end

@enduml

**Notes**:

* This diagram uses PlantUML syntax for clarity. You can generate a visual diagram using PlantUML tools or online editors.
* The loop represents iterative processing for multiple regulatory tasks.
* The backend orchestrates interactions between the frontend, AI models, data layer, and Git.

**2. Flow Diagram**

A flow diagram (or data flow diagram) illustrates the overall process flow for the system, showing how data moves between components to fulfill the use-case requirements.

**Diagram Description (Text-Based Representation)**

**Components**:

* **Input**: Historical documentation, manual inputs, Git repository data.
* **Processing**: Backend services, DART AI, Gen AI.
* **Output**: Impact assessments, requirements, code, test cases.
* **Storage**: Data layer (databases).

text

WrapCopy

@startuml

(\*) --> "SME/BA Inputs\n(Historical Docs, Manual Data)"

--> "Frontend\n(React.js/Angular)"

--> "Backend\n(Java Spring Boot)"

--> "DART AI\n(Analyze Data, Set Baseline)"

--> "Gen AI\n(Generate Requirements, Code, Test Cases)"

--> "Data Layer\n(Store Results)"

--> "Git\n(Identify Impacted Classes)"

--> "Backend\n(Process & Integrate)"

--> "Frontend\n(Display Results)"

--> (\*)

note right of "DART AI"

Analyzes historical data to avoid

stale data reuse

end note

note right of "Gen AI"

Generates 90% of requirements,

code, and test cases

end note

@enduml

**Notes**:

* This flow diagram uses PlantUML syntax for clarity. Use a PlantUML tool to visualize it.
* Arrows represent data flow, and notes provide additional context for AI model roles.

**3. Detailed Rules of Engagement (ROE)**

The Rules of Engagement (ROE) define the operational guidelines, responsibilities, and constraints for stakeholders (e.g., developers, SMEs, BAs, AI engineers) during the development, deployment, and maintenance of the Cash and Electronic Trading system.

**3.1. Purpose**

* Ensure clear communication, accountability, and alignment among team members and stakeholders.
* Minimize risks, ensure quality, and achieve project goals (e.g., 90% automation of requirements, reduced SME resourcing).

**3.2. Scope**

* Applies to all phases of the project (planning, design, development, testing, deployment, maintenance).
* Covers interactions between SMEs/BAs, developers, AI engineers, and DevOps teams.

**3.3. Roles and Responsibilities**

* **SMEs/BAs**:
  + Provide historical documentation, manual inputs, and validate AI-generated outputs (impact assessments, requirements, code, test cases).
  + Conduct final reviews (e.g., BA last look for requirements) to ensure accuracy.
  + Report issues or discrepancies to the project manager or developers.
* **Developers (Java, Frontend)**:
  + Develop and maintain the Java Spring Boot backend, frontend (React.js/Angular), and integrations with AI models, data layer, and Git.
  + Ensure RESTful APIs, database schemas, and code quality meet standards (e.g., JUnit, Mockito).
  + Address bugs and implement feedback from SMEs/BAs.
* **AI Engineers**:
  + Design, train, and maintain DART AI and Gen AI models for data analysis, baseline establishment, and generation tasks.
  + Validate model accuracy (e.g., 90% requirement automation) and retrain models with new data.
  + Collaborate with developers to integrate AI outputs into the backend.
* **DevOps Engineers**:
  + Set up and maintain Git repositories, CI/CD pipelines (Jenkins/GitHub Actions), Docker containers, and Kubernetes orchestration.
  + Ensure system scalability, reliability, and deployment readiness.
  + Monitor system performance and address deployment issues.
* **Project Manager**:
  + Oversee timeline, milestones, and resource allocation as per the project roadmap.
  + Facilitate communication between teams, manage risks, and ensure project goals are met.

**3.4. Communication Protocols**

* **Daily Standups**: Conduct daily 15-minute standups (via Zoom or Slack) to discuss progress, blockers, and action items.
* **Weekly Syncs**: Hold weekly meetings with all stakeholders to review milestones, deliverables, and feedback.
* **Issue Reporting**: Use a ticketing system (e.g., Jira) for reporting bugs, feature requests, or AI validation issues. Response time: 24 hours for critical issues, 48 hours for non-critical.
* **Documentation**: Maintain shared documentation (Confluence or Google Docs) for requirements, designs (HLD, LLD), and AI model outputs.

**3.5. Constraints and Guidelines**

* **Data Privacy**: Ensure all historical documentation and manual inputs comply with data privacy regulations (e.g., GDPR, CCPA). Use encryption (SSL/TLS) and role-based access control (Spring Security).
* **AI Accuracy**: AI-generated outputs (requirements, code, test cases) must achieve at least 90% accuracy, validated by SMEs/BAs. Retrain models if accuracy falls below this threshold.
* **Code Quality**: All generated and manually written code must pass unit tests (JUnit), integration tests (Spring Boot Test), and adhere to Java/JavaScript best practices.
* **Version Control**: Use Git branching strategies (e.g., Gitflow) for development, testing, and production deployments. No direct commits to the main branch without review.
* **Deployment**: Deploy updates only after passing all tests and receiving SME/BA approval. Use rollback mechanisms in case of deployment failures.

**3.6. Escalation Path**

* **Level 1 (Team Leads)**: Resolve minor issues within 24 hours. Contact: Team Leads (developers, AI engineers, DevOps).
* **Level 2 (Project Manager)**: Handle moderate issues (e.g., delays, stakeholder conflicts) within 48 hours. Contact: Project Manager.
* **Level 3 (Stakeholder Committee)**: Address major issues (e.g., project risks, budget overruns) within 1 week. Contact: Senior Management or Client Representatives.

**3.7. Change Management**

* Any changes to requirements, scope, or timeline must be documented, approved by the Project Manager, and communicated to all stakeholders within 48 hours.
* Use a change request form (in Jira or Confluence) to track and prioritize changes.

**3.8. Success Metrics**

* Achieve 90% automation of requirements generation within 6 months (by August 24, 2025).
* Reduce SME resourcing by 50% within the first year (by February 24, 2026).
* Ensure 99% system uptime and 99% accuracy in identifying impacted classes/code.

**3.9. Termination Conditions**

* The ROE may be revised or terminated if:
  + Project goals are unachievable due to resource constraints or technical limitations.
  + Stakeholders agree to pivot the project scope or cancel it entirely.
  + Major breaches of data privacy or security occur, requiring a system overhaul.

**Notes**

To provide you with visual diagrams (e.g., PNG/JPG exports) for the **Sequence Diagram**, **Flow Diagram**, and additional details like **Entity-Relationship Diagrams (ERDs)** and **Class Diagrams**, I’ll describe how you can generate these visuals using PlantUML (the syntax I provided earlier) and suggest specific adjustments or additional details.

Since I’m a text-based AI and cannot directly generate or export images, I’ll guide you on how to create these visuals using PlantUML tools and provide refined versions of the diagrams with specific adjustments. I’ll also include additional details like ERDs and class diagrams in text-based PlantUML format for you to visualize.

**1. Generating Visual Diagrams (PNG/JPG Exports)**

**Tools to Use**

* **PlantUML Online Editor**: Visit <https://www.plantuml.com/plantuml/> to paste the PlantUML code I provided and generate PNG/JPG exports.
* **PlantUML Desktop App**: Download and install the PlantUML plugin for IntelliJ IDEA, VS Code, or use the standalone PlantUML jar to generate images.
* **Integration with Tools**: If you use tools like Confluence, Jira, or draw.io, you can embed PlantUML scripts to generate diagrams.

**Steps to Generate Visuals**

1. Copy the PlantUML code for each diagram (Sequence Diagram and Flow Diagram) from my previous response.
2. Paste the code into the PlantUML online editor or your preferred tool.
3. Click “Generate” or “Render” to produce a PNG or JPG export.
4. Download the image for use in your documentation or presentation.

**2. Adjusted Sequence Diagram (Refined for Clarity)**

Here’s an updated and more detailed version of the Sequence Diagram with specific adjustments for better clarity and additional interactions (e.g., error handling, SME validation). Use this in PlantUML to generate a visual.

plantuml

WrapCopy

@startuml

actor "SME/BA" as User

participant "Frontend" as FE

participant "Backend" as BE

participant "DART AI" as DART

participant "Gen AI" as GEN

participant "Data Layer" as DB

participant "Git" as GIT

loop For each new regulatory task

User -> FE: Input historical docs & manual data

FE -> BE: POST /api/impact/assessment (workflowId, data)

BE -> DART: analyze(input, impactedSystems)

DART --> BE: DataAnalysisResult (impacted areas, baseline)

alt Success

BE -> DB: Store ImpactAssessment

DB --> BE: Confirmation

else Failure

BE --> FE: Error (Invalid Data)

FE --> User: Display Error

end

BE -> GEN: generateRequi*rements(impact, workflow)*

GEN --> BE: List<Requi*rement>*

BE -> DB: Store Requi*rements*

DB --> BE: Confirmation

BE -> GIT: getImpactedClasses(workflow)

GIT --> BE: List<ImpactedClass>

BE -> GEN: generateCodeTestCases(impactedClass, stack)

GEN --> BE: GeneratedOutput (code, testCases)

BE -> DB: Store GeneratedCode

DB --> BE: Confirmation

BE -> FE: Return JSON (impact, requi*rements, code, testCases)*

FE -> User: Display results for review

User -> FE: Provide SME/BA Validation (Approve/Reject)

FE -> BE: POST /api/validate (feedback)

BE -> DB: Update Validation Status

DB --> BE: Confirmation

BE --> FE: Success/Failure Response

FE --> User: Notify Validation Result

end

note right of User

SME/BA validates outputs to ensure

90% accuracy

end note

note right of GEN

Gen AI generates outputs based on

trained models and prompts

end note

@enduml

**Adjustments**:

* Added error handling (e.g., alt/else for failure cases).
* Included SME/BA validation step to ensure 90% accuracy.
* Added notes for clarity on SME/BA roles and Gen AI functionality.

