IntelliInspect

Software Design Document

Version: 1.0
Date: August 7, 2025
Team: Team 5 – Sentinels
Team Members: Naghul, Ragavi, Sri Mahalakshmi, Harsha
Mentor: Shri Hari M
Authors: Sri Mahalakshmi, Harsha

Table of Contents

1. Introduction

1.1 Purpose

This Software Design Document (SDD) provides a comprehensive technical specification for **IntelliInspect**, a real-time predictive quality control application. The document serves as the primary reference for developers, architects, and stakeholders involved in the development, deployment, and maintenance of the system. It details the architectural design, component interactions, API specifications, and deployment strategies for the complete solution.

1.2 Scope

IntelliInspect is a full-stack AI-powered application that enables real-time quality control prediction using Kaggle Production Line sensor data. The system encompasses:

- **Frontend Application**: Angular-based single-page application with step-based workflow navigation
- **Backend API Gateway**: ASP.NET Core service providing authentication, request orchestration, and WebSocket proxying
- **Machine Learning Service**: Python FastAPI service handling data processing, model training, and real-time inference
- **Containerized Deployment**: Docker-based orchestration enabling scalable, environment-agnostic deployment

The application processes large-scale manufacturing datasets (up to 2.5GB), performs sophisticated machine learning operations, and provides real-time streaming predictions through WebSocket connections.

1.3 Objectives

The primary objectives of IntelliInspect align with the hackathon specifications and business requirements:

Technical Objectives: - Implement a robust three-tier architecture supporting large-scale data processing - Provide seamless user experience through intuitive workflow navigation - Enable real-time prediction streaming with sub-second latency - Ensure scalable deployment through containerization

Business Objectives: - Demonstrate advanced machine learning capabilities for manufacturing quality control - Provide comprehensive model evaluation and performance visualization - Enable simulation of production-line monitoring scenarios - Support decision-making through predictive analytics and confidence scoring

Performance Objectives: - Handle CSV files up to 2.5GB through chunked upload processing - Achieve model training completion within acceptable timeframes - Maintain real-time streaming at one-second intervals during simulation - Ensure system availability and reliability across all deployment environments

2. System Architecture

2.1 Architectural Overview

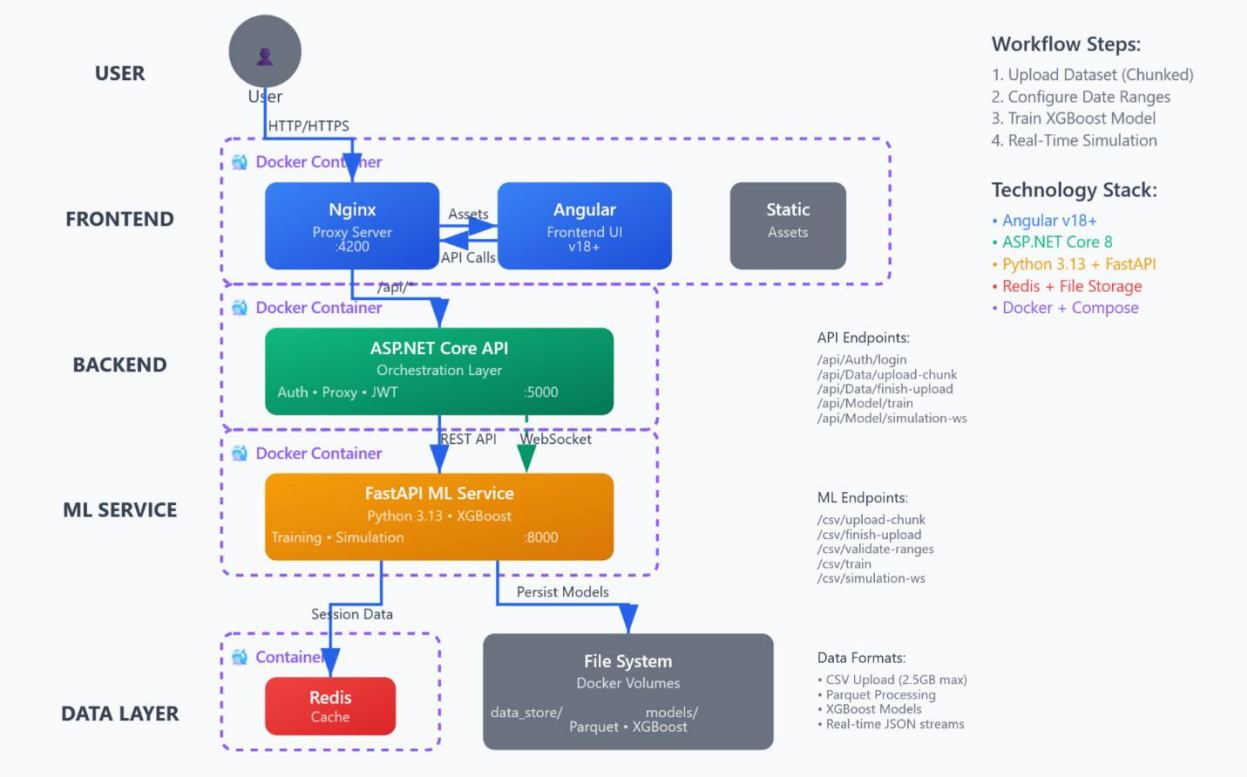
IntelliInspect implements a sophisticated three-tier architecture orchestrating modern web technologies with advanced machine learning capabilities. The system employs a containerized microservices approach, where each tier operates independently while maintaining seamless integration through well-defined interfaces.

