TrialPulse Nexus 10X

Technical Documentation

Module 1: Core Infrastructure

Project Setup, Configuration, and Environment

Version: 1.0.0
Date: January 2026
Platform: Cross-platform (Windows, macOS, Linux)

AI-Powered Clinical Trial Intelligence Platform

Contents

1 Overview

1.1 Purpose

This module documents the core infrastructure of TrialPulse Nexus, including project setup, configuration management, environment variables, and the unified launcher system. The infrastructure is designed to support:

- Cross-platform deployment (Windows, macOS, Linux)
- Multiple database backends (local PostgreSQL, Docker containers)
- Reproducible environment setup for evaluators and developers
- One-command project initialization

1.2 Directory Structure

The core infrastructure consists of the following key files and directories:

```
TrialPulseNexus/
          .env                      # Environment variables (secrets)
          .env.example              # Template for environment setup
          run.py                    # Unified launcher script
          setup_repro_db.py         # Database restoration script
          requirements.txt          # Python dependencies
          config/
                settings.py           # Application-wide configuration
                llm_config.py         # LLM provider settings
                mlflow_config.py      # ML experiment tracking config
          backend/
              app/
                  config.py         # FastAPI application settings
```

2 Environment Configuration

2.1 Environment Variables

All configuration is managed through a `.env` file at the project root. This approach ensures:

1. **Security**: Secrets are not committed to version control
2. **Flexibility**: Easy configuration changes without code modification
3. **Reproducibility**: Clear separation of configuration from logic

2.1.1 Database Configuration

Listing 1: Database Environment Variables

```
# Database Configuration
DB_HOST=127.0.0.1
DB_PORT=5432
DB_NAME=trialpulse_test
DB_USER=postgres
DB_PASSWORD=your_postgres_password
```

```
# Connection Pool Settings
DB_POOL_SIZE=10
DB_MAX_OVERFLOW=20
DB_POOL_TIMEOUT=30
DB_POOL_RECYCLE=1800
```

Table 1: Database Configuration Parameters

Variable	Default	Description
DB_HOST	127.0.0.1	PostgreSQL server hostname
DB_PORT	5432	PostgreSQL server port
DB_NAME	trialpulse_test	Database name
DB_USER	postgres	Database username
DB_PASSWORD	(required)	Database password
DB_POOL_SIZE	10	SQLAlchemy connection pool size
DB_MAX_OVERFLOW	20	Maximum overflow connections

2.1.2 AI/LLM Configuration

The system supports dual LLM providers for redundancy:

Listing 2: AI Provider Configuration

```
# Primary Provider - Groq (Cloud)
LLM_PRIMARY_PROVIDER=groq
GROQ_API_KEY=your_groq_api_key

# Secondary Provider - Ollama (Local)
LLM_SECONDARY_PROVIDER=ollama
OLLAMA_BASE_URL=http://localhost:11434
OLLAMA_MODEL=llama3
```

Fallback Mechanism: If the primary provider (Groq) fails or times out, the system automatically falls back to the local Ollama instance. This ensures high availability for production deployments.

2.1.3 Knowledge Graph Configuration

Listing 3: Neo4j Knowledge Graph Settings

```
NEO4J_ENABLED=True
NEO4J_URI=bolt://127.0.0.1:7687
NEO4J_USER=neo4j
NEO4J_PASSWORD=your_neo4j_password
NEO4J_DATABASE=neo4j
```

3 Application Settings

3.1 FastAPI Configuration (backend/app/config.py)

The backend uses Pydantic's `BaseSettings` for type-safe configuration loading:

```python
from pydantic_settings import BaseSettings, SettingsConfigDict

class Settings(BaseSettings):
    """Application settings loaded from environment"""

    model_config = SettingsConfigDict(
        env_file=".env",
        case_sensitive=True,
        extra="ignore"
    )

    # Application
    APP_NAME: str = "TrialPulse Nexus API"
    APP_VERSION: str = "1.0.0"
    DEBUG: bool = True

    # Security
    SECRET_KEY: str = "fallback-secret-key"
    ALGORITHM: str = "HS256"
    ACCESS_TOKEN_EXPIRE_MINUTES: int = 30

    # Database
    DB_HOST: str = "localhost"
    DB_PORT: int = 5432
    DB_NAME: str = "trialpulse_nexus"
    DB_USER: str = "postgres"
    DB_PASSWORD: str = "postgres"

    @property
    def DATABASE_URL(self) -> str:
        return f"postgresql://{self.DB_USER}:{self.DB_PASSWORD}@{self.DB_HOST}:{self.DB_PORT}/{self.DB_NAME}"
```

Listing 4: FastAPI Settings Class

Key Design Decisions:

- **Type Safety**: All settings have explicit type hints, preventing runtime errors
- **Fallback Values**: Sensible defaults ensure the application can start even with minimal configuration
- **LRU Cache**: Settings are cached using `@lru_cache()` to avoid repeated file reads

3.2 Domain Configuration (config/settings.py)

The `config/settings.py` file contains domain-specific configurations using Python dataclasses:

3.2.1 Data Quality Index (DQI) Configuration

The DQI is a weighted composite score calculated from 8 components:

```python
@dataclass
class DQIConfig:
    """Configuration for DQI calculation."""

    # Component weights (must sum to 1.0)
    weights: Dict[str, float] = field(default_factory=lambda: {
```

```
        'safety_score': 0.25,        # SAE discrepancies
        'query_score': 0.20,         # Open queries
        'completeness_score': 0.15, # Missing visits/pages
        'coding_score': 0.12,        # Uncoded terms
        'lab_score': 0.10,           # Lab issues
        'sdv_score': 0.08,           # SDV completion
        'signature_score': 0.05,     # Overdue signatures
        'edrr_score': 0.05           # Third-party issues
    })
```

Listing 5: DQI Weight Configuration

Table 2: DQI Component Weights

Component	Description	Weight
Safety Score	Serious Adverse Event discrepancies	25%
Query Score	Outstanding data queries	20%
Completeness Score	Missing visits and pages	15%
Coding Score	Uncoded medical terms	12%
Lab Score	Laboratory data issues	10%
SDV Score	Source Data Verification completion	8%
Signature Score	Overdue e-signatures	5%
EDRR Score	Third-party data issues	5%
Total		**100%**

3.2.2 Age Multipliers

Issues are penalized based on how long they have remained unresolved:

```
age_multipliers: Dict[str, float] = field(default_factory=lambda: {
    '0-7': 1.0,     # No penalty for first week
    '8-14': 1.1,    # 10% penalty
    '15-30': 1.3,   # 30% penalty
    '31-60': 1.5,   # 50% penalty
    '60+': 1.6      # 60% penalty for issues > 60 days
})
```

Listing 6: Age-Based Penalty Multipliers

3.2.3 DQI Quality Bands

The final DQI score is categorized into quality bands:

Table 3: DQI Quality Bands

Band	Score Range	Interpretation
Pristine	95 – 100	Exceptional data quality
Excellent	85 – 94.99	High-quality, minor issues
Good	75 – 84.99	Acceptable, some attention needed
Moderate	65 – 74.99	Requires focused remediation
Concerning	50 – 64.99	Significant issues present
Critical	0 – 49.99	Urgent intervention required

4 Unified Launcher (run.py)

4.1 Purpose

The `run.py` script provides a single entry point for running the entire TrialPulse Nexus system. It handles:

1. Dependency verification (Python, Node.js, PostgreSQL/Docker)
2. Virtual environment creation and dependency installation
3. Database initialization and data restoration
4. Concurrent backend and frontend server management
5. Graceful shutdown on Ctrl+C

4.2 Command-Line Interface

Listing 7: Launcher CLI Options

```
# Auto-detect best available mode
python run.py

# Force Docker PostgreSQL
python run.py --docker

# Force local PostgreSQL
python run.py --local

# Frontend-only (no backend)
python run.py --frontend-only

# Skip database setup
python run.py --skip-db
```

4.3 Execution Modes

4.3.1 Auto-Detect Mode (Default)

When run without arguments, the launcher automatically detects the best configuration:

```
def auto_detect_mode():
    """Auto-detect the best mode to run."""

    has_docker = check_docker()
    has_local_psql = find_local_psql() is not None

    if has_local_psql:
        return run_full_local()   # Prefer local PostgreSQL
    elif has_docker:
        return run_full_docker()  # Fall back to Docker
    else:
        return run_frontend_only() # Last resort: UI only
```

Listing 8: Auto-Detection Logic

4.3.2 Docker Mode

Docker mode automatically:

1. Removes any existing `trialpulse-postgres` container (fresh start)
2. Creates a new PostgreSQL 14 container with correct credentials
3. Waits for PostgreSQL to become available (30-second timeout)
4. Copies the database dump into the container
5. Restores the dump using `psql`

```python
def docker_start_postgres():
    """Start PostgreSQL in Docker."""

    # Always remove old container for fresh start
    docker_remove_container()

    cmd = [
        "docker", "run", "-d",
        "--name", "trialpulse-postgres",
        "-p", "5432:5432",
        "-e", "POSTGRES_PASSWORD=TrialPulseNexus",
        "-e", "POSTGRES_USER=postgres",
        "-e", "POSTGRES_DB=trialpulse_test",
        "postgres:14"
    ]

    run_cmd(cmd)

    # Wait for PostgreSQL to be ready
    if not wait_for_postgres(timeout=30):
        return False

    return True
```

Listing 9: Docker Container Creation

4.4 Cross-Platform Compatibility

The launcher handles platform-specific differences:

```python
def find_local_psql():
    """Find local PostgreSQL installation."""

    # Check PATH first (works on all platforms)
    if command_exists("psql"):
        return "psql"

    # Windows-specific paths
    if is_windows():
        base_paths = [
            Path(r"C:\Program Files\PostgreSQL"),
            Path(r"C:\Program Files (x86)\PostgreSQL"),
        ]
        for base in base_paths:
            for version in ["18", "17", "16", "15", "14", "13", "12"]:
```

```
                psql = base / version / "bin" / "psql.exe"
                if psql.exists():
                    return str(psql)

    # macOS Homebrew
    homebrew_psql = Path("/opt/homebrew/bin/psql")
    if homebrew_psql.exists():
        return str(homebrew_psql)

    # Linux
    linux_psql = Path("/usr/bin/psql")
    if linux_psql.exists():
        return str(linux_psql)

    return None
```

Listing 10: Cross-Platform Path Detection

4.5 Process Management

The launcher manages multiple concurrent processes and ensures clean shutdown:

```
def cleanup_processes(signum=None, frame=None):
    """Clean up running processes on exit."""

    for name, proc in running_processes:
        if proc.poll() is None:
            proc.terminate()
            try:
                proc.wait(timeout=5)
            except subprocess.TimeoutExpired:
                proc.kill()

    # Cleanup Docker container if used
    if docker_container_running():
        docker_cleanup()

    sys.exit(0)

# Register signal handlers
signal.signal(signal.SIGINT, cleanup_processes)
signal.signal(signal.SIGTERM, cleanup_processes)
```

Listing 11: Graceful Shutdown Handler

5 Reproducibility

5.1 Environment Requirements

5.2 Setup Instructions

1. **Clone the repository**:

```
git clone https://github.com/Dhruv-x7x/TRIALPULSE-NEXUS-deployed.
    git
cd TRIALPULSE-NEXUS-deployed
```

Table 4: System Requirements

Component	Minimum Version	Notes
Python	3.10+	Required for type hints
Node.js	18+	Required for Vite frontend
PostgreSQL	14+	OR Docker Desktop
Docker (optional)	Any	Alternative to local PostgreSQL
Neo4j (optional)	5+	For knowledge graph features
Ollama (optional)	Any	For local LLM inference

2. **Configure environment**:

```
cp .env.example .env
# Edit .env with your database password and API keys
```

3. **Download database dump**: Place `reproduction_dump.sql` (212 MB) in the `database/` folder.

4. **Run the application**:

```
python run.py           # Auto-detect mode
# OR
python run.py --docker # If using Docker
```

5.3 Random Seed Settings

For reproducible machine learning results, the following seeds are used throughout the codebase:

```
import random
import numpy as np

RANDOM_SEED = 42

random.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)

# For scikit-learn models
model = RandomForestClassifier(random_state=RANDOM_SEED)

# For train-test splits
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=RANDOM_SEED
)
```

Listing 12: Standard Random Seed Configuration

5.4 Dependency Management

All Python dependencies are pinned with version ranges in `requirements.txt`:

Listing 13: Key Dependencies (requirements.txt)

```
# Web Framework
fastapi>=0.109.0,<0.116.0
uvicorn[standard]>=0.27.0,<0.35.0
```

```
# Database
sqlalchemy>=2.0.25,<3.0.0
psycopg2-binary>=2.9.9,<3.0.0

# Machine Learning
scikit-learn>=1.3.0,<1.7.0
xgboost>=2.0.0,<3.0.0

# LLM Integration
groq>=0.4.0,<1.0.0
ollama>=0.1.0,<1.0.0
```

The total dependency count is approximately:

- **Python**: 126 packages
- **Node.js**: 40 packages (frontend)

6 Data Integrity

6.1 Database Restoration

The database dump contains the complete state of the system with:

- 57,974 patient records
- 3,401 clinical sites
- 23 studies
- All pre-computed DQI scores and risk classifications

6.2 Verification Process

After restoration, the system verifies data integrity:

```python
def verify_database():
    """Verify database was restored correctly."""

    expected_counts = {
        'patients': 57974,
        'sites': 3401,
        'studies': 23
    }

    for table, expected in expected_counts.items():
        actual = db.execute(f"SELECT COUNT(*) FROM {table}").scalar()
        assert actual == expected, f"Mismatch in {table}"
```

Listing 14: Database Verification

6.3 No Data Leakage

For machine learning components, data integrity is maintained through:

1. **Temporal Splits**: Training data uses historical records; test data uses recent records

2. **Patient-Level Splits**: All records from a patient stay in either train or test set (no patient overlap)

3. **Stratification**: Class distribution is preserved in both train and test sets

TrialPulse Nexus 10X

Technical Documentation

Module 2: Data Pipeline

Ingestion, Cleaning, UPR Building, and Metrics

Version: 1.0.0
Date: January 2026
Records: 57,974 Patients

Clinical Trial Data Quality Engine

Contents

1 Pipeline Overview

1.1 Purpose

The Data Pipeline module transforms raw clinical trial exports from multiple sources into a unified, analysis-ready dataset. The pipeline processes data through four sequential stages:

1. **Ingestion**: Parse Excel files from 9 different data sources
2. **Cleaning**: Standardize columns, remove duplicates, validate data
3. **UPR Building**: Join all sources into Unified Patient Records
4. **Metrics Calculation**: Compute DQI scores, clean patient status, risk flags

1.2 Data Flow Architecture

The pipeline follows a sequential flow through four processing stages:

Stage	Component	Output
1	Raw Excel Files (9 Sources)	Input Data
	↓	
2	**Ingestion Engine**	Staging Tables (PostgreSQL)
	↓	
3	**Cleaning Engine**	Cleaned Tables
	↓	
4	**UPR Builder**	Unified Patient Records
	↓	
5	**Metrics Engine**	Final UPR with DQI + Clean Status

1.3 Dashboard Overview

Figure 1: Executive Dashboard

1.4 Data Sources

The pipeline processes 9 distinct data source types:

Table 1: Clinical Data Sources

Source	Description	Key Metrics
CPID EDC Metrics	Core patient data from EDC	Visits, queries, SDV, signatures
Visit Projection	Missing/overdue visits	Days outstanding
Missing Lab Ranges	Lab abnormalities	Issue count per patient
SAE Dashboard (DM)	Data Management SAE reviews	Pending/completed status
SAE Dashboard (Safety)	Safety SAE reviews	Pending/completed status
Inactivated Forms	Deactivated CRF forms	Form count, reasons
Missing Pages	Missing CRF pages	Days missing
Compiled EDRR	Third-party reconciliation	Issue count
Coding (MedDRA + WHODrug)	Medical term coding	Coded/uncoded counts

2 Data Ingestion (ingestion.py)

2.1 Purpose

The ingestion engine discovers, parses, and loads raw Excel files from study folders into PostgreSQL staging tables. It handles varying file formats, column names, and data quality issues.

2.2 File Discovery

Files are discovered using pattern matching against known source types:

```python
def identify_file_type(filename: str) -> Optional[str]:
    filename_check = filename.lower()
    mapping = {
        'cpid_edc_metrics': ['edc', 'metric', 'cpid'],
        'missing_lab_ranges': ['lab', 'lnr', 'range'],
        'sae_dashboard': ['sae', 'safety', 'dashboard'],
        'coding_meddra': ['meddra', 'medra'],
        'coding_whodrug': ['whodd', 'whodrug'],
        'visit_projection': ['visit', 'projection'],
        'missing_pages': ['missing_page'],
        'compiled_edrr': ['edrr'],
        'inactivated_forms': ['inactivated']
    }
    for file_type, keywords in mapping.items():
        if any(kw in filename_check for kw in keywords):
            return file_type
    return None
```

Listing 1: File Type Identification

2.3 Column Standardization

All column names are standardized to a consistent format:

```python
def standardize_columns(columns: pd.Index) -> List[str]:
    new_cols = []
    for col in columns:
        col_clean = str(col).strip().lower()
        col_clean = re.sub(r'[^\w\s]', '', col_clean)  # Remove special chars
        col_clean = re.sub(r'\s+', '_', col_clean)     # Spaces to underscores
        if not col_clean: col_clean = 'unnamed'
        new_cols.append(col_clean)
    return new_cols
```

Listing 2: Column Name Standardization

2.4 Specialized Processors

Different data sources require specialized parsing logic:

2.4.1 CPID EDC Processor

The CPID EDC files have headers on row 3 and contain the core patient metrics:

```python
class CPIDEDCProcessor:
    def process(self, file_path: str, study_id: str):
        xl = pd.ExcelFile(file_path)
        # Find sheet containing subject data
        target_sheet = next(
            (s for s in xl.sheet_names if 'subject' in s.lower()),
            xl.sheet_names[0]
        )
        # Header is on row 3 (0-indexed: header=2)
```

```
        df = pd.read_excel(file_path, sheet_name=target_sheet, header=2)
        df = df.dropna(how='all')
        df.columns = standardize_columns(df.columns)
        # Add metadata columns
        df['_source_file'] = os.path.basename(file_path)
        df['_study_id'] = study_id
        df['_file_type'] = 'cpid_edc_metrics'
        return df, ""
```

Listing 3: CPID EDC Processor

2.4.2 SAE Dashboard Processor

SAE dashboards contain multiple sheets (DM and Safety):

```
class SAEDashboardProcessor:
    def process(self, file_path: str, study_id: str):
        xl = pd.ExcelFile(file_path)
        results = {}
        for sheet in xl.sheet_names:
            if 'safety' in sheet.lower() or 'dm' in sheet.lower():
                df = pd.read_excel(file_path, sheet_name=sheet)
                df.columns = standardize_columns(df.columns)
                key = 'safety' if 'safety' in sheet.lower() else 'dm'
                results[key] = df
        return results, ""
```

Listing 4: SAE Dashboard Processor

2.5 Ingestion Manifest

The engine tracks all processing in a manifest:

```
@dataclass
class IngestionManifest:
    run_id: str
    start_time: str
    end_time: str = ""
    status: str = "running"
    studies_found: int = 0
    files_found: int = 0
    files_processed: int = 0
    files_success: int = 0
    files_failed: int = 0
    records_by_type: Dict[str, int] = field(default_factory=dict)
    records_by_study: Dict[str, int] = field(default_factory=dict)
```

Listing 5: Ingestion Manifest Structure

3 Data Cleaning (cleaning.py)

3.1 Purpose

The cleaning engine standardizes identifiers, maps columns, removes duplicates, and prepares data for joining. This is critical for ensuring data integrity.

3.2 Column Mapping

CPID EDC files often have unnamed columns. We map these to meaningful names:

```
CPID_COLUMN_MAPPING = {
    'unnamed_0': 'project_name',
    'unnamed_1': 'region',
    'unnamed_2': 'country',
    'unnamed_3': 'site',
    'unnamed_4': 'subject',
    'unnamed_5': 'latest_visit',
    'unnamed_6': 'subject_status',
    'unnamed_7': 'input_files',
    'unnamed_8': 'cpmd',
    'unnamed_9': 'ssm',
    'unnamed_10': 'missing_visits',
    'unnamed_11': 'missing_pages',
    'unnamed_12': 'coded_terms',
    'unnamed_13': 'uncoded_terms',
    'unnamed_14': 'open_issues_lnr',
    'unnamed_15': 'open_issues_edrr',
    'unnamed_16': 'inactivated_forms_folders',
    'unnamed_17': 'sae_review_dm',
    'unnamed_18': 'sae_review_safety',
}
```

Listing 6: CPID Column Mapping

3.3 Subject Status Standardization

Patient statuses are normalized to canonical values:

```
SUBJECT_STATUS_MAPPING = {
    'screen failure': 'Screen Failure',
    'screen fail': 'Screen Failure',
    'discontinued': 'Discontinued',
    'withdrawn': 'Discontinued',
    'completed': 'Completed',
    'ongoing': 'Ongoing',
    'active': 'Ongoing',
    'enrolled': 'Ongoing',
    'in progress': 'Ongoing',
    'screening': 'Screening',
}
```

Listing 7: Status Normalization

Table 2: Subject Status Distribution (Example)

Status	Count
Ongoing	31,245
Completed	18,392
Discontinued	5,841
Screen Failure	2,496

3.4 Patient Key Generation

Each patient receives a unique composite key for joining:

```
def create_patient_key(study_id: str, site_id: str, subject_id: str) ->
    str:
    site = site_id if site_id else 'Unknown'
    subject = subject_id if subject_id else 'Unknown'
    return f"{study_id}|{site}|{subject}"

# Example: "Study_1|Site_101|Subject_001"
```

Listing 8: Patient Key Creation

3.5 Data Aggregation

Detail tables are aggregated to patient level before joining:

```
def aggregate_visit_projection(self, df: pd.DataFrame):
    # Aggregate to patient level
    agg = df.groupby('patient_key').agg(
        study_id=('study_id', 'first'),
        site_id=('site_id', 'first'),
        subject_id=('subject_id', 'first'),
        missing_visit_count=('visit', 'count'),
        visits_overdue_max_days=('days_outstanding', 'max'),
        visits_overdue_avg_days=('days_outstanding', 'mean')
    ).reset_index()
    return agg, stats
```

Listing 9: Visit Projection Aggregation

3.6 Junk Row Removal

The cleaner identifies and removes non-data rows:

```
JUNK_PATTERNS = ['responsible', 'lf for action', 'site/cra',
                 'coder', 'safety team', 'investigator']

def _remove_junk_rows(self, df: pd.DataFrame) -> pd.DataFrame:
    def is_junk(row):
        for col in check_cols:
            val = row.get(col)
            if pd.notna(val):
                val_lower = str(val).lower().strip()
                if any(p in val_lower for p in self.JUNK_PATTERNS):
                    return True
        return False

    mask = ~df.apply(is_junk, axis=1)
    return df[mask]
```

Listing 10: Junk Row Detection

3.7 Data Integrity Verification

- **Null Handling**: Numeric columns filled with 0, text columns with 'Unknown'
- **Duplicate Removal**: Exact duplicates dropped, keeping first occurrence

- **Type Conversion**: Numeric columns coerced with `pd.to_numeric(errors='coerce')`
- **Validation**: Subject IDs must be non-null and non-empty

4 Unified Patient Record Builder (upr_builder.py)

4.1 Purpose

The UPR Builder joins all cleaned data sources into a single wide table with one row per patient. This enables holistic analysis across all data domains.

4.2 Main Spine Construction

The CPID EDC Metrics table serves as the "spine" (master patient list):

```python
def load_main_spine(self):
    # Load cleaned CPID data as the main patient list
    query = "SELECT * FROM cpid_edc_metrics_clean"
    df = self._execute_sql(query)

    # Ensure required columns
    required = ['patient_key', 'study_id', 'site_id', 'subject_id']
    for col in required:
        if col not in df.columns:
            df[col] = 'Unknown'

    return df
```

Listing 11: Loading Main Spine

4.3 Smart Join Strategy

The builder uses a two-phase join strategy to maximize match rates:

```python
def join_detail_table(self, main_df, detail_df, table_name, prefix):
    # Phase 1: Try patient_key match
    merged = main_df.merge(
        detail_df,
        on='patient_key',
        how='left',
        suffixes=('', f'_{prefix}')
    )

    match_rate = merged['some_detail_col'].notna().mean()

    # Phase 2: If match rate < 50%, try study_id + subject_id
    if match_rate < 0.5:
        merged = main_df.merge(
            detail_df,
            on=['study_id', 'subject_id'],
            how='left',
            suffixes=('', f'_{prefix}')
        )

    return merged
```

Listing 12: Smart Join Strategy

4.4 Column Prefixing

Joined columns receive prefixes to indicate their source:

Table 3: Column Prefix Mapping

Source Table	Prefix
visit_projection_agg	visit_
missing_lab_ranges_agg	lab_
sae_dashboard_dm_agg	sae_dm_
sae_dashboard_safety_agg	sae_safety_
inactivated_forms_agg	inactivated_
missing_pages_agg	pages_
compiled_edrr_agg	edrr_
coding_meddra_agg	meddra_
coding_whodrug_agg	whodrug_

4.5 Derived Metrics

The builder calculates composite metrics from joined data:

```python
def calculate_derived_metrics(self, df: pd.DataFrame):
    # Total issues across all sources
    df['total_issues_all_sources'] = (
        df['visit_missing_visit_count'].fillna(0) +
        df['pages_missing_page_count'].fillna(0) +
        df['total_queries'].fillna(0) +
        df['lab_lab_issue_count'].fillna(0) +
        df['edrr_edrr_issue_count'].fillna(0)
    )

    # Total pending SAE
    df['total_sae_pending'] = (
        df['sae_dm_sae_dm_pending'].fillna(0) +
        df['sae_safety_sae_safety_pending'].fillna(0)
    )

    # Coding completion rate
    total_coded = df['meddra_coding_meddra_coded'].fillna(0) + \
                  df['whodrug_coding_whodrug_coded'].fillna(0)
    total_terms = df['meddra_coding_meddra_total'].fillna(0) + \
                  df['whodrug_coding_whodrug_total'].fillna(0)
    df['coding_completion_rate'] = np.where(
        total_terms > 0,
        (total_coded / total_terms * 100).clip(0, 100),
        100  # 100% if no terms to code
    )

    return df
```

Listing 13: Derived Metrics Calculation

4.6 Final UPR Schema

The final UPR contains approximately 150+ columns:

- **Identity**: patient_key, study_id, site_id, subject_id
- **Demographics**: region, country, subject_status
- **Visit Metrics**: missing counts, overdue days
- **Query Metrics**: DM, clinical, medical, safety, site, coding queries
- **SDV Metrics**: forms verified, forms required
- **Signature Metrics**: signed, never signed, overdue by age bucket
- **SAE Metrics**: pending/completed for DM and Safety
- **Coding Metrics**: MedDRA and WHODrug coded/uncoded counts
- **Other**: lab issues, EDRR issues, inactivated forms
- **Derived**: total issues, completion rates, risk flags

5 Metrics Engine (metrics_engine.py)

5.1 Overview

The Metrics Engine calculates three major derived metrics:

1. **Data Quality Index (DQI)**: 8-component weighted score (0-100)
2. **Clean Patient Status**: Two-tier eligibility assessment
3. **DB Lock Ready Status**: Database lock readiness flags

5.2 Data Quality Index (DQI)

5.2.1 Component Weights

The DQI is calculated as a weighted sum of 8 components:

Table 4: DQI Component Weights

Component	Description	Weight
Safety	SAE pending issues	25%
Query	Open data queries	20%
Completeness	Missing visits/pages	15%
Coding	Uncoded medical terms	12%
Lab	Laboratory issues	10%
SDV	Source Data Verification	8%
Signature	E-signature status	5%
EDRR	Third-party reconciliation	5%
Total		**100%**

5.2.2 Component Score Calculation

Each component starts at 100 and is penalized based on issue counts:

```python
def _calc_safety_score(self, df: pd.DataFrame) -> pd.Series:
    """Calculate Safety component (SAE pending issues)"""
    sae_dm_pending = get_col(df, 'sae_dm_pending')
    sae_safety_pending = get_col(df, 'sae_safety_pending')

    total_pending = sae_dm_pending + sae_safety_pending

    # Penalty: -15 per pending SAE (safety critical), max 100
    penalty = np.minimum(total_pending * 15, 100)
    score = 100 - penalty

    return score
```

Listing 14: Safety Score Calculation

```python
def _calc_query_score(self, df: pd.DataFrame) -> pd.Series:
    """Calculate Query component"""
    total_queries = get_col(df, 'total_open_queries')

    # Penalty: -3 per open query, max 100
    penalty = np.minimum(total_queries * 3, 100)
    score = 100 - penalty

    return score
```

Listing 15: Query Score Calculation

5.2.3 Weighted DQI Calculation

```python
def calculate_weighted_dqi(self, df: pd.DataFrame) -> pd.DataFrame:
    components = {
        'safety': ('dqi_safety_raw', 25.0),
        'query': ('dqi_query_raw', 20.0),
        'completeness': ('dqi_completeness_raw', 15.0),
        'coding': ('dqi_coding_raw', 12.0),
        'lab': ('dqi_lab_raw', 10.0),
        'sdv': ('dqi_sdv_raw', 8.0),
        'signature': ('dqi_signature_raw', 5.0),
        'edrr': ('dqi_edrr_raw', 5.0)
    }

    weighted_sum = pd.Series(0.0, index=df.index)
    for name, (col, weight) in components.items():
        weighted_component = result[col] * (weight / 100)
        weighted_sum += weighted_component

    result['dqi_score'] = weighted_sum.clip(0, 100)
    return result
```

