IINTS-AF SDK Technical Reference Manual

Research Use Only — Not for Clinical Care

Generated: 2026-03-01 12:59 UTC

IINTS-AF SDK Technical Reference Manual

Version 0.1.19 | Python SDK

PRE-CLINICAL USE ONLY - NOT FOR PATIENT CARE

This SDK is intended for research, simulation, and algorithm validation.
It has NOT received FDA clearance or CE marking for clinical use.

How to Use This Manual (Read First)

This manual is long. Use this map to find what you need fast:

- **Full SDK overview:** `docs/COMPREHENSIVE_GUIDE.md`
- **CLI reference:** `docs/TECHNICAL_README.md`
- **Research track (predictor training):** `research/README.md`
- **Notebooks:** `examples/notebooks/README.md`

Recommended notebook order (best for new users):

- `00_Quickstart.ipynb` — run a full simulation
- `01_Presets_and_Scenarios.ipynb` — scenarios + presets
- `02_Safety_and_Supervisor.ipynb` — safety checks
- `03_Audit_Trail_and_Report.ipynb` — audit trail + PDF
- `04_Baseline_and_Metrics.ipynb` — metrics & baselines
- `05_Devices_and_HumanInLoop.ipynb` — pumps/sensors + manual override
- `06_Optional_Torch_LSTM.ipynb` — predictor training
- `07_Ablation_Supervisor.ipynb` — safety ablation
- `08_Data_Registry_and_Import.ipynb` — data import + registry

Quick task routing:

- **Run a simulation fast:** Section 2.2
- **Customize safety limits:** Section 4.2
- **Generate audit report:** Section 5.4
- **Train an AI predictor:** Section 8.2 + `research/README.md`

Table of Contents

1. Executive Summary
   - Overview and key capabilities
   - Intended use and audiences
   - Safety-first philosophy

2. Getting Started (Step-by-Step)

2.1 Installation Guide
2.2 Your First Simulation (60 seconds)

2.3 Understanding the Output Files
2.4 Creating Custom Algorithms
2.5 Adding Stress Tests
2.6 Benchmarking Against Baselines
2.7 Using Preset Scenarios
2.8 Importing Real CGM Data
2.9 Generating Custom Reports
2.10 Next Steps

3. Architecture Overview

3.1 System Components
3.2 Data Flow Diagram
3.3 Safety Layer Integration

4. Safety Architecture (Critical)

4.1 Design Philosophy
4.2 SafetyConfig Configuration
4.3 9 Safety Checks Explained
4.4 Safety Levels and Interventions
4.5 Input Validation Rules
4.6 Simulation Termination Conditions

5. Tutorials and Cookbook

5.1 24-Hour Simulation Walkthrough
5.2 Building an ML-Hybrid Algorithm
5.3 Batch Experiment Execution
5.4 Audit Trail Analysis
5.5 Custom Safety Thresholds
5.6 Pump Emulator Benchmarking
5.7 Live Streaming Simulation
5.8 Reproducible Runs for Publications

6. API Reference

6.1 Core Classes and Interfaces
6.2 Algorithm Development Guide
6.3 Simulator Configuration
6.4 Patient Profile Customization
6.5 Device Models (Sensor/Pump)
6.6 High-Level API Functions

7. Practical Examples

7.1 Complete Algorithm Example
7.2 CLI vs Python API Comparison
7.3 Stress Testing Patterns
7.4 Human-in-the-Loop Integration
7.5 Data Import Workflows

8. Advanced Topics

8.1 Commercial Pump Emulators
8.2 Dataset Registry Usage
8.3 Reproducibility Techniques
8.4 Performance Profiling
8.5 Custom Metrics Calculation

9. Troubleshooting

9.1 Common Installation Issues
9.2 Simulation Problems
9.3 Algorithm Development Tips
9.4 Data Import Solutions
9.5 Performance Optimization

10. Quick Reference

10.1 Essential CLI Commands
10.2 Python Code Snippets
10.3 Safety Thresholds Cheatsheet
10.4 Clinical Metrics Targets

11. Glossary of Terms

1. Executive Summary

The IINTS-AF SDK (Intelligent Insulin Titration System for Artificial Pancreas) is a **safety-first simulation and validation platform** for insulin dosing algorithms targeting closed-loop insulin delivery research.

Key Capabilities

[OK] **Plug-and-play algorithm architecture** - Implement one method, get full simulation
[OK] **9-layer Independent Safety Supervisor** - Deterministic override guarantees
[OK] **Realistic device models** - CGM sensor and insulin pump error simulation
[OK] **Commercial pump emulators** - Medtronic 780G, Omnipod 5, Tandem Control-IQ
[OK] **Clinical metrics** - TIR, GMI, CV, LBGI, HBGI per ATTD/ADA guidelines
[OK] **Complete audit trail** - JSONL + CSV + JSON summary with integrity hashing
[OK] **PDF clinical reports** - Visual reports with glucose traces and safety summaries
[OK] **Benchmark mode** - Head-to-head algorithm comparison
[OK] **Real-world data import** - Dexcom, Libre, Nightscout, AIDE/PEDAP, AZT1D, HUPA-UCM
[OK] **Optional AI predictor** - Proactive glucose forecasting
[OK] **Reproducible runs** - Seeded randomness and signable manifests

Intended Use

This SDK is intended for:

- **Pre-clinical algorithm validation**
- **Academic research**
- **Educational purposes**
- **Regulatory submission preparation**

NOT intended for:

- Direct patient care
- Clinical decision-making
- Medical device deployment without regulatory review

2. Getting Started

2.1 Installation Guide

Option 1: Install from PyPI (Recommended)

```
# Create project folder
mkdir my-aps-research && cd my-aps-research

# Create virtual environment (recommended)
python3 -m venv .venv
source .venv/bin/activate  # macOS/Linux
# .venv\Scripts\activate   # Windows

# Install SDK
pip install iints-sdk-python35

# Verify installation
iints --help
```