Architecture Principles: - **Separation of Concerns**: Each tier handles distinct responsibilities with minimal coupling - **Scalability**: Horizontal scaling capabilities through

containerization - **Security**: JWT-based authentication with role-based access control - **Performance**: Optimized data processing through chunked uploads and streaming protocols - **Maintainability**: Modular design enabling independent service updates and deployments

The architecture supports both synchronous REST API communications for standard operations and asynchronous WebSocket connections for real-time data streaming, providing a robust foundation for complex machine learning workflows.

2.2 System Architecture Diagram



The system architecture diagram illustrates the three-tier design with Angular frontend, .NET Core backend, Python ML service, and supporting infrastructure including Nginx proxy, Redis cache, and Docker containerization.

2.3 Component Breakdown

Frontend Tier - Angular Application

Technology Stack: Angular 18+, TypeScript, Chart.js
Responsibilities: - User interface presentation and interaction management - Step-based workflow navigation (Upload → Date Ranges → Training → Simulation) - Real-time data visualization through interactive charts and dashboards - Client-side validation and user experience optimization - Theme management and responsive design implementation

Backend Tier - .NET Core API Gateway

Technology Stack: ASP.NET Core 8, JWT Authentication, SignalR
Responsibilities: - Request orchestration and service coordination - JWT-based authentication and authorization - Bidirectional WebSocket proxying for real-time communication - Error handling and response formatting - Cross-origin resource sharing (CORS) management

ML Service Tier - Python FastAPI

Technology Stack: Python 3.13, FastAPI, XGBoost, pandas, scikit-learn
Responsibilities: - Large-scale CSV data processing and augmentation - Advanced machine learning model training and evaluation - Real-time prediction inference and confidence scoring - Feature engineering and model optimization - Simulation data preparation and streaming

Infrastructure Components

Nginx Proxy: - Static asset serving and request routing - WebSocket connection upgrade handling - Load balancing and performance optimization

Redis Cache: - Temporary data storage for simulation sessions - High-performance caching for frequently accessed metadata - Session state management

Docker Containers: - Service isolation and deployment consistency - Multi-stage build optimization - Environment-agnostic deployment capabilities

3. Data Flow

3.1 Data Flow Overview

The IntelliInspect data flow architecture demonstrates sophisticated data movement patterns across three primary workflows: file upload processing, model training orchestration, and real-time simulation streaming.

