Opportunistic Nework Data Analyzer

Release 1.0.0

DHMAI Network Research Group

Jan 30, 2026

GETTING STARTED

CHAPTER

ONE

ARCHITECTURE AND DESIGN

This guide explains architecture and data flow of Opportunistic Network Data Analyzer (OppNDA).

1.1 Architecture Overview

OppNDA follows a modular architecture with clear separation of concerns:

```
+-------------------------------------------------------------+
|                        GUI Layer                            |
|  +--------------+  +--------------+  +-------------------+  |
|  | settings.html|  |   nda.html   |  |     config.js     |  |
|  |  (Config UI) |  |  (Results)   |  | (Frontend Logic)  |  |
|  +------+-------+  +------+-------+  +---------+---------+  |
+---------|-----------------|------------------------|----------+
          |                 |                        |
          v                 v                        v
+-------------------------------------------------------------+
|                    Flask Application                        |
|  +-----------------------------------------------------+  |
|  |                    app/api.py                       |  |
|  |  * Configuration endpoints                          |  |
|  |  * Execution endpoints (run-one, run-averager, etc) |  |
|  |  * Streaming endpoints (SSE for real-time output)   |  |
|  +--------------------------+--------------------------+  |
+-----------------------------|-------------------------------+
                              |
                              v
+-------------------------------------------------------------+
|                      Core Modules                           |
|  +----------------+  +----------------+  +---------------+  |
|  |   averager.py  |  |  analysis.py   |  | regression.py |  |
|  | (Data Avg)     |  | (Visualization)|  | (ML Models)   |  |
|  +-------+--------+  +-------+--------+  +------+--------+  |
|          +-------------------+-------------------+          |
|                              |                              |
|  +-----------------------------------------------------+  |
|  |              resource_manager.py                    |  |
|  |   Dynamic memory/CPU management for multiprocessing |  |
```

(continues on next page)

(continued from previous page)

```
|  +-----------------------------------------------------+  |
+-------------------------------------------------------------+
                               |
                               v
+-------------------------------------------------------------+
|                    Configuration                            |
|  +----------------+  +----------------+  +---------------+  |
|  |averager_config |  |analysis_config |  |regression_cfg |  |
|  |     .json      |  |     .json      |  |    .json      |  |
|  +----------------+  +----------------+  +---------------+  |
+-------------------------------------------------------------+
```

1.2 Data Flow

The typical data flow through OppNDA:

1. **ONE Simulator** generates report files (`*.txt`) in `reports/`
2. **Averager** groups and averages reports by configuration parameters
3. **Analysis** generates visualizations from averaged data
4. **Regression** (optional) trains ML models on the data

```
ONE Simulator
     |
     v (report files)
+-------------+     +-------------+     +-------------+
|  Averager   |---->|  Analysis   |---->|  Regression |
|             |     |             |     |   (ML)      |
+-------------+     +-------------+     +-------------+
     |                   |                   |
     v                   v                   v
averaged_*.txt       plots/*.png       regression_results/
```

CHAPTER

TWO

CORE MODULES

This section documents the core processing modules of OppNDA.

2.1 averager

Report file averaging and aggregation.

2.1.1 Report Averager Module

A flexible, high-performance tool for averaging multiple simulation report files. Optimized with multiprocessing for enhanced performance on multi-core systems.

This module processes report files generated by the ONE (Opportunistic Network Environment) simulator and groups them based on configurable parameters, calculating averages across multiple simulation runs.

Example

Basic usage with default config:

```
from core.averager import ReportAverager

averager = ReportAverager('config/averager_config.json')
averager.run()
```

Command-line usage:

```
python core/averager.py
```

`core.averager.SCRIPT_DIR`

Directory containing this script.

Type
Path

`core.averager.PROJECT_ROOT`

Root directory of the project.

Type
Path

core.averager.CONFIG_DIR

Directory containing configuration files.

Type
Path

core.averager.RESOURCE_MANAGER_AVAILABLE

Whether resource manager is available.

