
IINTS-AF SDK Manual

Research Use Only — Not for Clinical Care

Research, Benchmarking, and Edge Operations

IINTS-AF SDK

Version 1.5.6

Research Use Only — Not for Clinical Care

Safety-first simulation, deterministic supervision, and reproducible study workflows for insulin algorithm development.

Contents

0.1	How To Use This Manual	2
0.2	Fastest Paths Through The SDK	2
0.3	Manual Map	2
0.4	Canonical Workflows	3
0.5	Executive Summary	3
0.6	Getting Started	4
0.7	Architecture Overview	8
0.8	Safety Architecture (Critical Section)	9
0.9	Tutorials and Cookbook	12
0.10	API Reference	16
0.11	Practical Examples	18
0.12	Advanced Topics	22
0.13	Poster-Ready Results for Jury Demos	30
0.14	Booth / Jury Demo Bundle	30
0.15	Troubleshooting	32
0.16	Quick Reference	35
0.17	Glossary	37
0.18	Need More Help?	38

“Code shouldn’t be a secret when it’s managing a life.”

Version 1.5.6

Research use only — not for clinical care

IINTS-AF is an open-source research platform for insulin algorithm simulation, deterministic safety supervision, dataset certification, reproducible benchmarking, and edge deployment. It exists for the people who need to inspect, test, and explain dosing logic before it ever touches a real-world device.

Core research question

Can open-source simulation and deterministic safety supervision make insulin delivery algorithm development safer and more transparent for researchers and patients?

0.1 How To Use This Manual

This manual is the long-form technical guide for the SDK. It is the right place to start when you want more than a quickstart: a clear mental model of the system, a reproducible study workflow, and concrete guidance for local AI, data certification, and Raspberry Pi / Arduino showcase deployments.

0.2 Fastest Paths Through The SDK

If you want to...	Start here	Main command
Confirm the SDK works on this machine	Installation Guide	<code>iints doctor</code> <code>--smoke-run --suggest</code>
See a zero-config run first	Your First Simulation	<code>iints demo</code>
Build your own run interactively	Your First Simulation	<code>iints run --wizard</code>
Create a reusable project folder	Your First Simulation	<code>iints quickstart</code> <code>--project-name my_study</code>
Build a publication-ready bundle	Reproducible Runs For Publications	<code>iints study-ready</code> <code>--algo algorithms/example_algorithm.py</code> <code>--output-dir results/study_ready</code>
Run a benchmark protocol	Scientific Study Workflow	<code>iints study-protocol</code> then <code>iints run-study</code>
Start a live digital patient on Raspberry Pi	Booth / Jury Demo Bundle	<code>iints makerfaire up</code> <code>--project-dir .</code>

0.3 Manual Map

- Executive summary: what the SDK is, who it is for, and why the safety model matters.
- Getting started: install, first run, quick validation, and your first custom algorithm.
- Architecture and safety: how simulation, supervision, and audit output fit together.
- Cookbook and examples: practical CLI and Python patterns you can adapt quickly.
- Advanced workflows: study protocols, reproducibility, datasets, local AI, and jury/demo packaging.
- Troubleshooting and quick reference: the shortest route out of common setup or runtime problems.

0.4 Canonical Workflows

1. First-run path: `iints onboard -> iints demo -> iints quickstart`.
2. Algorithm development path: generate an algorithm template, run clinic-safe presets, inspect audit output, then benchmark against baselines.
3. Research path: write a protocol with `iints study-protocol`, execute with `iints run-study`, analyze with `iints analyze`, compare with `iints compare-study`, and package results with `iints poster-study` or `iints eucys-results`.
4. Edge/demo path: scaffold an SBC project with `iints edge setup`, then run the Pi-only Maker Faire flow with `iints makerfaire up`, `iints makerfaire autostart`, and `iints makerfaire watchdog`.