**3. Adjusted Flow Diagram (Refined for Clarity)**

Here’s an updated Flow Diagram with specific adjustments for better flow and additional steps (e.g., validation, error handling).

plantuml

WrapCopy

@startuml

(\*) --> "SME/BA Inputs\n(Historical Docs, Manual Data)"

--> "Frontend\n(React.js/Angular)"

--> "Backend\n(Java Spring Boot)"

--> "Validate Input Data"

if "Input Valid?" then

--> "DART AI\n(Analyze Data, Set Baseline)"

--> "Gen AI\n(Generate Requirements, Code, Test Cases)"

--> "Data Layer\n(Store Results)"

--> "Git\n(Identify Impacted Classes)"

--> "Backend\n(Process & Integrate)"

--> "Frontend\n(Display Results for Review)"

--> "SME/BA Validation\n(Approve/Reject)"

if "Validation Passed?" then

--> "Data Layer\n(Update Validation)"

--> "Deploy Updates\n(via CI/CD)"

--> (\*)

else

--> "Backend\n(Retrain AI Models)"

--> "Gen AI\n(Regenerate Outputs)"

--> "Frontend\n(Display Updated Results)"

--> "SME/BA Validation\n(Approve/Reject)"

endif

else

--> "Frontend\n(Display Error)"

--> (\*)

endif

note right of "DART AI"

Analyzes historical data to avoid

stale data reuse

end note

note right of "Gen AI"

Generates 90% of requirements,

code, and test cases

end note

note right of "SME/BA Validation"

Ensures 90% accuracy of AI outputs

end note

@enduml

**Adjustments**:

* Added input validation and error handling.
* Included SME/BA validation and a feedback loop for retraining AI models if validation fails.
* Added deployment via CI/CD as a final step.

**4. Entity-Relationship Diagram (ERD)**

An ERD visualizes the database schema relationships (from the LLD). Here’s the PlantUML code for the ERD, which you can use to generate a PNG/JPG.

plantuml

WrapCopy

@startuml

entity "Documents" {

\*id: BIGINT

\*content: TEXT

\*workflow\_id: VARCHAR(255)

\*created\_at: TIMESTAMP

}

entity "ImpactAssessments" {

\*id: BIGINT

\*document\_id: BIGINT

\*impacted\_systems: JSONB

\*baseline: JSONB

\*created\_at: TIMESTAMP

}

entity "Requirements" {

\*id: BIGINT

\*impact\_assessment\_id: BIGINT

\*description: TEXT

\*status: VARCHAR(50)

\*created\_at: TIMESTAMP

}

entity "GeneratedCode" {

\*id: BIGINT

\*class\_name: VARCHAR(255)

\*code: TEXT

\*test\_cases: TEXT

\*application\_stack: VARCHAR(255)

\*created\_at: TIMESTAMP

}

Documents ||--o{ ImpactAssessments : has

ImpactAssessments ||--o{ Requi*rements : generates*

ImpactAssessments ||--o{ GeneratedCode : generates

note right of Documents

Stores historical documentation

and workflow data

end note

note right of ImpactAssessments

Links documents to assessments,

impacted systems, and baselines

end note

note right of Requi*rements*

Stores AI-generated requi*rements*

with validation status

end note

note right of GeneratedCode

Stores generated code and test

cases for impacted classes

end note

@enduml

**Notes**:

* This ERD shows one-to-many relationships (e.g., one document can have multiple impact assessments).
* Use PlantUML to generate a visual representation (PNG/JPG).

**5. Class Diagram (Detailed)**

A class diagram provides a detailed view of the object-oriented design (from the LLD). Here’s the PlantUML code for the key classes.

plantuml

WrapCopy

@startuml

class ImpactAssessmentService {

-DARTAIClient dartAIClient

-DocumentRepository documentRepo

+ImpactAssessment generateImpactAssessment(Document, List<String>)

}

class Requi*rementGenerator {*

-GenAIClient genAIClient

+List<Requi*rement> generateRequirements(ImpactAssessment, Workflow)*

}

class CodeTestCaseGenerator {

-GenAIClient genAIClient

-GitClient gitClient

+GeneratedOutput generateCodeTestCases(String, ApplicationStack)

}

class WorkflowManager {

-BaselineRepository baselineRepo

+void updateBaseline(Release)

}

class Document {

-String content

-String workflowId

-Timestamp createdAt

+getContent(): String

+setContent(String): void

}

class ImpactAssessment {

-List<String> impactedAreas

-JSONObject baseline

+getImpactedAreas(): List<String>

+setBaseline(JSONObject): void

}

class Requi*rement {*

-String description

-String status

-Timestamp createdAt

+getDescription(): String

+setStatus(String): void

}

class GeneratedOutput {

-String code

-String testCases

+getCode(): String

+getTestCases(): String

}

class Release {

-JSONObject data

-Timestamp timestamp

+getData(): JSONObject

+setTimestamp(Timestamp): void

}

ImpactAssessmentService o--> DARTAIClient

ImpactAssessmentService o--> DocumentRepository

Requi*rementGenerator o--> GenAIClient*

CodeTestCaseGenerator o--> GenAIClient

CodeTestCaseGenerator o--> GitClient

WorkflowManager o--> BaselineRepository

Document o--> "1" ImpactAssessment

ImpactAssessment o--> "0..\*" Requi*rement*

ImpactAssessment o--> "0..\*" GeneratedOutput

WorkflowManager o--> Release

note right of ImpactAssessmentService

Orchestrates impact assessment

using DART AI

end note

note right of Requi*rementGenerator*

Generates requi*rements using*

Gen AI

end note

note right of CodeTestCaseGenerator

Generates code and test cases

for impacted classes

end note

@enduml

**Notes**:

* This class diagram shows relationships (associations, compositions) between key classes.
* Use PlantUML to generate a visual representation (PNG/JPG).

**6. Additional Details and Adjustments**

**Specific Adjustments**

* **Sequence Diagram**: Added error handling and SME/BA validation to reflect real-world scenarios.
* **Flow Diagram**: Included a feedback loop for AI retraining and deployment steps for completeness.
* **ERD**: Clarified relationships (e.g., one-to-many) and added notes for context.
* **Class Diagram**: Expanded to include more attributes and methods for clarity and implementation readiness.

**Additional Details**

* **API Documentation**: I can provide detailed Swagger/OpenAPI specifications for the RESTful endpoints if needed.
* **AI Model Training Details**: I can describe the training data, algorithms (e.g., transformers for Gen AI), and validation metrics for DART AI and Gen AI.
* **Deployment Configuration**: I can detail Docker/Kubernetes configurations, CI/CD pipeline scripts, or cloud setup (e.g., AWS, Azure).

**How to Export Visuals**

1. Copy each PlantUML code block into the PlantUML online editor or your tool of choice.
2. Select “PNG” or “JPG” as the output format.
3. Download the exported image and use it in your project documentation or presentations.

If you need further refinements (e.g., specific colors, layouts, or additional components), or if you want me to provide more detailed text-based outputs (e.g., JSON schemas, code snippets), please specify, and I’ll adjust accordingly!

Below is a detailed code snippet in Java (using Spring Boot) that demonstrates how to integrate with JIRA as a project planning and tracking tool, fetch commit information per JIRA ticket, and use that data to identify impacted systems and classes for the Cash and Electronic Trading system. This example assumes you’re using the JIRA REST API, Git integration (via JGit or GitHub API), and DART AI/Gen AI for analysis, as outlined in the previous requirements.

The code includes input handling, API calls to JIRA, Git commit analysis, and output generation to identify impacted systems and classes. I’ll also include comments for clarity and explain the input/output structure.

**Code Snippet: JIRA Integration for Project Planning and Impact Analysis**

**Prerequisites**

* **Dependencies**: Add jira-rest-java-client, jgit, and spring-boot-starter-web to your pom.xml:

xml

WrapCopy

<dependencies>

<dependency>

<groupId>com.atlassian.jira</groupId>

<artifactId>jira-rest-java-client-api</artifactId>

<version>5.2.5</version>

</dependency>

<dependency>

<groupId>org.eclipse.jgit</groupId>

<artifactId>org.eclipse.jgit</artifactId>

<version>6.5.0.202303070854-r</version>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

</dependencies>

* **Configuration**: Set up JIRA credentials, Git repository path, and API endpoints in application.properties:

properties

WrapCopy

jira.url=https://your-jira-instance.atlassian.net

jira.username=your-username

jira.api.token=your-api-token

git.repo.path=/path/to/your/git/repo

**Java Code (Spring Boot Controller, Service, and Models)**

java

WrapCopy

import com.atlassian.jira.rest.client.api.JiraRestClient;

import com.atlassian.jira.rest.client.api.domain.Issue;

import com.atlassian.jira.rest.client.internal.async.AsynchronousJiraRestClientFactory;

import org.eclipse.jgit.api.Git;

import org.eclipse.jgit.lib.Repository;

import org.eclipse.jgit.revwalk.RevCommit;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.\*;

import java.util.stream.Collectors;

*// Model for Impacted Systems and Classes*

class ImpactedAnalysis {

private List<String> impactedSystems;

private List<String> impactedClasses;

public ImpactedAnalysis(List<String> impactedSystems, List<String> impactedClasses) {

this.impactedSystems = impactedSystems;

this.impactedClasses = impactedClasses;

}

*// Getters and setters*

public List<String> getImpactedSystems() { return impactedSystems; }

public List<String> getImpactedClasses() { return impactedClasses; }

}

*// Service to handle JIRA, Git, and Impact Analysis*

@Service

class ImpactAnalysisService {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

@Value("${git.repo.path}")

private String gitRepoPath;

public ImpactedAnalysis analyzeImpactsFromJira(String jiraTicket) {

try {

*// Step 1: Connect to JIRA and fetch ticket details*

JiraRestClient jiraClient = new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

Issue issue = jiraClient.getIssueClient().getIssue(jiraTicket).claim();

*// Step 2: Extract commit references from JIRA ticket (assuming commits are linked in comments or custom fields)*

String commitRefs = extractCommitRefs(issue);

List<String> commits = Arrays.asList(commitRefs.split(","));

*// Step 3: Analyze Git commits to identify impacted files/classes*

List<String> impactedClasses = analyzeGitCommits(commits);