File Upload Data Flow

The upload process begins with user interaction through the Angular drag-and-drop interface. Large CSV files (up to 2.5GB) are processed through an intelligent chunked upload mechanism that divides files into manageable segments. The Angular frontend coordinates chunk transmission to the .NET backend, which proxies these segments to the FastAPI ML service. The ML service implements a revolutionary single-pass processing algorithm that simultaneously reassembles chunks, performs data validation, extracts metadata, and converts the dataset to optimized Parquet format for efficient storage and retrieval.

Model Training Data Flow

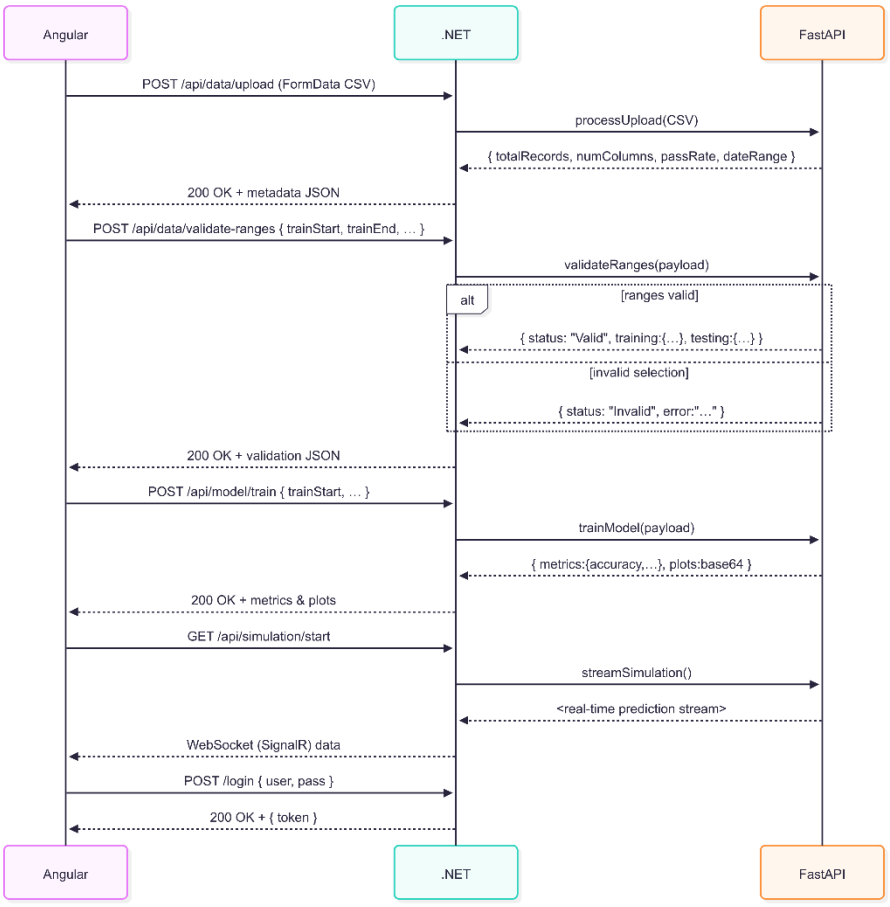
Training initiation occurs when users submit validated date ranges through the frontend interface. The .NET backend forwards comprehensive training requests to the ML service,

which loads preprocessed Parquet data and executes sophisticated XGBoost model training. The training pipeline incorporates advanced feature engineering, class imbalance handling through ADASYN resampling, and precision optimization specifically designed for manufacturing scenarios. Upon completion, the service returns comprehensive performance metrics, confusion matrix data, and Base64-encoded feature importance visualizations.

Real-Time Simulation Data Flow

The simulation workflow establishes bidirectional WebSocket connections enabling real-time prediction streaming. The .NET backend operates as an intelligent proxy, maintaining simultaneous connections to both the Angular frontend and Python ML service. The ML service streams individual predictions at one-second intervals, including row indices, prediction values, confidence scores, actual values, and correctness indicators. This streaming architecture enables real-time dashboard updates while maintaining memory efficiency through generator-based iteration.

3.2 Data Flow Diagram



The data flow diagram illustrates the three primary workflows: chunked file upload processing, model training orchestration, and real-time WebSocket simulation streaming, showing data movement between Angular frontend, .NET backend, and Python ML service.

