AsHDT

Astronaut Human Digital Twin

PoC Implementation Guide — v0.2

Kevin Dominey
Space Applications & Technologies Laboratory · Institute of Space Science — INFLPR
February 2026

How to Use This Document

This document is your implementation reference for the AsHDT proof-of-concept. It does not contain runnable code — instead it specifies, precisely and in order, every file you need to create, what it should contain, and why. Follow the steps in sequence. Each step ends in a concrete, testable result so you always know whether you are ready to move forward.

The build is organized into five phases:

- **0** Phase 0 — Project scaffolding: Git, folder structure, conda environment, Node check
- **1** Phase 1 — Synthetic test data: JSON files representing a single subject's biomarker readings over time
- **2** Phase 2 — Backend core: FastAPI server, SQLite database, trajectory computation engine
- **3** Phase 3 — Frontend shell: Svelte project, API connection, Plotly.js trajectory chart
- **4** Phase 4 — Integration: wiring frontend requests to backend analysis, end-to-end test

> *Decisions still open at the time of writing are marked with ⚠. These will not block progress but will need to be resolved before production use.*

Phase 0 — Project Scaffolding

0.1 Git Repository

Create a new folder called asHDT. Inside it, run:

```
git init
```

Create a .gitignore file immediately with the following contents before making your first commit:

```
# Python
__pycache__/
*.pyc
.env
.ipynb_checkpoints

# Node
node_modules/
dist/

# Database
*.db

# VS Code
.vscode/
```

Make your first commit after creating the .gitignore:

```
git add .gitignore
git commit -m "Initial commit"
```

0.2 Folder Structure

Create the following folder and file structure manually inside asHDT/. Empty folders need a placeholder file (e.g. .gitkeep) to be tracked by Git.

Path	Purpose
asHDT/	Project root
backend/	All Python source code and FastAPI server
backend/core/	Core modules: ingestion, state store, analysis, output
backend/core/ingestion/	Schema validator, format router, module registry loader
backend/core/state_store/	Archive reader/writer, snapshot manager, SQLite interface
backend/core/analysis/	Trajectory computer, multimodal integrator (stub for now)
backend/core/output/	Report serializer, snapshot serializer
backend/api/	FastAPI route definitions

backend/main.py	FastAPI app entry point
backend/requirements.txt	Python dependencies
registry/	Module registry config files
registry/module_registry.json	Single source of truth for all registered modules and markers
data/	All data files (not committed to Git)
data/archive/	Raw data point JSON files, organized by subject/module/marker
data/snapshots/	Generated snapshot JSON files
data/reports/	Generated time-graph report JSON files
data/asHDT.db	SQLite database file (auto-created on first run)
frontend/	Svelte application (initialized in Phase 3)
notebooks/	Jupyter notebooks for developing and validating backend logic before transcribing to .py files
test_data/	Synthetic JSON files used for Phase 1 testing

> ⚠ Add data/ to your .gitignore after creating the folder. You do not want to commit patient data or large archives to version control.

0.3 Python Environment

Create and activate a conda environment for the project. The environment is named asHDT and uses Python 3.11:

```
conda create -n asHDT python=3.11
conda activate asHDT
```

The conda environment lives outside the project folder in your conda installation — there is no venv/ folder inside the repository.

Create backend/requirements.txt with the following initial dependencies:

```
fastapi>=0.110.0
uvicorn>=0.29.0
numpy>=1.26.0
scipy>=1.12.0
python-dateutil>=2.9.0
pydantic>=2.6.0
jupyter>=1.0.0
```

From inside the backend/ folder, install them into the active conda environment:

```
pip install -r requirements.txt
```

> *scipy is included now because numpy.polyfit covers polynomial fitting but scipy will be needed later for more advanced smoothing options. jupyter is included so that notebooks/ can be used immediately for backend logic development.*

0.4 Jupyter Notebooks

The notebooks/ folder is used to develop and validate backend logic before transcribing it to .py files. This is the recommended workflow for all analysis code in backend/core/ — write and test in a notebook first, then move the validated logic into the corresponding Python module.