List<String> impactedSystems = inferImpactedSystems(impactedClasses);

return new ImpactedAnalysis(impactedSystems, impactedClasses);

} catch (Exception e) {

throw new RuntimeException("Error analyzing JIRA ticket and Git commits: " + e.getMessage());

}

}

private String extractCommitRefs(Issue issue) {

*// Simulate extracting commit hashes from JIRA ticket comments or custom fields*

*// In reality, this would parse JIRA comments or a custom field for Git commit hashes*

return issue.getComments().stream()

.filter(comment -> comment.getBody().contains("git commit"))

.map(comment -> comment.getBody().split("git commit ")[1].split("\n")[0])

.collect(Collectors.joining(","));

}

private List<String> analyzeGitCommits(List<String> commitHashes) {

List<String> impactedClasses = new ArrayList<>();

try (Repository repo = Git.open(new java.io.File(gitRepoPath)).getRepository()) {

Git git = new Git(repo);

for (String commitHash : commitHashes) {

RevCommit commit = git.log().add(repo.resolve(commitHash)).call().iterator().next();

*// Analyze commit diffs to find changed files/classes*

git.diff()

.setOldTree(git.getRepository().resolve(commit.getParent(0).getName()))

.setNewTree(git.getRepository().resolve(commit.getName()))

.call()

.forEach(diff -> {

String filePath = diff.getNewPath();

if (filePath.endsWith(".java")) { *// Assume Java files for classes*

String className = filePath.substring(filePath.lastIndexOf("/") + 1, filePath.lastIndexOf(".java"));

impactedClasses.add(className);

}

});

}

} catch (Exception e) {

throw new RuntimeException("Error analyzing Git commits: " + e.getMessage());

}

return impactedClasses.stream().distinct().collect(Collectors.toList());

}

private List<String> inferImpactedSystems(List<String> impactedClasses) {

*// Simulate inferring impacted systems based on class names or package structure*

*// This could use DART AI or a rule-based approach to map classes to systems (e.g., Equities Cash)*

List<String> systems = new ArrayList<>();

for (String className : impactedClasses) {

if (className.contains("Equity") || className.contains("Cash")) {

systems.add("Equities Cash System");

} else if (className.contains("Trading")) {

systems.add("Electronic Trading System");

}

}

return systems.stream().distinct().collect(Collectors.toList());

}

}

*// Controller to handle HTTP requests*

@RestController

@RequestMapping("/api/impact")

public class ImpactAnalysisController {

private final ImpactAnalysisService impactService;

public ImpactAnalysisController(ImpactAnalysisService impactService) {

this.impactService = impactService;

}

@PostMapping("/analyze-jira")

public ResponseEntity<ImpactedAnalysis> analyzeJiraImpact(@RequestBody String jiraTicket) {

ImpactedAnalysis analysis = impactService.analyzeImpactsFromJira(jiraTicket);

return ResponseEntity.ok(analysis);

}

}

*// Main Application Class*

@SpringBootApplication

public class TradingSystemApplication {

public static void main(String[] args) {

SpringApplication.run(TradingSystemApplication.class, args);

}

}

**Input and Output Structure**

**Input**

* **HTTP Request (POST /api/impact/analyze-jira)**:
  + **Body**: JSON containing the JIRA ticket ID (e.g., "PROJ-123").
  + Example:

json

WrapCopy

{

"jiraTicket": "PROJ-123"

}

**Output**

* **HTTP Response (200 OK)**:
  + **Body**: JSON containing the ImpactedAnalysis object with lists of impacted systems and classes.
  + Example:

json

WrapCopy

{

"impactedSystems": ["Equities Cash System", "Electronic Trading System"],

"impactedClasses": ["EquityTradeProcessor", "CashTransactionHandler", "TradingEngine"]

}

**Error Handling**

* **400 Bad Request**: If the JIRA ticket is invalid or missing.
* **500 Internal Server Error**: If there’s an issue connecting to JIRA, Git, or processing data.

**Explanation of the Code**

1. **JIRA Integration**:
   * Uses the jira-rest-java-client library to connect to JIRA via REST API with basic authentication (username and API token).
   * Fetches the JIRA ticket details (e.g., PROJ-123) and extracts commit references from comments or custom fields. In a real scenario, you’d need to configure JIRA to link commits (e.g., via commit messages or custom fields).
2. **Git Analysis**:
   * Uses JGit to interact with the Git repository, analyzing commits to identify changed files (e.g., .java files for classes).
   * Maps file paths to class names and tracks impacted classes.
3. **Impact Analysis**:
   * Infers impacted systems (e.g., “Equities Cash System”) based on class names or package structures. This could be enhanced with DART AI for more accurate system mapping.
   * Returns a structured ImpactedAnalysis object for use in the system (e.g., for Gen AI to generate requirements, code, and test cases).
4. **Spring Boot Integration**:
   * Exposes a REST endpoint (/api/impact/analyze-jira) to handle requests.
   * Uses dependency injection (ImpactAnalysisService) for modularity and testing.

**Using JIRA for Project Planning**

**JIRA Configuration**

* **Project**: Create a JIRA project (e.g., “TradingSystem”) to track tasks, bugs, and features for the Cash and Electronic Trading system.
* **Epics and Stories**: Use epics for major phases (e.g., “Requirement Analysis,” “Development”) and stories for specific tasks (e.g., “Integrate JIRA with Git,” “Analyze Impacted Classes”).
* **Custom Fields**: Add a custom field (e.g., “Git Commit Hash”) to link JIRA tickets with Git commits.
* **Workflows**: Define a workflow (e.g., “To Do → In Progress → Done”) to track task progress.

**JIRA Queries for Planning**

* Use JIRA Query Language (JQL) to monitor progress:
  + project = TradingSystem AND status = "In Progress" ORDER BY priority
  + project = TradingSystem AND labels = "ImpactAnalysis" AND resolution = Unresolved

**Tracking Commits Per JIRA**

* Configure Git hooks or CI/CD pipelines (e.g., Jenkins/GitHub Actions) to associate commits with JIRA tickets (e.g., commit message format: “PROJ-123: Update EquityTradeProcessor”).
* Use the JIRA API or plugins (e.g., JIRA Git Integration Plugin) to automatically link commits to tickets.

**Notes**

* This code is a simplified example. In a production environment, you’d need to handle:
  + Authentication securely (e.g., OAuth instead of basic auth).
  + Rate limiting and error handling for JIRA and Git APIs.
  + Scalability (e.g., batch processing for large repositories).
  + Integration with DART AI/Gen AI for deeper analysis (e.g., using AI to refine impacted system inference).
* To enhance this, you could:
  + Add caching (e.g., Spring Cache) for JIRA and Git data.
  + Use a message queue (e.g., Kafka) for asynchronous processing of large datasets.
  + Implement retry mechanisms for API failures.

enhanced version of the previous Java (Spring Boot) code snippet that extends the functionality to automatically create JIRA tickets for the impacted systems identified during the impact analysis. This version builds on the ImpactAnalysisService and adds a JiraTicketCreator component to create JIRA tickets programmatically using the JIRA REST API. I’ll include input/output structures, error handling, and comments for clarity.

**Updated Code Snippet: Automatic JIRA Ticket Creation for Impacted Systems**

**Prerequisites**

* Ensure you have the same dependencies as before (jira-rest-java-client, jgit, spring-boot-starter-web) in your pom.xml.
* Update application.properties with JIRA credentials and project details:

properties

WrapCopy

jira.url=https://your-jira-instance.atlassian.net

jira.username=your-username

jira.api.token=your-api-token

jira.project.key=TRAD # JIRA project key (e.g., "TRAD" for Trading System)

git.repo.path=/path/to/your/git/repo

**Java Code (Spring Boot Controller, Service, and Models)**

java

WrapCopy

import com.atlassian.jira.rest.client.api.JiraRestClient;

import com.atlassian.jira.rest.client.api.domain.BasicProject;

import com.atlassian.jira.rest.client.api.domain.Issue;

import com.atlassian.jira.rest.client.api.domain.IssueType;

import com.atlassian.jira.rest.client.internal.async.AsynchronousJiraRestClientFactory;

import com.atlassian.jira.rest.client.api.domain.input.IssueInput;

import com.atlassian.jira.rest.client.api.domain.input.FieldInput;

import org.eclipse.jgit.api.Git;

import org.eclipse.jgit.lib.Repository;

import org.eclipse.jgit.revwalk.RevCommit;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import java.net.URI;

import java.util.\*;

import java.util.stream.Collectors;

*// Model for Impacted Systems and Classes*

class ImpactedAnalysis {

private List<String> impactedSystems;

private List<String> impactedClasses;

public ImpactedAnalysis(List<String> impactedSystems, List<String> impactedClasses) {

this.impactedSystems = impactedSystems;

this.impactedClasses = impactedClasses;

}

*// Getters and setters*

public List<String> getImpactedSystems() { return impactedSystems; }

public List<String> getImpactedClasses() { return impactedClasses; }

}

*// Model for JIRA Ticket Creation Response*

class JiraTicketResponse {

private String ticketId;

private String ticketUrl;

private String status;

public JiraTicketResponse(String ticketId, String ticketUrl, String status) {

this.ticketId = ticketId;

this.ticketUrl = ticketUrl;

this.status = status;

}

*// Getters and setters*

public String getTicketId() { return ticketId; }

public String getTicketUrl() { return ticketUrl; }

public String getStatus() { return status; }

}

*// Service to handle JIRA, Git, Impact Analysis, and Ticket Creation*

@Service

class ImpactAnalysisService {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

@Value("${jira.project.key}")

private String jiraProjectKey;

@Value("${git.repo.path}")

private String gitRepoPath;

private final JiraTicketCreator jiraTicketCreator;

public ImpactAnalysisService(JiraTicketCreator jiraTicketCreator) {

this.jiraTicketCreator = jiraTicketCreator;

}

public ImpactedAnalysis analyzeImpactsFromJira(String jiraTicket) {

try {

*// Step 1: Connect to JIRA and fetch ticket details*

JiraRestClient jiraClient = new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

Issue issue = jiraClient.getIssueClient().getIssue(jiraTicket).claim();