4. API Contract

4.1 Frontend to Backend Communication

Base URL: `{environment.apiUrl}` (typically `/api`)

Authentication Controller (/api/Auth)

Endpoint	Method	Description
`/api/Auth/login`	POST	Authenticate user credentials and return JWT token
`/api/Auth/verify`	GET	Verify JWT token and return user role information

POST /api/Auth/login - **Request Body**: `json { "username": "string", "password": "string" }` - **Success Response (200 OK)**: `json { "token": "string", "username": "string" }` - **Error Response (401 Unauthorized)**: `"Invalid credentials."`

GET /api/Auth/verify - **Headers**: `Authorization: Bearer {token}` - **Success Response (200 OK)**: `json { "message": "Success", "role": "string" }`

Data Controller (/api/Data)

Endpoint	Method	Description
`/api/Data/upload-chunk`	POST	Upload individual file chunk as part of chunked upload process
`/api/Data/finish-upload`	POST	Finalize chunked upload by reassembling chunks and processing
`/api/Data/validate-ranges`	POST	Validate date ranges against stored dataset and return record counts

POST /api/Data/upload-chunk - **Headers**: `Authorization: Bearer {token}` - **Request Body (multipart/form-data)**: - `file`: IFormFile (chunk data) - `uploadId`: string - `chunkIndex`: string (numeric) - `userId`: string - **Success Response (200 OK)**: `{}`

POST /api/Data/finish-upload - **Headers**: `Authorization: Bearer {token}` - **Request Body**: `json { "uploadId": "string", "fileName": "string", "userId": "string", "totalChunks": 0 }` - **Success Response (200 OK)**: `json { "datasetId": "string", "userId": "string", "parquetPath": "string", "totalRecords": 0, "numColumns": 0, "passRate": 0.0, "dateRange": { "start": "string", "end": "string" } }`

POST /api/Data/validate-ranges - **Headers**: `Authorization: Bearer {token}` - **Request Body**: `json { "userId": "string", "datasetId": "string", "dateRanges": { "training": {"start": "string", "end": "string"}, "testing": {"start": "string", "end": "string"}, "simulation": {"start": "string", "end": "string"} } }` - **Success Response (200 OK)**: `json`

```
{     "status": "Valid",     "training": {"count": 0},     "testing":
{"count": 0},     "simulation": {"count": 0},     "monthlyCounts": {"2023-
01": 0, "2023-02": 0}   }
```

Model Controller (/api/Model)

Endpoint	Method	Description
`/api/Model/train`	POST	Train XGBoost model using specified date ranges
`/api/Model/simulation-ws`	WebSocket	Real-time ML model simulation streaming

POST /api/Model/train - **Headers**: `Authorization: Bearer {token}` - **Request Body**: `json   {     "userId": "string",     "datasetId": "string",     "dateRanges": {       "training": {"start": "string", "end": "string"},       "testing": {"start": "string", "end": "string"},       "simulation": {"start": "string", "end": "string"}     }   }` - **Success Response (200 OK)**: `json   {     "metrics": {       "accuracy": 0.0, "precision": 0.0, "recall": 0.0, "f1Score": 0.0, "truePositive": 0, "falsePositive": 0, "trueNegative": 0, "falseNegative": 0 },     "plots": {       "featureImportance": "string",       "trainingPlot": "string"     }   }`

4.2 Backend to ML Service Communication

Base URL: `http://localhost:8000` (Development) / `http://ml_service:8000` (Production)

CSV Processor Router (/csv)

Endpoint	Method	Description
`/csv/upload-chunk`	POST	Receive and store individual file chunk
`/csv/finish-upload`	POST	Reassemble chunks and process to Parquet format
`/csv/validate-ranges`	POST	Validate date ranges and return record counts
`/csv/train`	POST	Train XGBoost model with comprehensive metrics
`/csv/simulation-ws`	WebSocket	Real-time prediction streaming endpoint