Option 2: Development Install

```
git clone https://github.com/python35/IINTS-SDK.git
cd IINTS-SDK
pip install -e ".[dev]"
```

Option 3: With Research Extras (AI Predictor)

```
pip install iints-sdk-python35[research]
```

2.2 Your First Simulation (60 seconds)

Using CLI (Quickstart)

```
# Create quickstart project
iints quickstart --project-name iints_quickstart
cd iints_quickstart

# Run clinic-safe preset
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py
```

Using Python API

```
from iints import run_simulation
from iints.core.algorithms.pid_controller import PIDController

# Run 12-hour simulation
results = run_simulation(
    algorithm=PIDController(),
    duration_minutes=720,
    seed=42
)
```

```
print(f"Results saved to: {results['results_csv']}")
print(f"Report generated: {results['clinical_report']}")
```

What this does:

- Creates a virtual patient with default parameters
- Runs PID controller algorithm for 12 hours
- Generates results CSV, clinical report PDF, and audit trail
- Compares against baseline algorithms automatically

Sanity check (first run):

- `results/results.csv` exists and grows during the run
- `results/clinical_reports/` contains a PDF
- Console shows no safety contract violations

Expected runtime:

- Laptop CPU: ~30-90 seconds for the quickstart preset
- GPU not required

2.3 Understanding the Output Files

After running a simulation, you'll find these files in the `results/` folder:

File	Description
`results.csv`	Every simulation step with glucose, insulin, IOB, COB, safety events
`clinical_report.pdf`	Visual report with charts and clinical metrics
`config.json`	Exact configuration used (for reproducibility)
`run_metadata.json`	Run ID, seed, platform info, timestamps
`run_manifest.json`	SHA-256 hashes of all files (integrity verification)
`audit/audit_trail.csv`	Detailed per-step audit trail
`audit/safety_summary.json`	Safety interventions summary
`baseline/pid_results.csv`	PID controller baseline comparison
`baseline/standard_pump.csv`	Standard pump baseline comparison

Quick interpretation:

- Start with `clinical_report.pdf` for a human-readable summary.
- Use `results.csv` for plotting and deeper analysis.
- Use `audit/safety_summary.json` to explain *why* the supervisor intervened.

Data consistency checks:

- `glucose_actual_mgdl` should be in 40-400 mg/dL range.
- `patient_iob_units` and `patient_cob_grams` should be >= 0.
- Large jumps (>60 mg/dL in 5 min) usually indicate data issues.

Example: Loading Results in Python

```
import pandas as pd

# Load main results
df = pd.read_csv("results/your_run/results.csv")
print(df.head())
```

```
# Load safety summary
import json
with open("results/your_run/audit/safety_summary.json") as f:
    safety = json.load(f)
print(f"Total interventions: {safety['intervention_count']}")
```

2.4 Creating Your First Custom Algorithm

Step 1: Generate Template

```
iints new-algo --name MyAlgorithm --output-dir algorithms/
```

Step 2: Implement Logic

```
# algorithms/my_algorithm.py
from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput

class MyAlgorithm(InsulinAlgorithm):
    def get_algorithm_metadata(self):
        return {
            "name": "My Custom Algorithm",
            "version": "0.1.0",
            "description": "My first insulin dosing algorithm"
        }

    def predict_insulin(self, data: AlgorithmInput) -> dict:
        # Simple example: basal rate + correction
        basal = 0.9  # U/hr
        correction = 0.0

        # Add correction if glucose is high
        if data.current_glucose > 180:
            correction = (data.current_glucose - 180) / 50  # ISF of 50

        return {
            "basal_insulin": basal,
            "bolus_insulin": correction,
                      "reason": f"Basal {basal}U + correction {correction}U for glucose
{data.current_glucose}mg/dL"
        }
```

Step 3: Test Your Algorithm

```
# Run with your custom algorithm
iints run --algo algorithms/my_algorithm.py --duration 1440
```

2.5 Adding Stress Tests

Add realistic challenges to test algorithm robustness:

```
from iints.core.simulator import Simulator, StressEvent

sim = Simulator(algorithm=MyAlgorithm(), patient_config="default")

# Add meal at 8:00 AM (480 minutes)
sim.add_stress_event(StressEvent(
    start_time=480,
```

```
    event_type="meal",
    value=60  # 60g carbs
))

# Add exercise at 6:00 PM (1080 minutes)
sim.add_stress_event(StressEvent(
    start_time=1080,
    event_type="exercise",
    value=30  # 30 minutes moderate exercise
))

# Add sensor noise
sim.add_stress_event(StressEvent(
    start_time=300,
    event_type="sensor_noise",
    value=15.0  # 15 mg/dL standard deviation
))

results = sim.run_batch(duration_minutes=1440)
```

Available Stress Events:

- `meal`: Carbohydrate intake (value = grams)
- `exercise`: Physical activity (value = minutes)
- `sensor_noise`: CGM noise (value = std deviation)
- `sensor_dropout`: CGM signal loss (value = probability)
- `pump_failure`: Insulin delivery failure (value = probability)
- `hormonal_change`: Dawn phenomenon simulation

2.6 Benchmarking Against Baselines

Compare your algorithm against standard approaches:

```
# CLI benchmark
iints benchmark \
  --algo-to-benchmark algorithms/my_algorithm.py \
  --patient-configs-dir src/iints/data/virtual_patients \
  --scenarios-dir scenarios \
  --output-dir results/benchmark
```

```
# Python API benchmark
from iints.analysis.benchmark import run_benchmark

results = run_benchmark(
    algorithm_path="algorithms/my_algorithm.py",
    patient_configs=["default", "adolescent", "insulin_resistant"],
    scenarios=["baseline", "meal_challenge", "exercise_stress"],
    duration_minutes=1440
)
```

2.7 Using Preset Scenarios

Clinic-safe scenarios for reproducible testing:

```
# List available presets
iints presets list
```

```
# Run a specific preset
iints presets run --name hypo_prone_night --algo algorithms/my_algorithm.py
```

Available Presets:

- `baseline_t1d`: Standard Type 1 diabetes profile
- `hypo_prone_night`: Nighttime hypoglycemia risk
- `hyper_challenge`: Post-meal hyperglycemia
- `pizza_paradox`: Delayed carb absorption
- `midnight_crash`: Nocturnal hypoglycemia
- `exercise_stress`: Physical activity impact
- `sensor_noise`: CGM accuracy challenges

2.8 Importing Real CGM Data

Test against real patient data:

```
# From CSV file
iints import-data --input-csv my_cgm_export.csv \
  --data-format dexcom \
  --output-dir results/imported
```

```
# Python API
from iints.data.importer import import_cgm_csv
from iints.core.simulator import Simulator

result = import_cgm_csv(
    "my_cgm_export.csv",
    data_format="dexcom",  # or "libre", "generic"
    scenario_name="Patient A - Week 1",
)

sim = Simulator(
    algorithm=MyAlgorithm(),
    scenario=result.scenario,
)
```

Supported Formats:

- Dexcom CSV export
- Libre CSV export
- Generic CSV (auto-detects columns)

Minimum required columns (generic CSV):

- Timestamp (ISO or epoch minutes)
- Glucose (mg/dL)

Optional but recommended:

- Carbs (grams)
- Insulin (units)
- Notes/events

Common import pitfalls:

- Mixed timezones or missing timezone offsets
- Glucose in mmol/L (convert to mg/dL)

- Duplicate timestamps (keep the latest reading)

Quick validation:

- Plot glucose vs time to confirm it is smooth and within 40-400 mg/dL
- Ensure meal events align with glucose rises (15-90 minutes after)
- Nightscout JSON
- Dataset registry packs (AIDE, PEDAP, AZT1D, HUPA-UCM)

2.9 Generating Custom Reports

```
from iints.analysis.reporting import ClinicalReportGenerator

generator = ClinicalReportGenerator()
generator.generate_pdf(
    results_df,  # Your simulation results DataFrame
    safety_report,  # Safety summary dict
    "results/custom_report.pdf",
    title="My Algorithm Performance Report",
    include_trend_analysis=True,
    highlight_safety_events=True
)
```

Report Contents:

- Glucose trace chart with target range
- Insulin delivery chart
- Clinical metrics table (TIR, GMI, CV, LBGI, HBGI)
- Safety interventions summary
- Top intervention reasons
- Configuration overview

2.10 What Next?

Recommended Next Steps:

- [OK] Complete the Getting Started guide
- [Metrics] Run benchmark comparisons
- [Research] Test with stress scenarios
- [Performance] Import real CGM data
- [AI] Explore AI predictor integration
- [Notes] Generate clinical reports
- [Training] Review safety architecture
- [Tools] Customize patient profiles
- [Package] Package algorithm for distribution
- [Announcement] Share results with community

Need Help?

- Check Troubleshooting section (9.0)
- Review FAQ (17.0)
- Join community discussions

Submit issues on GitHub

3. Architecture Overview

3.1 System Components

```
IINTS-AF SDK Architecture

   Algorithm Layer    Safety Layer       Output
                                       Layer

 - Custom Algorithm  - Input Validator   - Results
 - PID Controller    - Safety Supervisor - Reports
 - ML Predictor      - Safety Config     - Audit
 - Pump Emulators                       - Metrics




                 Patient Simulation

 - Virtual Patient Model
 - CGM Sensor Model (noise, lag, dropout)
 - Insulin Pump Model (limits, quantization)
 - Pharmacokinetics (insulin absorption)
 - Pharmacodynamics (glucose response)
```

3.2 Data Flow

```
1. Algorithm requests insulin dose

2. Input Validator checks glucose values

3. Safety Supervisor applies 9 safety checks

4. Approved dose sent to pump model

5. Pump model simulates delivery (with possible errors)

6. Patient model calculates glucose impact

7. CGM sensor model adds noise/lag

8. New glucose reading returned to algorithm

9. Audit trail logs all decisions

10. Repeat every time step (default: 5 minutes)
```

3.3 Safety Layer Integration

The Independent Safety Supervisor runs **deterministically** and can:

[OK] Override dangerous algorithm requests

[OK] Log all interventions with reasons

[OK] Enforce hard limits (hypoglycemia protection)

[OK] Apply dynamic limits (IOB clamping)

[OK] Validate all inputs/outputs

Key Principle: Safety layer is **always active** and cannot be disabled.

4. Safety Architecture (Critical Section)

4.1 Design Philosophy

Safety-First Principles:

Deterministic Overrides: Same input → same safety decision

Fail-Safe Defaults: When in doubt, reduce insulin

Audit Everything: Every decision logged for accountability

Transparent Logic: Clear reasons for all interventions

Configurable Thresholds: Adapt to different patient profiles

Safety Guarantees:

No algorithm can deliver unsafe doses

Hypoglycemia protection is absolute (< 40 mg/dL emergency stop)

All interventions are logged and explainable

Configuration is validated before simulation starts

4.2 SafetyConfig Configuration

```
from iints.core.safety import SafetyConfig

# Default configuration (clinic-safe)
config = SafetyConfig(
    # Hypoglycemia protection
    hypo_cutoff=70.0,  # mg/dL - start reducing insulin
    severe_hypo_cutoff=54.0,  # mg/dL - emergency stop
    critical_hypo_cutoff=40.0,  # mg/dL - immediate termination

    # Hyperglycemia limits
    hyper_cutoff=300.0,  # mg/dL - maximum allowed

    # Insulin limits
    max_basal_rate=2.0,  # U/hr
    max_bolus=5.0,  # U per bolus
    max_iob=10.0,  # U total active insulin

    # Rate limits
    max_insulin_per_hour=15.0,  # U/hr rolling window
    max_insulin_per_day=80.0,  # U/day absolute limit

    # Trend protection
    contract_enabled=True,
    contract_glucose_threshold=90.0,  # mg/dL
    contract_trend_threshold=-1.0,  # mg/dL per 5 minutes
)
```

When to tune SafetyConfig

Only after you can reproduce a baseline run with stable glucose and no crashes.