*// Step 2: Extract commit references from JIRA ticket*

String commitRefs = extractCommitRefs(issue);

List<String> commits = Arrays.asList(commitRefs.split(","));

*// Step 3: Analyze Git commits to identify impacted files/classes*

List<String> impactedClasses = analyzeGitCommits(commits);

List<String> impactedSystems = inferImpactedSystems(impactedClasses);

*// Step 4: Automatically create JIRA tickets for impacted systems*

List<JiraTicketResponse> createdTickets = createJiraTicketsForImpactedSystems(impactedSystems, jiraClient);

return new ImpactedAnalysis(impactedSystems, impactedClasses);

} catch (Exception e) {

throw new RuntimeException("Error analyzing JIRA ticket and Git commits: " + e.getMessage());

}

}

private String extractCommitRefs(Issue issue) {

*// Simulate extracting commit hashes from JIRA ticket comments or custom fields*

return issue.getComments().stream()

.filter(comment -> comment.getBody().contains("git commit"))

.map(comment -> comment.getBody().split("git commit ")[1].split("\n")[0])

.collect(Collectors.joining(","));

}

private List<String> analyzeGitCommits(List<String> commitHashes) {

List<String> impactedClasses = new ArrayList<>();

try (Repository repo = Git.open(new java.io.File(gitRepoPath)).getRepository()) {

Git git = new Git(repo);

for (String commitHash : commitHashes) {

RevCommit commit = git.log().add(repo.resolve(commitHash)).call().iterator().next();

git.diff()

.setOldTree(git.getRepository().resolve(commit.getParent(0).getName()))

.setNewTree(git.getRepository().resolve(commit.getName()))

.call()

.forEach(diff -> {

String filePath = diff.getNewPath();

if (filePath.endsWith(".java")) {

String className = filePath.substring(filePath.lastIndexOf("/") + 1, filePath.lastIndexOf(".java"));

impactedClasses.add(className);

}

});

}

} catch (Exception e) {

throw new RuntimeException("Error analyzing Git commits: " + e.getMessage());

}

return impactedClasses.stream().distinct().collect(Collectors.toList());

}

private List<String> inferImpactedSystems(List<String> impactedClasses) {

List<String> systems = new ArrayList<>();

for (String className : impactedClasses) {

if (className.contains("Equity") || className.contains("Cash")) {

systems.add("Equities Cash System");

} else if (className.contains("Trading")) {

systems.add("Electronic Trading System");

}

}

return systems.stream().distinct().collect(Collectors.toList());

}

private List<JiraTicketResponse> createJiraTicketsForImpactedSystems(List<String> impactedSystems, JiraRestClient jiraClient) {

List<JiraTicketResponse> tickets = new ArrayList<>();

for (String system : impactedSystems) {

try {

*// Define issue type (e.g., "Task" or "Bug")*

IssueType issueType = jiraClient.getMetadataClient().getIssueTypes().claim()

.stream().filter(it -> "Task".equals(it.getName())).findFirst()

.orElseThrow(() -> new RuntimeException("Task issue type not found"));

*// Create issue input for JIRA*

IssueInput issueInput = new IssueInput(

jiraProjectKey,

issueType.getId(),

"Impact Analysis for " + system,

"Automatically generated ticket for impacted system: " + system + ". Needs review and updates due to recent changes."

);

*// Add custom fields if needed (e.g., priority, labels)*

issueInput.addFieldInput(new FieldInput("priority", "Medium")); *// Example priority*

issueInput.addFieldInput(new FieldInput("labels", Arrays.asList("ImpactAnalysis", "Automation")));

*// Create the issue in JIRA*

Issue newIssue = jiraClient.getIssueClient().createIssue(issueInput).claim();

String ticketUrl = jiraUrl + "/browse/" + newIssue.getKey();

tickets.add(new JiraTicketResponse(newIssue.getKey(), ticketUrl, "Created"));

} catch (Exception e) {

tickets.add(new JiraTicketResponse(null, null, "Failed: " + e.getMessage()));

}

}

return tickets;

}

}

*// Service to encapsulate JIRA ticket creation logic*

@Service

class JiraTicketCreator {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

public JiraRestClient getJiraClient() {

return new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

}

}

*// Controller to handle HTTP requests*

@RestController

@RequestMapping("/api/impact")

public class ImpactAnalysisController {

private final ImpactAnalysisService impactService;

public ImpactAnalysisController(ImpactAnalysisService impactService) {

this.impactService = impactService;

}

@PostMapping("/analyze-jira")

public ResponseEntity<ImpactedAnalysis> analyzeJiraImpact(@RequestBody String jiraTicket) {

ImpactedAnalysis analysis = impactService.analyzeImpactsFromJira(jiraTicket);

return ResponseEntity.ok(analysis);

}

@GetMapping("/created-tickets")

public ResponseEntity<List<JiraTicketResponse>> getCreatedTickets() {

*// This is a placeholder; in a real implementation, you'd track and return created tickets*

*// For this example, return an empty list (you'd need to store tickets in a service or DB)*

return ResponseEntity.ok(new ArrayList<>());

}

}

*// Main Application Class*

@SpringBootApplication

public class TradingSystemApplication {

public static void main(String[] args) {

SpringApplication.run(TradingSystemApplication.class, args);

}

}

**Input and Output Structure**

**Input**

* **HTTP Request (POST /api/impact/analyze-jira)**:
  + **Body**: JSON containing the JIRA ticket ID (e.g., "PROJ-123").
  + Example:

json

WrapCopy

{

"jiraTicket": "PROJ-123"

}

**Output**

* **HTTP Response (200 OK)**:
  + **Body**: JSON containing the ImpactedAnalysis object with lists of impacted systems and classes.
  + Example:

json

WrapCopy

{

"impactedSystems": ["Equities Cash System", "Electronic Trading System"],

"impactedClasses": ["EquityTradeProcessor", "CashTransactionHandler", "TradingEngine"]

}

* **Created JIRA Tickets**:
  + Automatically created JIRA tickets for each impacted system (e.g., “Equities Cash System,” “Electronic Trading System”).
  + Each ticket includes:
    - **Project**: TRAD (or your JIRA project key).
    - **Issue Type**: “Task” (configurable).
    - **Summary**: “Impact Analysis for [System Name]”.
    - **Description**: “Automatically generated ticket for impacted system: [System Name]. Needs review and updates due to recent changes.”
    - **Priority**: “Medium” (configurable).
    - **Labels**: “ImpactAnalysis, Automation” (configurable).
  + **Response (via GET /api/impact/created-tickets)**:
    - Example:

json

WrapCopy

[

{

"ticketId": "TRAD-456",

"ticketUrl": "https://your-jira-instance.atlassian.net/browse/TRAD-456",

"status": "Created"

},

{

"ticketId": "TRAD-457",

"ticketUrl": "https://your-jira-instance.atlassian.net/browse/TRAD-457",

"status": "Created"

}

]

**Error Handling**

* **400 Bad Request**: If the JIRA ticket is invalid or missing.
* **500 Internal Server Error**: If there’s an issue connecting to JIRA, Git, or creating tickets (e.g., authentication failure, rate limits).

**Explanation of the Code**

1. **JIRA Integration**:
   * Extends the previous ImpactAnalysisService to fetch JIRA ticket details and extract commit references as before.
   * Introduces JiraTicketCreator and createJiraTicketsForImpactedSystems to programmatically create JIRA tickets for each impacted system.
2. **Automatic Ticket Creation**:
   * Uses the JIRA REST API to create IssueInput objects with predefined fields (project key, issue type, summary, description, priority, labels).
   * Handles errors (e.g., if the project or issue type doesn’t exist) and returns a JiraTicketResponse with the ticket ID, URL, and status.
3. **Git and Impact Analysis**:
   * Retains the Git commit analysis and impacted system inference from the previous code, now triggering JIRA ticket creation post-analysis.
4. **Spring Boot Integration**:
   * Adds a new endpoint (GET /api/impact/created-tickets) to retrieve created tickets (though this is a placeholder; you’d typically store tickets in a database or service for tracking).
   * Uses dependency injection for JiraTicketCreator and ImpactAnalysisService for modularity.

**Using JIRA for Automatic Ticket Creation**

**JIRA Configuration**

* **Project Key**: Ensure the jira.project.key (e.g., TRAD) is set in application.properties and matches your JIRA project.
* **Issue Types**: Verify that “Task” (or your preferred issue type) exists in the JIRA project.
* **Permissions**: Ensure the JIRA user (via jira.username and jira.api.token) has permissions to create issues in the project.
* **Custom Fields**: Optionally, add custom fields (e.g., “ImpactedSystem”) to store additional metadata about impacted systems.

**JIRA Queries for Monitoring**

* Use JQL to track automatically created tickets:
  + project = TRAD AND labels = "ImpactAnalysis" AND status = "Open" ORDER BY created DESC
  + project = TRAD AND description ~ "Automatically generated ticket" AND created >= startOfDay()

**Notes**

* This code is a simplified example. In a production environment, you’d need to:
  + Handle authentication securely (e.g., OAuth instead of basic auth).
  + Implement rate limiting and retry logic for JIRA API calls.
  + Store created tickets in a database or service for tracking and retrieval via GET /api/impact/created-tickets.
  + Add validation for JIRA project existence, issue type availability, and user permissions.
  + Integrate with DART AI/Gen AI to refine ticket descriptions or prioritize based on impact severity.
* To enhance this, you could:
  + Add email notifications or Slack integrations for created tickets.
  + Use a queue (e.g., RabbitMQ, Kafka) for asynchronous ticket creation to handle large volumes of impacted systems.
  + Implement logging (e.g., SLF4J) to track ticket creation success/failure

To integrate the Java Spring Boot application (previously described for JIRA, Git, and impact analysis) with DART AI (a hypothetical Data Analysis and Reporting Tool for AI-driven analysis) and obtain refined and precise information about impacted systems and classes, we’ll need to enhance the system with DART AI capabilities. Since DART AI is not a specific, widely-known tool in this context, I’ll assume it’s a custom or third-party AI tool designed for data analysis, pattern recognition, and reporting, similar to tools like TensorFlow, PyTorch, or proprietary NLP/ML platforms.