The two notebooks to create at the start are:

- notebooks/archive_reader.ipynb — for developing and testing backend/core/state_store/archive_reader.py
- notebooks/trajectory_computer.ipynb — for developing and testing backend/core/analysis/trajectory.py

To start Jupyter from the project root with the asHDT environment active:

```
conda activate asHDT
jupyter notebook
```

> ⚠ FastAPI (main.py, routes.py) and the Svelte frontend are never developed in notebooks. These must be run from the terminal as persistent processes.

0.5 Node / Svelte Environment

The Svelte project will be initialized in Phase 3. For now, confirm Node.js is working:

```
node --version    # should be >= 18
npm --version
```

No action needed here yet.

0.6 Verification Checkpoint

Before moving to Phase 1, confirm the following:

- asHDT/ folder exists with the full structure above
- Git repository initialized, .gitignore in place, first commit made
- conda environment asHDT active, all requirements installed (pip list to verify)
- Jupyter notebook server starts successfully from project root
- data/ is in .gitignore

Phase 1 — Synthetic Test Data

1.1 Purpose

Before any backend code is written, we need data files that the system will read. This allows the archive reader and trajectory computer to be built and tested immediately, without waiting for real VTF data. The synthetic data should be physiologically plausible but does not need to be perfectly realistic — it just needs to cover enough variation to exercise all 27 trajectory states.

We will create data for one subject, one module, one marker. This is the minimum needed to produce a trajectory visualization.

1.2 Subject and Marker Choice

Field	Value
subject_id	`subject_001`
module_id	`vtf_stress_test`
marker_id	`vo2max`
unit	`ml/kg/min`
volatility_class	`short_term`
healthy_min (test)	`45.0`
healthy_max (test)	`60.0`
vulnerability_margin_pct (test)	`10.0 → margin = 1.5 units each side`

1.3 File Location

Each data point is a separate JSON file. The folder path encodes the subject, module, and marker identity:

```
data/archive/subject_001/vtf_stress_test/vo2max/{timestamp}.json
```

The timestamp in the filename should be ISO 8601 format with colons replaced by hyphens for Windows filesystem compatibility:

```
2026-01-05T08-00-00Z.json
```

1.4 Data Point JSON Schema

Every JSON file in the archive must conform to this structure exactly. The ingestion layer will validate against this schema when it is built in Phase 2.

Field	Type	Description

schema_version	string	Always "1.0" for PoC. Enables future migration.
module_id	string	Must match a registered module in module_registry.json
marker_id	string	Must match a marker declared by the module above
subject_id	string	Identifier for the subject this reading belongs to
timestamp	string	ISO 8601 UTC datetime: "2026-01-05T08:00:00Z"
value	number	The measured value in the declared unit
unit	string	Must match unit declared in module_registry.json
data_quality	enum	One of: "good", "degraded", "bad"

Example of a correctly formatted data point file:

```
{
  "schema_version": "1.0",
  "module_id": "vtf_stress_test",
  "marker_id": "vo2max",
  "subject_id": "subject_001",
  "timestamp": "2026-01-05T08:00:00Z",
  "value": 52.3,
  "unit": "ml/kg/min",
  "data_quality": "good"
}
```

1.5 Synthetic Data Points to Create

Create the following 12 data point files. The values are designed to produce a trajectory that starts in non-pathology, passes through vulnerability, enters pathology, then partially recovers — exercising a wide range of trajectory states and making the visualization meaningful from the first run.