Type
bool

Note

This module uses multiprocessing for parallel file reading. The number of workers is dynamically calculated based on available RAM and CPU cores when the ResourceManager is available.

core.averager.read_and_parse_file_parallel(*args*)

Worker function for parallel file reading and parsing

core.averager.average_group_data(*aggregated*)

Calculate averages from aggregated data.

Takes a dictionary of field names to lists of values and calculates the mean for each field, properly handling NaN values.

Parameters
aggregated (*dict*) – Dictionary mapping field names to lists of values. Example: {'delivery_prob': [0.9, 0.92, 0.88], 'latency': [100, 110, 105]}

Returns

Dictionary mapping field names to their averaged values.
Fields with no valid values are excluded from the result.

Return type
dict

Example

```
>>> data = {'metric1': [1.0, 2.0, 3.0], 'metric2': [np.nan, np.nan]}
>>> avg = average_group_data(data)
>>> print(avg)  # {'metric1': 2.0, 'metric2': nan}
```

class core.averager.ReportAverager(*config_path*, *safety_enabled=True*)

Bases: object

Report file averaging and aggregation engine.

This class groups simulation report files based on configurable parameters and calculates averages across multiple runs. It uses multiprocessing for parallel file reading to maximize performance.

The averaging process:

1. Scans the reports directory for matching files
2. Parses filenames to extract component values (router, TTL, buffer, etc.)
3. Groups files based on configured grouping parameters
4. Reads files in parallel using multiprocessing
5. Calculates averages for each metric within each group
6. Writes averaged results to output files

config

Loaded configuration from JSON file.

Type
dict

safety_enabled

Whether memory-safe resource management is enabled.

Type
bool

num_processes

Number of worker processes for parallel operations.

Type
int

resource_manager

Dynamic resource manager (if available).

Type
ResourceManager

Example

```
>>> averager = ReportAverager('config/averager_config.json')
>>> averager.run()
Processing 150 files with 8 workers...
Created 12 averaged reports
```

__init__(*config_path*, *safety_enabled=True*)

Initialize the ReportAverager with configuration.

Parameters

- **config_path** (*str*) – Path to the JSON configuration file.
- **safety_enabled** (*bool, optional*) – Enable memory-safe resource management. When True, worker count is dynamically adjusted based on available RAM. Defaults to True.

Raises
SystemExit – If configuration file is not found or contains invalid JSON.

load_config(*config_path*)
Load and parse configuration file

validate_config()
Validate required configuration fields

parse_filename(*filename*)
Extract components from filename based on pattern

read_report_file(*filepath*)
Read and parse a report file (legacy, kept for compatibility)

group_files(*files*, *group_by*)
Group files according to specified grouping fields

average_group(*file_list*)
Average data from multiple files using multiprocessing

generate_output_filename(*group_key*, *components*)
Generate output filename from template

save_averaged_data(*data*, *output_path*)
Save averaged data to file

run()
Main execution method with multiprocessing optimization

core.averager.**main**()

2.2 analysis

Data analysis and visualization generation.

2.2.1 Analysis and Visualization Module

Smart adaptive analysis tool that automatically detects data patterns and generates comprehensive visualizations from simulation results.

This module provides a complete pipeline for analyzing report data:

1. **File Parsing** - Intelligent parsing of both averaged and raw report files
2. **Data Organization** - Grouping and merging data by configurable parameters
3. **Plot Strategy** - Automatic determination of appropriate visualizations
4. **Parallel Rendering** - Multiprocessing-based plot generation

Classes:
SmartFileParser: Parses report files and extracts structured data. DataOrganizer: Organizes data for different visualization purposes. PlotStrategy: Determines optimal plots based on data characteristics. PlotGenerator: Creates matplotlib/seaborn visualizations.

Supported Plot Types:

- Line plots (metrics vs parameters, grouped by router)
- 3D surface plots (two parameters vs metric)

- Violin plots (distribution visualization)
- Correlation heatmaps
- Pairplots (multi-metric relationships)

Example

Run analysis from command line:

```
python core/analysis.py
```

Or use programmatically:

```
from core.analysis import load_config, SmartFileParser, DataOrganizer

config = load_config()
parser = SmartFileParser(config)
organizer = DataOrganizer(parser)
averaged_data = organizer.load_averaged_files()
```

`core.analysis.SCRIPT_DIR`

Directory containing this script.