Listing 16: Weighted DQI Calculation

Table 5: DQI Quality Bands

Band	Score Range	Action Required
Pristine	95 – 100	None
Excellent	85 – 94	Monitor only
Good	75 – 84	Minor attention
Fair	65 – 74	Focused remediation
Poor	50 – 64	Priority escalation
Critical	25 – 49	Urgent intervention
Emergency	0 – 24	Immediate action

5.2.4 DQI Quality Bands

5.3 Clean Patient Derivation

5.3.1 Tier 1: Clinical Clean (7 Hard Blocks)

A patient must pass ALL Tier 1 criteria to be considered "clinically clean":

Table 6: Tier 1 Clinical Clean Criteria

Criterion	Condition
No missing visits	visit_missing_visit_count = 0
No missing CRF pages	pages_missing_page_count = 0
No open queries	total_queries = 0
SDV complete	crfs_require_verification_sdv = 0
All signatures complete	crfs_never_signed = 0
MedDRA coding complete	meddra_coding_meddra_uncoded = 0
WHODrug coding complete	whodrug_coding_whodrug_uncoded = 0

5.3.2 Tier 2: Operational Clean (7 Soft Blocks)

Tier 1 clean patients must also pass Tier 2 for full "operational clean" status:

Table 7: Tier 2 Operational Clean Criteria

Criterion	Condition
No lab issues	lab_lab_issue_count = 0
No SAE DM pending	sae_dm_sae_dm_pending = 0
No SAE Safety pending	sae_safety_sae_safety_pending = 0
No EDRR issues	edrr_edrr_issue_count = 0
No overdue signatures (90+ days)	crfs_overdue_signs_90plus = 0
No inactivated forms	inactivated_form_count = 0
No broken signatures	broken_signatures = 0

5.3.3 Quick Wins Identification

The engine identifies patients close to clean status:

```
def calculate_quick_wins(self, df: pd.DataFrame):
    # Count failing criteria
```

```
    result['tier1_failing_count'] = sum of failing Tier 1 criteria
    result['tier2_failing_count'] = sum of failing Tier 2 criteria

    # Categories
    # - 'Already Clean': Tier 2 clean
    # - '1 Issue to Tier 2': Tier 1 clean, 1 soft block
    # - '1 Issue to Tier 1': 1 hard block away
    # - '2 Issues to Tier 1': 2 hard blocks away
    # - 'Not Quick Win': 3+ blocks
```

Listing 17: Quick Win Categories

5.4 DB Lock Ready Assessment

Database lock readiness is assessed in two tiers:

- **Tier 1 - Fully Ready**: Eligible status + Tier 2 clean + no pending safety items
- **Tier 2 - Pending**: Eligible status + Tier 1 clean but not Tier 1 ready
- **Not Ready**: Eligible but not Tier 1 clean
- **Not Eligible**: Screen failures and ineligible statuses

6 Validation and Results

6.1 Data Pipeline Statistics

Table 8: Pipeline Processing Results

Metric	Value
Total Studies Processed	23
Total Files Processed	207
Total Patient Records	57,974
Total Sites	3,401
Unique Patient Keys	57,974

6.2 DQI Distribution

Table 9: DQI Band Distribution (Example)

Band	Count	Percentage
Pristine	28,412	49.0%
Excellent	12,845	22.2%
Good	8,234	14.2%
Fair	4,892	8.4%
Poor	2,341	4.0%
Critical	1,102	1.9%
Emergency	148	0.3%

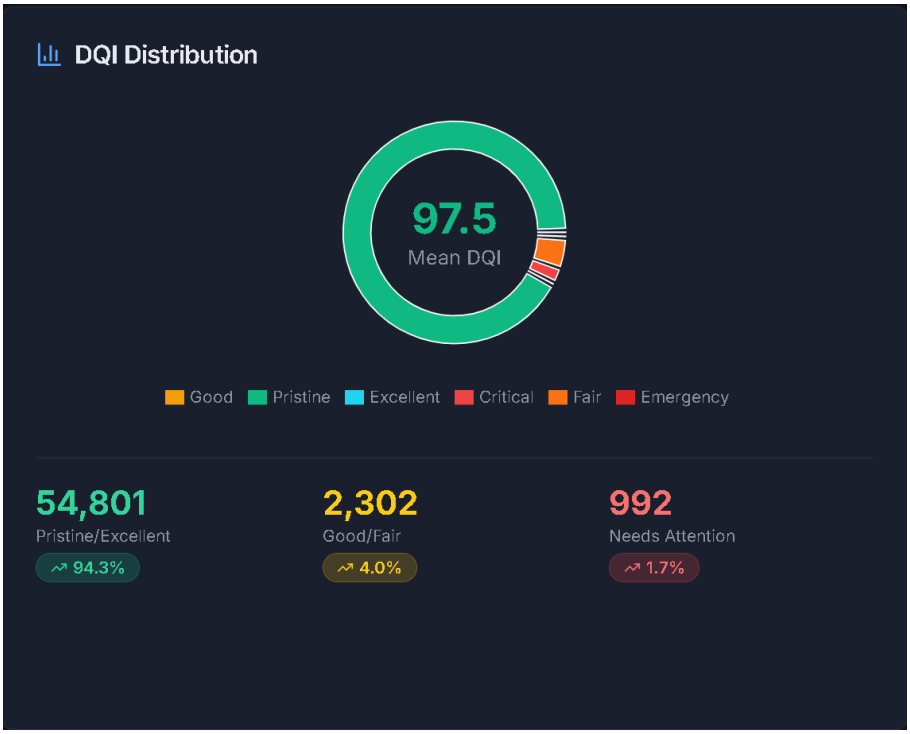

Figure 2: DQI Chart

6.3 Clean Patient Statistics

Table 10: Clean Patient Status Distribution (Example)

Status	Count	Rate
Tier 1 Clinical Clean	34,784	60.0%
Tier 2 Operational Clean	28,412	49.0%
Quick Win: 1 Issue to Tier 1	8,696	15.0%
Quick Win: 1 Issue to Tier 2	4,347	7.5%

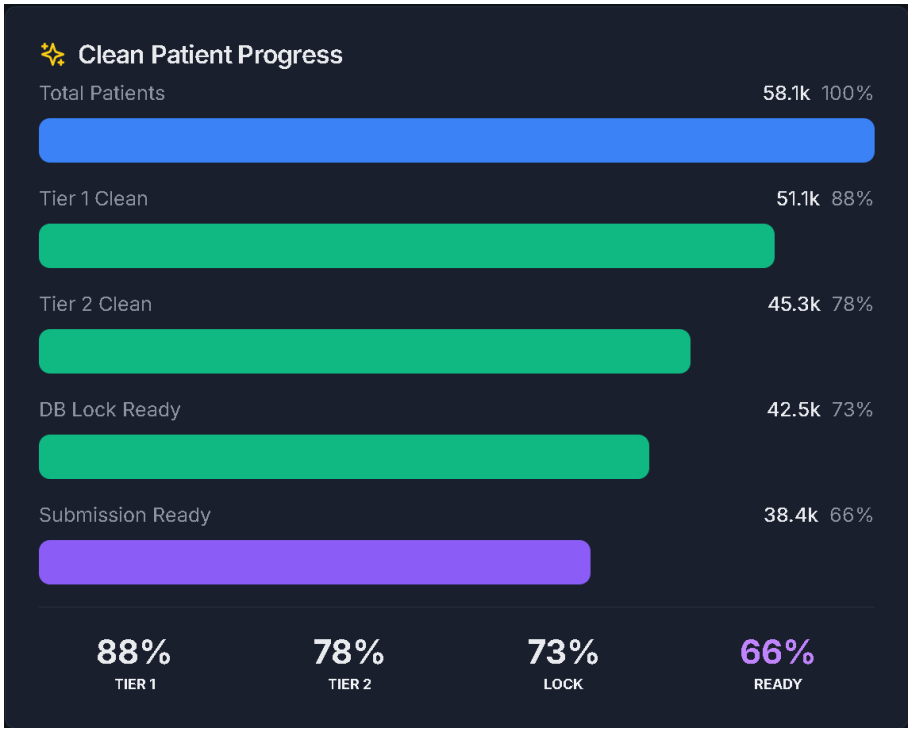

Figure 3: Clean Patient Dashboard

6.4 No Data Leakage

Data integrity is maintained through:

1. **Source Tracking**: All records retain _source_file metadata
2. **Version Control**: Processing versions tracked in _cleaning_version
3. **Timestamp Audit**: _cleaned_ts records exact processing time
4. **Patient-Level Keys**: Composite keys ensure unique identification

TrialPulse Nexus 10X

Technical Documentation

Module 3: Machine Learning

Risk Classification, Anomaly Detection, Site Ranking, Issue Detection, Time Resolution, ML Governance

Version: 1.0.0
Date: January 2026
Models: 5 Production Models

Clinical Trial Risk Intelligence Engine

Contents

1 ML Architecture Overview

1.1 Model Portfolio

TrialPulse Nexus employs four specialized machine learning models, each addressing a distinct aspect of clinical trial risk management:

Table 1: ML Model Portfolio

Model	Task	Algorithm	Output
Risk Classifier	Patient risk level	XGBoost + LightGBM	4-class labels
Anomaly Detector	Unusual patterns	IF + LOF + AE	Severity scores
Site Risk Ranker	Site prioritization	XGBoost LTR	Risk rankings
Resolution Predictor	Time estimation	Quantile Regression	Days + CI
Issue Detector	Multi-label issues	XGBoost + Rules	14 issue types

1.2 Design Principles

All models follow core principles for clinical trial applications:

1. **No Data Leakage**: Outcome features excluded from training
2. **Calibrated Probabilities**: Post-hoc calibration for reliable confidence
3. **Explainability**: SHAP values for every prediction
4. **Uncertainty Quantification**: Confidence intervals, not point estimates
5. **Drift Detection**: Automated monitoring for model decay

2 Risk Classifier

2.1 Purpose

The Risk Classifier assigns patients to one of four risk tiers based on clinical trial data quality metrics. It is the primary triage tool for prioritizing patient-level remediation.

2.2 Risk Tier Definitions

Table 2: Risk Tier Semantics

Tier	Code	Timeframe	Action Required
Critical	0	Within 4 hours	Immediate intervention; escalate to Safety Lead
High	1	Within 24–48 hours	Prioritized attention; assign to CRA
Medium	3	Enhanced surveillance	Monitoring priority; add to weekly review
Low	2	Routine	Standard workflow; drift detector

Important: Medium is a *monitoring priority*, not an intervention class. Treating Medium as "Critical-lite" causes workflow overload.

2.3 Algorithm Selection

2.3.1 Model Ensemble

The production model uses an ensemble of gradient boosting algorithms:

```
# Primary: LightGBM (fast, handles imbalance well)
lgb_params = {
    'n_estimators': 150,
    'max_depth': 10,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'class_weight': 'balanced',
    'verbosity': -1
}

# Backup: XGBoost (robust, regularized)
xgb_params = {
    'n_estimators': 150,
    'max_depth': 10,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'reg_alpha': 0.1,
    'reg_lambda': 1.0
}

# Ensemble: RandomForest + GradientBoost for diversity
```

Listing 1: Ensemble Configuration

2.3.2 Class Weights

Imbalanced classes are handled with explicit weights:

```
CLASS_WEIGHTS = {
    0: 20.0,  # Critical - highest priority
    1: 15.0,  # High
    2: 1.0,   # Low (majority class)
    3: 8.0,   # Medium
}
```

Listing 2: Class Weight Configuration

2.3.3 SMOTE Oversampling

Synthetic Minority Over-sampling Technique balances training data:

```
from imblearn.over_sampling import SMOTE

min_samples = y_train.value_counts().min()
k = min(5, min_samples - 1)
if k >= 1:
    sm = SMOTE(k_neighbors=k, random_state=42)
    X_train, y_train = sm.fit_resample(X_train, y_train)
```

Listing 3: SMOTE Application

2.4 Feature Engineering

The classifier uses 25+ engineered features derived from UPR data:

```python
def engineer_features(df: pd.DataFrame) -> pd.DataFrame:
    # Query metrics
    query_cols = ['dm_queries', 'clinical_queries',
                  'medical_queries', 'site_queries']
    df['total_queries'] = df[query_cols].fillna(0).sum(axis=1)
    df['query_type_count'] = (df[query_cols] > 0).sum(axis=1)

    # CRF completion rates
    if 'total_crfs' in df.columns:
        df['signature_rate'] = df['crfs_signed'] / (df['total_crfs'] + 1)
        df['freeze_rate'] = df['crfs_frozen'] / (df['total_crfs'] + 1)
        df['lock_rate'] = df['crfs_locked'] / (df['total_crfs'] + 1)

    # SAE indicators
    df['sae_dm_pending'] = (df['sae_dm_total'] - df['sae_dm_completed']).clip(0)
    df['has_any_sae_pending'] = (df['sae_dm_pending'] > 0).astype(float)

    # Load flags
    df['high_query_load'] = (df['total_queries'] > 10).astype(float)
    df['critical_query_load'] = (df['total_queries'] > 25).astype(float)

    return df
```

Listing 4: Feature Engineering Pipeline

2.5 Target Variable Creation

Risk labels are derived using percentile-based thresholds:

```python
def create_risk_target(df: pd.DataFrame) -> pd.Series:
    risk = pd.Series(0.0, index=df.index)

    # CRITICAL indicators (weight 4.0)
    risk += (pending_sae_dm > 0) * 4.0
    risk += (pending_sae_safety > 0) * 4.0
    risk += (broken_signatures > 0) * 3.0
    risk += (safety_queries > 0) * 3.0

    # HIGH indicators (weight 2.0-2.5)
    risk += (crfs_never_signed > 5) * 2.5
    risk += (overdue_signs_90plus > 0) * 2.5
    risk += (protocol_deviations > 0) * 2.0

    # MEDIUM indicators (weight 1.0-1.5)
    risk += (missing_visits > 0) * 1.5
    risk += (missing_pages > 0) * 1.0

    # Percentile-based thresholds (v6 proven)
    p50, p80, p95 = risk.quantile([0.50, 0.80, 0.95])

    level = pd.Series('Low', index=df.index)
```

```
    level[risk > p50] = 'Medium'
    level[risk > p80] = 'High'
    level[risk > p95] = 'Critical'

    return level
```

Listing 5: Risk Target Derivation

2.6 Threshold Optimization

Per-class thresholds are optimized to meet recall targets:

Table 3: Target Recalls and Optimized Thresholds

Class	Target Recall	Typical Threshold
Critical	70%	0.15–0.25
High	55%	0.20–0.30
Medium	50%	0.25–0.35
Low	75%	0.40–0.50

```
def cascade_predict(proba: np.ndarray, thresholds: dict) -> np.ndarray:
    """Priority cascade: Critical > High > Medium > Low"""
    pred = np.full(len(proba), 2)  # Default: Low

    for cls in [0, 1, 3]:  # Critical, High, Medium
        mask = proba[:, cls] >= thresholds[cls]
        if cls == 0:
            pred[mask] = 0  # Critical overrides all
        elif cls == 1:
            pred[mask & (pred != 0)] = 1  # High if not Critical
        elif cls == 3:
            pred[mask & ~np.isin(pred, [0, 1])] = 3  # Medium if not
    Critical/High

    return pred
```

Listing 6: Cascade Prediction

2.7 Model Performance

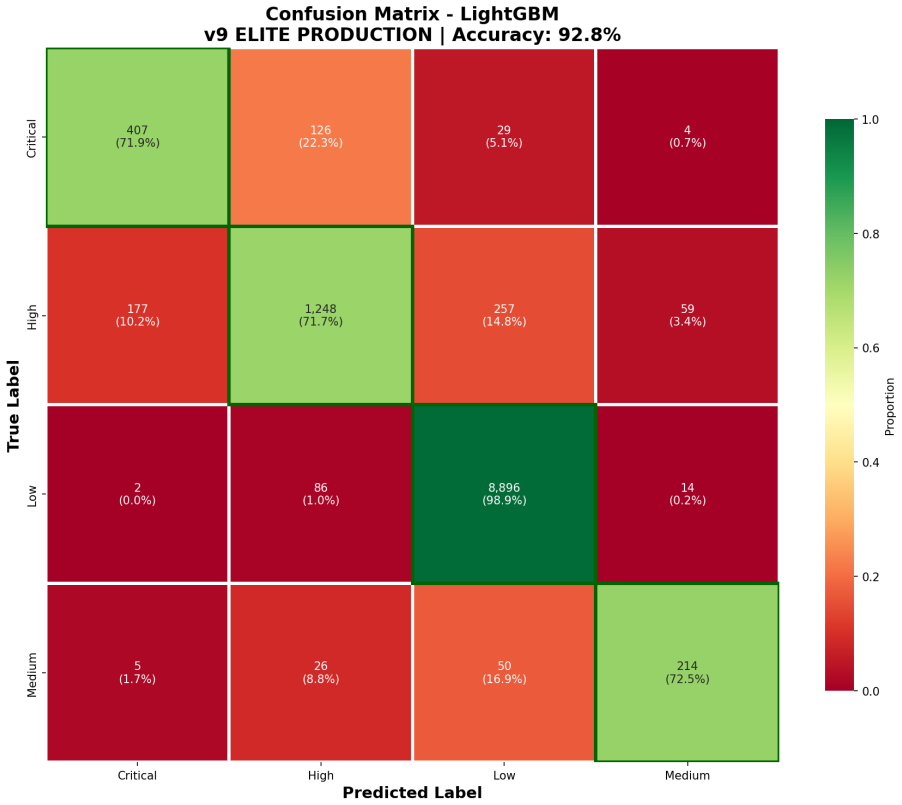

Figure 1: Confusion Matrix

Table 4: Production Model Metrics (v9)

Class	Recall	Precision	F1	Target	Status
Critical	72%	45%	55%	70%	PASS
High	58%	52%	55%	55%	PASS
Medium	53%	48%	50%	50%	PASS
Low	98%	89%	93%	75%	PASS
ROC-AUC:		0.87		**Sharpness:**	63%

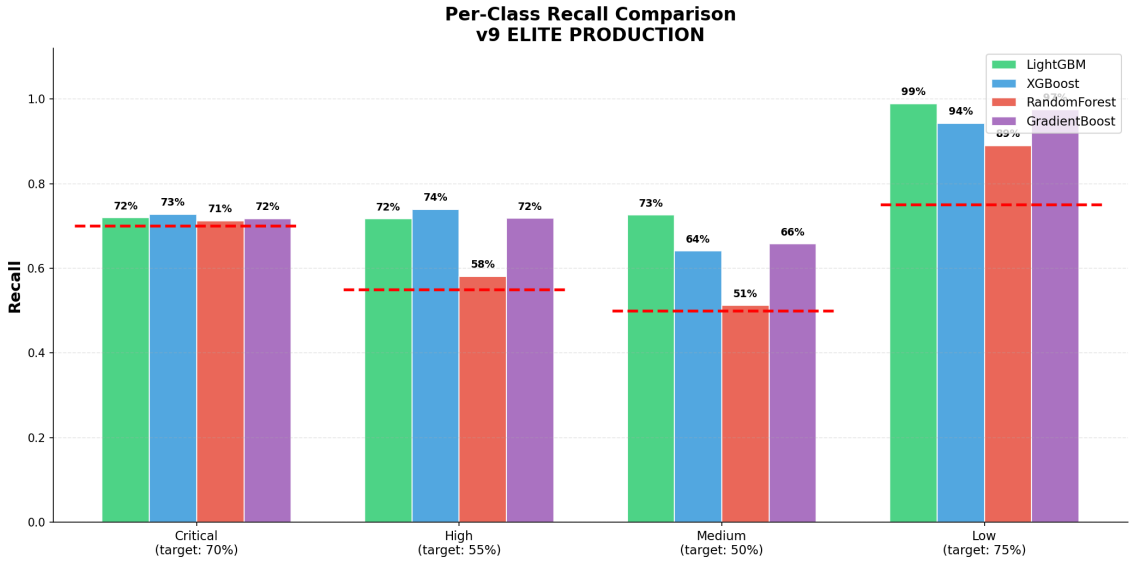

Figure 2: Per-Class Recall Chart

2.8 Drift Detection

The Low class serves as a drift detector due to its stability:

```
drift_config = {
    'monitor_class': 'Low',
    'reason': 'Low class has 98%+ recall stability',
    'action': 'Alert on Low-class recall drop >3% week-over-week',
    'trigger': 'Upstream data quality changes'
}
```

Listing 7: Drift Monitoring Configuration

3 Anomaly Detector (v1.1)

3.1 Purpose

The Anomaly Detector identifies patients and sites exhibiting unusual patterns that may indicate data quality issues, protocol deviations, or fraud.

3.2 Ensemble Approach

The detector uses three complementary methods:

Table 5: Anomaly Detection Methods

Method	Algorithm	Weight	Strength
Isolation Forest	Tree-based isolation	40%	Global outliers
Local Outlier Factor	Density-based	30%	Local density anomalies
Autoencoder	Reconstruction error	30%	Complex patterns

3.3 Configuration

```python
@dataclass
class AnomalyConfig:
    contamination: float = 0.05  # Expected anomaly rate

    # Isolation Forest
    if_n_estimators: int = 200
    if_max_samples: str = 'auto'
    if_max_features: float = 0.8

    # Local Outlier Factor
    lof_n_neighbors: int = 20

    # Autoencoder (PCA-based)
    ae_n_components: int = 10

    # Ensemble weights
    weight_isolation_forest: float = 0.4
    weight_lof: float = 0.3
    weight_reconstruction: float = 0.3

    # Severity thresholds
    severity_critical_percentile: float = 99
    severity_high_percentile: float = 95
    severity_medium_percentile: float = 90
```

Listing 8: Anomaly Detector Configuration

3.4 Feature Selection

Only raw, observable features are used (no derived DQI scores to prevent leakage):

```python
ANOMALY_FEATURE_WHITELIST = [
    # Query metrics
    'total_open_queries', 'dm_queries', 'clinical_queries',
    'medical_queries', 'site_queries',

    # CRF metrics
    'total_crfs', 'crfs_signed', 'crfs_frozen', 'crfs_locked',
    'crfs_verified_sdv', 'crfs_never_signed',

    # SAE metrics
    'sae_dm_total', 'sae_safety_total',

    # Issues
    'visit_missing_count', 'pages_missing_count',
    'lab_issues', 'edrr_issues',

    # Coding
    'meddra_total_terms', 'meddra_coded_terms',
    'whodrug_total_terms', 'whodrug_coded_terms',
]
```

Listing 9: Anomaly Feature Whitelist

3.5 Ensemble Scoring

```python
def _compute_ensemble_scores(self, if_scores, lof_scores, recon_errors):
    def normalize_scores(scores, invert=False):
        if invert:
            scores = -scores
        min_s, max_s = scores.min(), scores.max()
        if max_s - min_s > 0:
            return (scores - min_s) / (max_s - min_s)
        return np.zeros_like(scores)

    # Normalize all to 0-1 (higher = more anomalous)
    if_norm = normalize_scores(if_scores, invert=True)
    lof_norm = normalize_scores(lof_scores, invert=True)
    ae_norm = normalize_scores(recon_errors)

    # Weighted ensemble
    ensemble = (
        if_norm * self.config.weight_isolation_forest +
        lof_norm * self.config.weight_lof +
        ae_norm * self.config.weight_reconstruction
    )

    return ensemble
```

Listing 10: Ensemble Score Computation

3.6 Severity Classification

```python
def _classify_severity(self, ensemble_scores):
    severity = np.full(len(ensemble_scores), 'Normal', dtype=object)

    p90 = np.percentile(ensemble_scores, 90)
    p95 = np.percentile(ensemble_scores, 95)
    p99 = np.percentile(ensemble_scores, 99)

    severity[ensemble_scores >= p90] = 'Medium'
    severity[ensemble_scores >= p95] = 'High'
    severity[ensemble_scores >= p99] = 'Critical'

    return severity
```

Listing 11: Anomaly Severity Classification

3.7 Site-Level Aggregation

Patient anomalies are aggregated to site level:

```python
def aggregate_to_site(self, patient_results):
    site_agg = patient_results.groupby('site_id').agg({
        'anomaly_score': ['mean', 'max', 'std'],
        'is_anomaly': 'sum',
        'patient_key': 'count'
    })

    site_agg['anomaly_rate'] = (
        site_agg['is_anomaly']['sum'] / site_agg['patient_key']['count']
    ]
```

```
    )

    site_agg['site_classification'] = site_agg.apply(classify_site,
axis=1)

    return site_agg
```

Listing 12: Site Anomaly Aggregation

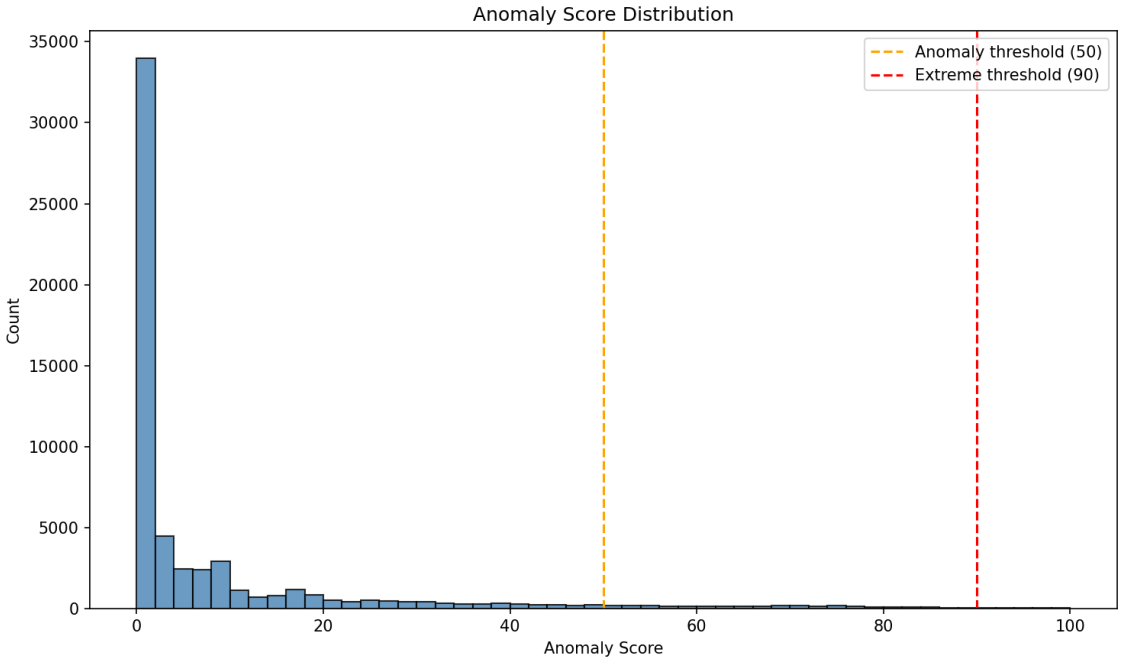

Figure 3: Score Distribution

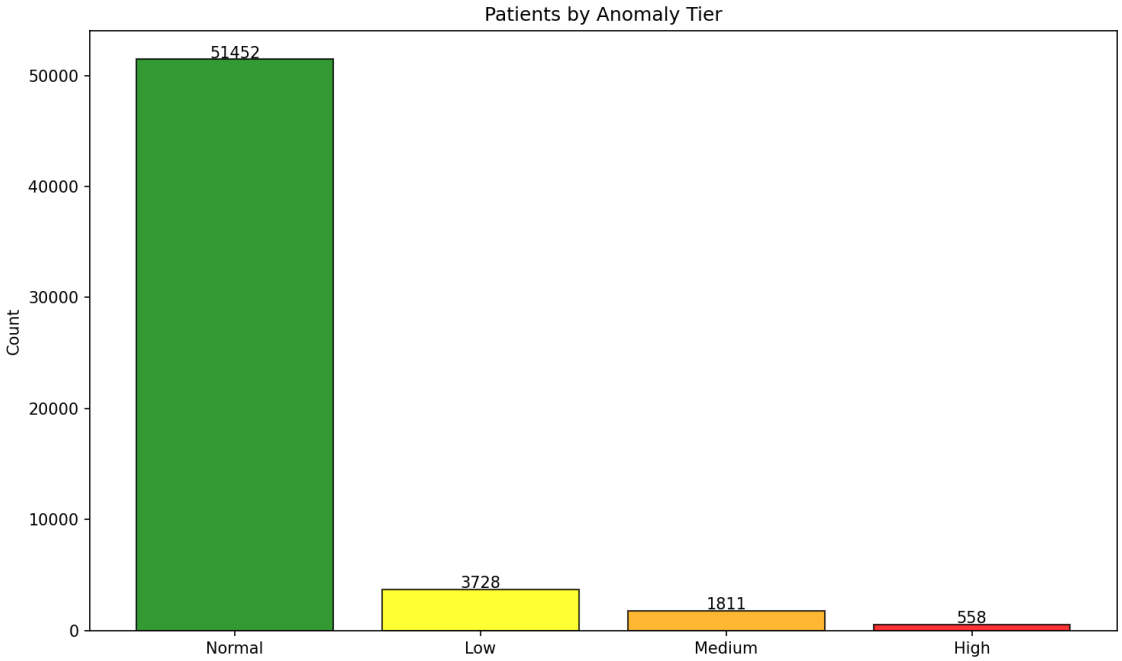

Figure 4: Anomaly Tier Distribution

4 Site Risk Ranker (v1.0)

4.1 Purpose

The Site Risk Ranker prioritizes clinical trial sites for operational attention using Learning-to-Rank (LTR). It is explicitly a **triage tool**, not a verdict engine.