Below, I’ll outline how to integrate DART AI into your Spring Boot application, refine the impact analysis process, and achieve more precise information about impacted systems and classes. I’ll provide a code snippet, explain the integration steps, and describe the input/output structure.

**Integration Approach with DART AI**

**1. Assumptions About DART AI**

* DART AI is an AI/ML tool that can analyze historical documentation, Git commit data, and JIRA ticket information to identify patterns, infer impacted systems/classes, and provide refined, precise outputs.
* It uses natural language processing (NLP), machine learning (ML), and potentially graph-based analysis to map relationships between systems, classes, and workflows (e.g., Equities Cash).
* It provides an API or library (e.g., Python-based) that can be called from Java via REST or a Java wrapper.

**2. Integration Steps**

* **Step 1**: Add DART AI as a dependency or service (e.g., REST API or Java library).
* **Step 2**: Modify the ImpactAnalysisService to use DART AI for analyzing JIRA tickets, Git commits, and historical data to refine impacted systems/classes.
* **Step 3**: Use DART AI’s outputs to create more precise JIRA tickets and improve automation (e.g., 90% accuracy for requirements, as specified).
* **Step 4**: Ensure DART AI integrates with the existing data layer (PostgreSQL/MySQL, MongoDB) and Git repositories.

**3. Technology Stack**

* **Java Spring Boot**: For backend logic and API integration.
* **DART AI**: Hypothetical AI tool with an API or library (e.g., Python-based, accessed via REST or a Java wrapper like JNA/JNI).
* **Python (Optional)**: If DART AI is Python-based, use a microservice or REST API to communicate between Java and Python.
* **Dependencies**: Add spring-boot-starter-web, jira-rest-java-client, jgit, and potentially unirest or okhttp for REST calls to DART AI.

**Updated Code Snippet: Integrating DART AI**

Here’s an updated version of the ImpactAnalysisService from the previous code, integrating DART AI for refined and precise impact analysis. I’ll assume DART AI provides a REST API endpoint (e.g., http://dart-ai-service:8080/analyze) that accepts input data and returns refined outputs.

java

WrapCopy

import com.atlassian.jira.rest.client.api.JiraRestClient;

import com.atlassian.jira.rest.client.api.domain.Issue;

import com.atlassian.jira.rest.client.internal.async.AsynchronousJiraRestClientFactory;

import org.eclipse.jgit.api.Git;

import org.eclipse.jgit.lib.Repository;

import org.eclipse.jgit.revwalk.RevCommit;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.\*;

import org.springframework.web.client.RestTemplate;

import java.net.URI;

import java.util.\*;

import java.util.stream.Collectors;

*// Model for Impacted Systems and Classes (unchanged)*

class ImpactedAnalysis {

private List<String> impactedSystems;

private List<String> impactedClasses;

public ImpactedAnalysis(List<String> impactedSystems, List<String> impactedClasses) {

this.impactedSystems = impactedSystems;

this.impactedClasses = impactedClasses;

}

*// Getters and setters*

public List<String> getImpactedSystems() { return impactedSystems; }

public List<String> getImpactedClasses() { return impactedClasses; }

}

*// Model for DART AI Input/Output*

class DartAIRequest {

private String jiraTicket;

private List<String> gitCommits;

private String historicalData; *// JSON or text from documents*

*// Constructor, getters, setters*

public DartAIRequest(String jiraTicket, List<String> gitCommits, String historicalData) {

this.jiraTicket = jiraTicket;

this.gitCommits = gitCommits;

this.historicalData = historicalData;

}

}

class DartAIResponse {

private List<String> refinedImpactedSystems;

private List<String> refinedImpactedClasses;

private double confidenceScore;

*// Constructor, getters, setters*

public DartAIResponse(List<String> refinedImpactedSystems, List<String> refinedImpactedClasses, double confidenceScore) {

this.refinedImpactedSystems = refinedImpactedSystems;

this.refinedImpactedClasses = refinedImpactedClasses;

this.confidenceScore = confidenceScore;

}

}

*// Service to handle JIRA, Git, Impact Analysis, and DART AI Integration*

@Service

class ImpactAnalysisService {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

@Value("${jira.project.key}")

private String jiraProjectKey;

@Value("${git.repo.path}")

private String gitRepoPath;

@Value("${dart.ai.url}")

private String dartAiUrl; *// e.g., "http://dart-ai-service:8080/analyze"*

private final JiraTicketCreator jiraTicketCreator;

private final RestTemplate restTemplate = new RestTemplate();

public ImpactAnalysisService(JiraTicketCreator jiraTicketCreator) {

this.jiraTicketCreator = jiraTicketCreator;

}

public ImpactedAnalysis analyzeImpactsFromJira(String jiraTicket) {

try {

*// Step 1: Connect to JIRA and fetch ticket details*

JiraRestClient jiraClient = new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

Issue issue = jiraClient.getIssueClient().getIssue(jiraTicket).claim();

*// Step 2: Extract commit references from JIRA ticket*

String commitRefs = extractCommitRefs(issue);

List<String> commits = Arrays.asList(commitRefs.split(","));

*// Step 3: Fetch historical data (e.g., from Data Layer or documents)*

String historicalData = fetchHistoricalData(jiraTicket); *// Placeholder method*

*// Step 4: Analyze with DART AI for refined and precise information*

DartAIRequest dartRequest = new DartAIRequest(jiraTicket, commits, historicalData);

DartAIResponse dartResponse = callDartAI(dartRequest);

*// Step 5: Validate DART AI output (ensure confidenceScore >= 0.9 for 90% accuracy)*

if (dartResponse.getConfidenceScore() < 0.9) {

throw new RuntimeException("DART AI confidence too low: " + dartResponse.getConfidenceScore());

}

*// Step 6: Automatically create JIRA tickets for refined impacted systems*

List<JiraTicketResponse> createdTickets = createJiraTicketsForImpactedSystems(dartResponse.getRefinedImpactedSystems(), jiraClient);

return new ImpactedAnalysis(dartResponse.getRefinedImpactedSystems(), dartResponse.getRefinedImpactedClasses());

} catch (Exception e) {

throw new RuntimeException("Error analyzing JIRA ticket with DART AI: " + e.getMessage());

}

}

private String extractCommitRefs(Issue issue) {

*// Simulate extracting commit hashes from JIRA ticket comments or custom fields*

return issue.getComments().stream()

.filter(comment -> comment.getBody().contains("git commit"))

.map(comment -> comment.getBody().split("git commit ")[1].split("\n")[0])

.collect(Collectors.joining(","));

}

private String fetchHistoricalData(String jiraTicket) {

*// Placeholder: Fetch historical data (e.g., from MongoDB or PostgreSQL)*

*// In reality, query the Data Layer for documents related to the JIRA ticket or workflow*

return "{\"workflow\": \"Equities Cash\", \"docs\": \"Historical trading data for equities...\"}";

}

private DartAIResponse callDartAI(DartAIRequest request) {

*// Call DART AI REST API for refined analysis*

ResponseEntity<DartAIResponse> response = restTemplate.postForEntity(

dartAiUrl,

request,

DartAIResponse.class

);

return response.getBody();

}

private List<JiraTicketResponse> createJiraTicketsForImpactedSystems(List<String> impactedSystems, JiraRestClient jiraClient) {

List<JiraTicketResponse> tickets = new ArrayList<>();

for (String system : impactedSystems) {

try {

IssueType issueType = jiraClient.getMetadataClient().getIssueTypes().claim()

.stream().filter(it -> "Task".equals(it.getName())).findFirst()

.orElseThrow(() -> new RuntimeException("Task issue type not found"));

IssueInput issueInput = new IssueInput(

jiraProjectKey,

issueType.getId(),

"Impact Analysis for " + system,

"Automatically generated ticket for refined impacted system: " + system + ". Needs review and updates due to DART AI analysis."

);

issueInput.addFieldInput(new FieldInput("priority", "Medium"));

issueInput.addFieldInput(new FieldInput("labels", Arrays.asList("ImpactAnalysis", "DARTAI", "Automation")));

Issue newIssue = jiraClient.getIssueClient().createIssue(issueInput).claim();

String ticketUrl = jiraUrl + "/browse/" + newIssue.getKey();

tickets.add(new JiraTicketResponse(newIssue.getKey(), ticketUrl, "Created"));

} catch (Exception e) {

tickets.add(new JiraTicketResponse(null, null, "Failed: " + e.getMessage()));

}

}

return tickets;

}

}

*// Service for JIRA ticket creation (unchanged)*

@Service

class JiraTicketCreator {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

public JiraRestClient getJiraClient() {

return new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

}

}

*// Controller (unchanged, but you can extend with DART AI-specific endpoints)*

@RestController

@RequestMapping("/api/impact")

public class ImpactAnalysisController {

private final ImpactAnalysisService impactService;

public ImpactAnalysisController(ImpactAnalysisService impactService) {

this.impactService = impactService;

}

@PostMapping("/analyze-jira")

public ResponseEntity<ImpactedAnalysis> analyzeJiraImpact(@RequestBody String jiraTicket) {

ImpactedAnalysis analysis = impactService.analyzeImpactsFromJira(jiraTicket);

return ResponseEntity.ok(analysis);

}

@GetMapping("/created-tickets")

public ResponseEntity<List<JiraTicketResponse>> getCreatedTickets() {

return ResponseEntity.ok(new ArrayList<>());

}

}

*// Main Application Class (unchanged)*

@SpringBootApplication

public class TradingSystemApplication {

public static void main(String[] args) {

SpringApplication.run(TradingSystemApplication.class, args);

}

}

**Input and Output Structure**

**Input**

* **HTTP Request (POST /api/impact/analyze-jira)**:
  + **Body**: JSON containing the JIRA ticket ID (e.g., "PROJ-123").
  + Example:

json

WrapCopy

{

"jiraTicket": "PROJ-123"

}

* **DART AI Request (DartAIRequest)**:
  + Sent internally to the DART AI REST API.
  + Example:

json

WrapCopy

{

"jiraTicket": "PROJ-123",

"gitCommits": ["abc123", "def456"],

"historicalData": "{\"workflow\": \"Equities Cash\", \"docs\": \"Historical trading data for equities...\"}"