0.5 Executive Summary

The IINTS-AF SDK (Intelligent Insulin Titration System for Artificial Pancreas) is a safety-first simulation and validation platform for insulin delivery research. Instead of treating dosing logic as a black box, it gives researchers, students, and patient-builders a way to inspect how an algorithm behaves, how the deterministic supervisor constrains it, and how the resulting evidence can be packaged for review.

0.5.1 Why This Platform Exists

Commercial insulin-delivery algorithms typically run behind closed interfaces. That makes them difficult to audit, difficult to compare fairly, and difficult for patients or researchers to improve. IINTS-AF addresses that gap by combining:

- Open simulation for virtual patients, meals, exercise, device noise, and failure modes.
- Deterministic safety supervision that can override unsafe algorithm proposals.
- Reproducible study tooling that turns one-off runs into protocol-driven benchmarks.
- Audit-grade outputs such as manifests, summaries, reports, posters, and evidence tables.
- Edge deployment paths for Raspberry Pi booths, local AI explanations, and hardware-linked demos.

0.5.2 Key Capabilities

- Algorithm sandbox — load a custom controller or fall back to the built-in Clinical Baseline.
- Deterministic Independent Safety Supervisor — nine safety checks, bounded actions, and explicit intervention logs.
- Study engine — protocol bundles, matrix generation, baseline registries, and cohort execution.
- Clinical and research metrics — TIR, TBR, TAR, variability, interventions, calibration, and uncertainty summaries.
- Traceability — manifests, hashes, dataset certification, and output bundles for review or publication.

- Local-first AI layer — optional Minstral/Ollama workflows for explanation and review on your own hardware.
- Edge operations — Raspberry Pi patient runtime, kiosk mode, watchdogs, and optional Arduino UNO Q signaling.

0.5.3 Who This Manual Is For

- Researchers who need reproducible benchmark workflows and publication-grade outputs.
- Algorithm developers who want to iterate safely and compare against transparent baselines.
- Students and science-fair builders who need one clear path from demo to evidence.
- Patient-builders and data explorers who want to inspect imports, certification, and audit trails without cloud lock-in.

0.5.4 Intended Use

This SDK is intended for: - pre-clinical algorithm validation - academic or student research - technical demonstrations and safety reviews - educational exploration of insulin-delivery logic

It is not intended for: - direct patient care - live dosing advice - deployment as a medical device without regulatory review

0.5.5 The Short Version

If you only remember four ideas from this manual, make them these:

1. The algorithm may suggest, but the deterministic supervisor decides.
2. One attractive run is not enough; protocol-driven studies matter.
3. Certified and traceable data matter as much as controller quality.
4. The SDK is strongest when it is used as a platform to understand algorithms, not just to run them.

0.6 Getting Started

0.6.1 Installation Guide

Choose the smallest path that matches what you want to do.

0.6.1.1 Option 1: Install From PyPI (Recommended)

```
python3 -m venv .venv
source .venv/bin/activate # macOS/Linux
# .venv\Scripts\activate # Windows
python -m pip install -U pip
python -m pip install -U "iints-sdk-python35[full,mdmp]"
iints doctor --smoke-run --suggest
```

Use this when you want the normal CLI, data certification, plotting, and the local AI hooks.

0.6.1.2 Option 2: Edge Install For Raspberry Pi / Arduino UNO Q

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip
python -m pip install -U "iints-sdk-python35[edge,mdmp]"
iints doctor --full --suggest
```

Then scaffold an edge project:

```
iints edge setup --output-dir iints_edge_demo --board raspberry_pi
cd iints_edge_demo
./run_edge_patient.sh
```

0.6.1.3 Option 3: Development Install

```
git clone https://github.com/python35/IINTS-SDK.git
cd IINTS-SDK
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip
python -m pip install -U -e ".*[full,mdmp]"
```

Use the development install when you are modifying the SDK itself or running the full test/documentation toolchain.

0.6.1.4 Optional: Add Ollama For Local AI If you want local research-only explanation and review features, connect a local Ollama runtime:

```
curl -fsSL https://ollama.com/install.sh | sh
ollama -v
ollama serve
ollama pull minstral-3:8b
export OLLAMA_HOST=http://127.0.0.1:11434
iints ai local-check --model minstral-3:8b
```

Practical notes: - the SDK defaults to `http://127.0.0.1:11434` when `OLLAMA_HOST` is not set - use `minstral-3:3b` on smaller systems - remote Ollama hosts are blocked by default unless explicitly enabled - for a deeper walkthrough, use the public AI guide on the docs site

0.6.2 Your First Simulation

The SDK now has three beginner-safe starting points.

0.6.2.1 Fastest possible: zero-config demo

```
iints demo
iints demo --dry-run
iints demo --full
```

Use demo when you want a working run immediately, without creating files by hand.

0.6.2.2 Guided path: interactive run builder

```
iints run --wizard
```

This is the best path when you know roughly what you want, but do not want to memorize flags yet.

0.6.2.3 Reusable path: create a project folder

```
iints quickstart --project-name iints_quickstart
cd iints_quickstart
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py
```

If you leave `--algo` blank in the generic `iints run` flow, the SDK will use the built-in Clinical Baseline.

0.6.2.4 Equivalent Python API example

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

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']}")
```

0.6.3 What A Good First Run Should Produce

After a healthy first run, you should have:

- a `results.csv` timeline that grows during execution
- a human-readable PDF report under `results/clinical_reports/`
- a manifest and metadata bundle that records the run configuration
- an audit summary that explains when and why the supervisor intervened

0.6.4 Understanding The Output Files

After running a simulation, look for these files inside the run directory:

File	Why it matters
results.csv	Step-by-step glucose, insulin, IOB, COB, events, and actions
clinical_report.pdf	Human-readable overview for quick review
config.json	Exact run configuration for reproducibility
run_metadata.json	Run ID, timestamps, platform details
run_manifest.json	File hashes and integrity manifest
audit/audit_trail.csv	Detailed intervention log
audit/safety_summary.json	Supervisor counts and intervention reasons
baseline/	Optional baseline comparison outputs

Quick interpretation: - start with the PDF if you need the story fast - open safety_summary.json if you need to explain a surprising result - open results.csv when you need full numerical detail or your own plots

0.6.5 Creating Your First Custom Algorithm

0.6.5.1 Step 1: Generate a template

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

0.6.5.2 Step 2: Implement the logic

```
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:
        basal = 0.9
        correction = 0.0
        if data.current_glucose > 180:
            correction = (data.current_glucose - 180) / 50
        return {
            "basal_insulin": basal,
            "bolus_insulin": correction,
            "reason": f"Basal {basal}U + correction {correction}U for glucose
↪ {data.current_glucose}mg/dL",
        }
```


0.6.5.3 Step 3: Validate without surprises

```
iints run --algo algorithms/my_algorithm.py --preset baseline_t1d --dry-run
iints run --algo algorithms/my_algorithm.py --preset baseline_t1d
```

If the algorithm path is wrong, the newer CLI will suggest likely paths instead of failing silently.