Filename (in vo2max/)	value	data_quality	Expected zone
2026-01-05T08-00-00Z.json	55.1	good	Non-pathology
2026-01-12T08-00-00Z.json	53.8	good	Non-pathology
2026-01-19T08-00-00Z.json	51.2	good	Non-pathology
2026-01-26T08-00-00Z.json	49.0	good	Non-pathology
2026-02-02T08-00-00Z.json	47.2	good	Vulnerability

2026-02-09T08-00-00Z.json	46.1	good	Vulnerability
2026-02-16T08-00-00Z.json	44.3	degraded	Pathology
2026-02-23T08-00-00Z.json	42.8	degraded	Pathology
2026-03-02T08-00-00Z.json	41.5	bad	Pathology
2026-03-09T08-00-00Z.json	43.1	good	Pathology
2026-03-16T08-00-00Z.json	45.8	good	Vulnerability
2026-03-23T08-00-00Z.json	48.4	good	Non-pathology

1.6 Index File

Alongside the data point files, create an index file at:

```
data/archive/subject_001/vtf_stress_test/vo2max/index.json
```

This file contains a time-ordered list of all data point filenames for this marker. The archive reader uses this to perform fast time-range queries without scanning all files. You must update this file manually whenever you add a new data point in the PoC. In a later version, the ingestion layer will maintain it automatically.

```
{
  "subject_id": "subject_001",
  "module_id": "vtf_stress_test",
  "marker_id": "vo2max",
  "unit": "ml/kg/min",
  "entries": [
    { "timestamp": "2026-01-05T08:00:00Z", "file": "2026-01-05T08-00-00Z.json" },
    { "timestamp": "2026-01-12T08:00:00Z", "file": "2026-01-12T08-00-00Z.json" },
    { "timestamp": "2026-01-19T08:00:00Z", "file": "2026-01-19T08-00-00Z.json" },
    { "timestamp": "2026-01-26T08:00:00Z", "file": "2026-01-26T08-00-00Z.json" },
    { "timestamp": "2026-02-02T08:00:00Z", "file": "2026-02-02T08-00-00Z.json" },
    { "timestamp": "2026-02-09T08:00:00Z", "file": "2026-02-09T08-00-00Z.json" },
    { "timestamp": "2026-02-16T08:00:00Z", "file": "2026-02-16T08-00-00Z.json" },
    { "timestamp": "2026-02-23T08:00:00Z", "file": "2026-02-23T08-00-00Z.json" },
```

```
    { "timestamp": "2026-03-02T08:00:00Z", "file": "2026-03-02T08-00-
00Z.json" },
    { "timestamp": "2026-03-09T08:00:00Z", "file": "2026-03-09T08-00-
00Z.json" },
    { "timestamp": "2026-03-16T08:00:00Z", "file": "2026-03-16T08-00-
00Z.json" },
    { "timestamp": "2026-03-23T08:00:00Z", "file": "2026-03-23T08-00-
00Z.json" }
  ]
}
```

1.7 Verification Checkpoint

Before moving to Phase 2, confirm:

- 12 individual data point JSON files exist in data/archive/subject_001/vtf_stress_test/vo2max/
- index.json exists in the same folder with 12 entries in chronological order
- Each JSON file passes a manual read — open it and confirm the structure matches the schema in 1.4

Phase 2 — Backend Core

2.1 Overview

The backend is a FastAPI application. It has four internal modules (ingestion, state_store, analysis, output) and one API layer that exposes endpoints to the frontend. For the PoC thin slice, the build order within the backend is:

- **2.2** Module Registry loader — reads module_registry.json into memory at startup
- **2.3** SQLite database setup — creates the database schema on first run
- **2.4** Archive Reader — reads JSON files from the data/archive/ hierarchy
- **2.5** Trajectory Computer — the analytical core, fully self-contained
- **2.6** FastAPI app and routes — exposes the above as HTTP endpoints

> *Sections 2.4 (Archive Reader) and 2.5 (Trajectory Computer) should be developed in Jupyter notebooks first (notebooks/archive_reader.ipynb and notebooks/trajectory_computer.ipynb respectively), then transcribed into their .py files once the logic is validated. The notebooks are the scratchpad; the .py files are the source of truth.*