}

**Output**

* **HTTP Response (200 OK)**:
  + **Body**: JSON containing the ImpactedAnalysis object with refined and precise lists of impacted systems and classes from DART AI.
  + Example:

json

WrapCopy

{

"impactedSystems": ["Equities Cash Trading System", "Electronic Order Management System"],

"impactedClasses": ["EquityTradeProcessorV2", "CashTransactionValidator", "TradingEngineCore"]

}

* **Created JIRA Tickets**:
  + Automatically created JIRA tickets for each refined impacted system (e.g., “Equities Cash Trading System”).
  + Each ticket includes:
    - **Project**: TRAD (or your JIRA project key).
    - **Issue Type**: “Task”.
    - **Summary**: “Impact Analysis for [System Name]”.
    - **Description**: “Automatically generated ticket for refined impacted system: [System Name]. Needs review and updates due to DART AI analysis.”
    - **Priority**: “Medium”.
    - **Labels**: “ImpactAnalysis, DARTAI, Automation”.
  + **Response (via GET /api/impact/created-tickets)**:
    - Example:

json

WrapCopy

[

{

"ticketId": "TRAD-456",

"ticketUrl": "https://your-jira-instance.atlassian.net/browse/TRAD-456",

"status": "Created"

},

{

"ticketId": "TRAD-457",

"ticketUrl": "https://your-jira-instance.atlassian.net/browse/TRAD-457",

"status": "Created"

}

]

**Error Handling**

* **400 Bad Request**: If the JIRA ticket is invalid or missing.
* **500 Internal Server Error**: If there’s an issue connecting to JIRA, Git, DART AI, or creating tickets (e.g., API failures, low confidence scores).

**Explanation of the Integration**

1. **DART AI Integration**:
   * The callDartAI method uses RestTemplate to send a DartAIRequest to the DART AI REST API (e.g., http://dart-ai-service:8080/analyze).
   * DART AI processes the JIRA ticket, Git commits, and historical data to refine impacted systems/classes using advanced NLP/ML techniques (e.g., entity recognition, graph analysis, pattern matching).
   * DART AI returns a DartAIResponse with refined outputs and a confidence score (e.g., ≥0.9 for 90% accuracy).
2. **Refined and Precise Information**:
   * DART AI improves precision by analyzing historical documentation, workflow data (e.g., Equities Cash), and Git commit diffs to map impacted classes to systems accurately.
   * It reduces false positives/negatives by learning from past data and SME feedback, ensuring outputs align with the 90% automation goal.
3. **Automatic JIRA Ticket Creation**:
   * After DART AI refines the impacted systems, the service creates JIRA tickets with updated descriptions reflecting DART AI’s analysis, ensuring stakeholders review precise impacts.
4. **Confidence Validation**:
   * The code checks DART AI’s confidenceScore to ensure outputs meet the 90% accuracy requirement. If the score is too low, it throws an exception, prompting retraining or manual review.

**Implementation Details for DART AI**

**DART AI API (Hypothetical)**

Assume DART AI has a REST API with the following endpoint:

* **URL**: http://dart-ai-service:8080/analyze
* **Method**: POST
* **Request Body**: DartAIRequest (JSON)
* **Response**: DartAIResponse (JSON)

**Example DART AI Implementation (Python Pseudocode)**

If DART AI is a Python microservice, here’s a high-level pseudocode for its logic:

python

WrapCopy

from flask import Flask, request, jsonify

import numpy as np

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.cluster import KMeans

app = Flask(\_\_name\_\_)

@app.route('/analyze', methods=['POST'])

def analyze\_impact():

data = request.json

jira\_ticket = data['jiraTicket']

commits = data['gitCommits']

historical\_data = data['historicalData']

*# Step 1: Parse and vectorize data (e.g., commits, historical docs)*

vectorizer = TfidfVectorizer()

text\_data = f"{commits} {historical\_data}"

vectors = vectorizer.fit\_transform([text\_data])

*# Step 2: Use ML (e.g., KMeans) to cluster and identify patterns*

kmeans = KMeans(n\_clusters=2, random\_state=42)

clusters = kmeans.fit\_predict(vectors)

*# Step 3: Map clusters to impacted systems/classes (e.g., rule-based or trained model)*

impacted\_systems = ["Equities Cash Trading System", "Electronic Order Management System"]

impacted\_classes = ["EquityTradeProcessorV2", "CashTransactionValidator"]

*# Step 4: Calculate confidence score (e.g., based on model accuracy)*

confidence\_score = 0.95 *# Simulate high confidence*

return jsonify({

"refinedImpactedSystems": impacted\_systems,

"refinedImpactedClasses": impacted\_classes,

"confidenceScore": confidence\_score

})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(host='0.0.0.0', port=8080)

**Java Wrapper for DART AI**

If DART AI is Python-based, use RestTemplate (as shown) or a Java wrapper (e.g., JNA/JNI) to call Python scripts. Alternatively, deploy DART AI as a microservice and communicate via REST.

**Configuration for Integration**

**application.properties**

Add the DART AI URL:

properties

WrapCopy

dart.ai.url=http://dart-ai-service:8080/analyze

**Dependencies**

Update pom.xml to include spring-boot-starter-web for RestTemplate:

xml

WrapCopy

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

**Notes**

* This integration assumes DART AI provides a REST API. If it’s a library or different interface, adjust the callDartAI method (e.g., use a Java wrapper or direct library calls).
* For production, secure the DART AI API with authentication (e.g., OAuth, API keys) and handle rate limiting/retry logic.
* Test DART AI’s precision with SME/BA validation to ensure 90% accuracy, retraining as needed.
* If you have specific DART AI documentation or an existing implementation, I can refine this further to match its capabilities.

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

To address your request comprehensively, I’ll walk through the process of gathering requirements from the provided image, creating a High-Level Design (HLD), Low-Level Design (LLD), and developing a Java Spring Boot application. This application will integrate with Confluence and JIRA to fetch historical data, identify impacted systems and classes, and automatically create JIRA tickets for impacted systems, using MongoDB as the data layer. I’ll include code snippets, input/output structures, and explanations for each step.

**1. Requirement Gathering**

Based on the image, the requirements for the Cash and Electronic Trading use-case are:

**Cash and Electronic Trading Use-Case:**

* **Input Historical Data**: Take input of historical documentations (based on a specific workflow, e.g., Equities Cash) and a list of impacted systems in the workflow.
* **Impact Assessment**: Use a model to generate an impact assessment for the workflow for a new regulatory/client ask, based on historical documentation (Confluence, PDFs, etc.) and manual inputs.
* **Model Evolution**: Upon rollout of new releases, the model identifies the new baseline to avoid re-use of stale data. Over multiple iterations, the model becomes an SME assistant, generating 90% of requirements, requiring only a BA last look, reducing SME resourcing within the organization.
* **Step-2 (Identify Impacted Classes/Code in Git)**:
  + Identify impacted classes/code in Git across Git projects.
  + Generate test cases by component, front to back.
  + Generate code per application stack > components.

**Additional Context:**

* Use Java Spring Boot for the application.
* Fetch historical data from Confluence and JIRA.
* Store data in MongoDB.
* Identify impacted systems/classes and create JIRA tickets automatically.

**2. High-Level Design (HLD)**

**2.1. Overview**

The HLD provides a high-level architectural overview of the Cash and Electronic Trading system, focusing on major components, interactions, and the technology stack.

**2.2. System Architecture**

**Layers:**

* **Presentation Layer (Frontend)**:
  + Description: User interface for SMEs/BAs to input manual data, view impact assessments, and review generated outputs.
  + Technology: HTML, CSS, JavaScript (React.js or Angular).
  + Interaction: Communicates with the backend via RESTful APIs.
* **Business Logic Layer (Backend)**:
  + Description: Handles core business logic, processes historical data from Confluence/JIRA, integrates with Git, and generates outputs using AI models.
  + Technology: Java Spring Boot (RESTful services, Spring Data MongoDB, Spring Security).
  + Interaction: Interfaces with the frontend, AI layer, data layer (MongoDB), and external systems (Confluence, JIRA, Git).
* **AI and Machine Learning Layer**:
  + Description: Uses DART AI for data analysis, baseline establishment, and pattern recognition; uses Gen AI for generating requirements, test cases, and code.
  + Technology: Custom-trained models (Python/TensorFlow/PyTorch), accessed via REST API or Java wrapper.
  + Interaction: Receives inputs from the backend, processes data, and returns refined outputs.
* **Data Layer**:
  + Description: Stores historical documentation, workflow data, impacted systems/classes, JIRA tickets, and generated outputs.
  + Technology: MongoDB (for flexible, unstructured data like Confluence pages, PDFs, JIRA data).
  + Interaction: Accessed via Spring Data MongoDB.
* **Integration Layer**:
  + Description: Manages interactions with Confluence, JIRA, and Git for data fetching and impact analysis.
  + Technology: JIRA REST API, Confluence REST API, JGit (for Git), REST clients (e.g., Unirest, OkHttp).
  + Interaction: Fetches data, identifies impacted systems/classes, and creates JIRA tickets.

**2.3. Major Components**

* **Impact Assessment Engine**: Uses DART AI to analyze historical data and manual inputs, generating impact assessments.
* **Requirement Generator**: Leverages Gen AI to produce 90% of requirements, requiring BA validation.
* **Code and Test Case Generator**: Uses Gen AI to generate code and test cases for impacted classes/components, integrated with Git.
* **Workflow Manager**: Manages workflows (e.g., Equities Cash), updating baselines to avoid stale data.
* **JIRA Ticket Creator**: Automatically creates JIRA tickets for impacted systems.

**2.4. Data Flow**

1. **Input**: Historical data from Confluence/JIRA, manual inputs from SMEs/BAs, and Git repository data.
2. **Processing**: Backend processes inputs, sends them to DART AI/Gen AI for analysis/generation, and integrates with Git.
3. **Output**: Generated impact assessments, requirements, test cases, code, and JIRA tickets, stored in MongoDB and displayed via the frontend.
4. **Integration**: Confluence/JIRA provide historical data, Git identifies impacted classes, and JIRA creates tickets.