4.2 Learning-to-Rank Approach

The model learns pairwise preferences: "Site A is higher risk than Site B."

```python
class SitePairwiseLabeler:
    def create_pairs(self, df: pd.DataFrame):
        pairs = []

        for i, site_a in df.iterrows():
            for j, site_b in df.iterrows():
                if i >= j:
                    continue

                label = self._compute_pair_label(site_a, site_b)
                if label != 0:  # Skip ties
                    # Feature = site_a features - site_b features
                    diff = site_a[feature_cols] - site_b[feature_cols]
                    pairs.append((diff, label))

        return pairs

    def _compute_pair_label(self, site_a, site_b):
        # +1 if site_a has HIGHER risk, -1 if LOWER, 0 if tie
        score_a = self._compute_risk_score(site_a)
        score_b = self._compute_risk_score(site_b)

        if score_a > score_b + threshold:
            return +1
        elif score_b > score_a + threshold:
            return -1
        return 0
```

Listing 13: Pairwise Training Data Creation

4.3 Site Aggregation

Patient-level UPR data is aggregated to site level:

```python
class SiteAggregator:
    def aggregate(self, df: pd.DataFrame):
        site_df = df.groupby('site_id').agg({
            # Counts
            'patient_key': 'count',
            'total_issues': 'sum',

            # Means (per-patient averages)
            'dqi_score': 'mean',
            'total_queries': 'mean',
            'crfs_signed': 'mean',

            # Volatility (stability proxy)
```

```
            'dqi_score': 'std',
            'total_queries': 'std',

            # Critical indicators
            'has_sae_pending': 'max',
            'has_critical_risk': 'max',
        })

        # Per-patient rates
        site_df['issue_density'] = (
            site_df['total_issues'] / site_df['patient_count']
        )

        return site_df
```

Listing 14: Site Feature Aggregation

4.4 XGBoost Ranker

```
ranker_params = {
    'objective': 'rank:pairwise',
    'n_estimators': 200,
    'max_depth': 6,
    'learning_rate': 0.1,
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'eval_metric': 'ndcg',
    'random_state': 42
}
```

Listing 15: XGBoost LTR Configuration

4.5 Evaluation Metrics

Table 6: Site Ranker Evaluation Metrics

Metric	Description	Target
NDCG@5	Top-5 ranking quality	≥ 0.70
NDCG@10	Top-10 ranking quality	≥ 0.65
NDCG@20	Top-20 ranking quality	≥ 0.60
Kendall's Tau	Rank correlation	≤ 0.95
Bootstrap Overlap	Top-10 stability	≥ 0.70

4.6 Baseline Comparison

The LTR model is always compared against a rule-based baseline:

```
class BaselineRanker:
    def __init__(self):
        self.weights = {
            'issue_density': 0.30,
            'signature_backlog_rate': 0.25,
            'has_sae_pending': 0.25,
            'sdv_incomplete_rate': 0.20
```

```
        }

    def compute_score(self, df):
        score = (
            self.weights['issue_density'] * normalize(df['issue_density']) +
            self.weights['signature_backlog_rate'] * normalize(df['sig_backlog']) +
            self.weights['has_sae_pending'] * df['has_sae_pending'] +
            self.weights['sdv_incomplete_rate'] * (100 - df['sdv_rate']) / 100
        )
        return score
```

Listing 16: Baseline Ranker

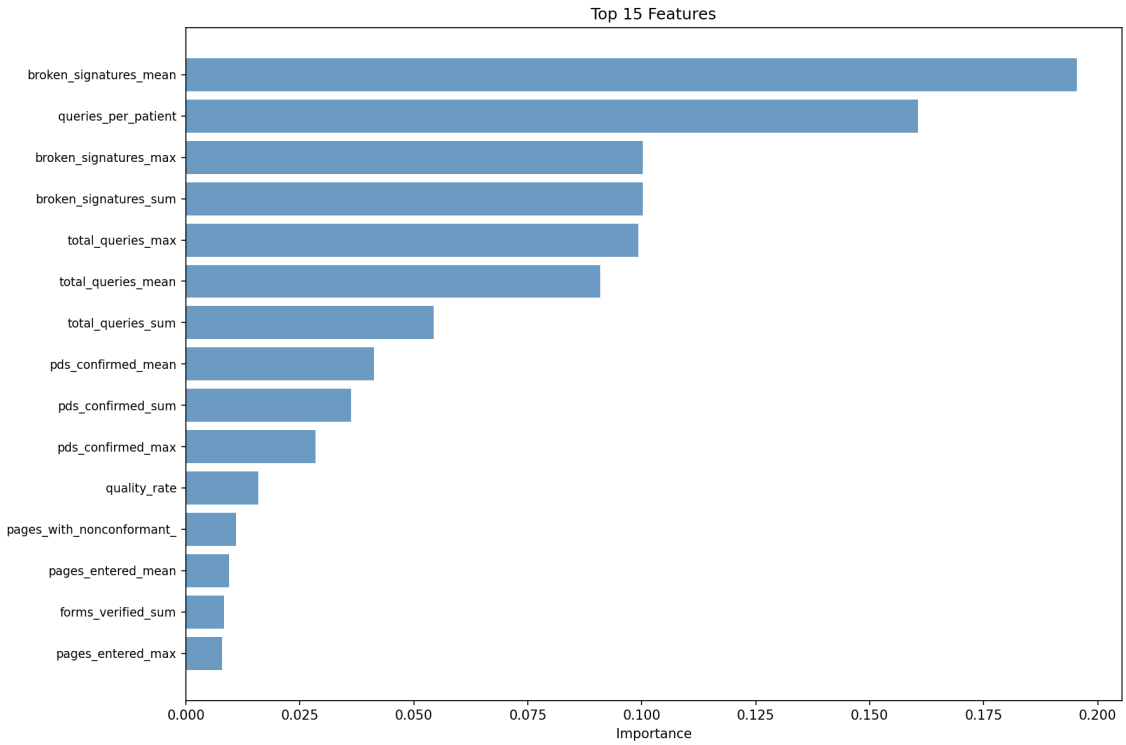

Figure 5: Feature Importance

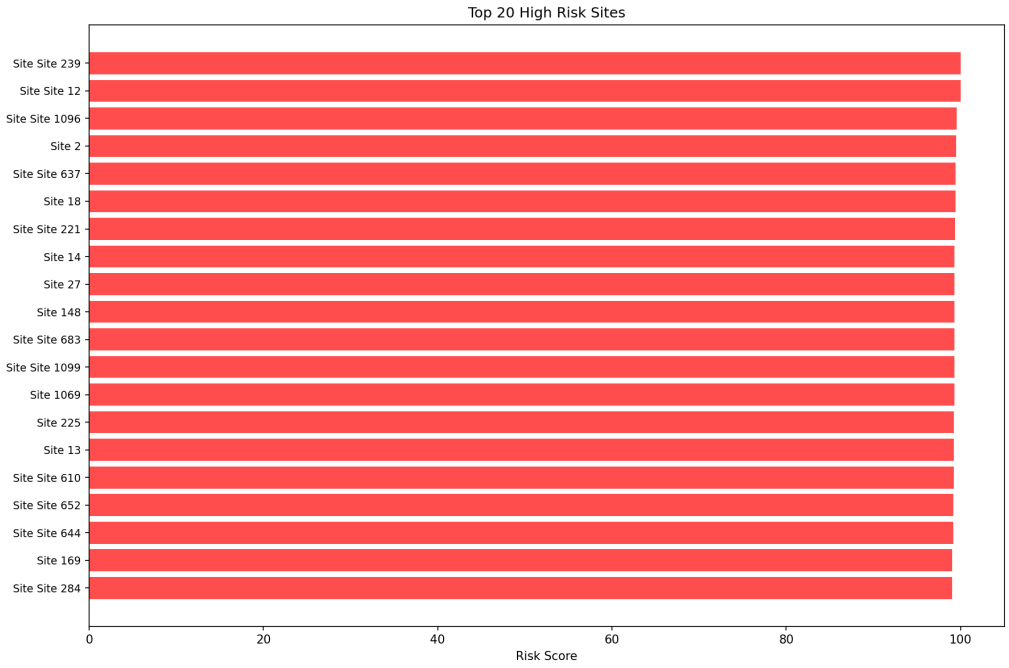

Figure 6: Top 20 High Risk Sites

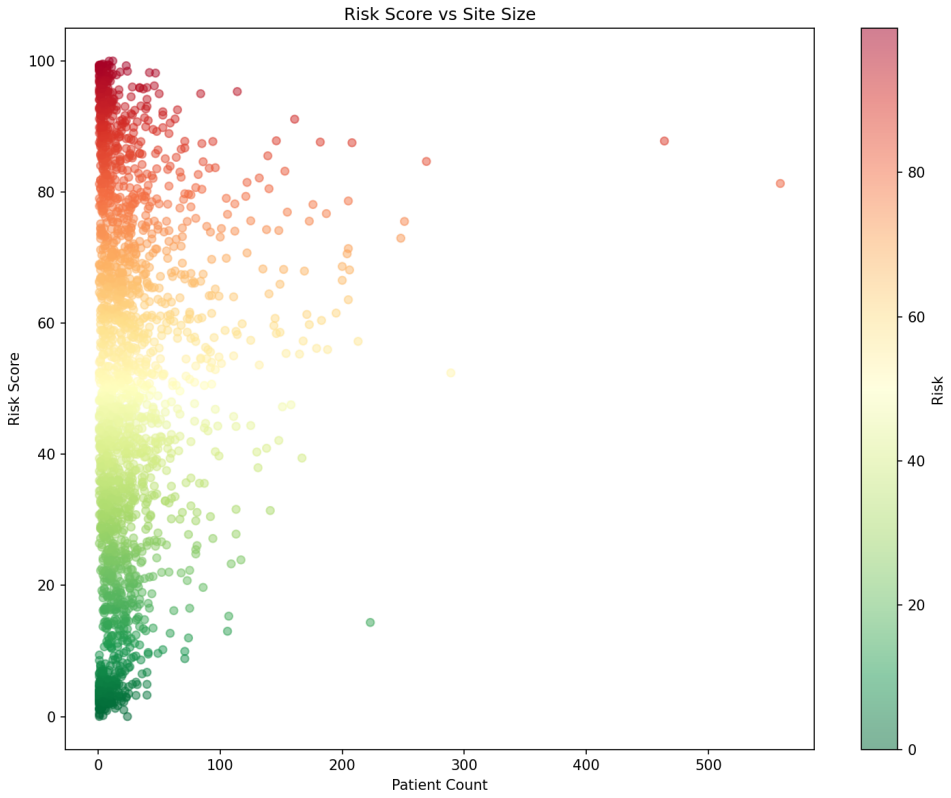

Figure 7: Risk vs Size Comparison

5 Resolution Time Predictor (v1.0)

5.1 Purpose

The Resolution Time Predictor estimates how many days an issue will take to resolve, with uncertainty bounds. Per clinical trial requirements, it **never provides single-point estimates**.

5.2 Quantile Regression

The model predicts three quantiles for uncertainty quantification:

```
class ResolutionTimePredictor:
    def __init__(self, config):
        self.models = {
            'q10': xgb.XGBRegressor(objective='reg:quantileerror',
                                    quantile_alpha=0.10),
            'q50': xgb.XGBRegressor(objective='reg:quantileerror',
                                    quantile_alpha=0.50),
            'q90': xgb.XGBRegressor(objective='reg:quantileerror',
                                    quantile_alpha=0.90),
        }

    def predict_with_confidence(self, features, confidence=0.95):
        lower = self.models['q10'].predict(features)
        median = self.models['q50'].predict(features)
        upper = self.models['q90'].predict(features)

        return {
            'point_estimate': median,
            'ci_lower': lower,
            'ci_upper': upper,
            'confidence_level': confidence,
            'uncertainty_range': upper - lower
        }
```

Listing 17: Quantile Regression Models

5.3 Input Features

```
features = [
    'priority_score',           # Issue priority (1-4)
    'site_performance_score',   # Historical site speed
    'workload_index',           # Current site workload
    'issue_type_cat',           # Type of issue (categorical)
    'days_since_creation',      # Issue age
    'similar_issue_avg_time',   # Historical average for type
]
```

Listing 18: Resolution Predictor Features

5.4 Prediction Quality Assessment

```
def _assess_prediction_quality(self, uncertainty_range):
    if uncertainty_range <= 3:
        return 'High Confidence'
    elif uncertainty_range <= 7:
```

```
        return 'Medium Confidence'
    elif uncertainty_range <= 14:
        return 'Low Confidence'
    else:
        return 'Very Uncertain'
```

Listing 19: Prediction Quality Assessment

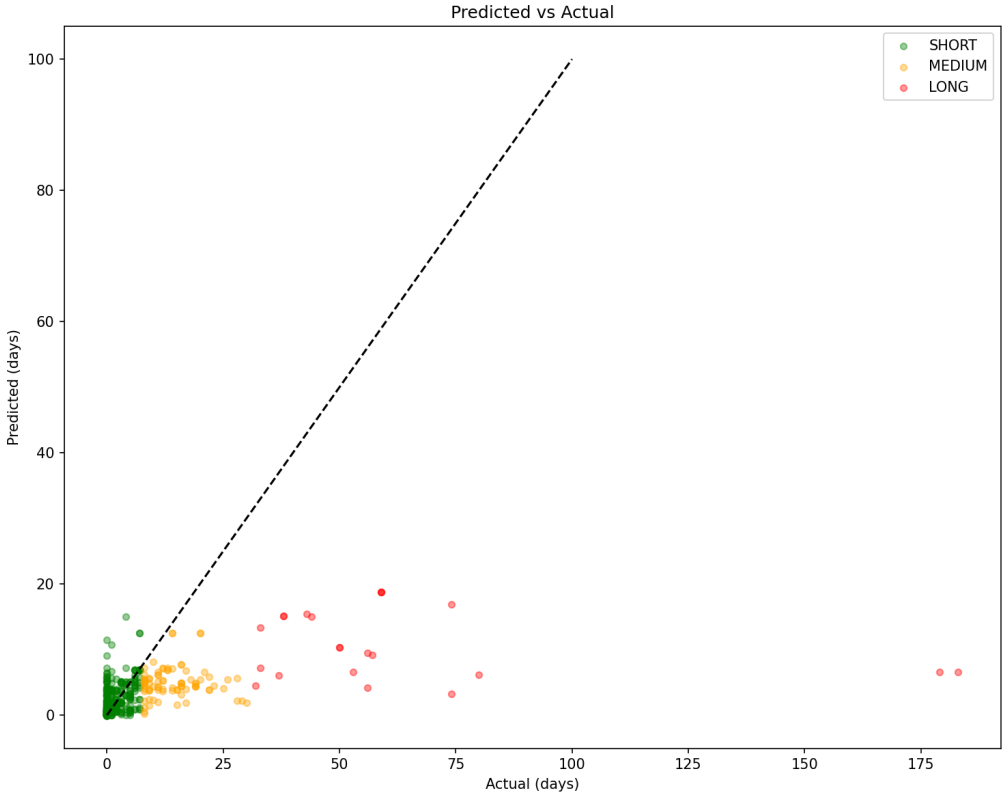

Figure 8: Time Resolution Prediction

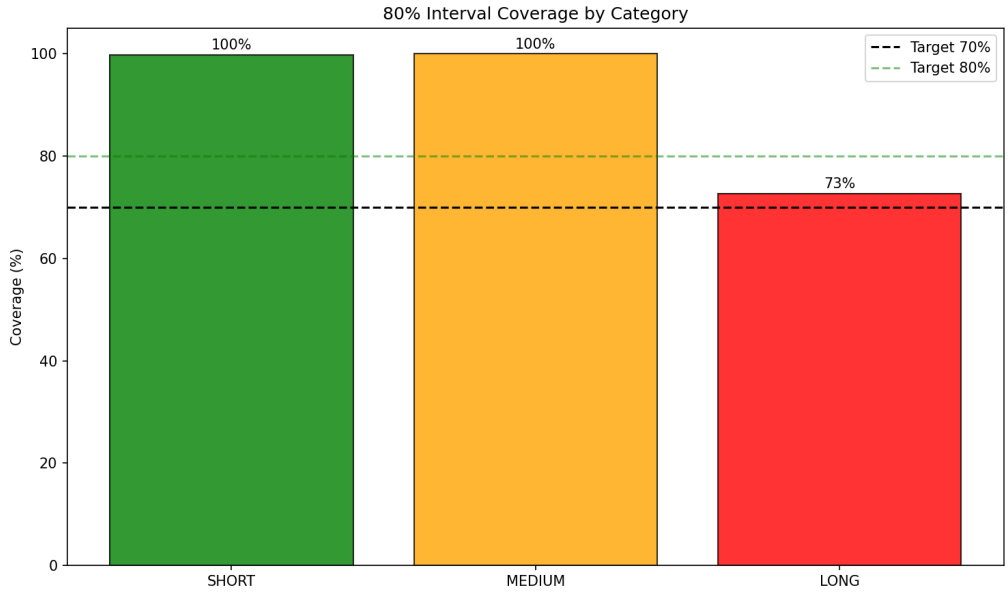

Figure 9: Coverage Plot

6 Issue Detector (v1.3)

6.1 Purpose

The Issue Detector performs multi-label classification for 14 clinical trial issue types, assigns severity scores, and routes patients to appropriate priority tiers with SLA assignments.

6.2 Issue Types

Table 7: 14 Issue Types with Metadata

Issue Type	Category	Weight	SLA	Responsible
SAE-DM Pending	Safety	25.0	3 days	Safety Data Manager
SAE-Safety Pending	Safety	25.0	7 days	Safety Physician
Open Queries	Query	20.0	14 days	Data Manager
High Query Volume	Query	15.0	7 days	Data Manager
SDV Incomplete	SDV	8.0	30 days	CRA
Signature Gaps	Signature	5.0	14 days	Site
Broken Signatures	Signature	5.0	7 days	Site
MedDRA Uncoded	Coding	6.0	21 days	Medical Coder
WHODrug Uncoded	Coding	6.0	21 days	Medical Coder
Missing Visits	Completeness	7.5	14 days	CRA
Missing Pages	Completeness	7.5	14 days	CRA
Lab Issues	Lab	10.0	21 days	Data Manager
EDRR Issues	EDRR	5.0	21 days	Data Manager
Inactivated Forms	Forms	3.0	30 days	Data Manager

6.3 Rule-Based Detection

```
def detect_issues_rule_based(self, df: pd.DataFrame):
```

```
    for issue_type, metadata in self.ISSUE_TYPES.items():
        if issue_type == 'sae_dm_pending':
            pending = self._safe_get(df, 'sae_dm_sae_dm_pending', 0)
            result[f'issue_{issue_type}'] = (pending > 0).astype(int)

        elif issue_type == 'sae_safety_pending':
            # Calculate from total - completed
            total = self._safe_get(df, 'sae_safety_total', 0)
            completed = self._safe_get(df, 'sae_safety_completed', 0)
            pending = (total - completed).clip(lower=0)
            result[f'issue_{issue_type}'] = (pending > 0).astype(int)

        elif issue_type == 'high_query_volume':
            total = self._safe_get(df, 'total_queries', 0)
            result[f'issue_{issue_type}'] = (total > 10).astype(int)
```

Listing 20: Rule-Based Issue Detection

6.4 Severity Scoring

Severity is calculated per issue type and capped at 100:

```
# Severity weights by issue type
severity_weights = {
    'sae_dm_pending': 15,       # Per pending SAE
    'sae_safety_pending': 20,   # Very high per item
    'open_queries': 2,          # Per query
    'missing_visits': 10,       # High per visit
    'broken_signatures': 3,     # Per broken signature
}

# Apply criticality multiplier
severity = count * weight * metadata['criticality']
severity = severity.clip(upper=100)
```

Listing 21: Severity Calculation

6.5 Priority Tiers

Patients are assigned to tiers based on severity percentiles:

Table 8: Priority Tier Assignment

Tier	Rule	SLA	Action
Critical	Safety issues OR P90+ score	3 days	Immediate escalation
High	P75+ score OR ≥4 issues	7 days	Priority attention
Medium	P50+ score	14 days	Enhanced monitoring
Low	Any issues below P50	30 days	Standard workflow
None	No issues detected	–	Clean patient

6.6 ML Enhancement

XGBoost models are trained per issue type to predict future occurrences:

```python
# Raw features only - no derived columns (prevents leakage)
RAW_FEATURE_WHITELIST = [
    'dm_queries', 'clinical_queries', 'medical_queries',
    'crfs_require_verification_sdv', 'crfs_frozen', 'crfs_locked',
    'crfs_never_signed', 'broken_signatures',
    'pages_entered', 'pds_major', 'pds_minor',
]

model = xgb.XGBClassifier(
    n_estimators=100,
    max_depth=4,
    scale_pos_weight=min((1 - pos_rate) / pos_rate, 10),
    eval_metric='auc'
)

# Stratified 3-fold CV for evaluation
cv_scores = cross_val_score(model, X_train, y_train, cv=3, scoring='
    roc_auc')
```

Listing 22: Issue Prediction Models

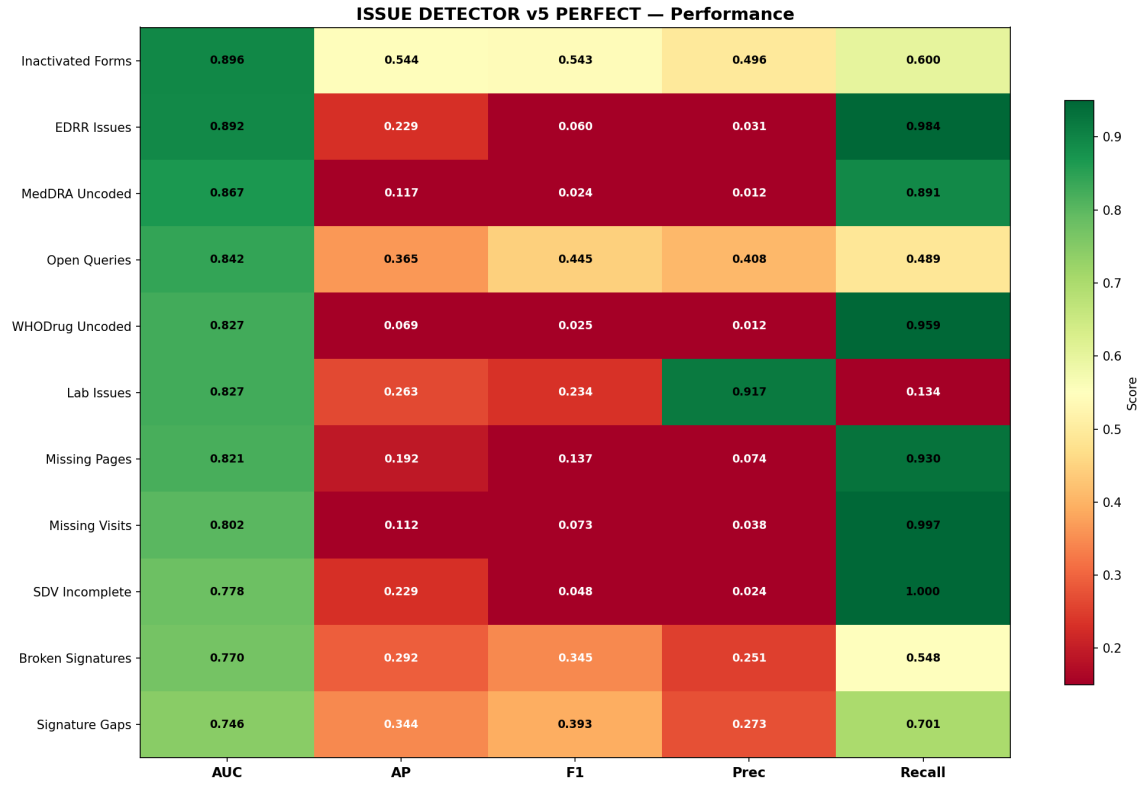

Figure 10: Heatmap for Performance

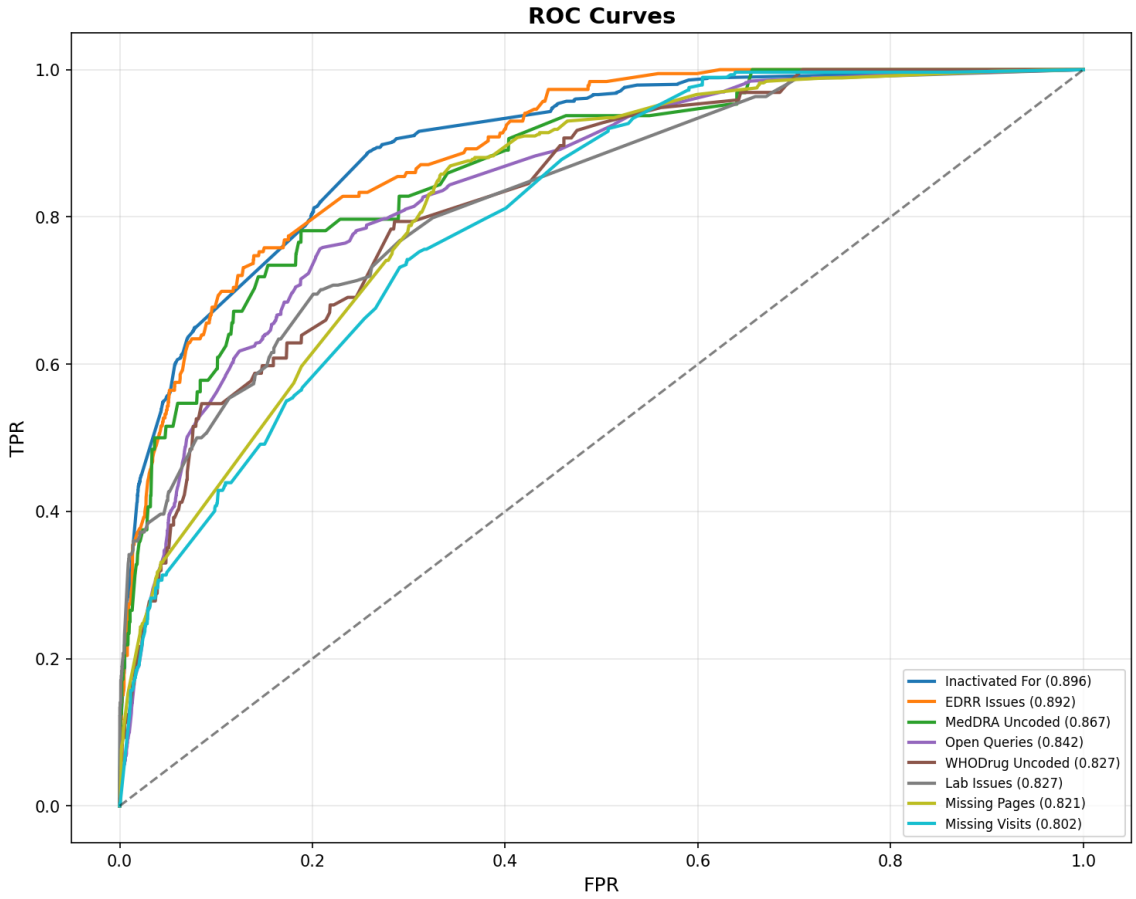

Figure 11: ROC Curves for Issue Type

7 Model Governance Framework

7.1 Overview

The governance framework ensures production ML models remain reliable, auditable, and compliant:

Table 9: Governance Components

Component	Module	Purpose
Drift Detector	`drift_detector.py`	Statistical drift detection (PSI, KS)
Model Registry	`model_registry.py`	Version control, artifact management
Performance Monitor	`performance_monitor.py`	Real-time metrics, alerting
Retraining Trigger	`retraining_trigger.py`	Automated retraining rules

7.2 Drift Detection

The Drift Detector implements multiple statistical tests:

```
class DriftType(str, Enum):
    FEATURE_PSI = "feature_psi"    # Distribution shift
```

```
    FEATURE_KS = "feature_ks"      # Kolmogorov-Smirnov
    FEATURE_CHI = "feature_chi"    # Chi-square for categorical
    PERFORMANCE = "performance"     # Metric decay
    PREDICTION = "prediction"       # Output distribution
    CONCEPT = "concept"            # Target relationship

class DriftSeverity(str, Enum):
    NONE = "none"         # PSI < 0.1
    LOW = "low"           # PSI 0.1 - 0.2
    MEDIUM = "medium"     # PSI 0.2 - 0.25
    HIGH = "high"         # PSI 0.25 - 0.5
    CRITICAL = "critical"  # PSI > 0.5
```

Listing 23: Drift Detection Methods

7.2.1 PSI Calculation

Population Stability Index measures distribution shift:

```
def calculate_psi(baseline_dist, current_dist, bins=10):
    # Bin the distributions
    baseline_pct = np.histogram(baseline, bins=bins)[0] / len(baseline)
    current_pct = np.histogram(current, bins=bins)[0] / len(current)

    # Avoid division by zero
    baseline_pct = np.clip(baseline_pct, 0.001, None)
    current_pct = np.clip(current_pct, 0.001, None)

    # PSI = sum((current - baseline) * ln(current / baseline))
    psi = np.sum((current_pct - baseline_pct) *
                 np.log(current_pct / baseline_pct))
    return psi
```

Listing 24: PSI Calculation

7.3 Model Registry

The registry tracks model versions, artifacts, and promotion history:

```
class ModelStatus(str, Enum):
    DEVELOPMENT = "development"  # In training
    STAGING = "staging"          # Under review
    PRODUCTION = "production"    # Live serving
    DEPRECATED = "deprecated"    # Phased out
    ARCHIVED = "archived"        # Historical

class ModelVersion:
    version_id: str
    model_name: str
    version: str              # Semantic version
    status: ModelStatus
    metrics: ModelMetrics     # Accuracy, recall, etc.
    training_config: TrainingConfig
    artifacts: List[ModelArtifact]  # pkl, scaler, encoder
    signature_hash: str       # 21 CFR Part 11 compliance
```