2.2 Module Registry

File: registry/module_registry.json

This file is the single source of truth for all registered modules. Create it now with the one module and one marker needed for Phase 1 testing. Additional modules and markers will be added as entries in the same file later.

```
{
  "registry_version": "1.0",
  "modules": [
    {
      "module_id": "vtf_stress_test",
      "description": "ESA Vertical Treadmill Facility microgravity stress
test",
      "format": "json",
      "markers": [
        {
          "marker_id": "vo2max",
          "description": "Maximal oxygen uptake",
          "unit": "ml/kg/min",
          "volatility_class": "short_term",
          "schema_version": "1.0"
        }
      ]
    }
  ]
}
```

File: backend/core/ingestion/registry_loader.py

Write a Python module with a single function load_registry() that reads module_registry.json from the registry/ folder and returns its contents as a Python dict. This function should be called once at FastAPI startup and the result stored in application state. It should raise a clear RuntimeError if the file is missing or malformed.

The function signature should be:

```
def load_registry(registry_path: str) -> dict
```

2.3 SQLite Database Setup

File: backend/core/state_store/database.py

This module is responsible for initializing the SQLite database and providing a connection factory. The database file lives at data/asHDT.db. On first run, if the file does not exist, it should be created automatically with the schema below.

The database has three tables for the PoC:

Table	Purpose	Columns
snapshots	Stores metadata for every generated snapshot	snapshot_id (TEXT PK), subject_id (TEXT), requested_at (TEXT), target_time (TEXT), marker_ids (TEXT — JSON array)
timegraph_reports	Stores metadata for every generated time-graph report	report_id (TEXT PK), subject_id (TEXT), marker_id (TEXT), module_id (TEXT), requested_at (TEXT), timeframe_from (TEXT), timeframe_to (TEXT), polynomial_degree (INTEGER), healthy_min (REAL), healthy_max (REAL), vulnerability_margin_pct (REAL)
subjects	Simple registry of known subjects	subject_id (TEXT PK), created_at (TEXT), notes (TEXT)

The full report and snapshot JSON payloads are stored as files in data/reports/ and data/snapshots/ respectively. The database tables store only the metadata needed for listing, searching, and linking to those files. This keeps the database lightweight and the full data human-readable.

Write a function init_db(db_path: str) that creates the database and tables if they do not already exist, using CREATE TABLE IF NOT EXISTS. Write a second function get_connection(db_path: str) that returns a sqlite3.Connection with row_factory set to sqlite3.Row so that results can be accessed by column name.

2.4 Archive Reader

File: backend/core/state_store/archive_reader.py

This module reads data point files from the archive folder hierarchy. It exposes one primary function:

```
def read_timeseries(
    archive_root: str,
    subject_id: str,
    module_id: str,
    marker_id: str,
    from_time: datetime,
    to_time: datetime
) -> list[dict]
```

The function should:

- Construct the folder path from archive_root / subject_id / module_id / marker_id
- Read the index.json file from that folder
- Filter the index entries to those whose timestamp falls within [from_time, to_time] inclusive
- Read each matching data point JSON file
- Return a list of dicts, sorted chronologically by timestamp, each containing all fields from the data point file plus a parsed_timestamp field as a Python datetime object

The function should raise a FileNotFoundError with a descriptive message if the index file does not exist, and silently skip any individual data point file that cannot be read (logging a warning instead of crashing).

> *For the PoC, timestamps in filenames use hyphens in place of colons. The index.json stores the canonical ISO timestamp with colons. Use the index file for filtering logic, not the filenames.*

2.5 Trajectory Computer

File: backend/core/analysis/trajectory.py

This is the analytical heart of the PoC. It takes a time-series and zone boundary parameters and returns a fully annotated trajectory. This module has no file I/O — it is pure computation and should be independently testable.