**2.5. Non-Functional Requirements**

* **Scalability**: Handle increasing volumes using cloud-based MongoDB (e.g., MongoDB Atlas).
* **Performance**: Process data within seconds for real-time use.
* **Security**: Use Spring Security, SSL, and role-based access control for Confluence/JIRA/Git APIs.
* **Reliability**: Achieve 99.9% uptime with monitoring and failover for MongoDB.

**2.6. Technology Stack**

* **Languages**: Java (Spring Boot), JavaScript (Frontend), Python (AI models).
* **Frameworks**: Spring Boot, React.js/Angular, Spring Data MongoDB, JUnit, Mockito.
* **AI Tools**: DART AI (custom or third-party), Gen AI (custom-trained or platforms like Hugging Face).
* **Databases**: MongoDB.
* **Integration**: JIRA REST API, Confluence REST API, JGit, REST clients (Unirest/OkHttp).

**3. Low-Level Design (LLD)**

**3.1. Overview**

The LLD provides detailed design for implementation, including class diagrams, API endpoints, MongoDB schema, and integration workflows.

**3.2. Class Diagram (Backend - Java Spring Boot)**

plantuml

WrapCopy

@startuml

class ImpactAssessmentService {

-ConfluenceClient confluenceClient

-JiraClient jiraClient

-GitClient gitClient

-DartAIClient dartAIClient

-GenAIClient genAIClient

-MongoRepository mongoRepo

+ImpactedAnalysis generateImpactAssessment(Document, List<String>)

}

class Requi*rementGenerator {*

-GenAIClient genAIClient

+List<Requi*rement> generateRequirements(ImpactAssessment, Workflow)*

}

class CodeTestCaseGenerator {

-GenAIClient genAIClient

-GitClient gitClient

+GeneratedOutput generateCodeTestCases(String, ApplicationStack)

}

class WorkflowManager {

-MongoRepository mongoRepo

+void updateBaseline(Release)

}

class JiraTicketCreator {

-JiraClient jiraClient

+List<JiraTicketResponse> createTickets(List<String>)

}

class Document {

-String content

-String workflowId

-Timestamp createdAt

+getContent(): String

}

class ImpactAssessment {

-List<String> impactedSystems

-List<String> impactedClasses

-JSONObject baseline

+getImpactedSystems(): List<String>

}

class Requi*rement {*

-String description

-String status

-Timestamp createdAt

+getDescription(): String

}

class GeneratedOutput {

-String code

-String testCases

+getCode(): String

}

class Release {

-JSONObject data

-Timestamp timestamp

+getData(): JSONObject

}

ImpactAssessmentService o--> ConfluenceClient

ImpactAssessmentService o--> JiraClient

ImpactAssessmentService o--> GitClient

ImpactAssessmentService o--> DartAIClient

ImpactAssessmentService o--> GenAIClient

ImpactAssessmentService o--> MongoRepository

Requi*rementGenerator o--> GenAIClient*

CodeTestCaseGenerator o--> GenAIClient

CodeTestCaseGenerator o--> GitClient

WorkflowManager o--> MongoRepository

JiraTicketCreator o--> JiraClient

Document o--> "1" ImpactAssessment

ImpactAssessment o--> "0..\*" Requi*rement*

ImpactAssessment o--> "0..\*" GeneratedOutput

WorkflowManager o--> Release

note right of ImpactAssessmentService

Orchestrates impact assessment

using DART AI and external data

end note

@enduml

**3.3. API Endpoints (RESTful - Spring Boot)**

* **POST /api/impact/assessment**:
  + Description: Generate impact assessment for a workflow (e.g., Equities Cash) using Confluence/JIRA data.
  + Input: Workflow ID, historical data (Confluence/JIRA), manual inputs.
  + Output: ImpactedAnalysis (JSON).
* **POST /api/requirements/generate**:
  + Description: Generate requirements using Gen AI.
  + Input: ImpactAssessment, manual inputs.
  + Output: List of Requirement objects (JSON).
* **POST /api/code-test-cases/generate**:
  + Description: Generate code and test cases for impacted classes.
  + Input: ImpactedClass, ApplicationStack.
  + Output: GeneratedOutput (JSON).
* **POST /api/jira/create-tickets**:
  + Description: Create JIRA tickets for impacted systems.
  + Input: List of impacted systems.
  + Output: List of JiraTicketResponse (JSON).

**3.4. MongoDB Schema**

json

WrapCopy

*// Document Collection*

{

"\_id": ObjectId,

"content": String,

"workflowId": String,

"createdAt": ISODate

}

*// ImpactAssessment Collection*

{

"\_id": ObjectId,

"documentId": ObjectId,

"impactedSystems": [String],

"impactedClasses": [String],

"baseline": Object,

"createdAt": ISODate

}

*// Requirement Collection*

{

"\_id": ObjectId,

"impactAssessmentId": ObjectId,

"description": String,

"status": String,

"createdAt": ISODate

}

*// GeneratedCode Collection*

{

"\_id": ObjectId,

"className": String,

"code": String,

"testCases": String,

"applicationStack": String,

"createdAt": ISODate

}

**3.5. Integration Workflows**

* **Confluence Integration**: Use Confluence REST API to fetch historical documents (e.g., pages, attachments) for workflows like Equities Cash.
* **JIRA Integration**: Use JIRA REST API to fetch ticket data, commit references, and create tickets for impacted systems.
* **Git Integration**: Use JGit to analyze commits and identify impacted classes/code.
* **DART AI/Gen AI**: Call REST APIs or libraries for refined analysis and generation.

**4. Developed Code (Java Spring Boot Application)**

**Dependencies (pom.xml)**

xml

WrapCopy

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-web</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-data-mongodb</artifactId>

</dependency>

<dependency>

<groupId>com.atlassian.jira</groupId>

<artifactId>jira-rest-java-client-api</artifactId>

<version>5.2.5</version>

</dependency>

<dependency>

<groupId>org.eclipse.jgit</groupId>

<artifactId>org.eclipse.jgit</artifactId>

<version>6.5.0.202303070854-r</version>

</dependency>

<dependency>

<groupId>com.mashape.unirest</groupId>

<artifactId>unirest-java</artifactId>

<version>1.4.9</version>

</dependency>

</dependencies>

**application.properties**

properties

WrapCopy

spring.data.mongodb.uri=mongodb://localhost:27017/trading-system

jira.url=https://your-jira-instance.atlassian.net

jira.username=your-username

jira.api.token=your-api-token

jira.project.key=TRAD

confluence.url=https://your-confluence-instance.atlassian.net

confluence.username=your-username

confluence.api.token=your-api-token

git.repo.path=/path/to/your/git/repo

dart.ai.url=http://dart-ai-service:8080/analyze

gen.ai.url=http://gen-ai-service:8080/generate

**Models**

java

WrapCopy

import org.springframework.data.annotation.Id;

import org.springframework.data.mongodb.core.mapping.Document;

import java.time.Instant;

@Document(collection = "documents")

public class Document {

@Id

private String id;

private String content;

private String workflowId;

private Instant createdAt;

*// Getters, setters, constructors*

}

@Document(collection = "impactAssessments")

public class ImpactAssessment {

@Id

private String id;

private String documentId;

private List<String> impactedSystems;

private List<String> impactedClasses;

private Object baseline;

private Instant createdAt;

*// Getters, setters, constructors*

}

@Document(collection = "requirements")

public class Requirement {

@Id

private String id;

private String impactAssessmentId;

private String description;

private String status;

private Instant createdAt;

*// Getters, setters, constructors*

}

@Document(collection = "generatedCode")

public class GeneratedOutput {

@Id

private String id;

private String className;

private String code;

private String testCases;

private String applicationStack;

private Instant createdAt;

*// Getters, setters, constructors*

}

public class ImpactedAnalysis {

private List<String> impactedSystems;

private List<String> impactedClasses;

*// Getters, setters, constructors*

}

public class DartAIRequest {

private String jiraTicket;

private List<String> gitCommits;

private String historicalData;

*// Getters, setters, constructors*

}

public class DartAIResponse {

private List<String> refinedImpactedSystems;

private List<String> refinedImpactedClasses;

private double confidenceScore;

*// Getters, setters, constructors*

}

**Services**

java

WrapCopy

import com.atlassian.jira.rest.client.api.JiraRestClient;

import com.atlassian.jira.rest.client.api.domain.Issue;

import com.atlassian.jira.rest.client.internal.async.AsynchronousJiraRestClientFactory;

import com.mashape.unirest.http.HttpResponse;

import com.mashape.unirest.http.Unirest;

import com.mashape.unirest.http.exceptions.UnirestException;

import org.eclipse.jgit.api.Git;

import org.eclipse.jgit.lib.Repository;

import org.eclipse.jgit.revwalk.RevCommit;

import org.springframework.beans.factory.annotation.Value;

import org.springframework.data.mongodb.core.MongoTemplate;

import org.springframework.stereotype.Service;

import org.springframework.web.client.RestTemplate;

import java.net.URI;

import java.util.\*;

import java.util.stream.Collectors;

@Service

public class ImpactAnalysisService {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

@Value("${jira.project.key}")

private String jiraProjectKey;

@Value("${confluence.url}")

private String confluenceUrl;

@Value("${confluence.username}")

private String confluenceUsername;

@Value("${confluence.api.token}")

private String confluenceApiToken;

@Value("${git.repo.path}")

private String gitRepoPath;

@Value("${dart.ai.url}")

private String dartAiUrl;

@Value("${gen.ai.url}")

private String genAiUrl;

private final MongoTemplate mongoTemplate;

private final JiraTicketCreator jiraTicketCreator;

private final RestTemplate restTemplate = new RestTemplate();

public ImpactAnalysisService(MongoTemplate mongoTemplate, JiraTicketCreator jiraTicketCreator) {

this.mongoTemplate = mongoTemplate;

this.jiraTicketCreator = jiraTicketCreator;

}

public ImpactedAnalysis generateImpactAssessment(String workflowId, String jiraTicket) {

try {

*// Step 1: Fetch historical data from Confluence*

String confluenceData = fetchHistoricalDataFromConfluence(workflowId);