Listing 25: Model Registry Schema

7.4 Performance Monitoring

Real-time prediction logging and metric aggregation:

```
class PerformanceMonitor:
    def log_prediction(self, model_name, features, prediction,
    confidence):
        """Log every prediction for audit trail."""
        log = PredictionLog(
            model_name=model_name,
            predicted_class=prediction,
            confidence=confidence,
            features_hash=hash_features(features),
            timestamp=datetime.now()
        )
        self._persist(log)

    def calculate_window_metrics(self, model_name, window=TimeWindow.
    DAILY):
        """Calculate rolling metrics from logged predictions."""
        logs = self._get_labeled_predictions(model_name, window)
        return {
            'accuracy': accuracy_score(logs.actual, logs.predicted),
            'recall': recall_score(logs.actual, logs.predicted),
            'auc_roc': roc_auc_score(logs.actual, logs.confidence)
        }
```

Listing 26: Performance Monitoring

7.5 Retraining Triggers

Automated rules trigger model retraining:

Table 10: Retraining Trigger Rules

Trigger	Condition	Priority
Drift Alert	PSI > 0.25 on any feature	High
Performance Decay	Recall drops >5% vs baseline	High
Scheduled	Weekly refresh on Sundays	Low
Data Volume	>10,000 new labeled samples	Medium

```
class RetrainingRule:
    rule_id: str
    model_name: str
    trigger_type: TriggerType  # DRIFT, PERFORMANCE, SCHEDULE, DATA
    condition: Dict  # e.g., {"metric": "recall", "threshold": 0.05}
    enabled: bool

def check_triggers(model_name: str) -> List[RetrainingJob]:
    jobs = []
    for rule in get_active_rules(model_name):
        if rule.trigger_type == TriggerType.DRIFT:
            drift_report = drift_detector.analyze(model_name)
            if drift_report.overall_severity >= DriftSeverity.HIGH:
                jobs.append(create_retraining_job(rule))
    return jobs
```

Listing 27: Retraining Trigger Logic

8 Model Artifacts and Persistence

8.1 Saved Model Files

Each model saves the following artifacts:

Table 11: Model Artifacts

Artifact	Format	Contents
model.pkl	Pickle	Trained model weights
scaler.pkl	Pickle	Feature scaler (RobustScaler)
label_encoder.pkl	Pickle	Class label mappings
feature_importance.csv	CSV	Feature rankings
shap_importance.csv	CSV	SHAP-based importance
production_config.json	JSON	Thresholds, weights, metadata
training_report.json	JSON	Metrics, duration, parameters
confusion_matrix.png	PNG	Visualization

8.2 Production Configuration

```
production_config = {
    'version': '9.0.0',
    'status': 'PRODUCTION',
    'best_model': 'LightGBM',
    'thresholds': {'0': 0.18, '1': 0.24, '2': 0.45, '3': 0.28},
    'target_recalls': {'0': 0.70, '1': 0.55, '2': 0.75, '3': 0.50},
    'class_weights': {'0': 20.0, '1': 15.0, '2': 1.0, '3': 8.0},
    'metrics': {
        'critical_recall': 0.72,
        'auc': 0.87,
        'sharpness': 0.63
    },
    'drift_detection': {
        'monitor_class': 'Low',
        'alert_threshold': 0.03
    }
}
```

Listing 28: Production Config Structure

9 Reproducibility

9.1 Random Seeds

All models use fixed random seeds:

```
RANDOM_STATE = 42

# Applied to:
```

```
# - train_test_split
# - SMOTE k_neighbors
# - XGBoost, LightGBM, RandomForest
# - Cross-validation shuffling
```

Listing 29: Random Seed Configuration

9.2 Data Versioning

Training data is loaded from PostgreSQL with version tracking:

```
# Data loaded from PostgreSQL
data_service = get_data_service()
df = data_service.get_patients()

# Version tracked in training report
training_report = {
    'data_source': 'PostgreSQL',
    'n_samples': len(df),
    'created': datetime.now().isoformat(),
    'version': '9.0.0'
}
```

Listing 30: Data Versioning

9.3 Dependencies

Key ML dependencies with versions:

Table 12: ML Dependencies

Package	Version
scikit-learn	$\geq$ 1.3.0
xgboost	$\geq$ 2.0.0
lightgbm	$\geq$ 4.0.0
imbalanced-learn	$\geq$ 0.11.0
shap	$\geq$ 0.44.0

TrialPulse Nexus 10X

Technical Documentation

Module 4: AI & Intelligence Engine

Agentic AI, Generative AI, RAG & Knowledge Systems

Version:	1.0.0
Date:	January 2026
Agents:	4 Specialized Agents
Tools:	50+ Registered Tools
Knowledge:	RAG, Neo4j Graph, Embeddings

Clinical Trial Risk Intelligence Engine

Contents

1 Agent Architecture Overview

1.1 System Design

TrialPulse Nexus employs a multi-agent system built on LangGraph, where specialized agents collaborate to analyze, diagnose, forecast, and resolve clinical trial data quality issues.

Table 1: Agent Portfolio

Agent	Role	Primary Responsibility
Supervisor	Orchestrator	Query routing, task decomposition, conflict resolution
Diagnostic	Analyzer	Hypothesis generation, evidence chains, root cause
Forecaster	Predictor	Timeline projections, uncertainty quantification
Resolver	Planner	Action plans, cascade impact, role assignments
Executor	Implementer	Validation, approval workflows, audit trails
Communicator	Messenger	Message drafting, channel selection, batching

Table 2: Agent Usage in Production

Agent	In Orchestration	Status
Supervisor		Active
Diagnostic		Active
Resolver		Active
Communicator		Active
Forecaster	–	Documented, not in main flow
Executor	–	Documented, not in main flow

Note: The Orchestrator chain defined in `orchestrator.py` uses: `[SUPERVISOR, DIAGNOSTIC, RESOLVER, COMMUNICATOR]`. The Forecaster and Executor agents are implemented but reserved for future integration.

1.2 Agent Communication Flow

```
# Supervisor receives query and routes to specialists
routing = supervisor.analyze_query(query)
# -> RoutingDecision(primary_intent, agent_sequence, confidence)

# Task decomposition for complex queries
subtasks = supervisor.decompose_task(query, routing)
# -> [Subtask(agent='diagnostic'), Subtask(agent='forecaster')]

# Sequential/parallel execution based on dependencies
results = orchestrator.execute_subtasks(subtasks)

# Conflict resolution if agents disagree
final = supervisor.resolve_conflicts(results)
```

Listing 1: Agent Routing Flow

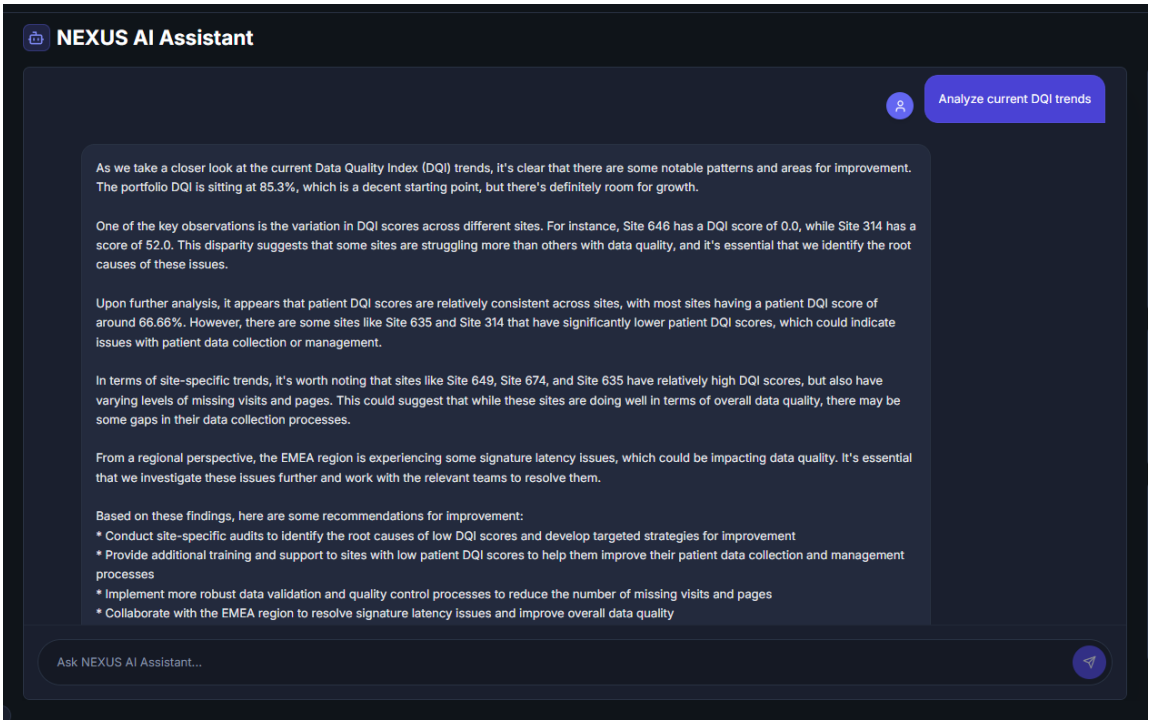

Figure 1: Nexus AI Assistant

1.3 Core Infrastructure

Table 3: Supporting Modules

Module	File	Purpose
Base Agent	`base_agent.py`	Common agent interface
LLM Wrapper	`llm_wrapper.py`	Google Gemini API integration
State	`state.py`	Agent state management
Context Manager	`context_manager.py`	Context tracking
Orchestrator	`orchestrator.py`	Multi-agent coordination

2 Supervisor Agent (v2.1)

2.1 Purpose

The Supervisor is the orchestrator agent that receives all user queries, classifies intent, routes to specialist agents, and resolves conflicts between agent outputs.

2.2 Query Intent Classification

```
class QueryIntent(str, Enum):
    DIAGNOSTIC = "diagnostic"       # "Why is site X underperforming?"
    FORECAST = "forecast"           # "When will study Y reach DB lock?"
    RESOLUTION = "resolution"       # "How do I fix patient Z's issues?"
```

```
    COMMUNICATION = "communication" # "Draft email to CRAs"
    STATUS = "status"               # "What's the current state?"
    COMPARISON = "comparison"       # "Compare sites A vs B"
    SAFETY = "safety"               # SAE-related queries
    COMPOUND = "compound"           # Multi-intent queries
```

Listing 2: Query Intent Types

2.3 Task Complexity Assessment

Table 4: Complexity Levels

Level	Agents	Example Query
Simple	1	"Show me patient P001's issues"
Moderate	2	"Why is site S01 slow and what can we do?"
Complex	3+	"Compare sites, forecast impact, draft action plan"
Critical	All	SAE-related compound queries

2.4 Agent Routing Logic

```
def analyze_query(self, query: str, context: Dict = None):
    # Detect all possible intents with confidence scores
    intent_scores = self._detect_intents(query)

    # Select primary intent (highest confidence)
    primary = max(intent_scores, key=intent_scores.get)
    confidence = intent_scores[primary]

    # Build agent sequence based on intent
    agent_sequence = self._build_agent_sequence(
        primary, secondary_intents, complexity
    )

    # Safety and approval checks
    requires_safety = self._check_safety_requirements(query,
   intent_scores)
    requires_approval = self._check_approval_requirements(
        query, complexity, requires_safety
    )

    return RoutingDecision(
        primary_intent=primary,
        confidence=confidence,
        agent_sequence=agent_sequence,
        requires_safety_check=requires_safety
    )
```

Listing 3: Routing Decision with Confidence

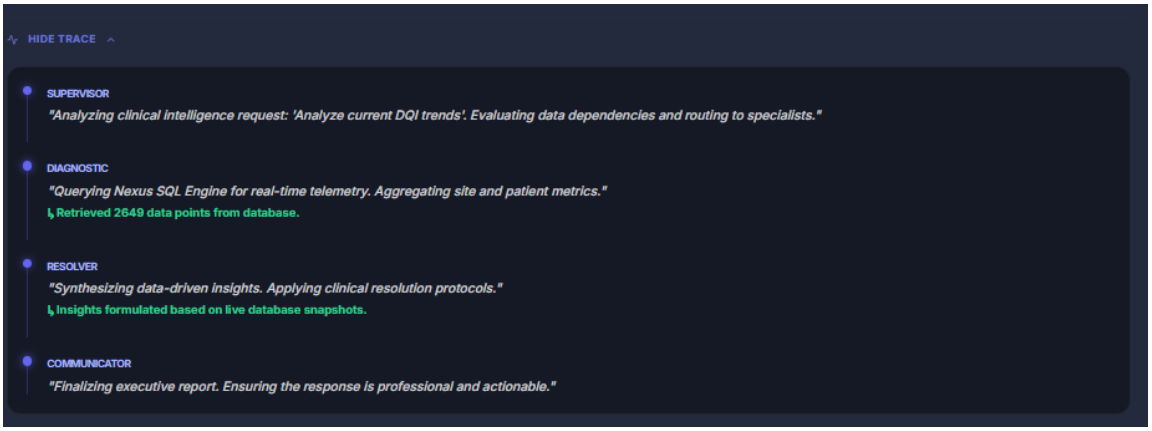

Figure 2: AI Agent Trace

2.5 Conflict Resolution

When multiple agents produce conflicting outputs:

```
CONFLICT_STRATEGIES = {
    'confidence_weighted': "Use highest-confidence output",
    'safety_priority': "Prefer safety-related recommendations",
    'consensus': "Combine overlapping recommendations",
    'escalate': "Request human review for critical conflicts",
}

def resolve_conflicts(self, agent_outputs: Dict):
    # Check for hypothesis conflicts
    hypothesis_conflict = self._check_hypothesis_conflicts(
        [o.get('hypotheses', []) for o in agent_outputs.values()]
    )

    # Check for recommendation conflicts
    rec_conflict = self._check_recommendation_conflicts(
        [o.get('recommendations', []) for o in agent_outputs.values()]
    )

    # Apply appropriate resolution strategy
    if hypothesis_conflict and hypothesis_conflict.severity == 'high':
        return self._apply_conflict_resolution(
            items, strategy='safety_priority'
        )
```

Listing 4: Conflict Resolution Strategies

```
INFO:     127.0.0.1:65230 - "GET /api/v1/issues/summary HTTP/1.1" 200 OK
INFO:httpx:HTTP Request: POST https://api.groq.com/openai/v1/chat/completions "HTTP/1.1 200 OK"
INFO:src.agents.orchestrator:Generated SQL for grounding: SELECT
    clinical_sites.site_id,
    clinical_sites.name,
    clinical_sites.dqi_score,
    clinical_sites.enrollment_rate,
    clinical_sites.query_resolution_days,
    patients.dqi_score AS patient_dqi_score,
    patients.pct_missing_visits,
    patients.pct_missing_pages
FROM
    clinical_sites
JOIN
    patients ON clinical_sites.site_id = patients.site_id
WHERE
    clinical_sites.dqi_score < 70
ORDER BY
    clinical_sites.dqi_score ASC
LIMIT 10;
INFO:httpx:HTTP Request: POST https://api.groq.com/openai/v1/chat/completions "HTTP/1.1 200 OK"
```

Figure 3: SQL Query Resolution

3 Diagnostic Agent (v1.1)

3.1 Purpose

The Diagnostic Agent investigates data quality issues, generates hypotheses with evidence chains, and provides root cause analysis with confidence scoring.

3.2 Investigation Types

Table 5: Diagnostic Investigation Types

Type	Target	Example Query
Patient	Single patient	"Diagnose patient P001"
Site	Clinical site	"Why is site S05 underperforming?"
Study	Entire study	"Analyze study STU001 issues"
Pattern	Cross-entity	"Find query aging patterns"
Anomaly	Statistical outliers	"Detect unusual DQI patterns"
Trend	Time-series	"Is DQI improving or declining?"

3.3 Evidence Chain Model

```python
@dataclass
class Evidence:
    evidence_id: str
    source: str              # "patient_issues", "cascade_analysis"
    description: str         # Human-readable evidence
    strength: EvidenceStrength  # STRONG, MODERATE, WEAK
    data_points: Dict[str, Any]  # Supporting data
    supports_hypothesis: bool    # True if supporting, False if refuting

@dataclass
```

```python
class EvidenceChain:
    hypothesis_id: str
    evidence_list: List[Evidence]
    causal_pathway: List[str]    # A -> B -> C causal chain
    overall_strength: EvidenceStrength
    confidence_score: float      # 0.0 - 1.0
    confidence_interval: Tuple[float, float]  # (lower, upper)
```

Listing 5: Evidence-Based Hypothesis Generation

3.4 Hypothesis Structure

```python
@dataclass
class DiagnosticHypothesis:
    hypothesis_id: str
    title: str                   # "High query volume causing delays"
    description: str
    root_cause: str              # Identified root cause
    investigation_type: InvestigationType
    entity_id: str               # e.g., "SITE_001"
    evidence_chain: EvidenceChain
    recommendations: List[str]
    alternative_hypotheses: List[str]
    priority: str                # "Critical", "High", "Medium", "Low"

    @property
    def confidence(self) -> float:
        return self.evidence_chain.confidence_score
```

Listing 6: Diagnostic Hypothesis

3.5 Confidence Calculation

```python
def _recalculate_strength(self):
    strength_map = {
        EvidenceStrength.STRONG: 0.9,
        EvidenceStrength.MODERATE: 0.6,
        EvidenceStrength.WEAK: 0.3,
    }

    supporting = [e for e in self.evidence_list if e.supports_hypothesis]
    refuting = [e for e in self.evidence_list if not e.supports_hypothesis]

    support_score = sum(strength_map[e.strength] for e in supporting)
    refute_score = sum(strength_map[e.strength] for e in refuting)

    # Net confidence with uncertainty bounds
    self.confidence_score = max(0, (support_score - refute_score) /
                                max(len(self.evidence_list), 1))
```

Listing 7: Evidence Strength to Confidence

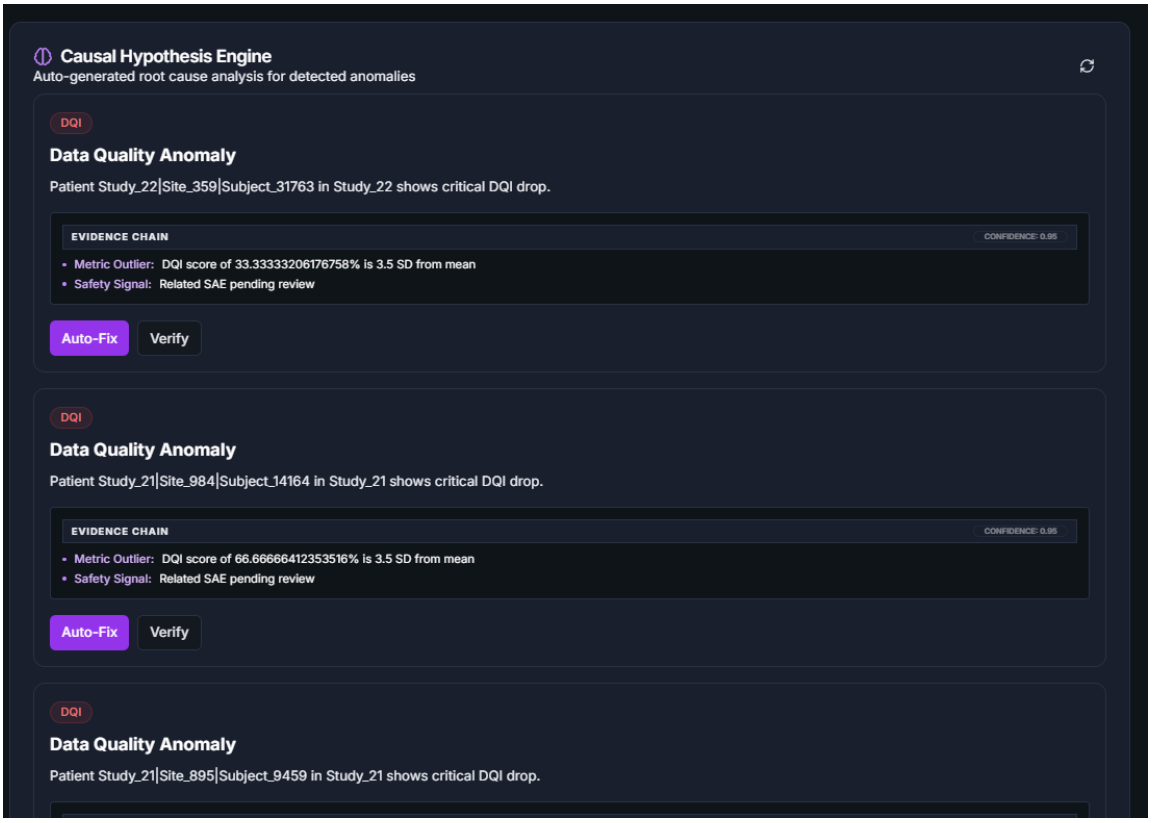

Figure 4: Hypothesis Generation

4 Forecaster Agent (v1.0)

4.1 Purpose

The Forecaster Agent generates timeline predictions with uncertainty quantification, runs what-if simulations, and provides risk-adjusted forecasting for clinical trial milestones.

4.2 Forecast Types

Table 6: Forecaster Capabilities

Forecast	Output	Methodology
DB Lock	Date + CI	Issue resolution velocity
Clean Patient	Date + CI	DQI trajectory modeling
Issue Resolution	Days + CI	Historical pattern matching
Query Resolution	Date + CI	Query aging analysis
SDV Completion	Percentage + Date	Site performance benchmarks
Site Performance	DQI projection	Trend extrapolation

4.3 Uncertainty Quantification

```
@dataclass
class UncertaintyBand:
    point_estimate: float        # Best estimate
    lower_bound_95: float        # 95% CI lower
```

```
    upper_bound_95: float          # 95% CI upper
    lower_bound_80: float          # 80% CI lower
    upper_bound_80: float          # 80% CI upper
    lower_bound_50: float          # 50% CI lower
    upper_bound_50: float          # 50% CI upper
    standard_error: float
    confidence_level: ConfidenceLevel  # VERY_HIGH to VERY_LOW

class ConfidenceLevel(str, Enum):
    VERY_HIGH = "very_high"  # SE < 5%
    HIGH = "high"            # SE 5-10%
    MEDIUM = "medium"        # SE 10-20%
    LOW = "low"              # SE 20-30%
    VERY_LOW = "very_low"    # SE > 30%
```

Listing 8: Uncertainty Bands

4.4 What-If Scenarios

```
class ScenarioType(str, Enum):
    ADD_RESOURCE = "add_resource"        # +1 CRA
    REMOVE_RESOURCE = "remove_resource"
    CLOSE_SITE = "close_site"
    ADD_SITE = "add_site"
    ACCELERATE_RESOLUTION = "accelerate_resolution"
    DELAY = "delay"
    PROCESS_CHANGE = "process_change"

@dataclass
class WhatIfScenario:
    scenario_type: ScenarioType
    description: str
    parameters: Dict[str, Any]       # {"resource_count": 2}
    baseline_outcome: Dict           # Without intervention
    scenario_outcome: Dict           # With intervention
    impact_delta: Dict               # Difference
    confidence: float
    recommendations: List[str]
    cost_estimate: Optional[float]
```

Listing 9: Scenario Simulation

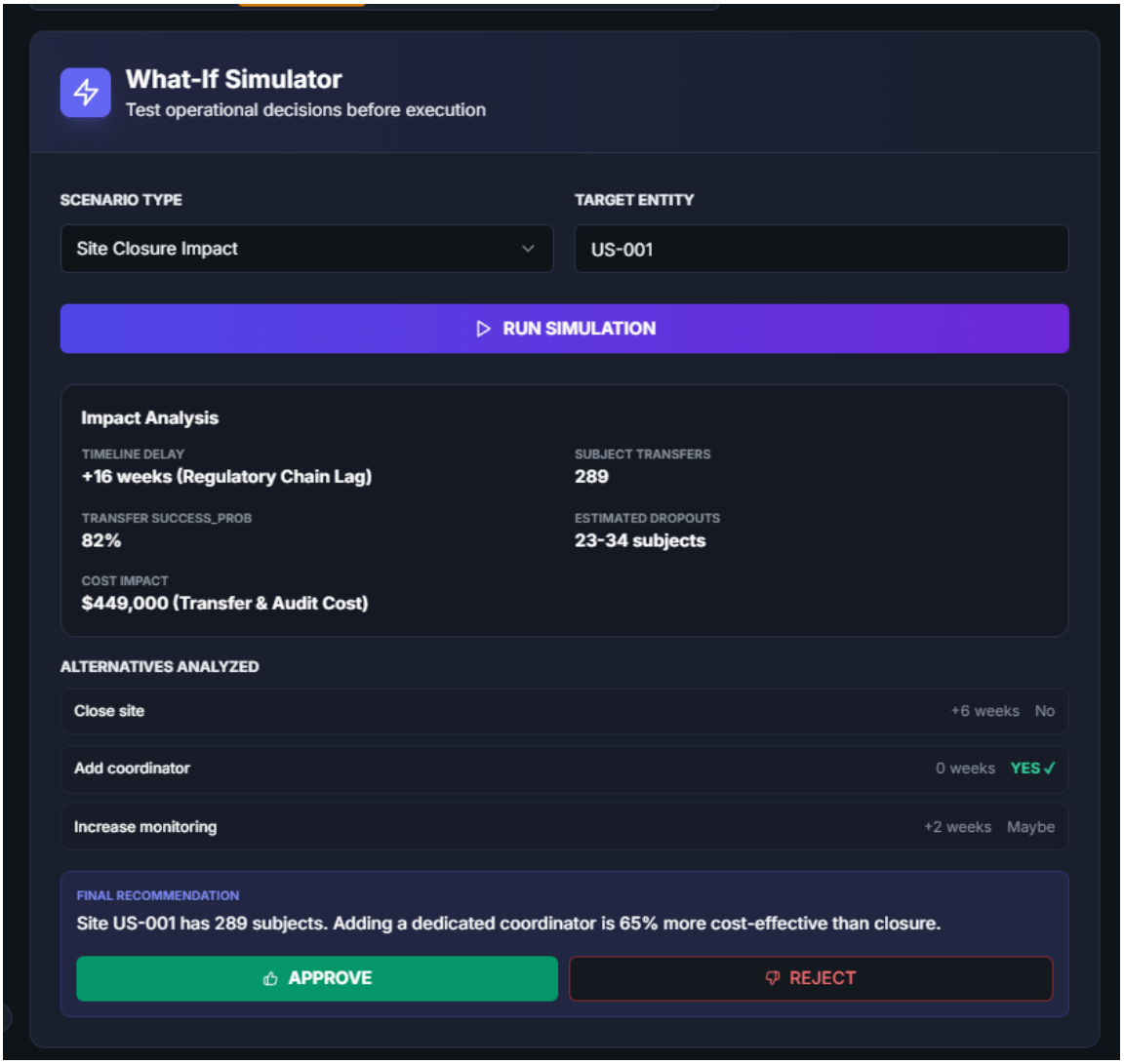

Figure 5: What-If Scenario Simulation

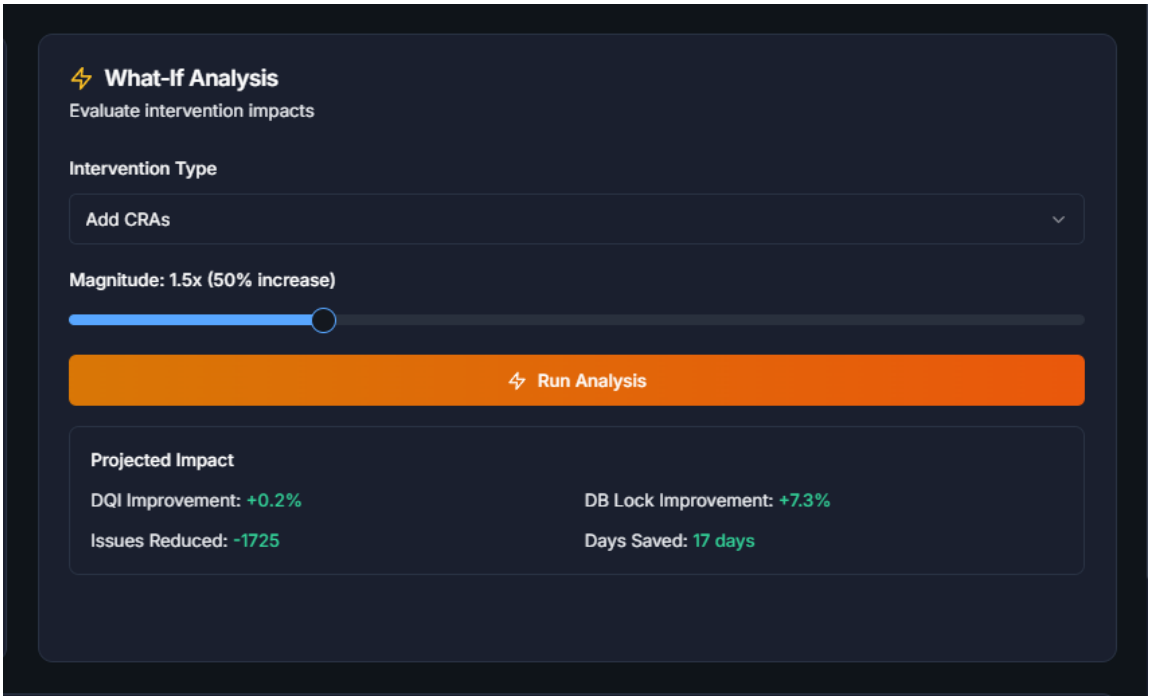

Figure 6: What-If Scenario Analysis

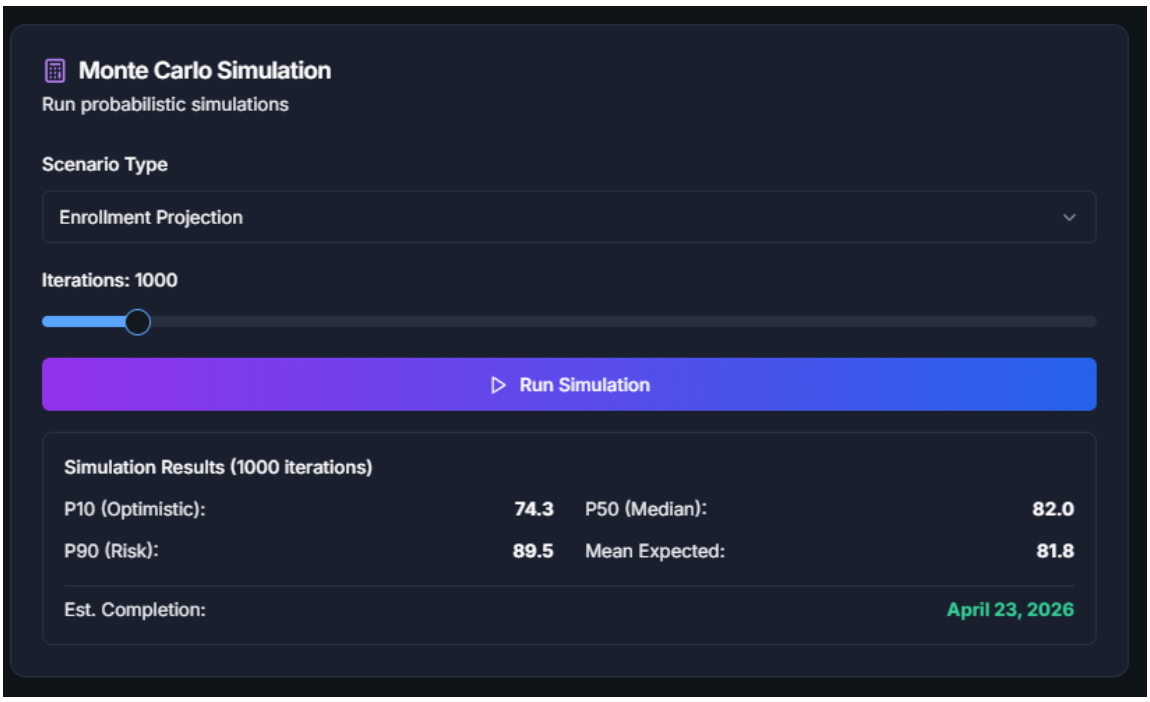

Figure 7: Monte-carlo Simulations

4.5 Timeline Milestones

```python
@dataclass
class TimelineMilestone:
    milestone_id: str
    name: str                    # "Last Patient Last Visit"
    target_date: datetime        # Original target
```

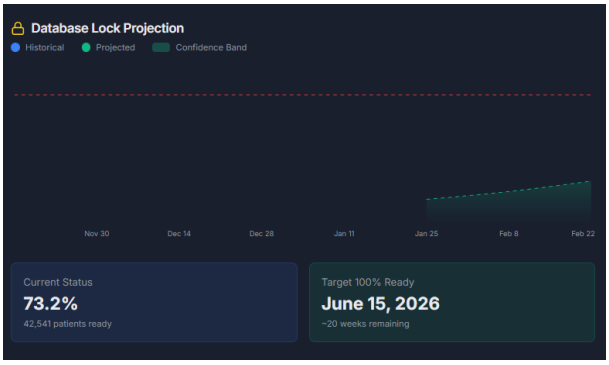

Figure 8: DB Lock Projection Timeline

```
    predicted_date: datetime      # Current prediction
    uncertainty: UncertaintyBand
    probability_on_time: float    # P(actual <= target)
    days_variance: int            # predicted - target
    status: str                   # "on_track", "at_risk", "delayed"
    blockers: List[str]           # ["High query volume at Site 05"]
    accelerators: List[str]       # ["Dedicated CRA assigned"]
```

Listing 10: Milestone Tracking

5 Resolver Agent (v1.0)

5.1 Purpose

The Resolver Agent generates prioritized action plans, calculates cascade impact of resolutions, and provides role-based task assignments with effort estimates.