Inputs

Parameter	Type	Description
`datapoints`	list[dict]	Output of archive_reader.read_timeseries(). Each dict has parsed_timestamp and value at minimum.
`healthy_min`	float	Lower bound of the healthy reference range for this subject and marker
`healthy_max`	float	Upper bound of the healthy reference range for this subject and marker

vulnerability_margin_pct	float	Percentage of the healthy range width defining the vulnerability band on each side
polynomial_degree	int	Degree of the polynomial to fit. 1 = linear, 2 = quadratic, etc.

Processing Steps

The function should perform the following steps in order:

- Convert each parsed_timestamp to a numeric x-value in hours elapsed since the earliest timestamp in the series. This makes the polynomial fit numerically stable and the derivative physically interpretable (units: marker_unit per hour).
- Fit a single polynomial of the requested degree to the (x, value) pairs using numpy.polyfit. Store the resulting coefficient array.
- Compute the first derivative polynomial coefficients using numpy.polyder on the fitted polynomial. Compute the second derivative polynomial coefficients using numpy.polyder again on the first derivative.
- For each data point, evaluate f(x), f'(x), and f''(x) using numpy.polyval. Note: f(x) from the polynomial is the smoothed fitted value, not the raw measured value. Store both.
- Compute zone boundaries: vulnerability_lower = healthy_min + (healthy_max - healthy_min) * vulnerability_margin_pct / 100. vulnerability_upper = healthy_max - (healthy_max - healthy_min) * vulnerability_margin_pct / 100. Note: vulnerability is the band between healthy_min and vulnerability_lower, and between vulnerability_upper and healthy_max. Pathology is below healthy_min or above healthy_max.
- Assign zone for each point based on the raw measured value (not the fitted value). Zone is one of: "non_pathology", "vulnerability", "pathology".
- Assign trajectory_state integer (1–27) from the lookup table in your trajectories document, based on the combination of zone, sign of f'(x), and sign of f''(x). Use a threshold of ±0.001 per hour for treating a derivative as zero.
- Compute time_to_transition_hours for each point: using the fitted polynomial, find the next time (after the current point's x value) at which the fitted value crosses either healthy_min, vulnerability_lower, vulnerability_upper, or healthy_max. This is a root-finding problem — use numpy.roots on the polynomial minus the boundary value, filter for real positive roots greater than the current x, and take the smallest. If no such root exists within a reasonable horizon (e.g. 10 × the total timespan), return null.

Output

The function returns a dict with two keys:

```
{
  "fit_metadata": {
    "polynomial_degree": 2,
    "coefficients": [...],
    "x_origin_timestamp": "2026-01-05T08:00:00Z",
    "healthy_min": 45.0,
    "healthy_max": 60.0,
    "vulnerability_margin_pct": 10.0,
```

```
    "vulnerability_lower": 46.5,
    "vulnerability_upper": 58.5
  },
  "trajectory": [
    {
      "timestamp": "2026-01-05T08:00:00Z",
      "x_hours": 0.0,
      "raw_value": 55.1,
      "fitted_value": 54.8,
      "zone": "non_pathology",
      "f_prime": -0.18,
      "f_double_prime": 0.002,
      "trajectory_state": 9,
      "time_to_transition_hours": 312.5
    },
    ...
  ]
}
```

> ⚠ For polynomial degrees >= 3 on sparse data (< 10 points), numpy.polyfit may overfit and produce oscillating derivatives. For the PoC with 12 points, degree 2 is recommended as the default. Expose degree as a user parameter but document this risk.

2.6 FastAPI Application

File: backend/main.py

The FastAPI app entry point. It should:

- Import and call init_db() on startup to ensure the database exists
- Import and call load_registry() on startup and store the result in app.state.registry
- Include the router from backend/api/routes.py
- Configure CORS to allow requests from http://localhost:5173 (the default Svelte dev server port)

To run the server:

```
cd backend
uvicorn main:app --reload --port 8000
```

File: backend/api/routes.py

Define the following three endpoints for the PoC thin slice. All endpoints return JSON.