Increase strictness (lower cutoffs / lower max rates) when testing new or unstable algorithms.

Relax cutoffs only for controlled research experiments with full audit logs.

Recommended baseline ranges (adult research)

`max_bolus`: 2-6 U

`max_basal_rate`: 1-3 U/hr

`max_iob`: 6-12 U

`hyper_cutoff`: 250-300 mg/dL

Audit note: All SafetyConfig values are written to `run_metadata.json` and `audit/safety_summary.json`.

4.3 9 Safety Checks Explained

The IndependentSupervisor applies these checks **in order**:

```
1. Predictive Hypo Guard [EMERGENCY]
   - If glucose < 70 AND falling fast (-3+ mg/dL per 5min)
   - Action: Suspend insulin for 30 minutes
   - Rationale: Prevent imminent severe hypo

2. Basal Rate Limit [WARNING]
   - If basal > max_basal_rate
   - Action: Cap at max_basal_rate
   - Rationale: Prevent basal overdose

3. Hard Hypo Cutoff [EMERGENCY]
   - If glucose < 54 mg/dL
   - Action: Suspend all insulin
   - Rationale: Severe hypoglycemia protection

4. Severe Hypo Emergency Stop [EMERGENCY]
   - If glucose < 40 mg/dL
   - Action: Terminate simulation
   - Rationale: Critical hypoglycemia - stop everything

5. Glucose Level Clamp [CRITICAL/WARNING]
   - If glucose > 300 mg/dL
   - Action: Reduce insulin by 50%
   - Rationale: Prevent over-correction

6. Rate-of-Change Trend Stop [CRITICAL]
   - If glucose falling > 5 mg/dL per 5min
   - Action: Suspend insulin
   - Rationale: Rapid drop protection

7. Dynamic IOB Clamp [WARNING]
   - If IOB > max_iob
   - Action: Reduce dose to stay under max_iob
   - Rationale: Prevent insulin stacking

8. Bolus Stacking Check [WARNING]
   - If recent boluses > safety limit
   - Action: Delay or reduce bolus
   - Rationale: Prevent bolus overlap
```

```
9. 60-Minute Rolling Cap [WARNING]
   - If insulin in last 60min > max_insulin_per_hour
   - Action: Reduce dose to stay under limit
   - Rationale: Hourly limit enforcement
```

4.4 Safety Levels

Level	Severity	Action
INFO	Informational	Log only, no intervention
WARNING	Potential issue	Adjust dose within safe limits
CRITICAL	Serious risk	Significant dose reduction
EMERGENCY	Immediate danger	Suspend insulin completely

4.5 Input Validation

The InputValidator checks:

```
# Glucose range validation
if glucose < 20 or glucose > 600:
    raise InvalidGlucoseError(f"Glucose {glucose} outside valid range")

# Insulin request validation
if insulin < 0 or insulin > config.max_bolus:
    raise InvalidInsulinError(f"Insulin {insulin} invalid")

# Timestep validation
if timestep < 1 or timestep > 15:
    raise InvalidTimestepError(f"Timestep {timestep} invalid")
```

4.6 Simulation Termination

Automatic termination occurs when:

- **Critical hypoglycemia**: Glucose < 40 mg/dL for 30+ minutes
- **Configuration error**: Invalid safety configuration
- **Algorithm error**: Unhandled exception in algorithm
- **Manual stop**: User interrupts simulation

Termination Output:

- `SimulationLimitError` exception raised
- Safety report marks `terminated_early: true`
- Final glucose and intervention reason logged
- Partial results still saved

5. Tutorials and Cookbook

5.1 24-Hour Simulation Walkthrough

Complete example from setup to analysis:

```
import iints
import pandas as pd
import matplotlib.pyplot as plt
from iints.core.algorithms.pid_controller import PIDController
```

```
from iints.core.simulator import Simulator, StressEvent

# 1. Setup simulation
sim = Simulator(
    algorithm=PIDController(),
    patient_config="default",
    time_step=5,  # 5-minute steps
    enable_profiling=True
)

# 2. Add realistic stress events
sim.add_stress_event(StressEvent(start_time=480, event_type="meal", value=60))   # 8:00 AM
breakfast
sim.add_stress_event(StressEvent(start_time=720, event_type="meal", value=45))   # 12:00 PM
lunch
sim.add_stress_event(StressEvent(start_time=1080, event_type="exercise", value=45))   # 6:00
PM workout
sim.add_stress_event(StressEvent(start_time=1320, event_type="meal", value=75))   # 8:00 PM
dinner

# 3. Run 24-hour simulation
results_df, safety_report = sim.run_batch(duration_minutes=1440)

# 4. Analyze results
print(f"Time in Range (70-180 mg/dL): {iints.metrics.calculate_tir(results_df):.1f}%")
print(f"Glucose Management Indicator: {iints.metrics.calculate_gmi(results_df):.1f}")
print(f"Safety interventions: {safety_report['intervention_count']}")

# 5. Visualize
plt.figure(figsize=(12, 6))
plt.plot(results_df['timestamp'], results_df['glucose_actual_mgdl'])
plt.axhline(180, color='red', linestyle='--', label='Hyperglycemia')
plt.axhline(70, color='green', linestyle='--', label='Target')
plt.axhline(54, color='orange', linestyle='--', label='Hypoglycemia')
plt.title('24-Hour Glucose Profile')
plt.xlabel('Time')
plt.ylabel('Glucose (mg/dL)')
plt.legend()
plt.grid(True)
plt.show()

# 6. Generate report
iints.generate_clinical_report(
    results_df,
    safety_report,
    "results/24hour_report.pdf"
)
```