*// Step 2: Fetch JIRA ticket details*

JiraRestClient jiraClient = new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

Issue issue = jiraClient.getIssueClient().getIssue(jiraTicket).claim();

String commitRefs = extractCommitRefs(issue);

List<String> commits = Arrays.asList(commitRefs.split(","));

*// Step 3: Fetch Git commits and analyze impacted classes*

List<String> impactedClasses = analyzeGitCommits(commits);

*// Step 4: Prepare DART AI request with all data*

String historicalData = "{\"workflow\": \"" + workflowId + "\", \"confluence\": \"" + confluenceData + "\", \"jira\": \"" + issue.getDescription() + "\"}";

DartAIRequest dartRequest = new DartAIRequest(jiraTicket, commits, historicalData);

DartAIResponse dartResponse = callDartAI(dartRequest);

*// Step 5: Validate DART AI output*

if (dartResponse.getConfidenceScore() < 0.9) {

throw new RuntimeException("DART AI confidence too low: " + dartResponse.getConfidenceScore());

}

*// Step 6: Store impact assessment in MongoDB*

ImpactAssessment assessment = new ImpactAssessment();

assessment.setDocumentId(generateId()); *// Placeholder for document ID*

assessment.setImpactedSystems(dartResponse.getRefinedImpactedSystems());

assessment.setImpactedClasses(dartResponse.getRefinedImpactedClasses());

assessment.setBaseline(new Object()); *// Placeholder for baseline*

assessment.setCreatedAt(Instant.now());

mongoTemplate.save(assessment, "impactAssessments");

*// Step 7: Generate requirements with Gen AI*

List<Requirement> requirements = generateRequirements(assessment);

*// Step 8: Generate code and test cases*

List<GeneratedOutput> outputs = generateCodeTestCases(dartResponse.getRefinedImpactedClasses());

*// Step 9: Create JIRA tickets for impacted systems*

List<JiraTicketResponse> createdTickets = createJiraTicketsForImpactedSystems(dartResponse.getRefinedImpactedSystems(), jiraClient);

return new ImpactedAnalysis(dartResponse.getRefinedImpactedSystems(), dartResponse.getRefinedImpactedClasses());

} catch (Exception e) {

throw new RuntimeException("Error generating impact assessment: " + e.getMessage());

}

}

private String fetchHistoricalDataFromConfluence(String workflowId) throws UnirestException {

*// Use Confluence REST API to fetch pages/documents for the workflow*

HttpResponse<String> response = Unirest.get(confluenceUrl + "/rest/api/content?spaceKey=TRAD&title=Equities+Cash")

.basicAuth(confluenceUsername, confluenceApiToken)

.asString();

return response.getBody();

}

private String extractCommitRefs(Issue issue) {

return issue.getComments().stream()

.filter(comment -> comment.getBody().contains("git commit"))

.map(comment -> comment.getBody().split("git commit ")[1].split("\n")[0])

.collect(Collectors.joining(","));

}

private List<String> analyzeGitCommits(List<String> commitHashes) {

List<String> impactedClasses = new ArrayList<>();

try (Repository repo = Git.open(new java.io.File(gitRepoPath)).getRepository()) {

Git git = new Git(repo);

for (String commitHash : commitHashes) {

RevCommit commit = git.log().add(repo.resolve(commitHash)).call().iterator().next();

git.diff()

.setOldTree(git.getRepository().resolve(commit.getParent(0).getName()))

.setNewTree(git.getRepository().resolve(commit.getName()))

.call()

.forEach(diff -> {

String filePath = diff.getNewPath();

if (filePath.endsWith(".java")) {

String className = filePath.substring(filePath.lastIndexOf("/") + 1, filePath.lastIndexOf(".java"));

impactedClasses.add(className);

}

});

}

} catch (Exception e) {

throw new RuntimeException("Error analyzing Git commits: " + e.getMessage());

}

return impactedClasses.stream().distinct().collect(Collectors.toList());

}

private DartAIResponse callDartAI(DartAIRequest request) {

ResponseEntity<DartAIResponse> response = restTemplate.postForEntity(

dartAiUrl,

request,

DartAIResponse.class

);

return response.getBody();

}

private List<Requirement> generateRequirements(ImpactAssessment assessment) {

*// Call Gen AI to generate 90% of requirements*

DartAIRequest genRequest = new DartAIRequest(

"N/A",

new ArrayList<>(),

"{\"impact\": \"" + assessment.getImpactedSystems() + "\", \"workflow\": \"Equities Cash\"}"

);

DartAIResponse genResponse = restTemplate.postForEntity(

genAiUrl + "/requirements",

genRequest,

DartAIResponse.class

).getBody();

List<Requirement> requirements = new ArrayList<>();

for (String desc : genResponse.getRefinedImpactedSystems()) {

Requirement req = new Requirement();

req.setImpactAssessmentId(assessment.getId());

req.setDescription(desc);

req.setStatus("Pending BA Review");

req.setCreatedAt(Instant.now());

requirements.add(req);

}

mongoTemplate.saveAll(requirements, "requirements");

return requirements;

}

private List<GeneratedOutput> generateCodeTestCases(List<String> impactedClasses) {

List<GeneratedOutput> outputs = new ArrayList<>();

for (String className : impactedClasses) {

DartAIRequest genRequest = new DartAIRequest(

"N/A",

new ArrayList<>(),

"{\"class\": \"" + className + "\", \"stack\": \"Java Spring Boot\"}"

);

DartAIResponse genResponse = restTemplate.postForEntity(

genAiUrl + "/code-test-cases",

genRequest,

DartAIResponse.class

).getBody();

GeneratedOutput output = new GeneratedOutput();

output.setClassName(className);

output.setCode(genResponse.getRefinedImpactedSystems().get(0)); *// Assume first item is code*

output.setTestCases(genResponse.getRefinedImpactedClasses().get(0)); *// Assume first item is test cases*

output.setApplicationStack("Java Spring Boot");

output.setCreatedAt(Instant.now());

outputs.add(output);

mongoTemplate.save(output, "generatedCode");

}

return outputs;

}

private List<JiraTicketResponse> createJiraTicketsForImpactedSystems(List<String> impactedSystems, JiraRestClient jiraClient) {

List<JiraTicketResponse> tickets = new ArrayList<>();

for (String system : impactedSystems) {

try {

IssueType issueType = jiraClient.getMetadataClient().getIssueTypes().claim()

.stream().filter(it -> "Task".equals(it.getName())).findFirst()

.orElseThrow(() -> new RuntimeException("Task issue type not found"));

IssueInput issueInput = new IssueInput(

jiraProjectKey,

issueType.getId(),

"Impact Analysis for " + system,

"Automatically generated ticket for impacted system: " + system + ". Needs review due to Confluence/JIRA/Git analysis."

);

issueInput.addFieldInput(new FieldInput("priority", "Medium"));

issueInput.addFieldInput(new FieldInput("labels", Arrays.asList("ImpactAnalysis", "Automation")));

Issue newIssue = jiraClient.getIssueClient().createIssue(issueInput).claim();

String ticketUrl = jiraUrl + "/browse/" + newIssue.getKey();

tickets.add(new JiraTicketResponse(newIssue.getKey(), ticketUrl, "Created"));

} catch (Exception e) {

tickets.add(new JiraTicketResponse(null, null, "Failed: " + e.getMessage()));

}

}

return tickets;

}

private String generateId() {

return UUID.randomUUID().toString();

}

}

@Service

class JiraTicketCreator {

@Value("${jira.url}")

private String jiraUrl;

@Value("${jira.username}")

private String jiraUsername;

@Value("${jira.api.token}")

private String jiraApiToken;

public JiraRestClient getJiraClient() {

return new AsynchronousJiraRestClientFactory()

.createWithBasicHttpAuthentication(URI.create(jiraUrl), jiraUsername, jiraApiToken);

}

}

@RestController

@RequestMapping("/api/impact")

public class ImpactAnalysisController {

private final ImpactAnalysisService impactService;

public ImpactAnalysisController(ImpactAnalysisService impactService) {

this.impactService = impactService;

}

@PostMapping("/assessment")

public ResponseEntity<ImpactedAnalysis> generateImpactAssessment(@RequestBody Map<String, String> request) {

String workflowId = request.get("workflowId");

String jiraTicket = request.get("jiraTicket");

ImpactedAnalysis analysis = impactService.generateImpactAssessment(workflowId, jiraTicket);

return ResponseEntity.ok(analysis);

}

@GetMapping("/created-tickets")

public ResponseEntity<List<JiraTicketResponse>> getCreatedTickets() {

return ResponseEntity.ok(new ArrayList<>());

}

}

@SpringBootApplication

public class TradingSystemApplication {

public static void main(String[] args) {

SpringApplication.run(TradingSystemApplication.class, args);

}

}

**5. Input and Output Structure**

**Input**

* **HTTP Request (POST /api/impact/assessment)**:
  + **Body**: JSON containing workflow ID and JIRA ticket.
  + Example:

json

WrapCopy

{

"workflowId": "EquitiesCash",

"jiraTicket": "PROJ-123"

}

**Output**

* **HTTP Response (200 OK)**:
  + **Body**: JSON containing the ImpactedAnalysis object.
  + Example:

json

WrapCopy

{

"impactedSystems": ["Equities Cash Trading System", "Electronic Order Management System"],

"impactedClasses": ["EquityTradeProcessorV2", "CashTransactionValidator"]

}

* **Created JIRA Tickets**:
  + Automatically created for each impacted system, with details as described in the LLD.

**6. Notes**

* **Confluence/JIRA Integration**: Use REST APIs to fetch data. You may need to handle pagination, authentication, and rate limits.
* **MongoDB**: Use Spring Data MongoDB for CRUD operations. Ensure indexes for performance (e.g., on workflowId, createdAt).
* **DART AI/Gen AI**: Assumes REST APIs. Replace with actual endpoints or libraries if available.
* **Scalability**: Consider cloud-based MongoDB (e.g., MongoDB Atlas) and microservices for AI components.
* **Testing**: Use JUnit, Mockito, and integration tests with real Confluence/JIRA/Git data for validation.