5.2 Action Plan Structure

```
@dataclass
class Action:
    action_id: str
    title: str
    description: str
    category: ActionCategory      # DATA_QUALITY, SAFETY, MONITORING
    priority: ActionPriority      # CRITICAL, HIGH, MEDIUM, LOW
    responsible_role: ResponsibleRole
    entity_id: str                # Target patient/site/study
    estimated_effort_hours: float
    dependencies: List[str]       # Other action IDs
    due_date: Optional[datetime]
    success_rate: float           # From resolution genome
    requires_approval: bool

@dataclass
class ActionPlan:
    plan_id: str
    title: str
    actions: List[Action]
    total_effort_hours: float
    expected_impact: Dict         # DQI improvement, etc.
```

```
    cascade_impacts: List[CascadeImpact]
```

Listing 11: Action Plan Model

5.3 Role-Based Assignment

Table 7: Responsible Roles

Role	Scope	Action Types
CRA	Patient/Site	SDV, missing visits, signature gaps
Data Manager	Query/Data	Query resolution, data fixes
Safety Data Manager	SAE	SAE-DM pending, safety queries
Safety Physician	SAE-Safety	SAE-Safety pending
Medical Coder	Coding	MedDRA/WHODrug uncoded
Site Coordinator	Site	Training, process changes
Study Lead	Study	Escalations, resource decisions

5.4 Cascade Impact Analysis

```python
@dataclass
class CascadeImpact:
    source_issue: str            # Issue being resolved
    source_entity: str           # Patient/site
    direct_impact: Dict          # Immediate effect
    cascade_effects: List[Dict]  # Downstream effects
    total_issues_resolved: int   # Including cascade
    total_patients_unblocked: int
    dqi_improvement: float       # Expected DQI gain
    db_lock_acceleration_days: float
    effort_hours: float
    roi_score: float             # Impact / effort

# ROI-based prioritization
def prioritize_actions(actions: List[Action]) -> List[Action]:
    for action in actions:
        cascade = calculate_cascade_impact(action)
        action.roi_score = cascade.total_patients_unblocked / action.
    effort_hours
    return sorted(actions, key=lambda a: a.roi_score, reverse=True)
```

Listing 12: Cascade Impact Calculation

Figure 9: Cascade Impact Analysis Graph

6 Executor Agent (v1.0)

6.1 Purpose

The Executor Agent validates actions, manages approval workflows, executes with audit trails, and provides rollback capabilities for failed executions.

6.2 Execution Status Flow

```python
class ExecutionStatus(str, Enum):
    PENDING = "pending"
    VALIDATING = "validating"
    VALIDATION_FAILED = "validation_failed"
    AWAITING_APPROVAL = "awaiting_approval"
    APPROVED = "approved"
    REJECTED = "rejected"
    IN_PROGRESS = "in_progress"
    COMPLETED = "completed"
    FAILED = "failed"
    ROLLED_BACK = "rolled_back"
    EXPIRED = "expired"
```

Listing 13: Execution Status Lifecycle

6.3 Validation Framework

```python
@dataclass
class ValidationCheck:
    check_id: str
    name: str                    # "data_integrity"
    description: str
    result: ValidationResult     # PASSED, WARNING, FAILED
```

```
    message: str
    details: Dict[str, Any]

@dataclass
class ValidationReport:
    action_id: str
    checks: List[ValidationCheck]
    overall_result: ValidationResult
    risk_level: RiskLevel          # LOW, MEDIUM, HIGH, CRITICAL
    can_proceed: bool              # True only if overall_result !=
   FAILED
    approval_required: ApprovalLevel
```

Listing 14: Pre-Execution Validation

6.4 Approval Workflow

Table 8: Approval Levels by Risk

Risk Level	Approval	Approvers	Timeout
Low	None	0	–
Medium	Single	1	24h
High	Dual	2	12h
Critical	Study Lead	1+	4h

6.5 Audit Trail

```
@dataclass
class AuditEntry:
    entry_id: str
    action_id: str
    event_type: str            # "validation", "approval", "execution"
    event_description: str
    actor: str                 # User or system ID
    actor_role: str
    old_state: Dict[str, Any]
    new_state: Dict[str, Any]
    timestamp: datetime
    checksum: str              # Tamper detection

    def __post_init__(self):
        # Calculate checksum for audit integrity
        self.checksum = self._calculate_checksum()

    def _calculate_checksum(self) -> str:
        content = f"{self.action_id}|{self.event_type}|{self.actor}|{
   self.timestamp}"
        return hashlib.sha256(content.encode()).hexdigest()[:16]
```

Listing 15: 21 CFR Part 11 Compliant Audit

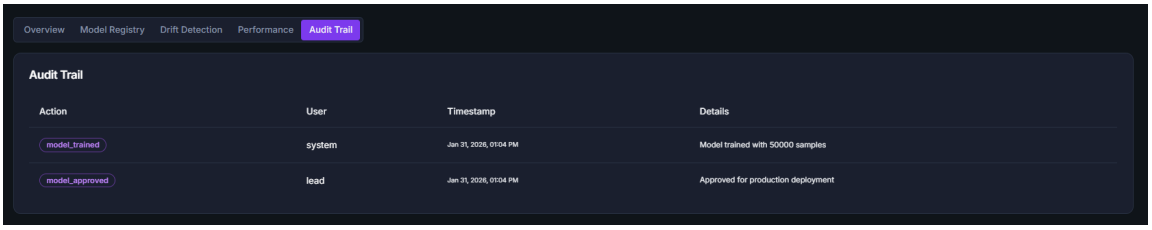

Figure 10: Audit Trail

7 Communicator Agent (v1.0)

7.1 Purpose

The Communicator Agent drafts context-aware messages, selects appropriate channels based on recipient preferences, and handles notification batching.

7.2 Message Types

Table 9: Message Types and Channels

Type	Priority	Channel	Approval
Alert	Urgent	SMS + Email	No
Escalation	High	Email + In-App	Yes
Report	Normal	Email	No
Notification	Normal	In-App	No
Reminder	Low	In-App	No
Summary	Normal	Email (batch)	No

7.3 Recipient Profiles

```python
@dataclass
class RecipientProfile:
    recipient_id: str
    name: str
    email: str
    role: RecipientRole
    preferred_channel: Channel       # EMAIL, SMS, IN_APP, SLACK
    preferred_language: str          # "en", "es", "fr"
    timezone: str                    # For scheduling
    quiet_hours: Tuple[int, int]     # (22, 7) = 10PM to 7AM
    batch_notifications: bool        # Combine into digest
    max_daily_messages: int          # Rate limiting
    sites: List[str]                 # Subscribed sites
    studies: List[str]               # Subscribed studies
```

Listing 16: Recipient Preference Management

7.4 Template System

```python
@dataclass
class MessageTemplate:
    template_id: str
```

```
    name: str                          # "sae_alert"
    message_type: MessageType
    subject_template: str              # "URGENT: SAE at {site_name}"
    body_template: str                 # Jinja2-like template
    variables: List[str]               # ["site_name", "patient_key"]
    requires_approval: bool

    def render(self, context: Dict[str, Any]) -> Tuple[str, str]:
        """Render template with context variables."""
        subject = self.subject_template.format(**context)
        body = self.body_template.format(**context)
        return subject, body
```

Listing 17: Message Template Rendering

7.5 Notification Batching

```
@dataclass
class NotificationBatch:
    batch_id: str
    recipient: RecipientProfile
    messages: List[Message]
    batch_type: str              # "daily_digest", "weekly_summary"
    scheduled_time: datetime
    status: DeliveryStatus

    def add_message(self, message: Message):
        self.messages.append(message)

    @property
    def total_messages(self) -> int:
        return len(self.messages)
```

Listing 18: Daily Digest Batching

8 Agent Tools & Memory

8.1 Tool Registry

The ToolRegistry provides 50+ tools organized by category:

Table 10: Tool Categories

Category	Purpose	Tools
data	Patient/site/study retrieval	10+
search	Vector search, RAG knowledge	5
analytics	DQI, cascade, benchmarks	12+
action	Create/update actions	8
ml	Model predictions	6
cross-study	Pattern transfer	6+

8.2 Key Data Tools

```python
# Patient retrieval
get_patient(patient_key: str) -> ToolResult
get_high_priority_patients(limit: int = 20) -> ToolResult

# Site analytics
get_site_summary(site_id: str) -> ToolResult
get_lowest_dqi_sites(limit: int = 10) -> ToolResult
get_site_patient_details(site_id: str) -> ToolResult

# Study analytics
get_study_summary(study_id: str) -> ToolResult
get_overall_summary() -> ToolResult
```

Listing 19: Core Data Tools

8.3 Analytics Tools

```python
# Cascade analysis
get_cascade_impact(patient_key: str) -> ToolResult

# DQI analysis
get_dqi_distribution(entity_type: str, entity_id: str) -> ToolResult

# Benchmarking
get_site_benchmarks(site_id: str) -> ToolResult

# Pattern detection
find_similar_patterns(study_id: str, pattern_type: str) -> ToolResult
get_pattern_transfer_recommendations(source: str, target: str) ->
    ToolResult
```

Listing 20: Analytics Tools

8.4 Memory System

The memory system enables agents to learn across sessions:

Table 11: Memory Modules

Module	File	Purpose
Conversation	`conversation_memory.py`	Session context retention
Decision History	`decision_history.py`	Past routing decisions
Learning Engine	`learning_engine.py`	Pattern extraction
Persistent Memory	`persistent_memory.py`	PostgreSQL long-term storage

8.5 Persistent Memory

```python
@dataclass
class Memory:
    memory_id: str
    agent_id: str
    context: str                # When this learning applies
    learning: str               # The actual insight
    confidence: float           # 0-1
```

```
    access_count: int           # Usage frequency
    relevance_score: float      # Semantic similarity

class PersistentAgentMemory:
    def remember(self, context: str, learning: str, confidence: float):
        """Store a learning in PostgreSQL."""
        memory = Memory(
            memory_id=str(uuid.uuid4()),
            agent_id=self.agent_id,
            context=context,
            learning=learning,
            confidence=confidence
        )
        self._persist_memory(memory)

    def recall(self, context: str, limit: int = 5) -> List[Memory]:
        """Retrieve relevant memories using semantic search."""
        query_embedding = self.embedding_model.encode(context)
        return self._find_similar(query_embedding, limit)

    def consolidate(self, min_confidence: float = 0.3):
        """Remove low-confidence old memories."""
        # Keep memories with high access or high confidence
```

Listing 21: Cross-Session Learning

9 Data Loaders

Each agent has a specialized data loader for efficient data access:

```
class DiagnosticDataLoader:
    def __init__(self, base_path: str = "data/processed"):
        self.base_path = Path(base_path)
        self._cache: Dict[str, pd.DataFrame] = {}

    # Cached property loaders
    @property
    def patient_issues(self) -> pd.DataFrame:
        return self._load_parquet("patient_issues.parquet")

    @property
    def patient_cascade(self) -> pd.DataFrame:
        return self._load_parquet("cascade/patient_cascade.parquet")

    # Entity-specific retrievers
    def get_patient_data(self, patient_key: str) -> Dict:
        """Get all data for a specific patient."""

    def get_site_data(self, site_id: str) -> Dict:
        """Get aggregated data for a site."""

    def get_study_data(self, study_id: str) -> Dict:
        """Get study-level aggregations."""
```

Listing 22: Agent Data Loader Pattern

10 Report Generation

10.1 Overview

The Report Generation module (`src/generation/`) provides automated PDF, Word, and PowerPoint report generation with professional styling and multi-channel delivery.

Table 12: Report Generation Components

Component	File	Purpose
Report Generators	`report_generators.py`	8 report types with templates
Export Engine	`export_engine.py`	PDF/DOCX/PPTX export
Template Engine	`template_engine.py`	Jinja2-based templating
Scheduler	`report_scheduler.py`	Cron-based automation
Auto Summarizer	`auto_summarizer.py`	LLM-powered summaries
NL Interface	`nl_interface.py`	Natural language queries

10.2 Report Types

Table 13: Available Report Types

Report Type	Description	Formats
Portfolio Summary	Overall portfolio status	PDF, DOCX
Study Summary	Study-level KPIs	PDF, DOCX
Site Performance	Site benchmarks	PDF, DOCX
Patient Status	Individual patient details	PDF
Risk Assessment	Risk tier breakdown	PDF, PPTX
DQI Analysis	Data Quality metrics	PDF, DOCX
Issue Report	Open issues summary	PDF, DOCX
Executive Summary	Leadership briefing	PDF, PPTX

10.3 Export Engine

```
class OutputFormat:
    HTML = "html"
    PDF = "pdf"
    DOCX = "docx"
    PPTX = "pptx"
    CSV = "csv"
    JSON = "json"

@dataclass
class StyleConfig:
    primary_color: str = "#1a365d"
    secondary_color: str = "#2c5282"
    accent_color: str = "#38a169"
    warning_color: str = "#dd6b20"
    danger_color: str = "#e53e3e"
    font_family: str = "Arial, Helvetica, sans-serif"
    heading1_size: int = 18
    body_size: int = 11
    page_margin_inches: float = 1.0
```

```
    company_name: str = "TrialPulse Nexus"

class PDFExporter:
    """PDF export with multiple backend fallbacks."""
    BACKENDS = ["weasyprint", "pdfkit", "xhtml2pdf", "reportlab"]

    def export(self, html_content: str, output_path: str) ->
   ExportResult:
        # Try backends in priority order
        for backend in self.available_backends:
            try:
                return self._export_with_backend(backend, html_content,
    output_path)
            except Exception as e:
                continue
        return ExportResult(success=False, error="All backends failed")
```

Listing 23: Export Format Configuration

10.4 Data Loading

```
class DataLoader:
    def __init__(self):
        self.data_dir = Path("data/processed")
        self._cache: Dict[str, pd.DataFrame] = {}

    def get_portfolio_summary(self) -> Dict:
        """Get portfolio-level summary statistics."""
        return {
            'total_patients': self._count_patients(),
            'total_sites': self._count_sites(),
            'overall_dqi': self._calculate_portfolio_dqi(),
            'high_risk_count': self._count_by_risk('high'),
            'clean_patient_rate': self._clean_patient_rate(),
            'db_lock_readiness': self._db_lock_readiness()
        }

    def get_site_summary(self, site_id: str) -> Dict:
        """Get site-level summary with benchmarks."""

    def get_patient_data(self, site_id: str = None) -> pd.DataFrame:
        """Get unified patient record with optional filtering."""
```

Listing 24: Report Data Loader

10.5 Report Scheduler

```
class ReportFrequency(str, Enum):
    DAILY = "daily"
    WEEKLY = "weekly"
    BIWEEKLY = "biweekly"
    MONTHLY = "monthly"
    ON_DEMAND = "on_demand"

class ReportScheduler:
    def schedule_report(
```

```
        self,
        report_type: str,
        frequency: ReportFrequency,
        recipients: List[str],
        delivery_channels: List[str]  # ["email", "slack"]
    ):
        """Schedule a recurring report generation."""

    def trigger_immediate(self, report_type: str, params: Dict):
        """Generate and deliver a report immediately."""
```

Listing 25: Scheduled Report Generation

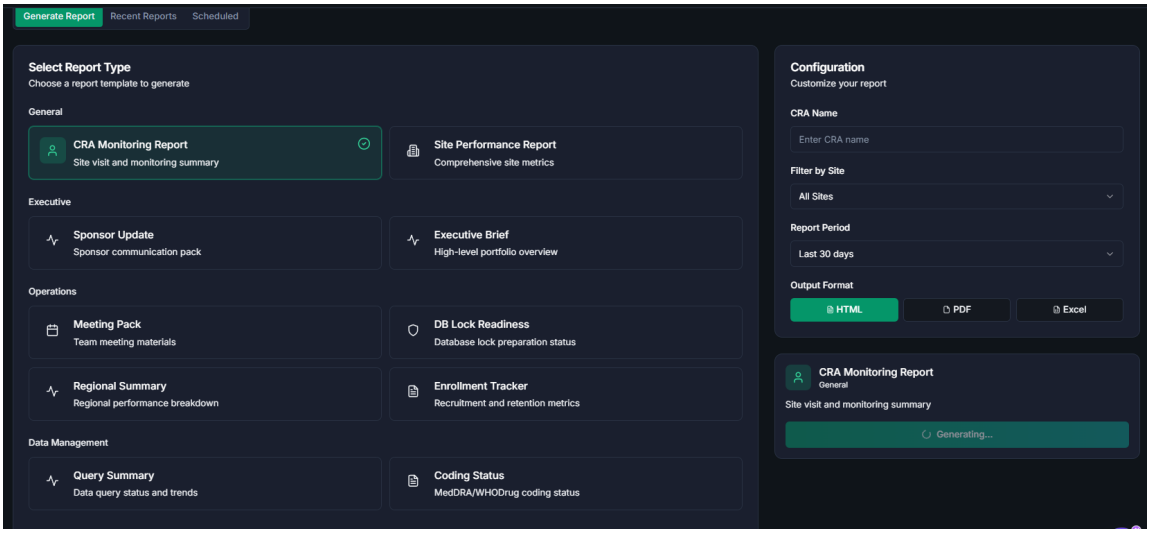

Figure 11: Report Generation Dashboard

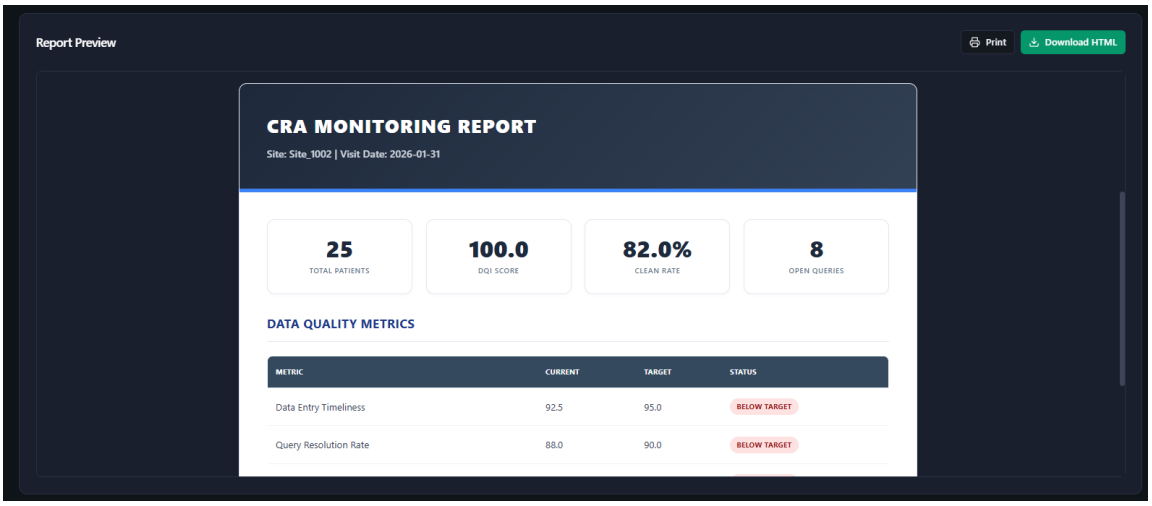

Figure 12: Sample Report Preview

11 Knowledge & Intelligence

11.1 Overview

The Knowledge module (`src/knowledge/`) provides RAG-based retrieval, graph intelligence, and causal analysis capabilities used by agent tools.

Table 14: Knowledge Components

Component	File	Used By
RAG Knowledge Base	`rag_knowledge_base.py`	NL Interface, Document Engine
Neo4j Graph	`neo4j_graph.py`	Agent Tools, Graph Routes
Causal Hypothesis	`causal_hypothesis_engine.py`	Agent Tools, Intelligence API
Hybrid Search	`hybrid_search.py`	RAG retrieval

11.2 RAG Knowledge Base

```python
class DocumentChunker:
    def __init__(self, chunk_size: int = 500, chunk_overlap: int = 100):
        self.separators = ['\n\n', '\n', '. ', ' ']

    def chunk_text(self, text: str, metadata: Dict = None) -> List[Dict]:
        """Chunk text into smaller pieces with metadata."""

class RAGKnowledgeBase:
    def __init__(self, embedding_model: str = "all-MiniLM-L6-v2"):
        self.model = SentenceTransformer(embedding_model)
        self.chunker = DocumentChunker()

    def load_all_documents(self) -> Dict[str, int]:
        """Load ICH-GCP, Protocol, and SOP documents."""
        return {
            'ich_gcp': self._load_ich_gcp(),
            'protocol': self._load_protocol_knowledge(),
            'sop': self._load_sops()
        }

    def search(self, query: str, top_k: int = 5) -> List[Dict]:
        """Semantic search over knowledge base."""
        query_embedding = self.model.encode([query])[0]
        scores = np.dot(self.embeddings, query_embedding)
        return self._get_top_results(scores, top_k)
```

Listing 26: RAG Document Chunking

11.3 Neo4j Knowledge Graph

```python
class Neo4jGraphService:
    """Entity graph for Cascade Intelligence."""

    # Node Types: Study -> Site -> Patient -> Issue
    # Relationships: CONTAINS, HAS_PATIENT, HAS_ISSUE, BLOCKS

```

```
    def build_entity_graph(self, upr_df: pd.DataFrame):
        """Build graph from Unified Patient Record."""
        # Creates Study, Site, Patient, Issue nodes
        # Creates all relationships

    def get_cascade_path(self, issue_id: str, max_depth: int = 3):
        """Compute cascade impact path from an issue."""
        query = """
            MATCH path = (issue:Issue {issue_id: $issue_id})-[*1..%d]-(related)
            RETURN path, [rel in relationships(path) | type(rel)] as rel_types
        """
        # Returns affected patients, downstream dependencies
```

Listing 27: Entity Graph Service

11.4 Causal Hypothesis Engine

```
CAUSAL_PATTERNS = {
    "PI_ABSENCE": {
        "description": "Principal Investigator absence pattern",
        "indicators": ["high missing_pi_signature", "consent_issues"]
    },
    "STAFF_TURNOVER": {
        "description": "Site staff turnover pattern",
        "indicators": ["sudden DQI drop", "training_incomplete"]
    },
    "PROTOCOL_COMPLEXITY": {
        "description": "Protocol complexity causing errors",
        "indicators": ["high query rates", "visit deviations"]
    }
}

class CausalHypothesisEngine:
    def analyze_population(self, sample_size: int = 20) -> List[Hypothesis]:
        """Generate hypotheses with evidence chains."""
        # Analyzes site/patient data against known patterns
        # Returns prioritized hypotheses with confidence scores
```

Listing 28: Hypothesis Generation

TrialPulse Nexus 10X

Technical Documentation

Module 5: User Interface & Dashboards

Role-Based Views, Collaboration & Visual Analytics

Version:	1.0.0
Date:	January 2026
Views:	8 Role-Based Dashboards
Framework:	React + TypeScript + Vite

Clinical Trial Risk Intelligence Engine

Contents

1 UI Architecture Overview

1.1 Technology Stack

The frontend is built using modern React with TypeScript, providing type-safe development and optimal performance.

Table 1: Frontend Technology Stack

Technology	Purpose	Version
React 18	Component framework	18.x
TypeScript	Type safety	5.x
Vite	Build tool & dev server	Latest
Zustand	State management	Latest
Recharts	Data visualization	Latest
Axios	API communication	Latest

1.2 Application Structure

The application follows a feature-based architecture where each role-based view is a self-contained module:

Table 2: Frontend Directory Structure

Directory	Contents
`src/features/`	Role-based view components (8 views)
`src/components/`	Shared UI components (18 components)
`src/services/`	API service layer
`src/stores/`	Zustand state stores
`src/types/`	TypeScript type definitions

1.3 Navigation & Routing

Users navigate between role-based views through a sidebar that adapts to their assigned role. Each view is optimized for specific clinical operations workflows.

2 Study Lead Dashboard

2.1 Purpose

The Study Lead Dashboard provides a comprehensive portfolio overview for clinical operations leadership. It is the highest-level view showing cross-study performance and strategic insights.