5.2 Building an ML-Hybrid Algorithm

Combine ML prediction with rule-based safety:

```
from iints.api.base_algorithm import InsulinAlgorithm
from iints.research.predictor import load_predictor_service

class MLHybridAlgorithm(InsulinAlgorithm):
```

```
    def __init__(self, predictor_path="models/predictor.pt"):
        super().__init__()
        self.predictor = load_predictor_service(predictor_path)
        self.last_prediction = None

    def predict_insulin(self, data: AlgorithmInput) -> dict:
        # 1. Get ML prediction (30-min forecast)
        prediction = self.predictor.predict(data)
        self.last_prediction = prediction['glucose_forecast']

        # 2. Rule-based decision with ML insight
        basal = 0.9  # U/hr
        bolus = 0.0

        # 3. Adjust based on prediction
        if prediction['glucose_forecast'] > 200:
            # Aggressive correction if rising
            bolus = (prediction['glucose_forecast'] - 180) / 40
        elif prediction['glucose_forecast'] < 90:
            # Conservative if dropping
            basal = max(0.3, basal * 0.7)  # Reduce basal but don't suspend

        return {
            "basal_insulin": basal,
            "bolus_insulin": bolus,
            "reason": f"ML forecast: {prediction['glucose_forecast']:.0f}mg/dL"
        }
```

5.3 Running Batch Experiments

Test multiple configurations efficiently:

```
from iints.analysis.batch import run_batch_experiment

configurations = [
    {"algorithm": "PIDController", "patient": "default", "scenario": "baseline"},
    {"algorithm": "PIDController", "patient": "adolescent", "scenario": "meal_challenge"},
    {"algorithm": "MyAlgorithm", "patient": "default", "scenario": "baseline"},
    {"algorithm": "MyAlgorithm", "patient": "adolescent", "scenario": "meal_challenge"},
]

results = run_batch_experiment(
    configurations=configurations,
    duration_minutes=1440,
    output_dir="results/batch_experiment",
    parallel_workers=4  # Use 4 CPU cores
)

# Compare metrics across all runs
comparison_df = results.compare_metrics()
print(comparison_df[['algorithm', 'patient', 'TIR', 'GMI', 'interventions']])
```

5.4 Audit Trail Analysis

Every run produces a structured audit trail that explains why the safety layer intervened.

```
import pandas as pd
```

```
audit = pd.read_csv("results/your_run/audit/audit_trail.csv")
print(audit[['timestamp', 'glucose_actual_mgdl', 'action', 'reason']].head())

# Count interventions by type
print(audit['action'].value_counts())
```

Interpretation tips:

- If `action` is `suspend` or `cap`, the supervisor overrode the algorithm.
- If `reason` repeats often, your algorithm is too aggressive for that patient profile.

5.5 Custom Safety Thresholds

Use SafetyConfig to tighten or relax constraints for research experiments.

```
from iints.core.safety import SafetyConfig
from iints.core.simulator import Simulator

safe_config = SafetyConfig(
    max_bolus=3.0,
    max_iob=8.0,
    hyper_cutoff=250.0,
)

sim = Simulator(algorithm=MyAlgorithm(), safety_config=safe_config)
```

5.6 Pump Emulator Benchmarking

Benchmark alternative pump behaviors or commercial-emulator presets.

```
from iints.analysis.hardware_benchmark import benchmark_pump_emulators

bench = benchmark_pump_emulators(duration_minutes=120)
print(bench[['model', 'avg_step_ms', 'max_step_ms']])
```

5.7 Live Streaming Simulation

Stream real-time values for dashboards or demos.

```
from iints.core.simulator import Simulator

sim = Simulator(algorithm=MyAlgorithm())
for state in sim.run_stream(duration_minutes=120):
    print(state['timestamp'], state['glucose_actual_mgdl'])
```

5.8 Reproducible Runs for Publications

To make results reproducible:

- Fix `seed`
- Persist `config.json`
- Record dataset hash and commit SHA

```
results = iints.run_simulation(
    algorithm=MyAlgorithm(),
    duration_minutes=720,
    seed=123,
)
print(results['run_manifest'])
```

6. API Reference

6.1 Core Classes

InsulinAlgorithm (Abstract Base Class)

```
from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput, AlgorithmResult

class MyAlgorithm(InsulinAlgorithm):
    def get_algorithm_metadata(self) -> dict:
        """Return algorithm identification and version info"""
        return {
            "name": "MyAlgorithm",
            "version": "1.0.0",
            "description": "My custom insulin dosing algorithm",
            "author": "Your Name",
            "reference": "Optional citation or paper reference"
        }

    def predict_insulin(self, data: AlgorithmInput) -> dict:
        """
        Calculate insulin dose based on current data

        Args:
            data: AlgorithmInput containing current state

        Returns:
            dict with keys: basal_insulin, bolus_insulin, reason
        """
        # Your algorithm logic here
        return {
            "basal_insulin": 0.9,  # U/hr
            "bolus_insulin": 0.0,  # U
            "reason": "Stable glucose, maintaining basal rate"
        }

    def reset(self):
        """Reset algorithm state for new simulation"""
        pass
```

AlgorithmInput (Dataclass)

```
@dataclass
class AlgorithmInput:
    # Current state
    current_glucose: float  # mg/dL
    current_time: datetime  # Simulation timestamp

    # Historical context
    glucose_history: List[float]  # Last 24 hours (5-min intervals)
    insulin_history: List[float]  # Last 24 hours
    carb_history: List[float]  # Last 24 hours

    # Calculated values
    iob: float  # Insulin on board (U)
    cob: float  # Carbs on board (g)
```

```
    # Trends
    glucose_trend: float  # mg/dL per 5 minutes
    glucose_acceleration: float  # mg/dL per 5 minutes

    # Patient info
    patient_config: dict  # ISF, ICR, basal rates, etc.

    # Safety context
    last_safety_intervention: Optional[dict]  # Last intervention reason
```