0.6.6 Adding Stress Tests

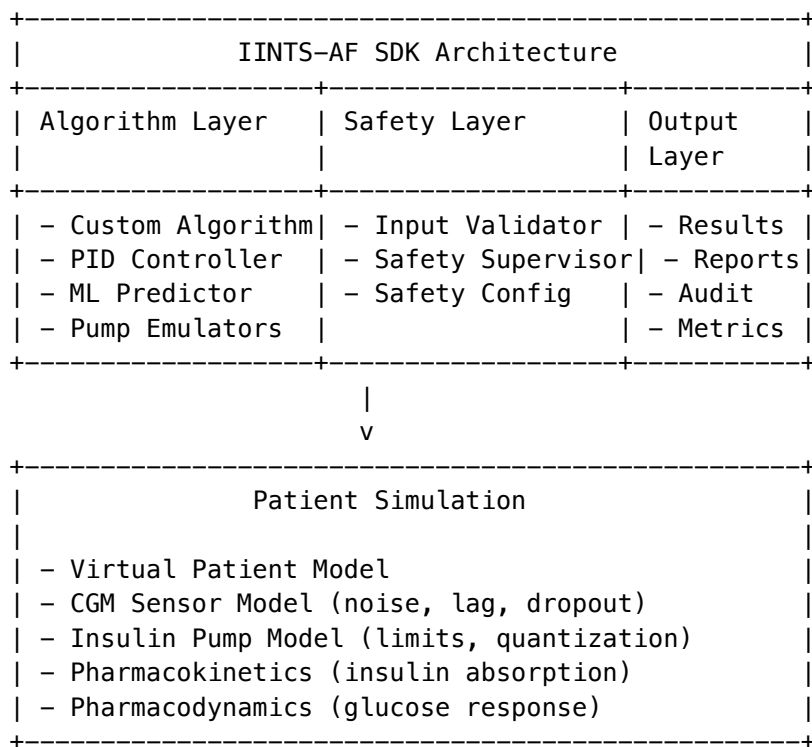
Use stress scenarios when you want to learn something real about robustness rather than just produce a pretty glucose trace.

Common failure-mode questions - How does the controller handle a large delayed meal? - Does sensor noise cause unsafe over-correction? - Can the safety layer recover when the controller becomes too aggressive? - What happens during exercise or a simulated pump issue?

The quickest CLI path is to start from a preset such as `baseline_t1d`, `exercise_stress`, or `sensor_noise`. The Python path remains useful when you need custom event timing.

0.7 Architecture Overview

0.7.1 System Components



0.7.2 Data Flow

1. Algorithm requests insulin dose
v
2. Input Validator checks glucose values
v
3. Safety Supervisor applies 9 safety checks
v
4. Approved dose sent to pump model

- v
- 5. Pump model simulates delivery (with possible errors)
- v
- 6. Patient model calculates glucose impact
- v
- 7. CGM sensor model adds noise/lag
- v
- 8. New glucose reading returned to algorithm
- v
- 9. Audit trail logs all decisions
- v
- 10. Repeat every time step (default: 5 minutes)

0.7.3 Safety Layer Integration

The Independent Safety Supervisor runs deterministically and can: - Override dangerous algorithm requests - Log all interventions with reasons - Enforce hard limits (hypoglycemia protection) - Apply dynamic limits (IOB clamping) - Validate all inputs/outputs

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

0.8 Safety Architecture (Critical Section)

0.8.1 Design Philosophy

Safety-First Principles:

1. Deterministic Overrides: Same input -> same safety decision
2. Fail-Safe Defaults: When in doubt, reduce insulin
3. Audit Everything: Every decision logged for accountability
4. Transparent Logic: Clear reasons for all interventions
5. 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

0.8.2 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.

0.8.3 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

0.8.4 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

0.8.5 Input Validation

The InputValidator checks:

```

# Broad CGM/sensor plausibility validation
if glucose < 40 or glucose > 500:
    raise InvalidGlucoseError(f"Glucose {glucose} outside broad sensor-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")

```

0.8.6 Simulation Termination

Automatic termination occurs when:

1. Critical hypoglycemia: Glucose < 40 mg/dL for 30+ minutes
2. Configuration error: Invalid safety configuration
3. Algorithm error: Unhandled exception in algorithm
4. 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

0.9 Tutorials and Cookbook

0.9.1 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"
)

```

0.9.2 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"
        }

```

0.9.3 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']])

```

0.9.4 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.

0.9.5 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)

```


0.9.6 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']])
```

0.9.7 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'])
```

0.9.8 Reproducible Runs For Publications

When you want evidence instead of a one-off run, move from ad hoc simulations to a documented study workflow.