Method	Path	Description
GET	`/registry`	Returns the full module registry. Used by the frontend to populate module and marker dropdowns.

GET	/subjects	Returns a list of all subject_ids found in data/archive/. Scans the archive folder rather than the database so it works even before any snapshots are generated.
POST	/timegraph	Accepts a JSON request body with all time-graph parameters (see below). Calls archive_reader and trajectory_computer. Saves the result to data/reports/ and records metadata in the timegraph_reports database table. Returns the full trajectory result.

POST /timegraph — Request Body

```
{
  "subject_id": "subject_001",
  "module_id": "vtf_stress_test",
  "marker_id": "vo2max",
  "timeframe": {
    "from": "2026-01-01T00:00:00Z",
    "to": "2026-03-31T00:00:00Z"
  },
  "zone_boundaries": {
    "healthy_min": 45.0,
    "healthy_max": 60.0,
    "vulnerability_margin_pct": 10.0
  },
  "fitting": {
    "polynomial_degree": 2
  }
}
```

Use Pydantic models to define the request body structure. FastAPI validates the incoming JSON automatically against the model — this is your schema validation layer for the PoC.

2.7 Verification Checkpoint

Before moving to Phase 3, verify the backend is working by calling the API directly:

- Start the server with uvicorn main:app --reload --port 8000
- Navigate to http://localhost:8000/docs in a browser — FastAPI auto-generates an interactive API explorer
- Call GET /registry and confirm the module registry is returned
- Call GET /subjects and confirm subject_001 appears
- Call POST /timegraph with the example request body above and confirm a trajectory array is returned with 12 points, each containing zone, trajectory_state, f_prime, f_double_prime, and time_to_transition_hours
- Confirm a report JSON file was created in data/reports/
- Confirm a row was inserted in the timegraph_reports database table (open asHDT.db with DB Browser for SQLite or any SQLite viewer)

Phase 3 — Frontend Shell

3.1 Initialize the Svelte Project

From the asHDT/ root folder, run:

```
npm create vite@latest frontend -- --template svelte
cd frontend
npm install
```

Then install Plotly.js:

```
npm install plotly.js-dist-min
```

Confirm it starts:

```
npm run dev
```

The default Vite dev server runs on http://localhost:5173. This is the URL you already configured in CORS on the backend.

3.2 Project Structure Inside frontend/src/

File	Purpose
App.svelte	Root component. Holds top-level layout and routes between views.
lib/api.js	All fetch calls to the FastAPI backend. No fetch calls should appear outside this file.
lib/stores.js	Svelte writable stores for shared state: registry, selected subject, current report.
components/TimeGraphForm.svelte	Form for user to input all time-graph request parameters.
components/TrajectoryChart.svelte	Plotly.js chart rendering the trajectory output.
components/TrajectoryTable.svelte	Tabular view of the trajectory output, one row per data point.

3.3 lib/api.js

Define the following functions. All functions are async and return parsed JSON or throw an Error with a descriptive message on failure:

```
const BASE = 'http://localhost:8000'

export async function getRegistry()
  // GET /registry

export async function getSubjects()
  // GET /subjects
```

```
export async function postTimegraph(params)
  // POST /timegraph
  // params matches the request body schema defined in 2.6
```

3.4 components/TimeGraphForm.svelte

This form collects all inputs needed for a time-graph request. It should contain the following fields:

- Subject selector — dropdown populated from getSubjects() on mount
- Module selector — dropdown populated from getRegistry() on mount, shows module_id values
- Marker selector — dropdown filtered by the selected module, shows marker_id values
- From date — date input
- To date — date input
- Healthy min — number input
- Healthy max — number input
- Vulnerability margin % — number input, default 10
- Polynomial degree — number input, default 2, min 1, max 5
- Generate button — calls postTimegraph() with the assembled params and emits the result upward via a Svelte dispatch event

On submit, the form should show a loading indicator while the request is in flight and display a plain error message if the request fails.