7. Practical Examples

7.1 Complete Algorithm Example

Full working algorithm with comprehensive logic:

```
from iints.api.base_algorithm import InsulinAlgorithm, AlgorithmInput
import numpy as np

class ComprehensiveAlgorithm(InsulinAlgorithm):
    def __init__(self):
        super().__init__()
        self.target_glucose = 120  # mg/dL
        self.isf = 50  # Insulin sensitivity factor
        self.icr = 10  # Insulin-to-carb ratio
        self.basal_rate = 0.9  # U/hr
        self.history = []

    def get_algorithm_metadata(self):
        return {
            "name": "Comprehensive Algorithm",
            "version": "1.0.0",
            "description": "Full-featured algorithm with meal detection and trend analysis"
        }

    def predict_insulin(self, data: AlgorithmInput) -> dict:
        # Store history for trend analysis
        self.history.append(data.current_glucose)
        if len(self.history) > 24:
            self.history.pop(0)

        # Calculate correction bolus
        correction = 0.0
        if data.current_glucose > self.target_glucose:
            correction = (data.current_glucose - self.target_glucose) / self.isf

        # Meal detection (simple version)
        meal_bolus = 0.0
        if data.carb_history and data.carb_history[-1] > 0:
            meal_bolus = data.carb_history[-1] / self.icr

        # Trend adjustment
        trend_adjustment = 0.0
        if data.glucose_trend > 2:  # Rising fast
            correction *= 1.2  # More aggressive
```

```
        elif data.glucose_trend < -2:  # Dropping fast
            correction *= 0.8  # More conservative

        # IOB safety
        if data.iob > 5:  # High IOB
            self.basal_rate = max(0.3, self.basal_rate * 0.8)

        # Final dose calculation
        basal = self.basal_rate
        bolus = correction + meal_bolus

        return {
            "basal_insulin": basal,
            "bolus_insulin": bolus,
            "reason": (
                f"Target {self.target_glucose}mg/dL, "
                f"correction {correction:.2f}U, "
                f"meal {meal_bolus:.2f}U, "
                f"trend {data.glucose_trend:.1f}mg/dL per 5min"
            )
        }

    def reset(self):
        self.history = []
```

7.2 CLI vs Python API Comparison

Same Task: Run Simulation with Custom Algorithm

CLI Version:

```
# Create algorithm
iints new-algo --name MyAlgorithm --output-dir algorithms/

# Edit algorithm file
nano algorithms/my_algorithm.py

# Run simulation
iints run \
  --algo algorithms/my_algorithm.py \
  --patient-config-name default \
  --scenario-path scenarios/meal_challenge.json \
  --duration 1440 \
  --output-dir results/cli_run
```

Python Version:

```
from iints import run_simulation
from iints.core.algorithms.custom import MyAlgorithm

results = run_simulation(
    algorithm=MyAlgorithm(),
    patient_config="default",
    scenario="scenarios/meal_challenge.json",
    duration_minutes=1440,
    output_dir="results/python_run"
)
```

When to Use CLI:

- Quick testing and iteration
- Batch processing multiple scenarios
- Integration with shell scripts
- CI/CD pipelines

When to Use Python API:

- Complex algorithm development
- Integration with other Python tools
- Custom analysis pipelines
- Jupyter notebook exploration

7.3 Stress Testing Patterns

Pattern 1: Meal Challenge Test

```
# Test algorithm response to large meal
sim.add_stress_event(StressEvent(
    start_time=480,  # 8:00 AM
    event_type="meal",
    value=100  # 100g carbs (large pizza meal)
))
```

Pattern 2: Exercise Stress Test

```
# Test algorithm response to exercise
sim.add_stress_event(StressEvent(
    start_time=1080,  # 6:00 PM
    event_type="exercise",
    value=60  # 60 minutes intense exercise
))
```

Pattern 3: Sensor Noise Test

```
# Test algorithm robustness to CGM noise
sim.add_stress_event(StressEvent(
    start_time=300,  # 5:00 AM
    event_type="sensor_noise",
    value=20.0  # 20 mg/dL standard deviation
))
```

Pattern 4: Combined Stress Test

```
# Realistic day with multiple stressors
sim.add_stress_event(StressEvent(420, "meal", 45))  # 7:00 AM breakfast
sim.add_stress_event(StressEvent(660, "meal", 60))  # 11:00 AM snack
sim.add_stress_event(StressEvent(900, "exercise", 30))  # 3:00 PM walk
sim.add_stress_event(StressEvent(1020, "meal", 80))  # 5:00 PM dinner
sim.add_stress_event(StressEvent(1200, "sensor_noise", 15.0))  # 8:00 PM sensor issues
```

7.4 Human-in-the-Loop Integration

Add manual interventions during simulation:

```
def human_in_loop_callback(context):
    """
    Called at each simulation step.
    Return dict with manual interventions or None.
    """
```

```
    # Example: Rescue carbs for hypoglycemia
    if context["glucose_actual_mgdl"] < 65:
        return {
            "additional_carbs": 15,  # 15g fast-acting carbs
            "note": "Human intervention: rescue carbs for hypo"
        }

    # Example: Manual bolus correction
    if context["glucose_actual_mgdl"] > 250:
        return {
            "additional_bolus": 1.5,  # 1.5U correction
            "note": "Human intervention: manual correction bolus"
        }

    # Example: Suspend insulin before exercise
    if context["time"] > 1080 and context["time"] < 1140:  # 6-7 PM
        return {
            "suspend_insulin": True,
            "note": "Human intervention: exercise suspension"
        }

    return None  # No intervention

# Use in simulator
sim = Simulator(
    algorithm=MyAlgorithm(),
    patient_config="default",
    on_step=human_in_loop_callback
)
```

7.5 Data Import Workflows

Workflow 1: Dexcom CSV Import

```
# CLI import
iints import-data \
  --input-csv dexcom_export.csv \
  --data-format dexcom \
  --output-dir results/dexcom_import

# Use imported scenario
iints run \
  --algo algorithms/my_algorithm.py \
  --scenario-path results/dexcom_import/scenario.json
```

Workflow 2: Nightscout Import

```
# Install Nightscout extra
pip install iints-sdk-python35[nightscout]

# Import from Nightscout
iints import-nightscout \
  --url https://your-nightscout-site.herokuapp.com \
  --days 7 \
  --output-dir results/nightscout_import
```