Minimum reproducibility checklist - fix seeds - save the run config and manifest - record dataset provenance and certification status - keep the exact algorithm file that was executed - preserve comparison outputs, not just the best trace

Single-bundle path

```
iints study-ready \
  --algo algorithms/example_algorithm.py \
  --output-dir results/study_ready
```

Protocol-driven study path

```
iints study-protocol --output-dir results/study_protocol
iints run-study --experiment results/study_protocol/study_experiment.yaml
iints analyze \
  results/study_bundle \
  --output-json results/study_summary.json
iints compare-study \
  results/study_clean \
  results/study_corrupted \
  --output-json results/study_comparison.json
```

Use the single-bundle path when you want one traceable result package. Use the protocol path when you are doing real benchmarking, ablations, or science-fair / paper workflows.

0.10 API Reference

0.10.1 Core Classes

0.10.1.1 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

```

0.10.1.2 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^2

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

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

```

0.11 Practical Examples

0.11.1 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 = []

```

0.11.2 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

0.11.3 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
```

0.11.4 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
)
```

0.11.5 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
export IINTS_NIGHTSCOUT_TOKEN="replace-me"
iints import-nightscout \
  --url https://your-nightscout-site.herokuapp.com \
  --token-env IINTS_NIGHTSCOUT_TOKEN \
  --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 --no-verify

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

0.12 Advanced Topics

0.12.1 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)
```

0.12.2 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 --no-verify

# Cite dataset in publication
iints data cite aide_t1d

# Check whether a trace still looks like believable T1D data
iints data realism-check data/my_trace.csv --output-json
↪ results/realism_report.json

# Compare against an empirical cohort envelope and export a dashboard
iints data realism-check data/my_trace.csv \
  --reference free_living_t1d \
  --output-html results/realism_dashboard.html

```

Available Datasets: - sample: Bundled full-day demo trace (no download needed) - ohio_t1dm: Classic request-gated benchmark with CGM, insulin, meals, and daily-life events - diatrend: Larger controlled-access CGM + pump dataset from 54 people with T1D - t1d_uom: 12-week multimodal dataset with CGM, insulin, meal macros, activity, and sleep - t1d_granada: Large longitudinal glucose-focused dataset from 736 people with T1D - azt1d: Arizona Type 1 Diabetes Dataset - hupa_ucm: HUPA-UCM free-living T1D dataset - aide_t1d: AIDE Type 1 Diabetes Dataset - pedap: Pediatric Artificial Pancreas dataset

Integrity and reproducibility - Public sources without a published pinned SHA-256 now require `--no-verify`. - Use that flag only when you trust the upstream source and understand that this is not cryptographic verification. - 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/...

```

0.12.3 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")
```

0.12.4 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")
```

0.12.5 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}")

```

0.12.6 Local AI Assistant (Minstral 3 Open-Weight via Ollama)

The SDK includes a research-only local AI assistant for simulation explanation and summary generation.

What it is for: - Explain a single decision step in plain language - Summarize glycemic trends from a run payload - Detect anomalies in a result summary - Generate a short markdown run report

What it is not for: - Clinical dosing advice - Live patient-facing recommendations - Medical decision support

Safety gate before any LLM call

Every AI command is blocked unless the MDMP artifact verifies successfully and meets the minimum required grade.

The AI flow is: 1. Load simulation JSON from disk 2. Verify signed MDMP artifact with MDMP-Guard 3. Check that Ollama is reachable 4. Resolve the requested local Ministral 3 model tag 5. Generate a response 6. Append a hard-coded research-only disclaimer

Recommended setup

Always use an active virtual environment:

```
python3 -m venv .venv
source .venv/bin/activate
python -m pip install -U pip
python -m pip install -e "[mdmp]"
```

Pull and validate the local model:

```
python -m pip install -U "iints-sdk-python35[full,mdmp]"
iints ai models
ollama pull ministral-3:8b
iints ai local-check --model ministral-3:8b
```

If you want to make the Ollama link explicit:

```
export OLLAMA_HOST=http://127.0.0.1:11434
iints ai local-check --model ministral-3:8b
```

local-check now includes a small generation smoke-test by default, so it verifies real model inference readiness and not just basic Ollama reachability.

Recommended run workflow

```
iints quickstart --project-name iints_quickstart
cd iints_quickstart
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py
iints ai prepare results/<run_id>
iints ai report results/<run_id>
iints ai review results/<run_id>
```

iints ai prepare creates AI-ready JSON payloads in results/<run_id>/ai/ and, when the MDMP extra is installed, also generates a local development MDMP certificate plus companion keypair so the assistant can run without hand-building step.json and report.signed.mdmp.

If local generation still disconnects, the first practical fallback is to switch from ministral-3:8b to ministral-3:3b.

Personal CareLink workflow

If you imported your own MiniMed / CareLink data, you can build a personal workspace that combines: - a standard IINTS timeline - a scenario for experiments - a visual dashboard - AI-ready payloads

```
iints carelink-workbench \  
  --input-csv "/path/to/CareLink export.csv" \  
  --output-dir results/personal_carelink
```

This creates: - carelink_timeline.csv - carelink_metrics.json - carelink_dashboard.png - carelink_poster.png - carelink_dashboard.html - ai/report_payload.json - ai/review_payload.json - ai/trends_payload.json - ai/anomalies_payload.json - ai/step_riskiest.json

If the MDMP extra is installed, the workbench also generates a local development certificate so the assistant can explain imported personal data without any manual JSON preparation.

Inference commands

Prepared run directory mode:

```
iints ai explain results/<run_id>  
iints ai trends results/<run_id>  
iints ai anomalies results/<run_id>  
iints ai report results/<run_id> \  
  --output results/<run_id>/ai/ai_report.md  
iints ai review results/<run_id>
```

Prepared personal CareLink workspace mode:

```
iints ai report results/personal_carelink --model minstral-3:3b  
iints ai review results/personal_carelink --model minstral-3:3b  
iints ai trends results/personal_carelink --model minstral-3:3b  
iints ai explain results/personal_carelink --model minstral-3:3b
```

Direct JSON mode:

```
iints ai explain results/step.json \
  --mdmp-cert results/report.signed.mdmp

iints ai trends results/glucose_payload.json \
  --mdmp-cert results/report.signed.mdmp

iints ai anomalies results/simulation_run.json \
  --mdmp-cert results/report.signed.mdmp

iints ai report results/simulation_run.json \
  --mdmp-cert results/report.signed.mdmp \
  --output results/ai_report.md

iints ai review results/simulation_run.json \
  --mdmp-cert results/report.signed.mdmp \
  --output results/realism_review.md
```

Operational notes - The default local model is `ministral-3:8b`. - Friendly aliases like `ministral` are resolved automatically against installed Ollama tags. - Older `Ministral 8B` tags remain accepted as backward-compatible fallbacks if they are already installed locally. - Users can select a different local model with `--model`, for example `ministral-3:3b` or `ministral-3:14b`.

PC specification guidance

Model	Best fit	Recommended system RAM	Recommended GPU VRAM	Approx download
ministral-3:3b	Smaller laptop / CPU-first setup	16 GB	6 GB	~3 GB
ministral-3:8b	Balanced desktop / strong laptop	24 GB	10 GB	~6 GB
ministral-3:14b	High-end workstation	32 GB	16 GB	~10 GB

Suggested rule: - start with `ministral-3:8b` - move down to `ministral-3:3b` if memory or latency is tight - move up to `ministral-3:14b` only if you have headroom and want better answer quality - Oversized JSON payloads are clipped automatically before prompt construction so local inference stays practical on smaller systems. - Use `--timeout-seconds 120` or higher for slower edge hardware.

Full source legend

If you want the full human-readable legend of the medical, dataset, runtime, and emulation references used across the SDK guides and defaults, see:

- docs/EVIDENCE_BASE.md

If you want the packaged machine-readable subset from the installed SDK:

```
iints sources --output-json results/source_manifest.json
```

0.13 Poster-Ready Results for Jury Demos

The SDK can now turn one to three completed run bundles into a single poster-style PNG. This is ideal for science fairs, thesis defenses, pitch decks, and jury presentations where you want three scenarios side by side.

Recommended story - Normal run - Meal stress test - Supervisor override / safety intervention

Command

```
iints poster \  
  --run-dir results/normal_run \  
  --run-dir results/meal_stress \  
  --run-dir results/supervisor_override \  
  --label "Normal Run" \  
  --label "Meal Stress Test" \  
  --label "Supervisor Override" \  
  --output-path results/posters/iints_results_poster.png
```

What it includes - Glucose curve for each run - Target range highlight (70–180 mg/dL) - Meal markers when carbs are present - Supervisor intervention markers when the safety layer triggered - Per-panel summary box with TIR, time below range, meal count, and intervention count

Outputs - results/posters/iints_results_poster.png - results/posters/iints_results_poster.json

If you omit `--run-dir`, the SDK auto-discovers the latest run bundles under `./results`.

0.14 Booth / Jury Demo Bundle

If you need a live demo for a science fair, jury room, or expo booth, the SDK now includes a one-command flow that builds the whole story for you.

Command

```
./scripts/run_booth_demo.sh
```

Or, if you want to stay inside the installed CLI:

```
iints demo-booth --output-dir results/booth_demo
```

What it creates - Normal Run - Meal Stress Test - Supervisor Override - a combined poster PNG - a markdown talk track for the jury - optional AI-ready artifacts for the safety scenario

Main outputs - examples/demos/07_live_stage_demo.py - results/-booth_demo/booth_demo_poster.png - results/booth_demo/JURY_TALK_TRACK.md - results/booth_demo/run_commands.md - results/booth_demo/demo_summary.json

Best live flow

For a booth or jury table, the cleanest explanation is:

1. Open examples/demos/07_live_stage_demo.py.
2. Point to PATIENT_CONFIG, OUTPUT_DIR, DURATION_MINUTES, TIME_STEP_MINUTES, and SEED.
3. Point out that the script visibly calls run_full(...), generate_results_poster(...), and prepare_ai_ready_artifacts(...).
4. Explain that swapping the patient profile reruns the same pipeline for another patient.
5. Run:

```
./scripts/run_live_stage_demo.sh
```

6. Open:

- results/booth_demo_live/booth_demo_poster.png
- results/booth_demo_live/JURY_TALK_TRACK.md
- results/booth_demo_live/BEURS_LIVE_DEMO_SCRIPT.txt

7. If the machine only has the installed SDK and not the repository checkout, export the same code first:

```
iints demo-export --output-dir iints_demo
cd iints_demo
python 07_live_stage_demo.py
```

8. If someone wants more proof, open a scenario folder and show results.csv, clinical_report.pdf, and run_manifest.json.

Why this is useful - You get real run bundles, not hand-built presentation images. - You can explain normal control, stress handling, and safety intervention in one pass. - The audience can see that the SDK is reproducible, visual, and safety-first.

Troubleshooting - If iints ai local-check fails, first confirm Ollama is running. - If Ollama is reachable but the model is missing, run the exact ollama pull ... command shown by the SDK. - If generation is slow, increase --timeout-seconds and use a smaller JSON payload.

0.15 Troubleshooting

0.15.1 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+
```

Issue: another machine still behaves like an old SDK after upgrading

Solution:

```
source .venv/bin/activate
python -m pip install -U pip
python -m pip install -U "iints-sdk-python35[full,mdmp]"
hash -r
python -c "import iints; print(iints.__version__)"
iints-sdk-doctor
```

If iints-sdk-doctor reports a conflict:

```
python -m pip uninstall -y iints iints-sdk-python35
python -m pip install -U "iints-sdk-python35[full,mdmp]"
hash -r
```

For the full upgrade walkthrough, see docs/UPDATING.md. By default, the upgrade commands in that guide track the latest release. Only pin an exact version when you need reproducible packaging for a demo, paper, or audit.