Type
Path

`core.analysis.PROJECT_ROOT`

Project root directory.

Type
Path

`core.analysis.CONFIG_DIR`

Configuration files directory.

Type
Path

`core.analysis.RESOURCE_MANAGER_AVAILABLE`

Whether resource manager is available.

Type
bool

Note

Uses multiprocessing with optimized pool initialization to avoid pickle overhead. Worker count is dynamically adjusted based on system resources.

`core.analysis.load_config`(*config_path=None*)

Load configuration from JSON file (cross-platform)

class core.analysis.**SmartFileParser**(*config*)

Bases: object

Intelligent file parser for averaged and raw simulation reports.

This class handles parsing of both averaged report files (generated by the averager module) and raw report files (directly from simulations). It extracts structured data including metrics, router information, and grouping parameters.

The parser automatically detects file types based on naming conventions and applies appropriate parsing strategies.

config

Configuration dictionary with parsing settings.

Type
dict

report_dir

Directory containing report files.

Type
str

separator

Field-value separator in report files (default ':').

Type
str

metrics

List of metric names to extract.

Type
list

ignore_fields

Set of field names to skip during parsing.

Type
set

Example

```
>>> parser = SmartFileParser(config)
>>> if parser.is_average_file('Router_10_ttl_average.txt'):
...     data = parser.parse_average_filename('Router_10_ttl_average.txt
↪')
...     metrics = parser.read_metrics(filepath)
```

__init__(*config*)

Initialize the parser with configuration.

Parameters
config (*dict*) – Configuration dictionary containing: - directories.report_dir: Path to reports directory - data_separator: Field-value

separator character - metrics.include: List of metrics to extract - metrics.ignore: List of fields to ignore

is_average_file(*filename*)

Check if file is an averaged report

get_report_type(*filename*)

Extract report type from filename

parse_average_filename(*filename*)

Parse averaged filename to extract: report_type, router, value, grouping_type Example: MessageStatsReport_EpidemicRouter_10_ttl_average.txt Returns: {report_type, router, value, grouping_type}

parse_raw_filename(*filename*)

Parse raw report filename Example: TEST_EpidemicRouter_12_300_5M_MessageStatsReport.txt

read_metrics(*filepath*)

Read metrics from a report file

class core.analysis.**DataOrganizer**(*parser*)

Bases: object

Organize data for visualization purposes.

This class loads and organizes parsed report data into DataFrames suitable for different visualization types. It handles both averaged and raw report files, merging data from multiple report types.

parser

Parser instance for reading files.

Type
SmartFileParser

report_dir

Directory containing report files.

Type
str

Example

```
>>> organizer = DataOrganizer(parser)
>>> averaged_dfs = organizer.load_averaged_files()
>>> raw_df = organizer.load_raw_files()
```

__init__(*parser*)

Initialize with a file parser.

Parameters
parser (SmartFileParser) – Configured parser instance.

load_averaged_files()

Load averaged files and organize by grouping type Merges data from multiple report types based on router, grouping_type, and value Returns: dict of {grouping_type: DataFrame}

load_raw_files()

Load raw (non-averaged) files Merges data from multiple report types based on common keys

class core.analysis.**PlotStrategy**(*config*)

Bases: object

Determine optimal plots based on data characteristics.

Analyzes available data and determines which visualizations are appropriate based on configurable thresholds (minimum values for line plots, surface plots, etc.).

config

Configuration dictionary.

Type
dict

thresholds

Plot thresholds from config.

Type
dict

__init__(*config*)

Initialize with configuration.