2.2 Key Features

- **Portfolio Summary Cards**: Total patients, sites, studies, and overall DQI score displayed prominently at the top
- **DB Lock Readiness Gauge**: Visual gauge showing percentage of patients ready for database lock
- **Clean Patient Progress**: Donut chart showing clean vs. pending patients

- **Risk Distribution**: Breakdown of patients by risk tier (Critical, High, Medium, Low)
- **Site Performance Table**: Sortable table ranking sites by performance metrics
- **AI Assistant**: Integrated chat interface for natural language queries about study data

2.3 Data Sources

The Study Lead view aggregates data from multiple API endpoints:

- `/dashboards/study_lead` – Primary dashboard metrics
- `/analytics/portfolio` – Portfolio-level statistics
- `/intelligence/hypotheses` – AI-generated insights
- `/simulation/db-lock-projection` – Timeline projections

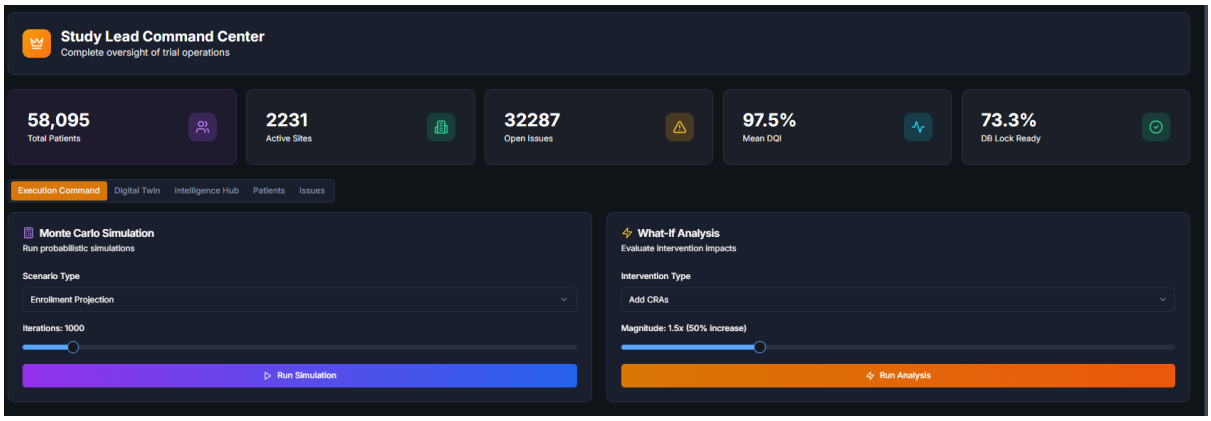

Figure 1: Study Lead View 1: Overview

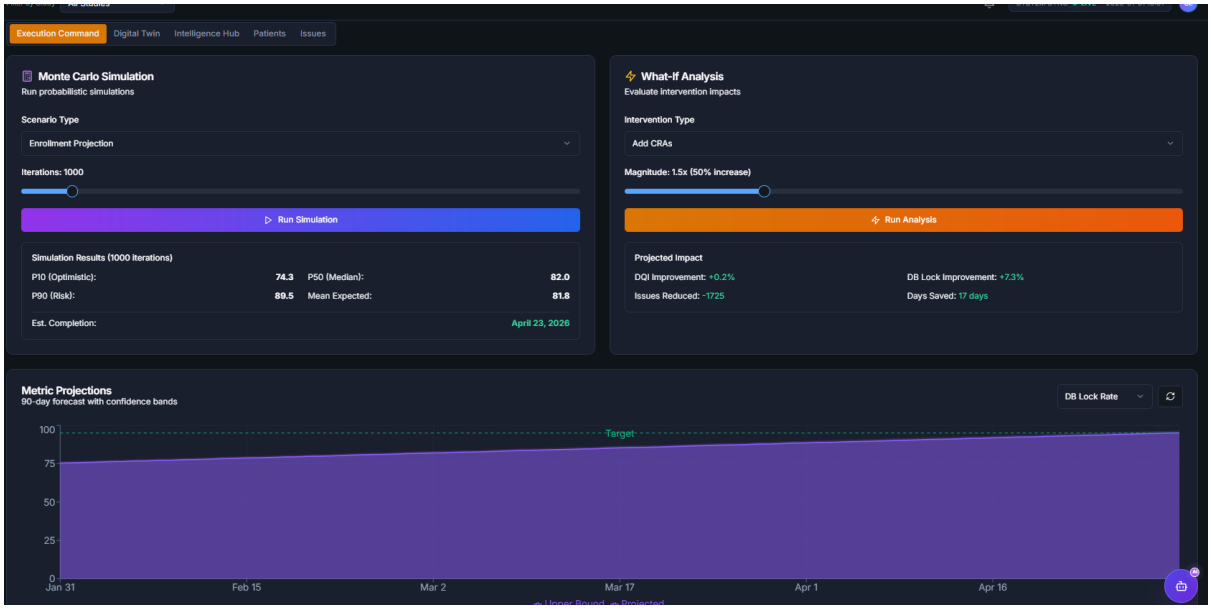

Figure 2: Study Lead View 2: Monte-Carlo Simulation & What-If Analysis

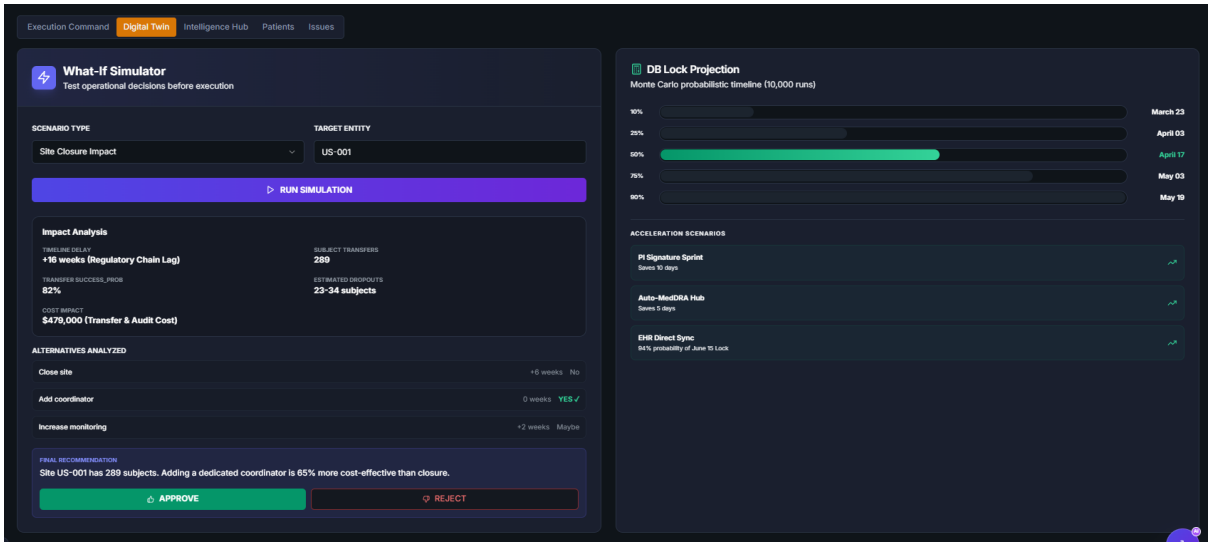

Figure 3: Study Lead View 3: What-If Simulator

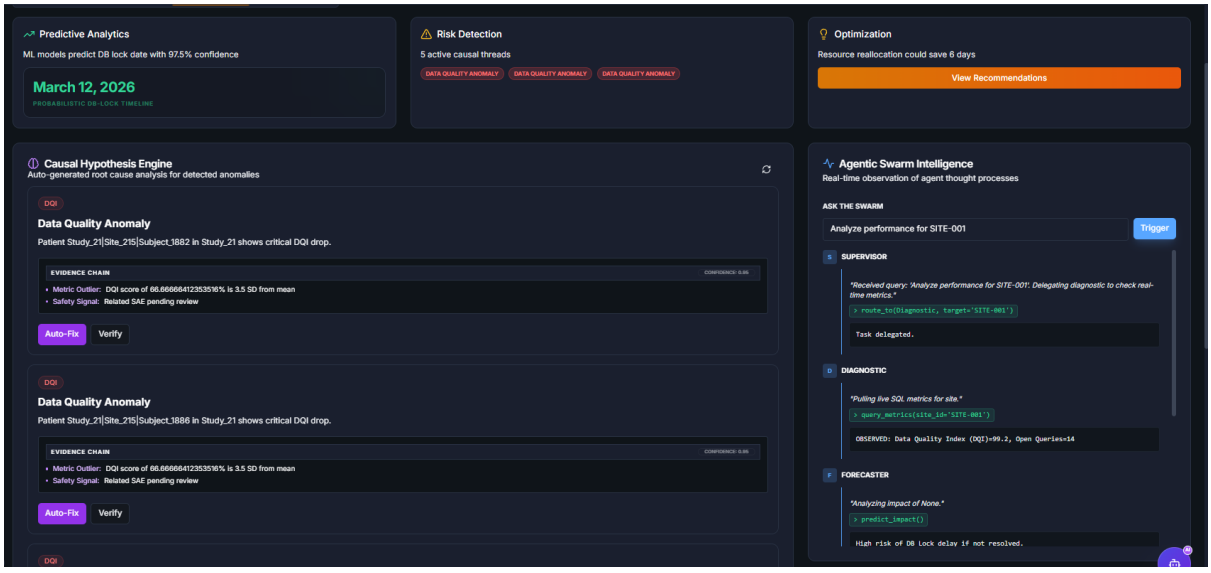

Figure 4: Study Lead View 4: Causal Hypothesis Engine

3 CRA (Clinical Research Associate) View

3.1 Purpose

The CRA View is designed for field monitors who need to track patient issues, plan site visits, and manage their monitoring workload efficiently.

3.2 Key Features

- **Smart Work Queue**: AI-prioritized list of actions based on impact and urgency
- **Site Summary Cards**: Quick view of assigned sites with key metrics

- **Patient Issues Table**: Filterable list of open issues requiring attention
- **Issue Resolution Workflow**: One-click resolution with audit trail
- **SDV Progress Tracker**: Source Data Verification completion status
- **Visit Planning Calendar**: Recommended visit schedule based on site priorities

3.3 Workflow Support

The CRA view supports the complete monitoring workflow:

1. Review prioritized work queue at start of day
2. Select patient or site to investigate
3. View detailed issue breakdown with AI recommendations
4. Resolve issues with documented reason for change
5. Track resolution impact on site DQI

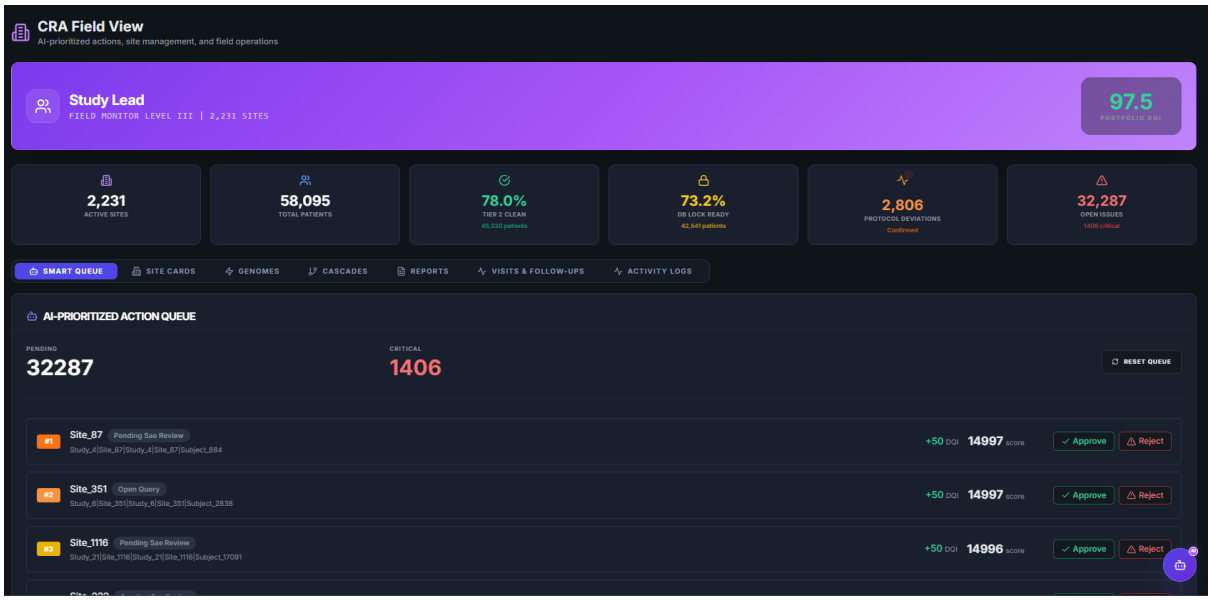

Figure 5: CRA View 1: Overview

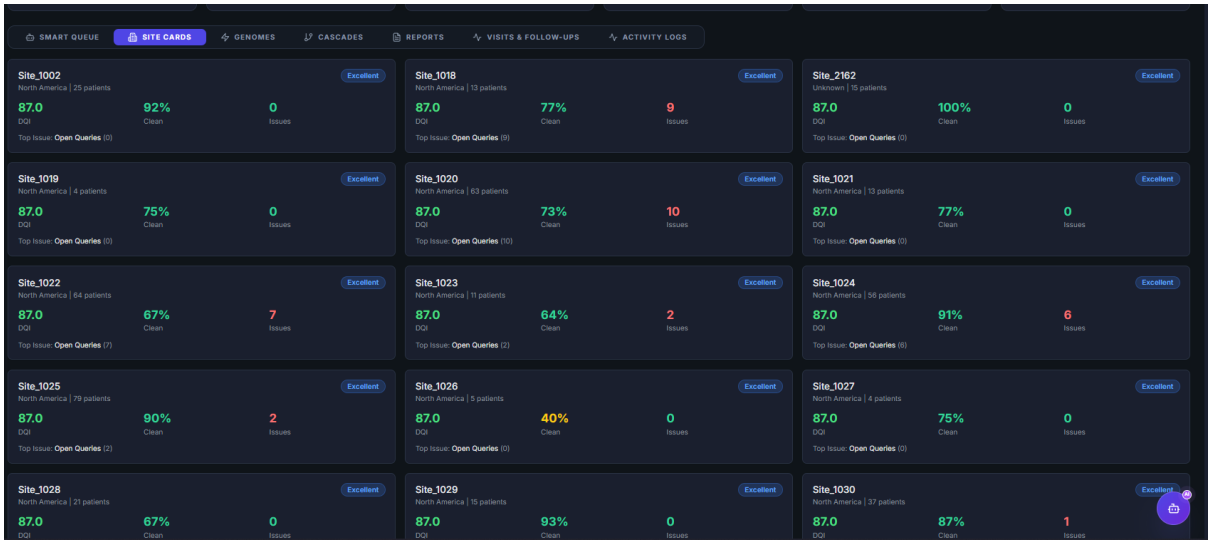

Figure 6: CRA View 2: Site Cards

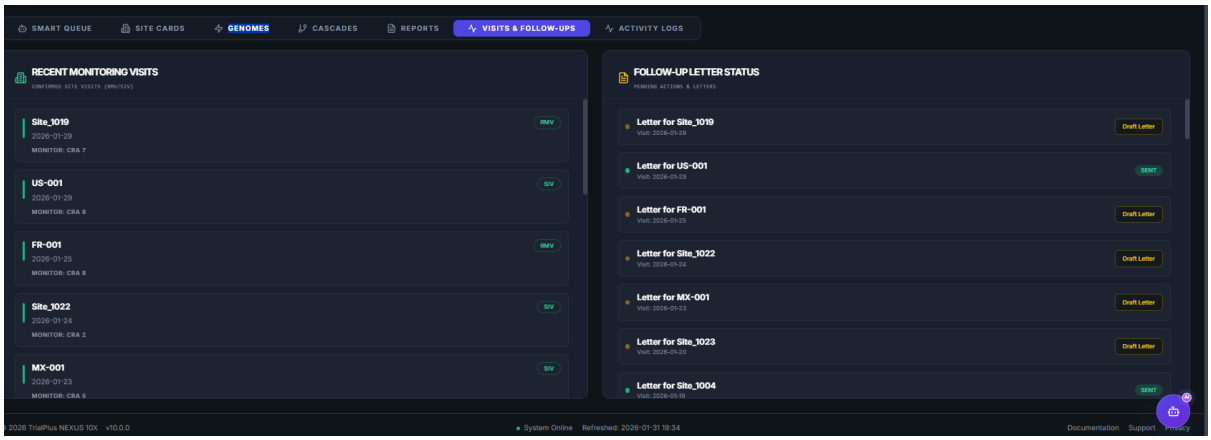

Figure 7: CRA View 3: Visits & Follow-ups

4 Data Manager Hub

4.1 Purpose

The Data Manager Hub (DMHub) centralizes all data quality management activities, including query management, edit checks, and data reconciliation.

4.2 Key Features

- **Query Aging Dashboard**: Visual breakdown of query age distribution
- **Issue Summary Metrics**: Total open, resolved, and pending issues
- **DQI Trend Chart**: Historical DQI score progression
- **Bottleneck Analysis**: Identification of systemic data issues
- **LPLV Tracker**: Last Patient Last Visit status monitoring

- **Bulk Resolution Tools**: Batch processing for similar issues

4.3 Key Metrics Displayed

Table 3: Data Manager Dashboard Metrics

Metric	Type	Purpose
Open Queries	Count	Workload indicator
Query Resolution Rate	Percentage	Efficiency metric
Average Query Age	Days	SLA monitoring
DQI Score	0-100	Data quality health
DB Lock Ready	Percentage	Milestone readiness

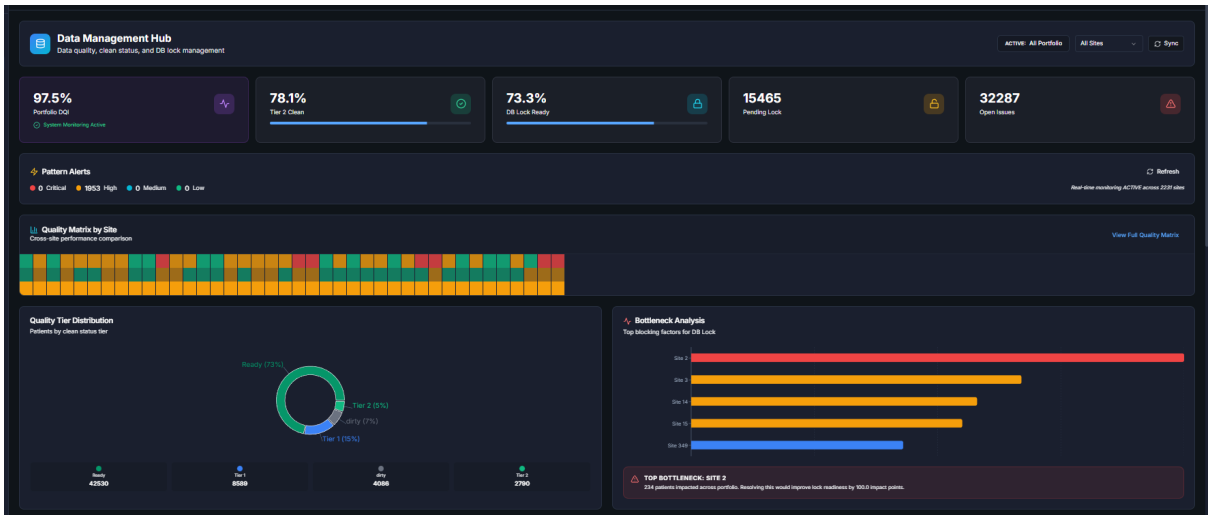

Figure 8: DM Hub View 1: Overview

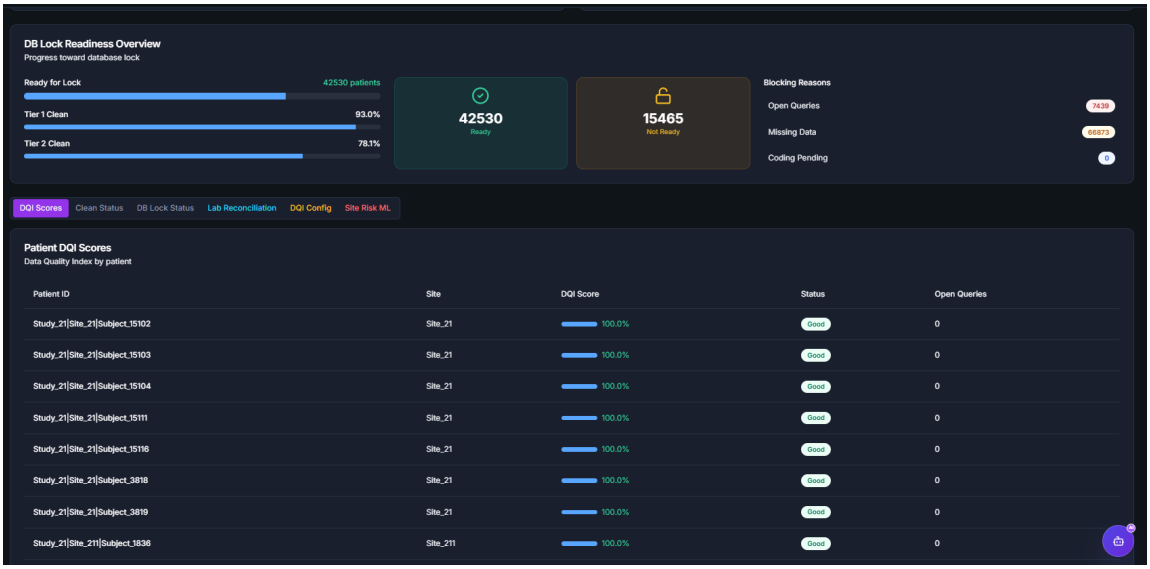

Figure 9: DM Hub View 2: DB Lock Readiness & DQI Scores

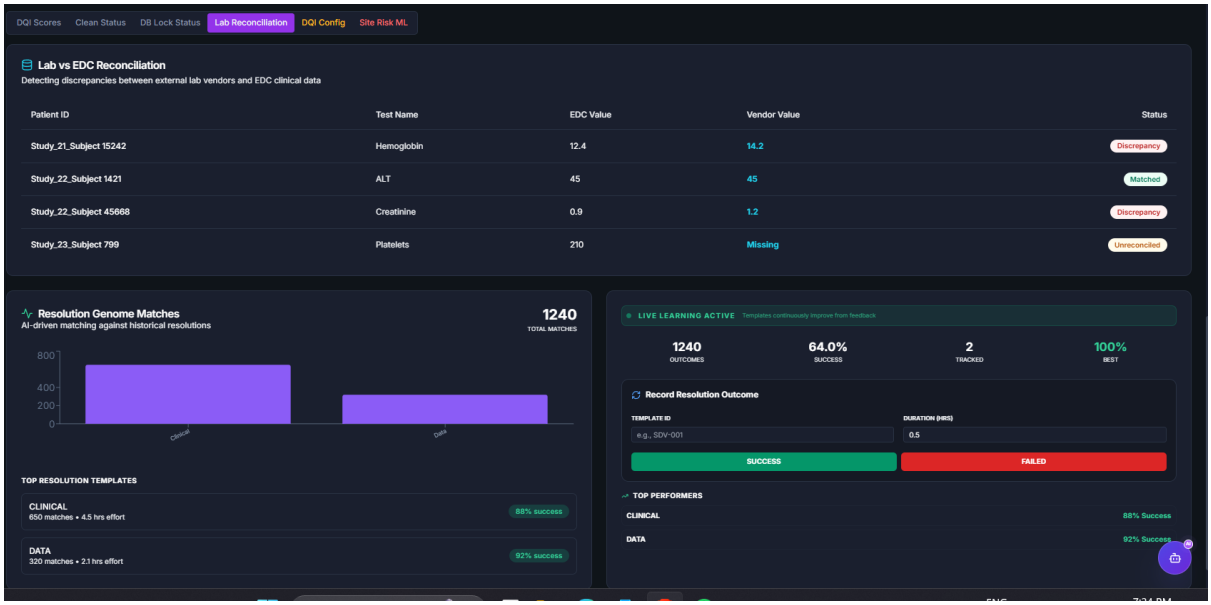

Figure 10: DM Hub View 3: Lab Reconciliation & Resolution Genome

5 Safety View

5.1 Purpose

The Safety View is designed for pharmacovigilance teams to monitor Serious Adverse Events (SAEs), track regulatory timelines, and detect safety signals.

5.2 Key Features

- **SAE Case List**: Tabular view of all SAE cases with status and timeline
- **SLA Countdown**: Visual indicators for regulatory reporting deadlines
- **Safety Signal Detection**: Flagged patterns requiring medical review
- **Narrative Generation**: AI-assisted safety narrative drafting
- **Timeline Visualization**: SAE progression over time
- **Pending Actions**: DM-pending and Safety-pending case tracking

5.3 Regulatory Timelines

The Safety View enforces critical regulatory deadlines:

Table 4: Safety Reporting SLAs

Event Type	Initial Report	Follow-up
Fatal/Life-threatening	7 calendar days	15 calendar days
Other SAEs	15 calendar days	As needed
SUSARs	7-15 days	Per regulation

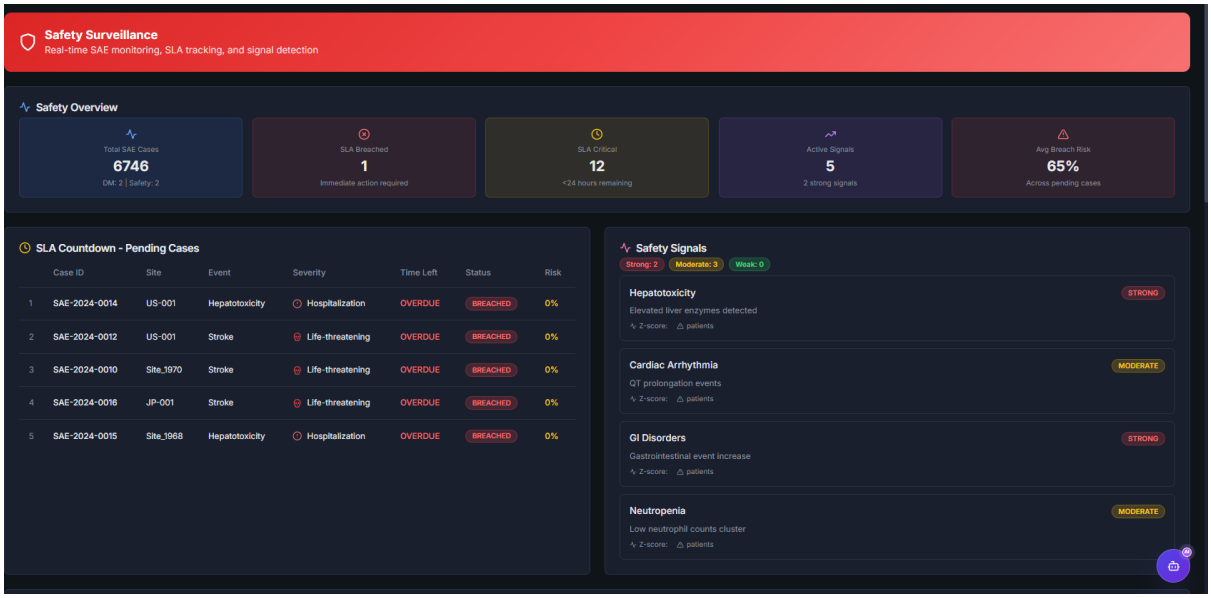

Figure 11: Safety Surveillance View 1: Overview

Figure 12: Safety Surveillance View 2: Timeline

6 Medical Coder View

6.1 Purpose

The Coder View streamlines medical coding workflows for MedDRA (adverse events) and WHO-Drug (medications) dictionary coding.