POST /csv/finish-upload - **Request Body**: `json   {     "uploadId": "string", "fileName": "string",     "userId": "string",     "totalChunks": 0   }` - **Success Response (200 OK)**: `json   {     "datasetId": "string", "userId": "string", "parquetPath": "string",     "totalRecords": 0, "numColumns": 0, "passRate": 0.0,     "dateRange": {"start": "string", "end": "string"}   }`

WebSocket /csv/simulation-ws - Client Messages: `json {"action": "start", "userId": "string", "datasetId": "string"} {"action": "stop", "userId": "string", "datasetId": "string"}` - **Server Messages**: `json { "rowIndex": 0, "prediction": 0, "confidence": 0.0, "actual": 0, "isCorrect": true, "timestamp": "string" }`

5. Deployment and Orchestration

5.1 Containerization Strategy

IntelliInspect employs a sophisticated multi-stage build strategy across all services, optimizing both build efficiency and runtime footprint while ensuring production-ready deployments.

Multi-Stage Build Benefits

- **Build Optimization**: Separation of build-time dependencies from runtime requirements
- **Image Size Reduction**: Exclusion of development tools and temporary artifacts
- **Security Enhancement**: Minimal attack surface through slim production images
- **Performance Improvement**: Faster deployment and network transfer times

Service-Specific Build Strategies

Frontend Container: - Stage 1: Node.js environment for Angular asset compilation - Stage 2: Nginx slim image serving optimized static assets - Features: Intelligent request routing and WebSocket upgrade handling

Backend Container: - Stage 1: .NET SDK for application compilation - Stage 2: ASP.NET runtime for minimal production footprint - Features: JWT authentication and WebSocket proxy capabilities

ML Service Container: - Stage 1: Python development environment with scientific computing libraries - Stage 2: Python slim runtime with optimized virtual environment - Features: XGBoost model training and real-time inference capabilities

5.2 Service Orchestration

The Docker Compose orchestration implements strategic service dependency management and persistent data handling through carefully designed volume mappings.

Service Dependency Chain

1. **Redis**: Foundational data layer initialization
2. **ML Service**: Requires Redis connectivity for caching operations
3. **Backend**: Depends on ML service availability for request forwarding
4. **Frontend**: Relies on backend API endpoints for complete functionality

Network Architecture

- **Custom Bridge Network**: `abb-network` provides isolated service communication

- **Service Discovery**: DNS resolution using compose service names
- **Security**: Internal communication isolation with controlled external access

Persistent Storage Strategy

- **ML Service Volumes**:
  - `data_store/`: Processed Parquet datasets
  - `models/`: Trained XGBoost model artifacts
- **Benefits**: State persistence across container lifecycle events
- **Development Integration**: Direct host access for debugging and analysis

Configuration Management

```
# Key orchestration elements
services:
  frontend:
    ports: ["4200:80"]
    depends_on: [backend]
  backend:
    ports: ["5000:80"]
    depends_on: [ml_service]
  ml_service:
    ports: ["8000:8000"]
    depends_on: [redis]
    volumes:
      - ./data_store:/app/data_store
      - ./models:/app/models
```

5.3 Hosting and Deployment

IntellInspect utilizes Docker Compose for containerized deployment with the following specifications:

Prerequisites

Software Requirements:

- Docker

- Docker Compose

Hardware Requirements:

- Minimum 8GB RAM (recommended for optimal performance)

- Available ports: 4200 (frontend), 5000 (backend), 8000 (ML service)

Deployment Process

Clone the repository

git clone [repository-url]

cd intelliinspect

Launch the Application

sudo docker compose up --build -d

Access the application

Web browser: http://localhost:4200

Verify status: docker compose ps

Shutdown the Application

sudo docker compose down

Service Endpoints

Component	Port	Access URL
Frontend	4200	http://localhost:4200
Backend API	5000	http://localhost:5000
ML Service	8000	http://localhost:8000

6. Key Architectural Decisions & Trade-offs

6.1 Technology Choices

Frontend: Angular 18+

Justification: Angular provides enterprise-grade framework capabilities with robust TypeScript support, comprehensive testing utilities, and extensive ecosystem integration. The framework's reactive programming model through RxJS aligns perfectly with real-time data streaming requirements, while its component architecture enables modular development and maintainability.