Parameters
config (*dict*) – Configuration with plot_thresholds key.

analyze_averaged_data(*averaged_dfs*)

Determine plot strategy for averaged data

class core.analysis.**PlotGenerator**(*config*, *output_dir*)

Bases: object

Generate all visualization types

__init__(*config*, *output_dir*)

get_axis_label(*grouping_type*)

Get proper axis label for a grouping type

create_line_plot(*job_data*)

Create line plot from averaged data

create_surface_plot(*job_data*)

Create 3D surface plot from two averaged datasets

create_violin_plot(*job_data*)

Create violin plot from averaged data

create_heatmap(*job_data*)

Create correlation heatmap from raw data

create_pairplot(*job_data*)

Create pairplot from raw data

core.analysis.**execute_plot_job**(*job_info*)

Dispatcher for multiprocessing - uses global config to avoid pickle overhead

core.analysis.**main**()

2.3 resource_manager

Dynamic memory and CPU resource management. Resource Manager - Dynamic Memory Management for OppNDA Implements memory-aware worker optimization as described in the research paper.

Mathematical Models Implemented: - Memory footprint: M(t) = M_base + sum(gamma * size(r_i) + M_overhead) - Optimal workers: P_opt = max{p in Z+ | M(t)|P=p <= eta * M_RAM}

class core.resource_manager.**ResourceConfig**

Bases: object

Configuration for resource management parameters.

ETA = 0.85

GAMMA = 2.5

M_OVERHEAD_MB = 30

MIN_WORKERS = 2

MAX_WORKERS = 64

FALLBACK_WORKERS = 8

SAFETY_ENABLED = True

class core.resource_manager.**MemoryEstimator**(*gamma=2.5*, *overhead_mb=30*)

Bases: object

Estimates memory consumption for batch processing.

Implements: M(t) = M_base + sum(gamma * size(r_i) + M_overhead)

Parameters

- **gamma** (*float*)
- **overhead_mb** (*float*)

__init__(*gamma=2.5*, *overhead_mb=30*)

Parameters

- **gamma** (*float*)
- **overhead_mb** (*float*)

estimate_file_memory(*file_size_bytes*)

Estimate memory needed to process a single file. Returns memory in bytes.

Parameters
file_size_bytes (*int*)

Return type
int

estimate_batch_memory(*file_sizes*, *num_workers*)

Estimate peak memory for a batch of files with given worker count.

Parameters

- **file_sizes** (*List[int]*) – List of file sizes in bytes
- **num_workers** (*int*) – Number of concurrent workers

Returns
Estimated peak memory in bytes

Return type
int

get_file_sizes(*file_paths*)

Get sizes of multiple files.

Parameters
file_paths (*List[str]*)

Return type
List[int]

class core.resource_manager.**DynamicSemaphore**(*initial_permits*, *eta=0.85*, *safety_enabled=True*)

Bases: object

A semaphore that dynamically adjusts based on available memory.

Implements: P_opt = max{p in Z+ | M(t)|P=p <= eta * M_RAM}

This prevents OS-level swap thrashing by capping concurrent workers when memory pressure is detected.

Parameters

- **initial_permits** (*int*)
- **eta** (*float*)
- **safety_enabled** (*bool*)

__init__(*initial_permits*, *eta=0.85*, *safety_enabled=True*)

Parameters

- **initial_permits** (*int*)
- **eta** (*float*)
- **safety_enabled** (*bool*)

`acquire(blocking=True)`

Acquire a permit, potentially waiting if none available.

Parameters
blocking (*bool*)

Return type
bool

`release()`

Release a permit.

`property current_permits: int`

`class core.resource_manager.ResourceManager(eta=0.85, gamma=2.5, overhead_mb=30, safety_enabled=True)`

Bases: `object`

Central resource manager for OppNDA's multiprocessing.

Provides: - Dynamic worker count calculation based on available RAM - Memory estimation for batch processing - Configurable safety measures (can be disabled)

Usage:
rm = ResourceManager() workers = rm.get_optimal_workers()

Or with safety disabled: rm = ResourceManager(safety_enabled=False)

Parameters

- **eta** (*float*)
- **gamma** (*float*)
- **overhead_mb** (*float*)
- **safety_enabled** (*bool*)

`__init__(eta=0.85, gamma=2.5, overhead_mb=30, safety_enabled=True)`

Initialize the resource manager.