0.15.2 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
```

0.15.3 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
)
```

0.15.4 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"
)
```

0.15.5 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)
```

0.16 Quick Reference

0.16.1 Essential CLI Commands

```
# Confirm the environment works
iints doctor --smoke-run --suggest
iints doctor --full --suggest

# Fastest first run
iints demo
iints demo --dry-run
iints guide
iints run --wizard

# Create a reusable project
iints quickstart --project-name my_project
cd my_project
iints presets run --name baseline_t1d --algo algorithms/example_algorithm.py

# Generic run flow
iints run --preset baseline_t1d
iints run --algo algorithms/my_algo.py --scenario
↳ scenarios/example_scenario.json --dry-run

# Build a study-ready bundle
iints study-ready --algo algorithms/example_algorithm.py --output-dir
↳ results/study_ready

# Protocol-driven benchmark path
iints study-protocol --output-dir results/study_protocol
iints run-study --experiment results/study_protocol/study_experiment.yaml

# Data and AI
iints data list
iints data fetch aide_t1d --output-dir data_packs/public/aide_t1d --no-verify
iints ai local-check --model ministrat-3:8b
iints ai prepare results/<run_id>
iints ai report results/<run_id>

# Edge and booth operations
iints edge setup --output-dir iints_pi_demo --board raspberry_pi
iints makerfaire up --project-dir .
iints makerfaire autostart --project-dir .
iints makerfaire watchdog --project-dir .
```

0.16.2 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 skeleton

```

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 quickly

```

import pandas as pd

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

```

0.16.3 Safety Thresholds (Defaults)

```

SafetyConfig(
    hypo_cutoff=70.0,
    severe_hypo_cutoff=54.0,
    critical_hypo_cutoff=40.0,
    hyper_cutoff=300.0,
    max_basal_rate=2.0,
    max_bolus=5.0,
    max_iob=10.0,
    max_insulin_per_hour=15.0,
    max_insulin_per_day=80.0,
)

```

0.16.4 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

0.16.5 Scientific Study Workflow

Use this flow when you want a real experimental protocol instead of a one-off demo:

```
iints study-protocol --output-dir results/study_protocol
iints run-study \
  --algo algorithms/example_algorithm.py \
  --output-dir results/study_bundle
iints analyze \
  results/study_bundle \
  --output-json results/study_summary.json \
  --output-markdown results/study_summary.md
iints compare-study \
  results/study_clean \
  results/study_corrupted \
  --output-json results/study_comparison.json
iints poster-study results/study_summary.json --output-path
  ↪ results/study_poster.png
```

This gives you: - a written protocol with hypotheses and a study matrix - baseline comparisons including the built-in Clinical Baseline - descriptive statistics and confidence intervals - safety summaries, failure analysis, and subgroup views - poster-ready outputs for reports or jury sessions

0.17 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

0.18 Need More Help?

Best next documents - Getting Started: [docs site](#) - Quickstart: [docs site](#) - Command reference: [docs site](#) - Scientific workflow: [docs site](#) - Maker Faire Pi guide: [docs site](#)

Project support routes - GitHub repository: <https://github.com/python35/IINTS-SDK> - Issue tracker: <https://github.com/python35/IINTS-SDK/issues> - Research report: [research/EUCYS_REPORT.md](#)

If you are stuck, the fastest recovery commands are usually:

```
iints doctor --full --suggest
iints demo --dry-run
iints run --wizard
```