Benefits: - Strong typing through TypeScript for reduced runtime errors - Reactive programming support for WebSocket integration - Comprehensive CLI tooling for

development efficiency - Enterprise-ready with extensive documentation and community support

Backend: ASP.NET Core 8

Justification: ASP.NET Core offers high-performance, cross-platform capabilities with built-in dependency injection, middleware pipeline flexibility, and native WebSocket support. The framework's integration with JWT authentication and its ability to handle concurrent connections makes it ideal for orchestrating complex service communications.

Benefits: - High-performance request processing with async/await patterns - Native WebSocket proxy capabilities for real-time communication - Robust security framework with JWT and CORS support - Excellent Docker integration and deployment flexibility

ML Service: Python 3.13 + FastAPI

Justification: Python's dominance in the machine learning ecosystem, combined with FastAPI's modern async capabilities and automatic API documentation generation, provides an optimal foundation for ML operations. The combination enables rapid development of sophisticated ML pipelines while maintaining production-grade performance.

Benefits: - Extensive ML library ecosystem (XGBoost, pandas, scikit-learn) - FastAPI's async capabilities for concurrent request handling - Automatic OpenAPI documentation generation - Native WebSocket support for real-time streaming

Containerization: Docker + Docker Compose

Justification: Docker provides consistent deployment environments across development, testing, and production scenarios. Docker Compose enables sophisticated multi-service orchestration with declarative configuration, dependency management, and network isolation.

Benefits: - Environment consistency and deployment reliability - Service isolation and scalability through horizontal scaling - Simplified dependency management and version control - Development workflow optimization through local orchestration

6.2 Architectural Trade-offs

Memory Architecture Decision

Decision: Load entire datasets into memory during model training
Trade-off: High memory requirements vs. maximum processing speed
Justification: This design choice prioritizes performance during the critical training phase, requiring servers to have sufficient RAM to hold 2GB Parquet files comfortably. While this increases infrastructure requirements, it ensures optimal training performance and eliminates I/O bottlenecks during computationally intensive operations.

Implications: - **Positive**: Lightning-fast model training and feature engineering - **Negative**: Higher infrastructure costs and memory requirements - **Mitigation**: Clear documentation of minimum system requirements

WebSocket Proxy Architecture

Decision: .NET backend serves as WebSocket proxy between frontend and ML service
Trade-off: Additional latency vs. security and authentication benefits
Justification: While introducing a proxy layer adds minimal latency, it enables consistent authentication, logging, and error handling across all communications. This architecture maintains security boundaries while providing transparent real-time communication.

Implications: - **Positive**: Centralized authentication, logging, and error handling - **Negative**: Slight increase in communication latency - **Mitigation**: Optimized proxy implementation with concurrent message relay

Single-Pass CSV Processing

Decision: Process entire CSV files in a single pass during upload
Trade-off: Complex algorithm implementation vs. dramatic performance improvement
Justification: The revolutionary single-pass algorithm eliminates redundant file I/O operations by simultaneously performing chunk reassembly, metadata extraction, validation, and Parquet conversion. This approach significantly outperforms traditional multi-pass processing methods.

Implications: - **Positive**: Exceptional performance for large file processing - **Negative**: Increased algorithm complexity and memory usage during processing - **Mitigation**: Comprehensive error handling and memory management

Authentication Strategy

Decision: JWT-based authentication with hardcoded demonstration credentials
Trade-off: Rapid development vs. production security requirements
Justification: For hackathon and demonstration purposes, this approach enables rapid development and testing while providing the infrastructure for proper authentication. The JWT framework supports easy migration to database-backed user management systems.

Implications: - **Positive**: Quick implementation and demonstration capabilities - **Negative**: Not suitable for production deployment without modification - **Mitigation**: Clear documentation marking this as demonstration-only with migration path outlined

This Software Design Document represents the comprehensive technical specification for IntelliInspect v1.0, providing the foundation for development, deployment, and maintenance of the real-time predictive quality control system.