Parameters

- **eta** (*float*) – RAM utilization threshold (0.0-1.0). Default 0.75
- **gamma** (*float*) – DataFrame expansion factor. Default 3.0
- **overhead_mb** (*float*) – Per-worker overhead in MB. Default 50
- **safety_enabled** (*bool*) – If False, disables memory checks and uses static workers

`get_optimal_workers(file_paths=None)`

Calculate optimal worker count based on available resources.

Implements: P_opt = max{p in Z+ | M(t)|P=p <= eta * M_RAM}

Parameters
file_paths (*List[str] | None*) – Optional list of file paths to process. If provided, uses actual file sizes for estimation.

Returns
Optimal number of worker processes

Return type
int

`create_semaphore`(*initial_permits=None*)

Create a dynamic semaphore for worker pool management.

Parameters
initial_permits (*int | None*) – Starting permit count. If None, uses optimal workers.

Returns
DynamicSemaphore instance

Return type
DynamicSemaphore

`get_memory_status`()

Get current memory status for monitoring/logging.

Returns
Dictionary with memory statistics

Return type
dict

`log_status`()

Print current memory status to console.

`core.resource_manager.get_optimal_workers`(*safety_enabled=True*, *file_paths=None*)

Quick function to get optimal worker count.

Parameters

- **safety_enabled** (*bool*) – If False, returns static fallback count
- **file_paths** (*List[str] | None*) – Optional file paths for size-based estimation

Returns
Optimal worker count

Return type
int

2.4 regression

Machine learning regression models.

`core.regression.load_config`(*path=None*)

Load configuration (cross-platform)

`class core.regression.DataProcessor`(*config*)

Bases: `object`

`__init__`(*config*)

get_files()

Find CSV files based on config mode

load_and_clean(*filepath*, *target=None*)

Load data and return X, y (Unscaled - Scaling happens in Pipeline)

Parameters

- **filepath** – Path to CSV file
- **target** – Target variable to predict (overrides config if provided)

core.regression.**create_pipeline**(*base_model*, *config*)

Builds a robust Sklearn Pipeline: Scaler -> Polynomials (Optional) -> Model

core.regression.**get_base_models**(*config*)

Instantiate base models (without pipeline wrappers)

core.regression.**plot_results**(*y_test*, *y_pred*, *model_name*, *router*, *out_dir*, *config*)

Standardized plotting function with style injection

core.regression.**plot_importance**(*pipeline*, *feature_names_in*, *model_name*, *router*, *out_dir*, *config*)

Extract feature names from pipeline (handling polynomials) and plot

2.5 path_utils

Cross-platform path utilities. Path utilities for cross-platform path handling and validation.

Provides functions for: - Resolving absolute paths from relative or user-specified paths - Validating path existence and permissions - Path normalization across Windows, Linux, macOS - Safe path construction

core.path_utils.**resolve_absolute_path**(*path*)

Resolve a path to its absolute form.

Handles: - Relative paths (converted to absolute based on current working directory) - Tilde expansion (~/) - Path separator normalization

Parameters

path (*str*) – Path string (can be relative or absolute)

Returns

Absolute path as string

Raises

ValueError – If path is empty or contains invalid characters

Return type

str

core.path_utils.**validate_path**(*path*, *must_exist=False*, *must_be_dir=False*)

Validate if a path is accessible and meets requirements.

Parameters

- **path** (*str*) – Path to validate
- **must_exist** (*bool*) – If True, path must exist

- **must_be_dir** (*bool*) – If True, path must be a directory

Returns
bool, error_message: Optional[str])

Return type
Tuple of (is_valid

core.path_utils.**safe_path_join**(**parts*)

Safely join path components and return absolute path.

Parameters
***parts** (*str*) – Path components to join

Returns
Absolute joined path

Return type
str

core.path_utils.**normalize_path_separators**(*path*)

Normalize path separators to forward slashes for consistency. Useful for storing paths in configs.