3.5 components/TrajectoryChart.svelte

This component receives the trajectory result from the API and renders it using Plotly.js. It accepts one prop: report (the full response from POST /timegraph).

The chart should display the following traces:

Trace	Type	Description
Raw values	Scatter markers	One marker per data point. Color-coded by zone: green for non_pathology, amber for vulnerability, red for pathology. Size should reflect data_quality: good = large, degraded = medium, bad = small.
Fitted curve	Line	The polynomial fit evaluated densely (100+ points) across the timeframe. Use the coefficients from fit_metadata and x_origin_timestamp to reconstruct this in the frontend. Color: blue, dashed.
Zone bands	Filled regions	Horizontal colored bands: green fill between vulnerability_lower and vulnerability_upper (non-pathology

		core), amber fill from healthy_min to vulnerability_lower and vulnerability_upper to healthy_max (vulnerability bands), red fill below healthy_min and above healthy_max (pathology). Use Plotly layout shapes with opacity 0.12.
f' trace	Line (secondary y-axis)	First derivative values at each data point timestamp. Displayed on a secondary y-axis on the right side. Color: purple.

Hover tooltips on each raw value marker should display: timestamp, raw_value, fitted_value, zone, trajectory_state (integer and label from your 27-state table), f_prime, f_double_prime, and time_to_transition_hours.

> *To render the fitted curve densely in the frontend, you need to evaluate the polynomial client-side. Plotly does not do this for you. Implement a small evaluatePolynomial(coefficients, x) helper function that uses Horner's method. The x values should be computed as hours elapsed from x_origin_timestamp across the requested timeframe.*

3.6 components/TrajectoryTable.svelte

A simple HTML table showing all 12 data points with columns: Timestamp, Raw Value, Fitted Value, Zone, State Index, State Label, f', f'', Time to Transition. Rows should be color-coded by zone using the same color scheme as the chart. This table is useful for precise inspection of values that are hard to read from the chart.

3.7 Verification Checkpoint

Before moving to Phase 4, verify the frontend independently:

- npm run dev starts without errors
- The form renders with all fields and dropdowns populated from the API
- Submitting the form with valid parameters returns data (check browser network tab)
- The chart renders with colored zone bands, raw value markers, and the fitted curve
- Hovering a marker shows the full tooltip
- The table renders with color-coded rows

Phase 4 — Integration and End-to-End Test

4.1 Running the Full Stack

Open two terminal windows in VS Code:

Terminal 1 — Backend:

```
conda activate asHDT
cd backend
uvicorn main:app --reload --port 8000
```

Terminal 2 — Frontend:

```
cd frontend
npm run dev
```

Navigate to http://localhost:5173 in a browser.

4.2 End-to-End Test Procedure

Execute the following test sequence manually and confirm each step:

- Open the dashboard. Confirm the subject dropdown shows subject_001 and the module dropdown shows vtf_stress_test.
- Select vo2max from the marker dropdown.
- Set timeframe from 2026-01-01 to 2026-03-31.
- Set healthy_min = 45, healthy_max = 60, vulnerability_margin_pct = 10.
- Set polynomial_degree = 2.
- Click Generate. Confirm the loading indicator appears and then disappears.
- Confirm the chart renders with three colored zone bands, 12 scatter markers, and a smooth fitted curve.
- Confirm at least one marker is green, at least one amber, and at least one red.
- Hover over the marker at 2026-03-02 (the bad-quality point at value 41.5). Confirm the tooltip shows zone = pathology, data_quality = bad, and a trajectory_state in the 19–27 range.
- Confirm the trajectory table below the chart shows all 12 rows with correct zone color coding.
- Check data/reports/ — confirm a new JSON file was created.
- Open the JSON file and confirm it contains the full trajectory array matching what is displayed in the UI.