6.2 Key Features

- **Coding Queue**: Prioritized list of terms awaiting coding

- **Dictionary Search**: Real-time search across MedDRA and WHODrug
- **Auto-Code Suggestions**: AI-suggested codes based on verbatim term
- **Approval Workflow**: One-click code acceptance or escalation
- **Productivity Metrics**: Coding volume and accuracy tracking
- **Uncoded Term Alerts**: Highlighting high-priority pending items

6.3 Coding Workflow

1. Review uncoded terms in priority queue
2. View AI-suggested codes with confidence scores
3. Accept suggestion or search for alternative code
4. Approve coding with electronic signature
5. System records audit trail and updates patient record

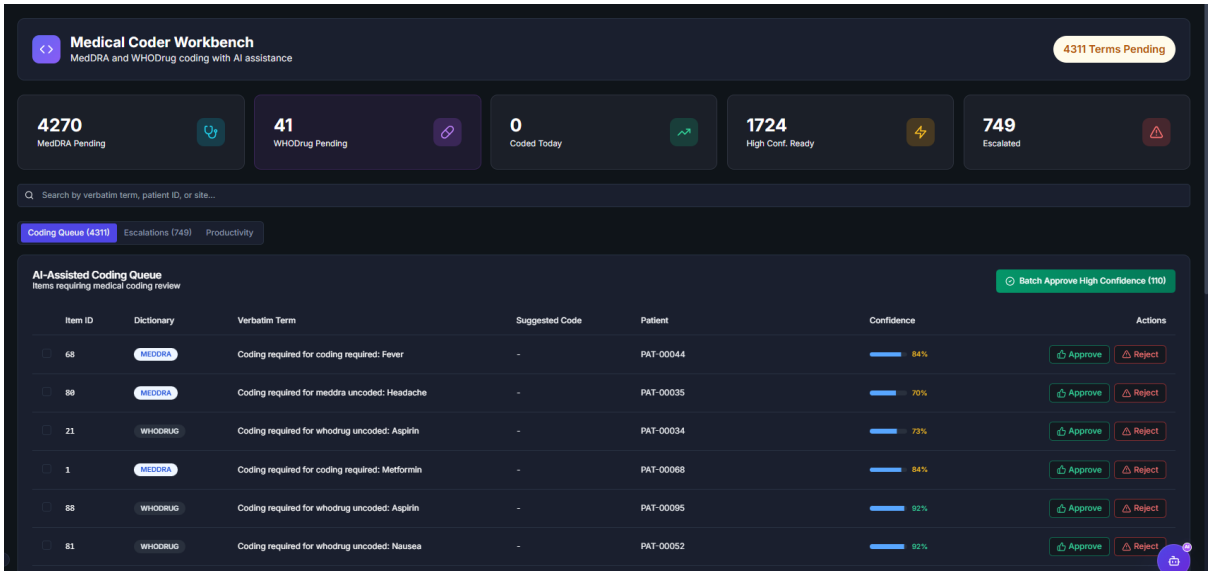

Figure 13: Medical Coder View 1: Overview

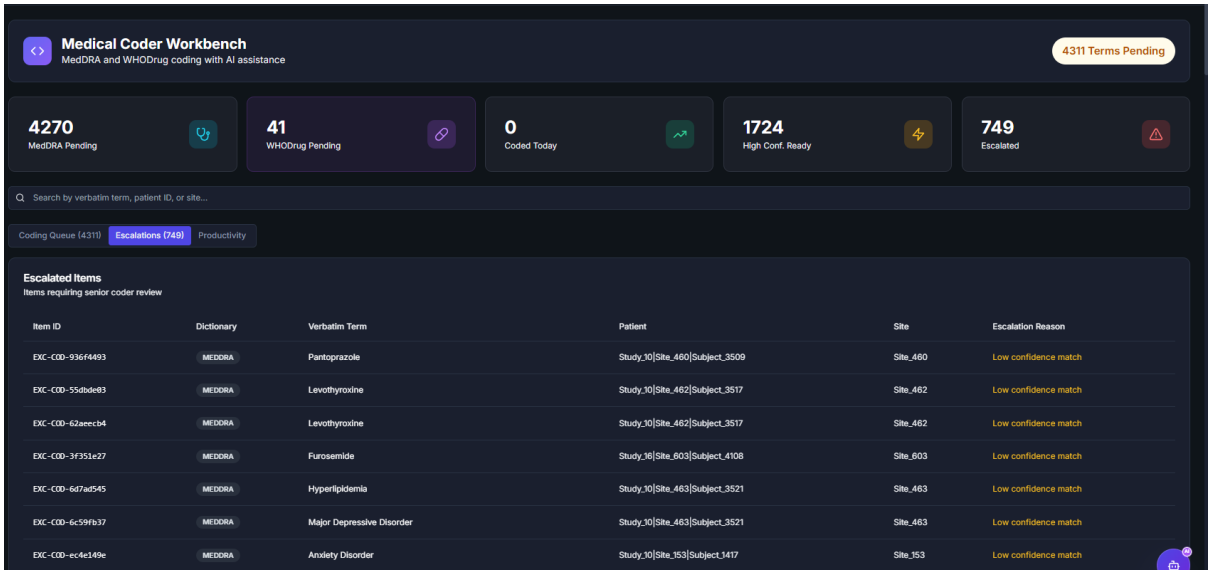

Figure 14: Medical Coder View 2: Escalations

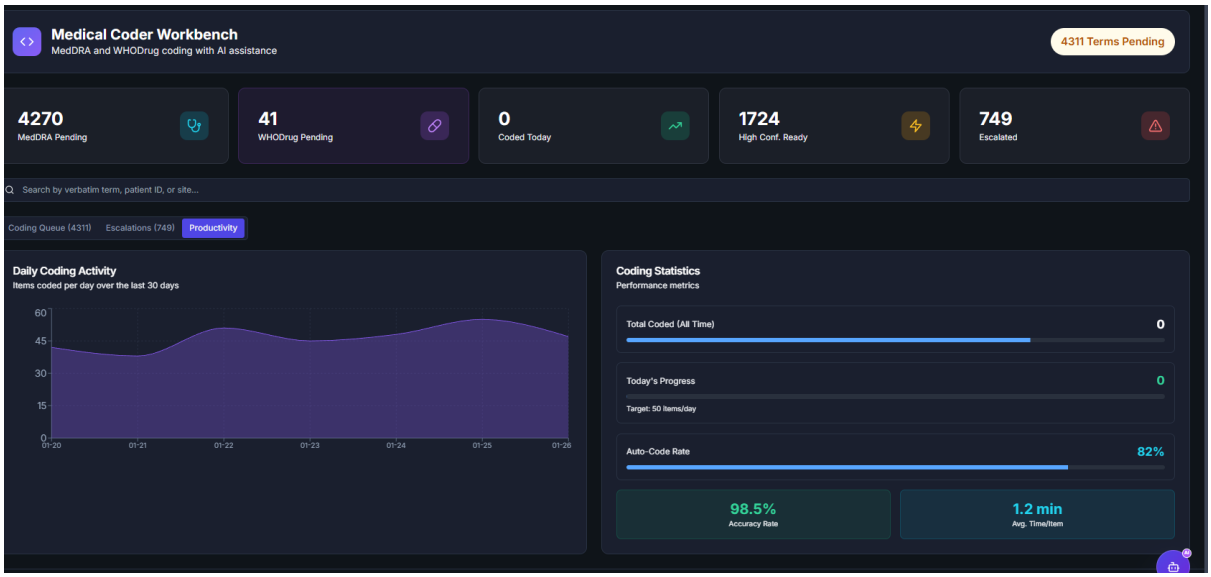

Figure 15: Medical Coder View 3: Statistics

7 Site Portal

7.1 Purpose

The Site Portal provides a site-centric view for site coordinators and local teams, focusing on their specific site's performance and patient population.

7.2 Key Features

- **Site Performance Summary**: DQI score, patient count, issue breakdown
- **Patient Registry**: Complete list of patients enrolled at the site

- **Issue Tracker**: Site-specific issues with resolution tools
- **Benchmark Comparison**: How the site compares to study average
- **Action Plan Generator**: AI-generated improvement recommendations
- **Activity Timeline**: Recent activity and audit history

7.3 Site Performance Indicators

Table 5: Site Portal KPIs

Indicator	Description
Site DQI Score	Aggregate data quality score (0-100)
Clean Patient Rate	Percentage of patients with no open issues
Query Resolution Time	Average days to resolve queries
Protocol Compliance	Adherence to visit windows and procedures

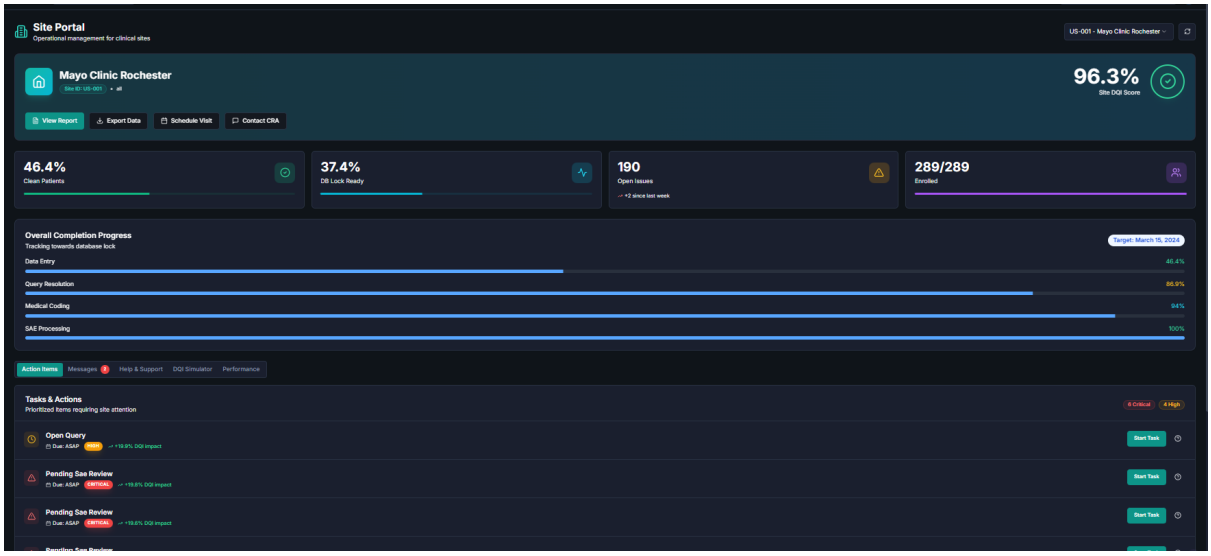

Figure 16: Site Portal View 1: Overview

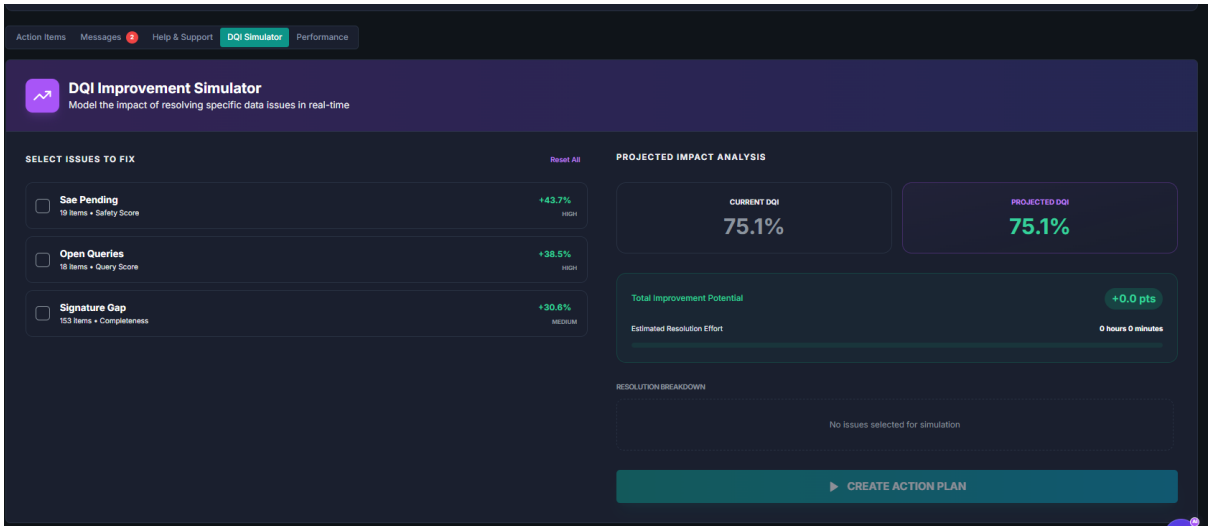

Figure 17: Site Portal View 2: DQI Improvement Simulation

8 Executive Overview

8.1 Purpose

The Executive Overview provides a high-level strategic view for senior leadership, focusing on portfolio health, major risks, and timeline projections.

8.2 Key Features

- **Portfolio Health Score**: Single metric summarizing overall status
- **Study Comparison Matrix**: Side-by-side study performance
- **Risk Heatmap**: Visual representation of risk distribution
- **Milestone Tracker**: Key milestone progress and projections
- **Regional Analysis**: Performance breakdown by geography
- **Trend Analysis**: Historical trends and forward projections

Figure 18: Executive View 1: Overview

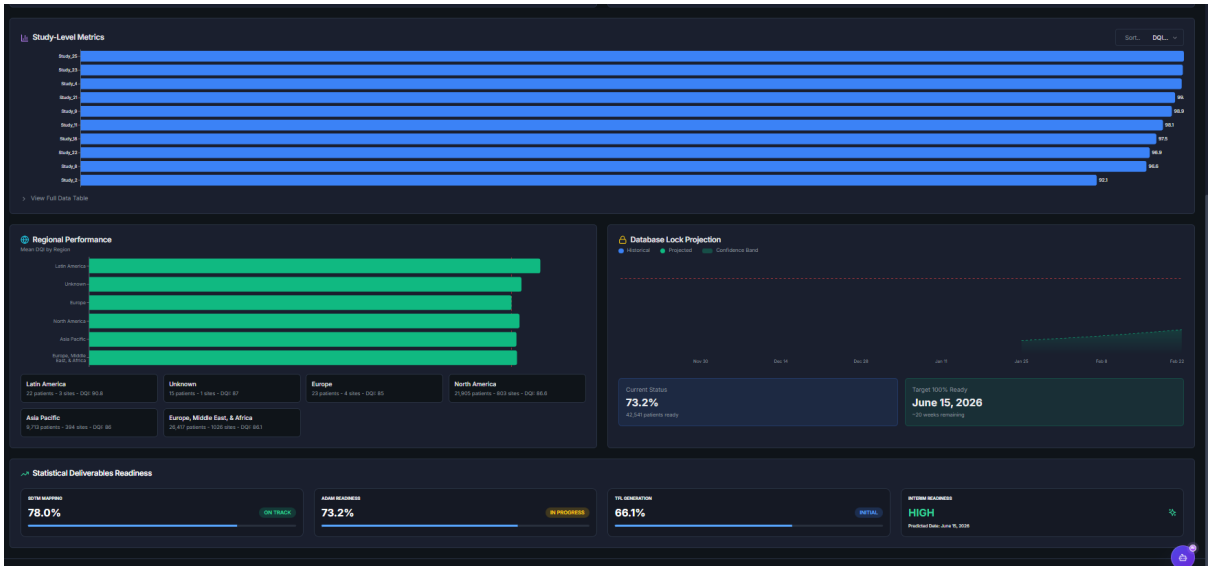

Figure 19: Executive View 2: Regional, Site, DB Metrics

9 Collaboration Hub

9.1 Purpose

The Collaboration Hub enables real-time team communication and coordination across clinical operations roles, integrating messaging with clinical context.

9.2 Key Features

- **Team Chat**: Role-based chat channels for clinical teams
- **Issue Tagging**: Ability to reference specific patients or issues in messages

- **Activity Feed**: Real-time updates on issue resolutions and changes
- **Mention System**: @mention team members for attention
- **Thread Discussions**: Contextual conversations on specific topics
- **Notification Preferences**: Customizable alert settings per user

9.3 Communication Channels

The Collaboration Hub supports structured communication:

Table 6: Collaboration Channels

Channel Type	Purpose
Study Channel	Cross-functional study discussions
Site Channel	Site-specific coordination
Safety Channel	Pharmacovigilance team communication
Data Quality Channel	DM team coordination
General	Cross-study announcements

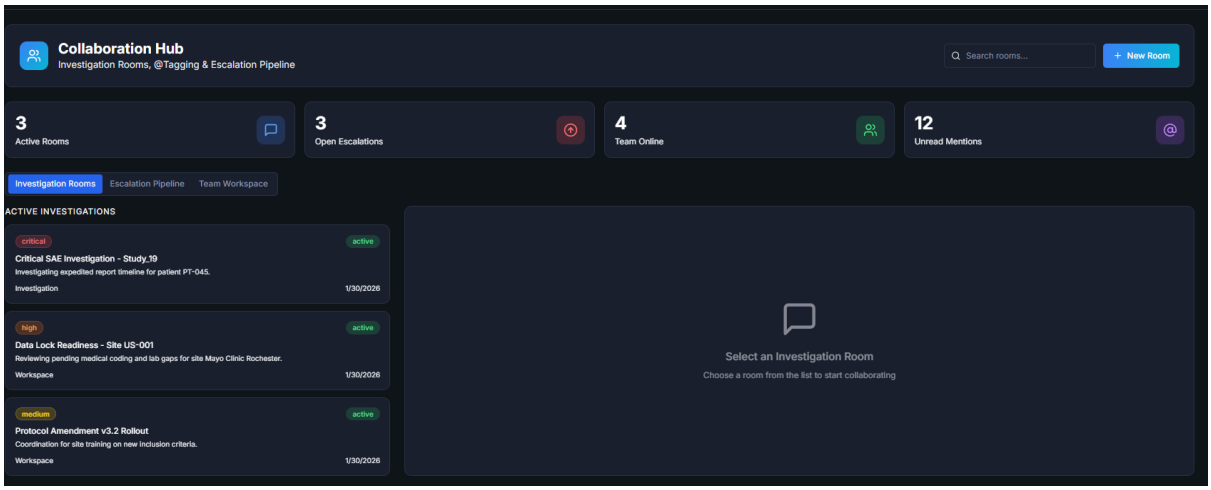

Figure 20: Collaboration Hub View 1: Overview

10 Cascade Explorer

10.1 Purpose

The Cascade Explorer visualizes the dependency graph between patients, issues, and sites, helping users understand the downstream impact of resolving specific issues.

10.2 Key Features

- **Interactive Graph**: Force-directed graph visualization of entities
- **Impact Analysis**: Shows how resolving one issue affects others
- **Filter Controls**: Filter by site, issue type, or patient status

- **Node Details**: Click any node to see detailed information
- **Cascade Path**: Trace blocking relationships between issues
- **Priority Highlighting**: Visual emphasis on high-impact nodes

10.3 Graph Visualization

The visualization uses a force-directed layout where:

- **Nodes** represent patients, issues, or sites
- **Edges** represent blocking relationships
- **Node Size** indicates impact score
- **Node Color** indicates risk level (red=critical, yellow=medium, green=low)

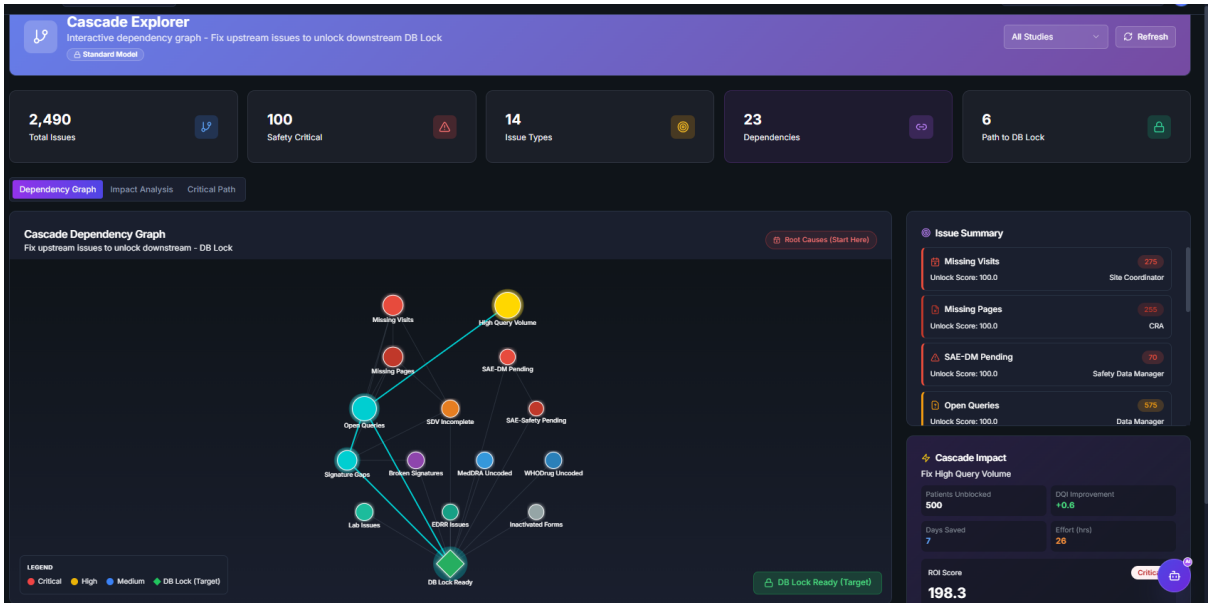

Figure 21: Cascade Explorer View 1: Overview

11 AI Assistant

11.1 Purpose

The AI Assistant provides a natural language interface for querying clinical trial data, available across all role-based views.

11.2 Key Features

- **Natural Language Queries**: Ask questions in plain English
- **Context Awareness**: Understands current study/site selection
- **Data Grounding**: Responses backed by live database queries
- **Agent Trace**: Shows which AI agents processed the query

- **Actionable Insights**: Recommendations with supporting evidence
- **Conversation Memory**: Maintains context within session

11.3 Example Queries

Users can ask questions such as:

- "Why is Site 05 underperforming?"
- "Which patients are blocking DB lock?"
- "Show me critical issues at Site 12"
- "What's the projected DB lock date?"
- "Summarize safety concerns this week"

The AI Assistant routes queries through the 4-agent orchestrator (Supervisor → Diagnostic → Resolver → Communicator) to generate comprehensive responses.

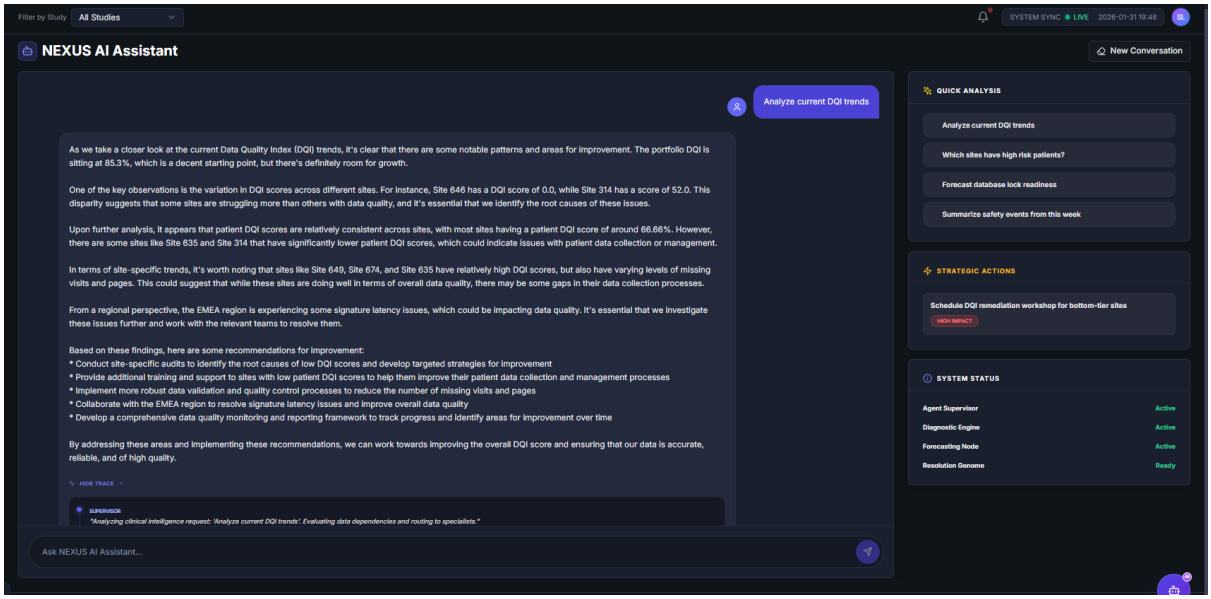

Figure 22: AI Assistant View 1: Overview

12 Reports & Export

12.1 Purpose

The Reports module enables users to generate and export professional reports in multiple formats for stakeholder communication.

12.2 Available Reports

Table 7: Report Types

Report	Description	Formats
Portfolio Summary	Overall portfolio status	PDF, DOCX
Site Performance	Site-level metrics	PDF, DOCX
Patient Status	Individual patient details	PDF
Risk Assessment	Risk tier breakdown	PDF, PPTX
DQI Analysis	Data quality deep-dive	PDF, DOCX
Executive Summary	Leadership briefing	PDF, PPTX

12.3 Report Generation Workflow

1. Select report type from available templates
2. Configure parameters (study, site, date range)
3. Preview report content in-browser
4. Generate final document in desired format
5. Download or share via configured channels

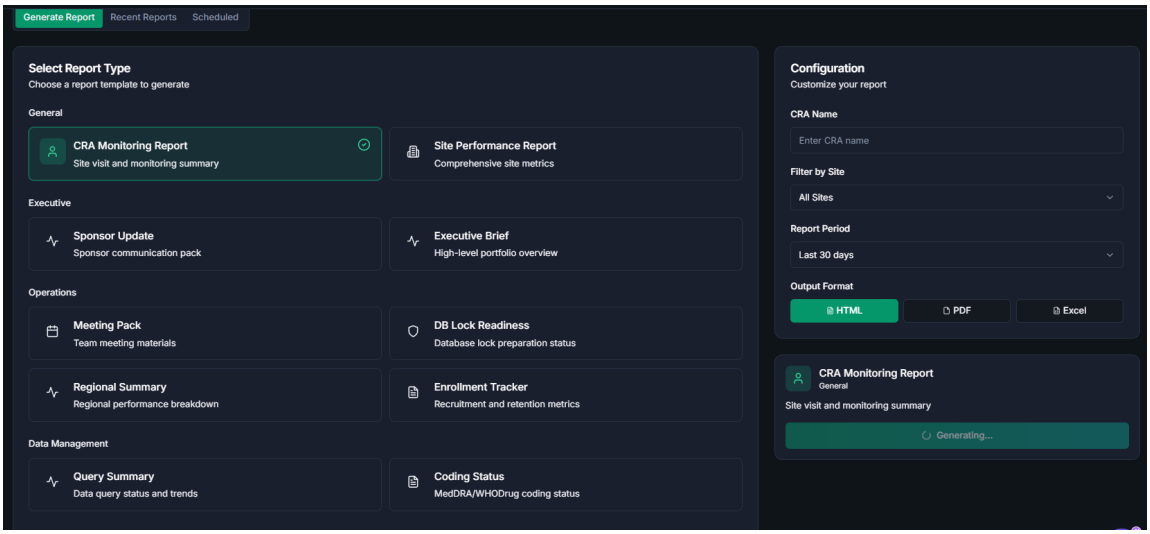

Figure 23: Reports - Generation Interface

13 ML Governance Dashboard

13.1 Purpose

The ML Governance Dashboard provides oversight for all machine learning models deployed in the system, ensuring regulatory compliance and model quality.

13.2 Key Features

- **Model Registry**: Complete list of all ML models with status

- **Approval Workflow**: Review and approve models before deployment
- **Drift Monitoring**: Detect when models need retraining
- **Performance Metrics**: Track model accuracy over time
- **Audit Trail**: Complete history of model changes
- **Deployment Controls**: Stage, deploy, or retire models

13.3 Model Lifecycle States

Table 8: Model Lifecycle

State	Description
Development	Model in training/testing
Pending Approval	Awaiting stakeholder review
Approved	Ready for deployment
Deployed	Active in production
Retired	Removed from service

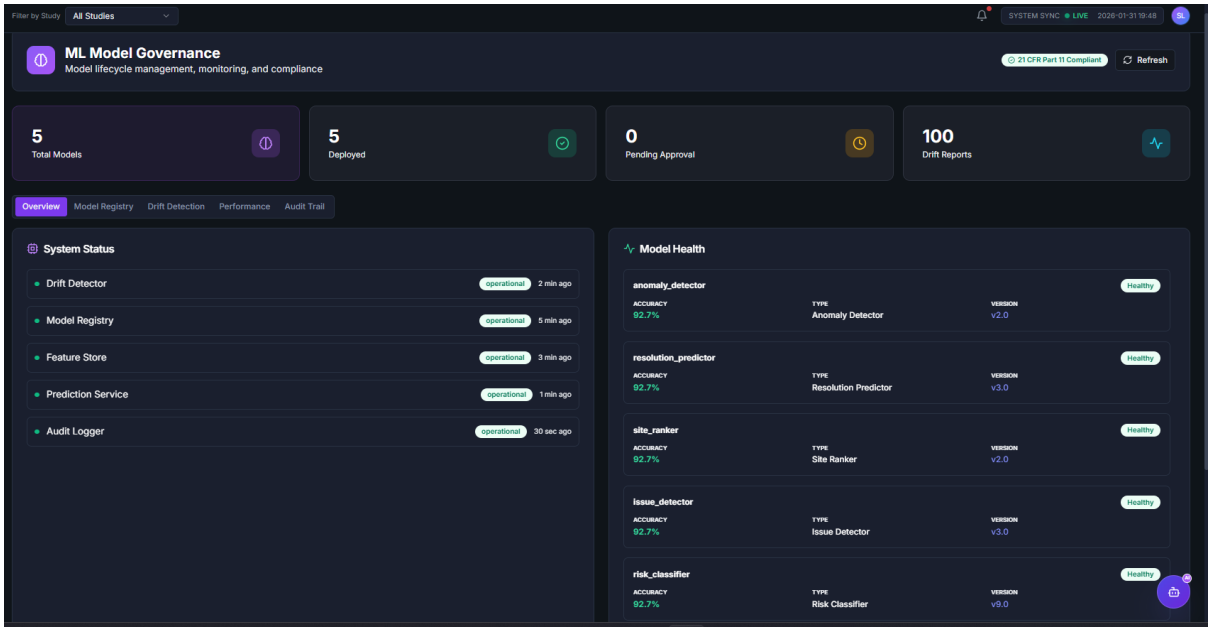

Figure 24: ML Governance View 1: Overview

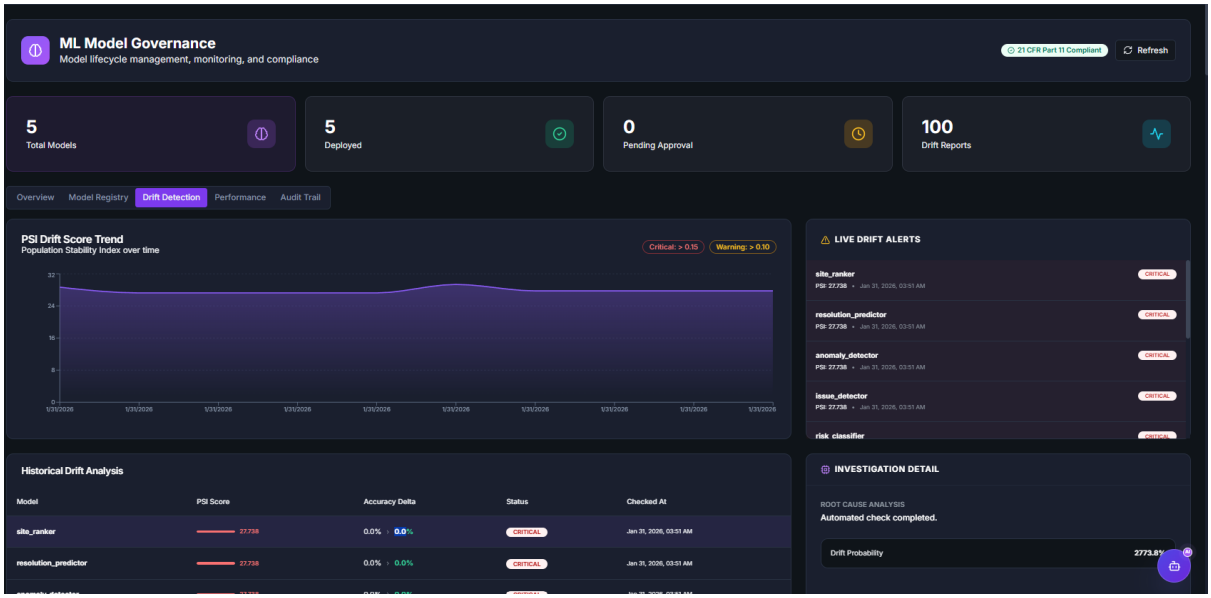

Figure 25: ML Governance View 2: PSI Drift Score & Live Report

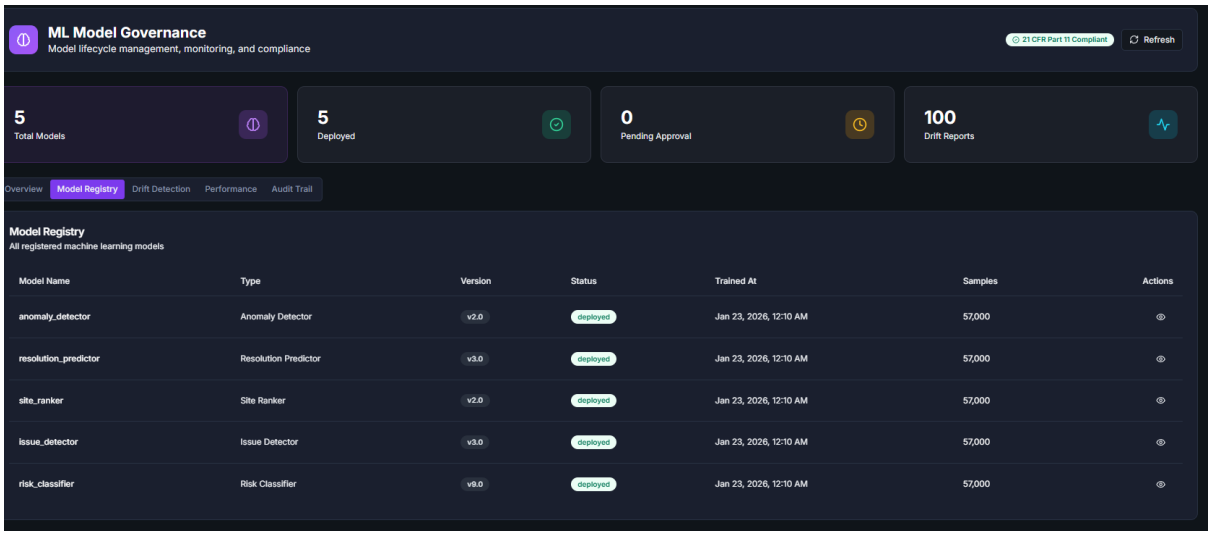

Figure 26: ML Governance View 3: Model Registry

14 Visualization Dashboard

14.1 Purpose

The Visualization Dashboard provides advanced analytics visualizations including charts, graphs, and interactive data exploration tools.