Parameters
path (*str*) – Path string

Returns
Path with normalized separators

Return type
str

core.path_utils.**get_relative_path**(*full_path*, *base_path*)

Get relative path from base_path to full_path.

Parameters

- **full_path** (*str*) – The target path
- **base_path** (*str*) – The base path to be relative to

Returns
Relative path string

Return type
str

core.path_utils.**is_path_within**(*target_path*, *base_path*)

Check if target_path is within base_path (for security validation).

Parameters

- **target_path** (*str*) – Path to check
- **base_path** (*str*) – Base/parent path

Returns
True if target is within base

Return type
bool

CHAPTER

THREE

REST API

This section documents the Flask REST API endpoints.

3.1 api

Main API endpoints for configuration and execution.

3.1.1 REST API Module for OppNDA

This module provides RESTful API endpoints for the OppNDA GUI application. It handles configuration management, settings persistence, and execution of post-processing scripts.

Endpoints Summary:

Configuration:

- GET/POST `/api/config/<name>` - Read/write config files
- POST `/api/save-all` - Save all configurations
- POST `/api/save-settings` - Save ONE simulator settings

Execution:

- POST `/api/run-one` - Run complete simulation pipeline
- POST `/api/run-averager` - Run report averaging
- POST `/api/run-analysis` - Run visualization generation
- POST `/api/run-regression` - Run ML regression

Streaming (SSE):

- GET `/api/stream-averager` - Stream averager output
- GET `/api/stream-analysis` - Stream analysis output
- GET `/api/stream-regression` - Stream regression output

Example

Starting the API server:

```
from flask import Flask
from app.api import api_bp

app = Flask(__name__)
```

```
app.register_blueprint(api_bp, url_prefix='/api')
app.run(port=5000)
```

app.api.**api_bp**

Flask Blueprint for API routes.

Type
Blueprint

app.api.**CONFIG_FILES**

Mapping of config names to filenames.

Type
dict

app.api.**DEFAULT_ONE_SETTINGS**

Default simulator settings.

Type
dict

Note

All endpoints return JSON responses with 'success' and optional 'message' keys. Streaming endpoints use Server-Sent Events (SSE) format.

app.api.**generate_default_settings**(*overrides=None*)

Generate a complete ONE simulator settings file content with sensible defaults.

Parameters
overrides (*Dict[str, Any] | None*) – Dict of values to override defaults (e.g., {'scenario_name': 'my_sim'})

Returns
Complete ONE settings file content ready to save

Return type
str

app.api.**get_config_path**(*config_name*)

Get the cross-platform path to a config file.

Parameters
config_name (*str*) – Name of the configuration

Returns
Path object pointing to the configuration file

Return type
Path

app.api.**get_default_settings**()

Return the default ONE simulator settings as JSON.

New users can use this to understand what defaults will be used.

Returns
JSON response with defaults and message

Return type
Tuple[*Dict*[str, *Any*], int]

app.api.**generate_default_settings_endpoint**()

Generate a default ONE settings file with optional overrides.

Request body:

{
"overrides": {"scenario_name": "my_sim", "router": "ProphetRouter"}, "save": true, "filename": "my_settings.txt"

}

Returns
Settings content and optionally saves to file

Return type
Tuple[*Dict*[str, *Any*], int]

app.api.**get_config**(*config_name*)

Read a configuration file and return its contents.

app.api.**update_config**(*config_name*)

Update a configuration file with new data.

Uses deep_merge to preserve fields not sent from the UI.

app.api.**update_config_field**(*config_name*)

Update a specific field in a configuration file.

app.api.**run_one_simulator**()

Complete simulation pipeline: Save config -> Run ONE -> Post-processing.

This endpoint handles the entire workflow: 1. Saves simulation settings file (.txt) 2. Saves post-processing configs (JSON) 3. Detects OS and builds appropriate command 4. Runs ONE simulator with correct settings file and batch count 5. Auto-triggers post-processing when simulation completes

app.api.**run_analysis**()

Execute analysis scripts (cross-platform compatible).

app.api.**process_data**()

Execute post-processing pipeline: averager -> analysis -> regression (optional).

app.api.**save_settings**()

Save simulation settings to a .txt file in the project directory.

app.api.**save_all_settings**()

Save both simulation settings (.txt) and all post-processing configs (JSON).