Workflow 3: Dataset Registry

```
# List available datasets
```

```
iints data list

# Fetch specific dataset
iints data fetch aide_t1d --output-dir data_packs/aide

# Use in simulation
iints run \
  --algo algorithms/my_algorithm.py \
  --scenario-path data_packs/aide/scenarios/patient_001.json
```

8. Advanced Topics

8.1 Commercial Pump Emulators

Test against real pump behavior:

```
from iints.emulation import Medtronic780G, Omnipod5, TandemControlIQ

# Compare your algorithm against commercial pumps
pumps = [
    ("MyAlgorithm", MyAlgorithm()),
    ("Medtronic 780G", Medtronic780G()),
    ("Omnipod 5", Omnipod5()),
    ("Tandem Control-IQ", TandemControlIQ())
]

results = {}
for name, pump_algo in pumps:
    results[name] = run_simulation(
        algorithm=pump_algo,
        patient_config="default",
        duration_minutes=1440
    )

# Compare metrics
compare_metrics(results)
```

8.2 Dataset Registry Usage

Access real-world datasets:

```
# List all available datasets
iints data list

# Get info about specific dataset
iints data info aide_t1d

# Fetch dataset
iints data fetch aide_t1d --output-dir data_packs/aide

# Cite dataset in publication
iints data cite aide_t1d
```

Available Datasets:

- `aide_t1d`: AIDE Type 1 Diabetes Dataset
- `azt1d`: Arizona Type 1 Diabetes Dataset
- `ohio_t1dm`: OhioT1DM Dataset

`sample`: Bundled demo data (no download needed)

Integrity and reproducibility

Every dataset entry includes a SHA-256 checksum and citation metadata.

The fetch command validates the checksum automatically.

Use `iints data info <dataset>` to record version and hash in your paper.

Typical layout after fetch

```
data_packs/
  public/
    ohio_t1dm/
      raw/...
      processed/...
```

8.3 Reproducibility Techniques

Ensure identical results across runs:

```
# Method 1: Set random seed
results = run_simulation(
    algorithm=MyAlgorithm(),
    seed=42,  # Fixed random seed
    patient_config="default"
)

# Method 2: Use deterministic patient profile
profile = PatientProfile(
    isf=50,
    icr=10,
    basal_rate=0.9,
    dawn_phenomenon=0.3,  # Fixed dawn effect
    seed=42
)

# Method 3: SHA-256 verification
from iints.utils.hashing import verify_manifest

if verify_manifest("results/run_001/run_manifest.json"):
    print("Results verified - not tampered with")
```

8.4 Performance Profiling

Measure algorithm performance:

```
sim = Simulator(
    algorithm=MyAlgorithm(),
    patient_config="default",
    enable_profiling=True
)

results_df, safety_report = sim.run_batch(duration_minutes=1440)

# Access performance data
profiling = safety_report["performance_report"]
print(f"Algorithm latency: {profiling['algorithm_latency_ms']:.2f}ms")
print(f"Safety latency: {profiling['safety_latency_ms']:.2f}ms")
print(f"Total steps: {profiling['total_steps']}")
```

```
print(f"Total time: {profiling['total_time_s']:.2f}s")
```

8.5 Custom Metrics Calculation

Define your own performance metrics:

```
def calculate_custom_metric(results_df):
    """Calculate custom performance metric"""

    # Time in tight range (80-140 mg/dL)
    tight_range = ((results_df['glucose_actual_mgdl'] >= 80) &
                   (results_df['glucose_actual_mgdl'] <= 140)).mean() * 100

    # Glucose variability score
    cv = results_df['glucose_actual_mgdl'].std() /
         results_df['glucose_actual_mgdl'].mean() * 100

    # Hypoglycemia risk score
    hypo_risk = (results_df['glucose_actual_mgdl'] < 70).sum() / len(results_df)

    # Hyperglycemia risk score
    hyper_risk = (results_df['glucose_actual_mgdl'] > 250).sum() / len(results_df)

    # Composite score (lower is better)
    composite_score = (100 - tight_range) + cv + (hypo_risk * 100) + (hyper_risk * 50)

    return {
        'tight_range_percent': tight_range,
        'cv_percent': cv,
        'hypo_risk_percent': hypo_risk * 100,
        'hyper_risk_percent': hyper_risk * 100,
        'composite_score': composite_score
    }

# Use with your results
metrics = calculate_custom_metric(results_df)
print(f"Custom Score: {metrics['composite_score']:.1f}")
```

9. Troubleshooting

9.1 Installation Issues

Issue: ModuleNotFoundError after installation

Solution:

```
# Make sure you're using the correct Python environment
which python  # Should show your virtual environment path

# Reinstall in development mode if needed
pip install -e .
```

Issue: pip install fails with dependency errors

Solution:

```
# Upgrade pip first
pip install --upgrade pip setuptools wheel

# Try installing with --no-cache-dir
```

```
pip install --no-cache-dir iints-sdk-python35

# For specific errors, check Python version
python3 --version  # Must be 3.10+
```

9.2 Simulation Issues

Issue: Simulation runs very slowly

Solution:

```
# Increase time step (default is 5 minutes)
sim = Simulator(time_step=10)  # 10-minute steps

# Disable profiling if not needed
sim = Simulator(enable_profiling=False)

# Reduce duration for testing
results = sim.run_batch(duration_minutes=720)  # 12 hours instead of 24
```

Issue: Simulation terminates early

Solution:

```
# Check safety report for termination reason
print(safety_report['termination_reason'])

# Common causes:
# - Critical hypoglycemia (< 40 mg/dL for 30+ minutes)
# - Algorithm exception
# - Configuration error

# Adjust safety limits if needed
from iints.core.safety import SafetyConfig
config = SafetyConfig(critical_hypo_cutoff=35.0)  # Lower threshold
```