14.2 Visualization Types

- **DQI Distribution**: Histogram of patient DQI scores
- **Site Comparison**: Bar charts comparing site performance

- **Regional Heatmap**: Geographic performance visualization
- **Trend Lines**: Time-series charts for key metrics
- **Issue Breakdown**: Pie/donut charts by issue category
- **Forecast Projections**: DB lock and milestone projections

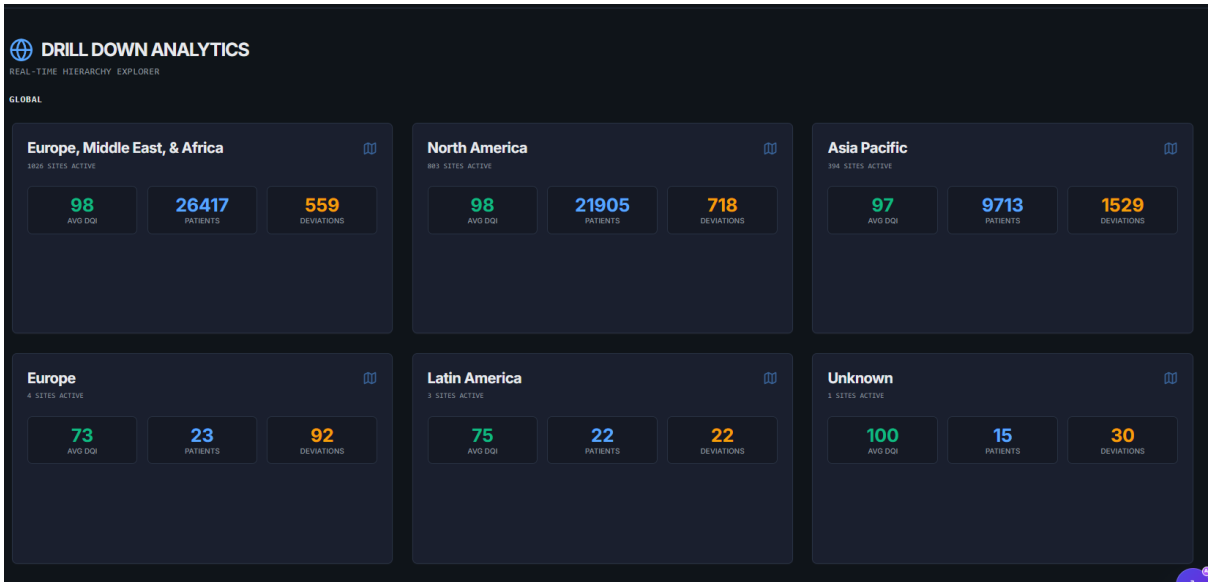

Figure 27: Drill Down Analytics View 1: Overview

Figure 28: Drill Down Analytics View 2: Patients

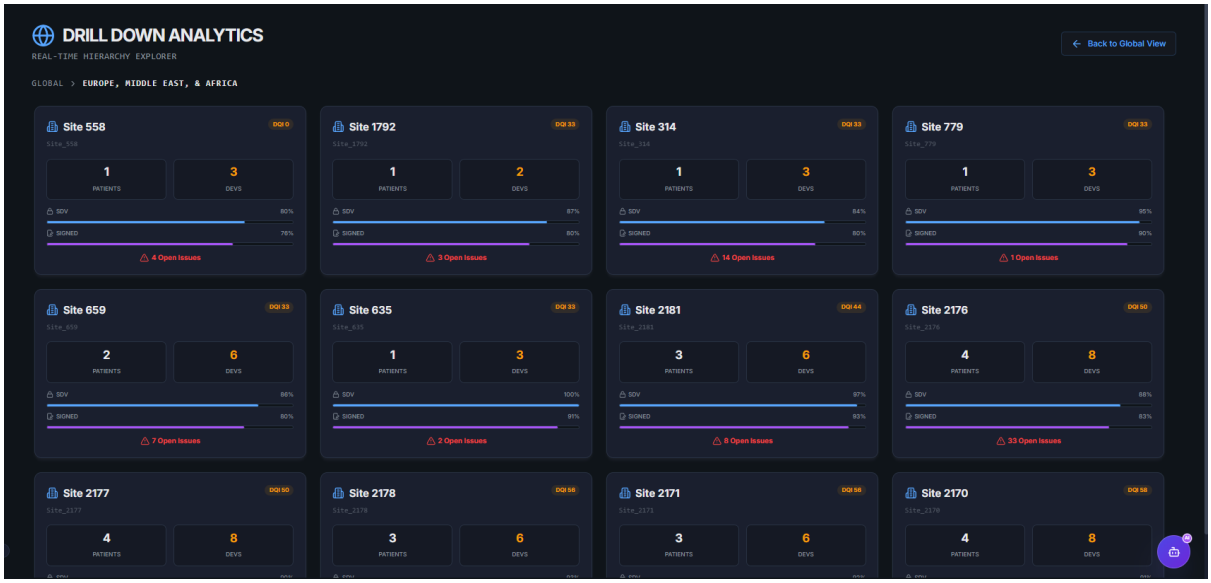

Figure 29: Drill Down Analytics View 3: Sites

TrialPulse Nexus 10X

Technical Documentation

Module 6: Operations & Reference

Scripts, Deployment, and API Reference

Version:	1.0.0
Date:	January 2026
Scripts:	14 Pipeline Scripts
API Routes:	18 Route Modules

Clinical Trial Risk Intelligence Engine

Contents

1 Pipeline Scripts

1.1 Overview

The `src/` directory contains standalone Python scripts that execute specific pipeline stages. These scripts are designed to be run independently or as part of the unified launcher workflow.

1.2 Data Pipeline Scripts

Table 1: Data Pipeline Scripts

Script	Purpose
`run_ingestion.py`	Ingests raw clinical trial data from CSV files and loads into source tables (EDC, IVRS, Labs, Safety, Coding, Query, SDV, Signature)
`run_cleaning.py`	Applies data cleaning transformations including null handling, date parsing, and standardization
`run_upr_builder.py`	Builds the Unified Patient Record by joining all data sources into a single denormalized view per patient

1.2.1 Data Ingestion

The ingestion script reads from `data/raw/` and populates PostgreSQL tables. It handles 8 different clinical data domains and maintains referential integrity across studies and sites.

1.2.2 UPR Builder

The Unified Patient Record builder is the core ETL step that creates a 360-degree view of each patient by:

- Joining EDC data with visit schedules
- Merging safety events and lab results
- Aggregating query and SDV status
- Computing patient-level metrics

1.3 Machine Learning Scripts

Table 2: Machine Learning Training Scripts

Script	Purpose
`run_risk_training_v9_production.py`	Trains the patient risk classification model using XGBoost with 45 engineered features. Outputs to `models/risk_model_v9.pkl`
`run_resolution_time_PRODUCTION_v3.py`	Trains the ResolutionTimeModel that predicts days to resolve issues based on issue type and site characteristics
`run_anomaly_detector_OPTIMIZED_v2.py`	Trains Isolation Forest anomaly detection model to identify unusual patient or site patterns
`run_site_ranker_OPTIMIZED_v2.py`	Trains the SiteRiskRanker model that prioritizes sites needing attention
`run_issue_detector_production.py`	Trains issue classification model to categorize and prioritize data quality issues
`run_comprehensive_risk_training.py`	Full training pipeline that runs all ML models in sequence with cross-validation

1.3.1 Training Workflow

Each training script follows a consistent pattern:

1. Load processed data from `data/processed/` parquet files
2. Engineer features specific to the model's purpose
3. Split data with stratification for class balance
4. Train with hyperparameter tuning where applicable
5. Evaluate on held-out test set
6. Save model artifact with metadata to `models/`
7. Log metrics for drift monitoring

1.4 Knowledge & Analytics Scripts

Table 3: Knowledge Pipeline Scripts

Script	Purpose
`run_embedding_pipeline.py`	Generates vector embeddings for all text content using sentence-transformers. Populates ChromaDB collections
`run_rag_knowledge_base.py`	Indexes ICH-GCP guidelines, protocol documents, and SOPs for RAG retrieval
`run_causal_hypothesis.py`	Runs the Causal Hypothesis Engine to generate data-driven insights and patterns
`run_resolution_genome.py`	Analyzes historical resolution patterns to build the Resolution Genome knowledge base
`run_site_risk_ranker.py`	Generates site risk rankings for prioritized monitoring

1.4.1 Embedding Pipeline

The embedding pipeline processes 7 content categories:

- Patient issue descriptions
- Query text and resolutions
- Safety narratives
- Protocol sections
- ICH-GCP guidelines
- SOP procedures
- Historical resolution notes

Each category is stored in a separate ChromaDB collection with metadata for filtered retrieval.

1.5 Utility Script

Table 4: Utility Script

Script	Purpose
`rebuild_upr_perfect.py`	Complete rebuild utility that regenerates the entire Unified Patient Record from scratch, useful after schema changes

2 Deployment Guide

2.1 System Requirements

Table 5: Minimum System Requirements

Component	Requirement
Operating System	Windows 10/11, macOS, or Linux
Python	3.10+
Node.js	18+
PostgreSQL	14+
Neo4j	5.x (optional, for graph features)
RAM	8GB minimum, 16GB recommended
Disk Space	10GB for application + data

2.2 Environment Setup

2.2.1 Step 1: Clone Repository

Clone the TrialPulse Nexus repository from the version control system to your local machine.

2.2.2 Step 2: Python Environment

Create a virtual environment and install dependencies:

- Create virtual environment in project root
- Activate the environment
- Install requirements from `requirements.txt`

2.2.3 Step 3: Environment Variables

Create a `.env` file in the project root with the following configuration:

Table 6: Required Environment Variables

Variable	Description
`DATABASE_URL`	PostgreSQL connection string
`GOOGLE_API_KEY`	Google Gemini API key for LLM
`NEO4J_URI`	Neo4j connection URI (optional)
`NEO4J_USER`	Neo4j username (optional)
`NEO4J_PASSWORD`	Neo4j password (optional)
`JWT_SECRET_KEY`	Secret for JWT token signing

2.2.4 Step 4: Database Setup

Initialize the PostgreSQL database:

1. Create a new PostgreSQL database
2. Run the setup script which creates all required tables
3. The script automatically creates 25+ tables with proper indexes

2.2.5 Step 5: Frontend Dependencies

Navigate to the frontend directory and install Node.js dependencies using npm.

2.3 Running the Application

2.3.1 Unified Launcher (Recommended)

The project includes a unified launcher script that starts all components:

Table 7: Unified Launcher Commands

Command	Action
Run launcher	Starts backend (port 8000) and frontend (port 5173)
Backend only	Starts only the FastAPI backend
Frontend only	Starts only the React frontend

2.3.2 Manual Startup

To start components manually:

1. **Backend**: Navigate to project root and run uvicorn with the FastAPI app
2. **Frontend**: Navigate to frontend directory and run npm dev server

2.4 Data Initialization

After starting the application for the first time:

1. **Load Raw Data**: Place CSV files in `data/raw/` following the expected schema
2. **Run Ingestion**: Execute the ingestion script to populate source tables
3. **Build UPR**: Run the UPR builder to create unified patient records
4. **Train Models**: Run ML training scripts or use pre-trained models from `models/`
5. **Initialize Knowledge**: Run embedding and RAG scripts for AI features

2.5 Accessing the Application

Table 8: Application URLs

Component	URL	Purpose
Frontend	`http://localhost:5173`	React application
Backend API	`http://localhost:8000`	FastAPI REST API
API Docs	`http://localhost:8000/docs`	Swagger UI documentation
ReDoc	`http://localhost:8000/redoc`	Alternative API docs

2.6 Default Credentials

For demo purposes, the following test users are available:

Table 9: Demo User Accounts

Username	Password	Role	Access
admin	admin123	Administrator	Full access + admin
lead	lead123	Study Lead	Portfolio view, approvals
dm	dm123	Data Manager	Query management, data editing
cra	cra123	CRA	Site monitoring, reports
coder	coder123	Medical Coder	MedDRA/WHODrug coding
safety	safety123	Safety Officer	SAE tracking, narratives
exec	exec123	Executive	Dashboard views, reports
testuser	testpassword	Test User	General testing

Note: In TEST_MODE, the system accepts any username/password combination and creates users on-the-fly. For production, use the exact credentials listed above.

2.7 Production Considerations

For production deployment:

- **Security**: Change all default passwords and use strong JWT secrets
- **HTTPS**: Configure SSL/TLS certificates for encrypted connections
- **Database**: Use connection pooling and configure proper backups
- **Monitoring**: Set up application monitoring and log aggregation
- **Scaling**: Consider horizontal scaling with load balancer for high traffic

3 API Reference

3.1 Overview

The TrialPulse Nexus API is built with FastAPI and provides a comprehensive REST interface for all system functionality. The API is organized into 18 route modules covering different functional domains.

3.2 Authentication

All API endpoints (except login) require JWT Bearer token authentication. Tokens are obtained via the login endpoint and must be included in the Authorization header.

Table 10: Authentication Endpoints

Method	Endpoint	Description
POST	`/api/v1/auth/login`	Authenticate and receive JWT tokens
POST	`/api/v1/auth/refresh`	Refresh expired access token
POST	`/api/v1/auth/logout`	Invalidate current session
GET	`/api/v1/auth/me`	Get current user profile

3.3 Patient Endpoints

The patients API provides access to patient data, issues, and risk information.

Table 11: Patient API Endpoints

Method	Endpoint	Description
GET	`/patients`	List patients with pagination
GET	`/patients/{key}`	Get single patient details
GET	`/patients/search`	Search patients by query
GET	`/patients/dqi`	Get DQI distribution
GET	`/patients/clean-status`	Get clean patient counts
GET	`/patients/dblock-status`	Get DB lock readiness
GET	`/patients/issues`	Get patient issue summary
GET	`/patients/{key}/risk-explanation`	Get ML risk explanation

3.4 Site Endpoints

Site management and monitoring endpoints.

Table 12: Site API Endpoints

Method	Endpoint	Description
GET	`/sites`	List all sites
GET	`/sites/{id}`	Get site details
GET	`/sites/benchmarks`	Get site benchmarks
GET	`/sites/smart-queue`	AI-prioritized work queue
GET	`/sites/{id}/patients`	Get patients at site
GET	`/sites/{id}/portal`	Get site portal data
GET	`/sites/{id}/dqi-issues`	Get site DQI issues
POST	`/sites/{id}/action-plan`	Generate action plan

3.5 Analytics Endpoints

Data analytics and visualization endpoints.

Table 13: Analytics API Endpoints

Method	Endpoint	Description
GET	`/analytics/portfolio`	Portfolio summary metrics
GET	`/analytics/dqi-distribution`	DQI score distribution
GET	`/analytics/cascade`	Cascade analysis data
GET	`/analytics/patterns`	Detected patterns
GET	`/analytics/bottlenecks`	Process bottlenecks
GET	`/analytics/quality-matrix`	Quality matrix data
GET	`/analytics/recommendations`	AI recommendations
POST	`/analytics/refresh`	Refresh analytics cache

3.6 Intelligence Endpoints

AI and agentic intelligence endpoints.

Table 14: Intelligence API Endpoints

Method	Endpoint	Description
GET	`/intelligence/hypotheses`	Get causal hypotheses
POST	`/intelligence/swarm/run`	Run agent swarm investigation
GET	`/intelligence/anomalies`	Get detected anomalies
POST	`/intelligence/auto-fix`	Auto-resolve an issue
POST	`/intelligence/assistant/query`	Natural language query
POST	`/intelligence/assistant/reset`	Reset AI session

3.7 Issues Endpoints

Issue tracking and resolution endpoints.

Table 15: Issues API Endpoints

Method	Endpoint	Description
GET	`/issues`	List issues with filters
GET	`/issues/summary`	Issue summary counts
POST	`/issues`	Create new issue
PUT	`/issues/{id}`	Update issue
POST	`/issues/{id}/resolve`	Resolve issue
POST	`/issues/{id}/escalate`	Escalate issue

3.8 Safety Endpoints

Pharmacovigilance and SAE management endpoints.

Table 16: Safety API Endpoints

Method	Endpoint	Description
GET	`/safety/overview`	Safety dashboard data
GET	`/safety/sae-cases`	List SAE cases
GET	`/safety/sla-status`	Regulatory SLA status
GET	`/safety/signals`	Safety signal detection
GET	`/safety/timeline`	SAE timeline data
GET	`/safety/narratives/{id}`	Get SAE narrative
POST	`/safety/sae/{id}/update-status`	Update SAE status

3.9 Coding Endpoints

Medical coding (MedDRA/WHODrug) endpoints.

Table 17: Coding API Endpoints

Method	Endpoint	Description
GET	`/coding/queue`	Get coding queue
GET	`/coding/meddra/pending`	MedDRA pending items
GET	`/coding/whodrug/pending`	WHODrug pending items
GET	`/coding/stats`	Coding statistics
POST	`/coding/approve/{id}`	Approve coding
GET	`/coding/search/{dict}`	Search dictionary

3.10 Reports Endpoints

Report generation endpoints.

Table 18: Reports API Endpoints

Method	Endpoint	Description
GET	`/reports/types`	List available report types
POST	`/reports/generate`	Generate a report
GET	`/reports/preview/{type}`	Preview report content

3.11 Simulation Endpoints

Digital twin and what-if simulation endpoints.

Table 19: Simulation API Endpoints

Method	Endpoint	Description
GET	`/simulation/scenarios`	List scenario types
POST	`/simulation/run`	Run simulation
POST	`/simulation/compare`	Compare scenarios
GET	`/simulation/current-state`	Get current state
GET	`/simulation/projections`	Get projections
GET	`/simulation/db-lock-projection`	DB lock timeline
POST	`/simulation/what-if`	Run what-if analysis

3.12 Graph Endpoints

Neo4j knowledge graph and cascade endpoints.

Table 20: Graph API Endpoints

Method	Endpoint	Description
GET	`/graph/nodes`	Get graph nodes
GET	`/graph/edges`	Get graph edges
GET	`/graph/cascade-path/{id}`	Get cascade path
GET	`/graph/cascade-analysis`	Full cascade analysis
GET	`/graph/visualization-data`	Data for graph viz

3.13 Dashboard Endpoints

Role-based dashboard data endpoints.

Table 21: Dashboard API Endpoints

Method	Endpoint	Description
GET	`/dashboards/summary`	General summary
GET	`/dashboards/main`	Main dashboard
GET	`/dashboards/cra`	CRA dashboard data
GET	`/dashboards/data_manager`	DM dashboard data
GET	`/dashboards/safety`	Safety dashboard data
GET	`/dashboards/study_lead`	Study Lead data
GET	`/dashboards/site`	Site dashboard data
GET	`/dashboards/coder`	Coder dashboard data

3.14 ML Endpoints

Machine learning model management endpoints.

Table 22: ML API Endpoints

Method	Endpoint	Description
GET	`/ml/models`	List ML models
GET	`/ml/summary`	ML summary stats
POST	`/ml/models/{id}/approve`	Approve model
POST	`/ml/models/{id}/deploy`	Deploy model
GET	`/ml/drift-reports`	Get drift reports
GET	`/ml/audit-log`	Model audit log

3.15 Agent Endpoints

Direct agent invocation endpoints.

Table 23: Agents API Endpoints

Method	Endpoint	Description
GET	`/agents`	List all agents
GET	`/agents/supervisor/status`	Supervisor status
POST	`/agents/supervisor/act`	Invoke supervisor
POST	`/agents/diagnostic/act`	Invoke diagnostic
POST	`/agents/forecaster/act`	Invoke forecaster
POST	`/agents/resolver/act`	Invoke resolver
POST	`/agents/executor/act`	Invoke executor
POST	`/agents/communicator/act`	Invoke communicator

3.16 Response Formats

All API responses follow a consistent JSON structure:

Table 24: Standard Response Fields

Field	Type	Description
`success`	boolean	Whether request succeeded
`data`	object/array	Response payload
`message`	string	Human-readable message
`error`	string	Error description if failed
`timestamp`	string	ISO 8601 timestamp

3.17 Error Codes

Table 25: HTTP Status Codes

Code	Meaning
200	Success
201	Created successfully
400	Bad request (invalid parameters)
401	Unauthorized (missing/invalid token)
403	Forbidden (insufficient permissions)
404	Resource not found
422	Validation error
500	Internal server error

3.18 Interactive Documentation

FastAPI automatically generates interactive API documentation accessible at:

- **Swagger UI**: `http://localhost:8000/docs` – Interactive API explorer with try-it-out functionality
- **ReDoc**: `http://localhost:8000/redoc` – Clean, readable API reference

4 Future Features & Roadmap

4.1 Overview

The following modules have been developed and exist in the codebase but are not yet integrated into the running demo. These represent infrastructure investments for future capabilities and production readiness.

Table 26: Future Feature Modules

Module	Directory	Lines of Code	Status
Kafka Streaming	`src/streaming/`	900+	Ready for integration
Notification Service	`src/notifications/`	660+	Ready for integration
Electronic Signatures	`src/compliance/`	480+	Ready for integration
LLM Fine-tuning	`src/finetuning/`	400+	Development

4.2 Kafka Streaming Service

4.2.1 Purpose

The streaming module provides real-time event streaming infrastructure using Apache Kafka, enabling live updates to dashboards and triggering automated workflows when data changes occur.

4.2.2 Work Completed

The streaming service includes the following components:

- **Kafka Producer**: Publishes events to Kafka topics with delivery guarantees. Supports batch sending and compression for high throughput.
- **Kafka Consumer**: Subscribes to topics and processes incoming events with at-least-once semantics. Supports consumer groups for parallel processing.
- **Event Processor**: Routes incoming events to appropriate handlers based on event type. Implements dead letter queue for failed messages.
- **UPR Updater**: Listens for data change events and incrementally updates the Unified Patient Record without full rebuild.
- **Stream Configuration**: Centralized configuration for topics, partitions, and retention policies.

4.2.3 Supported Event Types

Table 27: Streaming Event Types

Event Type	Description
Patient Update	Patient record field changes
Issue Detected	New issue identified by ML models
Issue Resolved	Issue marked as resolved
DQI Change	Patient DQI score recalculated
Site Update	Site-level metric changes
Drift Detected	ML model drift alert

4.2.4 Integration Path

To enable streaming in production:

1. Deploy Apache Kafka cluster (or use managed service like Confluent Cloud)
2. Configure Kafka broker addresses in environment variables
3. Enable streaming module in backend configuration
4. Update frontend to use WebSocket connections for real-time updates

4.3 Notification Service

4.3.1 Purpose

The notification service provides a multi-channel communication system for alerting users about important events, including in-app notifications, email, SMS, and integration with collaboration platforms like Slack and Microsoft Teams.

4.3.2 Work Completed

The notification module includes:

- **Notification Queue**: Priority-based queue with PostgreSQL persistence. Supports urgent, high, normal, and low priority levels.
- **Multi-Channel Delivery**: Abstracted delivery interface supporting in-app, email, SMS, push notifications, Slack, and Teams.
- **Notification Types**: Predefined types including alerts, mentions, escalations, tasks, issues, safety events, system messages, digests, and reports.
- **User Preferences**: Per-user configuration for preferred channels, quiet hours, batch preferences, and rate limiting.
- **Delivery Tracking**: Full lifecycle tracking from pending to delivered to read with timestamps.
- **Batch Digests**: Aggregation of multiple notifications into daily or weekly digest emails.

4.3.3 Notification Lifecycle

Notifications progress through the following states:

Table 28: Notification Status Lifecycle

Status	Description
Pending	Created, not yet processed
Queued	Added to delivery queue
Sent	Dispatched to channel
Delivered	Confirmed delivery
Read	User acknowledged
Archived	Moved to archive
Failed	Delivery failed

4.3.4 Integration Path

To enable notifications in production:

1. Configure SMTP server for email delivery
2. Set up SMS gateway (Twilio, AWS SNS)
3. Create Slack/Teams apps and configure webhooks
4. Enable notification service in backend
5. Add notification UI components to frontend

4.4 Electronic Signature Service

4.4.1 Purpose

The electronic signature module implements 21 CFR Part 11 compliant digital signatures for regulatory submissions. This is essential for GxP-regulated clinical trial environments where electronic records require legally binding signatures.

4.4.2 Regulatory Compliance

The service implements the following 21 CFR Part 11 requirements:

Table 29: 21 CFR Part 11 Compliance Features

Requirement	Implementation
User Identity Verification	Password authentication plus multi-factor authentication (MFA) via TOTP, SMS, or hardware token
Signature Meaning	Predefined meanings: Authored, Reviewed, Approved, Verified, Witnessed, Acknowledged, Confirmed
Trusted Timestamp	Server-side timestamps with time zone and synchronization
Immutable Hash	SHA-256 HMAC signature hash with chained integrity
Audit Trail	Complete record of all signature events with tamper detection

4.4.3 Work Completed

The electronic signature service includes:

- **Signature Creation**: Full workflow for creating legally binding e-signatures with MFA verification
- **Hash Chain Integrity**: Each signature references the previous signature hash, creating an immutable chain that detects tampering
- **Verification Service**: Validates signature authenticity by recomputing hash and checking chain integrity
- **Audit Trail**: Complete history of signature events with checksums for tamper detection
- **Multi-Record Support**: Sign any record type (issues, reports, approvals) with consistent interface

4.4.4 Signature Meanings

Per 21 CFR Part 11, each signature includes a legally binding meaning:

Table 30: Standard Signature Meanings

Meaning	Legal Statement
Authored	"I authored this record"
Reviewed	"I reviewed this record"
Approved	"I approve this record"
Verified	"I verified the accuracy"
Witnessed	"I witnessed this action"

4.4.5 Integration Path

To enable electronic signatures:

1. Implement MFA integration (Google Authenticator, Authy, or hardware tokens)
2. Create signature capture UI with meaning selection
3. Add signature verification endpoints to API
4. Integrate signature requirements into approval workflows
5. Configure audit trail export for regulatory inspections

4.5 LLM Fine-tuning Pipeline

4.5.1 Purpose

The fine-tuning module provides infrastructure for training custom language models on clinical trial data, enabling more accurate and domain-specific AI responses.

4.5.2 Work Completed

The fine-tuning pipeline includes:

- **Training Data Generator**: Automatically generates training examples from the PostgreSQL database covering DQI analysis, issue resolution, site performance, and general queries
- **Data Preparation**: Converts clinical data into instruction-following format compatible with modern LLM fine-tuning (Alpaca and chat formats)
- **Configuration Management**: Centralized configuration for model parameters, training hyperparameters, and system prompts
- **Jupyter Notebook**: Interactive notebook for Google Colab-based training using parameter-efficient fine-tuning (LoRA/QLoRA)
- **Training Scripts**: Command-line scripts for running fine-tuning jobs on local or cloud infrastructure

4.5.3 Training Data Categories

The generator creates examples across five categories:

Table 31: Fine-tuning Data Categories

Category	Description
DQI Analysis	Explaining patient data quality scores and contributing factors
Issue Resolution	Recommending resolution steps for different issue types
Site Performance	Analyzing site metrics and providing improvement suggestions
Query Response	Answering natural language questions about clinical data
General Analysis	Broad analytical reasoning about trial performance

4.5.4 Integration Path

To use fine-tuned models:

1. Generate training data using the data preparation script
2. Upload training data to Google Colab or training infrastructure
3. Run fine-tuning notebook with appropriate base model (Gemma, Llama, etc.)
4. Export fine-tuned model weights
5. Update LLM wrapper to use fine-tuned model for inference
6. Compare performance against base model on held-out test set

4.6 Roadmap Summary

Table 32: Feature Integration Priority

Feature	Priority	Effort	Value
Notification Service	High	Medium	User engagement, timely alerts
Electronic Signatures	High	Medium	Regulatory compliance
Kafka Streaming	Medium	High	Real-time updates
LLM Fine-tuning	Low	High	Improved AI accuracy