IMPORTANT: This function MERGES incoming config with existing config, preserving any fields not exposed in the UI (like plot_settings).

app.api.**run_averager_only**()

Run only the report averager script.

app.api.**run_analysis_only**()

Run only the analysis/visualization script.

app.api.**run_regression_only**()

Run only the ML regression script.

app.api.**stream_subprocess**(*command*, *cwd*)

Generator that yields SSE events from subprocess output line by line.

app.api.**stream_averager**()

Stream averager output in real-time using SSE.

app.api.**stream_analysis**()

Stream analysis output in real-time using SSE.

app.api.**stream_regression**()

Stream regression output in real-time using SSE.

app.api.**deep_merge**(*base*, *updates*)

Deep merge updates into base dict, preserving non-updated nested fields.

This ensures fields not exposed in the UI (like plot_settings) are preserved when saving config changes from the GUI.

Parameters

- **base** (*Dict[str, Any]*) – Base configuration dictionary
- **updates** (*Dict[str, Any]*) – Updates to apply to base

Returns

Merged configuration dictionary

Return type

Dict[str, *Any*]

app.api.**browse_directory**()

Browse and list directories for modern UI path selection.

Request JSON:

- path: (optional) Directory to browse. If empty, returns home/project dirs
- filter_type: (optional) 'dirs', 'files', or 'all'

Returns

{

'success': bool, 'current_path': str (absolute path), 'directories': [{'name': str, 'path': str}, ...], 'files': [{'name': str, 'path': str, 'size': int}, ...], 'parent_path': str or null

}

app.api.resolve_path()

Resolve a relative or absolute path to its absolute form.

Request JSON:

- path: (required) Path to resolve (can be relative or absolute)

Returns

{
'success': bool, 'absolute_path': str, 'exists': bool, 'is_dir': bool, 'is_file': bool
}

app.api.auto_save_config()

Auto-save configuration changes (silent endpoint for auto-save manager).

This endpoint handles incremental config saves from the UI without user notifications. It merges changes with existing config to preserve non-UI fields.

Request JSON:

- config: Config name ('analysis', 'averager', or 'regression')
- changes: Dict of field name -> value pairs

Returns
bool, 'message': str }

Return type
{ 'success'

3.2 routes

Web routes for serving the GUI. Web Routes for OppNDA Handles page rendering and static file serving.

app.routes.index()

Serve the main settings page.

Returns
Rendered settings.html template

Return type
str

app.routes.nda()

Serve the analysis results page with plot gallery.

Returns
Rendered nda.html template with plot files

Return type
Tuple[str, int]

app.routes.**serve_plot**(*filename*)

Serve plot images from the plots directory.

app.routes.**serve_gui_static**(*filename*)

Serve static GUI files.

app.routes.**run_one_pipeline**()

Complete simulation pipeline: Save config -> Run ONE -> Post-processing.

This handles both /run-one (legacy) and works for the complete pipeline.

3.3 API Endpoints Summary

3.3.1 Configuration Endpoints

Endpoint	Method	Description
/api/config/<name>	GET	Get configuration file (analysis, averager, regression)
/api/config/<name>	POST	Save configuration file
/api/save-all	POST	Save all configurations and settings

3.3.2 Execution Endpoints

Endpoint	Method	Description
/api/run-one	POST	Run complete simulation pipeline
/api/run-averager	POST	Run report averaging only
/api/run-analysis	POST	Run analysis/visualization only
/api/run-regression	POST	Run ML regression only

3.3.3 Streaming Endpoints

Endpoint	Method	Description
/api/stream-averager	GET	Stream averager output (SSE)
/api/stream-analysis	GET	Stream analysis output (SSE)
/api/stream-regressi	GET	Stream regression output (SSE)