9.3 Algorithm Development Issues

Issue: Algorithm not appearing in CLI

Solution:

```
# Make sure algorithm is properly registered
# 1. Inherits from InsulinAlgorithm
# 2. Implements all required methods
# 3. Has valid get_algorithm_metadata()

# Check with:
from iints.api.registry import list_algorithms
print(list_algorithms())
```

Issue: Safety supervisor blocking all doses

Solution:

```
# Check safety configuration
print(safety_report['config'])

# Common issues:
# - max_basal_rate too low
# - hypo_cutoff too high
# - contract enabled with aggressive settings
```

```
# Adjust configuration:
config = SafetyConfig(
    hypo_cutoff=60.0,  # Higher threshold
    max_basal_rate=1.5  # Higher limit
)
```

9.4 Data Import Issues

Issue: CSV import fails

Solution:

```
# Check CSV format
# Required columns: timestamp, glucose
# Optional: carbs, insulin

# Try auto-detect format
data = import_cgm_csv("your_file.csv", data_format="auto")

# Specify column names manually if needed
data = import_cgm_csv(
    "your_file.csv",
    data_format="generic",
    timestamp_col="Date",
    glucose_col="BG",
    carbs_col="Carbs"
)
```

9.5 Performance Issues

Issue: High memory usage

Solution:

```
# Reduce history size
sim = Simulator(max_history_hours=12)  # Default is 24

# Disable audit trail if not needed
sim = Simulator(enable_audit=False)

# Process in batches
for i in range(10):
    results = sim.run_batch(duration_minutes=144)  # 2.4 hours per batch
    process_results(results)
```

10. Quick Reference

10.1 Essential CLI Commands

```
# Initialize project
iints init --project-name my_project

# Quickstart
iints quickstart --project-name demo

# Run simulation
iints run --algo algorithms/my_algo.py --duration 1440
```

```
# Run with preset
iints presets run --name baseline_t1d --algo algorithms/my_algo.py

# Benchmark
iints benchmark --algo-to-benchmark algorithms/my_algo.py

# Import data
iints import-data --input-csv my_cgm.csv --data-format dexcom

# List presets
iints presets list

# Generate scenario
iints scenarios generate --name "Stress Test"

# Validate files
iints validate --scenario-path scenarios/my_scenario.json
```

10.2 Python Code Snippets

Minimal Simulation:

```
from iints import run_simulation
from iints.core.algorithms.pid_controller import PIDController

results = run_simulation(
    algorithm=PIDController(),
    duration_minutes=720
)
```

Custom Algorithm:

```
from iints.api.base_algorithm import InsulinAlgorithm

class MyAlgorithm(InsulinAlgorithm):
    def predict_insulin(self, data):
        return {"basal_insulin": 0.9, "bolus_insulin": 0.0}
```

Load Results:

```
import pandas as pd
df = pd.read_csv(results['results_csv'])
print(df[['timestamp', 'glucose_actual_mgdl', 'insulin_delivered']].head())
```

10.3 Safety Thresholds (Defaults)

```
SafetyConfig(
    hypo_cutoff=70.0,           # Start reducing insulin
    severe_hypo_cutoff=54.0,    # Emergency suspension
    critical_hypo_cutoff=40.0,  # Terminate simulation
    hyper_cutoff=300.0,         # Maximum glucose
    max_basal_rate=2.0,         # U/hr
    max_bolus=5.0,              # U per bolus
    max_iob=10.0,               # U total active insulin
    max_insulin_per_hour=15.0,  # U/hr rolling window
    max_insulin_per_day=80.0    # U/day absolute limit
)
```

10.4 Clinical Metrics Targets (ATTD/ADA)

Metric	Target Range
TIR (70-180 mg/dL)	>70%
TBR (<70 mg/dL)	<4%
TBR (<54 mg/dL)	<1%
GMI	<7.0%
CV	<36%
LBGI	<1.1
HBGI	<5.0

11. Glossary

Algorithm: Code that calculates insulin doses based on current and historical data

Basal Insulin: Background insulin delivery rate (U/hr)

Bolus Insulin: Additional insulin for meals or corrections (U)

CGM: Continuous Glucose Monitor - device that measures glucose every 5 minutes

COB: Carbs On Board - carbohydrates still being absorbed

GMI: Glucose Management Indicator - estimate of A1C from CGM data

IOB: Insulin On Board - insulin still active in the body

ISF: Insulin Sensitivity Factor - how much 1U of insulin lowers glucose (mg/dL)

ICR: Insulin-to-Carb Ratio - grams of carbs covered by 1U of insulin

TIR: Time In Range - percentage of time in target glucose range (70-180 mg/dL)

TBRL1: Time Below Range Level 1 - percentage of time <70 mg/dL

TBRL2: Time Below Range Level 2 - percentage of time <54 mg/dL

TAR: Time Above Range - percentage of time >180 mg/dL

CV: Coefficient of Variation - measure of glucose variability

LBGI: Low Blood Glucose Index - measure of hypoglycemia risk

HBGI: High Blood Glucose Index - measure of hyperglycemia risk

Need More Help?

Documentation:

- Comprehensive Guide: `docs/COMPREHENSIVE_GUIDE.md`
- Technical README: `docs/TECHNICAL_README.md`
- API Stability: `API_STABILITY.md`

Community:

- GitHub Issues: https://github.com/python35/IINTS-SDK/issues
- Discussion Forum: [Link]
- Email Support: support@iints.org

Citing IINTS-AF:

```
@software{IINTS_AF_SDK,
  author = {Bobbaers, Rune},
  title = {IINTS-AF: Intelligent Insulin Titration System for Artificial Pancreas},
  year = {2026},
```

```
  version = {0.1.19},
  url = {https://github.com/python35/IINTS-SDK},
  note = {Pre-clinical research software for insulin dosing algorithm validation}
}
```

PRE-CLINICAL USE ONLY - NOT FOR PATIENT CARE

This document is provided for research and educational purposes only.
The IINTS-AF SDK is not a medical device and should not be used for clinical decision-making.