4.3 Known Limitations of the PoC

The following are intentional omissions for the first build, noted here so they are not forgotten:

- No ingestion layer — JSON files are placed manually. Schema validation happens at the API request level (Pydantic) but not at data point ingest time.
- No snapshot generation — the snapshot pathway is architected but not yet implemented. This is the next feature to add after the trajectory visualization is validated.

- No alert system — alerts will be added after manual validation of the trajectory logic.
- No authentication — the API is open. Do not expose this server outside localhost.
- Index file is manually maintained — the archive reader depends on index.json being accurate. An automated index maintainer will be added in a later phase.
- Single subject, single marker — the architecture supports multiple subjects and markers but the UI only exposes one at a time in this version.

4.4 Next Steps After PoC Validation

Once the trajectory visualization is validated and you are satisfied with the trajectory logic, the recommended build order for subsequent phases is:

- Snapshot generation — implement the POST /snapshot endpoint and the SnapshotForm + SnapshotReport frontend components
- Ingestion layer — add schema validation and automatic index maintenance so new JSON files can be dropped into the archive without manual editing
- Second module and marker — add a second module to the registry and confirm the architecture handles it without modification
- Composite marker integration — implement the multimodal integrator stub in analysis/ as a real PCA or weighted combination
- Alert system — add threshold-crossing alerts triggered by trajectory state changes
- Continuous data formats — add Parquet and HDF5 support to the format router in the ingestion layer

Appendix A — Trajectory State Reference

This table is the complete reference for the 27 trajectory states defined in the Function-Dysfunction Trajectories framework. The trajectory computer maps every data point to one of these states based on zone (f(x)), sign of f'(x), and sign of f''(x).

Conventions: + = positive / non-pathology / improving / accelerating. 0 = neutral / vulnerability / stable / steady. − = negative / pathology / worsening / decelerating.

Index	f(x) Zone	f'(x)	f''(x)	State Label
1	+	+	+	Accelerating Improving Non-pathology
2	+	+	0	Steadily Improving Non-pathology
3	+	+	−	Decelerating Improving Non-pathology
4	+	0	+	Improving Reversal in Non-Pathology
5	+	0	0	Neutral Stable Non-pathology
6	+	0	−	Worsening Reversal in Non-Pathology
7	+	−	+	Decelerating Worsening Non-pathology
8	+	−	0	Steadily Worsening Non-pathology
9	+	−	−	Accelerating Worsening Non-pathology
10	0	+	+	Accelerating Improving Vulnerability
11	0	+	0	Steadily Improving Vulnerability
12	0	+	−	Decelerating Improving Vulnerability
13	0	0	+	Improving Reversal in Vulnerability
14	0	0	0	Neutral Stable Vulnerability
15	0	0	−	Worsening Reversal in Vulnerability
16	0	−	+	Decelerating Worsening Vulnerability
17	0	−	0	Steadily Worsening Vulnerability
18	0	−	−	Progressively Worsening Vulnerability
19	−	+	+	Accelerating Improving Pathology
20	−	+	0	Steadily Improving Pathology
21	−	+	−	Decelerating Improving Pathology
22	−	0	+	Improving Reversal in Pathology
23	−	0	0	Neutral Stable Pathology

24	−	0	−	Worsening Reversal in Pathology
25	−	−	+	Decelerating Worsening Pathology
26	−	−	0	Steadily Worsening Pathology
27	−	−	−	Accelerating Worsening Pathology

Appendix B — Confidence Score Reference

Confidence scores are computed at the moment of snapshot or report generation, not at data ingestion. The formula uses exponential decay parameterized by a half-life specific to each volatility class.

```
confidence = 0.5 ^ (age_hours / half_life_hours)
```

Volatility Class	Half-life	Half-life (hours)	Example markers
acute	6 hours	6	Cortisol, HRV during test, acute HR
short_term	7 days	168	VO_2max, muscle force output, gait metrics
medium_term	30 days	720	Bone density markers, body composition
stable	365 days	8760	Genetic markers, baseline anthropometrics